Power ISA™
Version 2.04

April 3, 2007

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

Preface

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

In 2006, Freescale and IBM collaborated on the creation of the Power ISA Version 2.03, which represented the reunification of the architecture by combining Book E content with the more general purpose PowerPC Version 2.02. A significant benefit of the reunification is the establishment of a single, compatible, 64-bit programming model. The combining also extends explicit architectural endorsement and control to Auxiliary Processing Units (APUs), units of function that were originally developed as implementation- or product family-specific extensions in the context of the Book E allocated opcode space. With the resulting architectural superset comes a framework that clearly establishes requirements and identifies options.

To a very large extent, application program compatibility has been maintained throughout the history of the architecture, with the main exception being application exploitation of APUs. The framework identifies the base, pervasive, part of the architecture, and differentiates it from "categories" of optional function (see Section 1.3.5 of Book I). Because of the substantial differences in the supervisor (privileged) architecture that developed as Book E was optimized for embedded systems, the supervisor architectures for embedded and general purpose implementations are represented as mutually exclusive categories. Future versions of the architecture will seek to converge on a common solution where possible.

This document defines the Power ISA Version 2.04. It is comprised of five 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. It includes five new chapters derived from APU function, including the vector extension also known as Altivec.

Book II, *Power ISA Virtual Environment Architecture,* defines the storage model and related instructions and facilities available to the application programmer.

Book III-S, *Power ISA Operating Environment Architecture,* defines the supervisor instructions and related facilities used for general purpose implementations. It consists mainly of the contents of Book III from PowerPC Version 2.02, with the addition of significant new large page and big segment support.

Book III-E, *Power ISA Operating Environment Architecture,* defines the supervisor instructions and related facilities used for embedded implementations. It was derived from Book E and extended to include APU function.

Book VLE, *Power ISAVariable Length Encoded Instructions Architecture,* defines alternative instruction encodings and definitions intended to increase instruction density for very low end implementations. It was derived from an APU description developed by Freescale Semiconductor.

As used in this document, the term "Power ISA" refers to the instructions and facilities described in Books I, II, III-S, III-E, and VLE.

Usage of the phrase "Book III" refers to both Book III-S and Book III-E. An exception to this rule is when, at the beginning of a Section or Book, it is specified that usage of the phrase "Book III" implies only either "Book III-S" or "Book III-E".

Change bars have been included to indicate changes from Version 2.03.

Summary of Changes in Version 2.04

Version 2.04 of this document differs from the previous version primarily by containing the definitions of the following facilities:

New Server Page Protection States. An additional state of the page protection bits in the page table entry is defined which can be used to provide privileged programs read only access and problem state programs no access to a virtual page.

Server Virtualized Partition Memory. Several new features are added to enable virtualization of a partition's memory in order to support more partitions and additional concurrent maintenance procedures transparently to operating system code.

Server Virtual Page Class Key Protection. A KEY field in the page table entry and associated features are added for the Server environment to facilitate quick modification of access permission for multiple pages at once.

Server Time Base Facility - TBU40. Support is added for time base synchronization via this new time base facility, in which only the upper 40 bits of the time base are accessed.

Version Verification

See the Power ISA representative for your company.

Table of Contents

Chapter 4. Floating-Point Processor [Category: Floating-Point] 93

Chapter 6. Reset and Initialization . . 595

Chapter 7. Timer Facilities 597

Chapter 8. Debug Facilities 605

Figures

Book III-S:

Power ISA Operating Environment Architecture - Server Environment 391

Book III-E:

Power ISA Operating Environment Architecture - Embedded Environment..... 507

Book VLE:

Appendices:

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 in Appendix D of Book I.

1.3 Document Conventions

1.3.1 Definitions

The following definitions are used throughout this document.

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

- **application program**
  A program that uses only the instructions and resources described in Books I and II.
- **quadwords, doublewords, words, halfwords, and bytes**
  128 bits, 64 bits, 32 bits, 16 bits, and 8 bits, respectively.
- **positive**
  Means greater than zero.
- **negative**
  Means less than zero.
- **floating-point single format** (or simply **single format**)
  Refers to the representation of a single-precision binary floating-point value in a register or storage.
- **floating-point double format** (or simply **double format**)
  Refers to the representation of a double-precision binary floating-point value in a register or storage.
- **system library program**
  A component of the system software that can be called by an application program using a *Branch* instruction.
- **system service program**
  A component of the system software that can be called by an application program using a *System Call* 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 25.
- **Auxiliary Processor**
  An implementation-specific processing unit. Previous versions of the architecture use the term Auxiliary Processing Unit (APU) to describe this extension of the architecture. Architectural support for auxiliary processors is part of the Embedded category.

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.
- 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.
- 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 category, bits in registers that are less than 64 bits start with bit number 64-L, where L is the register length; for the Vector category, bits in registers that are less than 128 bits start with bit number 128-L.
  - The leftmost bit of a sequence of bits is the most significant bit of the sequence.
  - X_p means bit p of register/instruction/field/bit_string X.
  - $X_{p:q}$ means bits p through q of register/instruction/field/bit_string X.
  - $X_{p\ q\ ...}$ means bits p, q, ... of register/instruction/field/bit_string X.
- ¬(RA) means the one's complement of the contents of register RA.
- A period (.) as the last character of an instruction mnemonic means that the instruction records status information in certain fields of the Condition Register as a side effect of execution.
- The symbol || is used to describe the concatenation of two values. For example, 010 || 111 is the same as 010111.
- x^n means x raised to the n^{th} power.
- ^nx means the replication of x, n times (i.e., x concatenated to itself n-1 times). (n)0 and (n)1 are special cases:
  - $^n0$ means a field of n bits with each bit equal to 0. Thus ^{50} is equivalent to 0b00000.
  - $^n1$ means a field of n bits with each bit equal to 1. Thus ^{51} 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 and Reserved Values

Reserved fields in instructions are ignored by the processor. This is a requirement in the Server environment and is being phased into the Embedded environment.

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 19. The only exception to the preceding rule is that it does not apply 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) is implementation-dependent. Unless otherwise stated, software is permitted to write any value to such a bit. 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.

Assembler Note

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

Programming Note

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

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

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

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

1.3.4 Description of Instruction Operation

Instruction descriptions (including related material such as the introduction to the section describing the instructions) mention that the instruction may cause a system error handler to be invoked, under certain conditions, if and only if the system error handler may treat the case as a programming error. (An instruction may cause a system error handler to be invoked under other conditions as well; see Chapter 6 of Book III-S and Chapter 5 of Book III-E).

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.
¬	NOT logical operator
+	Two’s complement addition
-	Two’s complement subtraction, unary minus
×	Multiplication
$\times_{si}$	Signed-integer multiplication
$\times_{ui}$	Unsigned-integer multiplication
/	Division
÷	Division, with result truncated to integer
√	Square root
=, ≠	Equals, Not Equals relations
<, ≤, >, ≥	Signed comparison relations
$<^u$, $>^u$	Unsigned comparison relations
?	Unordered comparison relation
&, $\vert$	AND, OR logical operators
⊕, ≡	Exclusive OR, Equivalence logical operators ((a≡b) = (a⊕¬b))
ABS(x)	Absolute value of x
CEIL(x)	Least integer ≥ x
DCR(x)	Device Control Register x
DOUBLE(x)	Result of converting x from floating-point single format to floating-point double format, using the model shown on page 111
EXTS(x)	Result of extending x on the left with sign bits
FLOOR(x)	Greatest integer ≤ x
GPR(x)	General Purpose Register x
MASK(x, y)	Mask having 1s in positions x through y (wrapping if x > y) and 0s elsewhere
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.
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$\Vert$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 114
SPR(x)	Special Purpose Register x
TRAP	Invoke the system trap handler
characterization	Reference to the setting of status bits, in a standard way that is explained in the text
undefined	An undefined value.
CIA	Current Instruction Address, which is the 64-bit address of the instruction being described by a sequence of RTL. Used by relative branches to set the Next Instruction Address (NIA), and by Branch instructions with LK=1 to set the Link Register. Does not correspond to any architected register.
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 (VLE behavior is different; see Book VLE). 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.

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

Table 1: Operator precedence

Operators	**Associativity**
subscript, function evaluation	left to right
pre-superscript (replication), post-superscript (exponentiation)	*right to left*
unary -, ¬	*right to left*
×, ÷	left to right
+, -,	left to right
||	left to right
=, ≠, <, ≤, >, ≥, $<^u$, $>^u$, ?	left to right
&, ⊕, ≡	left to right
|	left to right
: (range)	none
←, $\leftarrow_{iea}$	none

1.3.5 Categories

Each facility (including registers and fields therein) and instruction is in exactly one of the categories listed in Figure 1.

A category may be defined as a *dependent category*. These are categories that are supported only if the category they are dependent on is also supported. Dependent categories are identified by the "." in their category name, e.g., if an implementation supports the Floating-Point.Record category, then the Floating-Point category is also supported.

An implementation that supports a facility or instruction in a given category, except for the two categories described in Section 1.3.5.1, supports all facilities and instructions in that category.

Category	**Abvr.**	**Notes**
Base	B	Required for all implementations
Server	S	Required for Server implementations
Embedded	E	Required for Embedded implementations
Alternate Time Base	ATB	An additional Time Base; see Book II
Cache Specification	CS	Specify a specific cache for some instructions; see Book II
Embedded.Cache Debug	E.CD	Provides direct access to cache data and directory content
Embedded.Cache Initialization	E.CI	Instructions that invalidate the entire cache
Embedded.Enhanced Debug	E.ED	Embedded Enhanced Debug facility; see Book III-E
Embedded.External PID	E.PD	Embedded External PID facility; see Book III-E
Embedded.Little-Endian	E.LE	Embedded Little-Endian page attribute; see Book III-E
Embedded.MMU Type FSL	E.MF	Embedded MMU example Type FSL; see Book III-E
Embedded.Performance Monitor	E.PM	Embedded performance monitor example; see Book III-E
Embedded.Processor Control	E.PC	Processor control facility; see Book III-E
Embedded Cache Locking	ECL	Embedded Cache Locking facility; see Book III-E
External Control	EC	External Control facility; see Book II
External Proxy	EXP	External Proxy facility; see Book III-E
Floating-Point Floating-Point.Record	FP FP.R	Floating-Point Facilities Floating-Point instructions with Rc=1
Legacy Move Assist	LMV	*Determine Left most Zero Byte* instruction
Legacy Integer Multiply-Accumulate[1]	LMA	*Legacy Integer Multiply-accumulate* instructions
Load/Store Quadword	LSQ	Load/Store Quadword instructions; see Book III-S
Memory Coherence	MMC	Requirement for Memory Coherence; see Book II
Move Assist	MA	*Move Assist* instructions
Server.Performance Monitor	S.PM	Performance monitor example for Servers; see Book III-S
Signal Processing Engine[1, 2] SPE.Embedded Float Scalar Double SPE.Embedded Float Scalar Single SPE.Embedded Float Vector	SP SP.FD SP.FS SP.FV	Facility for signal processing GPR-based Floating-Point double-precision instruction set GPR-based Floating-Point single-precision instruction set GPR-based Floating-Point Vector instruction set
Stream	STM	Stream variant of ***dcbt*** instruction; see Book II
Trace	TRC	Trace Facility; see Book III-S
Variable Length Encoding	VLE	Variable Length Encoding facility; see Book VLE
Vector[1] Vector.Little-Endian	V V.LE	Vector facilities Little-Endian support for Vector storage operations.
Wait	WT	***wait*** instruction; see Book II
64-Bit	64	Required for 64-bit implementations; not defined for 32-bit impl's

1 Because of overlapping opcode usage, SPE is mutually exclusive with Vector and with Legacy Integer Multiply-Accumulate, and Legacy Integer Multiply-Accumulate is mutually exclusive with Vector.
2 The SPE-dependent Floating-Point categories are collectively referred to as SPE.Embedded Float_* or SP.*.

Figure 1. Category Listing

An instruction in a category that is not supported by the implementation is treated as an illegal instruction or an unimplemented instruction on that implementation (see Section 1.7.2).

For an instruction that is supported by the implementation with field values that are defined by the architecture, the field values defined as part of a category that is not supported by the implementation are treated as reserved values on that implementation (see Section 1.3.3 and Section 1.8.2).

Bits in a register that are in a category that is not supported by the implementation are treated as reserved.

1.3.5.1 Phased-In/Phased-Out

There are two special dependent categories, *Phased-In* and *Phased-Out*, defined below. These categories have the exception that an implementation may support a subset of the instructions or facilities defined as being part of the category.

Phased-In These are facilities and instructions that, in some future version of the architecture, will be required as part of the category they are dependent on.

Phased-Out 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.

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.

1.3.5.2 Corequisite Category

A corequisite category is an additional category that is associated with an instruction or facility, and must be implemented if the instruction or facility is implemented.

1.3.5.3 Category Notation

Instructions and facilities are considered part of the Base category unless otherwise marked. If a section is marked with a specific category tag, all material in that section and its subsections are considered part of the category, unless otherwise marked. Overview sections may contain discussion of instructions and facilities from various categories without being explicitly marked.

An example of a category tag is: [Category: Server].

An example of a dependent category is:
[Category: Server.Phased-In]

The shorthand <E> and <S> may also be used for Category: Embedded and Server respectively.

1.3.6 Environments

All implementations support one of the two defined environments, Server or Embedded. Environments refer to common subsets of instructions that are shared across many implementations. The Server environment describes implementations that support Category: Base and Server. The Embedded environment describes implementations that support Category: Base and Embedded.

1.4 Processor Overview

The processor implements the instruction set, the storage model, and other facilities defined in this document. There are four basic classes of instructions:

- branch instructions (Chapter 2)
- fixed-point instructions (Chapter 3), and other instructions that use the fixed-point registers (Chapters 6, 7, 8, and 9)
- floating-point instructions (Chapter 4)
- vector instructions (Chapter 5)

Fixed-point instructions operate on byte, halfword, word, and doubleword operands. Floating-point instructions operate on single-precision and double-precision floating-point operands. Vector instructions operate on vectors of scalar quantities and on scalar quantities where the scalar size is byte, halfword, word, and quadword. The Power ISA uses instructions that are four bytes long and word-aligned (VLE has different instruction characteristics; see Book VLE). It provides for byte, halfword, word, and doubleword operand fetches and stores between storage and a set of 32 General Purpose Registers (GPRs). It provides for word and doubleword operand fetches and stores between storage and a set of 32 Floating-Point Registers (FPRs). It also provides for byte, halfword, word, and quadword operand fetches and stores between storage and a set of 32 Vector Registers (VRs).

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 2 is a logical representation of instruction processing. Figure 3 shows the registers of the Power ISA User Instruction Set Architecture.

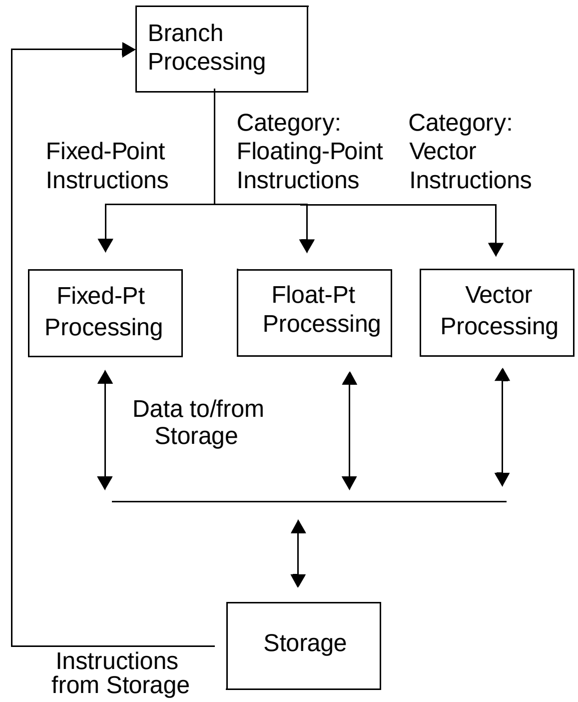


Figure 2. Logical processing model

CR	
32	63

"Condition Register" on page 26

LR	
0	63

"Link Register" on page 27

CTR	
0	63

"Count Register" on page 27

GPR 0	
GPR 1	
. . .	
. . .	
GPR 30	
GPR 31	
0	63

"General Purpose Registers" on page 38

XER	
0	63

"Fixed-Point Exception Register" on page 38

Category: Embedded:

SPRG4	
SPRG5	
SPRG6	
SPRG7	
0	63

"Software-use SPRs" on page 39.

Category: Embedded and Vector

VRSAVE	
32	63

"VR Save Register" on page 136

Category: Floating-Point:

FPR 0	
FPR 1	
. . .	
. . .	
FPR 30	
FPR 31	
0	63

"Floating-Point Registers" on page 95

FPSCR	
32	63

"Floating-Point Status and Control Register" on page 95

Category: Vector:

VR 0	
VR 1	
...	
...	
VR 30	
VR 31	
0	127

"Vector Registers" on page 135

VSCR	
96	127

"Vector Status and Control Register" on page 135

Category: SPE:

Accumulator	
0	63

"Accumulator" on page 202

SPEFSCR	
32	63

"Signal Processing and Embedded Floating-Point Status and Control Register" on page 202

Figure 3. Power ISA user register set

1.5 Computation modes

1.5.1 Modes [Category: Server]

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 status 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-S). 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.

1.5.2 Modes [Category: Embedded]

Processors may provide 32-bit mode, or both 64-bit mode and 32-bit mode. The modes differ in the following ways.

- In 64-bit mode, the processor behaves as described for 64-bit mode in the Server environment; see Section 1.5.1.
- In 32-bit mode, instructions other than SP, SP.Embedded Float Scalar Double, and SP.Embedded Float Vector use only the lower 32 bits of a GPR and produce a 32-bit result. Results written to the GPRs write only the lower 32-bits and the upper 32 bits are undefined except for SP.Embedded Float Scalar Single instructions which leave the upper 32-bits unchanged. SP, SP.Embedded Float Scalar Double, and SP.Embedded Float Vector instructions use all 64 bits of a GPR and produce a 64-bit result regardless of the mode.

  Instructions that set condition bits do so based on the 32-bit result computed. Effective addresses and all SPRs operate on the lower 32 bits only unless otherwise stated. The instructions in the 64-Bit category are not necessarily available; if they are not available, attempting to execute such an instruction causes the system illegal instruction error handler to be invoked.

Floating-Point and Vector instructions operate on FPRs and VPRs, respectively, independent of modes.

1.6 Instruction formats

All instructions are four bytes long and word-aligned (except for VLE instructions; see Book VLE). Thus, whenever instruction addresses are presented to the processor (as in *Branch* instructions) the low-order two bits are ignored. Similarly, whenever the processor develops an instruction address the low-order two bits are zero.

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

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

Split Field Notation

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

1.6.1 I-FORM

0	6	30	31
OPCD	LI	AA	LK

Figure 4. I instruction format

1.6.2 B-FORM

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

Figure 5. B instruction format

1.6.3 SC-FORM

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

Figure 6. SC instruction format

1.6.4 D-FORM

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

Figure 7. D instruction format

1.6.5 DS-FORM

0	6	11	16	30 31
OPCD	RT	RA	DS	XO
OPCD	RS	RA	DS	XO

Figure 8. DS instruction format

1.6.6 DQ-FORM

0	6	11	16	28 31
OPCD	RT	RA	DQ	PT

Figure 9. DQ instruction format

1.6.7 X-FORM

0	6	11	16	21	31
OPCD	RT	RA	///	XO	/
OPCD	RT	RA	RB	XO	/
OPCD	RT	RA	NB	XO	/
OPCD	RT	/ SR	///	XO	/
OPCD	RT	///	RB	XO	/
OPCD	RT	///	///	XO	/
OPCD	RS	RA	RB	XO	Rc
OPCD	RS	RA	RB	XO	1
OPCD	RS	RA	RB	XO	/
OPCD	RS	RA	NB	XO	/
OPCD	RS	RA	SH	XO	Rc
OPCD	RS	RA	///	XO	Rc
OPCD	RS	RA	///	XO	/
OPCD	RS	/ SR	///	XO	/
OPCD	RS	///	RB	XO	/
OPCD	RS	///	///	XO	/
OPCD	RS	/// L	///	XO	/
OPCD	BF / L	RA	RB	XO	/
OPCD	BF //	FRA	FRB	XO	/
OPCD	BF //	BFA //	///	XO	/
OPCD	BF //	///	U /	XO	Rc
OPCD	BF //	///	///	XO	/
OPCD	/ TH	RA	RB	XO	/
OPCD	/ CT	///	///	XO	/
OPCD	/ CT	RA	RB	XO	/
OPCD	/// L	RA	RB	XO	/
OPCD	/// L	///	RB	XO	/
OPCD	/// L	///	///	XO	/
OPCD	TO	RA	RB	XO	/
OPCD	FRT	RA	RB	XO	/
OPCD	FRT	///	FRB	XO	Rc
OPCD	FRT	///	///	XO	Rc
OPCD	FRS	RA	RB	XO	/
OPCD	BT	///	///	XO	Rc
OPCD	///	RA	RB	XO	/
OPCD	///	///	RB	XO	/
OPCD	///	///	///	XO	/
OPCD	///	///	E ///	XO	/
OPCD	???	RA	RB	XO	?
OPCD	???	???	???	XO	/
OPCD	VRT	RA	RB	XO	/
OPCD	VRS	RA	RB	XO	/
OPCD	MO	///	///	XO	/

Figure 10. X instruction format

1.6.8 XL-FORM

0	6		11		16		21	31
OPCD	BT		BA		BB		XO	/
OPCD	BO		BI		///	BH	XO	LK
OPCD	BF	//	BFA	//	///		XO	/
OPCD	///		///		///		XO	/

Figure 11. XL instruction format

1.6.9 XFX-FORM

0	6	11			21	31
OPCD	RT	spr			XO	/
OPCD	RT	tbr			XO	/
OPCD	RT	0	///		XO	/
OPCD	RT	1	FXM	/	XO	/
OPCD	RT	dcr			XO	/
OPCD	RT	pmrn			XO	/
OPCD	DUI	DUIS			XO	/
OPCD	RS	0	FXM	/	XO	/
OPCD	RS	1	FXM	/	XO	/
OPCD	RS	spr			XO	/
OPCD	RS	dcr			XO	/
OPCD	RS	pmrn			XO	/

Figure 12. XFX instruction format

1.6.10 XFL-FORM

0	6	7	15	16	21	31
OPCD	/	FLM	/	FRB	XO	Rc

Figure 13. XFL instruction format

1.6.11 XS-FORM

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

Figure 14. XS instruction format

1.6.12 XO-FORM

0	6	11	16	21	22	31
OPCD	RT	RA	RB	OE	XO	Rc
OPCD	RT	RA	RB	/	XO	Rc
OPCD	RT	RA	///	OE	XO	Rc

Figure 15. XO instruction format

1.6.13 A-FORM

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

Figure 16. A instruction format

1.6.14 M-FORM

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

Figure 17. M instruction format

1.6.15 MD-FORM

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

Figure 18. MD instruction format

1.6.16 MDS-FORM

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

Figure 19. MDS instruction format

1.6.17 VA-FORM

0	6	11	16	21		26	31
OPCD	VRT	VRA	VRB	VRC		XO	
OPCD	VRT	VRA	VRB	/	SHB	XO	

Figure 20. VA instruction format

1.6.18 VC-FORM

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

Figure 21. VC instruction format

1.6.19 VX-FORM

0	6	11	16	21 (31)
OPCD	VRT	VRA	VRB	XO
OPCD	VRT	///	VRB	XO
OPCD	VRT	UIM	VRB	XO
OPCD	VRT	/ UIM	VRB	XO
OPCD	VRT	// UIM	VRB	XO
OPCD	VRT	/// UIM	VRB	XO
OPCD	VRT	SIM	///	XO
OPCD	VRT	///		XO
OPCD	///		VRB	XO

Figure 22. VX instruction format

1.6.20 EVX-FORM

0	6	11	16	21 (31)
OPCD	RS	RA	RB	XO
OPCD	RS	RA	UI	XO
OPCD	RT	///	RB	XO
OPCD	RT	RA	RB	XO
OPCD	RT	RA	///	XO
OPCD	RT	UI	RB	XO
OPCD	BF //	RA	RB	XO
OPCD	RT	RA	UI	XO
OPCD	RT	SI	///	XO

Figure 23. EVX instruction format

1.6.21 EVS-FORM

0	6	11	16	21	29 (31)
OPCD	RT	RA	RB	XO	BFA

Figure 24. EVS instruction format

1.6.22 Instruction Fields

AA (30)

Absolute Address bit.

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.

BA (11:15)

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

BB (16:20)

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

BC (21:25)

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

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.

BF (6:8)

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

BFA (11:13 or 29:31)

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

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”.

BI (11:15)

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

BO (6:10)

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

BT (6:10)

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

CT (7:10)

Field used in X-form instructions to specify a cache target (see Section 3.2.2 of Book II).

D (16:31)

Immediate field used to specify a 16-bit signed two’s complement integer which is sign-extended to 64 bits.

DCR (11:20)

Field used by the *Move To/From Device Control Register* instructions (see Book III-E).

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.

DUI (6:10)
Field used by the ***dnh*** instruction (see Book II).

DUIS (11:20)
Field used by the ***dnh*** instruction (see Book II).

E (16)
Field used by the *Write MSR External Enable* instruction (see Book III-E).

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

FRA (11:15)
Field used to specify an FPR to be used as a source.

FRB (16:20)
Field used to specify an FPR to be used as a source.

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

FRS (6:10)
Field used to specify an FPR to be used as a source.

FRT (6:10)
Field used to specify an FPR to be used as a target.

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.

L (10 or 15)
Field used to specify whether a fixed-point Compare instruction is to compare 64-bit numbers or 32-bit numbers.

Field used by the *Data Cache Block Flush* instruction (see Section 3.2.2 of Book II).

Field used by the *Move To Machine State Register* and *TLB Invalidate Entry* instructions (see Book III).

L (9:10)
Field used by the *Synchronize* instruction (see Section 3.3.1 of Book II).

LEV (20:26)
Field used by the *System Call* instruction.

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.

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.

MB (21:25) and ME (26:30)
Fields used in M-form instructions to specify a 64-bit mask consisting of 1-bits from bit MB+32 through bit ME+32 inclusive and 0-bits elsewhere, as described in Section 3.3.13, "Fixed-Point Rotate and Shift Instructions" on page 77.

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 3.3.13, "Fixed-Point Rotate and Shift Instructions" on page 77.

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 3.3.13, "Fixed-Point Rotate and Shift Instructions" on page 77.

MO (6:10)
Field used in X-form instructions to specify a subset of storage accesses.

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

OPCD (0:5)
Primary opcode field.

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

PMRN (11:20)
Field used to specify a Performance Monitor Register for the ***mfpmr*** and ***mtpmr*** instructions.

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

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

Rc (21 OR 31)
RECORD bit.
- 0 Do not alter the Condition Register.

1 Set Condition Register Field 0, Field 1, or Field 6 as described in Section 2.3.1, “Condition Register” on page 26.

RS (6:10)
Field used to specify a GPR to be used as a source.

RT (6:10)
Field used to specify a GPR to be used as a target.

SH (16:20, or 16:20 and 30)
Field used to specify a shift amount.

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

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

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

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

SR (12:15)
Field used by the *Segment Register Manipulation* instructions (see Book III-S).

TBR (11:20)
Field used by the *Move From Time Base* instruction (see Section 4.2.1 of Book II).

TH (7:10)
Field used by the data stream variant of the ***dcbt*** and ***dcbtst*** instructions (see Section 3.2.2 of Book II).

TO (6:10)
Field used to specify the conditions on which to trap. The encoding is described in Section 3.3.10, “Fixed-Point Trap Instructions” on page 69.

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

UI (11:15, 16:20, or 16:31)
Immediate field used to specify an unsigned integer.

UIM (11:15, 12:15, 13:15, 14:15)
Immediate field used to specify an unsigned integer.

VRA (11:15)
Field used to specify a VR to be used as a source.

VRB (16:20)
Field used to specify a VR to be used as a source.

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

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

VRT (6:10)
Field used to specify a VR to be used as a target.

XO (21:28, 21:29, 21:30, 21:31, 22:30, 22:31, 26:30, 26:31, 27:29, 27:30, or 30:31)
Extended opcode field.

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 of 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”. Instructions that are part of a category that is not supported are treated as illegal instructions.

1.7.2 Illegal Instruction Class

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

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

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

1.7.3 Reserved Instruction Class

This class of instructions contains the set of instructions described in Appendix E 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/Store Multiple* instructions
- the *Load/Store String* instructions
- the *Or Immediate* instruction (preferred form of no-op)
- the *Move To Condition Register Fields* instruction

1.8.2 Invalid Instruction Forms

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

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

Some instruction forms are invalid because the instruction contains a reserved value in a defined field (see Section 1.3.3 on page 5); these invalid forms are not discussed further. All other invalid forms are identified in the instruction descriptions.

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

Assembler Note

Assemblers should report uses of invalid instruction forms as errors.

1.9 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 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* instruction (system service program)
- the execution of a *Trap* instruction that traps (system trap handler)
- the execution of a floating-point instruction that causes a floating-point enabled exception to exist (system floating-point enabled exception error handler)
- the execution of an auxiliary processor instruction that causes an auxiliary processor enabled exception to exist (system auxiliary processor enabled exception error handler)

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

The invocation of the system error handler is precise, except that the invocation of the auxiliary processor enabled exception error handler may be imprecise, and if one of the imprecise modes for invoking the system floating-point enabled exception error handler is in effect (see page 103), 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. In the Embedded environment this ordering is a page attribute (see Book II) and is specified independently for each virtual page, while in the Server environment it is a mode (see Book III-S) and applies to all storage.

1.10.1 Storage Operands

Storage operands may be bytes, halfwords, words, doublewords, or quadwords (see book III), or, for the *Load/Store Multiple* and *Move Assist* instructions, a sequence of bytes or words. The address of a storage operand is the address of its first byte (i.e., of its lowest-numbered byte).

Operand length is implicit for each instruction.

The operand of a single-register *Storage Access* instruction, or of a quadword *Load* or *Store* instruction, has a “natural” alignment boundary equal to the operand length. In other words, the “natural” address of an operand is an integral multiple of the operand length. A storage operand is said to be *aligned* if it is aligned at its natural boundary; otherwise it is said to be *unaligned.* See the following table.

Operand	Length	$\text{Addr}_{60:63}$ if aligned
Byte	8 bits	xxxx
Halfword	2 bytes	xxx0
Word	4 bytes	xx00
Doubleword	8 bytes	x000
Quadword	16 bytes	0000
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. For example, a 12-byte datum in storage is said to be word-aligned if its address is an integral multiple of 4.

Some instructions require their storage operands to have certain alignments. In addition, alignment may affect performance. For single-register *Storage Access* instructions, and for quadword *Load* and *Store* instructions, the best performance is obtained when storage operands are aligned. Additional effects of data placement on performance are described in Chapter 2 of Book II.

When a storage operand of length N bytes starting at effective address EA is copied between storage and a register that is R bytes long (i.e., the register contains bytes numbered from 0, most significant, through R-1, least significant), the bytes of the operand are placed into the register or into storage in a manner that depends on the byte ordering for the storage access as shown in Figure 25, unless otherwise specified in the instruction description.

Big-Endian Byte Ordering	
Load	Store
for i=0 to N-1: $RT_{(R-N)+i} \leftarrow$ MEM(EA+i,1)	for i=0 to N-1: MEM(EA+i,1) $\leftarrow (RS)_{(R-N)+i}$
Little-Endian Byte Ordering	
Load	Store
for i=0 to N-1: $RT_{(R-1)-i} \leftarrow$ MEM(EA+i,1)	for i=0 to N-1: MEM(EA+i,1) $\leftarrow (RS)_{(R-1)-i}$
Notes: 1. In this table, subscripts refer to bytes in a register rather than to bits as defined in Section 1.3.2. 2. This table does not apply to the ***lvebx***, ***lvehx***, ***lvewx, stvebx***, ***stvehx***, and ***stvewx*** instructions.	

Figure 25. Storage operands and byte ordering

Figure 26 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 27 and 28 show each scalar aligned at its natural boundary. 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 27. 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 26, are shown in hex (as characters for the elements of the string).

The Little-Endian mapping of structure s is shown in Figure 28. 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 26. 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				

Figure 27. 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
				23	22	21	20	

Figure 28. Little-Endian mapping of structure 's'

1.10.2 Instruction Fetches

Instructions are always four bytes long and word-aligned (except for VLE instructions; see Book VLE).

When an instruction starting at effective address EA is fetched from storage, the relative order of the bytes within the instruction depend on the byte ordering for the storage access as shown in Figure 29.

Big-Endian Byte Ordering
for i=0 to 3: $inst_i \leftarrow$ MEM(EA+i,1)
Little-Endian Byte Ordering
for i=0 to 3: $inst_{3-i} \leftarrow$ MEM(EA+i,1)
Note: In this table, subscripts refer to bytes of the instruction rather than to bits as defined in Section 1.3.2.

Figure 29. Instructions and byte ordering

Figure 30 shows an example of a small assembly language program **p**.

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

done:
        stw         r7,total
```

Figure 30. Assembly language program ‘p’

The Big-Endian mapping of program **p** is shown in Figure 31 (assuming the program starts at address 0).

<table>
<tr><td rowspan="2">00</td><td colspan="4">loop: cmplwi r5,0</td><td colspan="4">beq done</td></tr>
<tr><td>00</td><td>01</td><td>02</td><td>03</td><td>04</td><td>05</td><td>06</td><td>07</td></tr>
<tr><td rowspan="2">08</td><td colspan="4">lwzux r4,r5,r6</td><td colspan="4">add r7,r7,r4</td></tr>
<tr><td>08</td><td>09</td><td>0A</td><td>0B</td><td>0C</td><td>0D</td><td>0E</td><td>0F</td></tr>
<tr><td rowspan="2">10</td><td colspan="4">subi r5,r5,4</td><td colspan="4">b loop</td></tr>
<tr><td>10</td><td>11</td><td>12</td><td>13</td><td>14</td><td>15</td><td>16</td><td>17</td></tr>
<tr><td rowspan="2">18</td><td colspan="4">done: stw r7,total</td><td colspan="4"></td></tr>
<tr><td>18</td><td>19</td><td>1A</td><td>1B</td><td>1C</td><td>1D</td><td>1E</td><td>1F</td></tr>
</table>

Figure 31. Big-Endian mapping of program ‘p’

The Little-Endian mapping of program **p** is shown in Figure 32.

<table>
<tr><td colspan="4">beq done</td><td colspan="4">loop: cmplwi r5,0</td><td rowspan="2">00</td></tr>
<tr><td>07</td><td>06</td><td>05</td><td>04</td><td>03</td><td>02</td><td>01</td><td>00</td></tr>
<tr><td colspan="4">add r7,r7,r4</td><td colspan="4">lwzux r4,r5,r6</td><td rowspan="2">08</td></tr>
<tr><td>0F</td><td>0E</td><td>0D</td><td>0C</td><td>0B</td><td>0A</td><td>09</td><td>08</td></tr>
<tr><td colspan="4">b loop</td><td colspan="4">subi r5,r5,4</td><td rowspan="2">10</td></tr>
<tr><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td></tr>
<tr><td colspan="4"></td><td colspan="4">done: stw r7,total</td><td rowspan="2">18</td></tr>
<tr><td>1F</td><td>1E</td><td>1D</td><td>1C</td><td>1B</td><td>1A</td><td>19</td><td>18</td></tr>
</table>

Figure 32. Little-Endian mapping of program ‘p’

Programming Note

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

> ... our Histories of six Thousand Moons make no Mention of any other Regions, than the two great Empires of *Lilliput* and *Blefuscu*. Which two mighty Powers have, as I was going to tell you, been engaged in a most obstinate War for six and thirty Moons past. It began upon the following Occasion. It is allowed on all Hands, that the primitive Way of breaking Eggs before we eat them, was upon the larger End: But his present Majesty's Grand-father, while he was a Boy, going to eat an Egg, and breaking it according to the ancient Practice, happened to cut one of his Fingers. Whereupon the Emperor his Father, published an Edict, commanding all his Subjects, upon great Penalties, to break the smaller End of their Eggs. The People so highly resented this Law, that our Histories tell us, there have been six Rebellions raised on that Account; wherein one Emperor lost his Life, and another his Crown. These civil Commotions were constantly fomented by the Monarchs of *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, Book III, and Book VLE) 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 at effective address $2^{64} - 4$ the effective address of the next sequential instruction is undefined.

In 32-bit mode, the low-order 32 bits of the 64-bit result, preceded by 32 0 bits, comprise the 64-bit effective address for the purpose of addressing storage. When an effective address is placed into a register by an instruction or event, the value placed into the high-order 32 bits of the register differs between the Server environment and the Embedded environment.

- Server environment:
  - Load with Update and Store with Update instructions set the high-order 32 bits of register RA to the high-order 32 bits of the 64-bit result.
  - In all other cases (e.g., the Link Register when set by Branch instructions having LK=1, Special Purpose Registers when set to an effec-

tive address by invocation of a system error handler) the high-order 32 bits of the register are set to 0s except as described in the last sentence of this paragraph.

- Embedded environment:
  The high-order 32 bits of the register are set to an undefined value.

As used to address storage, the effective address arithmetic appears to wrap around from the maximum address, 2^{32} - 1, to address 0, except that if the current instruction is at effective address 2^{32} - 4 the effective address of the next sequential instruction is undefined.

The 64-bit current instruction address is not affected by a change from 32-bit mode to 64-bit mode, but is affected by a change from 64-bit mode to 32-bit mode. In the latter case, the high-order 32 bits are set to 0. The same is true for the 64-bit next instruction address, except as described in the last item of the list below.

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

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

- With X-form instructions, in computing the effective address of a data element, the contents of the GPR designated by RB (or the value zero for ***lswi*** and ***stswi***) are added to the contents of the GPR designated by RA or to zero if RA=0.
- With D-form instructions, the 16-bit D field is sign-extended to form a 64-bit address component. In computing the effective address of a data element, this address component is added to the contents of the GPR designated by RA or to zero if RA=0.
- With DS-form instructions, the 14-bit DS field is concatenated on the right with 0b00 and sign-extended to form a 64-bit address component. In computing the effective address of a data element, this address component is added to the contents of the GPR designated by RA or to zero if RA=0.
- With 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 next instruction. If AA=1, this address component is the effective address of the next 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 next instruction. If AA=1, this address component is the effective address of the next 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 next instruction.
- With sequential instruction fetching, the value 4 is added to the address of the current instruction to form the effective address of the next instruction, except that if the current instruction is at the maximum instruction effective address for the mode (2^{64} - 4 in 64-bit mode, 2^{32} - 4 in 32-bit mode) the effective address of the next sequential instruction is undefined. (There is one other exception to this rule; this exception involves changing between 32-bit mode and 64-bit mode and is described in Section 5.3.2 of Book III-S and Section 4.3.2 of Book III-E.)

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 Processor

2.1 Branch Processor Overview

This chapter describes the registers and instructions that make up the Branch Processor 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* instructions, cause the appropriate system handler to be invoked.
- Exceptions can cause the system error handler to be invoked, as described in Section 1.9, “Exceptions” on page 19.
- 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 Processor 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).

CR	
32	63

Figure 33. 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 $XER_{32:35}$ (***mcrxr***), or from the FPSCR (***mcrfs***).
- CR Field 0 can be set as the implicit result of a fixed-point instruction.
- CR Field 1 can be set as the implicit result of a floating-point instruction.
- CR Field 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	Description
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 ***stwcx.*** and ***stdcx.*** instructions (see Section 3.3.2, "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 0:3 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 102). These bits are interpreted as follows.

Bit	Description
0	***Floating-Point Exception Summary*** (FX) This is a copy of the contents of $FPSCR_{FX}$ at the completion of the instruction.
1	***Floating-Point Enabled Exception Summary*** (FEX) This is a copy of the contents of $FPSCR_{FEX}$ at the completion of the instruction.
2	***Floating-Point Invalid Operation Exception Summary*** (VX) This is a copy of the contents of $FPSCR_{VX}$ at the completion of the instruction.
3	***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 3.3.9, "Fixed-Point Compare Instructions" on page 67, Section 4.6.7, "Floating-Point Compare Instructions" on page 129, and Section 6.3.9, "SPE Instruction Set" on page 208.

Bit	Description
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.

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.

LR	
0	63

Figure 34. 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.

CTR	
0	63

Figure 35. Count Register

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 four ways, as described in Section 1.10.3, “Effective Address Calculation” on page 23.

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

In all four 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 and fourth 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 36. 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 $CTR_{M:63}\neq0$ 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 $CTR_{M:63}\neq0$ 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 $CTR_{M:63}\neq0$
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 36. 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 37.

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 37. "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* and *Branch Conditional to Count Register* instructions, the BH field provides a hint about the use of the instruction, as shown in Figure 38.

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

Figure 38. BH field encodings

Programming Note

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

Extended mnemonics for branches

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

Programming Note

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

The "z" bits should be set to 0, because they may be assigned a meaning in some future version of the architecture.

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) 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.).
- Pair each subroutine call (i.e., each *Branch* instruction for which LK=1 and the branch is taken) 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.

- 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.

Compatibility 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.

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)

18	LI	AA	LK
0	6	30	31

```
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:

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)

16	BO	BI	BD	AA	LK
0	6	11	16	30	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
  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 36. *target_addr* specifies the branch target address.

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

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

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

Special Registers Altered:

CTR (if BO_2=0)
LR (if LK=1)

Extended Mnemonics:

Examples of extended mnemonics for *Branch Conditional*:

Extended:		**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)

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

```
if (64-bit mode)
  then M ← 0
  else M ← 32
if ¬BO_2 then CTR ← CTR - 1
ctr_ok ← BO_2 | ((CTR_M:63 ≠ 0) ⊕ BO_3
cond_ok ← BO_0 | (CR_BI+32 ≡ BO_1)
if ctr_ok & cond_ok then NIA ←_iea LR_0:61 || 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 36. The BH field is used as described in Figure 38. The branch target address is $LR_{0:61}$ || 0b00, with the high-order 32 bits of the branch target address set to 0 in 32-bit mode.

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

Special Registers Altered:

CTR (if BO_2=0)
LR (if LK=1)

Extended Mnemonics:

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

Extended:		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)

19	BO	BI	///	BH	528	LK
0	6	11	16	19	21	31

```
cond_ok ← BO_0 | (CR_BI+32 ≡ BO_1)
if cond_ok then NIA ←_iea CTR_0:61 || 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 36. The BH field is used as described in Figure 38. 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:

LR (if LK=1)

Extended Mnemonics:

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

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

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 D 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:

CR_{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:

CR_{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:

CR_{BT+32}

Extended Mnemonics:

Example of extended mnemonics for *Condition Register OR*:

Extended:	**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:

CR_{BT+32}

Extended Mnemonics:

Example of extended mnemonics for *Condition Register XOR*:

Extended:	**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:

CR_{BT+32}

Extended Mnemonics:

Example of extended mnemonics for *Condition Register NOR*:

Extended:	**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:

CR_{BT+32}

Extended Mnemonics:

Example of extended mnemonics for *Condition Register Equivalent*:

Extended:	**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:

CR_{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:

CR_{BT+32}

2.5.2 Condition Register Field Instruction

Move Condition Register Field XL-form

mcrf BF,BFA

19	BF	//	BFA	//	///	0	/
0	6	9	11	14	16	21	31

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

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

Special Registers Altered:

CR field BF

2.6 System Call Instruction

This instruction provides the means by which a program can call upon the system to perform a service.

System Call *SC-form*

sc LEV

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

This instruction calls the system to perform a service. A complete description of this instruction can be found in Book III.

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

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

This instruction is context synchronizing (see Book III).

Special Registers Altered:

Dependent on the system service

Programming Note

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

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

Chapter 3. Fixed-Point Processor

3.1 Fixed-Point Processor Overview

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

3.2 Fixed-Point Processor Registers

3.2.1 General Purpose Registers

All manipulation of information is done in registers internal to the Fixed-Point Processor. The principal storage internal to the Fixed-Point Processor is a set of 32 General Purpose Registers (GPRs). See Figure 39.

GPR 0
GPR 1
. . . . . .
GPR 30
GPR 31

0 63

Figure 39. 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.

XER

0 63

Figure 40. 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 Description

0:31 Reserved

32 ***Summary Overflow*** (SO)
The Summary Overflow bit is set to 1 whenever an instruction (except ***mtspr***) sets the Overflow bit. Once set, the SO bit remains set until it is cleared by an ***mtspr*** instruction (specifying the XER) or an ***mcrxr*** instruction. It is not altered by *Compare* instructions, nor by other instructions (except ***mtspr*** to the XER, and ***mcrxr***) 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.
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***, ***divdu***) or in 32 bits (***mullw***, ***divw***, ***divwu***), and set it to 0 otherwise. The OV bit is not altered by *Compare* instructions, nor by other instructions (except ***mtspr*** to the XER, and ***mcrxr***) that cannot overflow.

[Category:
Legacy Integer Multiply-Accumulate]
XO-form *Legacy Integer Multiply-Accumulate* instructions set OV when OE=1 to reflect overflow of the 32-bit result. For signed-integer accumulation, overflow occurs when the add produces a carry out of bit 32 that is not equal to the carry out of bit 33. For unsigned-integer accumulation, overflow occurs when the add produces a carry out of bit 32.

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, nor by other instructions (except *Shift Right Algebraic*, ***mtspr*** to the XER, and ***mcrxr***) that cannot carry.

35:56 Reserved

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

[Category: Legacy Move Assist]
This field is used as a target by ***dmlzb*** to indicate the byte location of the leftmost zero byte found.

3.2.3 Program Priority Register [Category: Server]

The Program Priority Register (PPR) is a 64-bit register that controls the program's priority. The layout of the PPR is shown in Figure 41.

///	PRI	///	???
0	11	14	44 63

Bit(s) Description

11:13 ***Program Priority*** (PRI)

010 low
011 medium low
100 medium (normal)

44:63 implementation-specific (read-only; values written to this field by software are ignored)

All other fields are reserved.

Figure 41. Program Priority Register

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 of Book II), 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 3.3.14.

Programming Note

When the system error handler is invoked, the PRI field may be set to an undefined value.

3.2.4 Software Use SPRs [Category: Embedded]

Software Use SPRs are 64-bit registers that have no defined functionality. SPRG4-7 can be read by application programs. Additional Software Use SPRs are defined in Book III.

SPRG4
SPRG5
SPRG6
SPRG7
0 63

Figure 42. Software-use SPRs

The VRSAVE is a 32-bit register that also can be used as a software use SPR. VRSAVE is also defined as part of Category: Embedded and Vector (see Section 5.3.3)

Programming Note

USPRG0 was made a 32-bit register and renamed to VRSAVE; see Section 5.3.3

3.2.5 Device Control Registers [Category: Embedded]

Device Control Registers (DCRs) are on-chip registers that exist architecturally outside the processor and thus are not actually part of the processor architecture. This specification simply defines the existence of a Device Control Register 'address space' and the instructions to access them and does not define the Device Control Registers themselves.

Device Control Registers may control the use of on-chip peripherals, such as memory controllers (the definition of specific Device Control Registers is implementation-dependent).

The contents of user-mode-accessible Device Control Registers can be read using ***mfdcrux*** and written using ***mtdcrux***.

3.3 Fixed-Point Processor 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 23.

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.

Programming Note

The DS field in DS-form Storage Access instructions is a word offset, not a byte offset like the D field in D-form *Storage Access* instructions. However, for programming convenience, Assemblers should support the specification of byte offsets for both forms of instruction.

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

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + EXTS(D)
RT ← 560 || MEM(EA, 1)
```

Let the effective address (EA) be the sum (RA|0)+ D. 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 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 ← 560 || MEM(EA, 1)
```

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

Special Registers Altered:

None

Load Byte and Zero with Update D-form

lbzu RT,D(RA)

35	RT	RA	D
0	6	11	16 31

```
EA ← (RA) + EXTS(D)
RT ← 560 || 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

31	RT	RA	RB	119	/
0	6	11	16	21	31

```
EA ← (RA) + (RB)
RT ← 560 || 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

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + EXTS(D)
RT ← 480 || MEM(EA, 2)
```

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

Special Registers Altered:

None

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 ← 480 || MEM(EA, 2)
```

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

Special Registers Altered:

None

Load Halfword and Zero with Update D-form

lhzu RT,D(RA)

41	RT	RA	D
0	6	11	16 31

```
EA ← (RA) + EXTS(D)
RT ← 480 || 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

31	RT	RA	RB	311	/
0	6	11	16	21	31

```
EA ← (RA) + (RB)
RT ← 480 || 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)

42	RT	RA	D
0	6	11	16 31

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

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

Special Registers Altered:
None

Load Halfword Algebraic with Update D-form

lhau RT,D(RA)

43	RT	RA	D
0	6	11	16 31

```
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 Indexed X-form

lhax RT,RA,RB

31	RT	RA	RB	343	/
0	6	11	16	21	31

```
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 Indexed X-form

lhaux RT,RA,RB

31	RT	RA	RB	375	/
0	6	11	16	21	31

```
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)

32	RT	RA	D
0	6	11	16 31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + EXTS(D)
RT ← 320 || MEM(EA, 4)
```

Let the effective address (EA) be the sum (RA|0)+ D. 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 Indexed X-form

lwzx RT,RA,RB

31	RT	RA	RB	23	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + (RB)
RT ← 320 || MEM(EA, 4)
```

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

Special Registers Altered:

None

Load Word and Zero with Update D-form

lwzu RT,D(RA)

33	RT	RA	D
0	6	11	16 31

```
EA ← (RA) + EXTS(D)
RT ← 320 || MEM(EA, 4)
RA ← EA
```

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

31	RT	RA	RB	55	/
0	6	11	16	21	31

```
EA ← (RA) + (RB)
RT ← 320 || MEM(EA, 4)
RA ← EA
```

Let the effective address (EA) be the sum (RA)+ (RB). The word in storage addressed by EA is loaded into $RT_{32:63}$. $RT_{0:31}$ 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 [Category: 64-Bit]

Load Word Algebraic DS-form

lwa RT,DS(RA)

58	RT	RA	DS	2
0	6	11	16	30 31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + EXTS(DS || 0b00)
RT ← EXTS(MEM(EA, 4))
```

Let the effective address (EA) be the sum (RA|0)+ (DS||0b00). The word in storage addressed by EA is loaded into $RT_{32:63}$. $RT_{0:31}$ are filled with a copy of bit 0 of the loaded word.

Special Registers Altered:

None

Load Word Algebraic Indexed 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,DS(RA)

58	RT	RA	DS	0
0	6	11	16	30 31

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

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

Special Registers Altered:

None

Load Doubleword Indexed — *X-form*

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,DS(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)

38	RS	RA	D
0	6	11	16 … 31

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

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

Special Registers Altered:
None

Store Byte Indexed — *X-form*

stbx RS,RA,RB

31	RS	RA	RB	215	/
0	6	11	16	21	31

```
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)

39	RS	RA	D
0	6	11	16 … 31

```
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

31	RS	RA	RB	247	/
0	6	11	16	21	31

```
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

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + EXTS(D)
MEM(EA, 2) ← (RS)48:63
```

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

Special Registers Altered:

None

Store Halfword Indexed 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)

36	RS	RA	D
0	6	11	16 31

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

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

Special Registers Altered:
None

Store Word Indexed *X-form*

stwx RS,RA,RB

31	RS	RA	RB	151	/
0	6	11	16	21	31

```
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)

37	RS	RA	D
0	6	11	16 31

```
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

31	RS	RA	RB	183	/
0	6	11	16	21	31

```
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 [Category: 64-Bit]

Store Doubleword DS-form

std RS,DS(RA)

62	RS	RA	DS	0
0	6	11	16	30 31

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

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

Special Registers Altered:
None

Store Doubleword with Update DS-form

stdu RS,DS(RA)

62	RS	RA	DS	1
0	6	11	16	30 31

```
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 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 Indexed X-form

stdux RS,RA,RB

31	RS	RA	RB	181	/
0	6	11	16	21	31

```
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 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 ← 480 || load_data8:15 || load_data0:7
```

Let the effective address (EA) be the sum (RA|0)+(RB). Bits 0:7 of the halfword in storage addressed by EA are loaded into $RT_{56:63}$. 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 ← 320 || 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 Fixed-Point Load and Store Multiple Instructions

The *Load/Store Multiple* instructions have preferred forms; see Section 1.8.1, “Preferred Instruction Forms” on page 19. In the preferred forms, storage alignment satisfies the following rule.

- The combination of the EA and RT (RS) is such that the low-order byte of GPR 31 is loaded (stored) from (into) the last byte of an aligned quadword in storage.

For the Server environment, the *Load/Store Multiple* instructions are not supported in Little-Endian mode. If they are executed in Little-Endian mode, the system alignment error handler is invoked.

Load Multiple Word D-form

lmw RT,D(RA)

46	RT	RA	D
0	6	11	16 31

```
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.

Special Registers Altered:

None

Store Multiple Word *D-form*

stmw RS,D(RA)

47	RS	RA	D
0	6	11	16 31

```
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.

Special Registers Altered:
None

3.3.6 Fixed-Point Move Assist Instructions [Category: Move Assist]

The *Move Assist* instructions allow movement of data 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 *Load/Store String* instructions have preferred forms; see Section 1.8.1, “Preferred Instruction Forms” on page 19. In the preferred forms, register usage satisfies the following rules.

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

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

For the Server environment, the *Move Assist* instructions are not supported in Little-Endian mode. If they are executed in Little-Endian mode, the system alignment error handler may be invoked or the instructions may be treated as no-ops if the number of bytes specified by the instruction is 0.

Load String Word Immediate X-form

lswi RT,RA,NB

31	RT	RA	NB	597	/
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 ← 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.

Special Registers Altered:

None

Load String Word Indexed X-form

lswx RT,RA,RB

31	RT	RA	RB	533	/
0	6	11	16	21	31

```
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=$XER_{57:63}$; 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.

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.

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.

Special Registers Altered:
None

3.3.7 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 may execute slowly or may prevent the execution of subsequent instructions until the instruction has completed.

3.3.8 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 3.3.7, “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. The XO-form *Arithmetic* instructions set SO and OV when OE=1 to reflect overflow of the result. Except for the *Multiply Low* and *Divide* instructions, the setting of these bits 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. For XO-form *Multiply Low* and *Divide* instructions, the setting of these bits is mode-independent, and reflects overflow of the 64-bit result for ***mulld***, ***divd***, and ***divdu***, and overflow of the low-order 32-bit result for ***mullw***, ***divw***, and ***divwu***.

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 D for additional extended mnemonics.

Add Immediate — *D-form*

addi RT,RA,SI

14	RT	RA	SI
0	6	11	16 31

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

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

Special Registers Altered:

None

Extended Mnemonics:

Examples of extended mnemonics for *Add Immediate*:

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

Programming Note

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

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

Add Immediate Shifted — *D-form*

addis RT,RA,SI

15	RT	RA	SI
0	6	11	16 31

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

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

Special Registers Altered:

None

Extended Mnemonics:

Examples of extended mnemonics for *Add Immediate Shifted*:

Extended:		**Equivalent to:**	
lis	Rx,value	addis	Rx,0,value
subis	Rx,Ry,value	addis	Rx,Ry,-value

Add 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)

31	RT	RA	RB	OE	266	Rc
0	6	11	16	21	22	31

```
RT ← (RA) + (RB)
```

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

Special Registers Altered:

CR0 (if Rc=1)
SO OV (if OE=1)

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)

31	RT	RA	RB	OE	40	Rc
0	6	11	16	21	22	31

```
RT ← ¬(RA) + (RB) + 1
```

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

Special Registers Altered:

CR0 (if Rc=1)
SO OV (if OE=1)

Extended Mnemonics:

Example of extended mnemonics for *Subtract From*:

Extended:		Equivalent to:	
sub	Rx,Ry,Rz	subf	Rx,Rz,Ry

Add Immediate Carrying D-form

addic RT,RA,SI

12	RT	RA	SI
0	6	11	16 31

```
RT ← (RA) + EXTS(SI)
```

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

Special Registers Altered:

CA

Extended Mnemonics:

Example of extended mnemonics for *Add Immediate Carrying*:

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

Add Immediate Carrying and Record D-form

addic. RT,RA,SI

13	RT	RA	SI
0	6	11	16 31

```
RT ← (RA) + EXTS(SI)
```

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

Special Registers Altered:

CR0 CA

Extended Mnemonics:

Example of extended mnemonics for *Add Immediate Carrying and Record*:

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

Subtract From Immediate Carrying D-form

subfic RT,RA,SI

8	RT	RA	SI
0	6	11	16 31

RT ← ¬(RA) + EXTS(SI) + 1

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

Special Registers Altered:

CA

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)

31	RT	RA	RB	OE	10	Rc
0	6	11	16	21	22	31

```
RT ← (RA) + (RB)
```

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

Special Registers Altered:

CA	
CR0	(if Rc=1)
SO OV	(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)

31	RT	RA	RB	OE	8	Rc
0	6	11	16	21	22	31

```
RT ← ¬(RA) + (RB) + 1
```

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

Special Registers Altered:

CA	
CR0	(if Rc=1)
SO OV	(if OE=1)

Extended Mnemonics:

Example of extended mnemonics for *Subtract From Carrying*:

Extended:	**Equivalent to:**
subc Rx,Ry,Rz	subfc Rx,Rz,Ry

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)

31	RT	RA	RB	OE	138	Rc
0	6	11	16	21	22	31

```
RT ← (RA) + (RB) + CA
```

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

Special Registers Altered:

CA
CR0 (if Rc=1)
SO OV (if OE=1)

Add to Minus One Extended *XO-form*

addme RT,RA (OE=0 Rc=0)
addme. RT,RA (OE=0 Rc=1)
addmeo RT,RA (OE=1 Rc=0)
addmeo. RT,RA (OE=1 Rc=1)

31	RT	RA	///	OE	234	Rc
0	6	11	16	21	22	31

```
RT ← (RA) + CA - 1
```

The sum (RA) + CA + $^{64}1$ is placed into register RT.

Special Registers Altered:

CA
CR0 (if Rc=1)
SO OV (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)

31	RT	RA	RB	OE	136	Rc
0	6	11	16	21	22	31

```
RT ← ¬(RA) + (RB) + CA
```

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

Special Registers Altered:

CA
CR0 (if Rc=1)
SO OV (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)

31	RT	RA	///	OE	232	Rc
0	6	11	16	21	22	31

```
RT ← ¬(RA) + CA - 1
```

The sum ¬(RA) + CA + $^{64}1$ is placed into register RT.

Special Registers Altered:

CA
CR0 (if Rc=1)
SO OV (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)

31	RT	RA	///	OE	202	Rc
0	6	11	16	21	22	31

```
RT ← (RA) + CA
```

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

Special Registers Altered:

CA	
CR0	(if Rc=1)
SO OV	(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)

31	RT	RA	///	OE	200	Rc
0	6	11	16	21	22	31

```
RT ← ¬(RA) + CA
```

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

Special Registers Altered:

CA	
CR0	(if Rc=1)
SO OV	(if OE=1)

Programming Note

The setting of CA 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.

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)

31	RT	RA	///	OE	104	Rc
0	6	11	16	21	22	31

```
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. 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 is set to 1.

Special Registers Altered:

CR0	(if Rc=1)
SO OV	(if OE=1)

Multiply Low Immediate — *D-form*

mulli RT,RA,SI

7	RT	RA	SI
0	6	11	16 31

```
prod0:127 ← (RA) × EXTS(SI)
RT ← prod64:127
```

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)

31	RT	RA	RB	OE	235	Rc
0	6	11	16	21	22	31

```
RT ← (RA)32:63 × (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 is set to 1 if the product cannot be represented in 32 bits.

Both operands and the product are interpreted as signed integers.

Special Registers Altered:

CR0	(if Rc=1)
SO OV	(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)

31	RT	RA	RB	/	75	Rc
0	6	11	16	21	22	31

```
prod0:63 ← (RA)32:63 × (RB)32:63
RT32:63 ← prod0:31
RT0:31 ← 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:
CR0 (bits 0:2 undefined in 64-bit mode) (if Rc=1)

Multiply High Word Unsigned — *XO-form*

mulhwu	RT,RA,RB	(Rc=0)
mulhwu.	RT,RA,RB	(Rc=1)

31	RT	RA	RB	/	11	Rc
0	6	11	16	21	22	31

```
prod0:63 ← (RA)32:63 × (RB)32:63
RT32:63 ← prod0:31
RT0:31 ← 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:
CR0 (bits 0:2undefined in 64-bit mode) (if Rc=1)

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)

31	RT	RA	RB	OE	491	Rc
0	6	11	16	21	22	31

```
dividend0:63 ← EXTS((RA)32:63)
divisor0:63 ← EXTS((RB)32:63)
RT32:63 ← dividend ÷ divisor
RT0:31 ← undefined
```

The 64-bit dividend is the sign-extended value of $(RA)_{32:63}$. The 64-bit divisor is the sign-extended value of $(RB)_{32:63}$. The 64-bit quotient is formed. The low-order 32 bits of the 64-bit quotient are 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 $0 \le r < |divisor|$ if the dividend is nonnegative, and $-|divisor| < r \le 0$ 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 is set to 1.

Special Registers Altered:

CR0 (bits 0:2 undefined in 64-bit mode) (if Rc=1)
SO OV (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)

31	RT	RA	RB	OE	459	Rc
0	6	11	16	21	22	31

```
dividend0:63 ← 320 || (RA)32:63
divisor0:63 ← 320 || (RB)32:63
RT32:63 ← dividend ÷ divisor
RT0:31 ← undefined
```

The 64-bit dividend is the zero-extended value of $(RA)_{32:63}$. The 64-bit divisor is the zero-extended value of $(RB)_{32:63}$. The 64-bit quotient is formed. The low-order 32 bits of the 64-bit quotient are 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 $0 \le r <$ 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 is set to 1.

Special Registers Altered:

CR0 (bits 0:2 undefined in 64-bit mode) (if Rc=1)
SO OV (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
```

3.3.8.1 64-bit Fixed-Point Arithmetic Instructions [Category: 64-Bit]

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)

31	RT	RA	RB	OE	233	Rc
0	6	11	16	21	22	31

$prod_{0:127} \leftarrow (RA) \times (RB)$
$RT \leftarrow prod_{64:127}$

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 is set to 1 if the product cannot be represented in 64 bits.

Both operands and the product are interpreted as signed integers.

Special Registers Altered:

CR0 (if Rc=1)
SO OV (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 Unsigned XO-form

mulhdu	RT,RA,RB	(Rc=0)
mulhdu.	RT,RA,RB	(Rc=1)

31	RT	RA	RB	/	9	Rc
0	6	11	16	21	22	31

$prod_{0:127} \leftarrow (RA) \times (RB)$
$RT \leftarrow prod_{0:63}$

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:

CR0 (if Rc=1)

Multiply High Doubleword XO-form

mulhd	RT,RA,RB	(Rc=0)
mulhd.	RT,RA,RB	(Rc=1)

31	RT	RA	RB	/	73	Rc
0	6	11	16	21	22	31

$prod_{0:127} \leftarrow (RA) \times (RB)$
$RT \leftarrow prod_{0:63}$

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:

CR0 (if Rc=1)

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)

31	RT	RA	RB	OE	489	Rc
0	6	11	16	21	22	31

```
dividend0:63 ← (RA)
divisor0:63 ← (RB)
RT ← dividend ÷ divisor
```

The 64-bit dividend is (RA). The 64-bit divisor is (RB). The 64-bit quotient of the dividend and divisor 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 $0 \le r < |divisor|$ if the dividend is nonnegative, and $-|divisor| < r \le 0$ 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 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 is set to 1.

Special Registers Altered:

CR0	(if Rc=1)
SO OV	(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)

31	RT	RA	RB	OE	457	Rc
0	6	11	16	21	22	31

```
dividend0:63 ← (RA)
divisor0:63 ← (RB)
RT ← dividend ÷ divisor
```

The 64-bit dividend is (RA). The 64-bit divisor is (RB). The 64-bit quotient of the dividend and divisor 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 $0 \le r < divisor$.

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 is set to 1.

Special Registers Altered:

CR0	(if Rc=1)
SO OV	(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
```

3.3.9 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

L=1 is part of Category: 64-Bit.

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 D for additional extended mnemonics.

Compare Immediate *D-form*

cmpi BF,L,RA,SI

11	BF	/	L	RA	SI
0	6	9	10	11	16 31

```
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:
CR field BF

Extended Mnemonics:

Examples of extended mnemonics for Compare Immediate:

Extended:		Equivalent to:	
cmpdi	Rx,value	cmpi	0,1,Rx,value
cmpwi	cr3,Rx,value	cmpi	3,0,Rx,value

Compare *X-form*

cmp BF,L,RA,RB

31	BF	/	L	RA	RB	0	/
0	6	9	10	11	16	21	31

```
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:
CR field BF

Extended Mnemonics:

Examples of extended mnemonics for Compare:

Extended:		Equivalent to:	
cmpd	Rx,Ry	cmp	0,1,Rx,Ry
cmpw	cr3,Rx,Ry	cmp	3,0,Rx,Ry

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:
CR field BF

Extended Mnemonics:

Examples of extended mnemonics for *Compare Logical Immediate*:

Extended:		Equivalent to:	
cmpldi	Rx,value	cmpli	0,1,Rx,value
cmplwi	cr3,Rx,value	cmpli	3,0,Rx,value

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:
CR field BF

Extended Mnemonics:

Examples of extended mnemonics for *Compare Logical*:

Extended:		Equivalent to:	
cmpld	Rx,Ry	cmpl	0,1,Rx,Ry
cmplw	cr3,Rx,Ry	cmpl	3,0,Rx,Ry

3.3.10 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 D for additional extended mnemonics.

Trap Word Immediate — *D-form*

twi TO,RA,SI

3	TO	RA	SI
0	6	11	16 ... 31

```
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:		Equivalent to:	
twgti	Rx,value	twi	8,Rx,value
twllei	Rx,value	twi	6,Rx,value

Trap Word — *X-form*

tw TO,RA,RB

31	TO	RA	RB	4	/
0	6	11	16	21	31

```
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:		Equivalent to:	
tweq	Rx,Ry	tw	4,Rx,Ry
twlge	Rx,Ry	tw	5,Rx,Ry
trap		tw	31,0,0

3.3.10.1 64-bit Fixed-Point Trap Instructions [Category: 64-Bit]

Trap Doubleword Immediate *D-form*

tdi TO,RA,SI

2	TO	RA	SI
0	6	11	16 31

```
a ← (RA)
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 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:		**Equivalent to:**	
tdlti	Rx,value	tdi	16,Rx,value
tdnei	Rx,value	tdi	24,Rx,value

Trap Doubleword *X-form*

td TO,RA,RB

31	TO	RA	RB	68	/
0	6	11	16	21	31

```
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:		**Equivalent to:**	
tdge	Rx,Ry	td	12,Rx,Ry
tdlnl	Rx,Ry	td	5,Rx,Ry

3.3.11 Fixed-Point Select [Category: Base.Phased-In]

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

3.3.12 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 3.3.7, “Other Fixed-Point Instructions” on page 57. The Logical instructions do not change the SO, OV, and CA bits in the XER.

Extended mnemonics for logical operations

An extended mnemonic is provided that generates the preferred form of “no-op” (an instruction that does nothing). This is shown as an example with the *OR Immediate* instruction.

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 D, “Assembler Extended Mnemonics” on page 317 for additional extended mnemonics.

AND Immediate — *D-form*

andi. RA,RS,UI

28	RS	RA	UI
0	6	11	16 31

```
RA ← (RS) & (⁴⁸0 || UI)
```

The contents of register RS are ANDed with $^{48}0$ || UI and the result is placed into register RA.

Special Registers Altered:
CR0

AND Immediate Shifted — *D-form*

andis. RA,RS,UI

29	RS	RA	UI
0	6	11	16 31

```
RA ← (RS) & (³²0 || UI || ¹⁶0)
```

The contents of register RS are ANDed with $^{32}0$ || UI || $^{16}0$ and the result is placed into register RA.

Special Registers Altered:
CR0

OR Immediate — *D-form*

ori RA,RS,UI

24	RS	RA	UI
0	6	11	16 31

```
RA ← (RS) | (⁴⁸0 || 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
```

Special Registers Altered:
None

Extended Mnemonics:

Example of extended mnemonics for *OR Immediate*:

Extended:	**Equivalent to:**
nop	ori 0,0,0

OR Immediate Shifted *D-form*

oris RA,RS,UI

25	RS	RA	UI
0	6	11	16 31

```
RA ← (RS) | (³²0 || UI || ¹⁶0)
```

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 (⁴⁸0 || UI)
```

The contents of register RS are XORed with $^{48}0$ || UI and the result is placed into register RA.

Special Registers Altered:
None

XOR Immediate Shifted *D-form*

xoris RA,RS,UI

27	RS	RA	UI
0	6	11	16 31

```
RA ← (RS) XOR (³²0 || UI || ¹⁶0)
```

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)

31	RS	RA	RB	28	Rc
0	6	11	16	21	31

```
RA ← (RS) & (RB)
```

The contents of register RS are ANDed with the contents of register RB and the result is placed into register RA.

Special Registers Altered:
CR0 (if Rc=1)

XOR — *X-form*

xor RA,RS,RB (Rc=0)
xor. RA,RS,RB (Rc=1)

31	RS	RA	RB	316	Rc
0	6	11	16	21	31

```
RA ← (RS) ⊕ (RB)
```

The contents of register RS are XORed with the contents of register RB and the result is placed into register RA.

Special Registers Altered:
CR0 (if Rc=1)

NAND — *X-form*

nand RA,RS,RB (Rc=0)
nand. RA,RS,RB (Rc=1)

31	RS	RA	RB	476	Rc
0	6	11	16	21	31

```
RA ← ¬((RS) & (RB))
```

The contents of register RS are ANDed with the contents of register RB and the complemented result is placed into register RA.

Special Registers Altered:
CR0 (if Rc=1)

Programming Note

nand or ***nor*** with RS=RB can be used to obtain the one's complement.

OR — *X-form*

or RA,RS,RB (Rc=0)
or. RA,RS,RB (Rc=1)

31	RS	RA	RB	444	Rc
0	6	11	16	21	31

```
RA ← (RS) | (RB)
```

The contents of register RS are ORed with the contents of register RB and the result is placed into register RA.

For implementations that support the PPR (see Section 3.2.3), *or Rx,Rx,Rx* can be used to set PPR_{PRI} as shown in Figure 43. *or. Rx,Rx,Rx* does not set PPR_{PRI}.

Rx	PPR_{PRI}	Priority
1	010	low
6	011	medium low
2	100	medium (normal)

Figure 43. Priority levels for *or Rx,Rx,Rx*

Special Registers Altered:
CR0 (if Rc=1)

Extended Mnemonics:

Example of extended mnemonics for *OR*:

Extended:		**Equivalent to:**	
mr	Rx,Ry	or	Rx,Ry,Ry

Programming Note

Warning: Other forms of *or Rx,Rx,Rx* that are not described in Figure 43 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.

NOR — *X-form*

nor RA,RS,RB (Rc=0)
nor. RA,RS,RB (Rc=1)

31	RS	RA	RB	124	Rc
0	6	11	16	21	31

```
RA ← ¬((RS) | (RB))
```

The contents of register RS are ORed with the contents of register RB and the complemented result is placed into register RA.

Special Registers Altered:

CR0 (if Rc=1)

Extended Mnemonics:

Example of extended mnemonics for *NOR*:

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

Equivalent — *X-form*

eqv RA,RS,RB (Rc=0)
eqv. RA,RS,RB (Rc=1)

31	RS	RA	RB	284	Rc
0	6	11	16	21	31

```
RA ← (RS) ≡ (RB)
```

The contents of register RS are XORed with the contents of register RB and the complemented result is placed into register RA.

Special Registers Altered:

CR0 (if Rc=1)

AND with Complement — *X-form*

andc RA,RS,RB (Rc=0)
andc. RA,RS,RB (Rc=1)

31	RS	RA	RB	60	Rc
0	6	11	16	21	31

```
RA ← (RS) & ¬(RB)
```

The contents of register RS are ANDed with the complement of the contents of register RB and the result is placed into register RA.

Special Registers Altered:

CR0 (if Rc=1)

OR with Complement — *X-form*

orc RA,RS,RB (Rc=0)
orc. RA,RS,RB (Rc=1)

31	RS	RA	RB	412	Rc
0	6	11	16	21	31

```
RA ← (RS) | ¬(RB)
```

The contents of register RS are ORed with the complement of the contents of register RB and the result is placed into register RA.

Special Registers Altered:

CR0 (if Rc=1)

Extend Sign Byte — *X-form*

extsb RA,RS (Rc=0)
extsb. RA,RS (Rc=1)

31	RS	RA	///	954	Rc
0	6	11	16	21	31

$s \leftarrow (RS)_{56}$
$RA_{56:63} \leftarrow (RS)_{56:63}$
$RA_{0:55} \leftarrow {}^{56}s$

$(RS)_{56:63}$ are placed into $RA_{56:63}$. Bit 56 of register RS is placed into $RA_{0:55}$.

Special Registers Altered:

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

$s \leftarrow (RS)_{48}$
$RA_{48:63} \leftarrow (RS)_{48:63}$
$RA_{0:47} \leftarrow {}^{48}s$

$(RS)_{48:63}$ are placed into $RA_{48:63}$. Bit 48 of register RS is placed into $RA_{0:47}$.

Special Registers Altered:

CR0 (if Rc=1)

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=1, CR Field 0 is set to reflect the result.

Special Registers Altered:

CR0 (if Rc=1)

Programming Note

For both *Count Leading Zeros* instructions, if Rc=1 then LT is set to 0 in CR Field 0.

3.3.12.1 64-bit Fixed-Point Logical Instructions [Category: 64-Bit]

Extend Sign Word *X-form*

extsw	RA,RS	(Rc=0)
extsw.	RA,RS	(Rc=1)

31	RS	RA	///	986	Rc
0	6	11	16	21	31

```
s ← (RS)32
RA32:63 ← (RS)32:63
RA0:31 ← 32s
```

$(RS)_{32:63}$ are placed into $RA_{32:63}$. Bit 32 of register RS is placed into $RA_{0:31}$.

Special Registers Altered:

CR0 (if Rc=1)

Count Leading Zeros Doubleword *X-form*

cntlzd	RA,RS	(Rc=0)
cntlzd.	RA,RS	(Rc=1)

31	RS	RA	///	58	Rc
0	6	11	16	21	31

```
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:

CR0 (if Rc=1)

3.3.12.2 Phased-In Fixed-Point Logical Instructions [Category: Base.Phased-In]

Population Count Bytes *X-form*

popcntb RA, RS

31	RS	RA	///	122	/
0	6	11	16	21	31

```
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

Programming Note

The total number of one bits in register RS can be computed as follows. In this example it is assumed that register RB contains the value 0x0101_0101_0101_0101

```
popcntb   RA,RS
mulld     RT,RA,RB
srdi      RT,RT,56     # RT = population count
```

3.3.13 Fixed-Point Rotate and Shift Instructions

The Fixed-Point Processor 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 3.3.7, “Other Fixed-Point Instructions” on page 57. *Rotate* and *Shift* instructions do not change the OV and SO bits. *Rotate* and *Shift* instructions, except algebraic right shifts, do not change the CA bit.

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 D, “Assembler Extended Mnemonics” on page 317 for additional extended mnemonics.

3.3.13.1 Fixed-Point Rotate Instructions

These instructions rotate the contents of a register. The result of the rotation is

- inserted into the target register under control of a mask (if a mask bit is 1 the associated bit of the rotated data is placed into the target register, and if the mask bit is 0 the associated bit in the target register remains unchanged); or
- ANDed with a mask before being placed into the target register.

The *Rotate Left* instructions allow right-rotation of the contents of a register to be performed (in concept) by a left-rotation of 64-n, where n is the number of bits by which to rotate right. They allow right-rotation of the contents of the low-order 32 bits of a register to be performed (in concept) by a left-rotation of 32-n, where n is the number of bits by which to rotate right.

Rotate Left Word Immediate then AND with Mask *M-form*

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:

CR0 (if Rc=1)

Extended Mnemonics:

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

Extended:		**Equivalent to:**	
extlwi	Rx,Ry,n,b	rlwinm	Rx,Ry,b,0,n-1
srwi	Rx,Ry,n	rlwinm	Rx,Ry,32-n,n,31
clrrwi	Rx,Ry,n	rlwinm	Rx,Ry,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 D, "Assembler Extended Mnemonics" on page 317.

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)

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

$n \leftarrow (RB)_{59:63}$
$r \leftarrow ROTL_{32}((RS)_{32:63}, n)$
$m \leftarrow MASK(MB+32, ME+32)$
$RA \leftarrow r \& m$

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:
CR0 (if Rc=1)

Extended Mnemonics:

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

Extended:		Equivalent to:	
rotlw	Rx,Ry,Rz	rlwnm	Rx,Ry,Rz,0,31

Programming Note

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

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 D, “Assembler Extended Mnemonics” on page 317.

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)

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

$n \leftarrow SH$
$r \leftarrow ROTL_{32}((RS)_{32:63}, n)$
$m \leftarrow MASK(MB+32, ME+32)$
$RA \leftarrow r\&m \mid (RA)\&\neg m$

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:
CR0 (if Rc=1)

Extended Mnemonics:

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

Extended:		Equivalent to:	
inslwi	Rx,Ry,n,b	rlwimi	Rx,Ry,32-b,b,b+n-1

Programming Note

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

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 D, “Assembler Extended Mnemonics” on page 317.

3.3.13.1.1 64-bit Fixed-Point Rotate Instructions [Category: 64-Bit]

Rotate Left Doubleword Immediate then Clear Left — *MD-form*

rldicl	RA,RS,SH,MB	(Rc=0)
rldicl.	RA,RS,SH,MB	(Rc=1)

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

```
n ← sh5 || sh0:4
r ← ROTL64((RS), n)
b ← mb5 || mb0:4
m ← MASK(b, 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:

CR0 (if Rc=1)

Extended Mnemonics:

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

Extended:		Equivalent to:	
extrdi	Rx,Ry,n,b	rldicl	Rx,Ry,b+n,64-n
srdi	Rx,Ry,n	rldicl	Rx,Ry,64-n,n
clrldi	Rx,Ry,n	rldicl	Rx,Ry,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 D, “Assembler Extended Mnemonics” on page 317.

Rotate Left Doubleword Immediate then Clear Right — *MD-form*

rldicr	RA,RS,SH,ME	(Rc=0)
rldicr.	RA,RS,SH,ME	(Rc=1)

30	RS	RA	sh	me	1	sh	Rc
0	6	11	16	21	27	30	31

```
n ← sh5 || sh0:4
r ← ROTL64((RS), n)
e ← me5 || me0:4
m ← MASK(0, e)
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:

CR0 (if Rc=1)

Extended Mnemonics:

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

Extended:		Equivalent to:	
extldi	Rx,Ry,n,b	rldicr	Rx,Ry,b,n-1
sldi	Rx,Ry,n	rldicr	Rx,Ry,n,63-n
clrrdi	Rx,Ry,n	rldicr	Rx,Ry,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 D, “Assembler Extended Mnemonics” on page 317.

Rotate Left Doubleword Immediate then Clear MD-form

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

30	RS	RA	sh	mb	2	sh	Rc
0	6	11	16	21	27	30	31

```
n ← sh5 || sh0:4
r ← ROTL64((RS), n)
b ← mb5 || mb0:4
m ← MASK(b, ¬n)
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:

CR0 (if Rc=1)

Extended Mnemonics:

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

Extended:		**Equivalent to:**	
clrlsldi	Rx,Ry,b,n	rldic	Rx,Ry,n,b - n

Programming Note

rldic can be used to clear the high-order b bits of the contents of a register and then shift the result left by n bits, by setting SH=n and MB=b - n. It can be used to clear the high-order n bits of a register, by setting SH=0 and MB=n.

Extended mnemonics are provided for both of these uses (the second devolves to ***rldicl***); see Appendix D, “Assembler Extended Mnemonics” on page 317.

Rotate Left Doubleword then Clear Left MDS-form

rldcl	RA,RS,RB,MB	(Rc=0)
rldcl.	RA,RS,RB,MB	(Rc=1)

30	RS	RA	RB	mb	8	Rc
0	6	11	16	21	27	31

```
n ← (RB)58:63
r ← ROTL64((RS), n)
b ← mb5 || mb0:4
m ← MASK(b, 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:

CR0 (if Rc=1)

Extended Mnemonics:

Example of extended mnemonics for *Rotate Left Doubleword then Clear Left*:

Extended:		**Equivalent to:**	
rotld	Rx,Ry,Rz	rldcl	Rx,Ry,Rz,0

Programming Note

rldcl can be used to extract an n-bit field that starts at variable bit position b in register RS, right-justified into register RA (clearing the remaining 64 - n bits of RA), by setting $RB_{58:63}$=b+n and MB=64 - n. It can be used to rotate the contents of a register left (right) by variable n bits, by setting $RB_{58:63}$=n (64 - n) and MB=0.

Extended mnemonics are provided for some of these uses; see Appendix D, “Assembler Extended Mnemonics” on page 317.

Rotate Left Doubleword then Clear Right MDS-form

rldcr	RA,RS,RB,ME	(Rc=0)
rldcr.	RA,RS,RB,ME	(Rc=1)

30	RS	RA	RB	me	9	Rc
0	6	11	16	21	27	31

```
n ← (RB)58:63
r ← ROTL64((RS), n)
e ← me5 || me0:4
m ← MASK(0, e)
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:

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 D, “Assembler Extended Mnemonics” on page 317.

Rotate Left Doubleword Immediate then Mask Insert MD-form

rldimi	RA,RS,SH,MB	(Rc=0)
rldimi.	RA,RS,SH,MB	(Rc=1)

30	RS	RA	sh	mb	3	sh	Rc
0	6	11	16	21	27	30	31

```
n ← sh5 || sh0:4
r ← ROTL64((RS), n)
b ← mb5 || mb0:4
m ← MASK(b, ¬n)
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:

CR0 (if Rc=1)

Extended Mnemonics:

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

Extended:		**Equivalent to:**	
insrdi	Rx,Ry,n,b	rldimi	Rx,Ry,64-(b+n),b

Programming Note

rldimi can be used to insert an n-bit field that is right-justified in register RS, into register RA starting at bit position b, by setting SH=64-(b+n) and MB=b.

An extended mnemonic is provided for this use; see Appendix D, “Assembler Extended Mnemonics” on page 317.

3.3.13.2 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 D, “Assembler Extended Mnemonics” on page 317 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 bit by the Shift Right Algebraic instructions is independent of mode.

Programming Note

Multiple-precision shifts can be programmed as shown in Section E.1, “Multiple-Precision Shifts” on page 331.

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:

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:

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
CA ← s & ((r&¬m)32:63≠0)
```

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 is 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 is set to 0. A shift amount of zero causes RA to receive $EXTS((RS)_{32:63})$, and CA to be set to 0.

Special Registers Altered:

CA
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
CA ← s & ((r&¬m)32:63≠0)
```

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 is 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 is set to 0. A shift amount of zero causes RA to receive $EXTS((RS)_{32:63})$, and CA to be set to 0. Shift amounts from 32 to 63 give a result of 64 sign bits, and cause CA to receive the sign bit of $(RS)_{32:63}$.

Special Registers Altered:

CA
CR0 (if Rc=1)

3.3.13.2.1 64-bit Fixed-Point Shift Instructions [Category: 64-Bit]

Shift Left Doubleword *X-form*

sld RA,RS,RB (Rc=0)
sld. RA,RS,RB (Rc=1)

31	RS	RA	RB	27	Rc
0	6	11	16	21	31

```
n ← (RB)58:63
r ← ROTL64((RS), n)
if (RB)57 = 0 then
     m ← MASK(0, 63-n)
else m ← 640
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:
CR0 (if Rc=1)

Shift Right Doubleword *X-form*

srd RA,RS,RB (Rc=0)
srd. RA,RS,RB (Rc=1)

31	RS	RA	RB	539	Rc
0	6	11	16	21	31

```
n ← (RB)58:63
r ← ROTL64((RS), 64-n)
if (RB)57 = 0 then
    m ← MASK(n, 63)
else m ← 640
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:
CR0 (if Rc=1)

Shift Right Algebraic Doubleword Immediate *XS-form*

sradi RA,RS,SH (Rc=0)
sradi. RA,RS,SH (Rc=1)

31	RS	RA	sh	413	sh	Rc
0	6	11	16	21	30	31

```
n ← sh5 || sh0:4
r ← ROTL64((RS), 64-n)
m ← MASK(n, 63)
s ← (RS)0
RA ← r&m | (64s)&¬m
CA ← s & ((r&¬m)≠0)
```

The contents of register RS are shifted right SH bits. Bits shifted out of position 63 are lost. Bit 0 of RS is replicated to fill the vacated positions on the left. The result is placed into register RA. CA is set to 1 if (RS) is negative and any 1-bits are shifted out of position 63; otherwise CA is set to 0. A shift amount of zero causes RA to be set equal to (RS), and CA to be set to 0.

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

Shift Right Algebraic Doubleword *X-form*

srad RA,RS,RB (Rc=0)
srad. RA,RS,RB (Rc=1)

31	RS	RA	RB	794	Rc
0	6	11	16	21	31

```
n ← (RB)58:63
r ← ROTL64((RS), 64-n)
if (RB)57 = 0 then
    m ← MASK(n, 63)
else m ← 640
s ← (RS)0
RA ← r&m | (64s)&¬m
CA ← s & ((r&¬m)≠0)
```

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 is set to 1 if (RS) is negative and any 1-bits are shifted out of position 63; otherwise CA is set to 0. A shift amount of zero causes RA to be set equal to (RS), and CA to be set to 0. Shift amounts from 64 to 127 give a result of 64 sign bits in RA, and cause CA to receive the sign bit of (RS).

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

3.3.14 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 19. 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 D, “Assembler Extended Mnemonics” on page 317 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
if length(SPR(n)) = 64 then
  SPR(n) ← (RS)
else
  SPR(n) ← (RS)32:63
```

The SPR field denotes a Special Purpose Register, encoded as shown in the table below. The contents of register RS are placed into the designated Special Purpose Register. For Special Purpose Registers that are 32 bits long, the low-order 32 bits of RS are placed into the SPR.

decimal	SPR[1] $spr_{5:9}$ $spr_{0:4}$	Register Name
1	00000 00001	XER
8	00000 01000	LR
9	00000 01001	CTR
256	01000 00000	VRSAVE[2]
512	10000 00000	SPEFSCR[3]
896	11100 00000	PPR[4]

[1] Note that the order of the two 5-bit halves of the SPR number is reversed.
[2] Category: Embedded and Vector (<E> see Programming Note in Section 3.2.4).
[3] Category: SPE.
[4] Category: Server.

If the SPR field contains any value other than one of the values shown above then one of the following occurs.

- The system illegal instruction error handler is invoked.
- The system privileged instruction error handler is invoked.
- The results are boundedly undefined.

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:		Equivalent to:	
mtxer	Rx	mtspr	1,Rx
mtlr	Rx	mtspr	8,Rx
mtctr	Rx	mtspr	9,Rx

Compiler and Assembler Note

For the ***mtspr*** and ***mfspr*** instructions, the SPR number coded in assembler language does not appear directly as a 10-bit binary number in the instruction. The number coded is split into two 5-bit halves that are reversed in the instruction, with the high-order 5 bits appearing in bits 16:20 of the instruction and the low-order 5 bits in bits 11:15.

Move From Special Purpose Register XFX-form

mfspr RT,SPR

31	RT	spr	339	/
0	6	11	21	31

```
n ← spr5:9 || spr0:4
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. 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.

decimal	SPR[1] $spr_{5:9}$ $spr_{0:4}$	Register Name
1	00000 00001	XER
8	00000 01000	LR
9	00000 01001	CTR
256	01000 00000	VRSAVE[2]
260	01000 00100	SPRG4[3]
261	01000 00101	SPRG5[3]
262	01000 00110	SPRG6[3]
263	01000 00111	SPRG7[3]
268	01000 01100	TB[4]
269	01000 01101	TBU[4]
512	10000 00000	SPEFSCR[5]
526	10000 01110	ATB[4,6]
527	10000 01111	ATBU[4,6]
896	11100 00000	PPR[7]

1 Note that the order of the two 5-bit halves of the SPR number is reversed.
2 Category: Embedded and Vector (<E> see Programming Note in Section 3.2.4).
3 Category: Embedded.
4 See Chapter 4 of Book II.
5 Category: SPE.
6 Category: Alternate Time Base.
7 Category: Server.

If the SPR field contains any value other than one of the values shown above then one of the following occurs.

- The system illegal instruction error handler is invoked.
- The system privileged instruction error handler is invoked.
- The results are boundedly undefined.

A complete description of this instruction can be found in Book III.

Special Registers Altered:
None

Extended Mnemonics:

Examples of extended mnemonics for Move From Special Purpose Register:

Extended:		Equivalent to:	
mfxer	Rx	mfspr	Rx,1
mflr	Rx	mfspr	Rx,8
mfctr	Rx	mfspr	Rx,9

.

Note

See the Notes that appear with ***mtspr***.

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×i+32:4×i+35) is set to the contents of the corresponding field of the low-order 32 bits of RS.

Special Registers Altered:
CR fields selected by mask

Extended Mnemonics:

Example of extended mnemonics for *Move To Condition Register Fields*:

Extended:	**Equivalent to:**
mtcr Rx	mtcrf 0xFF,Rx

Programming Note

In the preferred form of this instruction (***mtocrf***), only one Condition Register field is updated.

Move From Condition Register XFX-form

mfcr RT

31	RT	0	///	19	/
0	6	11	12	21	31

```
RT ← ³²0 || CR
```

The contents of the Condition Register are placed into $RT_{32:63}$. $RT_{0:31}$ are set to 0.

Special Registers Altered:
None

Move To One Condition Register Field XFX-form

mtocrf FXM,RS
[Category: Phased-In]

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 ($0 \le n \le 7$). The contents of bits $4\times n+32:4\times n+35$ of register RS are placed into CR field n (CR bits $4\times n+32:4\times n+35$). Otherwise, the contents of the Condition Register are undefined.

Special Registers Altered:
CR field selected by FXM

Programming Note

These forms of the ***mtcrf*** and ***mfcr*** instructions are intended to replace the old forms of the instructions (the forms shown in page 89), which will eventually be phased out of the architecture. The new forms are backward compatible with most processors that comply with versions of the architecture that precede Version 2.00. On those processors, the new forms are treated as the old forms.

However, on some processors that comply with versions of the architecture that precede Version 2.00 the new forms may be treated as follows:

mtocrf: may cause the system illegal instruction error handler to be invoked

mfocrf: may place an undefined value into register RT

Move From One Condition Register Field XFX-form

mfocrf RT,FXM
[Category: Phased-In]

31	RT	1	FXM	/	19	/
0	6	11	12	20	21	31

```
RT ← undefined
count ← 0
do i = 0 to 7
  if FXM_i = 1 then
    n ← i
    count ← count + 1
if count = 1 then
   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 ($0 \le n \le 7$). The contents of CR field n (CR bits $4\times n+32:4\times n+35$) are placed into bits $4\times n+32:4\times n+35$ of register RT and the contents of the remaining bits of register RT are undefined. Otherwise, the contents of register RT are undefined.

Special Registers Altered:
None

3.3.14.1 Move To/From System Registers [Category: Embedded]

Move to Condition Register from XER X-form

mcrxr BF

31	BF	//	///	///	512	/
0	6	9	11	16	21	31

```
CR4×BF+32:4×BF+35 ← XER32:35
XER32:35 ← 0b0000
```

The contents of $XER_{32:35}$ are copied to Condition Register field BF. $XER_{32:35}$ are set to zero.

Special Registers Altered:
CR field BF $XER_{32:35}$

Move From APID Indirect X-form

mfapidi RT,RA

31	RT	RA	///	275	/
0	6	11	16	21	31

```
RT ← implementation-dependent value based on (RA)
```

The contents of RA are provided to any auxiliary processors that may be present. A value, that is implementation-dependent, is placed in RT.

Special Registers Altered:
None

Programming Note

This instruction is provided as a mechanism for software to query the presence and configuration of one or more auxiliary processors. See the implementation for details on the behavior of this instruction.

Move To Device Control Register User-mode Indexed X-form

mtdcrux RS,RA

31	RS	RA	///	419	/
0	6	11	16	21	31

```
DCRN ← (RA)
DCR(DCRN) ← RS
```

Let the contents of register RA denote a Device Control Register. (The supported Device Control Registers are implementation-dependent.)

The contents of RS are placed into the designated Device Control Register. For 32-bit Device Control Registers, the contents of bits 32:63 of RS are placed into the Device Control Register.

See “Move To Device Control Register Indexed X-form” on page 526 in Book III for more information on this instruction.

Special Registers Altered:
Implementation-dependent

Move From Device Control Register User-mode Indexed X-form

mfdcrux RT,RA

31	RT	RA	///	291	/
0	6	11	16	21	31

```
DCRN ← (RA)
RT ← DCR(DCRN)
```

Let the contents of register RA denote a Device Control Register. (The supported Device Control Registers are implementation-dependent.)

The contents of the designated Device Control Register are placed into RT. For 32-bit Device Control Registers, the contents of bits 32:63 of the designated Device Control Register are placed into RT.

See “Move From Device Control Register Indexed X-form” on page 527 in Book III for more information on this instruction.

Special Registers Altered:
Implementation-dependent

Chapter 4. Floating-Point Processor [Category: Floating-Point]

4.1 Floating-Point Processor Overview

This chapter describes the registers and instructions that make up the Floating-Point Processor facility.

The processor (augmented by appropriate software support, where required) implements a floating-point system compliant with the ANSI/IEEE Standard 754-1985, "IEEE Standard for Binary Floating-Point Arithmetic" (hereafter referred to as "the IEEE standard"). That standard defines certain required "operations" (addition, subtraction, etc.). Herein, the term "floating-point operation" is used to refer to one of these required operations and to additional operations defined (e.g., those performed by *Multiply-Add* or

Reciprocal Estimate instructions). A Non-IEEE mode is also provided. This mode, which may produce results not in strict compliance with the IEEE standard, allows shorter latency.

Instructions are provided to perform arithmetic, rounding, conversion, comparison, and other operations in floating-point registers; to move floating-point data between storage and these registers; and to manipulate the Floating-Point Status and Control Register explicitly.

These instructions are divided into two categories.

- computational instructions

  The computational instructions are those that perform addition, subtraction, multiplication, division, extracting the square root, rounding, conversion, comparison, and combinations of these operations. These instructions provide the floating-point operations. They place status information into the Floating-Point Status and Control Register. They are the instructions described in Sections 4.6.5 through 4.6.7.

- non-computational instructions

  The non-computational instructions are those that perform loads and stores, move the contents of a floating-point register to another floating-point register possibly altering the sign, manipulate the Floating-Point Status and Control Register explicitly, and select the value from one of two floating-point registers based on the value in a third floating-point register. The operations performed by these instructions are not considered floating-point operations. With the exception of the instructions that manipulate the Floating-Point Status and Control Register explicitly, they do not alter the Floating-Point Status and Control Register. They are the instructions described in Sections 4.6.2 through 4.6.4, and 4.6.9.

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 Processor: 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 95 for a description of these exception and enable bits, and Section 4.4, “Floating-Point Exceptions” on page 102 for a detailed discussion of floating-point exceptions, including the effects of the enable bits.

4.2 Floating-Point Processor 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 44 on page 95.

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. Instruction forms with Rc=1 are part of Category: Floating-Point.Record.

Load Double and *Store Double* instructions are provided that transfer 64 bits of data between storage and the FPRs with no conversion. *Load Single* instructions are provided to transfer and convert floating-point values in floating-point single format from storage to the same value in floating-point double format in the FPRs. *Store Single* instructions are provided to transfer and convert floating-point values in floating-point double format from the FPRs to the same value in floating-point single format in storage.

Instructions are provided that manipulate the Floating-Point Status and Control Register and the Condition Register explicitly. Some of these instructions copy data from an FPR to the Floating-Point Status and Control Register or vice versa.

The computational instructions and the *Select* instruction accept values from the FPRs in double format. For single-precision arithmetic instructions, all input values must be representable in single format; if they are not, the result placed into the target FPR, and the setting of status bits in the FPSCR and in the Condition Register (if Rc=1), are undefined.

FPR 0
FPR 1
. . . . . .
FPR 30
FPR 31

0 63

Figure 44. 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.

FPSCR

32 63

Figure 45. Floating-Point Status and Control Register

The bit definitions for the FPSCR are as follows.

Bit(s) Description

32 ***Floating-Point Exception Summary*** (FX)
Every floating-point instruction, except ***mtfsfi*** and ***mtfsf***, implicitly sets $FPSCR_{FX}$ 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 $FPSCR_{FX}$ explicitly.

Programming Note

$FPSCR_{FX}$ is defined not to be altered implicitly by ***mtfsfi*** and ***mtfsf*** because permitting these instructions to alter $FPSCR_{FX}$ implicitly could cause a paradox. An example is an ***mtfsfi*** or ***mtfsf*** instruction that supplies 0 for $FPSCR_{FX}$ and 1 for $FPSCR_{OX}$, and is executed when $FPSCR_{OX}$=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 $FPSCR_{FEX}$ 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 $FPSCR_{VX}$ explicitly.

35 ***Floating-Point Overflow Exception*** (OX)
See Section 4.4.3, “Overflow Exception” on page 105.

36 ***Floating-Point Underflow Exception*** (UX)
See Section 4.4.4, “Underflow Exception” on page 106.

37 ***Floating-Point Zero Divide Exception*** (ZX)
See Section 4.4.2, “Zero Divide Exception” on page 105.

38 ***Floating-Point Inexact Exception*** (XX)
See Section 4.4.5, “Inexact Exception” on page 107.

$FPSCR_{XX}$ is a sticky version of $FPSCR_{FI}$ (see below). Thus the following rules completely describe how $FPSCR_{XX}$ is set by a given instruction.

- If the instruction affects $FPSCR_{FI}$, the new value of $FPSCR_{XX}$ is obtained by ORing the old value of $FPSCR_{XX}$ with the new value of $FPSCR_{FI}$.
- If the instruction does not affect $FPSCR_{FI}$, the value of $FPSCR_{XX}$ is unchanged.

39 ***Floating-Point Invalid Operation Exception (SNaN)*** (VXSNAN)
See Section 4.4.1, "Invalid Operation Exception" on page 104.

40 ***Floating-Point Invalid Operation Exception (∞ - ∞)*** (VXISI)
See Section 4.4.1.

41 ***Floating-Point Invalid Operation Exception (∞÷∞)*** (VXIDI)
See Section 4.4.1.

42 ***Floating-Point Invalid Operation Exception (0÷0)*** (VXZDZ)
See Section 4.4.1.

43 ***Floating-Point Invalid Operation Exception (∞×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 101. 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 $FPSCR_{XX}$, above, regarding the relationship between $FPSCR_{FI}$ and $FPSCR_{XX}$.

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 46 on page 97.

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 46 on page 97. 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 105.

58 ***Floating-Point Underflow Exception Enable*** (UE)
See Section 4.4.4, “Underflow Exception” on page 106.

59 ***Floating-Point Zero Divide Exception Enable*** (ZE)
See Section 4.4.2, “Zero Divide Exception” on page 105.

60 ***Floating-Point Inexact Exception Enable*** (XE)
See Section 4.4.5, “Inexact Exception” on page 107.

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 101.

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 46. 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.

S	EXP	FRACTION
32	33	41 63

Figure 47. Floating-point single format

S	EXP	FRACTION
0	1	12 ... 63

Figure 48. 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 49.

	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 49. IEEE floating-point fields

The architecture requires that the FPRs of the Floating-Point Processor 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 50.

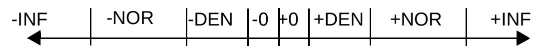


Figure 50. Approximation to real numbers

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:

$$1.2\text{x}10^{-38} \leq M \leq 3.4\text{x}10^{38}$$

Double Format:

$$2.2\text{x}10^{-308} \leq M \leq 1.8\text{x}10^{308}$$

Zero values (± 0)
These are values that have a biased exponent value of zero and a fraction value of zero. Zeros can have a positive or negative sign. The sign of zero is ignored by comparison operations (i.e., comparison regards +0 as equal to -0).

Denormalized numbers (± DEN)
These are values that have a biased exponent value of zero and a nonzero fraction value. They are nonzero numbers smaller in magnitude than the representable normalized numbers. They are values in which the implied unit bit is 0. Denormalized numbers are interpreted as follows:

$$DEN = (-1)^s \times 2^{Emin} \times (0.fraction)$$

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 104.

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 ($FPSCR_{VE}$=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 || 290
         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 107) 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 106) and Underflow Exception is signaled.

4.3.5 Data Handling and Precision

Most of the *Floating-Point Processor 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.

   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.

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 processors, 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 51 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”.

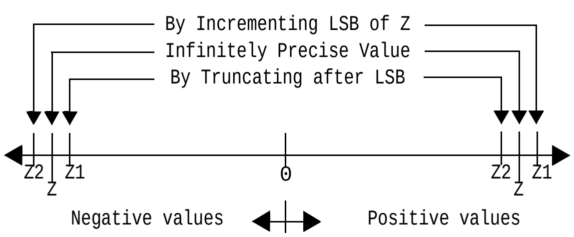


Figure 51. Selection of Z1 and Z2

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 107 for a detailed explanation of rounding.

4.4 Floating-Point Exceptions

This architecture defines the following floating-point exceptions:

- Invalid Operation Exception
  - SNaN
  - Infinity - Infinity
  - Infinity÷Infinity
  - Zero÷Zero
  - Infinity×Zero
  - Invalid Compare
  - Software-Defined Condition
  - Invalid Square Root
  - Invalid Integer Convert
- Zero Divide Exception
- Overflow Exception
- Underflow Exception
- Inexact Exception

These exceptions, other than Invalid Operation Exception due to Software-Defined Condition, may occur during execution of computational instructions. An Invalid Operation Exception due to Software-Defined Condition occurs when a *Move To FPSCR* instruction sets $FPSCR_{VXSOFT}$ 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 103), 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 (∞×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.

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 $FPSCR_{VXSOFT}$ 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 ($FPSCR_{VE}$=1) and an Invalid Operation Exception occurs, the following actions are taken:

1. One or two Invalid Operation Exceptions are set
   - $FPSCR_{VXSNAN}$ (if SNaN)
   - $FPSCR_{VXISI}$ (if $\infty - \infty$)
   - $FPSCR_{VXIDI}$ (if $\infty \div \infty$)
   - $FPSCR_{VXZDZ}$ (if $0 \div 0$)
   - $FPSCR_{VXIMZ}$ (if $\infty \times 0$)
   - $FPSCR_{VXVC}$ (if invalid comp)
   - $FPSCR_{VXSOFT}$ (if sfw-def cond)
   - $FPSCR_{VXSQRT}$ (if invalid sqrt)
   - $FPSCR_{VXCVI}$ (if invalid int cvrt)
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
   - $FPSCR_{FR\ FI}$ are set to zero
   - $FPSCR_{FPRF}$ is unchanged
3. If the operation is a compare,
   - $FPSCR_{FR\ FI\ C}$ are unchanged
   - $FPSCR_{FPCC}$ is set to reflect unordered
4. If an ***mtfsfi***, ***mtfsf***, or ***mtfsb1*** instruction is executed that sets $FPSCR_{VXSOFT}$ to 1,
   - The FPSCR is set as specified in the instruction description.

When Invalid Operation Exception is disabled ($FPSCR_{VE}$=0) and an Invalid Operation Exception occurs, the following actions are taken:

1. One or two Invalid Operation Exceptions are set
   - $FPSCR_{VXSNAN}$ (if SNaN)
   - $FPSCR_{VXISI}$ (if ∞ - ∞)
   - $FPSCR_{VXIDI}$ (if ∞ ÷ ∞)
   - $FPSCR_{VXZDZ}$ (if 0 ÷ 0)
   - $FPSCR_{VXIMZ}$ (if ∞ × 0)
   - $FPSCR_{VXVC}$ (if invalid comp)
   - $FPSCR_{VXSOFT}$ (if sfw-def cond)
   - $FPSCR_{VXSQRT}$ (if invalid sqrt)
   - $FPSCR_{VXCVI}$ (if invalid int cvrt)
2. If the operation is an arithmetic or *Floating Round to Single-Precision* operation,
   - the target FPR is set to a Quiet NaN
   - $FPSCR_{FR\ FI}$ are set to zero
   - $FPSCR_{FPRF}$ is set to indicate the class of the result (Quiet NaN)
3. If the operation is a convert to 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 + ∞, and to the most negative 64-bit integer if the operand in FRB is a negative number, - ∞, or NaN
   - $FPSCR_{FR\ FI}$ are set to zero
   - $FPSCR_{FPRF}$ is undefined
4. If the operation is a convert to 32-bit integer operation,
   - the target FPR is set as follows:
     $FRT_{0:31}$ ← 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
   - $FPSCR_{FR\ FI}$ are set to zero
   - $FPSCR_{FPRF}$ is undefined
5. If the operation is a compare,
   - $FPSCR_{FR\ FI\ C}$ are unchanged
   - $FPSCR_{FPCC}$ is set to reflect unordered
6. If an ***mtfsfi***, ***mtfsf***, or ***mtfsb1*** instruction is executed that sets $FPSCR_{VXSOFT}$ to 1,
   - The FPSCR is set as specified in the instruction description.

4.4.2 Zero Divide Exception

4.4.2.1 Definition

A Zero Divide Exception occurs when a *Divide* instruction is executed with a zero divisor value and a finite nonzero dividend value. It also occurs when a *Reciprocal Estimate* instruction (***fre***[***s***] or ***frsqrte***[***s***]) is executed with an operand value of zero.

4.4.2.2 Action

The action to be taken depends on the setting of the Zero Divide Exception Enable bit of the FPSCR.

When Zero Divide Exception is enabled ($FPSCR_{ZE}$=1) and a Zero Divide Exception occurs, the following actions are taken:

1. Zero Divide Exception is set
   $FPSCR_{ZX}$ ← 1
2. The target FPR is unchanged
3. $FPSCR_{FR\ FI}$ are set to zero
4. $FPSCR_{FPRF}$ is unchanged

When Zero Divide Exception is disabled ($FPSCR_{ZE}$=0) and a Zero Divide Exception occurs, the following actions are taken:

1. Zero Divide Exception is set
   $FPSCR_{ZX}$ ← 1
2. The target FPR is set to ± Infinity, where the sign is determined by the XOR of the signs of the operands
3. $FPSCR_{FR\ FI}$ are set to zero
4. $FPSCR_{FPRF}$ 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 ($FPSCR_{OE}$=1) and an Overflow Exception occurs, the following actions are taken:

1. Overflow Exception is set
   $FPSCR_{OX}$ ← 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. $FPSCR_{FPRF}$ is set to indicate the class and sign of the result (± Normal Number)

When Overflow Exception is disabled ($FPSCR_{OE}$=0) and an Overflow Exception occurs, the following actions are taken:

1. Overflow Exception is set
   $FPSCR_{OX} \leftarrow 1$
2. Inexact Exception is set
   $FPSCR_{XX} \leftarrow 1$
3. The result is determined by the rounding mode ($FPSCR_{RN}$) 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. $FPSCR_{FR}$ is undefined
6. $FPSCR_{FI}$ is set to 1
7. $FPSCR_{FPRF}$ 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 non-zero intermediate result computed as though both the precision and the exponent range were unbounded would be less in magnitude than the smallest normalized number.

If the intermediate result is "Tiny" and Underflow Exception is disabled ($FPSCR_{UE}$=0) then the intermediate result is denormalized (see Section 4.3.4, "Normalization and Denormalization" on page 100) and rounded (see Section 4.3.6, "Rounding" on page 101) before being placed into the target FPR.

"Loss of Accuracy" is detected when the delivered result value differs from what would have been computed were both the precision and the exponent range unbounded.

4.4.4.2 Action

The action to be taken depends on the setting of the Underflow Exception Enable bit of the FPSCR.

When Underflow Exception is enabled ($FPSCR_{UE}$=1) and an Underflow Exception occurs, the following actions are taken:

1. Underflow Exception is set
   $FPSCR_{UX} \leftarrow 1$
2. For double-precision arithmetic instructions, the exponent of the normalized intermediate result is adjusted by adding 1536
3. For single-precision arithmetic instructions and the *Floating Round to Single-Precision* instruction, the exponent of the normalized intermediate result is adjusted by adding 192
4. The adjusted rounded result is placed into the target FPR
5. $FPSCR_{FPRF}$ 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 ($FPSCR_{UE}$=0) and an Underflow Exception occurs, the following actions are taken:

1. Underflow Exception is set
   $FPSCR_{UX} \leftarrow 1$
2. The rounded result is placed into the target FPR
3. $FPSCR_{FPRF}$ 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
   $FPSCR_{XX} \leftarrow 1$
2. The rounded or overflowed result is placed into the target FPR
3. $FPSCR_{FPRF}$ is set to indicate the class and sign of the result

Programming Note

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.

S	C	L	FRACTION	G	R	X
		0	1	53	54	55

Figure 52. 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 53 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 53. Interpretation of G, R, and X bits

Figure 54 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 52.

Format	Guard	Round	Sticky
Double	G bit	R bit	X bit
Single	24	25	OR of 26:52, G, R, X

Figure 54. 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 $FPSCR_{RN}$ as described in Section 4.3.6, “Rounding” on page 101. Using Z1 and Z2 as defined on page 101, the rules for rounding in each mode are as follows.

- ***Round to Nearest***

  Guard bit = 0
  The result is truncated. (Result exact (GRX=000) or closest to next lower value in magnitude (GRX=001, 010, or 011))

  Guard bit = 1
  Depends on Round and Sticky bits:

  Case a
  If the Round or Sticky bit is 1 (inclusive), the result is incremented. (Result closest to next higher value in magnitude (GRX=101, 110, or 111))

  Case b
  If the Round and Sticky bits are 0 (result midway between closest representable values), then if the low-order bit of the result is 1 the result is incremented. Otherwise (the low-order bit of the result is 0) the result is truncated (this is the case of a tie rounded to even).

- ***Round toward Zero***
  Choose the smaller in magnitude of Z1 or Z2. If the Guard, Round, or Sticky bit is nonzero, the result is inexact.

- ***Round toward* + *Infinity***
  Choose Z1.

- ***Round toward* - *Infinity***
  Choose Z2.

If rounding results in a carry into C, the significand is shifted right one position and the exponent is incremented by one. This yields an inexact result, and possibly also exponent overflow. If any of the Guard, Round, or Sticky bits is nonzero, then the result is also inexact. Fraction bits are stored to the target FPR. For *Floating Round to Integer*, *Floating Round to Single-Precision*, and single-precision arithmetic instructions, low-order zeros must be appended as appropriate to fill out the double-precision fraction.

4.5.2 Execution Model for Multiply-Add Type Instructions

The Power ISA provides a special form of instruction that performs up to three operations in one instruction (a multiplication, an addition, and a negation). With this added capability comes the special ability to produce a more exact intermediate result as input to the rounder. 32-bit arithmetic is similar except that the FRACTION field is smaller.

Multiply-add significand arithmetic is considered to be performed with a floating-point accumulator having the following format, where bits 0:106 comprise the significand of the intermediate result.

S	C	L	FRACTION	X'
0	1	2	3	106

Figure 55. 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 56 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 56. 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 Processor Instructions

For each instruction in this section that defines the use of an Rc bit, the behavior defined for the instruction corresponding to Rc=1 is considered part of the Floating-Point.Record category.

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 23.

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 D.9, “Miscellaneous Mnemonics” on page 327.

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 two basic forms of load instruction: single-precision and double-precision. 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 $WORD_{0:31}$ be the floating-point single-precision operand accessed from storage.

Normalized Operand

if $WORD_{1:8} > 0$ and $WORD_{1:8} < 255$ then
  $FRT_{0:1} \leftarrow WORD_{0:1}$
  $FRT_2 \leftarrow \neg WORD_1$
  $FRT_3 \leftarrow \neg WORD_1$
  $FRT_4 \leftarrow \neg WORD_1$
  $FRT_{5:63} \leftarrow WORD_{2:31} \parallel {}^{29}0$

Denormalized Operand

  if $WORD_{1:8} = 0$ and $WORD_{9:31} \neq 0$ then
  sign $\leftarrow WORD_0$
  exp $\leftarrow$ -126
  $frac_{0:52} \leftarrow$ 0b0 $\parallel WORD_{9:31} \parallel {}^{29}0$
  normalize the operand
    do while $frac_0 = 0$
      $frac_{0:52} \leftarrow frac_{1:52} \parallel$ 0b0
      exp $\leftarrow$ exp - 1
  $FRT_0 \leftarrow$ sign
  $FRT_{1:11} \leftarrow$ exp + 1023
  $FRT_{12:63} \leftarrow frac_{1:52}$

Zero / Infinity / NaN

if $WORD_{1:8} = 255$ or $WORD_{1:31} = 0$ then
  $FRT_{0:1} \leftarrow WORD_{0:1}$
  $FRT_2 \leftarrow WORD_1$
  $FRT_3 \leftarrow WORD_1$
  $FRT_4 \leftarrow WORD_1$
  $FRT_{5:63} \leftarrow WORD_{2:31} \parallel {}^{29}0$

For double-precision *Load Floating-Point* instructions 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

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + EXTS(D)
FRT ← DOUBLE(MEM(EA, 4))
```

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

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 111) and placed into register FRT.

Special Registers Altered:
None

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 111) and placed into register FRT.

Special Registers Altered:
None

Load Floating-Point Single with Update D-form

lfsu FRT,D(RA)

49	FRT	RA	D
0	6	11	16 31

```
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 111) 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

31	FRT	RA	RB	567	/
0	6	11	16	21	31

```
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 111) 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)

50	FRT	RA	D
0	6	11	16 31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + EXTS(D)
FRT ← MEM(EA, 8)
```

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

The doubleword in storage addressed by EA is placed into register FRT.

Special Registers Altered:
None

Load Floating-Point Double Indexed X-form

lfdx FRT,RA,RB

31	FRT	RA	RB	599	/
0	6	11	16	21	31

```
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 placed into register FRT.

Special Registers Altered:
None

Load Floating-Point Double with Update D-form

lfdu FRT,D(RA)

51	FRT	RA	D
0	6	11	16 31

```
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 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 with Update Indexed X-form

lfdux FRT,RA,RB

31	FRT	RA	RB	631	/
0	6	11	16	21	31

```
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 placed into register FRT.

EA is placed into register RA.

If RA=0, the instruction form is invalid.

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 117. 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 $FRS_{1:11}$ > 896 or $FRS_{1:63}$ = 0 then
    $WORD_{0:1} \leftarrow FRS_{0:1}$
    $WORD_{2:31} \leftarrow FRS_{5:34}$

Denormalization Required

if 874 ≤ $FRS_{1:11}$ ≤ 896 then
    sign ← FRS_0
    exp ← $FRS_{1:11}$ - 1023
    $frac_{0:52} \leftarrow$ 0b1 || $FRS_{12:63}$
    denormalize operand
        do while exp < -126
            $frac_{0:52} \leftarrow$ 0b0 || $frac_{0:51}$
            exp ← exp + 1
    $WORD_0 \leftarrow$ sign
    $WORD_{1:8} \leftarrow$ 0x00
    $WORD_{9:31} \leftarrow frac_{1:23}$
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).

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)

52	FRS	RA	D
0	6	11	16 31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + EXTS(D)
MEM(EA, 4) ← SINGLE((FRS))
```

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

The contents of register FRS are converted to single format (see page 114) 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)

53	FRS	RA	D
0	6	11	16 31

```
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 114) 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 Indexed X-form

stfsx FRS,RA,RB

31	FRS	RA	RB	663	/
0	6	11	16	21	31

```
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 114) and stored into the word in storage addressed by EA.

Special Registers Altered:
None

Store Floating-Point Single with Update Indexed X-form

stfsux FRS,RA,RB

31	FRS	RA	RB	695	/
0	6	11	16	21	31

```
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 114) 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)

54	FRS	RA	D
0	6	11	16 31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + EXTS(D)
MEM(EA, 8) ← (FRS)
```

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

The contents of register FRS are stored into the doubleword in storage addressed by EA.

Special Registers Altered:
None

Store Floating-Point Double Indexed X-form

stfdx FRS,RA,RB

31	FRS	RA	RB	727	/
0	6	11	16	21	31

```
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 doubleword in storage addressed by EA.

Special Registers Altered:
None

Store Floating-Point Double with Update D-form

stfdu FRS,D(RA)

55	FRS	RA	D
0	6	11	16 31

```
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 doubleword in storage addressed by EA.

EA is placed into register RA.

If RA=0, the instruction form is invalid.

Special Registers Altered:
None

Store Floating-Point Double with Update Indexed X-form

stfdux FRS,RA,RB

31	FRS	RA	RB	759	/
0	6	11	16	21	31

```
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 doubleword in storage addressed by EA.

EA is placed into register RA.

If RA=0, the instruction form is invalid.

Special Registers Altered:
None

Store Floating-Point as Integer Word Indexed *X-form*

stfiwx FRS,RA,RB

31	FRS	RA	RB	983	/
0	6	11	16	21	31

```
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).

The contents of the low-order 32 bits of register FRS are stored, without conversion, into the word in storage addressed by EA.

If the contents of register FRS were produced, either directly or indirectly, by a *Load Floating-Point Single* instruction, a single-precision *Arithmetic* instruction, or ***frsp***, then the value stored is undefined. (The contents of register FRS are produced directly by such an instruction if FRS is the target register for the instruction. The contents of register FRS are produced indirectly by such an instruction if FRS is the final target register of a sequence of one or more *Floating-Point Move* instructions, with the input to the sequence having been produced directly by such an instruction.)

Special Registers Altered:

None

4.6.4 Floating-Point 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***, and ***fnabs***. 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***, and ***fnabs***). 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:
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:
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:
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:
CR1 (if Rc=1)

4.6.5 Floating-Point Arithmetic Instructions

4.6.5.1 Floating-Point Elementary Arithmetic Instructions

Floating Add [Single] — *A-form*

fadd FRT,FRA,FRB (Rc=0)
fadd. FRT,FRA,FRB (Rc=1)

63	FRT	FRA	FRB	///	21	Rc
0	6	11	16	21	26	31

fadds FRT,FRA,FRB (Rc=0)
fadds. FRT,FRA,FRB (Rc=1)

59	FRT	FRA	FRB	///	21	Rc
0	6	11	16	21	26	31

The floating-point operand in register FRA is added to the floating-point operand in register FRB.

If the most significant bit of the resultant significand is not 1, the result is normalized. The result is rounded to the target precision under control of the Floating-Point Rounding Control field RN of the FPSCR and placed into register FRT.

Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.

If a carry occurs, the sum's significand is shifted right one bit position and the exponent is increased by one.

$FPSCR_{FPRF}$ is set to the class and sign of the result, except for Invalid Operation Exceptions when $FPSCR_{VE}$=1.

Special Registers Altered:

FPRF FR FI
FX OX UX XX
VXSNAN VXISI
CR1 (if Rc=1)

Floating Subtract [Single] — *A-form*

fsub FRT,FRA,FRB (Rc=0)
fsub. FRT,FRA,FRB (Rc=1)

63	FRT	FRA	FRB	///	20	Rc
0	6	11	16	21	26	31

fsubs FRT,FRA,FRB (Rc=0)
fsubs. FRT,FRA,FRB (Rc=1)

59	FRT	FRA	FRB	///	20	Rc
0	6	11	16	21	26	31

The floating-point operand in register FRB is subtracted from the floating-point operand in register FRA.

If the most significant bit of the resultant significand is not 1, the result is normalized. The result is rounded to the target precision under control of the Floating-Point Rounding Control field RN of the FPSCR and placed into register FRT.

The execution of the Floating Subtract instruction is identical to that of Floating Add, except that the contents of FRB participate in the operation with the sign bit (bit 0) inverted.

$FPSCR_{FPRF}$ is set to the class and sign of the result, except for Invalid Operation Exceptions when $FPSCR_{VE}$=1.

Special Registers Altered:

FPRF FR FI
FX OX UX XX
VXSNAN VXISI
CR1 (if Rc=1)

Floating Multiply [Single] A-form

fmul FRT,FRA,FRC (Rc=0)
fmul. FRT,FRA,FRC (Rc=1)

63	FRT	FRA	///	FRC	25	Rc
0	6	11	16	21	26	31

fmuls FRT,FRA,FRC (Rc=0)
fmuls. FRT,FRA,FRC (Rc=1)

59	FRT	FRA	///	FRC	25	Rc
0	6	11	16	21	26	31

The floating-point operand in register FRA is multiplied by the floating-point operand in register FRC.

If the most significant bit of the resultant significand is not 1, the result is normalized. The result is rounded to the target precision under control of the Floating-Point Rounding Control field RN of the FPSCR and placed into register FRT.

Floating-point multiplication is based on exponent addition and multiplication of the significands.

$FPSCR_{FPRF}$ is set to the class and sign of the result, except for Invalid Operation Exceptions when $FPSCR_{VE}$=1.

Special Registers Altered:

FPRF FR FI
FX OX UX XX
VXSNAN VXIMZ
CR1 (if Rc=1)

Floating Divide [Single] A-form

fdiv FRT,FRA,FRB (Rc=0)
fdiv. FRT,FRA,FRB (Rc=1)

63	FRT	FRA	FRB	///	18	Rc
0	6	11	16	21	26	31

fdivs FRT,FRA,FRB (Rc=0)
fdivs. FRT,FRA,FRB (Rc=1)

59	FRT	FRA	FRB	///	18	Rc
0	6	11	16	21	26	31

The floating-point operand in register FRA is divided by the floating-point operand in register FRB. The remainder is not supplied as a result.

If the most significant bit of the resultant significand is not 1, the result is normalized. The result is rounded to the target precision under control of the Floating-Point Rounding Control field RN of the FPSCR and placed into register FRT.

Floating-point division is based on exponent subtraction and division of the significands.

$FPSCR_{FPRF}$ is set to the class and sign of the result, except for Invalid Operation Exceptions when $FPSCR_{VE}$=1 and Zero Divide Exceptions when $FPSCR_{ZE}$=1.

Special Registers Altered:

FPRF FR FI
FX OX UX ZX XX
VXSNAN VXIDI VXZDZ
CR1 (if Rc=1)

Floating Square Root [Single] A-form

fsqrt	FRT,FRB	(Rc=0)
fsqrt.	FRT,FRB	(Rc=1)

63	FRT	///	FRB	///	22	Rc
0	6	11	16	21	26	31

fsqrts	FRT,FRB	(Rc=0)
fsqrts.	FRT,FRB	(Rc=1)

59	FRT	///	FRB	///	22	Rc
0	6	11	16	21	26	31

The square root of the floating-point operand in register FRB is placed into register FRT.

If the most significant bit of the resultant significand is not 1, the result is normalized. The result is rounded to the target precision under control of the Floating-Point Rounding Control field RN of the FPSCR and placed into register FRT.

Operation with various special values of the operand is summarized below.

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 $FPSCR_{VE}$ = 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:

FPRF FR FI
FX XX
VXSNAN VXSQRT
CR1 (if Rc=1)

Floating Reciprocal Estimate [Single] A-form

fre FRT,FRB (Rc=0)
[Category:Floating-Point.Phased-In]

fre. FRT,FRB (Rc=1)
[Category: Floating-Point.Record.Phased-In]

63	FRT	///	FRB	///	24	Rc
0	6	11	16	21	26	31

fres	FRT,FRB	(Rc=0)
fres.	FRT,FRB	(Rc=1)

59	FRT	///	FRB	///	24	Rc
0	6	11	16	21	26	31

A estimate of the reciprocal 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 256 of the reciprocal of (FRB), i.e.,

$$ABS\left(\frac{estimate - 1/x}{1/x}\right) \le \frac{1}{256}$$

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 $FPSCR_{ZE}$ = 1.
[2] No result if $FPSCR_{VE}$ = 1.

$FPSCR_{FPRF}$ is set to the class and sign of the result, except for Invalid Operation Exceptions when $FPSCR_{VE}$=1 and Zero Divide Exceptions when $FPSCR_{ZE}$=1.

The results of executing this instruction may vary between implementations, and between different executions on the same implementation.

Special Registers Altered:

FPRF FR (undefined) FI (undefined)
FX OX UX ZX XX (undefined)
VXSNAN
CR1 (if Rc=1)

Floating Reciprocal Square Root Estimate [Single] A-form

frsqrte	FRT,FRB	(Rc=0)

[Category:Floating-Point.Phased-In]

frsqrte.	FRT,FRB	(Rc=1)

[Category:Floating-Point.Record.Phased-In]

63	FRT	///	FRB	///	26	Rc
0	6	11	16	21	26	31

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.,

$$ABS\left(\frac{\text{estimate} - 1/(\sqrt{x})}{1/(\sqrt{x})}\right) \leq \frac{1}{32}$$

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 $FPSCR_{ZE}$ = 1.
[2] No result if $FPSCR_{VE}$ = 1.

$FPSCR_{FPRF}$ is set to the class and sign of the result, except for Invalid Operation Exceptions when $FPSCR_{VE}$=1 and Zero Divide Exceptions when $FPSCR_{ZE}$=1.

The results of executing this instruction may vary between implementations, and between different executions on the same implementation.

Special Registers Altered:

FPRF FR (undefined) FI (undefined)
FX ZX XX (undefined)
VXSNAN VXSQRT
CR1 (if Rc=1)

4.6.5.2 Floating-Point Multiply-Add Instructions

These instructions combine a multiply and an add operation without an intermediate rounding operation. The fraction part of the intermediate product is 106 bits wide (L bit, FRACTION), and all 106 bits take part in the add/subtract portion of the instruction.

Status bits are set as follows.

- Overflow, Underflow, and Inexact Exception bits, the FR and FI bits, and the FPRF field are set based on the final result of the operation, and not on the result of the multiplication.
- Invalid Operation Exception bits are set as if the multiplication and the addition were performed using two separate instructions (***fmul***[***s***], followed by ***fadd***[***s***] or ***fsub***[***s***]). That is, multiplication of infinity by 0 or of anything by an SNaN, and/or addition of an SNaN, cause the corresponding exception bits to be set.

Floating Multiply-Add [Single] A-form

fmadd	FRT,FRA,FRC,FRB	(Rc=0)
fmadd.	FRT,FRA,FRC,FRB	(Rc=1)

63	FRT	FRA	FRB	FRC	29	Rc
0	6	11	16	21	26	31

fmadds	FRT,FRA,FRC,FRB	(Rc=0)
fmadds.	FRT,FRA,FRC,FRB	(Rc=1)

59	FRT	FRA	FRB	FRC	29	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 the Floating-Point Rounding Control field RN of the FPSCR and placed into register FRT.

$FPSCR_{FPRF}$ is set to the class and sign of the result, except for Invalid Operation Exceptions when $FPSCR_{VE}$=1.

Special Registers Altered:

FPRF FR FI
FX OX UX XX
VXSNAN VXISI VXIMZ
CR1 (if Rc=1)

Floating Multiply-Subtract [Single] A-form

fmsub	FRT,FRA,FRC,FRB	(Rc=0)
fmsub.	FRT,FRA,FRC,FRB	(Rc=1)

63	FRT	FRA	FRB	FRC	28	Rc
0	6	11	16	21	26	31

fmsubs	FRT,FRA,FRC,FRB	(Rc=0)
fmsubs.	FRT,FRA,FRC,FRB	(Rc=1)

59	FRT	FRA	FRB	FRC	28	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 the Floating-Point Rounding Control field RN of the FPSCR and placed into register FRT.

$FPSCR_{FPRF}$ is set to the class and sign of the result, except for Invalid Operation Exceptions when $FPSCR_{VE}$=1.

Special Registers Altered:

FPRF FR FI
FX OX UX XX
VXSNAN VXISI VXIMZ
CR1 (if Rc=1)

Floating Negative Multiply-Add [Single] 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

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

$$FRT \leftarrow - ([(FRA)\times(FRC)] + (FRB))$$

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 the Floating-Point Rounding Control field RN of the FPSCR, then negated and placed into register FRT.

This instruction produces the same result as would be obtained by using the *Floating Multiply-Add* instruction and then negating the result, with the following exceptions.

- QNaNs propagate with no effect on their “sign” bit.
- QNaNs that are generated as the result of a disabled Invalid Operation Exception have a “sign” bit of 0.
- SNaNs that are converted to QNaNs as the result of a disabled Invalid Operation Exception retain the “sign” bit of the SNaN.

$FPSCR_{FPRF}$ is set to the class and sign of the result, except for Invalid Operation Exceptions when $FPSCR_{VE}$=1.

Special Registers Altered:

FPRF FR FI
FX OX UX XX
VXSNAN VXISI VXIMZ
CR1 (if Rc=1)

Floating Negative Multiply-Subtract [Single] 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

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

$$FRT \leftarrow - ([(FRA)\times(FRC)] - (FRB))$$

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 the Floating-Point Rounding Control field RN of the FPSCR, then negated and placed into register FRT.

This instruction produces the same result as would be obtained by using the *Floating Multiply-Subtract* instruction and then negating the result, with the following exceptions.

- QNaNs propagate with no effect on their “sign” bit.
- QNaNs that are generated as the result of a disabled Invalid Operation Exception have a “sign” bit of 0.
- SNaNs that are converted to QNaNs as the result of a disabled Invalid Operation Exception retain the “sign” bit of the SNaN.

$FPSCR_{FPRF}$ is set to the class and sign of the result, except for Invalid Operation Exceptions when $FPSCR_{VE}$=1.

Special Registers Altered:

FPRF FR FI
FX OX UX XX
VXSNAN VXISI VXIMZ
CR1 (if Rc=1)

4.6.6 Floating-Point Rounding and Conversion Instructions

Programming Note

Examples of uses of these instructions to perform various conversions can be found in Section E.2, "Floating-Point Conversions [Category: Floating-Point]" on page 334.

4.6.6.1 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 $FPSCR_{RN}$, and placed into register FRT.

The rounding is described fully in Section A.1, "Floating-Point Round to Single-Precision Model" on page 299.

$FPSCR_{FPRF}$ is set to the class and sign of the result, except for Invalid Operation Exceptions when $FPSCR_{VE}$=1.

Special Registers Altered:

FPRF FR FI
FX OX UX XX
VXSNAN
CR1 (if Rc=1)

4.6.6.2 Floating-Point Convert To/From Integer Instructions

Floating Convert To Integer Doubleword X-form

fctid	FRT,FRB	(Rc=0)
fctid.	FRT,FRB	(Rc=1)

63	FRT	///	FRB	814	Rc
0	6	11	16	21	31

The floating-point operand in register FRB is converted to a 64-bit signed fixed-point integer, using the rounding mode specified by $FPSCR_{RN}$, and placed into register FRT.

If the operand in FRB is greater than $2^{63} - 1$, then FRT is set to 0x7FFF_FFFF_FFFF_FFFF. If the operand in FRB is less than -2^{63}, then FRT is set to 0x8000_0000_0000_0000.

The conversion is described fully in Section A.2, "Floating-Point Convert to Integer Model" on page 303.

Except for enabled Invalid Operation Exceptions, $FPSCR_{FPRF}$ is undefined. $FPSCR_{FR}$ is set if the result is incremented when rounded. $FPSCR_{FI}$ is set if the result is inexact.

Special Registers Altered:

FPRF (undefined) FR FI
FX XX
VXSNAN VXCVI
CR1 (if Rc=1)

Floating Convert To Integer Doubleword with round toward Zero X-form

fctidz FRT,FRB (Rc=0)
fctidz. FRT,FRB (Rc=1)

63	FRT	///	FRB	815	Rc
0	6	11	16	21	31

The floating-point operand in register FRB is converted to a 64-bit signed fixed-point integer, using the rounding mode Round toward Zero, and placed into register FRT.

If the operand in FRB is greater than 2^{63} - 1, then FRT is set to 0x7FFF_FFFF_FFFF_FFFF. If the operand in FRB is less than -2^{63}, then FRT is set to 0x8000_0000_0000_0000.

The conversion is described fully in Section A.2, "Floating-Point Convert to Integer Model" on page 303.

Except for enabled Invalid Operation Exceptions, $FPSCR_{FPRF}$ is undefined. $FPSCR_{FR}$ is set if the result is incremented when rounded. $FPSCR_{FI}$ is set if the result is inexact.

Special Registers Altered:

FPRF (undefined) FR FI
FX XX
VXSNAN VXCVI
CR1 (if Rc=1)

Floating Convert To Integer Word X-form

fctiw FRT,FRB (Rc=0)
fctiw. FRT,FRB (Rc=1)

63	FRT	///	FRB	14	Rc
0	6	11	16	21	31

The floating-point operand in register FRB is converted to a 32-bit signed fixed-point integer, using the rounding mode specified by $FPSCR_{RN}$, and placed into $FRT_{32:63}$. The contents of $FRT_{0:31}$ are undefined.

If the operand in FRB is greater than 2^{31} - 1, then bits 32:63 of FRT are set to 0x7FFF_FFFF. If the operand in FRB is less than -2^{31}, then bits 32:63 of FRT are set to 0x8000_0000.

The conversion is described fully in Section A.2, "Floating-Point Convert to Integer Model" on page 303.

Except for enabled Invalid Operation Exceptions, $FPSCR_{FPRF}$ is undefined. $FPSCR_{FR}$ is set if the result is incremented when rounded. $FPSCR_{FI}$ is set if the result is inexact.

Special Registers Altered:

FPRF (undefined) FR FI
FX XX
VXSNAN VXCVI
CR1 (if Rc=1)

Floating Convert To Integer Word with round toward Zero *X-form*

fctiwz	FRT,FRB	(Rc=0)
fctiwz.	FRT,FRB	(Rc=1)

63	FRT	///	FRB	15	Rc
0	6	11	16	21	31

The floating-point operand in register FRB is converted to a 32-bit signed fixed-point integer, using the rounding mode Round toward Zero, and placed into $FRT_{32:63}$. The contents of $FRT_{0:31}$ are undefined.

If the operand in FRB is greater than 2^{31} - 1, then bits 32:63 of FRT are set to 0x7FFF_FFFF. If the operand in FRB is less than -2^{31}, then bits 32:63 of FRT are set to 0x8000_0000.

The conversion is described fully in Section A.2, "Floating-Point Convert to Integer Model".

Except for enabled Invalid Operation Exceptions, $FPSCR_{FPRF}$ is undefined. $FPSCR_{FR}$ is set if the result is incremented when rounded. $FPSCR_{FI}$ is set if the result is inexact.

Special Registers Altered:

FPRF (undefined) FR FI
FX XX
VXSNAN VXCVI
CR1 (if Rc=1)

Floating Convert From Integer Doubleword *X-form*

fcfid	FRT,FRB	(Rc=0)
fcfid.	FRT,FRB	(Rc=1)

63	FRT	///	FRB	846	Rc
0	6	11	16	21	31

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 $FPSCR_{RN}$, and placed into register FRT.

The conversion is described fully in Section A.3, "Floating-Point Convert from Integer Model".

$FPSCR_{FPRF}$ is set to the class and sign of the result. $FPSCR_{FR}$ is set if the result is incremented when rounded. $FPSCR_{FI}$ is set if the result is inexact.

Special Registers Altered:

FPRF FR FI
FX XX
CR1 (if Rc=1)

4.6.6.3 Floating Round to Integer Instructions [Category: Floating-Point.Phased-In]

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 307.

Programming Note

These instructions set $FPSCR_{FR\ FI}$ to 0b00 regardless of whether the result is inexact or rounded because there is a desire to preserve the value of $FPSCR_{XX}$. Furthermore, it is believed that most programs do not need to know whether these rounding operations produce inexact or rounded results. If it is necessary to determine whether the result is inexact or rounded, software must compare the result with the original source operand.

Floating Round to Integer Nearest X-form

frin FRT,FRB (Rc=0)
frin. FRT,FRB (Rc=1)

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.

$FPSCR_{FPRF}$ is set to the class and sign of the result, except for Invalid Operation Exceptions when $FPSCR_{VE}$ = 1.

Special Registers Altered:

FPRF FR (set to 0) FI (set to 0)
FX
VXSNAN
CR1 (if Rc = 1)

Floating Round to Integer Toward Zero X-form

friz FRT,FRB (Rc=0)
friz. FRT,FRB (Rc=1)

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.

$FPSCR_{FPRF}$ is set to the class and sign of the result, except for Invalid Operation Exceptions when $FPSCR_{VE}$ = 1.

Special Registers Altered:

FPRF FR (set to 0) FI (set to 0)
FX
VXSNAN
CR1 (if Rc = 1)

Floating Round to Integer Plus X-form

frip FRT,FRB (Rc=0)
frip. FRT,FRB (Rc=1)

63	FRT	///	FRB	456	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 +infinity, and the result is placed into register FRT.

$FPSCR_{FPRF}$ is set to the class and sign of the result, except for Invalid Operation Exceptions when $FPSCR_{VE}$ = 1.

Special Registers Altered:

FPRF FR (set to 0) FI (set to 0)
FX
VXSNAN
CR1 (if Rc = 1)

Floating Round to Integer Minus X-form

frim FRT,FRB (Rc=0)
frim. FRT,FRB (Rc=1)

63	FRT	///	FRB	488	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 -infinity, and the result is placed into register FRT.

$FPSCR_{FPRF}$ is set to the class and sign of the result, except for Invalid Operation Exceptions when $FPSCR_{VE}$ = 1.

Special Registers Altered:

FPRF FR (set to 0) FI (set to 0)
FX
VXSNAN
CR1 (if Rc = 1)

4.6.7 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:

CR field BF
FPCC
FX
VXSNAN

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:

CR field BF
FPCC
FX
VXSNAN VXVC

4.6.8 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:
CR1 (if Rc=1)

Programming Note

Examples of uses of this instruction can be found in Sections E.2, “Floating-Point Conversions [Category: Floating-Point]” on page 334 and E.3, “Floating-Point Selection [Category: Floating-Point]” on page 336.

Warning: Care must be taken in using ***fsel*** if IEEE compatibility is required, or if the values being tested can be NaNs or infinities; see Section E.3.4, “Notes” on page 336.

4.6.9 Floating-Point Status and Control Register Instructions

Every *Floating-Point Status and Control Register* instruction synchronizes the effects of all floating-point instructions executed by a given processor. Executing a *Floating-Point Status and Control Register* instruction ensures that all floating-point instructions previously initiated by the given processor have completed before the *Floating-Point Status and Control Register* instruction is initiated, and that no subsequent floating-point instructions are initiated by the given processor until the *Floating-Point Status and Control Register* instruction has completed. In particular:

- All exceptions that will be caused by the previously initiated instructions are recorded in the FPSCR before the Floating-Point Status and Control Register instruction is initiated.
- All invocations of the system floating-point enabled exception error handler that will be caused by the previously initiated instructions have occurred before the Floating-Point Status and Control Register instruction is initiated.
- No subsequent floating-point instruction that depends on or alters the settings of any FPSCR bits is initiated until the Floating-Point Status and Control Register instruction has completed.

(Floating-point Storage Access instructions are not affected.)

Move From FPSCR X-form

mffs FRT (Rc=0)
mffs. FRT (Rc=1)

63	FRT	///	///	583	Rc
0	6	11	16	21	31

The contents of the FPSCR are placed into $FRT_{32:63}$. The contents of $FRT_{0:31}$ are undefined.

Special Registers Altered:

CR1 (if Rc=1)

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 field BFA are copied to Condition Register field BF. All exception bits copied are set to 0 in the FPSCR. If the FX bit is copied, it is set to 0 in the FPSCR.

Special Registers Altered:

CR field BF
FX OX (if BFA=0)
UX ZX XX VXSNAN (if BFA=1)
VXISI VXIDI VXZDZ VXIMZ (if BFA=2)
VXVC (if BFA=3)
VXSOFT VXSQRT VXCVI (if BFA=5)

Move To FPSCR Field Immediate X-form

mtfsfi BF,U (Rc=0)
mtfsfi. BF,U (Rc=1)

63	BF	//	///	U	/	134	Rc
0	6	9	11	16	20	21	31

The value of the U field is placed into FPSCR field BF.

$FPSCR_{FX}$ is altered only if BF=0.

Special Registers Altered:

FPSCR field BF
CR1 (if Rc=1)

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 95, and not from $U_{1:2}$.

Move To FPSCR Fields XFL-form

mtfsf FLM,FRB (Rc=0)
mtfsf. FLM,FRB (Rc=1)

63	/	FLM	/	FRB	711	Rc
0	6	7	15	16	21	31

The contents of bits 32:63 of register FRB are placed into the FPSCR under control of the field mask specified by FLM. The field mask identifies the 4-bit fields affected. Let i be an integer in the range 0-7. If FLM_i=1 then FPSCR field i (FPSCR 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 register FRB.

$FPSCR_{FX}$ is altered only if FLM_0 = 1.

Special Registers Altered:

FPSCR fields selected by mask
CR1 (if Rc=1)

Programming Note

Updating fewer than all eight fields of the FPSCR may have substantially poorer performance on some implementations than updating all the fields.

Programming Note

When $FPSCR_{32:35}$ 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 95, and not from $(FRB)_{33:34}$.

Move To FPSCR Bit 0 *X-form*

mtfsb0 BT (Rc=0)
mtfsb0. BT (Rc=1)

63	BT	///	///	70	Rc
0	6	11	16	21	31

Bit BT+32 of the FPSCR is set to 0.

Special Registers Altered:
FPSCR bit BT+32
CR1 (if Rc=1)

Programming Note

Bits 33 and 34 (FEX and VX) cannot be explicitly reset.

Move To FPSCR Bit 1 *X-form*

mtfsb1 BT (Rc=0)
mtfsb1. BT (Rc=1)

63	BT	///	///	38	Rc
0	6	11	16	21	31

Bit BT+32 of the FPSCR is set to 1.

Special Registers Altered:
FPSCR bits BT+32 and FX
CR1 (if Rc=1)

Programming Note

Bits 32 and 34 (FEX and VX) cannot be explicitly set.

Chapter 5. Vector Processor [Category: Vector]

5.1 Vector Processor Overview

This chapter describes the registers and instructions that make up the Vector Processor facility.

5.2 Chapter Conventions

5.2.1 Description of Instruction Operation

The following notation, in addition to that described in Section 1.3.2, is used in this chapter. Additional RTL functions are described in Appendix B.

Notation **Meaning**

x ? y : z — if the value of x is true, then the value of y, otherwise the value z.

$+_{int}$ — Integer addition.

$+_{fp}$ — Floating-point addition.

$-_{fp}$ — Floating-point subtraction.

$\times_{sui}$ — Multiplication of a signed-integer (first operand) by an unsigned-integer (second operand).

$\times_{fp}$ — Floating-point multiplication.

$=_{int}$ — Integer equals relation.

$=_{fp}$ — Floating-point equals relation.

$<_{ui}, \leq_{ui}, >_{ui}, \geq_{ui}$
Unsigned-integer comparison relations.

$<_{si}, \leq_{si}, >_{si}, \geq_{si}$
Signed-integer comparison relations.

$<_{fp}, \leq_{fp}, >_{fp}, \geq_{fp}$
Floating-point comparison relations.

LENGTH(x) — Length of x, in bits. If x is the word “element”, LENGTH(x) is the length, in bits, of the element implied by the instruction mnemonic.

x << y — Result of shifting x left by y bits, filling vacated bits with zeros.

b ← LENGTH(x)
result ← (y < b) ? ($x_{y:b-1}$ || $^{y}0$) : $^{b}0$

$x >>_{ui} y$ — Result of shifting x right by y bits, filling vacated bits with zeros.

b ← LENGTH(x)
result ← (y < b) ? ($^{y}0$ || $x_{0:(b-y)-1}$) : $^{b}0$

x >> y — Result of shifting x right by y bits, filling vacated bits with copies of bit 0 (sign bit) of x.

b ← LENGTH(x)
result ← (y<b) ? ($^{y}x_0$ || $x_{0:(b-y)-1}$) : $^{b}x_0$

x <<< y — Result of rotating x left by y bits.

b ← LENGTH(x)
result ← $x_{y:b-1}$ || $x_{0:y-1}$

Chop(x, y) — Result of extending the right-most y bits of x on the left with zeros.

result ← x & ((1<<y)-1)

EXTZ(x) — Result of extending x on the left with zeros.

b ← LENGTH(x)
result ← x & ((1<<b)-1)

Clamp(x, y, z)
x is interpreted as a signed integer. If the value of x is less than y, then the value y is returned, else if the value of x is greater than z, the value z is returned, else the value x is returned.

if (x < y) then
  result ← y
  $VSCR_{SAT}$ ← 1
else if (x > z) then
  result ← z
  $VSCR_{SAT}$ ← 1
else result ← x

RoundToSPIntCeil(x)
The value x if x is a single-precision floating-point integer; otherwise the smallest single-precision floating-point integer that is greater than x.

RoundToSPIntFloor(x)
The value x if x is a single-precision floating-point integer; otherwise the largest single-precision floating-point integer that is less than x.

RoundToSPIntNear(x)
The value x if x is a single-precision floating-point integer; otherwise the single-precision floating-point integer that is nearest in value to x (in case of a tie, the even single-precision floating-point integer is used).

RoundToSPIntTrunc(x)
The value x if x is a single-precision floating-point integer; otherwise the largest single-precision floating-point integer that is less than x if x>0, or the smallest single-precision floating-point integer that is greater than x if x<0.

RoundToNearSP(x)
The single-precision floating-point number that is nearest in value to the infinitely-precise floating-point intermediate result x (in case of a tie, the single-precision floating-point value with the least-significant bit equal to 0 is used).

ReciprocalEstimateSP(x)
A single-precision floating-point estimate of the reciprocal of the single-precision floating-point number x.

ReciprocalSquareRootEstimateSP(x)
A single-precision floating-point estimate of the reciprocal of the square root of the single-precision floating-point number x.

LogBase2EstimateSP(x)
A single-precision floating-point estimate of the base 2 logarithm of the single-precision floating-point number x.

Power2EstimateSP(x)
A single-precision floating-point estimate of the 2 raised to the power of the single-precision floating-point number x.

Quadword															
Word 0				Word 1				Word 2				Word 3			
Halfword 0		Halfword 1		Halfword 2		Halfword 3		Halfword 4		Halfword 5		Halfword 6		Halfword 7	
B0	B1	B2	B3	B4	B5	B6	B7	B8	B9	B10	B11	B12	B13	B14	B15
0	8	16	24	32	40	48	56	64	72	80	88	96	104	112	120 127

Figure 57. Vector Register elements

5.3 Vector Processor Registers

5.3.1 Vector Registers

There are 32 Vector Registers (VRs), each containing 128 bits. See Figure 58. All computations and other data manipulation are performed on data residing in Vector Registers, and results are placed into a VR.

VR0
VR1
...
...
VR30
VR31
0   127

Figure 58. Vector Registers

Depending on the instruction, the contents of a Vector Register are interpreted as a sequence of equal-length elements (bytes, halfwords, or words) or as a quadword. Each of the elements is aligned at its natural boundary within the Vector Register, as shown in Figure 57. Many instructions perform a given operation in parallel on all elements in a Vector Register. Depending on the instruction, a byte, halfword, or word element can be interpreted as a signed-integer, an unsigned-integer, or a logical value; a word element can also be interpreted as a single-precision floating-point value. In the instruction descriptions, phrases like "signed-integer word element" are used as shorthand for "word element, interpreted as a signed-integer".

Load and *Store* instructions are provided that transfer a byte, halfword, word, or quadword between storage and a Vector Register.

5.3.2 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 vector register. When moved to or from a vector register, the 32-bit VSCR is right justified in the 128-bit vector register. When moved to a vector register, bits 0-95 of the vector register are cleared (set to 0).

VSCR
96   127

Figure 59. Vector Status and Control Register

The bit definitions for the VSCR are as follows.

Bit(s) Description

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 5.6.1.

1 If an element in a source VR contains a denormalized value, the value 0 is used instead. If an instruction causes an Underflow Exception, the corresponding element in the target VR is set to 0. In both cases the 0 has the same sign as the denormalized or underflowing value.

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 5.8. ***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 vector register will be architecturally precise. That is, it will reflect all updates to the SAT bit that could have been made by vector instructions logically preceding it in the program flow, and further, it will not reflect any SAT updates that may be made to it by vector instructions logically following it in the program flow. To implement this, processors may choose to make the ***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 5.5.1, 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***.

5.3.3 VR Save Register

The VR Save Register (VRSAVE) is a 32-bit register provided for application and operating system use.

VRSAVE
32   63

Figure 60. VR Save Register

Programming Note

The VRSAVE register can be used to indicate which VRs are currently being used by a program. If this is done, the operating system could save only those VRs when an “interrupt” occurs (see Book III), and could restore only those VRs when resuming the interrupted program.

If this approach is taken it must be applied rigorously; if a program fails to indicate that a given VR 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 vector registers.

5.4 Vector Storage Access Operations

The *Vector Storage Access* instructions provide the means by which data can be copied from storage to a Vector Register or from a Vector Register to storage. Instructions are provided that access byte, halfword, word, and quadword storage operands. These instructions differ from the fixed-point and floating-point *Storage Access* instructions in that vector storage operands are assumed to be aligned, and vector storage accesses are performed as if the appropriate number of low-order bits of the specified effective address (EA) were zero. For example, the low-order bit of EA is ignored for halfword *Vector Storage Access* instructions, and the low-order four bits of EA are ignored for quadword *Vector Storage Access* instructions. The effect is to load or store the storage operand of the specified length that contains the byte addressed by EA.

If a storage operand is unaligned, additional instructions must be used to ensure that the operand is correctly placed in a Vector Register or in storage. Instructions are provided that shift and merge the contents of two Vector Registers, such that an unaligned quadword storage operand can be copied between storage and the Vector Registers in a relatively efficient manner.

As shown in Figure 57, the elements in Vector Registers are numbered; the high-order (or most significant) byte element is numbered 0 and the low-order (or least significant) byte element is numbered 15. The numbering affects the values that must be placed into the permute control vector for the *Vector Permute* instruction in order for that instruction to achieve the desired effects, as illustrated by the examples in the following subsections.

A vector quadword *Load* instruction for which the effective address (EA) is quadword-aligned places the byte in storage addressed by EA into byte element 0 of the target Vector Register, the byte in storage addressed by EA+1 into byte element 1 of the target Vector Register, etc. Similarly, a vector quadword *Store* instruction for which the EA is quadword-aligned places the contents of byte element 0 of the source Vector Register into the byte in storage addressed by EA, the contents of byte element 1 of the source Vector Register into the byte in storage addressed by EA+1, etc.

Figure 61 shows an aligned quadword in storage. Figure 62 shows the result of loading that quadword into a Vector Register or, equivalently, shows the contents that must be in a Vector Register if storing that Vector Register is to produce the storage contents shown in Figure 61.

When an aligned byte, halfword, or word storage operand is loaded into a Vector Register, the element (byte, halfword, or word respectively) that receives the data is the element that would have received the data had the entire aligned quadword containing the storage operand addressed by EA been loaded. Similarly, when a byte, halfword, or word element in a Vector Register is stored into an aligned storage operand (byte, halfword, or word respectively), the element selected to be stored is the element that would have been stored into the storage operand addressed by EA had the entire Vector Register been stored to the aligned quadword containing the storage operand addressed by EA. (Byte storage operands are always aligned.)

For aligned byte, halfword, and word storage operands, if the corresponding element number is known when the program is written, the appropriate *Vector Splat* and *Vector Permute* instructions can be used to copy or replicate the data contained in the storage operand after loading the operand into a Vector Register. An example of this is given in the Programming Note for *Vector Splat*; see page 156. Another example is to replicate the element across an entire Vector Register before storing it into an arbitrary aligned storage operand of the same length; the replication ensures that the correct data are stored regardless of the offset of the storage operand in its aligned quadword in storage.

00	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
10																
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

Figure 61. Aligned quadword storage operand

00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

Figure 62. Vector Register contents for aligned quadword Load or Store

00												00	01	02	03	04
10	05	06	07	08	09	0A	0B	0C	0D	0E	0F					
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

Figure 63. Unaligned quadword storage operand

Vhi												00	01	02	03	04
Vlo	05	06	07	08	09	0A	0B	0C	0D	0E	0F					
Vt,Vs	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
	0															15

Figure 64. Vector Register contents

5.4.1 Accessing Unaligned Storage Operands

Figure 63 shows an unaligned quadword storage operand that spans two aligned quadwords. In the remainder of this section, the aligned quadword that contains the most significant bytes of the unaligned quadword is called the most significant quadword (MSQ) and the aligned quadword that contains the least significant bytes of the unaligned quadword is called the least significant quadword (LSQ). Because the *Vector Storage Access* instructions ignore the low-order bits of the effective address, the unaligned quadword cannot be transferred between storage and a Vector Register using a single instruction. The remainder of this section gives examples of accessing unaligned quadword storage operands. Similar sequences can be used to access unaligned halfword and word storage operands.

Programming Note

The sequence of instructions given below is one approach that can be used to load the unaligned quadword shown in Figure 63 into a Vector Register. In Figure 64 Vhi and Vlo are the Vector Registers that will receive the most significant quadword and least significant quadword respectively. VRT is the target Vector Register.

After the two quadwords have been loaded into Vhi and Vlo, using *Load Vector Indexed* instructions, the alignment is performed by shifting the 32-byte quantity Vhi || Vlo left by an amount determined by the address of the first byte of the desired data. The shifting is done using a *Vector Permute* instruction for which the permute control vector is generated by a *Load Vector for Shift Left* instruction. The *Load Vector* for *Shift Left* instruction uses the same address specification as the *Load Vector Indexed* instruction that loads the Vhi register; this is the address of the desired unaligned quadword.

The following sequence of instructions copies the unaligned quadword storage operand into register Vt.

```
# Assumptions:
# Rb != 0 and contents of Rb = 0xB
  lvx       Vhi,0,Rb      # load MSQ
  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 64 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
```

Vt	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
	0															15

Figure 65. Vector Register contents after Vector OR

5.5 Vector Integer Operations

Many of the instructions that produce fixed-point integer results have the potential to compute a result value that cannot be represented in the target format. When this occurs, this unrepresentable intermediate value is converted to a representable result value using one of the following methods.

1. The high-order bits of the intermediate result that do not fit in the target format are discarded. This method is used by instructions having names that include the word "Modulo".
2. The intermediate result is converted to the nearest value that is representable in the target format (i.e., to the minimum or maximum representable value, as appropriate). This method is used by instructions having names that include the word "Saturate". An intermediate result that is forced to the minimum or maximum representable value as just described is said to "saturate".

   An instruction for which an intermediate result saturates causes $VSCR_{SAT}$ to be set to 1; see Section 5.3.2.
3. If the intermediate result includes non-zero fraction bits it is rounded up to the nearest fixed-point integer value. This method is used by the six *Vector Average Integer* instructions and by the *Vector Multiply-High-Round-Add Signed Halfword Saturate* instruction. The latter instruction then uses method 2, if necessary.

Programming Note

Because $VSCR_{SAT}$ is sticky, it can be used to detect whether any instruction in a sequence of “Saturate”-type instructions produced an inexact result due to saturation. For example, the contents of the VSCR can be copied to a VR (***mfvscr***), bits other than the SAT bit can be cleared in the VR (***vand*** with a constant), the result can be compared to zero setting CR6 (***vcmpequb***.), and a branch can be taken according to whether $VSCR_{SAT}$ was set to 1 (*Branch Conditional* that tests CR field 6).

Testing $VSCR_{SAT}$ after each “Saturate”-type instruction would degrade performance considerably. Alternative techniques include the following:

- Retain sufficient information at "checkpoints" that the sequence of computations performed between one checkpoint and the next can be redone (more slowly) in a manner that detects exactly when saturation occurs. Test $VSCR_{SAT}$ only at checkpoints, or when redoing a sequence of computations that saturated.
- Perform intermediate computations using an element length sufficient to prevent saturation, and then use a *Vector Pack Integer Saturate* instruction to pack the final result to the desired length. (Vector Pack Integer Saturate causes results to saturate if necessary, and sets $VSCR_{SAT}$ to 1 if any result saturates.)

5.5.1 Integer Saturation

Saturation occurs whenever the result of a saturating instruction does not fit in the result field. Unsigned saturation clamps results to zero (0) on underflow and to the maximum positive integer value (2^{n-1}, e.g. 255 for byte fields) on overflow. Signed saturation clamps results to the smallest representable negative number (-2^{n-1}, e.g. -128 for byte fields) on underflow, and to the largest representable positive number ($2^{n-1}-1$, e.g. +127 for byte fields) on overflow.

In most cases, the simple maximum/minimum saturation performed by the vector instructions is adequate. However, sometimes, e.g. in the creation of very high quality images, more complex saturation functions must be applied. To support this, the Vector facility provides a mechanism for detecting that saturation has occurred. The VSCR has a bit, the SAT bit, which is set to a one (1) anytime any field in a saturating instruction saturates. The SAT bit can only be cleared by explicitly writing zero to it. Thus SAT accumulates a summary result of any integer overflow or underflow that occurs on a saturating instruction.

Borderline cases that generate results equal to saturation values, for example unsigned 0+0=0 and unsigned byte 1+254=255, are not considered saturation conditions and do not cause SAT to be set.

The SAT bit can be set by the following types of instructions:

- Move To VSCR
- Vector Add Integer with Saturation
- Vector Subtract Integer with Saturation
- Vector Multiply-Add Integer with Saturation
- Vector Multiply-Sum with Saturation
- Vector Sum-Across with Saturation
- Vector Pack with Saturation
- Vector Convert to Fixed-point with Saturation

Note that only instructions that explicitly call for "saturation" can set SAT. "Modulo" integer instructions and floating-point arithmetic instructions never set SAT.

Programming Note

The SAT state can be tested and used to alter program flow by moving the VSCR to a vector register (with ***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).

5.6 Vector Floating-Point Operations

5.6.1 Floating-Point Overview

Unless $VSCR_{NJ}$=1 (see Section 5.3.2), the floating-point model provided by the Vector Processor conforms to The Java Language Specification (hereafter referred to as "Java"), which is a subset of the default environment specified by the IEEE standard (i.e., by ANSI/IEEE Standard 754-1985, "IEEE Standard for Binary Floating-Point Arithmetic"). For aspects of floating-point behavior that are not defined by Java but are defined by the IEEE standard, vector floating-point conforms to the IEEE standard. For aspects of floating-point behavior that are defined neither by Java nor by the IEEE standard but are defined by the "C9X Floating-Point Proposal" (hereafter referred to as "C9X"), vector floating-point conforms to C9X.

The single-precision floating-point data format, value representations, and computational models defined in Chapter 4. "Floating-Point Processor [Category: Floating-Point]" on page 93 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 $VSCR_{SAT}$ 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 5.6.2) 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 Processor, and cannot be emulated satisfactorily using the functions that are supported by the Vector Processor, the functions provided by the Floating-Point Processor should be used; see Chapter 4.

5.6.2 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 $VSCR_{NJ}$=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 $VSCR_{NJ}$=1.
- Exceptions that can be caused by a nonzero source value cannot be caused by a denormalized source value when $VSCR_{NJ}$=1.

5.6.2.1 NaN Operand Exception

A NaN Operand Exception occurs when a source value for any of the following instructions is a NaN.

- A vector instruction that would normally produce floating-point results
- Either of the two *Vector Convert To Fixed-Point Word* instructions
- Any of the four *Vector Floating-Point Compare* instructions

The following actions are taken:

If the vector instruction would normally produce floating-point results, the corresponding result is a source NaN selected as follows. In all cases, if the selected source NaN is a Signaling NaN it is converted to the corresponding Quiet NaN (by setting the high-order bit of the fraction field to 1) before being placed into the target element.

```
if the element in VRA is a NaN
  then the result is that NaN
  else if the element in VRB is a NaN
       then the result is that NaN
       else if the element in VRC is a NaN
              then the result is that NaN
              else if Invalid Operation exception
                (Section 5.6.2.2)
              then the result is the QNaN 0x7FC0_0000
```

If the instruction is either of the two *Vector Convert To Fixed-Point Word* instructions, the corresponding result is 0x0000_0000. $VSCR_{SAT}$ is not affected.

If the instruction is *Vector Compare Bounds Floating-Point*, the corresponding result is 0xC000_0000.

If the instruction is one of the other *Vector Floating-Point Compare* instructions, the corresponding result is 0x0000_0000.

5.6.2.2 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.

5.6.2.3 Zero Divide Exception

A Zero Divide Exception occurs when a V*ector 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.

5.6.2.4 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.

5.6.2.5 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. $VSCR_{SAT}$ is set to 1.

3. If the instruction is *Vector Convert To Signed Fixed-Point Word Saturate*, the corresponding result is 0x7FFF_FFFF if the source value is a positive number or +infinity., and is 0x8000_0000 if the source value is a negative number or -infinity. $VSCR_{SAT}$ is set to 1.

5.6.2.6 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 $VSCR_{NJ}$=0, the corresponding result is the value produced by denormalizing and rounding the intermediate result.
2. If $VSCR_{NJ}$=1, the corresponding result is a zero, where the sign is the sign of the intermediate result.

5.7 Vector 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 23. The low-order bits of the EA that would correspond to an unaligned storage operand are ignored.

The *Load Vector Element Indexed* and *Store Vector Element Indexed* instructions transfer a byte, halfword, or word element between storage and a Vector Register. The *Load Vector Indexed* and *Store Vector Indexed* instructions transfer an aligned quadword between storage and a Vector Register.

5.7.1 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.

5.7.2 Vector Load Instructions

The aligned byte, halfword, word, or quadword in storage addressed by EA is loaded into register VRT.

Programming Note

The *Load Vector Element* instructions load the specified element into the same location in the target register as the location into which it would be loaded using the *Load Vector* instruction.

Load Vector Element Byte Indexed X-form

lvebx VRT,RA,RB

31	VRT	RA	RB	7	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA  ← b + (RB)
eb  ← EA60:63

VRT ← undefined
if Big-Endian byte ordering then
   VRT8×eb:8×eb+7 ← MEM(EA,1)
else
   VRT120-(8×eb):127-(8×eb) ← MEM(EA,1)
```

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

Let eb be bits 60:63 of EA.

If Big-Endian byte ordering is used for the storage access, the contents of the byte in storage at address EA are placed into byte eb of register VRT. The remaining bytes in register VRT are set to undefined values.

If Category: Vector.Little-Endian is supported, then if Little-Endian byte ordering is used for the storage access, the contents of the byte in storage at address EA are placed into byte 15-eb of register VRT. The remaining bytes in register VRT are set to undefined values.

Special Registers Altered:
None

Load Vector Element Halfword Indexed X-form

lvehx VRT,RA,RB

31	VRT	RA	RB	39	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA  ← (b + (RB)) & 0xFFFF_FFFF_FFFF_FFFE
eb ← EA60:63

VRT ← undefined
if Big-Endian byte ordering then
   VRT8×eb:8×eb+15 ← MEM(EA,2)
else
   VRT112-(8×eb):127-(8×eb) ← MEM(EA,2)
```

Let the effective address (EA) be the result of ANDing 0xFFFF_FFFF_FFFF_FFFE with the sum (RA|0)+(RB).

Let eb be bits 60:63 of EA.

If Big-Endian byte ordering is used for the storage access,

- the contents of the byte in storage at address EA are placed into byte eb of register VRT,
- the contents of the byte in storage at address EA+1 are placed into byte eb+1 of register VRT, and
- the remaining bytes in register VRT are set to undefined values.

If Category: Vector.Little-Endian is supported, then if Little-Endian byte ordering is used for the storage access,

- the contents of the byte in storage at address EA are placed into byte 15-eb of register VRT,
- the contents of the byte in storage at address EA+1 are placed into byte 14-eb of register VRT, and
- the remaining bytes in register VRT are set to undefined values.

Special Registers Altered:
None

Load Vector Element Word Indexed X-form

lvewx VRT,RA,RB

31	VRT	RA	RB	71	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA  ← (b + (RB)) & 0xFFFF_FFFF_FFFF_FFFC

eb  ← EA60:63
VRT ← undefined
if Big-Endian byte ordering then
   VRT8×eb:8×eb+31 ← MEM(EA,4)
else
   VRT96-(8×eb):127-(8×eb) ← MEM(EA,4)
```

Let the effective address (EA) be the result of ANDing 0xFFFF_FFFF_FFFF_FFFC with the sum (RA|0)+(RB).

Let eb be bits 60:63 of EA.

If Big-Endian byte ordering is used for the storage access,

- the contents of the byte in storage at address EA are placed into byte eb of register VRT,
- the contents of the byte in storage at address EA+1 are placed into byte eb+1 of register VRT,
- the contents of the byte in storage at address EA+2 are placed into byte eb+2 of register VRT,
- the contents of the byte in storage at address EA+3 are placed into byte eb+3 of register VRT, and
- the remaining bytes in register VRT are set to undefined values.

If Category: Vector.Little-Endian is supported, then if Little-Endian byte ordering is used for the storage access,

- the contents of the byte in storage at address EA are placed into byte 15-eb of register VRT,
- the contents of the byte in storage at address EA+1 are placed into byte 14-eb of register VRT,
- the contents of the byte in storage at address EA+2 are placed into byte 13-eb of register VRT,
- the contents of the byte in storage at address EA+3 are placed into byte 12-eb of register VRT, and
- the remaining bytes in register VRT are set to undefined values.

Special Registers Altered:

None

Load Vector Indexed X-form

lvx VRT,RA,RB

31	VRT	RA	RB	103	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA  ← b + (RB)
VRT ← MEM(EA & 0xFFFF_FFFF_FFFF_FFF0, 16)
```

Let the effective address (EA) be the sum (RA|0)+(RB). The quadword in storage addressed by the result of EA ANDed with 0xFFFF_FFFF_FFFF_FFF0 is loaded into VRT.

Special Registers Altered:

None

Load Vector Indexed LRU X-form

lvxl VRT,RA,RB

31	VRT	RA	RB	359	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA  ← b + (RB)
VRT ← MEM(EA & 0xFFFF_FFFF_FFFF_FFF0, 16)
mark_as_not_likely_to_be_needed_again_anytime_soon
( EA )
```

Let the effective address (EA) be the sum (RA|0)+(RB). The quadword in storage addressed by the result of EA ANDed with 0xFFFF_FFFF_FFFF_FFF0 is loaded into VRT.

lvxl provides a hint that the quadword in storage addressed by EA will probably not be needed again by the program in the near future.

Special Registers Altered:

None

Programming Note

On some implementations, the hint provided by the ***lvxl*** instruction and the corresponding hint provided by the ***stvxl***, ***lvepxl***, and ***stvepxl*** instructions are applied to the entire cache block containing the specified quadword. On such implementations, the effect of the hint may be to cause that cache block to be considered a likely candidate for replacement when space is needed in the cache for a new block. Thus, on such implementations, the hint should be used with caution if the cache block containing the quadword also contains data that may be needed by the program in the near future. Also, the hint may be used before the last reference in a sequence of references to the quadword if the subsequent references are likely to occur sufficiently soon that the cache block containing the quadword is not likely to be displaced from the cache before the last reference.

5.7.3 Vector Store Instructions

Some portion or all of the contents of VRS are stored into the aligned byte, halfword, word, or quadword in storage addressed by EA.

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 RA = 0 then b ← 0
else           b ← (RA)
EA  ← b + (RB)
eb  ← EA60:63
if Big-Endian byte ordering then
   MEM(EA,1) ← VRS8×eb:8×eb+7
else
   MEM(EA,1) ← VRS120-(8×eb):127-(8×eb)
```

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

Let eb be bits 60:63 of EA.

If Big-Endian byte ordering is used for the storage access, the contents of byte eb of register VRS are placed in the byte in storage at address EA.

If Category: Vector.Little-Endian is supported, then if Little-Endian byte ordering is used for the storage access, the contents of byte 15-eb of register VRS are placed in the byte in storage at address EA.

Special Registers Altered:

None

Programming Note

Unless bits 60:63 of the address are known to match the byte offset of the subject byte element in register VRS, software should use *Vector Splat* to splat the subject byte element before performing the store.

Store Vector Element Halfword Indexed X-form

stvehx VRS,RA,RB

31	VRS	RA	RB	167	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA  ← (b + (RB)) & 0xFFFF_FFFF_FFFF_FFFE
eb  ← EA60:63
if Big-Endian byte ordering then
   MEM(EA,2) ← VRS8×eb:8×eb+15
else
   MEM(EA,2) ← VRS112-(8×eb):127-(8×eb)
```

Let the effective address (EA) be the result of ANDing 0xFFFF_FFFF_FFFF_FFFE with the sum (RA|0)+(RB).

Let eb be bits 60:63 of EA.

If Big-Endian byte ordering is used for the storage access,

- the contents of byte eb of register VRS are placed in the byte in storage at address EA, and
- the contents of byte eb+1 of register VRS are placed in the byte in storage at address EA+1.

If Category: Vector.Little-Endian is supported, then if Little-Endian byte ordering is used for the storage access,

- the contents of byte 15-eb of register VRS are placed in the byte in storage at address EA, and
- the contents of byte 14-eb of register VRS are placed in the byte in storage at address EA+1.

Special Registers Altered:

None

Programming Note

Unless bits 60:62 of the address are known to match the halfword offset of the subject halfword element in register VRS software should use *Vector Splat* to splat the subject halfword element before performing the store.

Store Vector Element Word Indexed X-form

stvewx VRS,RA,RB

31	VRS	RA	RB	199	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA  ← (b + (RB)) & 0xFFFF_FFFF_FFFF_FFFC
eb  ← EA60:63
if Big-Endian byte ordering then
   MEM(EA,4) ← VRS8×eb:8×eb+31
else
   MEM(EA,4) ← VRS96-(8×eb):127-(8×eb)
```

Let the effective address (EA) be the result of ANDing 0xFFFF_FFFF_FFFF_FFFC with the sum (RA|0)+(RB).

Let eb be bits 60:63 of EA.

If Big-Endian byte ordering is used for the storage access,

- the contents of byte eb of register VRS are placed in the byte in storage at address EA,
- the contents of byte eb+1 of register VRS are placed in the byte in storage at address EA+1,
- the contents of byte eb+2 of register VRS are placed in the byte in storage at address EA+2, and
- the contents of byte eb+3 of register VRS are placed in the byte in storage at address EA+3.

If Category: Vector.Little-Endian is supported, then if Little-Endian byte ordering is used for the storage access,

- the contents of byte 15-eb of register VRS are placed in the byte in storage at address EA,
- the contents of byte 14-eb of register VRS are placed in the byte in storage at address EA+1,
- the contents of byte 13-eb of register VRS are placed in the byte in storage at address EA+2, and
- the contents of byte 12-eb of register VRS 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 register VRS, software should use *Vector Splat* to splat the subject word element before performing the store.

Store Vector Indexed X-form

stvx VRS,RA,RB (L=0)

31	VRS	RA	RB	231	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA  ← b + (RB)
MEM(EA & 0xFFFF_FFFF_FFFF_FFF0, 16) ← (VRS)
```

Let the effective address (EA) be the sum (RA|0)+(RB). The contents of VRS are stored into the quadword in storage addressed by the result of EA ANDed with 0xFFFF_FFFF_FFFF_FFF0.

Store Vector Indexed LRU X-form

stvxl VRS,RA,RB (L=1)

31	VRS	RA	RB	487	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA  ← b + (RB)
MEM(EA & 0xFFFF_FFFF_FFFF_FFF0, 16) ← (VRS)
mark_as_not_likely_to_be_needed_again_anytime_soon
(EA)
```

Let the effective address (EA) be the sum (RA|0)+(RB). The contents of VRS are stored into the quadword in storage addressed by the result of EA ANDed with 0xFFFF_FFFF_FFFF_FFF0.

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 144.

5.7.4 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 157). Let X and Y be the contents of register VRA and VRB specified by the ***vperm***. The control vector created by ***lvsl*** causes the ***vperm*** to select the high-order 16 bytes of the result of shifting the 32-byte value X || Y left by sh bytes. The control vector created by ***lvsr*** causes the ***vperm*** to select the low-order 16 bytes of the result of shifting X || Y right by sh bytes.

Programming Note

Examples of uses of ***lvsl***, ***lvsr***, and ***vperm*** to load and store unaligned data are given in Section 5.4.1.

These instructions can also be used to rotate or shift the contents of a Vector Register left (***lvsl***) or right (***lvsr***) by sh bytes. For rotating, the Vector Register to be rotated should be specified as both register VRA and VRB for ***vperm***. For shifting left, VRB for ***vperm*** should be a register containing all zeros and VRA should contain the value to be shifted, and vice versa for shifting right.

Load Vector for Shift Left Indexed X-form

lvsl VRT,RA,RB

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

```
if RA = 0 then b ← 0
else           b ← (RA)
sh  ← (b + (RB))60:63
switch(sh)
 case(0x0): VRT←0x000102030405060708090A0B0C0D0E0F
 case(0x1): VRT←0x0102030405060708090A0B0C0D0E0F10
 case(0x2): VRT←0x02030405060708090A0B0C0D0E0F1011
 case(0x3): VRT←0x030405060708090A0B0C0D0E0F101112
 case(0x4): VRT←0x0405060708090A0B0C0D0E0F10111213
 case(0x5): VRT←0x05060708090A0B0C0D0E0F1011121314
 case(0x6): VRT←0x060708090A0B0C0D0E0F101112131415
 case(0x7): VRT←0x0708090A0B0C0D0E0F10111213141516
 case(0x8): VRT←0x08090A0B0C0D0E0F1011121314151617
 case(0x9): VRT←0x090A0B0C0D0E0F101112131415161718
 case(0xA): VRT←0x0A0B0C0D0E0F10111213141516171819
 case(0xB): VRT←0x0B0C0D0E0F101112131415161718191A
 case(0xC): VRT←0x0C0D0E0F101112131415161718191A1B
 case(0xD): VRT←0x0D0E0F101112131415161718191A1B1C
 case(0xE): VRT←0x0E0F101112131415161718191A1B1C1D
 case(0xF): VRT←0x0F101112131415161718191A1B1C1D1E
```

Let sh be bits 60:63 of the sum (RA|0)+(RB). Let X be the 32 byte value 0x00 || 0x01 || 0x02 || … || 0x1E || 0x1F.

Bytes sh to sh+15 of X are placed into VRT.

Special Registers Altered:

None

Load Vector for Shift Right Indexed X-form

lvsr VRT,RA,RB

31	VRT	RA	RB	38	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
sh  ← (b + (RB))60:63
switch(sh)
 case(0x0): VRT←0x101112131415161718191A1B1C1D1E1F
 case(0x1): VRT←0x0F101112131415161718191A1B1C1D1E
 case(0x2): VRT←0x0E0F101112131415161718191A1B1C1D
 case(0x3): VRT←0x0D0E0F101112131415161718191A1B1C
 case(0x4): VRT←0x0C0D0E0F101112131415161718191A1B
 case(0x5): VRT←0x0B0C0D0E0F101112131415161718191A
 case(0x6): VRT←0x0A0B0C0D0E0F10111213141516171819
 case(0x7): VRT←0x090A0B0C0D0E0F101112131415161718
 case(0x8): VRT←0x08090A0B0C0D0E0F1011121314151617
 case(0x9): VRT←0x0708090A0B0C0D0E0F10111213141516
 case(0xA): VRT←0x060708090A0B0C0D0E0F101112131415
 case(0xB): VRT←0x05060708090A0B0C0D0E0F1011121314
 case(0xC): VRT←0x0405060708090A0B0C0D0E0F10111213
 case(0xD): VRT←0x030405060708090A0B0C0D0E0F101112
 case(0xE): VRT←0x02030405060708090A0B0C0D0E0F1011
 case(0xF): VRT←0x0102030405060708090A0B0C0D0E0F10
```

Let sh be bits 60:63 of the sum (RA|0)+(RB). Let X be the 32-byte value 0x00 || 0x01 || 0x02 || … || 0x1E || 0x1F.

Bytes 16-sh to 31-sh of X are placed into VRT.

Special Registers Altered:

None

5.8 Vector Permute and Formatting Instructions

5.8.1 Vector Pack and Unpack Instructions

Vector Pack Pixel *VX-form*

vpkpx VRT,VRA,VRB

4	VRT	VRA	VRB	782
0	6	11	16	21 31

```
do i←0 to 63 by 16
   VRT_i          ← (VRA)_i×2+7
   VRT_i+1:i+5    ← (VRA)_i×2+8:i×2+12
   VRT_i+6:i+10   ← (VRA)_i×2+16:i×2+20
   VRT_i+11:i+15  ← (VRA)_i×2+24:i×2+28
   VRT_i+64       ← (VRB)_i×2+7
   VRT_i+65:i+69  ← (VRB)_i×2+8:i×2+12
   VRT_i+70:i+74  ← (VRB)_i×2+16:i×2+20
   VRT_i+75:i+79  ← (VRB)_i×2+24:i×2+28
```

Let the source vector be the concatenation of the contents of VRA followed by the contents of VRB.

For each vector element *i* from 0 to 7, do the following.

Word element *i* in the source vector is packed to produce a 16-bit value as described below.

- bit 7 of the first byte (bit 7 of the word)
- bits 0:4 of the second byte (bits 8:12 of the word)
- bits 0:4 of the third byte (bits 16:20 of the word)
- bits 0:4 of the fourth byte (bits 24:28 of the word)

The result is placed into halfword element *i* of VRT.

Special Registers Altered:

None

Programming Note

Each source word can be considered to be a 32-bit "pixel", consisting of four 8-bit "channels". Each target halfword can be considered to be a 16-bit pixel, consisting of one 1-bit channel and three 5-bit channels. A channel can be used to specify the intensity of a particular color, such as red, green, or blue, or to provide other information needed by the application.

Vector Pack Signed Halfword Signed Saturate VX-form

vpkshss VRT,VRA,VRB

4	VRT	VRA	VRB	398
0	6	11	16	21 31

```
do i=0 to 63 by 8
   VRT_{i:i+7} ←
   Clamp(EXTS((VRA)_{i×2:i×2+15} ), -128, 127)_{24:31}
   VRT_{i+64:i+71}←
   Clamp(EXTS((VRB)_{i×2:i×2+15} ), -128, 127)_{24:31}
```

Let the source vector be the concatenation of the contents of VRA followed by the contents of VRB.

For each vector element *i* from 0 to 15, do the following.

Signed-integer halfword element *i* in the source vector is converted to an signed-integer byte.

- If the value of the element is greater than 127 the result saturates to 127
- If the value of the element is less than -128 the result saturates to -128.

The low-order 8 bits of the result is placed into byte element *i* of VRT.

Special Registers Altered:
SAT

Vector Pack Signed Halfword Unsigned Saturate VX-form

vpkshus VRT,VRA,VRB

4	VRT	VRA	VRB	270
0	6	11	16	21 31

```
do i=0 to 63 by 8
   VRT_{i:i+7}
     ← Clamp(EXTS((VRA)_{i×2:i×2+15}), 0, 255)_{24:31}
   VRT_{i+64:i+71}
     ← Clamp(EXTS((VRB)_{i×2:i×2+15}), 0, 255)_{24:31}
```

Let the source vector be the concatenation of the contents of VRA followed by the contents of VRB.

For each vector element *i* from 0 to 15, do the following.

Signed-integer halfword element *i* in the source vector is converted to an unsigned-integer byte.

- If the value of the element is greater than 255 the result saturates to 255
- If the value of the element is less than 0 the result saturates to 0.

The low-order 8 bits of the result is placed into byte element *i* of VRT.

Special Registers Altered:
SAT

Vector Pack Signed Word Signed Saturate VX-form

vpkswss VRT,VRA,VRB

4	VRT	VRA	VRB	462
0	6	11	16	21 31

```
do i=0 to 63 by 16
   VRT_{i:i+15}
     ←  Clamp(EXTS((VRA)_{i×2:i×2+31}, -2^15, 2^15-1)_{16:31}
   VRT_{i+64:i+79}
     ← Clamp(EXTS((VRB)_{i×2:i×2+31}, -2^15, 2^15-1)_{16:31}
```

Let the source vector be the concatenation of the contents of VRA followed by the contents of VRB.

For each vector element *i* from 0 to 7, do the following.

Signed-integer word element *i* in the source vector is converted to an signed-integer halfword.

- If the value of the element is greater than $2^{15}-1$ the result saturates to $2^{15}-1$
- If the value of the element is less than -2^{15} the result saturates to -2^{15}.

The low-order 16 bits of the result is placed into halfword element *i* of VRT.

Special Registers Altered:
SAT

Vector Pack Signed Word Unsigned Saturate VX-form

vpkswus VRT,VRA,VRB

4	VRT	VRA	VRB	334
0	6	11	16	21 31

```
do i=0 to 63 by 16
   VRT_{i:i+15}
     ← Clamp(EXTS((VRA)_{i×2:i×2+31}), 0, 2^16-1)_{16:31}
   VRT_{i+64:i+79}
     ← Clamp(EXTS((VRB)_{i×2:i×2+31}), 0, 2^16-1 )_{16:31}
```

Let the source vector be the concatenation of the contents of VRA followed by the contents of VRB.

For each vector element *i* from 0 to 7, do the following.

Signed-integer word element *i* in the source vector is converted to an unsigned-integer halfword.

- If the value of the element is greater than $2^{16}-1$ the result saturates to $2^{16}-1$
- If the value of the element is less than 0 the result saturates to 0.

The low-order 16 bits of the result is placed into halfword element *i* of VRT.

Special Registers Altered:
SAT

Vector Pack Unsigned Halfword Unsigned Modulo VX-form

vpkuhum VRT,VRA,VRB

4	VRT	VRA	VRB	14
0	6	11	16	21 31

```
do i=0 to 63 by 8
   VRT[i:i+7]     ← (VRA)[i×2+8:i×2+15]
   VRT[i+64:i+71] ← (VRB)[i×2+8:i×2+15]
```

Let the source vector be the concatenation of the contents of VRA followed by the contents of VRB.

For each vector element *i* from 0 to 15, do the following.

The contents of bits 8:15 of halfword element *i* in the source vector is placed into byte element *i* of VRT.

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

```
do i=0 to 63 by 8
   VRT[i:i+7]
     ← Clamp( EXTZ((VRA)[i×2:i×2+15]), 0, 255 )[24:31]
   VRT[i+64:i+71]
     ← Clamp( EXTZ((VRB)[i×2:i×2+15]), 0, 255 )[24:31]
```

Let the source vector be the concatenation of the contents of VRA followed by the contents of VRB.

For each vector element *i* from 0 to 15, do the following.

Unsigned-integer halfword element *i* in the source vector is converted to an unsigned-integer byte.

- If the value of the element is greater than 255 the result saturates to 255.

The low-order 8 bits of the result is placed into byte element *i* of VRT.

Special Registers Altered:
SAT

Vector Pack Unsigned Word Unsigned Modulo VX-form

vpkuwum VRT,VRA,VRB

4	VRT	VRA	VRB	78
0	6	11	16	21 31

```
do i=0 to 63 by 16
   VRT[i:i+15]    ← (VRA)[i×2+16:i×2+31]
   VRT[i+64:i+79] ← (VRB)[i×2+16:i×2+31]
```

Let the source vector be the concatenation of the contents of VRA followed by the contents of VRB.

For each vector element *i* from 0 to 7, do the following.

The contents of bits 16:31 of word element *i* in the source vector is placed into halfword element *i* of VRT.

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

```
do i=0 to 63 by 16
   VRT[i:i+15]
     ← Clamp( EXTZ((VRA)[i×2:i×2+31] ), 0, 2^16-1 )[16:31]
   VRT[i+64:i+79]
     ← Clamp( EXTZ((VRB)[i×2:i×2+31] ), 0, 2^16-1 )[16:31]
```

Let the source vector be the concatenation of the contents of VRA followed by the contents of VRB.

For each vector element *i* from 0 to 7, do the following.

Unsigned-integer word element *i* in the source vector is converted to an unsigned-integer halfword.

- If the value of the element is greater than 2^{16}-1 the result saturates to 2^{16}-1.

The low-order 16 bits of the result is placed into halfword element *i* of VRT.

Special Registers Altered:
SAT

Vector Unpack High Pixel VX-form

vupkhpx VRT,VRB

4	VRT	///	VRB	846
0	6	11	16	21 31

```
do i=0 to 63 by 16
   VRT_{i×2:i×2+7}      ← EXTS((VRB)_{i}        )
   VRT_{i×2+8:i×2+15}   ← EXTZ((VRB)_{i+1:i+5}  )
   VRT_{i×2+16:i×2+23}  ← EXTZ((VRB)_{i+6:i+10} )
   VRT_{i×2+24:i×2+31}  ← EXTZ((VRB)_{i+11:i+15})
```

For each vector element *i* from 0 to 3, do the following.

Halfword element *i* in VRB is unpacked as follows.

- sign-extend bit 0 of the halfword to 8 bits
- zero-extend bits 1:5 of the halfword to 8 bits
- zero-extend bits 6:10 of the halfword to 8 bits
- zero-extend bits 11:15 of the halfword to 8 bits

The result is placed in word element *i* of VRT.

Special Registers Altered:
None

Programming Note

The source and target elements can be considered to be 16-bit and 32-bit "pixels" respectively, having the formats described in the Programming Note for the *Vector Pack Pixel* instruction on page 149.

Programming Note

Notice that the unpacking done by the *Vector Unpack Pixel* instructions does not reverse the packing done by the *Vector Pack Pixel* instruction. Specifically, if a 16-bit pixel is unpacked to a 32-bit pixel which is then packed to a 16-bit pixel, the resulting 16-bit pixel will not, in general, be equal to the original 16-bit pixel (because, for each channel except the first, *Vector Unpack Pixel* inserts high-order bits while *Vector Pack Pixel* discards low-order bits).

Vector Unpack High Signed Byte VX-form

vupkhsb VRT,VRB

4	VRT	///	VRB	526
0	6	11	16	21 31

```
do i=0 to 63 by 8
   VRT_{i×2:i×2+15} ← EXTS((VRB)_{i:i+7})
```

For each vector element *i* from 0 to 7, do the following.

Signed-integer byte element *i* in VRB is sign-extended to produce a signed-integer halfword and placed into halfword element *i* in VRT.

Special Registers Altered:
None

Vector Unpack High Signed Halfword VX-form

vupkhsh VRT,VRB

4	VRT	///	VRB	590
0	6	11	16	21 31

```
do i=0 to 63 by 16
   VRT_{i×2:i×2+31} ← EXTS((VRB)_{i:i+15})
```

For each vector element *i* from 0 to 3, do the following.

Signed-integer halfword element *i* in VRB is sign-extended to produce a signed-integer word and placed into word element *i* in VRT.

Special Registers Altered:
None

Vector Unpack Low Pixel VX-form

vupklpx VRT,VRB

4	VRT	///	VRB	974
0	6	11	16	21 31

```
do i=0 to 63 by 16
   VRT_{i×2:i×2+7}      ← EXTS((VRB)_{i+64}     )
   VRT_{i×2+8:i×2+15}   ← EXTZ((VRB)_{i+65:i+69})
   VRT_{i×2+16:i×2+23}  ← EXTZ((VRB)_{i+70:i+74})
   VRT_{i×2+24:i×2+31}  ← EXTZ((VRB)_{i+75:i+79})
```

For each vector element *i* from 0 to 3, do the following.

Halfword element *i*+4 in VRB is unpacked as follows.
- sign-extend bit 0 of the halfword to 8 bits
- zero-extend bits 1:5 of the halfword to 8 bits
- zero-extend bits 6:10 of the halfword to 8 bits
- zero-extend bits 11:15 of the halfword to 8 bits

The result is placed in word element *i* of VRT.

Special Registers Altered:
None

Vector Unpack Low Signed Byte VX-form

vupklsb VRT,VRB

4	VRT	///	VRB	654
0	6	11	16	21 31

```
do i=0 to 63 by 8
   VRT_{i×2:i×2+15} ← EXTS((VRB)_{i+64:i+71})
```

For each vector element *i* from 0 to 7, do the following.

Signed-integer byte element *i*+8 in VRB is sign-extended to produce a signed-integer halfword and placed into halfword element *i* in VRT.

Special Registers Altered:
None

Vector Unpack Low Signed Halfword VX-form

vupklsh VRT,VRB

4	VRT	///	VRB	718
0	6	11	16	21 31

```
do i=0 to 63 by 16
   VRT_{i×2:i×2+31} ← EXTS((VRB)_{i+64:i+79})
```

For each vector element *i* from 0 to 3, do the following.

Signed-integer halfword element *i*+4 in VRB is sign-extended to produce a signed-integer word and placed into word element *i* in VRT.

Special Registers Altered:
None

5.8.2 Vector Merge Instructions

Vector Merge High Byte *VX-form*

vmrghb VRT,VRA,VRB

4	VRT	VRA	VRB	12
0	6	11	16	21 31

```
do i=0 to 63 by 8
   VRT_{i×2:i×2+7}     ← (VRA)_{i:i+7}
   VRT_{i×2+8:i×2+15} ← (VRB)_{i:i+7}
```

For each vector element *i* from 0 to 7, do the following.

Byte element *i* in VRA is placed into byte element 2×i in VRT.

Byte element *i* in VRB is placed into byte element 2×i+1 in VRT.

Special Registers Altered:
None

Vector Merge High Halfword *VX-form*

vmrghh VRT,VRA,VRB

4	VRT	VRA	VRB	76
0	6	11	16	21 31

```
do i=0 to 63 by 16
   VRT_{i×2:i×2+15}     ← (VRA)_{i:i+15}
   VRT_{i×2+16:i×2+31} ← (VRB)_{i:i+15}
```

For each vector element *i* from 0 to 3, do the following.

Halfword element *i* in VRA is placed into halfword element 2×i in VRT.

Halfword element *i* in VRB is placed into halfword element 2×i+1 in VRT.

Special Registers Altered:
None

Vector Merge High Word *VX-form*

vmrghw VRT,VRA,VRB

4	VRT	VRA	VRB	140
0	6	11	16	21 31

```
do i=0 to 63 by 32
   VRT_{i×2:i×2+31}     ← (VRA)_{i:i+31}
   VRT_{i×2+32:i×2+63} ← (VRB)_{i:i+31}
```

For each vector element *i* from 0 to 1, do the following.

Word element *i* in VRA is placed into word element 2×i in VRT.

Word element *i* in VRB is placed into word element 2×i+1 in VRT.

The word elements in the high-order half of VRA are placed, in the same order, into the even-numbered word elements of VRT. The word elements in the high-order half of VRB are placed, in the same order, into the odd-numbered word elements of VRT.

Special Registers Altered:
None

Vector Merge Low Byte VX-form

vmrglb VRT,VRA,VRB

4	VRT	VRA	VRB	268
0	6	11	16	21 31

```
do i=0 to 63 by 8
   VRT[i×2:i×2+7]    ← (VRA)[i+64:i+71]
   VRT[i×2+8:i×2+15] ← (VRB)[i+64:i+71]
```

For each vector element *i* from 0 to 7, do the following.

Byte element i+8 in VRA is placed into byte element 2×i in VRT.

Byte element i+8 in VRB is placed into byte element 2×i+1 in VRT.

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

```
do i=0 to 63 by 16
   VRT[i×2:i×2+15]    ← (VRA)[i+64:i+79]
   VRT[i×2+16:i×2+31] ← (VRB)[i+64:i+79]
```

For each vector element *i* from 0 to 3, do the following.

Halfword element i+4 in VRA is placed into halfword element 2×i in VRT.

Halfword element i+4 in VRB is placed into halfword element 2×i+1 in VRT.

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

```
do i=0 to 63 by 32
   VRT[i×2:i×2+31]    ← (VRA)[i+64:i+95]
   VRT[i×2+32:i×2+63] ← (VRB)[i+64:i+95]
```

For each vector element *i* from 0 to 1, do the following.

Word element i+2 in VRA is placed into word element 2×i in VRT.

Word element i+2 in VRB is placed into word element 2×i+1 in VRT.

Special Registers Altered:
None

5.8.3 Vector Splat Instructions

Programming Note

The *Vector Splat* instructions can be used in preparation for performing arithmetic for which one source vector is to consist of elements that all have the same value (e.g., multiplying all elements of a Vector Register by a constant).

Vector Splat Byte *VX-form*

vspltb VRT,VRB,UIM

4	VRT	/	UIM	VRB	524
0	6	11	12	16	21 31

```
b ← UIM || 0b000
do i=0 to 127 by 8
   VRT_{i:i+7} ← (VRB)_{b:b+7}
```

For each vector element *i* from 0 to 15, do the following.
The contents of byte element UIM in VRB are placed into byte element *i* of VRT.

Special Registers Altered:
None

Vector Splat Halfword *VX-form*

vsplth VRT,VRB,UIM

4	VRT	//	UIM	VRB	588
0	6	11	13	16	21 31

```
b ← UIM || 0b0000
do i=0 to 127 by 16
   VRT_{i:i+15} ← (VRB)_{b:b+15}
```

For each vector element *i* from 0 to 7, do the following.
The contents of halfword element UIM in VRB are placed into halfword element *i* of VRT.

Special Registers Altered:
None

Vector Splat Word *VX-form*

vspltw VRT,VRB,UIM

4	VRT	///	UIM	VRB	652
0	6	11	14	16	21 31

```
b ← UIM || 0b00000
do i=0 to 127 by 32
   VRT_{i:i+31} ← (VRB)_{b:b+31}
```

For each vector element *i* from 0 to 3, do the following.
The contents of word element UIM in VRB are placed into word element *i* of VRT.

Special Registers Altered:
None

Vector Splat Immediate Signed Byte *VX-form*

vspltisb VRT,SIM

4	VRT	SIM	///	780
0	6	11	16	21 31

```
do i=0 to 127 by 8
   VRT_{i:i+7} ← EXTS(SIM, 8)
```

For each vector element *i* from 0 to 15, do the following.
The value of the SIM field, sign-extended to 8 bits, is placed into byte element *i* of VRT.

Special Registers Altered:
None

Vector Splat Immediate Signed Halfword *VX-form*

vspltish VRT,SIM

4	VRT	SIM	///	844
0	6	11	16	21 31

```
do i=0 to 127 by 16
   VRT_{i:i+15} ← EXTS(SIM, 16)
```

For each vector element *i* from 0 to 7, do the following.
The value of the SIM field, sign-extended to 16 bits, is placed into halfword element *i* of VRT.

Special Registers Altered:
None

Vector Splat Immediate Signed Word *VX-form*

vspltisw VRT,SIM

4	VRT	SIM	///	908
0	6	11	16	21 31

```
do i=0 to 127 by 32
   VRT_{i:i+31} ← EXTS(SIM, 32)
```

For each vector element *i* from 0 to 3, do the following.
The value of the SIM field, sign-extended to 32 bits, is placed into word element *i* of VRT.

Special Registers Altered:
None

5.8.4 Vector Permute Instruction

The *Vector Permute* instruction allows any byte in two source Vector Registers to be copied to any byte in the target Vector Register. The bytes in a third source Vector Register specify from which byte in the first two source Vector Registers the corresponding target byte is to be copied. The contents of the third source Vector Register are sometimes referred to as the “permute control vector”.

Vector Permute *VA-form*

vperm VRT,VRA,VRB,VRC

4	VRT	VRA	VRB	VRC	43
0	6	11	16	21	26 31

```
temp0:255 ← (VRA) || (VRB)
do i=0 to 127 by 8
   b ← (VRC)i+3:i+7 || 0b000
   VRTi:i+7 ← tempb:b+7
```

Let the source vector be the concatenation of the contents of VRA followed by the contents of VRB.

For each vector element *i* from 0 to 15, do the following.

> The contents of the byte element in the source vector specified by bits 3:7 of byte element *i* of VRC are placed into byte element *i* of VRT.

Special Registers Altered:
None

Programming Note

See the Programming Notes with the *Load Vector for Shift Left* and *Load Vector for Shift Right* instructions on page 148 for examples of uses of ***vperm***.

5.8.5 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

```
do i=0 to 127
   VRTi ← ((VRC)i=0) ? (VRA)i : (VRB)i
```

For each bit in VRC that contains the value 0, the corresponding bit in VRA is placed into the corresponding bit of VRT. Otherwise, the corresponding bit in VRB is placed into the corresponding bit of VRT.

Special Registers Altered:
None

5.8.6 Vector Shift Instructions

The *Vector Shift* instructions rotate or shift the contents of a Vector Register or a pair of Vector Registers left or right by a specified number of bytes (***vslo***, ***vsro***, ***vsldoi***) or bits (***vsl***, ***vsr***). Depending on the instruction, this “shift count” is specified either by the contents of a Vector Register or by an immediate field in the instruction. In the former case, 7 bits of the shift count register give the shift count in bits ($0 \le \text{count} \le 127$). 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 Vector Register left or right by the number of bits (0-127) specified in the shift count register. The following example shifts the contents of register Vx left by the number of bits specified in register Vy and places the result into register Vz.

```
vslo    Vz,Vx,Vy
vsl     Vz,Vz,Vy
```

Vector Shift Left — *VX-form*

vsl VRT,VRA,VRB

4	VRT	VRA	VRB	452
0	6	11	16	21 … 31

```
sh ← (VRB)125:127
t ← 1
do i=0 to 127 by 8
   t ← t & ((VRB)i+5:i+7=sh)
if t=1 then VRT ← (VRA) << sh
else        VRT ← undefined
```

The contents of VRA are shifted left by the number of bits specified in $(VRB)_{125:127}$.

- Bits shifted out of bit 0 are lost.
- Zeros are supplied to the vacated bits on the right.

The result is place into VRT, except if, for any byte element in register VRB, the low-order 3 bits are not equal to the shift amount, then VRT is undefined.

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

```
VRT ← ( (VRA) || (VRB) )8×SHB:8×SHB+127
```

Let the source vector be the concatenation of the contents of VRA followed by the contents of VRB. Bytes SHB:SHB+15 of the source vector are placed into VRT.

Special Registers Altered:

None

Vector Shift Left by Octet — *VX-form*

vslo VRT,VRA,VRB

4	VRT	VRA	VRB	1036
0	6	11	16	21 … 31

```
shb ← (VRB)121:124
VRT ← (VRA) << ( shb || 0b000 )
```

The contents of VRA are shifted left by the number of bytes specified in $(VRB)_{121:124}$.

- Bytes shifted out of byte 0 are lost.
- Zeros are supplied to the vacated bytes on the right.

The result is placed into VRT.

Special Registers Altered:

None

Vector Shift Right *VX-form*

vsr VRT,VRA,VRB

4	VRT	VRA	VRB	708
0	6	11	16	21 31

```
sh ← (VRB)125:127
t ← 1
do i=0 to 127 by 8
   t ← t & ((VRB)i+5:i+7=sh)
if t=1 then VRT ← (VRA) >>ui sh
else        VRT ← undefined
```

The contents of VRA are shifted right by the number of bits specified in $(VRB)_{125:127}$.

- Bits shifted out of bit 127 are lost.
- Zeros are supplied to the vacated bits on the left.

The result is place into VRT, except if, for any byte element in register VRB, the low-order 3 bits are not equal to the shift amount, then VRT is undefined.

Special Registers Altered:
None

Programming Note

A double-register shift by a dynamically specified number of bits (0-127) can be performed in six instructions. The following example shifts 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
vsl   Vt1,Vt1,Vy
vsububm Vt3,V0,Vy #adjust shift count ((V0)=0)
vsro  Vt2,Vx,Vt3  #shift low-order reg right
vsr   Vt2,Vt2,Vt3
vor   Vz,Vt1,Vt2  #merge to get final result
```

Vector Shift Right by Octet *VX-form*

vsro VRT,VRA,VRB

4	VRT	VRA	VRB	1100
0	6	11	16	21 31

```
shb ← (VRB)121:124
VRT ← (VRA) >>ui ( shb || 0b000 )
```

The contents of VRA are shifted right by the number of bytes specified in $(VRB)_{121:124}$.

- Bytes shifted out of byte 15 are lost.
- Zeros are supplied to the vacated bytes on the left.

The result is placed into VRT.

Special Registers Altered:
None

5.9 Vector Integer Instructions

5.9.1 Vector Integer Arithmetic Instructions

5.9.1.1 Vector Integer Add Instructions

Vector Add and Write Carry-out Unsigned Word VX-form

vaddcuw VRT,VRA,VRB

4	VRT	VRA	VRB	384
0	6	11	16	21 31

```
do i=0 to 127 by 32
   aop  ← EXTZ((VRA)i:i+31)
   bop  ← EXTZ((VRB)i:i+31)
   VRTi:i+31 ← Chop( ( aop +int bop ) >>ui 32,1)
```

For each vector element *i* from 0 to 3, do the following.

Unsigned-integer word element *i* in VRA is added to unsigned-integer word element *i* in VRB. The carry out of the 32-bit sum is zero-extended to 32 bits and placed into word element *i* of VRT.

Special Registers Altered:
None

Vector Add Signed Byte Saturate VX-form

vaddsbs VRT,VRA,VRB

4	VRT	VRA	VRB	768
0	6	11	16	21 31

```
do i=0 to 127 by 8
   aop ← EXTS(VRAi:i+7)
   bop ← EXTS(VRBi:i+7)
   VRTi:i+7 ← Clamp( aop +int bop, -128, 127 )24:31
```

For each vector element *i* from 0 to 15, do the following.

Signed-integer byte element *i* in VRA is added to signed-integer byte element *i* in VRB.

- If the sum is greater than 127 the result saturates to 127.
- If the sum is less than -128 the result saturates to -128.

The low-order 8 bits of the result are placed into byte element *i* of VRT.

Special Registers Altered:
SAT

Vector Add Signed Halfword Saturate VX-form

vaddshs VRT,VRA,VRB

4	VRT	VRA	VRB	832
0	6	11	16	21 31

```
do i=0 to 127 by 16
   aop ← EXTS((VRA)i:i+15)
   bop ← EXTS((VRB)i:i+15)
   VRTi:i+15
     ← Clamp(aop +int bop, -2^15, 2^15-1)16:31
```

For each vector element *i* from 0 to 7, do the following.

Signed-integer halfword element *i* in VRA is added to signed-integer halfword element *i* in VRB.

- If the sum is greater than 2^{15}-1 the result saturates to 2^{15}-1
- If the sum is less than -2^{15} the result saturates to -2^{15}.

The low-order 16 bits of the result are placed into halfword element *i* of VRT.

Special Registers Altered:
SAT

Vector Add Signed Word Saturate VX-form

vaddsws VRT,VRA,VRB

4	VRT	VRA	VRB	896
0	6	11	16	21 31

```
do i=0 to 127 by 32
   aop ← EXTS((VRA)i:i+31)
   bop ← EXTS((VRB)i:i+31)
   VRTi:i+31 ← Clamp(aop +int bop, -2^31, 2^31-1)
```

For each vector element *i* from 0 to 3, do the following.

Signed-integer word element *i* in VRA is added to signed-integer word element *i* in VRB.

- If the sum is greater than 2^{31}-1 the result saturates to 2^{31}-1.
- If the sum is less than -2^{31} the result saturates to -2^{31}.

The low-order 32 bits of the result are placed into word element *i* of VRT.

Special Registers Altered:
SAT

Vector Add Unsigned Byte Modulo VX-form

vaddubm VRT,VRA,VRB

4	VRT	VRA	VRB	0
0	6	11	16	21 31

```
do i=0 to 127 by 8
   aop ← EXTZ((VRA)i:i+7)
   bop ← EXTZ((VRB)i:i+7)
   VRTi:i+7 ← Chop( aop +int bop, 8 )
```

For each vector element *i* from 0 to 15, do the following.

Unsigned-integer byte element *i* in VRA is added to unsigned-integer byte element *i* in VRB.

The low-order 8 bits of the result are placed into byte element *i* of VRT.

Special Registers Altered:
None

Programming Note

vaddubm can be used for unsigned or signed-integers.

Vector Add Unsigned Word Modulo VX-form

vadduwm VRT,VRA,VRB

4	VRT	VRA	VRB	128
0	6	11	16	21 31

```
do i=0 to 127 by 32
   aop  ← EXTZ((VRA)i:i+31)
   bop  ← EXTZ((VRB)i:i+31)
   temp ← aop +int bop
   VRTi:i+31 ← Chop( aop +int bop, 32 )
```

For each vector element *i* from 0 to 3, do the following.

Unsigned-integer word element *i* in VRA is added to unsigned-integer word element *i* in VRB.

The low-order 32 bits of the result are placed into word element *i* of VRT.

Special Registers Altered:
None

Programming Note

vadduwm 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

```
do i=0 to 127 by 16
   aop  ← EXTZ((VRA)i:i+15)
   bop  ← EXTZ((VRB)i:i+15)
   VRTi:i+15 ← Chop( aop +int bop, 16 )
```

For each vector element *i* from 0 to 7, do the following.

Unsigned-integer halfword element *i* in VRA is added to unsigned-integer halfword element *i* in VRB.

The low-order 16 bits of the result are placed into halfword element *i* of VRT.

Special Registers Altered:
None

Programming Note

vadduhm can be used for unsigned or signed-integers.

Vector Add Unsigned Byte Saturate VX-form

vaddubs VRT,VRA,VRB

4	VRT	VRA	VRB	512
0	6	11	16	21 31

```
do i=0 to 127 by 8
   aop ← EXTZ((VRA)i:i+7)
   bop ← EXTZ((VRB)i:i+7)
   VRTi:i+7 ← Clamp( aop +int bop, 0, 255 )24:31
```

For each vector element *i* from 0 to 15, do the following.

Unsigned-integer byte element *i* in VRA is added to unsigned-integer byte element *i* in VRB.

- If the sum is greater than 255 the result saturates to 255.

The low-order 8 bits of the result are placed into byte element *i* of VRT.

Special Registers Altered:
SAT

Vector Add Unsigned Word Saturate VX-form

vadduws VRT,VRA,VRB

4	VRT	VRA	VRB	640
0	6	11	16	21 31

```
do i=0 to 127 by 32
   aop  ← EXTZ((VRA)i:i+31)
   bop  ← EXTZ((VRB)i:i+31)
   VRTi:i+31 ← Clamp(aop +int bop, 0, 2^32-1)
```

For each vector element *i* from 0 to 3, do the following.

Unsigned-integer word element *i* in VRA is added to unsigned-integer word element *i* in VRB.

- If the sum is greater than 2^{32}-1 the result saturates to 2^{32}-1.

The low-order 32 bits of the result are placed into word element *i* of VRT.

Special Registers Altered:
SAT

Vector Add Unsigned Halfword Saturate VX-form

vadduhs VRT,VRA,VRB

4	VRT	VRA	VRB	576
0	6	11	16	21 31

```
do i=0 to 127 by 16
   aop ← EXTZ((VRA)i:i+15)
   bop ← EXTZ((VRB)i:i+15)
   VRTi:i+15 ← Clamp(aop +int bop, 0, 2^16-1)16:31
```

For each vector element *i* from 0 to 7, do the following.

Unsigned-integer halfword element *i* in VRA is added to unsigned-integer halfword element *i* in VRB.

- If the sum is greater than 2^{16}-1 the result saturates to 2^{16}-1.

The low-order 16 bits of the result are placed into halfword element *i* of VRT.

Special Registers Altered:
SAT

5.9.1.2 Vector Integer Subtract Instructions

Vector Subtract and Write Carry-Out Unsigned Word VX-form

vsubcuw VRT,VRA,VRB

4 (0)	VRT (6)	VRA (11)	VRB (16)	1408 (21–31)

```
do i=0 to 127 by 32
   aop  ← (VRA)i:i+31
   bop  ← (VRB)i:i+31
   temp ← (EXTZ(aop) +int EXTZ(¬bop) +int 1) >> 32
   VRTi:i+31 ← temp & 0x0000_0001
```

For each vector element *i* from 0 to 3, do the following.

Unsigned-integer word element *i* in VRB is subtracted from unsigned-integer word element *i* in VRA. The complement of the borrow out of bit 0 of the 32-bit difference is zero-extended to 32 bits and placed into word element *i* of VRT.

Special Registers Altered:
None

Vector Subtract Signed Byte Saturate VX-form

vsubsbs VRT,VRA,VRB

4 (0)	VRT (6)	VRA (11)	VRB (16)	1792 (21–31)

```
do i=0 to 127 by 8
   aop ← EXTS((VRA)i:i+7)
   bop ← EXTS((VRB)i:i+7)
   VRTi:i+7 ←
        Clamp(aop +int ¬bop +int 1, -128, 127)24:31
```

For each vector element *i* from 0 to 15, do the following.

Signed-integer byte element *i* in VRB is subtracted from signed-integer byte element *i* in VRA.
- If the intermediate result is greater than 127 the result saturates to 127.
- If the intermediate result is less than -128 the result saturates to -128.

The low-order 8 bits of the result are placed into byte element *i* of VRT.

Special Registers Altered:
SAT

Vector Subtract Signed Halfword Saturate VX-form

vsubshs VRT,VRA,VRB

4 (0)	VRT (6)	VRA (11)	VRB (16)	1856 (21–31)

```
do i=0 to 127 by 16
   aop ← EXTS((VRA)i:i+15)
   bop ← EXTS((VRB)i:i+15)
   VRTi:i+15
     ← Clamp(aop +int ¬bop +int 1, -2^15, 2^15-1)16:31
```

For each vector element *i* from 0 to 7, do the following.

Signed-integer halfword element *i* in VRB is subtracted from signed-integer halfword element *i* in VRA.
- If the intermediate result is greater than 2^{15}-1 the result saturates to 2^{15}-1.
- If the intermediate result is less than -2^{15} the result saturates to -2^{15}.

The low-order 16 bits of the result are placed into halfword element *i* of VRT.

Special Registers Altered:
SAT

Vector Subtract Signed Word Saturate VX-form

vsubsws VRT,VRA,VRB

4 (0)	VRT (6)	VRA (11)	VRB (16)	1920 (21–31)

```
do i=0 to 127 by 32
   aop ← EXTS((VRA)i:i+31)
   bop ← EXTS((VRB)i:i+31)
   VRTi:i+31 ← Clamp(aop +int ¬bop +int 1,-2^31,2^31-1)
```

For each vector element *i* from 0 to 3, do the following.

Signed-integer word element *i* in VRB is subtracted from signed-integer word element *i* in VRA.
- If the intermediate result is greater than 2^{31}-1 the result saturates to 2^{31}-1.
- If the intermediate result is less than -2^{31} the result saturates to -2^{31}.

The low-order 32 bits of the result are placed into word element *i* of VRT.

Special Registers Altered:
SAT

Vector Subtract Unsigned Byte Modulo VX-form

vsububm VRT,VRA,VRB

4	VRT	VRA	VRB	1024
0	6	11	16	21 31

```
do i=0 to 127 by 8
   aop ← EXTZ((VRA)i:i+7)
   bop ← EXTZ((VRB)i:i+7)
   VRTi:i+7 ← Chop( aop +int ¬bop +int 1, 8 )
```

For each vector element *i* from 0 to 15, do the following.

Unsigned-integer byte element *i* in VRB is subtracted from unsigned-integer byte element *i* in VRA. The low-order 8 bits of the result are placed into byte element *i* of VRT.

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

```
do i=0 to 127 by 16
   aop  ← EXTZ((VRA)i:i+15)
   bop  ← EXTZ((VRB)i:i+15)
   VRTi:i+16 ← Chop( aop +int ¬bop +int 1, 16 )
```

For each vector element *i* from 0 to 7, do the following.

Unsigned-integer halfword element *i* in VRB is subtracted from unsigned-integer halfword element *i* in VRA. The low-order 16 bits of the result are placed into halfword element *i* of VRT.

Special Registers Altered:
None

Vector Subtract Unsigned Word Modulo VX-form

vsubuwm VRT,VRA,VRB

4	VRT	VRA	VRB	1152
0	6	11	16	21 31

```
do i=0 to 127 by 32
   aop  ← EXTZ((VRA)i:i+31)
   bop  ← EXTZ((VRB)i:i+31)
   VRTi:i+31 ← Chop( aop +int ¬bop +int 1, 32 )
```

For each vector element *i* from 0 to 3, do the following.

Unsigned-integer word element *i* in VRB is subtracted from unsigned-integer word element *i* in VRA. The low-order 32 bits of the result are placed into word element *i* of VRT.

Special Registers Altered:
None

Vector Subtract Unsigned Byte Saturate VX-form

vsububs VRT,VRA,VRB

4	VRT	VRA	VRB	1536
0	6	11	16	21 31

```
do i=0 to 127 by 8
   aop ← EXTZ((VRA)i:i+7)
   bop ← EXTZ((VRB)i:i+7)
   VRTi:i+7 ← Clamp(aop +int ¬bop +int 1, 0, 255)24:31
```

For each vector element *i* from 0 to 15, do the following.

> Unsigned-integer byte element *i* in VRB is subtracted from unsigned-integer byte element *i* in VRA. If the intermediate result is less than 0 the result saturates to 0. The low-order 8 bits of the result are placed into byte element *i* of VRT.

Special Registers Altered:
SAT

Vector Subtract Unsigned Halfword Saturate VX-form

vsubuhs VRT,VRA,VRB

4	VRT	VRA	VRB	1600
0	6	11	16	21 31

```
do i=0 to 127 by 16
   aop  ← EXTZ((VRA)i:i+15)
   bop  ← EXTZ((VRB)i:i+15)
   VRTi:i+15 ← Clamp(aop +int ¬bop +int 1,0,2^16-1)16:31
```

For each vector element *i* from 0 to 7, do the following.

> Unsigned-integer halfword element *i* in VRB is subtracted from unsigned-integer halfword element *i* in VRA. If the intermediate result is less than 0 the result saturates to 0. The low-order 16 bits of the result are placed into halfword element *i* of VRT.

Special Registers Altered:
SAT

Vector Subtract Unsigned Word Saturate VX-form

vsubuws VRT,VRA,VRB

4	VRT	VRA	VRB	1664
0	6	11	16	21 31

```
do i=0 to 127 by 32
   aop  ← EXTZ((VRA)i:i+31)
   bop  ← EXTZ((VRB)i:i+31)
   VRTi:i+31 ← Clamp(aop +int ¬bop +int 1, 0, 2^32-1)
```

For each vector element *i* from 0 to 7, do the following.

> Unsigned-integer word element *i* in VRB is subtracted from unsigned-integer word element *i* in VRA.
>
> - If the intermediate result is less than 0 the result saturates to 0.
>
> The low-order 32 bits of the result are placed into word element *i* of VRT.

Special Registers Altered:
SAT

5.9.1.3 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

```
do i=0 to 127 by 16
   prod ← EXTS((VRA)i:i+7) ×si EXTS((VRB)i:i+7)
   VRTi:i+15 ← Chop( prod, 16 )
```

For each vector element i from 0 to 7, do the following.

> Signed-integer byte element i×2 in VRA is multiplied by signed-integer byte element i×2 in VRB. The low-order 16 bits of the product are placed into halfword element i VRT.

Special Registers Altered:
None

Vector Multiply Even Signed Halfword VX-form

vmulesh VRT,VRA,VRB

4	VRT	VRA	VRB	840
0	6	11	16	21 31

```
do i=0 to 127 by 32
  prod ← EXTS((VRA)i:i+15) ×si EXTS((VRB)i:i+15)
  VRTi:i+31 ← Chop( prod, 32 )
```

For each vector element i from 0 to 3, do the following.

> Signed-integer halfword element i×2 in VRA is multiplied by signed-integer halfword element i×2 in VRB. The low-order 32 bits of the product are placed into halfword element i VRT.

Special Registers Altered:
None

Vector Multiply Even Unsigned Byte VX-form

vmuleub VRT,VRA,VRB

4	VRT	VRA	VRB	520
0	6	11	16	21 31

```
do i=0 to 127 by 16
  prod ← EXTZ((VRA)i:i+7) ×ui EXTZ((VRB)i:i+7)
  VRTi:i+15 ← Chop(prod, 16)
```

For each vector element i from 0 to 7, do the following.

> Unsigned-integer byte element i×2 in VRA is multiplied by unsigned-integer byte element i×2 in VRB. The low-order 16 bits of the product are placed into halfword element i VRT.

Special Registers Altered:
None

Vector Multiply Even Unsigned Halfword VX-form

vmuleuh VRT,VRA,VRB

4	VRT	VRA	VRB	584
0	6	11	16	21 31

```
do i=0 to 127 by 32
  prod ← EXTZ((VRA)i:i+15) ×ui EXTZ((VRB)i:i+15)
  VRTi:i+31 ← Chop(prod, 32)
```

For each vector element i from 0 to 3, do the following.

> Unsigned-integer halfword element i×2 in VRA is multiplied by unsigned-integer halfword element i×2 in VRB. The low-order 32 bits of the product are placed into halfword element i VRT.

Special Registers Altered:
None

Vector Multiply Odd Signed Byte VX-form

vmulosb VRT,VRA,VRB

4	VRT	VRA	VRB	264
0	6	11	16	21 31

```
do i=0 to 127 by 16
  prod ← EXTS((VRA)i+8:i+15) ×si EXTS((VRB)i+8:i+15)
  VRTi:i+15 ← Chop( prod, 16 )
```

For each vector element i from 0 to 7, do the following.

> Signed-integer byte element i×2+1 in VRA is multiplied by signed-integer byte element i×2+1 in VRB. The low-order 16 bits of the product are placed into halfword element i VRT.

Special Registers Altered:
None

Vector Multiply Odd Signed Halfword VX-form

vmulosh VRT,VRA,VRB

4	VRT	VRA	VRB	328
0	6	11	16	21 31

```
do i=0 to 127 by 32
  prod ← EXTS((VRA)i+16:i+31) ×si EXTS((VRB)i+16:i+31)
  VRTi:i+31 ← Chop( prod, 32 )
```

For each vector element i from 0 to 3, do the following.

> Signed-integer halfword element i×2+1 in VRA is multiplied by signed-integer halfword element i×2+1 in VRB. The low-order 32 bits of the product are placed into halfword element i VRT.

Special Registers Altered:
None

Vector Multiply Odd Unsigned Byte VX-form

vmuloub VRT,VRA,VRB

4	VRT	VRA	VRB	8
0	6	11	16	21 31

```
do i=0 to 127 by 16
  prod ← EXTZ((VRA)i+8:i+15) ×ui EXTZ((VRB)i+8:i+15)
  VRTi:i+15 ← Chop( prod, 16 )
```

For each vector element i from 0 to 7, do the following.

> Unsigned-integer byte element i×2+1 in VRA is multiplied by unsigned-integer byte element i×2+1 in VRB. The low-order 16 bits of the product are placed into halfword element i VRT.

Special Registers Altered:
None

Vector Multiply Odd Unsigned Halfword VX-form

vmulouh VRT,VRA,VRB

4	VRT	VRA	VRB	72
0	6	11	16	21 31

```
do i=0 to 127 by 32
  prod ← EXTZ((VRA)i+16:i+31)×ui EXTZ((VRB)i+16:i+31)
  VRTi:i+31 ← Chop( prod, 32 )
```

For each vector element i from 0 to 3, do the following.

> Unsigned-integer halfword element i×2+1 in VRA is multiplied by unsigned-integer halfword element i×2+1 in VRB. The low-order 32 bits of the product are placed into halfword element i VRT.

Special Registers Altered:
None

5.9.1.4 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

```
do i=0 to 127 by 16
   prod ← EXTS((VRA)i:i+15) ×si EXTS((VRB)i:i+15)
   sum  ← (prod >>si 15) +int EXTS((VRC)i:i+15
   VRTi:i+15 ← Clamp(sum, -2^15, 2^15-1)16:31
```

For each vector element i from 0 to 7, do the following.

Signed-integer halfword element i in VRA is multiplied by signed-integer halfword element i in VRB, producing a 32-bit signed-integer product. Bits 0:16 of the product are added to signed-integer halfword element i in VRC.

- If the intermediate result is greater than $2^{15}-1$ the result saturates to $2^{15}-1$.
- If the intermediate result is less than -2^{15} the result saturates to -2^{15}.

The low-order 16 bits of the result are placed into halfword element i of VRT.

Special Registers Altered:

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

```
do i=0 to 127 by 16
   prod ← EXTS((VRA)i:i+15) ×si EXTS((VRB)i:i+15)
   sum  ← ((prod +int 0x0000_4000) >>si 15)
           +int EXTS((VRC)i:i+15)
   VRTi:i+15 ← Clamp(sum, -2^15, 2^15-1)16:31
```

For each vector element i from 0 to 7, do the following.

Signed-integer halfword element i in VRA is multiplied by signed-integer halfword element i in VRB, producing a 32-bit signed-integer product. The value 0x0000_4000 is added to the product, producing a 32-bit signed-integer sum. Bits 0:16 of the sum are added to signed-integer halfword element i in VRC.

- If the intermediate result is greater than $2^{15}-1$ the result saturates to $2^{15}-1$.
- If the intermediate result is less than -2^{15} the result saturates to -2^{15}.

The low-order 16 bits of the result are placed into halfword element i of VRT.

Special Registers Altered:

SAT

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

```
do i=0 to 127 by 16
   prod ← EXTZ((VRA)i:i+15) ×ui EXTZ((VRB)i:i+15)
   sum  ← Chop( prod, 16 ) +int (VRC)i:i+15
   VRTi:i+15 ← Chop( sum, 16 )
```

For each vector element i from 0 to 3, do the following.

> Unsigned-integer halfword element i in VRA is multiplied by unsigned-integer halfword element i in VRB, producing a 32-bit unsigned-integer product. The low-order 16 bits of the product are added to unsigned-integer halfword element i in VRC.
>
> The low-order 16 bits of the sum are placed into halfword element i of VRT.

Special Registers Altered:
None

Programming Note

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

```
do i=0 to 127 by 32
   temp ← EXTZ((VRC)i:i+31)
   do j=0 to 31 by 8
     prod ← EXTZ((VRA)i+j:i+j+7)
            ×ui EXTZ((VRB)i+j:i+j+7)
     temp ← temp +int prod
   VRTi:i+31 ← Chop( temp, 32 )
```

For each word element in VRT the following operations are performed, in the order shown.

- Each of the four unsigned-integer byte elements contained in the corresponding word element of VRA is multiplied by the corresponding unsigned-integer byte element in VRB, producing an unsigned-integer halfword product.
- The sum of these four unsigned-integer halfword products is added to the unsigned-integer word element in VRC.
- The unsigned-integer word result is placed into the corresponding word element of VRT.

Special Registers Altered:
None

Vector Multiply-Sum Mixed Byte Modulo VA-form

vmsummbm VRT,VRA,VRB,VRC

4	VRT	VRA	VRB	VRC	37
0	6	11	16	21	26 31

```
do i=0 to 127 by 32
  temp ← (VRC)i:i+31
  do j=0 to 31 by 8
    prod0:15 ← (VRA)i+j:i+j+7 ×sui (VRB)i+j:i+j+7
    temp    ← temp +int EXTS(prod)
  VRTi:i+31 ← temp
```

For each word element in VRT the following operations are performed, in the order shown.

- Each of the four signed-integer byte elements contained in the corresponding word element of VRA is multiplied by the corresponding unsigned-integer byte element in VRB, producing a signed-integer product.
- The sum of these four signed-integer halfword products is added to the signed-integer word element in VRC.
- The signed-integer result is placed into the corresponding word element of VRT.

Special Registers Altered:
None

Vector Multiply-Sum Signed Halfword Modulo VA-form

vmsumshm VRT,VRA,VRB,VRC

4	VRT	VRA	VRB	VRC	40
0	6	11	16	21	26 31

```
do i=0 to 127 by 32
   temp ← (VRC)i:i+31
   do j=0 to 31 by 16
     prod0:31 ← (VRA)i+j:i+j+15 ×si (VRB)i+j:i+j+15
     temp     ← temp +int prod
   VRTi:i+31 ← temp
```

For each word element in VRT the following operations are performed, in the order shown.

- Each of the two signed-integer halfword elements contained in the corresponding word element of VRA is multiplied by the corresponding signed-integer halfword element in VRB, producing a signed-integer product.
- The sum of these two signed-integer word products is added to the signed-integer word element in VRC.
- The signed-integer word result is placed into the corresponding word element of VRT.

Special Registers Altered:
None

Vector Multiply-Sum Signed Halfword Saturate *VA-form*

vmsumshs VRT,VRA,VRB,VRC

4	VRT	VRA	VRB	VRC	41
0	6	11	16	21	26 31

```
do i=0 to 127 by 32
   temp ← EXTS((VRC)i:i+31)
   do j=0 to 31 by 16
      prod ← EXTS((VRA)i+j:i+j+15)
          ×si EXTS((VRB)i+j:i+j+15)
     temp ← temp +int prod
   VRTi:i+31 ← Clamp(temp, -2^31, 2^31-1)
```

For each word element in VRT the following operations are performed, in the order shown.

- Each of the two signed-integer halfword elements contained in the corresponding word element of VRA is multiplied by the corresponding signed-integer halfword element in VRB, producing a signed-integer product.
- The sum of these two signed-integer word products is added to the signed-integer word element in VRC.
- If the intermediate result is greater than 2^{31}-1 the result saturates to 2^{31}-1 and if it is less than -2^{31} it saturates to -2^{31}.
- The result is placed into the corresponding word element of VRT.

Special Registers Altered:
SAT

Vector Multiply-Sum Unsigned Halfword Modulo *VA-form*

vmsumuhm VRT,VRA,VRB,VRC

4	VRT	VRA	VRB	VRC	38
0	6	11	16	21	26 31

```
do i=0 to 127 by 32
   temp ← EXTZ((VRC)i:i+31)
   do j=0 to 31 by 16
      prod ← EXTZ((VRA)i+j:i+j+15)
             ×ui EXTZ((VRB)i+j:i+j+15)
     temp ← temp +int prod
   VRTi:i+31 ← Chop( temp, 32 )
```

For each word element in VRT the following operations are performed, in the order shown.

- Each of the two unsigned-integer halfword elements contained in the corresponding word element of VRA is multiplied by the corresponding unsigned-integer halfword element in VRB, producing an unsigned-integer word product.
- The sum of these two unsigned-integer word products is added to the unsigned-integer word element in VRC.
- The unsigned-integer result is placed into the corresponding word element of VRT.

Special Registers Altered:
None

Vector Multiply-Sum Unsigned Halfword Saturate VA-form

vmsumuhs VRT,VRA,VRB,VRC

4	VRT	VRA	VRB	VRC	39
0	6	11	16	21	26 31

```
do i=0 to 127 by 32
   temp ← EXTZ((VRC)i:i+31)
   do j=0 to 31 by 16
      prod ← EXTZ((VRA)i+j:i+j+15)
                  ×ui EXTZ((VRB)i+j:i+j+15)
   temp ← temp +int prod
   VRTi:i+31 ← Clamp(temp, 0, 2^32-1)
```

For each word element in VRT the following operations are performed, in the order shown.

- Each of the two unsigned-integer halfword elements contained in the corresponding word element of VRA is multiplied by the corresponding unsigned-integer halfword element in VRB, producing an unsigned-integer product.
- The sum of these two unsigned-integer word products is added to the unsigned-integer word element in VRC.
- If the intermediate result is greater than 2^{32}-1 the result saturates to 2^{32}-1.
- The result is placed into the corresponding word element of VRT.

Special Registers Altered:
SAT

5.9.1.5 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

```
temp ← EXTS((VRB)96:127)
do i=0 to 127 by 32
   temp ← temp +int EXTS((VRA)i:i+31)
VRT0:31   ← 0x0000_0000
VRT32:63  ← 0x0000_0000
VRT64:95  ← 0x0000_0000
VRT96:127 ← Clamp(temp, -2^31, 2^31-1)
```

The sum of the four signed-integer word elements in VRA is added to signed-integer word element 3 of VRB.

- If the intermediate result is greater than 2^{31}-1 the result saturates to 2^{31}-1.
- If the intermediate result is less than -2^{31} the result saturates to -2^{31}.

The low-end 32 bits of the result are placed into word element 3 of VRT.

Word elements 0 to 2 of VRT are set to 0.

Special Registers Altered:
SAT

Vector Sum across Half Signed Word Saturate VX-form

vsum2sws VRT,VRA,VRB

4	VRT	VRA	VRB	1672
0	6	11	16	21 31

```
do i=0 to 127 by 64
   temp ← EXTS((VRB)i+32:i+63)
   do j=0 to 63 by 32
      temp ← temp +int EXTS((VRA)i+j:i+j+31)
   VRTi:i+63 ← 0x0000_0000 || Clamp(temp, -2^31, 2^31-1)
```

Word elements 0 and 2 of VRT are set to 0.

The sum of the signed-integer word elements 0 and 1 in VRA is added to the signed-integer word element in bits 32:63 of VRB.

- If the intermediate result is greater than 2^{31}-1 the result saturates to 2^{31}-1.
- If the intermediate result is less than -2^{31} the result saturates to -2^{31}.

The low-order 32 bits of the result are placed into word element 1 of VRT.

The sum of signed-integer word elements 2 and 3 in VRA is added to the signed-integer word element in bits 96:127 of VRB.

- If the intermediate result is greater than 2^{31}-1 the result saturates to 2^{31}-1.
- If the intermediate result is less than -2^{31} the result saturates to -2^{31}.

The low-order 32 bits of the result are placed into word element 3 of VRT.

Special Registers Altered:
SAT

Vector Sum across Quarter Signed Byte Saturate VX-form

vsum4sbs VRT,VRA,VRB

4	VRT	VRA	VRB	1800
0	6	11	16	21 31

```
do i=0 to 127 by 32
   temp ← EXTS((VRB)i:i+31)
   do j=0 to 31 by 8
      temp ← temp +int EXTS((VRA)i+j:i+j+7)
   VRTi:i+31 ← Clamp(temp, -2^31, 2^31-1)
```

For each vector element i from 0 to 3, do the following.

The sum of the four signed-integer byte elements contained in word element i of VRA is added to signed-integer word element i in VRB.

- If the intermediate result is greater than 2^{31}-1 the result saturates to 2^{31}-1.
- If the intermediate result is less than -2^{31} the result saturates to -2^{31}.

The low-order 32 bits of the result are placed into word element i of VRT.

Special Registers Altered:
SAT

Vector Sum across Quarter Unsigned Byte Saturate VX-form

vsum4ubs VRT,VRA,VRB

4	VRT	VRA	VRB	1544
0	6	11	16	21 31

```
do i=0 to 127 by 32
   temp ← EXTZ((VRB)i:i+31)
   do j=0 to 31 by 8
      temp ← temp +int EXTZ((VRA)i+j:i+j+7)
   VRTi:i+31 ← Clamp( temp, 0, 2^32-1 )
```

For each vector element i from 0 to 3, do the following.

The sum of the four unsigned-integer byte elements contained in word element i of VRA is added to unsigned-integer word element i in VRB.

- If the intermediate result is greater than 2^{32}-1 it saturates to 2^{32}-1.

The low-order 32 bits of the result are placed into word element i of VRT.

Special Registers Altered:
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

```
do i=0 to 127 by 32
   temp ← EXTS((VRB)i:i+31)
   do j=0 to 31 by 16
      temp ← temp +int EXTS((VRA)i+j:i+j+15)
   VRTi:i+31 ← Clamp(temp, -2^31, 2^31-1)
```

For each vector element i from 0 to 3, do the following.

The sum of the two signed-integer halfword elements contained in word element i of VRA is added to signed-integer word element i in VRB.

- If the intermediate result is greater than 2^{31}-1 the result saturates to 2^{31}-1.
- If the intermediate result is less than -2^{31} the result saturates to -2^{31}.

The low-order 32 bits of the result are placed into the corresponding word element of VRT.

Special Registers Altered:
SAT

5.9.1.6 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

```
do i=0 to 127 by 8
   aop ← EXTS((VRA)i:i+7)
   bop ← EXTS((VRB)i:i+7)
   VRTi:i+7 ← Chop(( aop +int bop +int 1 ) >> 1, 8)
```

For each vector element *i* from 0 to 15, do the following.

Signed-integer byte element *i* in VRA is added to signed-integer byte element *i* in VRB. The sum is incremented by 1 and then shifted right 1 bit.

The low-order 8 bits of the result are placed into byte element *i* of VRT.

Special Registers Altered:
None

Vector Average Signed Word VX-form

vavgsw VRT,VRA,VRB

4	VRT	VRA	VRB	1410
0	6	11	16	21 31

```
do i=0 to 127 by 32
   aop ← EXTS((VRA)i:i+31)
   bop ← EXTS((VRB)i:i+31)
   VRTi:i+31 ← Chop(( aop +int bop +int 1 ) >> 1, 32)
```

For each vector element *i* from 0 to 3, do the following.

Signed-integer word element *i* in VRA is added to signed-integer word element *i* in VRB. The sum is incremented by 1 and then shifted right 1 bit.

The low-order 32 bits of the result are placed into word element *i* of VRT.

Special Registers Altered:
None

Vector Average Signed Halfword VX-form

vavgsh VRT,VRA,VRB

4	VRT	VRA	VRB	1346
0	6	11	16	21 31

```
do i=0 to 127 by 16
   aop ← EXTS((VRA)i:i+15)
   bop ← EXTS((VRB)i:i+15)
   VRTi:i+15 ← Chop(( aop +int bop +int 1 ) >> 1, 16)
```

For each vector element *i* from 0 to 7, do the following.

Signed-integer halfword element *i* in VRA is added to signed-integer halfword element *i* in VRB. The sum is incremented by 1 and then shifted right 1 bit.

The low-order 16 bits of the result are placed into halfword element *i* of VRT.

Special Registers Altered:
None

Vector Average Unsigned Byte VX-form

vavgub VRT,VRA,VRB

4	VRT	VRA	VRB	1026
0	6	11	16	21 31

```
do i=0 to 127 by 8
   aop ← EXTZ((VRA)i:i+7)
   bop ← EXTZ((VRB)i:i+7)
   VRTi:i+7 ← Chop((aop +int bop +int 1) >>ui 1, 8)
```

For each vector element *i* from 0 to 15, do the following.

Unsigned-integer byte element *i* in VRA is added to unsigned-integer byte element *i* in VRB. The sum is incremented by 1 and then shifted right 1 bit.

The low-order 8 bits of the result are placed into byte element *i* of VRT.

Special Registers Altered:
None

Vector Average Unsigned Halfword VX-form

vavguh VRT,VRA,VRB

4	VRT	VRA	VRB	1090
0	6	11	16	21 31

```
do i=0 to 127 by 16
   aop ← EXTZ((VRA)i:i+15)
   bop ← EXTZ((VRB)i:i+15)
   VRTi:i+15 ← Chop((aop +int bop +int 1) >>ui 1, 16)
```

For each vector element *i* from 0 to 7, do the following.

Unsigned-integer halfword element *i* in VRA is added to unsigned-integer halfword element *i* in VRB. The sum is incremented by 1 and then shifted right 1 bit.

The low-order 16 bits of the result are placed into halfword element *i* of VRT.

Special Registers Altered:
None

Vector Average Unsigned Word VX-form

vavguw VRT,VRA,VRB

4	VRT	VRA	VRB	1154
0	6	11	16	21 31

```
do i=0 to 127 by 32
   aop ← EXTZ((VRA)i:i+31)
   bop ← EXTZ((VRB)i:i+31)
   VRTi:i+31 ← Chop((aop +int bop +int 1) >>ui 1, 32)
```

For each vector element *i* from 0 to 3, do the following.

Unsigned-integer word element *i* in VRA is added to unsigned-integer word element *i* in VRB. The sum is incremented by 1 and then shifted right 1 bit.

The low-order 32 bits of the result are placed into word element *i* of VRT.

Special Registers Altered:
None

5.9.1.7 Vector Integer Maximum and Minimum Instructions

Vector Maximum Signed Byte VX-form

vmaxsb VRT,VRA,VRB

4	VRT	VRA	VRB	258
0	6	11	16	21 31

```
do i=0 to 127 by 8
   aop ← EXTS((VRA)i:i+7)
   bop ← EXTS((VRB)i:i+7)
   VRTi:i+7 ← ( aop >si bop )
         ? (VRA)i:i+7 : (VRB)i:i+7
```

For each vector element *i* from 0 to 15, do the following.

Signed-integer byte element *i* in VRA is compared to signed-integer byte element *i* in VRB. The larger of the two values is placed into byte element *i* of VRT.

Special Registers Altered:
None

Vector Maximum Signed Word VX-form

vmaxsw VRT,VRA,VRB

4	VRT	VRA	VRB	386
0	6	11	16	21 31

```
do i=0 to 127 by 32
   aop ← EXTS((VRA)i:i+31)
   bop ← EXTS((VRB)i:i+31)
   VRTi:i+31 ← ( aop >si bop )
         ? (VRA)i:i+31 : (VRB)i:i+31
```

For each vector element *i* from 0 to 3, do the following.

Signed-integer word element *i* in VRA is compared to signed-integer word element *i* in VRB. The larger of the two values is placed into word element *i* of VRT.

Special Registers Altered:
None

Vector Maximum Signed Halfword VX-form

vmaxsh VRT,VRA,VRB

4	VRT	VRA	VRB	322
0	6	11	16	21 31

```
do i=0 to 127 by 16
   aop ← EXTS((VRA)i:i+15)
   bop ← EXTS((VRB)i:i+15
   VRTi:i+15 ← ( aop >si bop )
         ? (VRA)i:i+15 : (VRB)i:i+15
```

For each vector element *i* from 0 to 7, do the following.

Signed-integer halfword element *i* in VRA is compared to signed-integer halfword element *i* in VRB. The larger of the two values is placed into halfword element *i* of VRT.

Special Registers Altered:
None

Vector Maximum Unsigned Byte VX-form

vmaxub VRT,VRA,VRB

4	VRT	VRA	VRB	2
0	6	11	16	21 31

```
do i=0 to 127 by 8
   aop ← EXTZ((VRA)i:i+7)
   bop ← EXTZ((VRB)i:i+7)
   VRTi:i+7 ← (aop >ui bop) ? (VRA)i:i+7 : (VRB)i:i+7
```

For each vector element *i* from 0 to 15, do the following.

Unsigned-integer byte element *i* in VRA is compared to unsigned-integer byte element *i* in VRB. The larger of the two values is placed into byte element *i* of VRT.

Special Registers Altered:
None

Vector Maximum Unsigned Word VX-form

vmaxuw VRT,VRA,VRB

4	VRT	VRA	VRB	130
0	6	11	16	21 31

```
do i=0 to 127 by 32
   aop ← EXTZ((VRA)i:i+31)
   bop ← EXTZ((VRB)i:i+31)
   VRTi:i+31 ← (aop >ui bop)
         ? (VRA)i:i+31 : (VRB)i:i+31
```

For each vector element *i* from 0 to 3, do the following.

Unsigned-integer word element *i* in VRA is compared to unsigned-integer word element *i* in VRB. The larger of the two values is placed into word element *i* of VRT.

Special Registers Altered:
None

Vector Maximum Unsigned Halfword VX-form

vmaxuh VRT,VRA,VRB

4	VRT	VRA	VRB	66
0	6	11	16	21 31

```
do i=0 to 127 by 16
   aop ← EXTZ((VRA)i:i+15)
   bop ← EXTZ((VRB)i:i+15)
   VRTi:i+15 ← (aop >ui bop)
         ? (VRA)i:i+15 : (VRB)i:i+15
```

For each vector element *i* from 0 to 7, do the following.

Unsigned-integer halfword element *i* in VRA is compared to unsigned-integer halfword element *i* in VRB. The larger of the two values is placed into halfword element *i* of VRT.

Special Registers Altered:
None

Vector Minimum Signed Byte VX-form

vminsb VRT,VRA,VRB

4	VRT	VRA	VRB	770
0	6	11	16	21 31

```
do i=0 to 127 by 8
   aop ← EXTS((VRA)i:i+7)
   bop ← EXTS((VRB)i:i+7)
   VRTi:i+7 ← (aop <si bop) ? (VRA)i:i+7 : (VRB)i:i+7
```

For each vector element *i* from 0 to 15, do the following.

Signed-integer byte element *i* in VRA is compared to signed-integer byte element *i* in VRB. The smaller of the two values is placed into byte element *i* of VRT.

Special Registers Altered:
None

Vector Minimum Signed Halfword VX-form

vminsh VRT,VRA,VRB

4	VRT	VRA	VRB	834
0	6	11	16	21 31

```
do i=0 to 127 by 16
   aop ← EXTS((VRA)i:i+15)
   bop ← EXTS((VRB)i:i+15)
   VRTi:i+15 ← ( aop <si bop )
         ? (VRA)i:i+15 : (VRB)i:i+15
```

For each vector element *i* from 0 to 7, do the following.

Signed-integer halfword element *i* in VRA is compared to signed-integer halfword element *i* in VRB. The smaller of the two values is placed into halfword element *i* of VRT.

Special Registers Altered:
None

Vector Minimum Signed Word VX-form

vminsw VRT,VRA,VRB

4	VRT	VRA	VRB	898
0	6	11	16	21 31

```
do i=0 to 127 by 32
   aop ← EXTS((VRA)i:i+31)
   bop ← EXTS((VRB)i:i+31)
   VRTi:i+31 ← ( aop <si bop )
          ? (VRA)i:i+31 : (VRB)i:i+31
```

For each vector element *i* from 0 to 3, do the following.

Signed-integer word element *i* in VRA is compared to signed-integer word element *i* in VRB. The smaller of the two values is placed into word element *i* of VRT.

Special Registers Altered:
None

Vector Minimum Unsigned Byte VX-form

vminub VRT,VRA,VRB

4	VRT	VRA	VRB	514
0	6	11	16	21 31

```
do i=0 to 127 by 8
   aop ← EXTZ((VRA)i:i+7)
   bop ← EXTZ((VRB)i:i+7
   VRTi:i+7 ← ( aop <ui bop )
         ? (VRA)i:i+7 : (VRB)i:i+7
```

For each vector element *i* from 0 to 15, do the following.

Unsigned-integer byte element *i* in VRA is compared to unsigned-integer byte element *i* in VRB. The smaller of the two values is placed into byte element *i* of VRT.

Special Registers Altered:
None

Vector Minimum Unsigned Word VX-form

vminuw VRT,VRA,VRB

4	VRT	VRA	VRB	642
0	6	11	16	21 31

```
do i=0 to 127 by 32
   aop ← EXTZ((VRA)i:i+31)
   bop ← EXTZ((VRB)i:i+31)
   VRTi:i+31 ← ( aop <ui bop )
         ? (VRA)i:i+31 : (VRB)i:i+31
```

For each vector element *i* from 0 to 3, do the following.

Unsigned-integer word element *i* in VRA is compared to unsigned-integer word element *i* in VRB. The smaller of the two values is placed into word element *i* of VRT.

Special Registers Altered:
None

Vector Minimum Unsigned Halfword VX-form

vminuh VRT,VRA,VRB

4	VRT	VRA	VRB	578
0	6	11	16	21 31

```
do i=0 to 127 by 16
   aop ← EXTZ((VRA)i:i+15)
   bop ← EXTZ((VRB)i:i+15)
   VRTi:i+15 ← ( aop <ui bop )
         ? (VRA)i:i+15 : (VRB)i:i+15
```

For each vector element *i* from 0 to 7, do the following.

Unsigned-integer halfword element *i* in VRA is compared to unsigned-integer halfword element *i* in VRB. The smaller of the two values is placed into halfword element *i* of VRT.

Special Registers Altered:
None

5.9.2 Vector Integer Compare Instructions

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

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

Bit	Description
0	The relation is true for all element pairs (i.e., VRT is set to all 1s)
1	0
2	The relation is false for all element pairs (i.e., VRT is set to all 0s)
3	0

Programming Note

***vcmpequb*[.]**, ***vcmpequh*[.]** and ***vcmpequw*[.]** can be used for unsigned or signed-integers.

Vector Compare Equal To Unsigned Byte VC-form

```
vcmpequb   VRT,VRA,VRB     (Rc=0)
vcmpequb.  VRT,VRA,VRB     (Rc=1)
```

4	VRT	VRA	VRB	Rc	6
0	6	11	16	21	22 31

```
do i=0 to 127 by 8
   VRT_{i:i+7} ← ((VRA)_{i:i+7} =_int (VRB)_{i:i+7}) ? ^8 1 : ^8 0
if Rc=1 then do
   t ← (VRT=^128 1)
   f ← (VRT=^128 0)
   CR6 ← t || 0b0 || f || 0b0
```

For each vector element *i* from 0 to 15, do the following.

Unsigned-integer byte element *i* in VRA is compared to unsigned-integer byte element *i* in VRB. Byte element *i* in VRT is set to all 1s if unsigned-integer byte element *i* in VRA is equal to unsigned-integer byte element *i* in VRB, and is set to all 0s otherwise.

Special Registers Altered:

CR6 (if Rc=1)

Vector Compare Equal To 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

```
do i=0 to 127 by 16
   VRT_{i:i+15} ←((VRA)_{i:i+15} =_int (VRB)_{i:i+15}) ? ^16 1 : ^16 0
if Rc=1 then do
   t ← (VRT=^128 1)
   f ← (VRT=^128 0)
   CR6 ← t || 0b0 || f || 0b0
```

For each vector element *i* from 0 to 7, do the following.

Unsigned-integer halfword element *i* in VRA is compared to unsigned-integer halfword element element *i* in VRB. Halfword element *i* in VRT is set to all 1s if unsigned-integer halfword element *i* in VRA is equal to unsigned-integer halfword element *i* in VRB, and is set to all 0s otherwise.

Special Registers Altered:

CR6 (if Rc=1)

Vector Compare Equal To 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

```
do i=0 to 127 by 32
   VRT_{i:i+31} ←((VRA)_{i:i+31} =_int (VRB)_{i:i+31}) ? ^32 1 : ^32 0
if Rc=1 then do
   t ← (VRT=^128 1)
   f ← (VRT=^128 0)
   CR6 ← t || 0b0 || f || 0b0
```

For each vector element *i* from 0 to 3, do the following.

> Unsigned-integer word element *i* in VRA is compared to unsigned-integer word element *i* in VRB. Word element *i* in VRT is set to all 1s if unsigned-integer word element *i* in VRA is equal to unsigned-integer word element *i* in VRB, and is set to all 0s otherwise.

Special Registers Altered:

CR6 (if Rc=1)

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

```
do i=0 to 127 by 8
   VRT_{i:i+7} ← ((VRA)_{i:i+7} >_si (VRB)_{i:i+7}) ? ^8 1 : ^8 0
if Rc=1 then do
   t ← (VRT=^128 1)
   f ← (VRT=^128 0)
   CR6 ← t || 0b0 || f || 0b0
```

For each vector element *i* from 0 to 15, do the following.

> Signed-integer byte element *i* in VRA is compared to signed-integer byte element *i* in VRB. Byte element *i* in VRT is set to all 1s if signed-integer byte element *i* in VRA is greater than to signed-integer byte element *i* in VRB, and is set to all 0s otherwise.

Special Registers Altered:

CR6 (if Rc=1)

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

```
do i=0 to 127 by 16
   VRT_{i:i+15} ← ((VRA)_{i:i+15} >_si (VRB)_{i:i+15}) ? ^16 1 : ^16 0
if Rc=1 then do
   t ← (VRT=^128 1)
   f ← (VRT=^128 0)
   CR6 ← t || 0b0 || f || 0b0
```

For each vector element *i* from 0 to 7, do the following.

> Signed-integer halfword element *i* in VRA is compared to signed-integer halfword element *i* in VRB. Halfword element *i* in VRT is set to all 1s if signed-integer halfword element *i* in VRA is greater than signed-integer halfword element *i* in VRB, and is set to all 0s otherwise.

Special Registers Altered:

CR6 (if Rc=1)

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

```
do i=0 to 127 by 32
   VRT_{i:i+31} ← ((VRA)_{i:i+31} >_si (VRB)_{i:i+31}) ? ^32 1 : ^32 0
if Rc=1 then do
   t ← (VRT=^128 1)
   f ← (VRT=^128 0)
   CR6 ← t || 0b0 || f || 0b0
```

For each vector element *i* from 0 to 3, do the following.

> Signed-integer word element *i* in VRA is compared to signed-integer word element *i* in VRB. Word element *i* in VRT is set to all 1s if signed-integer word element *i* in VRA is greater than signed-integer word element *i* in VRB, and is set to all 0s otherwise.

Special Registers Altered:

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

```
do i=0 to 127 by 8
   VRT_{i:i+7} ← ((VRA)_{i:i+7} >ui (VRB)_{i:i+7}) ? ^8 1 : ^8 0
if Rc=1 then do
   t ← (VRT=^128 1)
   f ← (VRT=^128 0)
   CR6 ← t || 0b0 || f || 0b0
```

For each vector element *i* from 0 to 15, do the following.

> Unsigned-integer byte element *i* in VRA is compared to unsigned-integer byte element *i* in VRB. Byte element *i* in VRT is set to all 1s if unsigned-integer byte element *i* in VRA is greater than to unsigned-integer byte element *i* in VRB, and is set to all 0s otherwise.

Special Registers Altered:

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

```
do i=0 to 127 by 16
   VRT_{i:i+15} ← ((VRA)_{i:i+15} >ui (VRB)_{i:i+15}) ? ^16 1 : ^16 0
if Rc=1 then do
   t ← (VRT=^128 1)
   f ← (VRT=^128 0)
   CR6 ← t || 0b0 || f || 0b0
```

For each vector element *i* from 0 to 7, do the following.

> Unsigned-integer halfword element *i* in VRA is compared to unsigned-integer halfword element *i* in VRB. Halfword element *i* in VRT is set to all 1s if unsigned-integer halfword element *i* in VRA is greater than to unsigned-integer halfword element *i* in VRB, and is set to all 0s otherwise.

Special Registers Altered:

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

```
do i=0 to 127 by 32
   VRT_{i:i+31} ← ((VRA)_{i:i+31} >ui (VRB)_{i:i+31}) ? ^32 1 : ^32 0
if Rc=1 then do
   t ← (VRT=^128 1)
   f ← (VRT=^128 0)
   CR6 ← t || 0b0 || f || 0b0
```

For each vector element *i* from 0 to 3, do the following.

> Unsigned-integer word element *i* in VRA is compared to unsigned-integer word element *i* in VRB. Word element *i* in VRT is set to all 1s if unsigned-integer word element *i* in VRA is greater than to unsigned-integer word element *i* in VRB, and is set to all 0s otherwise.

Special Registers Altered:

CR6 (if Rc=1)

5.9.3 Vector Logical Instructions

Extended mnemonics for vector logical operations

Extended mnemonics are provided that use the Vector OR and Vector NOR instructions to copy the contents of one Vector Register to another, with and without complementing. These are shown as examples with the two instructions.

Vector Move Register

Several vector instructions can be coded in a way such that they simply copy the contents of one Vector Register to another. An extended mnemonic is provided to convey the idea that no computation is being performed but merely data movement (from one register to another).

The following instruction copies the contents of register Vy to register Vx.

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 Vector Register and places the result into another Vector Register. An extended mnemonic is provided that allows this operation to be coded easily.

The following instruction complements the contents of register Vy and places the result into register Vx.

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

```
VRT ← (VRA) & (VRB)
```

The contents of VRA are ANDed with the contents of VRB and the result is placed into VRT.

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

```
VRT ← (VRA) & ¬(VRB)
```

The contents of VRA are ANDed with the complement of the contents of VRB and the result is placed into VRT.

Special Registers Altered:
None

Vector Logical NOR *VX-form*

vnor VRT,VRA,VRB

4	VRT	VRA	VRB	1284
0	6	11	16	21 31

```
VRT ← ¬( (VRA) | (VRB) )
```

The contents of VRA are ORed with the contents of VRB and the complemented result is placed into VRT.

Special Registers Altered:
None

Vector Logical OR *VX-form*

vor VRT,VRA,VRB

4	VRT	VRA	VRB	1156
0	6	11	16	21 31

```
VRT ← (VRA) | (VRB)
```

The contents of VRA are ORed with the contents of VRB and the result is placed into VRT.

Special Registers Altered:
None

Vector Logical XOR *VX-form*

vxor VRT,VRA,VRB

4	VRT	VRA	VRB	1220
0	6	11	16	21 31

```
VRT ← (VRA) ⊕ (VRB)
```

The contents of VRA are XORed with the contents of VRB and the result is placed into VRT.

Special Registers Altered:
None

5.9.4 Vector Integer Rotate and Shift Instructions

Vector Rotate Left Byte *VX-form*

vrlb VRT,VRA,VRB

4	VRT	VRA	VRB	4
0	6	11	16	21 31

```
do i=0 to 127 by 8
   sh ← (VRB)i+5:i+7
   VRTi:i+7 ← (VRA)i:i+7 <<< sh
```

For each vector element *i* from 0 to 15, do the following.

Byte element *i* in VRA is rotated left by the number of bits specified in the low-order 3 bits of the corresponding byte element *i* in VRB.

The result is placed into byte element *i* in VRT.

Special Registers Altered:
None

Vector Rotate Left Halfword *VX-form*

vrlh VRT,VRA,VRB

4	VRT	VRA	VRB	68
0	6	11	16	21 31

```
do i=0 to 127 by 16
   sh ← (VRB)i+12:i+15
   VRTi:i+15 ← (VRA)i:i+15 <<< sh
```

For each vector element *i* from 0 to 7, do the following.

Halfword element *i* in VRA is rotated left by the number of bits specified in the low-order 4 bits of the corresponding halfword element *i* in VRB.

The result is placed into halfword element *i* in VRT.

Special Registers Altered:
None

Vector Rotate Left Word *VX-form*

vrlw VRT,VRA,VRB

4	VRT	VRA	VRB	132
0	6	11	16	21 31

```
do i=0 to 127 by 32
   sh ← (VRB)i+27:i+31
   VRTi:i+31 ← (VRA)i:i+31 <<< sh
```

For each vector element *i* from 0 to 3, do the following.

Word element *i* in VRA is rotated left by the number of bits specified in the low-order 5 bits of the corresponding word element *i* in VRB.

The result is placed into word element *i* in VRT.

Special Registers Altered:
None

Vector Shift Left Byte — *VX-form*

vslb VRT,VRA,VRB

4	VRT	VRA	VRB	260
0	6	11	16	21 31

```
do i=0 to 127 by 8
   sh ← (VRB)i+5:i+7
   VRTi:i+7 ← (VRA)i:i+7 << sh
```

For each vector element *i* from 0 to 15, do the following.

Byte element *i* in VRA is shifted left by the number of bits specified in the low-order 3 bits of byte element *i* in VRB.

- Bits shifted out of bit 0 are lost.
- Zeros are supplied to the vacated bits on the right.

The result is placed into byte element *i* of VRT.

Special Registers Altered:
None

Vector Shift Left Word — *VX-form*

vslw VRT,VRA,VRB

4	VRT	VRA	VRB	388
0	6	11	16	21 31

```
do i=0 to 127 by 32
   sh ← (VRB)i+27:i+31
   VRTi:i+31 ← (VRA)i:i+31 << sh
```

For each vector element *i* from 0 to 3, do the following.

Word element *i* in VRA is shifted left by the number of bits specified in the low-order 5 bits of word element *i* in VRB.

- Bits shifted out of bit 0 are lost.
- Zeros are supplied to the vacated bits on the right.

The result is placed into word element *i* of VRT.

Special Registers Altered:
None

Vector Shift Left Halfword — *VX-form*

vslh VRT,VRA,VRB

4	VRT	VRA	VRB	324
0	6	11	16	21 31

```
do i=0 to 127 by 16
   sh ← (VRB)i+12:i+15
   VRTi:i+15 ← (VRA)i:i+15 << sh
```

For each vector element *i* from 0 to 7, do the following.

Halfword element *i* in VRA is shifted left by the number of bits specified in the low-order 4 bits of halfword element *i* in VRB.

- Bits shifted out of bit 0 are lost.
- Zeros are supplied to the vacated bits on the right.

The result is placed into halfword element *i* of VRT.

Special Registers Altered:
None

Vector Shift Right Byte *VX-form*

vsrb VRT,VRA,VRB

4	VRT	VRA	VRB	516
0	6	11	16	21 31

```
do i=0 to 127 by 8
   sh ← (VRB)i+5:i+7
   VRTi:i+7 ← (VRA)i:i+7 >>ui sh
```

For each vector element *i* from 0 to 15, do the following.

Byte element *i* in VRA is shifted right by the number of bits specified in the low-order 3 bits of byte element *i* in VRB. Bits shifted out of the least-significant bit are lost. Zeros are supplied to the vacated bits on the left. The result is placed into byte element *i* of VRT.

Special Registers Altered:
None

Vector Shift Right Halfword *VX-form*

vsrh VRT,VRA,VRB

4	VRT	VRA	VRB	580
0	6	11	16	21 31

```
do i=0 to 127 by 16
   sh ← (VRB)i+12:i+15
   VRTi:i+15 ← (VRA)i:i+15 >>ui sh
```

For each vector element *i* from 0 to 7, do the following.

Halfword element *i* in VRA is shifted right by the number of bits specified in the low-order 4 bits of halfword element *i* in VRB. Bits shifted out of the least-significant bit are lost. Zeros are supplied to the vacated bits on the left. The result is placed into halfword element *i* of VRT.

Special Registers Altered:
None

Vector Shift Right Word *VX-form*

vsrw VRT,VRA,VRB

4	VRT	VRA	VRB	644
0	6	11	16	21 31

```
do i=0 to 127 by 32
   sh ← (VRB)i+27:i+31
   VRTi:i+31 ← (VRA)i:i+31 >>ui sh
```

For each vector element *i* from 0 to 3, do the following.

Word element *i* in VRA is shifted right by the number of bits specified in the low-order 5 bits of word element *i* in VRB. Bits shifted out of the least-significant bit are lost. Zeros are supplied to the vacated bits on the left. The result is placed into word element *i* of VRT.

Special Registers Altered:
None

Vector Shift Right Algebraic Byte VX-form

vsrab VRT,VRA,VRB

4	VRT	VRA	VRB	772
0	6	11	16	21 31

```
do i=0 to 127 by 8
   sh ← (VRB)i+5:i+7
   VRTi:i+7 ← (VRA)i:i+7 >>si sh
```

For each vector element *i* from 0 to 15, do the following.

Byte element *i* in VRA is shifted right by the number of bits specified in the low-order 3 bits of the corresponding byte element *i* in VRB. Bits shifted out of bit 7 of the byte element are lost. Bit 0 of the byte element is replicated to fill the vacated bits on the left. The result is placed into byte element *i* of VRT.

Special Registers Altered:

None

Vector Shift Right Algebraic Word VX-form

vsraw VRT,VRA,VRB

4	VRT	VRA	VRB	900
0	6	11	16	21 31

```
do i=0 to 127 by 32
   sh ← (VRB)i+27:i+31
   VRTi:i+31 ← (VRA)i:i+31 >>si sh
```

For each vector element *i* from 0 to 3, do the following.

Word element *i* in VRA is shifted right by the number of bits specified in the low-order 5 bits of the corresponding word element *i* in VRB. Bits shifted out of bit 31 of the word are lost. Bit 0 of the word is replicated to fill the vacated bits on the left. The result is placed into word element *i* of VRT.

Special Registers Altered:

None

Vector Shift Right Algebraic Halfword VX-form

vsrah VRT,VRA,VRB

4	VRT	VRA	VRB	836
0	6	11	16	21 31

```
do i=0 to 127 by 16
   sh ← (VRB)i+12:i+15
   VRTi:i+15 ← (VRA)i:i+15 >>si sh
```

For each vector element *i* from 0 to 7, do the following.

Halfword element *i* in VRA is shifted right by the number of bits specified in the low-order 4 bits of the corresponding halfword element *i* in VRB. Bits shifted out of bit 15 of the halfword are lost. Bit 0 of the halfword is replicated to fill the vacated bits on the left. The result is placed into halfword element *i* of VRT.

Special Registers Altered:

None

5.10 Vector Floating-Point Instruction Set

5.10.1 Vector Floating-Point Arithmetic Instructions

Vector Add Single-Precision VX-form

vaddfp VRT,VRA,VRB

4	VRT	VRA	VRB	10
0	6	11	16	21 31

```
do i=0 to 127 by 32
   VRT_{i:i+31} ←
         RoundToNearSP((VRA)_{i:i+31} +_fp (VRB)_{i:i+31})
```

For each vector element *i* from 0 to 3, do the following.

Single-precision floating-point element *i* in VRA is added to single-precision floating-point element *i* in VRB. The intermediate result is rounded to the nearest single-precision floating-point number and placed into word element *i* of VRT.

Special Registers Altered:
None

Vector Subtract Single-Precision VX-form

vsubfp VRT,VRA,VRB

4	VRT	VRA	VRB	74
0	6	11	16	21 31

```
do i=0 to 127 by 32
   VRT_{i:i+31} ←
         RoundToNearSP((VRA)_{i:i+31} -_fp (VRB)_{i:i+31})
```

For each vector element *i* from 0 to 3, do the following.

Single-precision floating-point element *i* in VRB is subtracted from single-precision floating-point element *i* in VRA. The intermediate result is rounded to the nearest single-precision floating-point number and placed into word element *i* of VRT.

Special Registers Altered:
None

Vector Multiply-Add Single-Precision VA-form

vmaddfp VRT,VRA,VRC,VRB

4	VRT	VRA	VRB	VRC	46
0	6	11	16	21	26 31

```
do i=0 to 127 by 32
   prod ← (VRA)i:i+31 ×fp (VRC)i:i+31
   VRTi:i+31  ← RoundToNearSP( prod +fp (VRB)i:i+31 )
```

For each vector element *i* from 0 to 3, do the following.

> Single-precision floating-point element *i* in VRA is multiplied by single-precision floating-point element *i* in VRC. Single-precision floating-point element *i* in VRB is added to the infinitely-precise product. The intermediate result is rounded to the nearest single-precision floating-point number and placed into word element *i* of VRT.

Special Registers Altered:
None

Programming Note

To use a multiply-add to perform an IEEE or Java compliant multiply, the addend must be -0.0. This is necessary to insure that the sign of a zero result will be correct when the product is -0.0 (+0.0 + -0.0 ≥ +0.0, and -0.0 + -0.0≥ -0.0). When the sign of a resulting 0.0 is not important, then +0.0 can be used as an addend which may, in some cases, avoid the need for a second register to hold a -0.0 in addition to the integer 0/floating-point +0.0 that may already be available.

Vector Negative Multiply-Subtract Single-Precision VA-form

vnmsubfp VRT,VRA,VRC,VRB

4	VRT	VRA	VRB	VRC	47
0	6	11	16	21	26 31

```
do i=0 to 127 by 32
   prod0:inf  ← (VRA)i:i+31 ×fp (VRC)i:i+31
   VRTi:i+31 ←
          -RoundToNearSP(prod0:inf -fp (VRB)i:i+31)
```

For each vector element *i* from 0 to 3, do the following.

> Single-precision floating-point element *i* in VRA is multiplied by single-precision floating-point element *i* in VRC. Single-precision floating-point element *i* in VRB is subtracted from the infinitely-precise product. The intermediate result is rounded to the nearest single-precision floating-point number, then negated and placed into word element *i* of VRT.

Special Registers Altered:
None

5.10.2 Vector Floating-Point Maximum and Minimum Instructions

Vector Maximum Single-Precision VX-form

vmaxfp VRT,VRA,VRB

4	VRT	VRA	VRB	1034
0	6	11	16	21 31

```
do i=0 to 127 by 32
   VRT_{i:i+31} ← ( (VRA)_{i:i+31} >_fp (VRB)_{i:i+31} )
          ? (VRA)_{i:i+31} : (VRB)_{i:i+31}
```

For each vector element *i* from 0 to 3, do the following.

> Single-precision floating-point element *i* in VRA is compared to single-precision floating-point element *i* in VRB. The larger of the two values is placed into word element *i* of VRT.

The maximum of +0 and -0 is +0. The maximum of any value and a NaN is a QNaN.

Special Registers Altered:
None

Vector Minimum Single-Precision VX-form

vminfp VRT,VRA,VRB

4	VRT	VRA	VRB	1098
0	6	11	16	21 31

```
do i=0 to 127 by 32
   VRT_{i:i+31} ← ( (VRA)_{i:i+31} <_fp (VRB)_{i:i+31} )
          ? (VRA)_{i:i+31} : (VRB)_{i:i+31}
```

For each vector element *i* from 0 to 3, do the following.

> Single-precision floating-point element *i* in VRA is compared to single-precision floating-point element *i* in VRB. The smaller of the two values is placed into word element *i* of VRT.

The minimum of +0 and -0 is -0. The minimum of any value and a NaN is a QNaN.

Special Registers Altered:
None

5.10.3 Vector Floating-Point Rounding and Conversion Instructions

See Appendix B, “Vector RTL Functions [Category: Vector]” on page 309, for RTL function descriptions.

Vector Convert to Signed Fixed-Point Word Saturate VX-form

vctsxs VRT,VRB,UIM

4	VRT	UIM	VRB	970
0	6	11	16	21 31

```
do i=0 to 127 by 32
   VRT_{i:i+31} ←
          ConvertSPtoSXWsaturate((VRB)_{i:i+31}, UIM)
```

For each vector element *i* from 0 to 3, do the following.

Single-precision floating-point word element *i* in VRB is multiplied by 2^{UIM}. The product is converted to a 32-bit signed fixed-point integer using the rounding mode Round toward Zero.

- If the intermediate result is greater than $2^{31}-1$ the result saturates to $2^{31}-1$.
- If the intermediate result is less than -2^{31} the result saturates to -2^{31}.

The result is placed into word element *i* of VRT.

Special Registers Altered:
SAT

Extended Mnemonics:

Example of an extended mnemonics for *Vector Convert to Signed Fixed-Point Word Saturate*:

Extended:	**Equivalent to:**
vcfpsxws VRT,VRB,UIM	vctsxs VRT,VRB,UIM

Vector Convert to Unsigned Fixed-Point Word Saturate VX-form

vctuxs VRT,VRB,UIM

4	VRT	UIM	VRB	906
0	6	11	16	21 31

```
do i=0 to 127 by 32
   VRT_{i:i+31} ←
          ConvertSPtoUXWsaturate((VRB)_{i:i+31}, UIM)
```

For each vector element *i* from 0 to 3, do the following.

Single-precision floating-point word element *i* in VRB is multiplied by 2^{UIM}. The product is converted to a 32-bit unsigned fixed-point integer using the rounding mode Round toward Zero.

- If the intermediate result is greater than $2^{32}-1$ the result saturates to $2^{32}-1$.

The result is placed into word element *i* of VRT.

Special Registers Altered:
SAT

Extended Mnemonics:

Example of an extended mnemonics for *Vector Convert to Unsigned Fixed-Point Word Saturate*:

Extended:	**Equivalent to:**
vcfpuxws VRT,VRB,UIM	vctuxs VRT,VRB,UIM

Vector Convert from Signed Fixed-Point Word VX-form

vcfsx VRT,VRB,UIM

4	VRT	UIM	VRB	842
0	6	11	16	21 31

```
do i=0 to 127 by 32
   VRT_{i:i+31} ←
          ConvertSXWtoSP( (VRB)_{i:i+31} ) ÷fp 2^UIM
```

For each vector element *i* from 0 to 3, do the following.

> Signed fixed-point word element *i* in VRB is converted to the nearest single-precision floating-point value. Each result is divided by 2^{UIM} and placed into word element *i* of VRT.

Special Registers Altered:
None

Extended Mnemonics:

Examples of extended mnemonics for *Vector Convert from Signed Fixed-Point Word*

Extended:		Equivalent to:	
vcsxwfp	VRT,VRB,UIM	vcfsx	VRT,VRB,UIM

Vector Convert from Unsigned Fixed-Point Word VX-form

vcfux VRT,VRB,UIM

4	VRT	UIM	VRB	778
0	6	11	16	21 31

```
do i=0 to 127 by 32
   VRT_{i:i+31} ←
          ConvertUXWtoSP( (VRB)_{i:i+31} ) ÷fp 2^UIM
```

For each vector element *i* from 0 to 3, do the following.

> Unsigned fixed-point word element *i* in VRB is converted to the nearest single-precision floating-point value. The result is divided by 2^{UIM} and placed into word element *i* of VRT.

Special Registers Altered:
None

Extended Mnemonics:

Examples of extended mnemonics for *Vector Convert from Unsigned Fixed-Point Word*

Extended:		Equivalent to:	
vcuxwfp	VRT,VRB,UIM	vcfux	VRT,VRB,UIM

Vector Round to Single-Precision Integer toward -Infinity VX-form

vrfim VRT,VRB

4	VRT	///	VRB	714
0	6	11	16	21 31

```
do i=0 to 127 by 32
   VRT0:31 ← RoundToSPIntFloor( (VRB)0:31 )
```

For each vector element *i* from 0 to 3, do the following.

Single-precision floating-point element *i* in VRB is rounded to a single-precision floating-point integer using the rounding mode Round toward -Infinity.

The result is placed into the corresponding word element *i* of VRT.

Special Registers Altered:
None

Programming Note

The *Vector Convert To Fixed-Point Word* instructions support only the rounding mode Round toward Zero. A floating-point number can be converted to a fixed-point integer using any of the other three rounding modes by executing the appropriate *Vector Round to Floating-Point Integer* instruction before the *Vector Convert To Fixed-Point Word* instruction.

Programming Note

The fixed-point integers used by the *Vector Convert* instructions can be interpreted as consisting of 32-UIM integer bits followed by UIM fraction bits.

Vector Round to Single-Precision Integer Nearest VX-form

vrfin VRT,VRB

4	VRT	///	VRB	522
0	6	11	16	21 31

```
do i=0 to 127 by 32
   VRT0:31 ← RoundToSPIntNear( (VRB)0:31 )
```

For each vector element *i* from 0 to 3, do the following.

Single-precision floating-point element *i* in VRB is rounded to a single-precision floating-point integer using the rounding mode Round to Nearest.

The result is placed into the corresponding word element *i* of VRT.

Special Registers Altered:
None

Vector Round to Single-Precision Integer toward +Infinity VX-form

vrfip VRT,VRB

4	VRT	///	VRB	650
0	6	11	16	21 31

```
do i=0 to 127 by 32
   VRT0:31 ← RoundToSPIntCeil( (VRB)0:31 )
```

For each vector element *i* from 0 to 3, do the following.

Single-precision floating-point element *i* in VRB is rounded to a single-precision floating-point integer using the rounding mode Round toward +Infinity.

The result is placed into the corresponding word element *i* of VRT.

Special Registers Altered:
None

Vector Round to Single-Precision Integer toward Zero VX-form

vrfiz VRT,VRB

4	VRT	///	VRB	586
0	6	11	16	21 31

```
do i=0 to 127 by 32
   VRT0:31 ← RoundToSPIntTrunc( (VRB)0:31 )
```

For each vector element *i* from 0 to 3, do the following.

Single-precision floating-point element *i* in VRB is rounded to a single-precision floating-point integer using the rounding mode Round toward Zero.

The result is placed into the corresponding word element *i* of VRT.

Special Registers Altered:
None

5.10.4 Vector Floating-Point Compare Instructions

The *Vector Floating-Point Compare* instructions compare two Vector Registers word element by word element, interpreting the elements as single-precision floating-point numbers. With the exception of the *Vector Compare Bounds Floating-Point* instruction, they set the target Vector Register, and CR Field 6 if Rc=1, in the same manner as do the *Vector Integer Compare* instructions; see Section 5.9.2.

The *Vector Compare Bounds Floating-Point* instruction sets the target Vector Register, and CR Field 6 if Rc=1, to indicate whether the elements in VRA are within the bounds specified by the corresponding element in VRB, as explained in the instruction description. A single-precision floating-point value x is said to be "within the bounds" specified by a single-precision floating-point value y if $-y \le x \le y$.

Vector Compare Bounds Single-Precision 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

```
do i=0 to 127 by 32
   le ← ( (VRA)i:i+31 ≤fp (VRB)i:i+31 )
   ge ← ( (VRA)i:i+31 ≥fp -(VRB)i:i+31 )
   VRTi:i+31 ← ¬le || ¬ge || 30 0
if Rc=1 then do
   ib ← (VRT=128 0)
   CR6 ← 0b00 || ib || 0b0
```

For each vector element *i* from 0 to 3, do the following.

Single-precision floating-point word element *i* in VRA is compared to single-precision floating-point word element *i* in VRB. A 2-bit value is formed that indicates whether the element in VRA is within the bounds specified by the element in VRB, as follows.

- Bit 0 of the 2-bit value is set to 0 if the element in VRA is less than or equal to the element in VRB, and is set to 1 otherwise.
- Bit 1 of the 2-bit value is set to 0 if the element in VRA is greater than or equal to the negation of the element in VRB, and is set to 1 otherwise.

The 2-bit value is placed into the high-order two bits of word element *i* of VRT and the remaining bits of element *i* are set to 0.

If Rc=1, CR field 6 is set as follows.

Bit	Description
0	Set to 0
1	Set to 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	Set to 0

Special Registers Altered:
CR6 (if Rc=1)

Programming Note

Each single-precision floating-point word element in VRB should be non-negative; if it is negative, the corresponding element in VRA will necessarily be out of bounds.

One exception to this is when the value of an element in VRB is -0.0 and the value of the corresponding element in VRA is either +0.0 or -0.0. +0.0 and -0.0 compare equal to -0.0.

Vector Compare Equal To Single-Precision 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

```
do i=0 to 127 by 32
   VRTi:i+31 ← ((VRA)i:i+31 =fp (VRB)i:i+31) ? 32 1 : 32 0
if Rc=1 then do
   t ← ( VRT=128 1 )
   f ← ( VRT=128 0 )
   CR6 ← t || 0b0 || f || 0b0
```

For each vector element *i* from 0 to 3, do the following.

Single-precision floating-point element *i* in VRA is compared to single-precision floating-point element *i* in VRB. Word element *i* in VRT is set to all 1s if single-precision floating-point element *i* in VRA is equal to single-precision floating-point element *i* in VRB, and is set to all 0s otherwise.

If the source element i in VRA or the source element i in VRB is a NaN, VRT is set to all 0s, indicating "not equal to". If the source element i in VRA and the source element i in VRB are both infinity with the same sign, VRT is set to all 1s, indicating "equal to".

Special Registers Altered:
CR6 (if Rc=1)

Vector Compare Greater Than or Equal To Single-Precision 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

```
do i=0 to 127 by 32
   VRT_{i:i+31} ←((VRA)_{i:i+31} ≥fp (VRB)_{i:i+31}) ? ³²1 : ³²0
if Rc=1 then do
   t ← ( VRT=¹²⁸1 )
   f ← ( VRT=¹²⁸0 )
   CR6 ← t || 0b0 || f || 0b0
```

For each vector element *i* from 0 to 3, do the following.

Single-precision floating-point element *i* in VRA is compared to single-precision floating-point element *i* in VRB. Word element *i* in VRT is set to all 1s if single-precision floating-point element *i* in VRA is greater than or equal to single-precision floating-point element *i* in VRB, and is set to all 0s otherwise.

If the source element i in VRA or the source element i in VRB is a NaN, VRT is set to all 0s, indicating “not greater than or equal to”. If the source element i in VRA and the source element i in VRB are both infinity with the same sign, VRT is set to all 1s, indicating “greater than or equal to”.

Special Registers Altered:
CR6 (if Rc=1)

Vector Compare Greater Than Single-Precision 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

```
do i=0 to 127 by 32
   VRT_{i:i+31} = ((VRA)_{i:i+31} >fp (VRB)_{i:i+31}) ? ³²1 : ³²0
if Rc=1 then do
   t ← ( VRT=¹²⁸1 )
   f ← ( VRT=¹²⁸0 )
   CR6 ← t || 0b0 || f || 0b0
```

For each vector element *i* from 0 to 3, do the following.

Single-precision floating-point element *i* in VRA is compared to single-precision floating-point element *i* in VRB. Word element *i* in VRT is set to all 1s if single-precision floating-point element *i* in VRA is greater than single-precision floating-point element *i* in VRB, and is set to all 0s otherwise.

If the source element i in VRA or the source element i in VRB is a NaN, VRT is set to all 0s, indicating “not greater than”. If the source element i in VRA and the source element i in VRB are both infinity with the same sign, VRT is set to all 0s, indicating “not greater than”.

Special Registers Altered:
CR6 (if Rc=1)

5.10.5 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

```
do i=0 to 127 by 32
   VRTi:i+31 ← Power2EstimateSP( (VRB)i:i+31 )
```

For each vector element *i* from 0 to 3, do the following.

> The single-precision floating-point estimate of 2 raised to the power of single-precision floating-point element *i* in VRB is placed into word element *i* of VRT.

Let x be any single-precision floating-point input value. Unless x< -146 or the single-precision floating-point result of computing 2 raised to the power x would be a zero, an infinity, or a QNaN, the estimate has a relative error in precision no greater than one part in 16. The most significant 12 bits of the estimate's significand are monotonic. An integral input value returns an integral value when the result is representable.

The result for various special cases of the source value is given below.

Value	Result
- Infinity	+0
-0	+1
+0	+1
+Infinity	+Infinity
NaN	QNaN

Special Registers Altered:

None

Vector Log Base 2 Estimate Floating-Point VX-form

vlogefp VRT,VRB

4	VRT	///	VRB	458
0	6	11	16	21 31

```
do i=0 to 127 by 32
   VRTi:i+31 ← LogBase2EstimateSP((VRB)i:i+31)
```

For each vector element *i* from 0 to 3, do the following.

> The single-precision floating-point estimate of the base 2 logarithm of single-precision floating-point element *i* in VRB is placed into the corresponding word element of VRT.

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 register VRT may vary between implementations, and between different executions on the same implementation.

The result for various special cases of the source value is given below.

Value	Result
- Infinity	QNaN
< 0	QNaN
- 0	- Infinity
+0	- Infinity
+Infinity	+Infinity
NaN	QNaN

Special Registers Altered:

None

Vector Reciprocal Estimate Single-Precision *VX-form*

vrefp VRT,VRB

4	VRT	///	VRB	266
0	6	11	16	21 31

```
do i=0 to 127 by 32
   VRT_{i:i+31} ← ReciprocalEstimateSP( (VRB)_{i:i+31} )
```

For each vector element *i* from 0 to 3, do the following.

> The single-precision floating-point estimate of the reciprocal of single-precision floating-point element *i* in VRB is placed into word element *i* of VRT.

Unless the single-precision floating-point result of computing the reciprocal of a value would be a zero, an infinity, or a QNaN, the estimate has a relative error in precision no greater than one part in 4096.

Note that results may vary between implementations, and between different executions on the same implementation.

The result for various special cases of the source value is given below.

Value	Result
- Infinity	-0
- 0	- Infinity
+0	+ Infinity
+Infinity	+0
NaN	QNaN

Special Registers Altered:
None

Vector Reciprocal Square Root Estimate Single-Precision *VX-form*

vrsqrtefp VRT,VRB

4	VRT	///	VRB	330
0	6	11	16	21 31

```
do i=0 to 127 by 32
   VRT_{i:i+31} ← ReciprocalSquareRootEstimateSP(
(VRB)_{i:i+31} )
```

For each vector element *i* from 0 to 3, do the following.

> The single-precision floating-point estimate of the reciprocal of the square root of single-precision floating-point element *i* in VRB is placed into word element *i* of VRT.

Let *x* be any single-precision floating-point value. Unless the single-precision floating-point result of computing the reciprocal of the square root of *x* would be a zero, an infinity, or a QNaN, the estimate has a relative error in precision no greater than one part in 4096.

Note that results may vary between implementations, and between different executions on the same implementation.

The result for various special cases of the source value is given below.

Value	Result
- Infinity	QNaN
< 0	QNaN
- 0	- Infinity
+0	+ Infinity
+Infinity	+0
NaN	QNaN

Special Registers Altered:
None

5.11 Vector Status and Control Register Instructions

Move To Vector Status and Control Register *VX-form*

mtvscr VRB

4	///	VRB	1604
0	6	16	21 31

$VSCR \leftarrow (VRB)_{96:127}$

The contents of word element 3 of VRB are placed into the VSCR.

Special Registers Altered:
None

Move From Vector Status and Control Register *VX-form*

mfvscr VRT

4	VRT	///	1540
0	6	11	21 31

$VRT \leftarrow {}^{96}0 \parallel (VSCR)$

The contents of the VSCR are placed into word element 3 of VRT.

The remaining word elements in VRT are set to 0.

Special Registers Altered:
None

Chapter 6. Signal Processing Engine (SPE) [Category: Signal Processing Engine]

6.1 Overview

The Signal Processing Engine (SPE) accelerates signal processing applications normally suited to DSP operation. This is accomplished using short vectors (two element) within 64-bit GPRs and using single instruction multiple data (SIMD) operations to perform the requisite computations. SPE also architects an Accumulator register to allow for back to back operations without loop unrolling.

6.2 Nomenclature and Conventions

Several conventions regarding nomenclature are used for SPE:

- The Signal Processing Engine category is abbreviated as SPE.
- Bits 0 to 31 of a 64-bit register are referenced as upper word, even word or high word element of the register. Bits 32:63 are referred to as lower word, odd word or low word element of the register. Each half is an element of a 64-bit GPR.
- Bits 0 to 15 and bits 32 to 47 are referenced as even halfwords. Bits 16 to 31 and bits 48 to 63 are referenced as odd halfwords.
- Mnemonics for SPE instructions generally begin with the letters ‘ev’ (embedded vector).

The RTL conventions in described below are used in addition to those described in Section 1.3:Additional RTL functions are described in Appendix C.

Notation	Meaning
$\times_{sf}$	Signed fractional multiplication. Result of multiplying 2 signed fractional quantities having bit length n taking the least significant 2n-1 bits of the sign extended product and concatenating a 0 to the least significant bit forming a signed fractional result of 2n bits. Two 16-bit signed fractional quantities, a and b are multiplied, as shown below: $ea_{0:31}$ = EXTS(a) $eb_{0:31}$ = EXTS(b) $prod_{0:63}$ = ea X eb $eprod_{0:63}$ = EXTS($prod_{32:63}$) $result_{0:31}$ = $eprod_{33:63}$ || 0b0
$\times_{gsf}$	Guarded signed fractional multiplication. Result of multiplying 2 signed fractional quantities having bit length 16 taking the least significant 31 bits of the sign extended product and concatenating a 0 to the least significant bit forming a guarded signed fractional result of 64 bits. Since guarded signed fractional multiplication produces a 64-bit result, fractional input quantities of -1 and -1 can produce +1 in the intermediate product. Two 16-bit fractional quantities, a and b are multiplied, as shown below:

$ea_{0:31}$ = EXTS(a)
$eb_{0:31}$ = EXTS(b)
$prod_{0:63}$ = ea X eb
$eprod_{0:63}$ = EXTS($prod_{32:63}$)
$result_{0:63}$ = $eprod_{1:63}$ || 0b0

<< Logical shift left. x << y shifts value x left by y bits, leaving zeros in the vacated bits.

>> Logical shift right. x >> y shifts value x right by y bits, leaving zeros in the vacated bits.

6.3 Programming Model

6.3.1 General Operation

SPE instructions generally take elements from one source register and operate on them with the corresponding elements of a second source register (and/or the accumulator) to produce results. Results are placed in the destination register and/or the accumulator. Instructions that are vector in nature (i.e. produce results of more than one element) provide results for each element that are independent of the computation of the other elements. These instructions can also be used to perform scalar DSP operations by ignoring the results of the upper 32-bit half of the register file.

There are no record forms of *SPE* instructions. As a result, the meaning of bits in the CR is different than for other categories. *SPE Compare* instructions specify a CR field, two source registers, and the type of compare: greater than, less than, or equal. Two bits of the CR field are written with the result of the vector compare, one for each element. The remaining two bits reflect the ANDing and ORing of the vector compare results.

6.3.2 GPR Registers

The SPE requires a GPR register file with thirty-two 64-bit registers. For 32-bit implementations, instructions that normally operate on a 32-bit register file access and change only the least significant 32-bits of the GPRs leaving the most significant 32-bits unchanged. For 64-bit implementations, operation of these instructions is unchanged, i.e. those instructions continue to operate on the 64-bit registers as they would if the SPE was not implemented. Most *SPE* instructions view the 64-bit register as being composed of a vector of two elements, each of which is 32 bits wide (some instructions read or write 16-bit elements). The most significant 32-bits are called the upper word, high word or even word. The least significant 32-bits are called the lower word, low word or odd word. Unless otherwise specified, *SPE* instructions write all 64-bits of the destination register.

GPR Upper Word	GPR Lower Word
0	32 63

Figure 66. GPR

6.3.3 Accumulator Register

A partially visible accumulator register (ACC) is provided for some SPE instructions. The accumulator is a 64-bit register that holds the results of the *Multiply Accumulate* (*MAC*) forms of *SPE Fixed-Point* instructions. The accumulator allows the back-to-back execution of dependent *MAC* instructions, something that is found in the inner loops of DSP code such as FIR and FFT filters. The accumulator is partially visible to the programmer in the sense that its results do not have to be explicitly read to use them. Instead they are always copied into a 64-bit destination GPR which is specified as part of the instruction. Based upon the type of instruction, the accumulator can hold either a single 64-bit value or a vector of two 32-bit elements.

ACC Upper Word	ACC Lower Word
0	32 63

Figure 67. Accumulator

6.3.4 Signal Processing Embedded Floating-Point Status and Control Register (SPEFSCR)

Status and control for SPE uses the SPEFSCR register. This register is also used by the SPE.Embedded Float Scalar Double, SPE.Embedded Float Scalar Single, and SPE.Embedded Float Vector categories. Status and control bits are shared with these categories. The SPEFSCR register is implemented as special purpose register (SPR) number 512 and is read and written by the ***mfspr*** and ***mtspr*** instructions. *SPE* instructions affect both the high element (bits 32:33) and low element status flags (bits 48:49) of the SPEFSCR.

SPEFSCR
32 63

Figure 68. Signal Processing and Embedded Floating-Point Status and Control Register

The SPEFSCR bits are defined as shown below.

Bit	Description
32	***Summary Integer Overflow High*** (SOVH) SOVH is set to 1 when an *SPE* instruction sets OVH. This is a sticky bit.

33 ***Integer Overflow High*** (OVH)
OVH is set to 1 to indicate that an overflow has occurred in the upper element during execution of an *SPE* instruction. The bit is set to 1 if a result of an operation performed by the instruction cannot be represented in the number of bits into which the result is to be placed, and is set to 0 otherwise. The OVH bit is not altered by *Modulo* instructions, nor by other instructions that cannot overflow.

34 ***Embedded Floating-Point Guard Bit High*** (FGH) [Category: SP.FV]
FGH is supplied for use by the Embedded Floating-Point Round interrupt handler. FGH is an extension of the low-order bits of the fractional result produced from an *SPE.Embedded Float Vector* instruction on the high word. FGH is zeroed if an overflow, underflow, or invalid input error is detected on the high element of an *SPE.Embedded Float Vector* instruction.

Execution of an *SPE.Embedded Float Scalar* instruction leaves FGH undefined.

35 ***Embedded Floating-Point Inexact Bit High*** (FXH) [Category: SP.FV]
FXH is supplied for use by the Embedded Floating-Point Round interrupt handler. FXH is an extension of the low-order bits of the fractional result produced from an *SPE.Embedded Float Vector* instruction on the high word. FXH represents the logical 'or' of all the bits shifted right from the Guard bit when the fractional result is normalized. FXH is zeroed if an overflow, underflow, or invalid input error is detected on the high element of an *SPE.Embedded Float Vector* instruction.

Execution of an *SPE.Embedded Float Scalar* instruction leaves FXH undefined.

36 ***Embedded Floating-Point Invalid Operation/Input Error High*** (FINVH) [Category: SP.FV]
The FINVH bit is set to 1 if any high word operand of an *SPE.Embedded Float Vector* instruction is infinity, NaN, or a denormalized value, or if the instruction is a divide and the dividend and divisor are both 0, or if a conversion to integer or fractional value overflows.

Execution of an *SPE.Embedded Float Scalar* instruction leaves FINVH undefined.

37 ***Embedded Floating-Point Divide By Zero High*** (FDBZH) [Category: SP.FV]
The FDBZH bit is set to 1 when an *SPE.Embedded Vector Floating-Point Divide* instruction is executed with a divisor of 0 in the high word operand, and the dividend is a finite nonzero number.

Execution of an *SPE.Embedded Float Scalar* instruction leaves FDBZH undefined.

38 ***Embedded Floating-Point Underflow High*** (FUNFH) [Category: SP.FV]
The FUNFH bit is set to 1 when the execution of an *SPE.Embedded Float Vector* instruction results in an underflow on the high word operation.

Execution of an *SPE.Embedded Float Scalar* instruction leaves FUNFH undefined.

39 ***Embedded Floating-Point Overflow High*** (FOVFH) [Category: SP.FV]
The FOVFH bit is set to 1 when the execution of an *SPE.Embedded Float Vector* instruction results in an overflow on the high word operation.

Execution of an *SPE.Embedded Float Scalar* instruction leaves FOVFH undefined.

40:41 Reserved

42 ***Embedded Floating-Point Inexact Sticky Flag*** (FINXS) [Categories: SP.FV, SP.FD, SP.FS]
The FINXS bit is set to 1 whenever the execution of an *Embedded Floating-Point* instruction delivers an inexact result for either the low or high element and no Embedded Floating-Point Data interrupt is taken for either element, or if an *Embedded Floating-Point* instruction results in overflow (FOVF=1 or FOVFH=1), but Embedded Floating-Point Overflow exceptions are disabled (FOVFE=0), or if an *Embedded Floating-Point* instruction results in underflow (FUNF=1 or FUNFH=1), but Embedded Floating-Point Underflow exceptions are disabled (FUNFE=0), and no Embedded Floating-Point Data interrupt occurs. This is a sticky bit.

43 ***Embedded Floating-Point Invalid Operation/Input Sticky Flag*** (FINVS) [Categories: SP.FV, SP.FD, SP.FS]
The FINVS bit is defined to be the sticky result of any *Embedded Floating-Point* instruction that causes FINVH or FINV to be set to 1. That is, FINVS ← FINVS | FINV | FINVH. This is a sticky bit.

44 ***Embedded Floating-Point Divide By Zero Sticky Flag*** (FDBZS) [Categories: SP.FV, SP.FD, SP.FS]
The FDBZS bit is set to 1 when an *Embedded Floating-Point Divide* instruction sets FDBZH or FDBZ to 1. That is, FDBZS ← FDBZS | FDBZ | FDBZH. This is a sticky bit.

45 ***Embedded Floating-Point Underflow Sticky Flag*** (FUNFS) [Categories: SP.FV, SP.FD, SP.FS]
The FUNFS bit is defined to be the sticky

result of any *Embedded Floating-Point* instruction that causes FUNFH or FUNF to be set to 1. That is, FUNFS ← FUNFS | FUNF | FUNFH. This is a sticky bit.

46 ***Embedded Floating-Point Overflow Sticky Flag*** (FOVFS) [Categories: SP.FV, SP.FD, SP.FS]
The FOVFS bit is defined to be the sticky result of any *Embedded Floating-Point* instruction that causes FOVH or FOVF to be set to 1. That is, FOVFS ← FOVFS | FOVF | FOVFH. This is a sticky bit.

47 Reserved

48 ***Summary Integer Overflow*** (SOV)
SOV is set to 1 when an *SPE* instruction sets OV to 1. This is a sticky bit.

49 ***Integer Overflow*** (OV)
OV is set to 1 to indicate that an overflow has occurred in the lower element during execution of an *SPE* instruction. The bit is set to 1 if a result of an operation performed by the instruction cannot be represented in the number of bits into which the result is to be placed, and is set to 0 otherwise. The OV bit is not altered by *Modulo* instructions, nor by other instructions that cannot overflow.

50 ***Embedded Floating-Point Guard Bit (Low/scalar)*** (FG) [Categories: SP.FV, SP.FD, SP.FS]
FG is supplied for use by the Embedded Floating-Point Round interrupt handler. FG is an extension of the low-order bits of the fractional result produced from an *Embedded Floating-Point* instruction on the low word. FG is zeroed if an overflow, underflow, or invalid input error is detected on the low element of an *Embedded Floating-Point* instruction.

51 ***Embedded Floating-Point Inexact Bit (Low/scalar)*** (FX) [Categories: SP.FV, SP.FD, SP.FS]
FX is supplied for use by the Embedded Floating-Point Round interrupt handler. FX is an extension of the low-order bits of the fractional result produced from an *Embedded Floating-Point* instruction on the low word. FX represents the logical 'or' of all the bits shifted right from the Guard bit when the fractional result is normalized. FX is zeroed if an overflow, underflow, or invalid input error is detected on *Embedded Floating-Point* instruction

52 ***Embedded Floating-Point Invalid Operation/Input Error (Low/scalar)*** (FINV) [Categories: SP.FV, SP.FD, SP.FS]
The FINV bit is set to 1 if any low word operand of an *Embedded Floating-Point* instruction is infinity, NaN, or a denormalized value, or if the operation is a divide and the dividend and divisor are both 0, or if a conversion to integer or fractional value overflows.

53 ***Embedded Floating-Point Divide By Zero (Low/scalar)*** (FDBZ) [Categories: SP.FV, SP.FD, SP.FS]
The FDBZ bit is set to 1 when an *Embedded Floating-Point Divide* instruction is executed with a divisor of 0 in the low word operand, and the dividend is a finite nonzero number.

54 ***Embedded Floating-Point Underflow (Low/scalar)*** (FUNF) [Categories: SP.FV, SP.FD, SP.FS]
The FUNF bit is set to 1 when the execution of an *Embedded Floating-Point* instruction results in an underflow on the low word operation.

55 ***Embedded Floating-Point Overflow (Low/scalar)*** (FOVF) [Categories: SP.FV, SP.FD, SP.FS]
The FOVF bit is set to 1 when the execution of an *Embedded Floating-Point* instruction results in an overflow on the low word operation.

56 Reserved

57 ***Embedded Floating-Point Round (Inexact) Exception Enable*** (FINXE) [Categories: SP.FV, SP.FD, SP.FS]

0 Exception disabled
1 Exception enabled

The Embedded Floating-Point Round interrupt is taken if the exception is enabled and if FG | FGH | FX | FXH (signifying an inexact result) is set to 1 as a result of an *Embedded Floating-Point* instruction.

If an *Embedded Floating-Point* instruction results in overflow or underflow and the corresponding Embedded Floating-Point Underflow or Embedded Floating-Point Overflow exception is disabled then the Embedded Floating-Point Round interrupt is taken.

58 ***Embedded Floating-Point Invalid Operation/Input Error Exception Enable*** (FINVE) [Categories: SP.FV, SP.FD, SP.FS]

0 Exception disabled
1 Exception enabled

If the exception is enabled, an Embedded Floating-Point Data interrupt is taken if the FINV or FINVH bit is set to 1 by an *Embedded Floating-Point* instruction.

59 ***Embedded Floating-Point Divide By Zero Exception Enable*** (FDBZE) [Categories: SP.FV, SP.FD, SP.FS]

0 Exception disabled

1 Exception enabled

If the exception is enabled, an Embedded Floating-Point Data interrupt is taken if the FDBZ or FDBZH bit is set to 1 by an *Embedded Floating-Point* instruction.

60 ***Embedded Floating-Point Underflow Exception Enable*** (FUNFE) [Categories: SP.FV, SP.FD, SP.FS]

0 Exception disabled
1 Exception enabled

If the exception is enabled, an Embedded Floating-Point Data interrupt is taken if the FUNF or FUNFH bit is set to 1 by an *Embedded Floating-Point* instruction.

61 ***Embedded Floating-Point Overflow Exception Enable*** (FOVFE) [Categories: SP.FV, SP.FD, SP.FS]

0 Exception disabled
1 Exception enabled

If the exception is enabled, an Embedded Floating-Point Data interrupt is taken if the FOVF or FOVFH bit is set to 1 by an *Embedded Floating-Point* instruction.

62:63 ***Embedded Floating-Point Rounding Mode Control*** (FRMC) [Categories: SP.FV, SP.FD, SP.FS]

00 Round to Nearest
01 Round toward Zero
10 Round toward +Infinity
11 Round toward -Infinity

Programming Note

Rounding modes 0b10 (+Infinity) and 0b11 (-Infinity) may not be supported by some implementations. If an implementation does not support these, Embedded Floating-Point Round interrupts are generated for every *Embedded Floating-Point* instruction for which rounding is required when +Infinity or -Infinity modes are set and software is required to produce the correctly rounded result

6.3.5 Data Formats

The SPE provides two different data formats, integer and fractional. Both data formats can be treated as signed or unsigned quantities.

6.3.5.1 Integer Format

Unsigned integers consist of 16, 32, or 64-bit binary integer values. The largest representable value is 2^n-1 where n represents the number of bits in the value. The smallest representable value is 0. Computations that produce values larger than 2^n-1 or smaller than 0 may set OV or OVH in the SPEFSCR.

Signed integers consist of 16, 32, or 64-bit binary values in two's complement form. The largest representable value is 2^{n-1}-1 where n represents the number of bits in the value. The smallest representable value is -2^{n-1}. Computations that produce values larger than 2^{n-1}-1 or smaller than -2^{n-1} may set OV or OVH in the SPEFSCR.

6.3.5.2 Fractional Format

Fractional data format is conventionally used for DSP fractional arithmetic. Fractional data is useful for representing data converted from analog devices.

Unsigned fractions consist of 16, 32, or 64-bit binary fractional values that range from 0 to less than 1. Unsigned fractions place the radix point immediately to the left of the most significant bit. The most significant bit of the value represents the value 2^{-1}, the next most significant bit represents the value 2^{-2} and so on. The largest representable value is 1-2^{-n} where n represents the number of bits in the value. The smallest representable value is 0. Computations that produce values larger than 1-2^{-n} or smaller than 0 may set OV or OVH in the SPEFSCR. The SPE category does not define unsigned fractional forms of instructions to manipulate unsigned fractional data since the unsigned integer forms of the instructions produce the same results as would the unsigned fractional forms.

Guarded unsigned fractions are 64-bit binary fractional values. Guarded unsigned fractions place the decimal point immediately to the left of bit 32. The largest representable value is 2^{32}-2^{-32}. The smallest representable value is 0. Guarded unsigned fractional computations are always modulo and do not set OV or OVH in the SPEFSCR.

Signed fractions consist of 16, 32, or 64-bit binary fractional values in two's-complement form that range from -1 to less than 1. Signed fractions place the decimal point immediately to the right of the most significant bit. The largest representable value is 1-$2^{-(n-1)}$ where n represents the number of bits in the value. The smallest representable value is -1. Computations that produce values larger than 1-$2^{-(n-1)}$or smaller than -1 may set OV or OVH in the SPEFSCR. Multiplication of two signed fractional values causes the result to be shifted left one bit to remove the resultant redundant sign bit in the product. In this case, a 0 bit is concatenated as the least significant bit of the shifted result.

Guarded signed fractions are 64-bit binary fractional values. Guarded signed fractions place the decimal point immediately to the left of bit 33. The largest representable value is 2^{32}-2^{-31}. The smallest representable value is -2^{32}-1+2^{-31}. Guarded signed fractional computations are always modulo and do not set OV or OVH in the SPEFSCR.

6.3.6 Computational Operations

The SPE category supports several different computational capabilities. Both modulo and saturation results can be performed. Modulo results produce truncation of the overflow bits in a calculation, therefore overflow does not occur and no saturation is performed. For instructions for which overflow occurs, saturation provides a maximum or minimum representable value (for the data type) in the case of overflow. Instructions are provided for a wide range of computational capability. The operation types can be divided into 4 basic categories:

- *Simple Vector* instructions. These instructions use the corresponding low and high word elements of the operands to produce a vector result that is placed in the destination register, the accumulator, or both.
- *Multiply and Accumulate* instructions. These instructions perform multiply operations, optionally add the result to the accumulator, and place the result into the destination register and optionally into the accumulator. These instructions are composed of different multiply forms, data formats and data accumulate options. The mnemonics for these instructions indicate their various characteristics. These are shown in Table 2.
- *Load* and *Store* instructions. These instructions provide load and store capabilities for moving data to and from memory. A variety of forms are provided that position data for efficient computation.
- *Compare* and miscellaneous instructions. These instructions perform miscellaneous functions such as field manipulation, bit reversed incrementing, and vector compares.

Table 2: Mnemonic Extensions for Multiply Accumulate Instructions

Extension	Meaning	Comments
	Multiply Form	
he	halfword even	16 X 16 → 32
heg	halfword even guarded	16 X 16 → 32, 64-bit final accumulate result
ho	halfword odd	16 X 16 → 32
hog	halfword odd guarded	16 X 16 → 32, 64-bit final accumulate result
w	word	32 X 32 → 64
wh	word high	32 X 32 → 32 (high-order 32 bits of product)
wl	word low	32 X 32 → 32 (low-order 32 bits of product)
	Data Format	
smf	signed modulo fractional	modulo, no saturation or overflow
smi	signed modulo integer	modulo, no saturation or overflow
ssf	signed saturate fractional	saturation on product and accumulate
ssi	signed saturate integer	saturation on product and accumulate
umi	unsigned modulo integer	modulo, no saturation or overflow
usi	unsigned saturate integer	saturation on product and accumulate
	Accumulate Option	
a	place in accumulator	result → accumulator
aa	add to accumulator	accumulator + result → accumulator
aaw	add to accumulator as word elements	$accumulator_{0:31} + result_{0:31} \rightarrow accumulator_{0:31}$ $accumulator_{32:63} + result_{32:63} \rightarrow accumulator_{32:63}$
an	add negated to accumulator	accumulator - result → accumulator
anw	add negated to accumulator as word elements	$accumulator_{0:31} - result_{0:31} \rightarrow accumulator_{0:31}$ $accumulator_{32:63} - result_{32:63} \rightarrow accumulator_{32:63}$

6.3.7 SPE Instructions

6.3.8 Saturation, Shift, and Bit Reverse Models

For saturation, left shifts, and bit reversal, the pseudo RTL is provided here to more accurately describe those functions that are referenced in the instruction pseudo RTL.

6.3.8.1 Saturation

SATURATE(ov, carry, sat_ovn, sat_ov, val)

```
if ov then
    if carry then
        return sat_ovn
    else
        return sat_ov
else
    return val
```

6.3.8.2 Shift Left

SL(value, cnt)

```
if cnt > 31 then
    return 0
else
    return (value << cnt)
```

6.3.8.3 Bit Reverse

BITREVERSE(value)

```
result ← 0
mask ← 1
shift ← 31
cnt ← 32
while cnt > 0 then do
    t ← value & mask
    if shift >= 0 then
        result ←(t << shift) | result
    else
        result ←(t >> -shift) | result
    cnt ← cnt - 1
    shift ← shift - 2
    mask ← mask << 1
return result
```

6.3.9 SPE Instruction Set

Bit Reversed Increment *EVX-form*

brinc RT,RA,RB

4	RT	RA	RB	527
0	6	11	16	21 31

```
n ← implementation-dependent number of mask bits
mask ← (RB)64-n:63
a ← (RA)64-n:63
d ← BITREVERSE(1 + BITREVERSE(a | (¬ mask)))
RT ← (RA)0:63-n || (d & mask)
```

brinc computes a bit-reverse index based on the contents of RA and a mask specified in RB. The new index is written to RT.

The number of bits in the mask is implementation-dependent but may not exceed 32.

Special Registers Altered:
None

Programming Note

brinc provides a way for software to access FFT data in a bit-reversed manner. RA contains the index into a buffer that contains data on which FFT is to be performed. RB contains a mask that allows the index to be updated with bit-reversed addressing. Typically this instruction precedes a load with index instruction; for example,

```
brinc r2, r3, r4
lhax r8, r5, r2
```

RB contains a bit-mask that is based on the number of points in an FFT. To access a buffer containing n byte sized data that is to be accessed with bit-reversed addressing, the mask has $\log_2 n$ 1s in the least significant bit positions and 0s in the remaining most significant bit positions. If, however, the data size is a multiple of a halfword or a word, the mask is constructed so that the 1s are shifted left by $\log_2$ (size of the data) and 0s are placed in the least significant bit positions.

Programming Note

This instruction only modifies the lower 32 bits of the destination register in 32-bit implementations. For 64-bit implementations in 32-bit mode, the contents of the upper 32-bits of the destination register are undefined.

Programming Note

Execution of ***brinc*** does not cause SPE Unavailable exceptions regardless of MSR_{SPV}.

Vector Absolute Value *EVX-form*

evabs RT,RA

4	RT	RA	///	520
0	6	11	16	21 31

$RT_{0:31} \leftarrow ABS((RA)_{0:31})$
$RT_{32:63} \leftarrow ABS((RA)_{32:63})$

The absolute value of each element of RA is placed in the corresponding elements of RT. An absolute value of 0x8000_0000 (most negative number) returns 0x8000_0000.

Special Registers Altered:
None

Vector Add Immediate Word *EVX-form*

evaddiw RT,RB,UI

4	RT	UI	RB	514
0	6	11	16	21 31

$RT_{0:31} \leftarrow (RB)_{0:31} + EXTZ(UI)$
$RT_{32:63} \leftarrow (RB)_{32:63} + EXTZ(UI)$

UI is zero-extended and added to both the high and low elements of RB and the results are placed in RT. Note that the same value is added to both elements of the register.

Special Registers Altered:
None

Vector Add Signed, Modulo, Integer to Accumulator Word *EVX-form*

evaddsmiaaw RT,RA

4	RT	RA	///	1225
0	6	11	16	21 31

$RT_{0:31} \leftarrow (ACC)_{0:31} + (RA)_{0:31}$
$RT_{32:63} \leftarrow (ACC)_{32:63} + (RA)_{32:63}$
$ACC_{0:63} \leftarrow (RT)_{0:63}$

Each word element in RA is added to the corresponding element in the accumulator and the results are placed in RT and into the accumulator.

Special Registers Altered:
ACC

Vector Add Signed, Saturate, Integer to Accumulator Word EVX-form

evaddssiaaw RT,RA

4	RT	RA	///	1217
0	6	11	16	21 31

$temp_{0:63} \leftarrow EXTS((ACC)_{0:31}) + EXTS((RA)_{0:31})$
$ovh \leftarrow temp_{31} \oplus temp_{32}$
$RT_{0:31} \leftarrow SATURATE(ovh, temp_{31}, 0x8000\_0000, 0x7FFF\_FFFF, temp_{32:63})$

$temp_{0:63} \leftarrow EXTS((ACC)_{32:63}) + EXTS((RA)_{32:63})$
$ovl \leftarrow temp_{31} \oplus temp_{32}$
$RT_{32:63} \leftarrow SATURATE(ovl, temp_{31}, 0x8000\_0000, 0x7FFF\_FFFF, temp_{32:63})$

$ACC_{0:63} \leftarrow (RT)_{0:63}$
$SPEFSCR_{OVH} \leftarrow ovh$
$SPEFSCR_{OV} \leftarrow ovl$
$SPEFSCR_{SOVH} \leftarrow SPEFSCR_{SOVH} \mid ovh$
$SPEFSCR_{SOV} \leftarrow SPEFSCR_{SOV} \mid ovl$

Each signed-integer word element in RA is sign-extended and added to the corresponding sign-extended element in the accumulator saturating if overflow occurs, and the results are placed in RT and the accumulator.

Special Registers Altered:
ACC OV OVH SOV SOVH

Vector Add Unsigned, Modulo, Integer to Accumulator Word EVX-form

evaddumiaaw RT,RA

4	RT	RA	///	1224
0	6	11	16	21 31

$RT_{0:31} \leftarrow (ACC)_{0:31} + (RA)_{0:31}$
$RT_{32:63} \leftarrow (ACC)_{32:63} + (RA)_{32:63}$
$ACC_{0:63} \leftarrow (RT)_{0:63}$

Each unsigned-integer word element in RA is added to the corresponding element in the accumulator and the results are placed in RT and the accumulator.

Special Registers Altered:
ACC

Vector Add Unsigned, Saturate, Integer to Accumulator Word EVX-form

evaddusiaaw RT,RA

4	RT	RA	///	1216
0	6	11	16	21 31

$temp_{0:63} \leftarrow EXTZ((ACC)_{0:31}) + EXTZ((RA)_{0:31})$
$ovh \leftarrow temp_{31}$
$RT_{0:31} \leftarrow SATURATE(ovh, temp_{31}, 0xFFFF\_FFFF, 0xFFFF\_FFFF, temp_{32:63})$

$temp_{0:63} \leftarrow EXTZ((ACC)_{32:63}) + EXTZ((RA)_{32:63})$
$ovl \leftarrow temp_{31}$
$RT_{32:63} \leftarrow SATURATE(ovl, temp_{31}, 0xFFFF\_FFFF, 0xFFFF\_FFFF, temp_{32:63})$

$ACC_{0:63} \leftarrow (RT)_{0:63}$

$SPEFSCR_{OVH} \leftarrow ovh$
$SPEFSCR_{OV} \leftarrow ovl$
$SPEFSCR_{SOVH} \leftarrow SPEFSCR_{SOVH} \mid ovh$
$SPEFSCR_{SOV} \leftarrow SPEFSCR_{SOV} \mid ovl$

Each unsigned-integer word element in RA is zero-extended and added to the corresponding zero-extended element in the accumulator saturating if overflow occurs, and the results are placed in RT and the accumulator.

Special Registers Altered:
ACC OV OVH SOV SOVH

Vector Add Word EVX-form

evaddw RT,RA,RB

4	RT	RA	RB	512
0	6	11	16	21 31

$RT_{0:31} \leftarrow (RA)_{0:31} + (RB)_{0:31}$
$RT_{32:63} \leftarrow (RA)_{32:63} + (RB)_{32:63}$

The corresponding elements of RA and RB are added and the results are placed in RT. The sum is a modulo sum.

Special Registers Altered:
None

Vector AND *EVX-form*

evand RT,RA,RB

4	RT	RA	RB	529
0	6	11	16	21 31

$RT_{0:31} \leftarrow (RA)_{0:31}$ & $(RB)_{0:31}$
$RT_{32:63} \leftarrow (RA)_{32:63}$ & $(RB)_{32:63}$

The corresponding elements of RA and RB are ANDed bitwise and the results are placed in the corresponding element of RT.

Special Registers Altered:
None

Vector AND with Complement *EVX-form*

evandc RT,RA,RB

4	RT	RA	RB	530
0	6	11	16	21 31

$RT_{0:31} \leftarrow (RA)_{0:31}$ & $(\neg(RB)_{0:31})$
$RT_{32:63} \leftarrow (RA)_{32:63}$ & $(\neg(RB)_{32:63})$

The word elements of RA are ANDed bitwise with the complement of the corresponding elements of RB. The results are placed in the corresponding element of RT.

Special Registers Altered:
None

Vector Compare Equal *EVX-form*

evcmpeq BF,RA,RB

4	BF	//	RA	RB	564
0	6	9	11	16	21 31

```
ah ← (RA)0:31
al ← (RA)32:63
bh ← (RB)0:31
bl ← (RB)32:63
if (ah = bh) then ch ← 1
else ch ← 0
if (al = bl) then cl ← 1
else cl ← 0
CR4×BF+32:4×BF+35 ← ch || cl || (ch | cl) || (ch & cl)
```

The most significant bit in BF is set if the high-order element of RA is equal to the high-order element of RB; it is cleared otherwise. The next bit in BF is set if the low-order element of RA is equal to the low-order element of RB and cleared otherwise. The last two bits of BF are set to the OR and AND of the result of the compare of the high and low elements.

Special Registers Altered:
CR field BF

Vector Compare Greater Than Signed *EVX-form*

evcmpgts BF,RA,RB

4	BF	//	RA	RB	561
0	6	9	11	16	21 31

```
ah ← (RA)0:31
al ← (RA)32:63
bh ← (RB)0:31
bl ← (RB)32:63
if (ah > bh) then ch ← 1
else ch ← 0
if (al > bl) then cl ← 1
else cl ← 0
CR4×BF+32:4×BF+35 ← ch || cl || (ch | cl) || (ch & cl)
```

The most significant bit in BF is set if the high-order element of RA is greater than the high-order element of RB; it is cleared otherwise. The next bit in BF is set if the low-order element of RA is greater than the low-order element of RB and cleared otherwise. The last two bits of BF are set to the OR and AND of the result of the compare of the high and low elements.

Special Registers Altered:
CR field BF

Vector Compare Greater Than Unsigned EVX-form

evcmpgtu BF,RA,RB

4	BF	//	RA	RB	560
0	6	9	11	16	21 31

```
ah ← (RA)0:31
al ← (RA)32:63
bh ← (RB)0:31
bl ← (RB)32:63
if (ah >u bh) then ch ← 1
else ch ← 0
if (al >u bl) then cl ← 1
else cl ← 0
CR4×BF+32:4×BF+35 ← ch || cl || (ch | cl) || (ch & cl)
```

The most significant bit in BF is set if the high-order element of RA is greater than the high-order element of RB; it is cleared otherwise. The next bit in BF is set if the low-order element of RA is greater than the low-order element of RB and cleared otherwise. The last two bits of BF are set to the OR and AND of the result of the compare of the high and low elements.

Special Registers Altered:

CR field BF

Vector Compare Less Than Unsigned EVX-form

evcmpltu BF,RA,RB

4	BF	//	RA	RB	562
0	6	9	11	16	21 31

```
ah ← (RA)0:31
al ← (RA)32:63
bh ← (RB)0:31
bl ← (RB)32:63
if (ah <u bh) then ch ← 1
else ch ← 0
if (al <u bl) then cl ← 1
else cl ← 0
CR4×BF+32:4×BF+35 ← ch || cl || (ch | cl) || (ch & cl)
```

The most significant bit in BF is set if the high-order element of RA is less than the high-order element of RB; it is cleared otherwise. The next bit in BF is set if the low-order element of RA is less than the low-order element of RB and cleared otherwise. The last two bits of BF are set to the OR and AND of the result of the compare of the high and low elements.

Special Registers Altered:

CR field BF

Vector Compare Less Than Signed EVX-form

evcmplts BF,RA,RB

4	BF	//	RA	RB	563
0	6	9	11	16	21 31

```
ah ← (RA)0:31
al ← (RA)32:63
bh ← (RB)0:31
bl ← (RB)32:63
if (ah < bh) then ch ← 1
else ch ← 0
if (al < bl) then cl ← 1
else cl ← 0
CR4×BF+32:4×BF+35 ← ch || cl || (ch | cl) || (ch & cl)
```

The most significant bit in BF is set if the high-order element of RA is less than the high-order element of RB; it is cleared otherwise. The next bit in BF is set if the low-order element of RA is less than the low-order element of RB and cleared otherwise. The last two bits of BF are set to the OR and AND of the result of the compare of the high and low elements.

Special Registers Altered:

CR field BF

Vector Count Leading Signed Bits Word EVX-form

evcntlsw RT,RA

4	RT	RA	///	526
0	6	11	16	21 31

```
n ← 0
s ← (RA)n
do while n < 32
   if (RA)n ≠ s then leave
   n ← n + 1
RT0:31 ← n
n ← 0
s ← (RA)n+32
do while n < 32
   if (RA)n+32 ≠ s then leave
   n ← n + 1
RT32:63 ← n
```

The leading sign bits in each element of RA are counted, and the respective count is placed into each element of RT.

Special Registers Altered:
None

Programming Note

evcntlzw is used for unsigned operands; ***evcntlsw*** is used for signed operands.

Vector Count Leading Zeros Word EVX-form

evcntlzw RT,RA

4	RT	RA	///	525
0	6	11	16	21 31

```
n ← 0
do while n < 32
   if (RA)n = 1 then leave
   n ← n + 1
RT0:31 ← n
n ← 0
do while n < 32
   if (RA)n+32 = 1 then leave
   n ← n + 1
RT32:63 ← n
```

The leading zero bits in each element of RA are counted, and the respective count is placed into each element of RT.

Special Registers Altered:
None

Vector Divide Word Signed EVX-form

evdivws RT,RA,RB

4	RT	RA	RB	1222
0	6	11	16	21 31

```
ddh ← (RA)0:31
ddl ← (RA)32:63
dvh ← (RB)0:31
dvl ← (RB)32:63
RT0:31 ← ddh ÷ dvh
RT32:63 ← ddl ÷ dvl
ovh ← 0
ovl ← 0
if ((ddh < 0) & (dvh = 0)) then
   RT0:31 ← 0x8000_0000
   ovh ← 1
else if ((ddh >= 0) & (dvh = 0)) then
   RT0:31 ← 0x7FFFFFFF
   ovh ← 1
else if (ddh = 0x8000_0000)&(dvh = 0xFFFF_FFFF)
then
   RT0:31 ← 0x7FFFFFFF
   ovh ← 1
if ((ddl < 0) & (dvl = 0)) then
   RT32:63 ← 0x8000_0000
   ovl ← 1
else if ((ddl >= 0) & (dvl = 0)) then
   RT32:63 ← 0x7FFFFFFF
   ovl ← 1
else if (ddl = 0x8000_0000)&(dvl = 0xFFFF_FFFF)
then
   RT32:63 ← 0x7FFFFFFF
   ovl ← 1
SPEFSCROVH ← ovh
SPEFSCROV ← ovl
SPEFSCRSOVH ← SPEFSCRSOVH | ovh
SPEFSCRSOV ← SPEFSCRSOV | ovl
```

The two dividends are the two elements of the contents of RA. The two divisors are the two elements of the contents of RB. The resulting two 32-bit quotients on each element are placed into RT. The remainders are not supplied. The operands and quotients are interpreted as signed integers.

Special Registers Altered:
OV OVH SOV SOVH

Programming Note

Note that any overflow indication is always set as a side effect of this instruction. No form is defined that disables the setting of the overflow bits. In case of overflow, a saturated value is delivered into the destination register.

Vector Divide Word Unsigned *EVX-form*

evdivwu RT,RA,RB

4	RT	RA	RB	1223
0	6	11	16	21 31

```
ddh ← (RA)0:31
ddl ←(RA)32:63
dvh ← (RB)0:31
dvl ← (RB)32:63
RT0:31 ← ddh ÷ dvh
RT32:63 ← ddl ÷ dvl
ovh ← 0
ovl ← 0
if (dvh = 0) then
   RT0:31 ← 0xFFFFFFFF
   ovh ← 1
if (dvl = 0) then
    RT32:63 ← 0xFFFFFFFF
    ovl ← 1
SPEFSCROVH ← ovh
SPEFSCROV ← ovl
SPEFSCRSOVH ← SPEFSCRSOVH | ovh
SPEFSCRSOV ← SPEFSCRSOV | ovl
```

The two dividends are the two elements of the contents of RA. The two divisors are the two elements of the contents of RB. Two 32-bit quotients are formed as a result of the division on each of the high and low elements and the quotients are placed into RT. Remainders are not supplied. Operands and quotients are interpreted as unsigned integers.

Special Registers Altered:
OV OVH SOV SOVH

Programming Note

Note that any overflow indication is always set as a side effect of this instruction. No form is defined that disables the setting of the overflow bits. In case of overflow, a saturated value is delivered into the destination register.

Vector Equivalent *EVX-form*

eveqv RT,RA,RB

4	RT	RA	RB	537
0	6	11	16	21 31

```
RT0:31 ← (RA)0:31 ≡ (RB)0:31
RT32:63 ← (RA)32:63 ≡ (RB)32:63
```

The corresponding elements of RA and RB are XORed bitwise, and the complemented results are placed in RT.

Special Registers Altered:
None

Vector Extend Sign Byte *EVX-form*

evextsb RT,RA

4	RT	RA	///	522
0	6	11	16	21 31

```
RT0:31 ← EXTS((RA)24:31)
RT32:63 ← EXTS((RA)56:63)
```

The signs of the low-order byte in each of the elements in RA are extended, and the results are placed in RT.

Special Registers Altered:
None

Vector Extend Sign Halfword *EVX-form*

evextsh RT,RA

4	RT	RA	///	523
0	6	11	16	21 31

```
RT0:31 ← EXTS((RA)16:31)
RT32:63 ← EXTS((RA)48:63)
```

The signs of the odd halfwords in each of the elements in RA are extended, and the results are placed in RT.

Special Registers Altered:
None

Vector Load Double Word into Double Word *EVX-form*

evldd RT,D(RA)

4	RT	RA	UI	769
0	6	11	16	21 31

```
if (RA = 0) then b ← 0
else b ← (RA)
EA ← b + EXTZ(UI×8)
RT ← MEM(EA, 8)
```

D in the instruction mnemonic is UI × 8. The doubleword addressed by EA is loaded from memory and placed in RT.

Special Registers Altered:
None

Vector Load Double Word into Double Word Indexed *EVX-form*

evlddx RT,RA,RB

4	RT	RA	RB	768
0	6	11	16	21 31

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

The doubleword addressed by EA is loaded from memory and placed in RT.

Special Registers Altered:
None

Vector Load Double into Four Halfwords *EVX-form*

evldh RT,D(RA)

4	RT	RA	UI	773
0	6	11	16	21 31

```
if (RA = 0) then b ← 0
else b ← (RA)
EA ← b + EXTZ(UI×8)
RT0:15 ← MEM(EA, 2)
RT16:31 ← MEM(EA+2,2)
RT32:47 ← MEM(EA+4,2)
RT48:63 ← MEM(EA+6,2)
```

D in the instruction mnemonic is UI × 8. The doubleword addressed by EA is loaded from memory and placed in RT.

Special Registers Altered:
None

Vector Load Double into Four Halfwords Indexed *EVX-form*

evldhx RT,RA,RB

4	RT	RA	RB	772
0	6	11	16	21 31

```
if (RA = 0) then b ← 0
else b ← (RA)
EA ← b + (RB)
RT0:15 ← MEM(EA, 2)
RT16:31 ← MEM(EA+2,2)
RT32:47 ← MEM(EA+4,2)
RT48:63 ← MEM(EA+6,2)
```

The doubleword addressed by EA is loaded from memory and placed in RT.

Special Registers Altered:
None

Vector Load Double into Two Words EVX-form

evldw RT,D(RA)

4	RT	RA	UI	771
0	6	11	16	21 31

```
if (RA = 0) then b ← 0
else b ← (RA)
EA ← b + EXTZ(UI×8)
RT0:31 ← MEM(EA, 4)
RT32:63 ← MEM(EA+4, 4)
```

D in the instruction mnemonic is UI × 8. The doubleword addressed by EA is loaded from memory and placed in RT.

Special Registers Altered:
None

Vector Load Double into Two Words Indexed EVX-form

evldwx RT,RA,RB

4	RT	RA	RB	770
0	6	11	16	21 31

```
if (RA = 0) then b ← 0
else b ← (RA)
EA ← b + (RB)
RT0:31 ← MEM(EA, 4)
RT32:63 ← MEM(EA+4, 4)
```

The doubleword addressed by EA is loaded from memory and placed in RT.

Special Registers Altered:
None

Vector Load Halfword into Halfwords Even and Splat EVX-form

evlhhesplat RT,D(RA)

4	RT	RA	UI	777
0	6	11	16	21 31

```
if (RA = 0) then b ← 0
else b ← (RA)
EA ← b + EXTZ(UI×2)
RT0:15 ← MEM(EA,2)
RT16:31 ← 0x0000
RT32:47 ← MEM(EA,2)
RT48:63 ← 0x0000
```

D in the instruction mnemonic is UI × 2. The halfword addressed by EA is loaded from memory and placed in the even halfwords of each element of RT. The odd halfwords of each element of RT are set to 0.

Special Registers Altered:
None

Vector Load Halfword into Halfwords Even and Splat Indexed EVX-form

evlhhesplatx RT,RA,RB

4	RT	RA	RB	776
0	6	11	16	21 31

```
if (RA = 0) then b ← 0
else b ← (RA)
EA ← b + (RB)
RT0:15 ← MEM(EA,2)
RT16:31 ← 0x0000
RT32:47 ← MEM(EA,2)
RT48:63 ← 0x0000
```

The halfword addressed by EA is loaded from memory and placed in the even halfwords of each element of RT. The odd halfwords of each element of RT are set to 0.

Special Registers Altered:
None

Vector Load Halfword into Halfword Odd Signed and Splat EVX-form

evlhhossplat RT,D(RA)

4	RT	RA	UI	783
0	6	11	16	21 31

```
if (RA = 0) then b ← 0
else b ← (RA)
EA ← b + EXTZ(UI×2)
RT0:31 ← EXTS(MEM(EA,2))
RT32:63 ← EXTS(MEM(EA,2))
```

D in the instruction mnemonic is UI × 2. The halfword addressed by EA is loaded from memory and placed in the odd halfwords sign extended in each element of RT.

Special Registers Altered:

None

Vector Load Halfword into Halfword Odd Signed and Splat Indexed EVX-form

evlhhossplatx RT,RA,RB

4	RT	RA	RB	782
0	6	11	16	21 31

```
if (RA = 0) then b ← 0
else b ← (RA)
EA ← b + (RB)
RT0:31 ← EXTS(MEM(EA,2))
RT32:63 ← EXTS(MEM(EA,2))
```

The halfword addressed by EA is loaded from memory and placed in the odd halfwords sign extended in each element of RT.

Special Registers Altered:

None

Vector Load Halfword into Halfword Odd Unsigned and Splat EVX-form

evlhhousplat RT,D(RA)

4	RT	RA	UI	781
0	6	11	16	21 31

```
if (RA = 0) then b ← 0
else b ← (RA)
EA ← b + EXTZ(UI×2)
RT0:31 ← EXTZ(MEM(EA,2))
RT32:63 ← EXTZ(MEM(EA,2))
```

D in the instruction mnemonic is UI × 2. The halfword addressed by EA is loaded from memory and placed in the odd halfwords zero-extended in each element of RT.

Special Registers Altered:

None

Vector Load Halfword into Halfword Odd Unsigned and Splat Indexed EVX-form

evlhhousplatx RT,RA,RB

4	RT	RA	RB	780
0	6	11	16	21 31

```
if (RA = 0) then b ← 0
else b ← (RA)
EA ← b + (RB)
RT0:31 ← EXTZ(MEM(EA,2))
RT32:63 ← EXTZ(MEM(EA,2))
```

The halfword addressed by EA is loaded from memory and placed in the odd halfwords zero-extended in each element of RT.

Special Registers Altered:

None

Vector Load Word into Two Halfwords Even *EVX-form*

evlwhe RT,D(RA)

4	RT	RA	UI	785
0	6	11	16	21 31

```
if (RA = 0) then b ← 0
else b ← (RA)
EA ← b + EXTZ(UI×4)
RT0:15 ← MEM(EA,2)
RT16:31 ← 0x0000
RT32:47 ← MEM(EA+2,2)
RT48:63 ← 0x0000
```

D in the instruction mnemonic is UI × 4. The word addressed by EA is loaded from memory and placed in the even halfwords of each element of RT. The odd halfwords of each element of RT are set to 0.

Special Registers Altered:
None

Vector Load Word into Two Halfwords Even Indexed *EVX-form*

evlwhex RT,RA,RB

4	RT	RA	RB	784
0	6	11	16	21 31

```
if (RA = 0) then b ← 0
else b ← (RA)
EA ← b + (RB)
RT0:15 ← MEM(EA,2)
RT16:31 ← 0x0000
RT32:47 ← MEM(EA+2,2)
RT48:63 ← 0x0000
```

The word addressed by EA is loaded from memory and placed in the even halfwords in each element of RT. The odd halfwords of each element of RT are set to 0.

Special Registers Altered:
None

Vector Load Word into Two Halfwords Odd Signed (with sign extension) *EVX-form*

evlwhos RT,D(RA)

4	RT	RA	UI	791
0	6	11	16	21 31

```
if (RA = 0) then b ← 0
else b ← (RA)
EA ← b + EXTZ(UI×4)
RT0:31 ← EXTS(MEM(EA,2))
RT32:63 ← EXTS(MEM(EA+2,2))
```

D in the instruction mnemonic is UI × 4. The word addressed by EA is loaded from memory and placed in the odd halfwords sign extended in each element of RT.

Special Registers Altered:
None

Vector Load Word into Two Halfwords Odd Signed Indexed (with sign extension) *EVX-form*

evlwhosx RT,RA,RB

4	RT	RA	RB	790
0	6	11	16	21 31

```
if (RA = 0) then b ← 0
else b ← (RA)
EA ← b + (RB)
RT0:31 ← EXTS(MEM(EA,2))
RT32:63 ← EXTS(MEM(EA+2,2))
```

The word addressed by EA is loaded from memory and placed in the odd halfwords sign extended in each element of RT.

Special Registers Altered:
None

Vector Load Word into Two Halfwords Odd Unsigned (zero-extended) EVX-form

evlwhou RT,D(RA)

4	RT	RA	UI	789
0	6	11	16	21 31

```
if (RA = 0) then b ← 0
else b ← (RA)
EA ← b + EXTZ(UI×4)
RT0:31 ← EXTZ(MEM(EA,2))
RT32:63 ← EXTZ(MEM(EA+2,2))
```

D in the instruction mnemonic is UI × 4. The word addressed by EA is loaded from memory and placed in the odd halfwords zero-extended in each element of RT.

Special Registers Altered:
None

Vector Load Word into Two Halfwords Odd Unsigned Indexed (zero-extended) EVX-form

evlwhoux RT,RA,RB

4	RT	RA	RB	788
0	6	11	16	21 31

```
if (RA = 0) then b ← 0
else b ← (RA)
EA ← b + (RB)
RT0:31 ← EXTZ(MEM(EA,2))
RT32:63 ← EXTZ(MEM(EA+2,2))
```

The word addressed by EA is loaded from memory and placed in the odd halfwords zero-extended in each element of RT.

Special Registers Altered:
None

Vector Load Word into Two Halfwords and Splat EVX-form

evlwhsplat RT,D(RA)

4	RT	RA	UI	797
0	6	11	16	21 31

```
if (RA = 0) then b ← 0
else b ← (RA)
EA ← b + EXTZ(UI×4)
RT0:15 ← MEM(EA,2)
RT16:31 ← MEM(EA,2)
RT32:47 ← MEM(EA+2,2)
RT48:63 ← MEM(EA+2,2)
```

D in the instruction mnemonic is UI × 4. The word addressed by EA is loaded from memory and placed in both the even and odd halfwords in each element of RT.

Special Registers Altered:
None

Vector Load Word into Two Halfwords and Splat Indexed EVX-form

evlwhsplatx RT,RA,RB

4	RT	RA	RB	796
0	6	11	16	21 31

```
if (RA = 0) then b ← 0
else b ← (RA)
EA ← b + (RB)
RT0:15 ← MEM(EA,2)
RT16:31 ← MEM(EA,2)
RT32:47 ← MEM(EA+2,2)
RT48:63 ← MEM(EA+2,2)
```

The word addressed by EA is loaded from memory and placed in both the even and odd halfwords in each element of RT.

Special Registers Altered:
None

Vector Load Word into Word and Splat EVX-form

evlwwsplat RT,D(RA)

4	RT	RA	UI	793
0	6	11	16	21 31

```
if (RA = 0) then b ← 0
else b ← (RA)
EA ← b + EXTZ(UI×4)
RT0:31 ← MEM(EA,4)
RT32:63 ← MEM(EA,4)
```

D in the instruction mnemonic is UI × 4. The word addressed by EA is loaded from memory and placed in both elements of RT.

Special Registers Altered:
None

Vector Load Word into Word and Splat Indexed EVX-form

evlwwsplatx RT,RA,RB

4	RT	RA	RB	792
0	6	11	16	21 31

```
if (RA = 0) then b ← 0
else b ← (RA)
EA ← b + (RB)
RT0:31 ← MEM(EA,4)
RT32:63 ← MEM(EA,4)
```

The word addressed by EA is loaded from memory and placed in both elements of RT.

Special Registers Altered:
None

Vector Merge High EVX-form

evmergehi RT,RA,RB

4	RT	RA	RB	556
0	6	11	16	21 31

$RT_{0:31} \leftarrow (RA)_{0:31}$
$RT_{32:63} \leftarrow (RB)_{0:31}$

The high-order elements of RA and RB are merged and placed in RT.

Special Registers Altered:
None

Programming Note

A vector splat high can be performed by specifying the same register in RA and RB.

Vector Merge Low EVX-form

evmergelo RT,RA,RB

4	RT	RA	RB	557
0	6	11	16	21 31

$RT_{0:31} \leftarrow (RA)_{32:63}$
$RT_{32:63} \leftarrow (RB)_{32:63}$

The low-order elements of RA and RB are merged and placed in RT.

Special Registers Altered:
None

Programming Note

A vector splat low can be performed by specifying the same register in RA and RB.

Vector Merge High/Low EVX-form

evmergehilo RT,RA,RB

4	RT	RA	RB	558
0	6	11	16	21 31

$RT_{0:31} \leftarrow (RA)_{0:31}$
$RT_{32:63} \leftarrow (RB)_{32:63}$

The high-order element of RA and the low-order element of RB are merged and placed in RT.

Special Registers Altered:
None

Programming Note

With appropriate specification of RA and RB, ***evmergehi***, ***evmergelo***, ***evmergehilo***, and ***evmergelohi*** provide a full 32-bit permute of two source operands.

Vector Merge Low/High EVX-form

evmergelohi RT,RA,RB

4	RT	RA	RB	559
0	6	11	16	21 31

$RT_{0:31} \leftarrow (RA)_{32:63}$
$RT_{32:63} \leftarrow (RB)_{0:31}$

The low-order element of RA and the high-order element of RB are merged and placed in RT.

Special Registers Altered:
None

Programming Note

A vector swap can be performed by specifying the same register in RA and RB.

Vector Multiply Halfwords, Even, Guarded, Signed, Modulo, Fractional and Accumulate EVX-form

evmhegsmfaa RT,RA,RB

4	RT	RA	RB	1323
0	6	11	16	21 31

$temp_{0:63} \leftarrow (RA)_{32:47} \times_{gsf} (RB)_{32:47}$
$RT_{0:63} \leftarrow (ACC)_{0:63} + temp_{0:63}$
$ACC_{0:63} \leftarrow (RT)_{0:63}$

The corresponding low even-numbered, halfword signed fractional elements in RA and RB are multiplied using guarded signed fractional multiplication producing a sign extended 64-bit fractional product with the decimal between bits 32 and 33. The product is added to the contents of the 64-bit accumulator and the result is placed in RT and the accumulator

Special Registers Altered:
ACC

Note

If the two input operands are both -1.0, the intermediate product is represented as +1.0.

Vector Multiply Halfwords, Even, Guarded, Signed, Modulo, Fractional and Accumulate Negative EVX-form

evmhegsmfan RT,RA,RB

4	RT	RA	RB	1451
0	6	11	16	21 31

$temp_{0:63} \leftarrow (RA)_{32:47} \times_{gsf} (RB)_{32:47}$
$RT_{0:63} \leftarrow (ACC)_{0:63} - temp_{0:63}$
$ACC_{0:63} \leftarrow (RT)_{0:63}$

The corresponding low even-numbered, halfword signed fractional elements in RA and RB are multiplied using guarded signed fractional multiplication producing a sign extended 64-bit fractional product with the decimal between bits 32 and 33. The product is subtracted from the contents of the 64-bit accumulator and the result is placed in RT and the accumulator.

Special Registers Altered:
ACC

Note

If the two input operands are both -1.0, the intermediate product is represented as +1.0.

Vector Multiply Halfwords, Even, Guarded, Signed, Modulo, Integer and Accumulate EVX-form

evmhegsmiaa RT,RA,RB

4	RT	RA	RB	1321
0	6	11	16	21 31

$temp_{0:31} \leftarrow (RA)_{32:47} \times_{si} (RB)_{32:47}$
$temp_{0:63} \leftarrow EXTS(temp_{0:31})$
$RT_{0:63} \leftarrow (ACC)_{0:63} + temp_{0:63}$
$ACC_{0:63} \leftarrow (RT)_{0:63}$

The corresponding low even-numbered halfword signed-integer elements in RA and RB are multiplied. The intermediate product is sign-extended and added to the contents of the 64-bit accumulator, and the resulting sum is placed in RT and into the accumulator.

Special Registers Altered:
ACC

Vector Multiply Halfwords, Even, Guarded, Unsigned, Modulo, Integer and Accumulate EVX-form

evmhegumiaa RT,RA,RB

4	RT	RA	RB	1320
0	6	11	16	21 31

$temp_{0:31} \leftarrow (RA)_{32:47} \times_{ui} (RB)_{32:47}$
$temp_{0:63} \leftarrow EXTZ(temp_{0:31})$
$RT_{0:63} \leftarrow (ACC)_{0:63} + temp_{0:63}$
$ACC_{0:63} \leftarrow (RT)_{0:63}$

The corresponding low even-numbered halfword unsigned-integer elements in RA and RB are multiplied. The intermediate product is zero-extended and added to the contents of the 64-bit accumulator. The resulting sum is placed in RT and into the accumulator.

Special Registers Altered:
ACC

Vector Multiply Halfwords, Even, Guarded, Signed, Modulo, Integer and Accumulate Negative EVX-form

evmhegsmian RT,RA,RB

4	RT	RA	RB	1449
0	6	11	16	21 31

$temp_{0:31} \leftarrow (RA)_{32:47} \times_{si} (RB)_{32:47}$
$temp_{0:63} \leftarrow EXTS(temp_{0:31})$
$RT_{0:63} \leftarrow (ACC)_{0:63} - temp_{0:63}$
$ACC_{0:63} \leftarrow (RT)_{0:63}$

The corresponding low even-numbered halfword signed-integer elements in RA and RB are multiplied. The intermediate product is sign-extended and subtracted from the contents of the 64-bit accumulator, and the result is placed in RT and into the accumulator.

Special Registers Altered:
ACC

Vector Multiply Halfwords, Even, Guarded, Unsigned, Modulo, Integer and Accumulate Negative EVX-form

evmhegumian RT,RA,RB

4	RT	RA	RB	1448
0	6	11	16	21 31

$temp_{0:31} \leftarrow (RA)_{32:47} \times_{ui} (RB)_{32:47}$
$temp_{0:63} \leftarrow EXTZ(temp_{0:31})$
$RT_{0:63} \leftarrow (ACC)_{0:63} - temp_{0:63}$
$ACC_{0:63} \leftarrow (RT)_{0:63}$

The corresponding low even-numbered unsigned-integer elements in RA and RB are multiplied. The intermediate product is zero-extended and subtracted from the contents of the 64-bit accumulator. The result is placed in RT and into the accumulator.

Special Registers Altered:
ACC

Vector Multiply Halfwords, Even, Signed, Modulo, Fractional EVX-form

evmhesmf RT,RA,RB

4	RT	RA	RB	1035
0	6	11	16	21 31

$RT_{0:31} \leftarrow (RA)_{0:15} \times_{sf} (RB)_{0:15}$
$RT_{32:63} \leftarrow (RA)_{32:47} \times_{sf} (RB)_{32:47}$

The corresponding even-numbered halfword signed fractional elements in RA and RB are multiplied then placed into the corresponding words of RT.

Special Registers Altered:
None

Vector Multiply Halfwords, Even, Signed, Modulo, Fractional to Accumulator EVX-form

evmhesmfa RT,RA,RB

4	RT	RA	RB	1067
0	6	11	16	21 31

$RT_{0:31} \leftarrow (RA)_{0:15} \times_{sf} (RB)_{0:15}$

$RT_{32:63} \leftarrow (RA)_{32:47} \times_{sf} (RB)_{32:47}$
$ACC_{0:63} \leftarrow (RT)_{0:63}$

The corresponding even-numbered halfword signed fractional elements in RA and RB are multiplied then placed into the corresponding words of RT and into the accumulator.

Special Registers Altered:
ACC

Vector Multiply Halfwords, Even, Signed, Modulo, Fractional and Accumulate into Words EVX-form

evmhesmfaaw RT,RA,RB

4	RT	RA	RB	1291
0	6	11	16	21 31

$temp_{0:31} \leftarrow (RA)_{0:15} \times_{sf} (RB)_{0:15}$
$RT_{0:31} \leftarrow (ACC)_{0:31} + temp_{0:31}$
$temp_{0:31} \leftarrow (RA)_{32:47} \times_{sf} (RB)_{32:47}$
$RT_{32:63} \leftarrow (ACC)_{32:63} + temp_{0:31}$
$ACC_{0:63} \leftarrow (RT)_{0:63}$

For each word element in the accumulator, the corresponding even-numbered halfword signed fractional elements in RA and RB are multiplied. The 32 bits of each intermediate product are added to the contents of the accumulator words to form intermediate sums, which are placed into the corresponding RT words and into the accumulator.

Special Registers Altered:
ACC

Vector Multiply Halfwords, Even, Signed, Modulo, Fractional and Accumulate Negative into Words EVX-form

evmhesmfanw RT,RA,RB

4	RT	RA	RB	1419
0	6	11	16	21 31

$temp_{0:31} \leftarrow (RA)_{0:15} \times_{sf} (RB)_{0:15}$
$RT_{0:31} \leftarrow (ACC)_{0:31} - temp_{0:31}$

$temp_{0:31} \leftarrow (RA)_{32:47} \times_{sf} (RB)_{32:47}$
$RT_{32:63} \leftarrow (ACC)_{32:63} - temp_{0:31}$
$ACC_{0:63} \leftarrow (RT)_{0:63}$

For each word element in the accumulator, the corresponding even-numbered halfword signed fractional elements in RA and RB are multiplied. The 32-bit intermediate products are subtracted from the contents of the accumulator words to form intermediate differences, which are placed into the corresponding RT words and into the accumulator.

Special Registers Altered:
ACC

Vector Multiply Halfwords, Even, Signed, Modulo, Integer EVX-form

evmhesmi RT,RA,RB

4	RT	RA	RB	1033
0	6	11	16	21 31

$RT_{0:31} \leftarrow (RA)_{0:15} \times_{si} (RB)_{0:15}$
$RT_{32:63} \leftarrow (RA)_{32:47} \times_{si} (RB)_{32:47}$

The corresponding even-numbered halfword signed-integer elements in RA and RB are multiplied. The two 32-bit products are placed into the corresponding words of RT.

Special Registers Altered:
None

Vector Multiply Halfwords, Even, Signed, Modulo, Integer to Accumulator EVX-form

evmhesmia RT,RA,RB

4	RT	RA	RB	1065
0	6	11	16	21 31

$RT_{0:31} \leftarrow (RA)_{0:15} \times_{si} (RB)_{0:15}$
$RT_{32:63} \leftarrow (RA)_{32:47} \times_{si} (RB)_{32:47}$
$ACC_{0:63} \leftarrow (RT)_{0:63}$

The corresponding even-numbered halfword signed-integer elements in RA and RB are multiplied. The two 32-bit products are placed into the corresponding words of RT and into the accumulator.

Special Registers Altered:
ACC

Vector Multiply Halfwords, Even, Signed, Modulo, Integer and Accumulate into Words EVX-form

evmhesmiaaw RT,RA,RB

4	RT	RA	RB	1289
0	6	11	16	21 31

$temp_{0:31} \leftarrow (RA)_{0:15} \times_{si} (RB)_{0:15}$
$RT_{0:31} \leftarrow (ACC)_{0:31} + temp_{0:31}$
$temp_{0:31} \leftarrow (RA)_{32:47} \times_{si} (RB)_{32:47}$
$RT_{32:63} \leftarrow (ACC)_{32:63} + temp_{0:31}$
$ACC_{0:63} \leftarrow (RT)_{0:63}$

For each word element in the accumulator, the corresponding even-numbered halfword signed-integer elements in RA and RB are multiplied. Each intermediate 32-bit product is added to the contents of the accumulator words to form intermediate sums, which are placed into the corresponding RT words and into the accumulator.

Special Registers Altered:
ACC

Vector Multiply Halfwords, Even, Signed, Modulo, Integer and Accumulate Negative into Words EVX-form

evmhesmianw RT,RA,RB

4	RT	RA	RB	1417
0	6	11	16	21 31

$temp_{0:31} \leftarrow (RA)_{0:15} \times_{si} (RB)_{0:15}$
$RT_{0:31} \leftarrow (ACC)_{0:31} - temp_{0:31}$
$temp_{0:31} \leftarrow (RA)_{32:47} \times_{si} (RB)_{32:47}$
$RT_{32:63} \leftarrow (ACC)_{32:63} - temp_{0:31}$
$ACC_{0:63} \leftarrow (RT)_{0:63}$

For each word element in the accumulator, the corresponding even-numbered halfword signed-integer elements in RA and RB are multiplied. Each intermediate 32-bit product is subtracted from the contents of the accumulator words to form intermediate differences, which are placed into the corresponding RT words and into the accumulator.

Special Registers Altered:
ACC

Vector Multiply Halfwords, Even, Signed, Saturate, Fractional EVX-form

evmhessf RT,RA,RB

4	RT	RA	RB	1027
0	6	11	16	21 31

```
temp0:31 ← (RA)0:15 ×sf (RB)0:15
if ((RA)0:15 = 0x8000) & ((RB)0:15 = 0x8000) then
   RT0:31 ← 0x7FFF_FFFF
   movh ← 1
else
   RT0:31 ← temp0:31
   movh ← 0
temp0:31 ← (RA)32:47 ×sf (RB)32:47
if ((RA)32:47 = 0x8000) & ((RB)32:47 = 0x8000) then
   RT32:63 ← 0x7FFF_FFFF
   movl ← 1
else
   RT32:63 ← temp0:31
   movl ← 0
SPEFSCROVH ← movh
SPEFSCROV ← movl
SPEFSCRSOVH ← SPEFSCRSOVH | movh
SPEFSCRSOV ← SPEFSCRSOV | movl
```

The corresponding even-numbered halfword signed fractional elements in RA and RB are multiplied. The 32 bits of each product are placed into the corresponding words of RT. If both inputs are -1.0, the result saturates to the largest positive signed fraction.

Special Registers Altered:
OV OVH SOV SOVH

Vector Multiply Halfwords, Even, Signed, Saturate, Fractional to Accumulator EVX-form

evmhessfa RT,RA,RB

4	RT	RA	RB	1059
0	6	11	16	21 31

```
temp0:31 ← (RA)0:15 ×sf (RB)0:15
if ((RA)0:15 = 0x8000) & ((RB)0:15 = 0x8000) then
   RT0:31 ← 0x7FFF_FFFF
   movh ← 1
else
   RT0:31 ← temp0:31
   movh ← 0
temp0:31 ← (RA)32:47 ×sf (RB)32:47
if ((RA)32:47 = 0x8000) & ((RB)32:47 = 0x8000) then
   RT32:63 ← 0x7FFF_FFFF
   movl ← 1
else
   RT32:63 ← temp0:31
   movl ← 0
ACC0:63 ← (RT)0:63
SPEFSCROVH ← movh
SPEFSCROV ← movl
SPEFSCRSOVH ← SPEFSCRSOVH | movh
SPEFSCRSOV ← SPEFSCRSOV | movl
```

The corresponding even-numbered halfword signed fractional elements in RA and RB are multiplied. The 32 bits of each product are placed into the corresponding words of RT and into the accumulator. If both inputs are -1.0, the result saturates to the largest positive signed fraction.

Special Registers Altered:
ACC OV OVH SOV SOVH

Vector Multiply Halfwords, Even, Signed, Saturate, Fractional and Accumulate into Words *EVX-form*

evmhessfaaw RT,RA,RB

4	RT	RA	RB	1283
0	6	11	16	21 31

```
temp0:31 ← (RA)0:15 ×sf (RB)0:15
if ((RA)0:15 = 0x8000) & ((RB)0:15 = 0x8000) then
   temp0:31 ← 0x7FFF_FFFF
   movh ← 1
else
   movh ← 0
temp0:63 ← EXTS((ACC)0:31) + EXTS(temp0:31)
ovh ← (temp31 ⊕ temp32)
RT0:31 ← SATURATE(ovh, temp31, 0x8000_0000,
                 0x7FFF_FFFF, temp32:63)

temp0:31 ← (RA)32:47 ×sf (RB)32:47
if ((RA)32:47 = 0x8000) & ((RB)32:47 = 0x8000) then
   temp0:31 ← 0x7FFF_FFFF
   movl ← 1
else
   movl ← 0
temp0:63 ← EXTS((ACC)32:63) + EXTS(temp0:31)
ovl ← (temp31 ⊕ temp32)
RT32:63 ← SATURATE(ovl, temp31, 0x8000_0000,
                  0x7FFF_FFFF, temp32:63)
ACC0:63 ← (RT)0:63
SPEFSCROVH ← ovh | movh
SPEFSCROV ← ovl| movl
SPEFSCRSOVH ← SPEFSCRSOVH | ovh | movh
SPEFSCRSOV ← SPEFSCRSOV | ovl| movl
```

The corresponding even-numbered halfword signed fractional elements in RA and RB are multiplied producing a 32-bit product. If both inputs are -1.0, the result saturates to 0x7FFF_FFFF. Each 32-bit product is then added to the corresponding word in the accumulator saturating if overflow occurs, and the result is placed in RT and the accumulator.

Special Registers Altered:

ACC OV OVH SOV SOVH

Vector Multiply Halfwords, Even, Signed, Saturate, Fractional and Accumulate Negative into Words *EVX-form*

evmhessfanw RT,RA,RB

4	RT	RA	RB	1411
0	6	11	16	21 31

```
temp0:31 ← (RA)0:15 ×sf (RB)0:15
if ((RA)0:15 = 0x8000) & ((RB)0:15 = 0x8000) then
   temp0:31 ← 0x7FFF_FFFF
   movh ← 1
else
   movh ← 0
temp0:63 ← EXTS((ACC)0:31) - EXTS(temp0:31)
ovh ← (temp31 ⊕ temp32)
RT0:31 ← SATURATE(ovh, temp31, 0x8000_0000,
                 0x7FFF_FFFF, temp32:63)
temp0:31 ← (RA)32:47 ×sf (RB)32:47
if ((RA)32:47 = 0x8000) & ((RB)32:47 = 0x8000) then
   temp0:31 ← 0x7FFF_FFFF
   movl ← 1
else
   movl ← 0
temp0:63 ← EXTS((ACC)32:63) - EXTS(temp0:31)
ovl ← (temp31 ⊕ temp32)
RT32:63 ← SATURATE(ovl, temp31, 0x8000_0000,
                  0x7FFF_FFFF, temp32:63)
ACC0:63 ← (RT)0:63
SPEFSCROVH ← ovh | movh
SPEFSCROV ← ovl| movl
SPEFSCRSOVH ← SPEFSCRSOVH | ovh | movh
SPEFSCRSOV ← SPEFSCRSOV | ovl| movl
```

The corresponding even-numbered halfword signed fractional elements in RA and RB are multiplied producing a 32-bit product. If both inputs are -1.0, the result saturates to 0x7FFF_FFFF. Each 32-bit product is then subtracted from the corresponding word in the accumulator saturating if overflow occurs, and the result is placed in RT and the accumulator.

Special Registers Altered:

ACC OV OVH SOV SOVH

Vector Multiply Halfwords, Even, Signed, Saturate, Integer and Accumulate into Words EVX-form

evmhessiaaw RT,RA,RB

4	RT	RA	RB	1281
0	6	11	16	21 … 31

$temp_{0:31} \leftarrow (RA)_{0:15} \times_{si} (RB)_{0:15}$
$temp_{0:63} \leftarrow EXTS((ACC)_{0:31}) + EXTS(temp_{0:31})$
$ovh \leftarrow (temp_{31} \oplus temp_{32})$
$RT_{0:31} \leftarrow SATURATE(ovh,\ temp_{31},\ 0x8000\_0000,\ 0x7FFF\_FFFF,\ temp_{32:63})$

$temp_{0:31} \leftarrow (RA)_{32:47} \times_{si} (RB)_{32:47}$
$temp_{0:63} \leftarrow EXTS((ACC)_{32:63}) + EXTS(temp_{0:31})$
$ovl \leftarrow (temp_{31} \oplus temp_{32})$
$RT_{32:63} \leftarrow SATURATE(ovl,\ temp_{31},\ 0x8000\_0000,\ 0x7FFF\_FFFF,\ temp_{32:63})$
$ACC_{0:63} \leftarrow (RT)_{0:63}$
$SPEFSCR_{OVH} \leftarrow ovh$
$SPEFSCR_{OV} \leftarrow ovl$
$SPEFSCR_{SOVH} \leftarrow SPEFSCR_{SOVH}\ |\ ovh$
$SPEFSCR_{SOV} \leftarrow SPEFSCR_{SOV}\ |\ ovl$

The corresponding even-numbered halfword signed-integer elements in RA and RB are multiplied producing a 32-bit product. Each 32-bit product is then added to the corresponding word in the accumulator saturating if overflow occurs, and the result is placed in RT and the accumulator.

Special Registers Altered:
ACC OV OVH SOV SOVH

Vector Multiply Halfwords, Even, Signed, Saturate, Integer and Accumulate Negative into Words EVX-form

evmhessianw RT,RA,RB

4	RT	RA	RB	1409
0	6	11	16	21 … 31

$temp_{0:31} \leftarrow (RA)_{0:15} \times_{si} (RB)_{0:15}$
$temp_{0:63} \leftarrow EXTS((ACC)_{0:31}) - EXTS(temp_{0:31})$
$ovh \leftarrow (temp_{31} \oplus temp_{32})$
$RT_{0:31} \leftarrow SATURATE(ovh,\ temp_{31},\ 0x8000\_0000,\ 0x7FFF\_FFFF,\ temp_{32:63})$
$temp_{0:31} \leftarrow (RA)_{32:47} \times_{si} (RB)_{32:47}$
$temp_{0:63} \leftarrow EXTS((ACC)_{32:63}) - EXTS(temp_{0:31})$
$ovl \leftarrow (temp_{31} \oplus temp_{32})$
$RT_{32:63} \leftarrow SATURATE(ovl,\ temp_{31},\ 0x8000\_0000,\ 0x7FFF\_FFFF,\ temp_{32:63})$
$ACC_{0:63} \leftarrow RT_{0:63}$
$SPEFSCR_{OVH} \leftarrow ovh$
$SPEFSCR_{OV} \leftarrow ovl$
$SPEFSCR_{SOVH} \leftarrow SPEFSCR_{SOVH}\ |\ ovh$
$SPEFSCR_{SOV} \leftarrow SPEFSCR_{SOV}\ |\ ovl$

The corresponding even-numbered halfword signed-integer elements in RA and RB are multiplied producing a 32-bit product. Each 32-bit product is then subtracted from the corresponding word in the accumulator saturating if overflow occurs, and the result is placed in RT and the accumulator.

Special Registers Altered:
ACC OV OVH SOV SOVH

Vector Multiply Halfwords, Even, Unsigned, Modulo, Integer EVX-form

evmheumi RT,RA,RB

4	RT	RA	RB	1032
0	6	11	16	21 31

$RT_{0:31} \leftarrow (RA)_{0:15} \times_{ui} (RB)_{0:15}$
$RT_{32:63} \leftarrow (RA)_{32:47} \times_{ui} (RB)_{32:47}$

The corresponding even-numbered halfword unsigned-integer elements in RA and RB are multiplied. The two 32-bit products are placed into the corresponding words of RT.

Special Registers Altered:
None

Vector Multiply Halfwords, Even, Unsigned, Modulo, Integer to Accumulator EVX-form

evmheumia RT,RA,RB

4	RT	RA	RB	1064
0	6	11	16	21 31

$RT_{0:31} \leftarrow (RA)_{0:15} \times_{ui} (RB)_{0:15}$
$RT_{32:63} \leftarrow (RA)_{32:47} \times_{ui} (RB)_{32:47}$
$ACC_{0:63} \leftarrow (RT)_{0:63}$

The corresponding even-numbered halfword unsigned-integer elements in RA and RB are multiplied. The two 32-bit products are placed into RT and into the accumulator.

Special Registers Altered:
ACC

Vector Multiply Halfwords, Even, Unsigned, Modulo, Integer and Accumulate into Words EVX-form

evmheumiaaw RT,RA,RB

4	RT	RA	RB	1288
0	6	11	16	21 31

$temp_{0:31} \leftarrow (RA)_{0:15} \times_{ui} (RB)_{0:15}$
$RT_{0:31} \leftarrow (ACC)_{0:31} + temp_{0:31}$
$temp_{0:31} \leftarrow (RA)_{32:47} \times_{ui} (RB)_{32:47}$
$RT_{32:63} \leftarrow (ACC)_{32:63} + temp_{0:31}$
$ACC_{0:63} \leftarrow (RT)_{0:63}$

For each word element in the accumulator, the corresponding even-numbered halfword unsigned-integer elements in RA and RB are multiplied. Each intermediate product is added to the contents of the corresponding accumulator words and the sums are placed into the corresponding RT and accumulator words.

Special Registers Altered:
ACC

Vector Multiply Halfwords, Even, Unsigned, Modulo, Integer and Accumulate Negative into Words EVX-form

evmheumianw RT,RA,RB

4	RT	RA	RB	1416
0	6	11	16	21 31

$temp_{0:31} \leftarrow (RA)_{0:15} \times_{ui} (RB)_{0:15}$
$RT_{0:31} \leftarrow (ACC)_{0:31} - temp_{0:31}$
$temp_{0:31} \leftarrow (RA)_{32:47} \times_{ui} (RB)_{32:47}$
$RT_{32:63} \leftarrow (ACC)_{32:63} - temp_{0:31}$
$ACC_{0:63} \leftarrow (RT)_{0:63}$

For each word element in the accumulator, the corresponding even-numbered halfword unsigned-integer elements in RA and RB are multiplied. Each intermediate product is subtracted from the contents of the corresponding accumulator words. The differences are placed into the corresponding RT and accumulator words.

Special Registers Altered:
ACC

Vector Multiply Halfwords, Even, Unsigned, Saturate, Integer and Accumulate into Words EVX-form

evmheusiaaw RT,RA,RB

4	RT	RA	RB	1280
0	6	11	16	21 31

```
temp0:31 ← (RA)0:15 ×ui (RB)0:15
temp0:63 ← EXTZ((ACC)0:31) + EXTZ(temp0:31)
ovh ← temp31
RT0:31 ← SATURATE(ovh, 0, 0xFFFF_FFFF, 0xFFFF_FFFF,
                  temp32:63)
temp0:31 ← (RA)32:47 ×ui (RB)32:47
temp0:63 ← EXTZ((ACC)32:63) + EXTZ(temp0:31)
ovl ← temp31
RT32:63 ← SATURATE(ovl, 0, 0xFFFF_FFFF,
                   0xFFFF_FFFF, temp32:63)
ACC0:63 ← (RT)0:63
SPEFSCROVH ← ovh
SPEFSCROV ← ovl
SPEFSCRSOVH ← SPEFSCRSOVH | ovh
SPEFSCRSOV ← SPEFSCRSOV | ovl
```

For each word element in the accumulator, corresponding even-numbered halfword unsigned-integer elements in RA and RB are multiplied producing a 32-bit product. Each 32-bit product is then added to the corresponding word in the accumulator saturating if overflow occurs, and the result is placed in RT and the accumulator.

Special Registers Altered:
ACC OV OVH SOV SOVH

Vector Multiply Halfwords, Even, Unsigned, Saturate, Integer and Accumulate Negative into Words EVX-form

evmheusianw RT,RA,RB

4	RT	RA	RB	1408
0	6	11	16	21 31

```
temp0:31 ← (RA)0:15 ×ui (RB)0:15
temp0:63 ← EXTZ((ACC)0:31) - EXTZ(temp0:31)
ovh ← temp31
RT0:31 ← SATURATE(ovh, 0, 0x0000_0000, 0x0000_0000,
                  temp32:63)
temp0:31 ← (RA)32:47 ×ui (RB)32:47
temp0:63 ← EXTZ((ACC)32:63) - EXTZ(temp0:31)
ovl ← temp31
RT32:63 ← SATURATE(ovl, 0, 0x0000_0000,
                   0x0000_0000, temp32:63)
ACC0:63 ← (RT)0:63
SPEFSCROVH ← ovh
SPEFSCROV ← ovl
SPEFSCRSOVH ← SPEFSCRSOVH | ovh
SPEFSCRSOV ← SPEFSCRSOV | ovl
```

For each word element in the accumulator, corresponding even-numbered halfword unsigned-integer elements in RA and RB are multiplied producing a 32-bit product. Each 32-bit product is then subtracted from the corresponding word in the accumulator saturating if overflow occurs, and the result is placed in RT and the accumulator.

Special Registers Altered:
ACC OV OVH SOV SOVH

Vector Multiply Halfwords, Odd, Guarded, Signed, Modulo, Fractional and Accumulate EVX-form

evmhogsmfaa RT,RA,RB

4	RT	RA	RB	1327
0	6	11	16	21 31

$temp_{0:63} \leftarrow (RA)_{48:63} \times_{gsf} (RB)_{48:63}$
$RT_{0:63} \leftarrow (ACC)_{0:63} + temp_{0:63}$
$ACC_{0:63} \leftarrow (RT)_{0:63}$

The corresponding low odd-numbered, halfword signed fractional elements in RA and RB are multiplied using guarded signed fractional multiplication producing a sign extended 64-bit fractional product with the decimal between bits 32 and 33. The product is added to the contents of the 64-bit accumulator and the result is placed in RT and the accumulator.

Special Registers Altered:
ACC

Note

If the two input operands are both -1.0, the intermediate product is represented as +1.0.

Vector Multiply Halfwords, Odd, Guarded, Signed, Modulo, Fractional and Accumulate Negative EVX-form

evmhogsmfan RT,RA,RB

4	RT	RA	RB	1455
0	6	11	16	21 31

$temp_{0:63} \leftarrow (RA)_{48:63} \times_{gsf} (RB)_{48:63}$
$RT_{0:63} \leftarrow (ACC)_{0:63} - temp_{0:63}$
$ACC_{0:63} \leftarrow (RT)_{0:63}$

The corresponding low odd-numbered, halfword signed fractional elements in RA and RB are multiplied using guarded signed fractional multiplication producing a sign extended 64-bit fractional product with the decimal between bits 32 and 33. The product is subtracted from the contents of the 64-bit accumulator and the result is placed in RT and the accumulator.

Special Registers Altered:
ACC

Note

If the two input operands are both -1.0, the intermediate product is represented as +1.0.

Vector Multiply Halfwords, Odd, Guarded, Signed, Modulo, Integer and Accumulate EVX-form

evmhogsmiaa RT,RA,RB

4	RT	RA	RB	1325
0	6	11	16	21 31

$temp_{0:31} \leftarrow (RA)_{48:63} \times_{si} (RB)_{48:63}$
$temp_{0:63} \leftarrow EXTS(temp_{0:31})$
$RT_{0:63} \leftarrow (ACC)_{0:63} + temp_{0:63}$
$ACC_{0:63} \leftarrow (RT)_{0:63}$

The corresponding low odd-numbered halfword signed-integer elements in RA and RB are multiplied. The intermediate product is sign-extended to 64 bits then added to the contents of the 64-bit accumulator, and the result is placed in RT and into the accumulator.

Special Registers Altered:
ACC

Vector Multiply Halfwords, Odd, Guarded, Signed, Modulo, Integer and Accumulate Negative EVX-form

evmhogsmian RT,RA,RB

4	RT	RA	RB	1453
0	6	11	16	21 31

$temp_{0:31} \leftarrow (RA)_{48:63} \times_{si} (RB)_{48:63}$
$temp_{0:63} \leftarrow EXTS(temp_{0:31})$
$RT_{0:63} \leftarrow (ACC)_{0:63} - temp_{0:63}$
$ACC_{0:63} \leftarrow (RT)_{0:63}$

The corresponding low odd-numbered halfword signed-integer elements in RA and RB are multiplied. The intermediate product is sign-extended to 64 bits then subtracted from the contents of the 64-bit accumulator, and the result is placed in RT and into the accumulator.

Special Registers Altered:
ACC

Vector Multiply Halfwords, Odd, Guarded, Unsigned, Modulo, Integer and Accumulate EVX-form

evmhogumiaa RT,RA,RB

4	RT	RA	RB	1324
0	6	11	16	21 31

$temp_{0:31} \leftarrow (RA)_{48:63} \times_{ui} (RB)_{48:63}$
$temp_{0:63} \leftarrow EXTZ(temp_{0:31})$
$RT_{0:63} \leftarrow (ACC)_{0:63} + temp_{0:63}$
$ACC_{0:63} \leftarrow (RT)_{0:63}$

The corresponding low odd-numbered halfword unsigned-integer elements in RA and RB are multiplied. The intermediate product is zero-extended to 64 bits then added to the contents of the 64-bit accumulator, and the result is placed in RT and into the accumulator.

Special Registers Altered:
ACC

Vector Multiply Halfwords, Odd, Guarded, Unsigned, Modulo, Integer and Accumulate Negative EVX-form

evmhogumian RT,RA,RB

4	RT	RA	RB	1452
0	6	11	16	21 31

$temp_{0:31} \leftarrow (RA)_{48:63} \times_{ui} (RB)_{48:63}$
$temp_{0:63} \leftarrow EXTZ(temp_{0:31})$
$RT_{0:63} \leftarrow (ACC)_{0:63} - temp_{0:63}$
$ACC_{0:63} \leftarrow (RT)_{0:63}$

The corresponding low odd-numbered halfword unsigned-integer elements in RA and RB are multiplied. The intermediate product is zero-extended to 64 bits then subtracted from the contents of the 64-bit accumulator, and the result is placed in RT and into the accumulator.

Special Registers Altered:
ACC

Vector Multiply Halfwords, Odd, Signed, Modulo, Fractional EVX-form

evmhosmf RT,RA,RB

4	RT	RA	RB	1039
0	6	11	16	21 31

$RT_{0:31} \leftarrow (RA)_{16:31} \times_{sf} (RB)_{16:31}$
$RT_{32:63} \leftarrow (RA)_{48:63} \times_{sf} (RB)_{48:63}$

The corresponding odd-numbered, halfword signed fractional elements in RA and RB are multiplied. Each product is placed into the corresponding words of RT.

Special Registers Altered:
None

Vector Multiply Halfwords, Odd, Signed, Modulo, Fractional to Accumulator EVX-form

evmhosmfa RT,RA,RB

4	RT	RA	RB	1071
0	6	11	16	21 31

$RT_{0:31} \leftarrow (RA)_{16:31} \times_{sf} (RB)_{16:31}$
$RT_{32:63} \leftarrow (RA)_{48:63} \times_{sf} (RB)_{48:63}$
$ACC_{0:63} \leftarrow (RT)_{0:63}$

The corresponding odd-numbered, halfword signed fractional elements in RA and RB are multiplied. Each product is placed into the corresponding words of RT. and into the accumulator.

Special Registers Altered:
ACC

Vector Multiply Halfwords, Odd, Signed, Modulo, Fractional and Accumulate into Words EVX-form

evmhosmfaaw RT,RA,RB

4	RT	RA	RB	1295
0	6	11	16	21 31

$temp_{0:31} \leftarrow (RA)_{16:31} \times_{sf} (RB)_{16:31}$
$RT_{0:31} \leftarrow (ACC)_{0:31} + temp_{0:31}$
$temp_{0:31} \leftarrow (RA)_{48:63} \times_{sf} (RB)_{48:63}$
$RT_{32:63} \leftarrow (ACC)_{32:63} + temp_{0:31}$
$ACC_{0:63} \leftarrow (RT)_{0:63}$

For each word element in the accumulator, the corresponding odd-numbered halfword signed fractional elements in RA and RB are multiplied. The 32 bits of each intermediate product are added to the contents of the corresponding accumulator word and the results are placed into the corresponding RT words and into the accumulator.

Special Registers Altered:
ACC

Vector Multiply Halfwords, Odd, Signed, Modulo, Fractional and Accumulate Negative into Words EVX-form

evmhosmfanw RT,RA,RB

4	RT	RA	RB	1423
0	6	11	16	21 31

$temp_{0:31} \leftarrow (RA)_{16:31} \times_{sf} (RB)_{16:31}$
$RT_{0:31} \leftarrow (ACC)_{0:31} - temp_{0:31}$
$temp_{0:31} \leftarrow (RA)_{48:63} \times_{sf} (RB)_{48:63}$
$RT_{32:63} \leftarrow (ACC)_{32:63} - temp_{0:31}$
$ACC_{0:63} \leftarrow (RT)_{0:63}$

For each word element in the accumulator, the corresponding odd-numbered halfword signed fractional elements in RA and RB are multiplied. The 32 bits of each intermediate product are subtracted from the contents of the corresponding accumulator word and the results are placed into the corresponding RT words and into the accumulator.

Special Registers Altered:
ACC

Vector Multiply Halfwords, Odd, Signed, Modulo, Integer EVX-form

evmhosmi RT,RA,RB

4	RT	RA	RB	1037
0	6	11	16	21 31

$RT_{0:31} \leftarrow (RA)_{16:31} \times_{si} (RB)_{16:31}$
$RT_{32:63} \leftarrow (RA)_{48:63} \times_{si} (RB)_{48:63}$

The corresponding odd-numbered halfword signed-integer elements in RA and RB are multiplied. The two 32-bit products are placed into the corresponding words of RT.

Special Registers Altered:
None

Vector Multiply Halfwords, Odd, Signed, Modulo, Integer to Accumulator EVX-form

evmhosmia RT,RA,RB

4	RT	RA	RB	1069
0	6	11	16	21 31

$RT_{0:31} \leftarrow (RA)_{16:31} \times_{si} (RB)_{16:31}$
$RT_{32:63} \leftarrow (RA)_{48:63} \times_{si} (RB)_{48:63}$
$ACC_{0:63} \leftarrow (RT)_{0:63}$

The corresponding odd-numbered halfword signed-integer elements in RA and RB are multiplied. The two 32-bit products are placed into the corresponding words of RT and into the accumulator.

Special Registers Altered:
ACC

Vector Multiply Halfwords, Odd, Signed, Modulo, Integer and Accumulate into Words EVX-form

evmhosmiaaw RT,RA,RB

4	RT	RA	RB	1293
0	6	11	16	21 31

$temp_{0:31} \leftarrow (RA)_{16:31} \times_{si} (RB)_{16:31}$
$RT_{0:31} \leftarrow (ACC)_{0:31} + temp_{0:31}$
$temp_{0:31} \leftarrow (RA)_{48:63} \times_{si} (RB)_{48:63}$
$RT_{32:63} \leftarrow (ACC)_{32:63} + temp_{0:31}$
$ACC_{0:63} \leftarrow (RT)_{0:63}$

For each word element in the accumulator, the corresponding odd-numbered halfword signed-integer elements in RA and RB are multiplied. Each intermediate 32-bit product is added to the contents of the corresponding accumulator word and the results are placed into the corresponding RT words and into the accumulator.

Special Registers Altered:
ACC

Vector Multiply Halfwords, Odd, Signed, Modulo, Integer and Accumulate Negative into Words EVX-form

evmhosmianw RT,RA,RB

4	RT	RA	RB	1421
0	6	11	16	21 31

$temp_{0:31} \leftarrow (RA)_{16:31} \times_{si} (RB)_{16:31}$
$RT_{0:31} \leftarrow (ACC)_{0:31} - temp_{0:31}$
$temp_{0:31} \leftarrow (RA)_{48:63} \times_{si} (RB)_{48:63}$
$RT_{32:63} \leftarrow (ACC)_{32:63} - temp_{0:31}$
$ACC_{0:63} \leftarrow (RT)_{0:63}$

For each word element in the accumulator, the corresponding odd-numbered halfword signed-integer elements in RA and RB are multiplied. Each intermediate 32-bit product is subtracted from the contents of the corresponding accumulator word and the results are placed into the corresponding RT words and into the accumulator.

Special Registers Altered:
ACC

Vector Multiply Halfwords, Odd, Signed, Saturate, Fractional EVX-form

evmhossf RT,RA,RB

4	RT	RA	RB	1031
0	6	11	16	21 31

```
temp0:31 ← (RA)16:31 ×sf (RB)16:31
if ((RA)16:31 = 0x8000) & ((RB)16:31 = 0x8000) then
   RT0:31 ← 0x7FFF_FFFF
   movh ← 1
else
   RT0:31 ← temp0:31
   movh ← 0
temp0:31 ← (RA)48:63 ×sf (RB)48:63
if ((RA)48:63 = 0x8000) & ((RB)48:63 = 0x8000) then
   RT32:63 ← 0x7FFF_FFFF
   movl ← 1
else
   RT32:63 ← temp0:31
   movl ← 0
SPEFSCROVH ← movh
SPEFSCROV ← movl
SPEFSCRSOVH ← SPEFSCRSOVH | movh
SPEFSCRSOV ← SPEFSCRSOV | movl
```

The corresponding odd-numbered halfword signed fractional elements in RA and RB are multiplied. The 32 bits of each product are placed into the corresponding words of RT. If both inputs are -1.0, the result saturates to the largest positive signed fraction.

Special Registers Altered:
OV OVH SOV SOVH

Vector Multiply Halfwords, Odd, Signed, Saturate, Fractional to Accumulator EVX-form

evmhossfa RT,RA,RB

4	RT	RA	RB	1063
0	6	11	16	21 31

```
temp0:31 ← (RA)16:31 ×sf (RB)16:31
if ((RA)16:31 = 0x8000) & ((RB)16:31 = 0x8000) then
   RT0:31 ← 0x7FFF_FFFF
   movh ← 1
else
   RT0:31 ← temp0:31
   movh ← 0
temp0:31 ← (RA)48:63 ×sf (RB)48:63
if ((RA)48:63 = 0x8000) & ((RB)48:63 = 0x8000) then
   RT32:63 ← 0x7FFF_FFFF
   movl ← 1
else
   RT32:63 ← temp0:31
   movl ← 0
ACC0:63 ← (RT)0:63
SPEFSCROVH ← movh
SPEFSCROV ← movl
SPEFSCRSOVH ← SPEFSCRSOVH | movh
SPEFSCRSOV ← SPEFSCRSOV | movl
```

The corresponding odd-numbered halfword signed fractional elements in RA and RB are multiplied. The 32 bits of each product are placed into the corresponding words of RT and into the accumulator. If both inputs are -1.0, the result saturates to the largest positive signed fraction.

Special Registers Altered:
ACC OV OVH SOV SOVH

Vector Multiply Halfwords, Odd, Signed, Saturate, Fractional and Accumulate into Words EVX-form

evmhossfaaw RT,RA,RB

4	RT	RA	RB	1287
0	6	11	16	21 31

```
temp0:31 ← (RA)16:31 ×sf (RB)16:31
if ((RA)16:31 = 0x8000) & ((RB)16:31 = 0x8000) then
   temp0:31 ← 0x7FFF_FFFF
   movh ← 1
else
   movh ← 0
temp0:63 ← EXTS((ACC)0:31) + EXTS(temp0:31)
ovh ← (temp31 ⊕ temp32)
RT0:31 ← SATURATE(ovh, temp31, 0x8000_0000,
                 0x7FFF_FFFF, temp32:63)
temp0:31 ← (RA)48:63 ×sf (RB)48:63
if ((RA)48:63 = 0x8000) & ((RB)48:63 = 0x8000) then
   temp0:31 ← 0x7FFF_FFFF
   movl ← 1
else
   movl ← 0
temp0:63 ← EXTS((ACC)32:63) + EXTS(temp0:31)
ovl ← (temp31 ⊕ temp32)
RT32:63 ← SATURATE(ovl, temp31, 0x8000_0000,
                  0x7FFF_FFFF, temp32:63)
ACC0:63 ← (RT)0:63
SPEFSCROVH ← ovh | movh
SPEFSCROV ← ovl| movl
SPEFSCRSOVH ← SPEFSCRSOVH | ovh | movh
SPEFSCRSOV ← SPEFSCRSOV | ovl| movl
```

The corresponding odd-numbered halfword signed fractional elements in RA and RB are multiplied producing a 32-bit product. If both inputs are -1.0, the result saturates to 0x7FFF_FFFF. Each 32-bit product is then added to the corresponding word in the accumulator saturating if overflow occurs, and the result is placed in RT and the accumulator.

Special Registers Altered:

ACC OV OVH SOV SOVH

Vector Multiply Halfwords, Odd, Signed, Saturate, Fractional and Accumulate Negative into Words EVX-form

evmhossfanw RT,RA,RB

4	RT	RA	RB	1415
0	6	11	16	21 31

```
temp0:31 ← (RA)16:31 ×sf (RB)16:31
if ((RA)16:31 = 0x8000) & ((RB)16:31 = 0x8000) then
   temp0:31 ← 0x7FFF_FFFF
   movh ← 1
else
   movh ← 0
temp0:63 ← EXTS((ACC)0:31) - EXTS(temp0:31)
ovh ← (temp31 ⊕ temp32)
RT0:31 ← SATURATE(ovh, temp31, 0x8000_0000,
                 0x7FFF_FFFF, temp32:63)
temp0:31 ← (RA)48:63 ×sf (RB)48:63
if ((RA)48:63 = 0x8000) & ((RB)48:63 = 0x8000) then
   temp0:31 ← 0x7FFF_FFFF
   movl ← 1
else
   movl ← 0
temp0:63 ← EXTS((ACC)32:63) - EXTS(temp0:31)
ovl ← (temp31 ⊕ temp32)
RT32:63 ← SATURATE(ovl, temp31, 0x8000_0000,
                  0x7FFF_FFFF, temp32:63)
ACC0:63 ← (RT)0:63
SPEFSCROVH ← ovh | movh
SPEFSCROV ← ovl| movl
SPEFSCRSOVH ← SPEFSCRSOVH | ovh | movh
SPEFSCRSOV ← SPEFSCRSOV | ovl| movl
```

The corresponding odd-numbered halfword signed fractional elements in RA and RB are multiplied producing a 32-bit product. If both inputs are -1.0, the result saturates to 0x7FFF_FFFF. Each 32-bit product is then subtracted from the corresponding word in the accumulator saturating if overflow occurs, and the result is placed in RT and the accumulator.

Special Registers Altered:

ACC OV OVH SOV SOVH

Vector Multiply Halfwords, Odd, Signed, Saturate, Integer and Accumulate into Words EVX-form

evmhossiaaw RT,RA,RB

4	RT	RA	RB	1285
0	6	11	16	21 31

```
temp0:31 ← (RA)16:31 ×si (RB)16:31
temp0:63 ← EXTS((ACC)0:31) + EXTS(temp0:31)
ovh ← (temp31 ⊕ temp32)
RT0:31 ← SATURATE(ovh, temp31, 0x8000_0000,
                  0x7FFF_FFFF, temp32:63)
temp0:31 ← (RA)48:63 ×si (RB)48:63
temp0:63 ← EXTS((ACC)32:63) + EXTS(temp0:31)
ovl ← (temp31 ⊕ temp32)
RT32:63 ← SATURATE(ovl, temp31, 0x8000_0000,
                   0x7FFF_FFFF, temp32:63)
ACC0:63 ← (RT)0:63
SPEFSCROVH ← ovh
SPEFSCROV ← ovl
SPEFSCRSOVH ← SPEFSCRSOVH | ovh
SPEFSCRSOV ← SPEFSCRSOV | ovl
```

The corresponding odd-numbered halfword signed-integer elements in RA and RB are multiplied producing a 32-bit product. Each 32-bit product is then added to the corresponding word in the accumulator saturating if overflow occurs, and the result is placed in RT and the accumulator.

Special Registers Altered:
ACC OV OVH SOV SOVH

Vector Multiply Halfwords, Odd, Unsigned, Modulo, Integer EVX-form

evmhoumi RT,RA,RB

4	RT	RA	RB	1036
0	6	11	16	21 31

```
RT0:31 ← (RA)16:31 ×ui (RB)16:31
RT32:63 ← (RA)48:63 ×ui (RB)48:63
```

The corresponding odd-numbered halfword unsigned-integer elements in RA and RB are multiplied. The two 32-bit products are placed into the corresponding words of RT.

Special Registers Altered:
None

Vector Multiply Halfwords, Odd, Signed, Saturate, Integer and Accumulate Negative into Words EVX-form

evmhossianw RT,RA,RB

4	RT	RA	RB	1413
0	6	11	16	21 31

```
temp0:31 ← (RA)16:31 ×si (RB)16:31
temp0:63 ← EXTS((ACC)0:31) - EXTS(temp0:31)
ovh ← (temp31 ⊕ temp32)
RT0:31 ← SATURATE(ovh, temp31, 0x8000_0000,
                  0x7FFF_FFFF, temp32:63)
temp0:31 ← (RA)48:63 ×si (RB)48:63
temp0:63 ← EXTS((ACC)32:63) - EXTS(temp0:31)
ovl ← (temp31 ⊕ temp32)
RT32:63 ← SATURATE(ovl, temp31, 0x8000_0000,
                   0x7FFF_FFFF, temp32:63)
ACC0:63 ← (RT)0:63
SPEFSCROVH ← ovh
SPEFSCROV ← ovl
SPEFSCRSOVH ← SPEFSCRSOVH | ovh
SPEFSCRSOV ← SPEFSCRSOV | ovl
```

The corresponding odd-numbered halfword signed-integer elements in RA and RB are multiplied producing a 32-bit product. Each 32-bit product is then subtracted from the corresponding word in the accumulator saturating if overflow occurs, and the result is placed in RT and the accumulator.

Special Registers Altered:
ACC OV OVH SOV SOVH

Vector Multiply Halfwords, Odd, Unsigned, Modulo, Integer to Accumulator EVX-form

evmhoumia RT,RA,RB

4	RT	RA	RB	1068
0	6	11	16	21 31

```
RT0:31 ← (RA)16:31 ×ui (RB)16:31
RT32:63 ← (RA)48:63 ×ui (RB)48:63
ACC0:63 ← (RT)0:63
```

The corresponding odd-numbered halfword unsigned-integer elements in RA and RB are multiplied. The two 32-bit products are placed into RT and into the accumulator.

Special Registers Altered:
ACC

Vector Multiply Halfwords, Odd, Unsigned, Modulo, Integer and Accumulate into Words EVX-form

evmhoumiaaw RT,RA,RB

4 (0)	RT (6)	RA (11)	RB (16)	1292 (21–31)

```
temp0:31 ← (RA)16:31 ×ui (RB)16:31
RT0:31 ← (ACC)0:31 + temp0:31
temp0:31 ← (RA)48:63 ×ui (RB)48:63
RT32:63 ← (ACC)32:63 + temp0:31
ACC0:63 ← (RT)0:63
```

For each word element in the accumulator, the corresponding odd-numbered halfword unsigned-integer elements in RA and RB are multiplied. Each intermediate product is added to the contents of the corresponding accumulator word. The sums are placed into the corresponding RT and accumulator words.

Special Registers Altered:
ACC

Vector Multiply Halfwords, Odd, Unsigned, Modulo, Integer and Accumulate Negative into Words EVX-form

evmhoumianw RT,RA,RB

4 (0)	RT (6)	RA (11)	RB (16)	1420 (21–31)

```
temp0:31 ← (RA)16:31 ×ui (RB)16:31
RT0:31 ← (ACC)0:31 - temp0:31
temp0:31 ← (RA)48:63 ×ui (RB)48:63
RT32:63 ← (ACC)32:63 - temp0:31
ACC0:63 ← (RT)0:63
```

For each word element in the accumulator, the corresponding odd-numbered halfword unsigned-integer elements in RA and RB are multiplied. Each intermediate product is subtracted from the contents of the corresponding accumulator word. The results are placed into the corresponding RT and accumulator words.

Special Registers Altered:
ACC

Vector Multiply Halfwords, Odd, Unsigned, Saturate, Integer and Accumulate into Words EVX-form

evmhousiaaw RT,RA,RB

4 (0)	RT (6)	RA (11)	RB (16)	1284 (21–31)

```
temp0:31 ← (RA)16:31 ×ui (RB)16:31
temp0:63 ← EXTZ((ACC)0:31) + EXTZ(temp0:31)
ovh ← temp31
RT0:31 ← SATURATE(ovh, 0, 0xFFFF_FFFF, 0xFFFF_FFFF,
                  temp32:63)
temp0:31 ← (RA)48:63 ×ui (RB)48:63
temp0:63 ← EXTZ((ACC)32:63) + EXTZ(temp0:31)
ovl ← temp31
RT32:63 ← SATURATE(ovl, 0, 0xFFFF_FFFF,
                   0xFFFF_FFFF, temp32:63)
ACC0:63 ← (RT)0:63
SPEFSCROVH ← ovh
SPEFSCROV ← ovl
SPEFSCRSOVH ← SPEFSCRSOVH | ovh
SPEFSCRSOV ← SPEFSCRSOV | ovl
```

For each word element in the accumulator, corresponding odd-numbered halfword unsigned-integer elements in RA and RB are multiplied producing a 32-bit product. Each 32-bit product is then added to the corresponding word in the accumulator saturating if overflow occurs, and the result is placed in RT and the accumulator.

Special Registers Altered:
ACC OV OVH SOV SOVH

Vector Multiply Halfwords, Odd, Unsigned, Saturate, Integer and Accumulate Negative into Words EVX-form

evmhousianw RT,RA,RB

4 (0)	RT (6)	RA (11)	RB (16)	1412 (21–31)

```
temp0:31 ← (RA)16:31 ×ui (RB)16:31
temp0:63 ← EXTZ((ACC)0:31) - EXTZ(temp0:31)
ovh ← temp31
RT0:31 ← SATURATE(ovh, 0, 0x0000_0000, 0x0000_0000,
                  temp32:63)
temp0:31 ← (RA)48:63 ×ui (RB)48:63
temp0:63 ← EXTZ((ACC)32:63) - EXTZ(temp0:31)
ovl ← temp31
RT32:63 ← SATURATE(ovl, 0, 0x0000_0000,0x0000_0000,
                  temp32:63)
ACC0:63 ← (RT)0:63
SPEFSCROVH ← ovh
SPEFSCROV ← ovl
SPEFSCRSOVH ← SPEFSCRSOVH | ovh
SPEFSCRSOV ← SPEFSCRSOV | ovl
```

For each word element in the accumulator, corresponding odd-numbered halfword unsigned-integer elements in RA and RB are multiplied producing a 32-bit product. Each 32-bit product is then subtracted from the corresponding word in the accumulator saturating if overflow occurs, and the result is placed in RT and the accumulator.

Special Registers Altered:
ACC OV OVH SOV SOVH

Initialize Accumulator *EVX-form*

evmra RT,RA

4	RT	RA	///	1220
0	6	11	16	21 31

$ACC_{0:63} \leftarrow (RA)_{0:63}$
$RT_{0:63} \leftarrow (RA)_{0:63}$

The contents of RA are placed into the accumulator and RT. This is the method for initializing the accumulator.

Special Registers Altered:
ACC

Vector Multiply Word High Signed, Modulo, Fractional *EVX-form*

evmwhsmf RT,RA,RB

4	RT	RA	RB	1103
0	6	11	16	21 31

$temp_{0:63} \leftarrow (RA)_{0:31} \times_{sf} (RB)_{0:31}$
$RT_{0:31} \leftarrow temp_{0:31}$
$temp_{0:63} \leftarrow (RA)_{32:63} \times_{sf} (RB)_{32:63}$
$RT_{32:63} \leftarrow temp_{0:31}$

The corresponding word signed fractional elements in RA and RB are multiplied and bits 0:31 of the two products are placed into the two corresponding words of RT.

Special Registers Altered:
None

Vector Multiply Word High Signed, Modulo, Fractional to Accumulator *EVX-form*

evmwhsmfa RT,RA,RB

4	RT	RA	RB	1135
0	6	11	16	21 31

$temp_{0:63} \leftarrow (RA)_{0:31} \times_{sf} (RB)_{0:31}$
$RT_{0:31} \leftarrow temp_{0:31}$
$temp_{0:63} \leftarrow (RA)_{32:63} \times_{sf} (RB)_{32:63}$
$RT_{32:63} \leftarrow temp_{0:31}$
$ACC_{0:63} \leftarrow (RT)_{0:63}$

The corresponding word signed fractional elements in RA and RB are multiplied and bits 0:31 of the two products are placed into the two corresponding words of RT and into the accumulator.

Special Registers Altered:
ACC

Vector Multiply Word High Signed, Modulo, Integer *EVX-form*

evmwhsmi RT,RA,RB

4	RT	RA	RB	1101
0	6	11	16	21 31

$temp_{0:63} \leftarrow (RA)_{0:31} \times_{si} (RB)_{0:31}$
$RT_{0:31} \leftarrow temp_{0:31}$
$temp_{0:63} \leftarrow (RA)_{32:63} \times_{si} (RB)_{32:63}$
$RT_{32:63} \leftarrow temp_{0:31}$

The corresponding word signed-integer elements in RA and RB are multiplied. Bits 0:31 of the two 64-bit products are placed into the two corresponding words of RT.

Special Registers Altered:
None

Vector Multiply Word High Signed, Modulo, Integer to Accumulator *EVX-form*

evmwhsmia RT,RA,RB

4	RT	RA	RB	1133
0	6	11	16	21 31

$temp_{0:63} \leftarrow (RA)_{0:31} \times_{si} (RB)_{0:31}$
$RT_{0:31} \leftarrow temp_{0:31}$
$temp_{0:63} \leftarrow (RA)_{32:63} \times_{si} (RB)_{32:63}$
$RT_{32:63} \leftarrow temp_{0:31}$
$ACC_{0:63} \leftarrow (RT)_{0:63}$

The corresponding word signed-integer elements in RA and RB are multiplied. Bits 0:31 of the two 64-bit products are placed into the two corresponding words of RT and into the accumulator.

Special Registers Altered:
ACC

Vector Multiply Word High Signed, Saturate, Fractional EVX-form

evmwhssf RT,RA,RB

4	RT	RA	RB	1095
0	6	11	16	21 31

```
temp_{0:63} ← (RA)_{0:31} ×_{sf} (RB)_{0:31}
if ((RA)_{0:31} = 0x8000_0000)&((RB)_{0:31} = 0x8000_0000)
then
   RT_{0:31} ← 0x7FFF_FFFF
   movh ← 1
else
   RT_{0:31} ← temp_{0:31}
   movh ← 0
temp_{0:63} ← (RA)_{32:63} ×_{sf} (RB)_{32:63}
if ((RA)_{32:63} = 0x8000_0000 &(RB)_{32:63} = 0x8000_0000)
then
   RT_{32:63} ← 0x7FFF_FFFF
   movl ← 1
else
   RT_{32:63} ← temp_{0:31}
   movl ← 0
SPEFSCR_{OVH} ← movh
SPEFSCR_{OV} ← movl
SPEFSCR_{SOVH} ← SPEFSCR_{SOVH} | movh
SPEFSCR_{SOV} ← SPEFSCR_{SOV} | movl
```

The corresponding word signed fractional elements in RA and RB are multiplied. Bits 0:31 of each product are placed into the corresponding words of RT. If both inputs are -1.0, the result saturates to the largest positive signed fraction.

Special Registers Altered:
OV OVH SOV SOVH

Vector Multiply Word High Signed, Saturate, Fractional to Accumulator EVX-form

evmwhssfa RT,RA,RB

4	RT	RA	RB	1127
0	6	11	16	21 31

```
temp_{0:63} ← (RA)_{0:31} ×_{sf} (RB)_{0:31}
if ((RA)_{0:31} = 0x8000_0000) &((RB)_{0:31} =0x8000_0000)
then
   RT_{0:31} ← 0x7FFF_FFFF
   movh ← 1
else
   RT_{0:31} ← temp_{0:31}
   movh ← 0
temp_{0:63} ← (RA)_{32:63} ×_{sf} (RB)_{32:63}
if ((RA)_{32:63}=0x8000_0000)&((RB)_{32:63}=0x8000_0000)
then
   RT_{32:63} ← 0x7FFF_FFFF
   movl ← 1
else
   RT_{32:63} ← temp_{0:31}
   movl ← 0
ACC_{0:63} ← (RT)_{0:63}
SPEFSCR_{OVH} ← movh
SPEFSCR_{OV} ← movl
SPEFSCR_{SOVH} ← SPEFSCR_{SOVH} | movh
SPEFSCR_{SOV} ← SPEFSCR_{SOV} | movl
```

The corresponding word signed fractional elements in RA and RB are multiplied. Bits 0:31 of each product are placed into the corresponding words of RT and into the accumulator. If both inputs are -1.0, the result saturates to the largest positive signed fraction.

Special Registers Altered:
ACC OV OVH SOV SOVH

Vector Multiply Word High Unsigned, Modulo, Integer EVX-form

evmwhumi RT,RA,RB

4	RT	RA	RB	1100
0	6	11	16	21 31

```
temp_{0:63} ← (RA)_{0:31} ×_{ui} (RB)_{0:31}
RT_{0:31} ← temp_{0:31}
temp_{0:63} ← (RA)_{32:63} ×_{ui} (RB)_{32:63}
RT_{32:63} ← temp_{0:31}
```

The corresponding word unsigned-integer elements in RA and RB are multiplied. Bits 0:31 of the two products are placed into the two corresponding words of RT.

Special Registers Altered:
None

Vector Multiply Word High Unsigned, Modulo, Integer to Accumulator EVX-form

evmwhumia RT,RA,RB

4	RT	RA	RB	1132
0	6	11	16	21 31

```
temp_{0:63} ← (RA)_{0:31} ×_{ui} (RB)_{0:31}
RT_{0:31} ← temp_{0:31}
temp_{0:63} ← (RA)_{32:63} ×_{ui} (RB)_{32:63}
RT_{32:63} ← temp_{0:31}
ACC_{0:63} ← (RT)_{0:63}
```

The corresponding word unsigned-integer elements in RA and RB are multiplied. Bits 0:31 of the two products are placed into the two corresponding words of RT and into the accumulator.

Special Registers Altered:
ACC

Vector Multiply Word Low Signed, Modulo, Integer and Accumulate into Words EVX-form

evmwlsmiaaw RT,RA,RB

4	RT	RA	RB	1353
0	6	11	16	21 31

$temp_{0:63} \leftarrow (RA)_{0:31} \times_{si} (RB)_{0:31}$
$RT_{0:31} \leftarrow (ACC)_{0:31} + temp_{32:63}$
$temp_{0:63} \leftarrow (RA)_{32:63} \times_{si} (RB)_{32:63}$
$RT_{32:63} \leftarrow (ACC)_{32:63} + temp_{32:63}$
$ACC_{0:63} \leftarrow (RT)_{0:63}$

For each word element in the accumulator, the corresponding word signed-integer elements in RA and RB are multiplied. The least significant 32 bits of each intermediate product are added to the contents of the corresponding accumulator words, and the result is placed in RT and the accumulator.

Special Registers Altered:
ACC

Vector Multiply Word Low Signed, Saturate, Integer and Accumulate into Words EVX-form

evmwlssiaaw RT,RA,RB

4	RT	RA	RB	1345
0	6	11	16	21 31

$temp_{0:63} \leftarrow (RA)_{0:31} \times_{si} (RB)_{0:31}$
$temp_{0:63} \leftarrow EXTS((ACC)_{0:31}) + EXTS(temp_{32:63})$
$ovh \leftarrow (temp_{31} \oplus temp_{32})$
$RT_{0:31} \leftarrow SATURATE(ovh, temp_{31}, 0x8000\_0000, 0x7FFF\_FFFF, temp_{32:63})$
$temp_{0:63} \leftarrow (RA)_{32:63} \times_{si} (RB)_{32:63}$
$temp_{0:63} \leftarrow EXTS((ACC)_{32:63}) + EXTS(temp_{32:63})$
$ovl \leftarrow (temp_{31} \oplus temp_{32})$
$RT_{32:63} \leftarrow SATURATE(ovl, temp_{31}, 0x8000\_0000, 0x7FFF\_FFFF, temp_{32:63})$
$ACC_{0:63} \leftarrow (RT)_{0:63}$
$SPEFSCR_{OVH} \leftarrow ovh$
$SPEFSCR_{OV} \leftarrow ovl$
$SPEFSCR_{SOVH} \leftarrow SPEFSCR_{SOVH} \mid ovh$
$SPEFSCR_{SOV} \leftarrow SPEFSCR_{SOV} \mid ovl$

The corresponding word signed-integer elements in RA and RB are multiplied producing a 64-bit product. The least significant 32 bits of each product are then added to the corresponding word in the accumulator saturating if overflow occurs, and the result is placed in RT and the accumulator.

Special Registers Altered:
ACC OV OVH SOV SOVH

Vector Multiply Word Low Signed, Modulo, Integer and Accumulate Negative in Words EVX-form

evmwlsmianw RT,RA,RB

4	RT	RA	RB	1481
0	6	11	16	21 31

$temp_{0:63} \leftarrow (RA)_{0:31} \times_{si} (RB)_{0:31}$
$RT_{0:31} \leftarrow (ACC)_{0:31} - temp_{32:63}$
$temp_{0:63} \leftarrow (RA)_{32:63} \times_{si} (RB)_{32:63}$
$RT_{32:63} \leftarrow (ACC)_{32:63} - temp_{32:63}$
$ACC_{0:63} \leftarrow (RT)_{0:63}$

For each word element in the accumulator, the corresponding word elements in RA and RB are multiplied. The least significant 32 bits of each intermediate product are subtracted from the contents of the corresponding accumulator words and the result is placed in RT and the accumulator.

Special Registers Altered:
ACC

Vector Multiply Word Low Signed, Saturate, Integer and Accumulate Negative in Words EVX-form

evmwlssianw RT,RA,RB

4	RT	RA	RB	1473
0	6	11	16	21 31

$temp_{0:63} \leftarrow (RA)_{0:31} \times_{si} (RB)_{0:31}$
$temp_{0:63} \leftarrow EXTS((ACC)_{0:31}) - EXTS(temp_{32:63})$
$ovh \leftarrow (temp_{31} \oplus temp_{32})$
$RT_{0:31} \leftarrow SATURATE(ovh, temp_{31}, 0x8000\_0000, 0x7FFF\_FFFF, temp_{32:63})$
$temp_{0:63} \leftarrow (RA)_{32:63} \times_{si} (RB)_{32:63}$
$temp_{0:63} \leftarrow EXTS((ACC)_{32:63}) - EXTS(temp_{32:63})$
$ovl \leftarrow (temp_{31} \oplus temp_{32})$
$RT_{32:63} \leftarrow SATURATE(ovl, temp_{31}, 0x8000\_0000, 0x7FFF\_FFFF, temp_{32:63})$
$ACC_{0:63} \leftarrow (RT)_{0:63}$
$SPEFSCR_{OVH} \leftarrow ovh$
$SPEFSCR_{OV} \leftarrow ovl$
$SPEFSCR_{SOVH} \leftarrow SPEFSCR_{SOVH} \mid ovh$
$SPEFSCR_{SOV} \leftarrow SPEFSCR_{SOV} \mid ovl$

The corresponding word signed-integer elements in RA and RB are multiplied producing a 64-bit product. The least significant 32 bits of each product are then subtracted from the corresponding word in the accumulator saturating if overflow occurs, and the result is placed in RT and the accumulator.

Special Registers Altered:
ACC OV OVH SOV SOVH

Vector Multiply Word Low Unsigned, Modulo, Integer *EVX-form*

evmwlumi RT,RA,RB

4	RT	RA	RB	1096
0	6	11	16	21 31

$temp_{0:63} \leftarrow (RA)_{0:31} \times_{ui} (RB)_{0:31}$
$RT_{0:31} \leftarrow temp_{32:63}$
$temp_{0:63} \leftarrow (RA)_{32:63} \times_{ui} (RB)_{32:63}$
$RT_{32:63} \leftarrow temp_{32:63}$

The corresponding word unsigned-integer elements in RA and RB are multiplied. The least significant 32 bits of each product are placed into the two corresponding words of RT.

Special Registers Altered:
None

Programming Note

The least significant 32 bits of the product are independent of whether the word elements in RA and RB are treated as signed or unsigned 32-bit integers.

Note that ***evmwlumi*** can be used for signed or unsigned integers.

Vector Multiply Word Low Unsigned, Modulo, Integer to Accumulator *EVX-form*

evmwlumia RT,RA,RB

4	RT	RA	RB	1128
0	6	11	16	21 31

$temp_{0:63} \leftarrow (RA)_{0:31} \times_{ui} (RB)_{0:31}$
$RT_{0:31} \leftarrow temp_{32:63}$
$temp_{0:63} \leftarrow (RA)_{32:63} \times_{ui} (RB)_{32:63}$
$RT_{32:63} \leftarrow temp_{32:63}$
$ACC_{0:63} \leftarrow (RT)_{0:63}$

The corresponding word unsigned-integer elements in RA and RB are multiplied. The least significant 32 bits of each product are placed into the two corresponding words of RT and into the accumulator.

Special Registers Altered:
ACC

Programming Note

The least significant 32 bits of the product are independent of whether the word elements in RA and RB are treated as signed or unsigned 32-bit integers.

Note that ***evmwlumia*** can be used for signed or unsigned integers.

Vector Multiply Word Low Unsigned, Modulo, Integer and Accumulate into Words *EVX-form*

evmwlumiaaw RT,RA,RB

4	RT	RA	RB	1352
0	6	11	16	21 31

$temp_{0:63} \leftarrow (RA)_{0:31} \times_{ui} (RB)_{0:31}$
$RT_{0:31} \leftarrow (ACC)_{0:31} + temp_{32:63}$

$temp_{0:63} \leftarrow (RA)_{32:63} \times_{ui} (RB)_{32:63}$
$RT_{32:63} \leftarrow (ACC)_{32:63} + temp_{32:63}$
$ACC_{0:63} \leftarrow (RT)_{0:63}$

For each word element in the accumulator, the corresponding word unsigned-integer elements in RA and RB are multiplied. The least significant 32 bits of each product are added to the contents of the corresponding accumulator word and the result is placed in RT and the accumulator.

Special Registers Altered:
ACC

Vector Multiply Word Low Unsigned, Modulo, Integer and Accumulate Negative in Words *EVX-form*

evmwlumianw RT,RA,RB

4	RT	RA	RB	1480
0	6	11	16	21 31

$temp_{0:63} \leftarrow (RA)_{0:31} \times_{ui} (RB)_{0:31}$
$RT_{0:31} \leftarrow (ACC)_{0:31} - temp_{32:63}$
$temp_{0:63} \leftarrow (RA)_{32:63} \times_{ui} (RB)_{32:63}$
$RT_{32:63} \leftarrow (ACC)_{32:63} - temp_{32:63}$
$ACC_{0:63} \leftarrow (RT)_{0:63}$

For each word element in the accumulator, the corresponding word unsigned-integer elements in RA and RB are multiplied. The least significant 32 bits of each product are subtracted from the contents of the corresponding accumulator word and the result is placed in RT and the accumulator.

Special Registers Altered:
ACC

Vector Multiply Word Low Unsigned, Saturate, Integer and Accumulate into Words *EVX-form*

evmwlusiaaw RT,RA,RB

4	RT	RA	RB	1344
0	6	11	16	21 31

```
temp0:63 ← (RA)0:31 ×ui (RB)0:31
temp0:63 ← EXTZ((ACC)0:31) + EXTZ(temp32:63)
ovh ← temp31
RT0:31 ← SATURATE(ovh, 0, 0xFFFF_FFFF, 0xFFFF_FFFF,
                    temp32:63)
temp0:63 ← (RA)32:63 ×ui (RB)32:63
temp0:63 ← EXTZ((ACC)32:63) + EXTZ(temp32:63)
ovl ← temp31
RT32:63 ← SATURATE(ovl, 0, 0xFFFF_FFFF,
                    0xFFFF_FFFF, temp32:63)
ACC0:63 ← (RT)0:63
SPEFSCROVH ← ovh
SPEFSCROV ← ovl
SPEFSCRSOVH ← SPEFSCRSOVH | ovh
SPEFSCRSOV ← SPEFSCRSOV | ovl
```

For each word element in the accumulator, corresponding word unsigned-integer elements in RA and RB are multiplied producing a 64-bit product. The least significant 32 bits of each product are then added to the corresponding word in the accumulator saturating if overflow occurs, and the result is placed in RT and the accumulator.

Special Registers Altered:
ACC OV OVH SOV SOVH

Vector Multiply Word Low Unsigned, Saturate, Integer and Accumulate Negative in Words *EVX-form*

evmwlusianw RT,RA,RB

4	RT	RA	RB	1472
0	6	11	16	21 31

```
temp0:63 ← (RA)0:31 ×ui (RB)0:31
temp0:63 ← EXTZ((ACC)0:31) - EXTZ(temp32:63)
ovh ← temp31
RT0:31 ← SATURATE(ovh, 0, 0x0000_0000, 0x0000_0000,
                    temp32:63)
temp0:63 ← (RA)32:63 ×ui (RB)32:63
temp0:63 ← EXTZ((ACC)32:63) - EXTZ(temp32:63)
ovl ← temp31
RT32:63 ← SATURATE(ovl, 0, 0x0000_0000,
                    0x0000_0000, temp32:63)
ACC0:63 ← (RT)0:63
SPEFSCROVH ← ovh
SPEFSCROV ← ovl
SPEFSCRSOVH ← SPEFSCRSOVH | ovh
SPEFSCRSOV ← SPEFSCRSOV | ovl
```

For each word element in the accumulator, corresponding word unsigned-integer elements in RA and RB are multiplied producing a 64-bit product. The least significant 32 bits of each product are then subtracted from the corresponding word in the accumulator saturating if overflow occurs, and the result is placed in RT and the accumulator.

Special Registers Altered:
ACC OV OVH SOV SOVH

Vector Multiply Word Signed, Modulo, Fractional *EVX-form*

evmwsmf RT,RA,RB

4	RT	RA	RB	1115
0	6	11	16	21 31

```
RT0:63 ←(RA)32:63 ×sf (RB)32:63
```

The corresponding low word signed fractional elements in RA and RB are multiplied. The product is placed in RT.

Special Registers Altered:
None

Vector Multiply Word Signed, Modulo, Fractional to Accumulator *EVX-form*

evmwsmfa RT,RA,RB

4	RT	RA	RB	1147
0	6	11	16	21 31

```
RT0:63 ←(RA)32:63 ×sf (RB)32:63
ACC0:63 ←(RT)0:63
```

The corresponding low word signed fractional elements in RA and RB are multiplied. The product is placed in RT and into the accumulator.

Special Registers Altered:
ACC

Vector Multiply Word Signed, Modulo, Fractional and Accumulate EVX-form

evmwsmfaa RT,RA,RB

4	RT	RA	RB	1371
0	6	11	16	21 31

$temp_{0:63} \leftarrow (RA)_{32:63} \times_{sf} (RB)_{32:63}$
$RT_{0:63} \leftarrow (ACC)_{0:63} + temp_{0:63}$
$ACC_{0:63} \leftarrow (RT)_{0:63}$

The corresponding low word signed fractional elements in RA and RB are multiplied. The intermediate product is added to the contents of the 64-bit accumulator and the result is placed in RT and the accumulator.

Special Registers Altered:
ACC

Vector Multiply Word Signed, Modulo, Fractional and Accumulate Negative EVX-form

evmwsmfan RT,RA,RB

4	RT	RA	RB	1499
0	6	11	16	21 31

$temp_{0:63} \leftarrow (RA)_{32:63} \times_{sf} (RB)_{32:63}$
$RT_{0:63} \leftarrow (ACC)_{0:63} - temp_{0:63}$
$ACC_{0:63} \leftarrow (RT)_{0:63}$

The corresponding low word signed fractional elements in RA and RB are multiplied. The intermediate product is subtracted from the contents of the accumulator and the result is placed in RT and the accumulator.

Special Registers Altered:
ACC

Vector Multiply Word Signed, Modulo, Integer EVX-form

evmwsmi RT,RA,RB

4	RT	RA	RB	1113
0	6	11	16	21 31

$RT_{0:63} \leftarrow (RA)_{32:63} \times_{si} (RB)_{32:63}$

The low word signed-integer elements in RA and RB are multiplied. The product is placed in RT.

Special Registers Altered:
None

Vector Multiply Word Signed, Modulo, Integer to Accumulator EVX-form

evmwsmia RT,RA,RB

4	RT	RA	RB	1145
0	6	11	16	21 31

$RT_{0:63} \leftarrow (RA)_{32:63} \times_{si} (RB)_{32:63}$
$ACC_{0:63} \leftarrow (RT)_{0:63}$

The low word signed-integer elements in RA and RB are multiplied. The product is placed in RT and the accumulator.

Special Registers Altered:
ACC

Vector Multiply Word Signed, Modulo, Integer and Accumulate EVX-form

evmwsmiaa RT,RA,RB

4	RT	RA	RB	1369
0	6	11	16	21 31

$temp_{0:63} \leftarrow (RA)_{32:63} \times_{si} (RB)_{32:63}$
$RT_{0:63} \leftarrow (ACC)_{0:63} + temp_{0:63}$
$ACC_{0:63} \leftarrow (RT)_{0:63}$

The low word signed-integer elements in RA and RB are multiplied. The intermediate product is added to the contents of the 64-bit accumulator and the result is placed in RT and the accumulator.

Special Registers Altered:
ACC

Vector Multiply Word Signed, Modulo, Integer and Accumulate Negative EVX-form

evmwsmian RT,RA,RB

4	RT	RA	RB	1497
0	6	11	16	21 31

$temp_{0:63} \leftarrow (RA)_{32:63} \times_{si} (RB)_{32:63}$
$RT_{0:63} \leftarrow (ACC)_{0:63} - temp_{0:63}$
$ACC_{0:63} \leftarrow (RT)_{0:63}$

The low word signed-integer elements in RA and RB are multiplied. The intermediate product is subtracted from the contents of the 64-bit accumulator and the result is placed in RT and the accumulator.

Special Registers Altered:
ACC

Vector Multiply Word Signed, Saturate, Fractional EVX-form

evmwssf RT,RA,RB

4	RT	RA	RB	1107
0	6	11	16	21 31

```
temp0:63 ← (RA)32:63 ×sf (RB)32:63
if ((RA)32:63 = 0x8000_0000) & (RB32:63 = 0x8000_0000)
then
   RT0:63 ← 0x7FFF_FFFF_FFFF_FFFF
   mov ← 1
else
   RT0:63 ← temp0:63
   mov ← 0
SPEFSCROVH ← 0
SPEFSCROV ← mov
SPEFSCRSOV ← SPEFSCRSOV | mov
```

The low word signed fractional elements in RA and RB are multiplied. The 64-bit product is placed in RT. If both inputs are -1.0, the result saturates to the largest positive signed fraction.

Special Registers Altered:
OV OVH SOV

Vector Multiply Word Signed, Saturate, Fractional to Accumulator EVX-form

evmwssfa RT,RA,RB

4	RT	RA	RB	1139
0	6	11	16	21 31

```
temp0:63 ← (RA)32:63 ×sf (RB)32:63
if ((RA)32:63=0x8000_0000)&((RB)32:63=0x8000_0000)
then
   RT0:63 ← 0x7FFF_FFFF_FFFF_FFFF
   mov ← 1
else
   RT0:63 ← temp0:63
   mov ← 0
ACC0:63 ← (RT)0:63
SPEFSCROVH ← 0
SPEFSCROV ← mov
SPEFSCRSOV ← SPEFSCRSOV | mov
```

The low word signed fractional elements in RA and RB are multiplied. The 64-bit product is placed in RT and into the accumulator. If both inputs are -1.0, the result saturates to the largest positive signed fraction.

Special Registers Altered:
ACC OV OVH SOV

Vector Multiply Word Signed, Saturate, Fractional and Accumulate EVX-form

evmwssfaa RT,RA,RB

4	RT	RA	RB	1363
0	6	11	16	21 31

```
temp0:63 ← (RA)32:63 ×sf (RB)32:63
if ((RA)32:63=0x8000_0000)&((RB)32:63=0x8000_0000)
then
   temp0:63 ← 0x7FFF_FFFF_FFFF_FFFF
   mov ← 1
else
   mov ← 0
temp0:64 ← EXTS((ACC)0:63) + EXTS(temp0:63)
ov ← (temp0 ⊕ temp1)
RT0:63 ← temp1:64

ACC0:63 ← (RT)0:63
SPEFSCROVH ← 0
SPEFSCROV ← ov | mov
SPEFSCRSOV ← SPEFSCRSOV | ov | mov
```

The low word signed fractional elements in RA and RB are multiplied producing a 64-bit product. If both inputs are -1.0, the product saturates to the largest positive signed fraction. The 64-bit product is then added to the accumulator and the result is placed in RT and the accumulator.

Special Registers Altered:
ACC OV OVH SOV

Vector Multiply Word Signed, Saturate, Fractional and Accumulate Negative EVX-form

evmwssfan RT,RA,RB

4	RT	RA	RB	1491
0	6	11	16	21 31

```
temp0:63 ← (RA)32:63 ×sf (RB)32:63
if ((RA)32:63=0x8000_0000)&((RB)32:63=0x8000_0000)
then
   temp0:63 ← 0x7FFF_FFFF_FFFF_FFFF
   mov ← 1
else
   mov ← 0
temp0:64 ← EXTS((ACC)0:63) - EXTS(temp0:63)
ov ← (temp0 ⊕ temp1)
RT0:63 ← temp1:64
ACC0:63 ← (RT)0:63
SPEFSCROVH ← 0
SPEFSCROV ← ov | mov
SPEFSCRSOV ← SPEFSCRSOV | ov | mov
```

The low word signed fractional elements in RA and RB are multiplied producing a 64-bit product. If both inputs are -1.0, the product saturates to the largest positive signed fraction. The 64-bit product is then subtracted from the accumulator and the result is placed in RT and the accumulator.

Special Registers Altered:
ACC OV OVH SOV

Vector Multiply Word Unsigned, Modulo, Integer EVX-form

evmwumi RT,RA,RB

4	RT	RA	RB	1112
0	6	11	16	21 31

```
RT0:63 ← (RA)32:63 ×ui (RB)32:63
```

The low word unsigned-integer elements in RA and RB are multiplied to form a 64-bit product that is placed in RT.

Special Registers Altered:
None

Vector Multiply Word Unsigned, Modulo, Integer to Accumulator EVX-form

evmwumia RT,RA,RB

4	RT	RA	RB	1144
0	6	11	16	21 31

```
RT0:63 ← (RA)32:63 ×ui (RB)32:63
ACC0:63 ← (RT)0:63
```

The low word unsigned-integer elements in RA and RB are multiplied to form a 64-bit product that is placed in RT and into the accumulator.

Special Registers Altered:
ACC

Vector Multiply Word Unsigned, Modulo, Integer and Accumulate EVX-form

evmwumiaa RT,RA,RB

4	RT	RA	RB	1368
0	6	11	16	21 31

$temp_{0:63} \leftarrow (RA)_{32:63} \times_{ui} (RB)_{32:63}$
$RT_{0:63} \leftarrow (ACC)_{0:63} + temp_{0:63}$
$ACC_{0:63} \leftarrow (RT)_{0:63}$

The low word unsigned-integer elements in RA and RB are multiplied. The intermediate product is added to the contents of the 64-bit accumulator, and the resulting value is placed into the accumulator and in RT.

Special Registers Altered:
ACC

Vector Multiply Word Unsigned, Modulo, Integer and Accumulate Negative EVX-form

evmwumian RT,RA,RB

4	RT	RA	RB	1496
0	6	11	16	21 31

$temp_{0:63} \leftarrow (RA)_{32:63} \times_{ui} (RB)_{32:63}$
$RT_{0:63} \leftarrow (ACC)_{0:63} - temp_{0:63}$
$ACC_{0:63} \leftarrow (RT)_{0:63}$

The low word unsigned-integer elements in RA and RB are multiplied. The intermediate product is subtracted from the contents of the 64-bit accumulator, and the resulting value is placed into the accumulator and in RT.

Special Registers Altered:
ACC

Vector NAND EVX-form

evnand RT,RA,RB

4	RT	RA	RB	542
0	6	11	16	21 31

$RT_{0:31} \leftarrow \neg((RA)_{0:31} \ \& \ (RB)_{0:31})$
$RT_{32:63} \leftarrow \neg((RA)_{32:63} \ \& \ (RB)_{32:63})$

Each element of RA and RB is bitwise NANDed. The result is placed in the corresponding element of RT.

Special Registers Altered:
None

Vector Negate EVX-form

evneg RT,RA

4	RT	RA	///	521
0	6	11	16	21 31

$RT_{0:31} \leftarrow NEG((RA)_{0:31})$
$RT_{32:63} \leftarrow NEG((RA)_{32:63})$

The negative of each element of RA is placed in RT. The negative of 0x8000_0000 (most negative number) returns 0x8000_0000.

Special Registers Altered:
None

Vector NOR *EVX-form*

evnor RT,RA,RB

4	RT	RA	RB	536
0	6	11	16	21 31

$RT_{0:31} \leftarrow \neg((RA)_{0:31} \mid (RB)_{0:31})$
$RT_{32:63} \leftarrow \neg((RA)_{32:63} \mid (RB)_{32:63})$

Each element of RA and RB is bitwise NORed. The result is placed in the corresponding element of RT.

Special Registers Altered:
None

Extended Mnemonics:

Extended mnemonics are provided for the *Vector NOR* instruction to produce a vector bitwise complement operation.

Extended:	Equivalent to:
evnot RT,RA	***evnor*** RT,RA,RA

Vector OR *EVX-form*

evor RT,RA,RB

4	RT	RA	RB	535
0	6	11	16	21 31

$RT_{0:31} \leftarrow (RA)_{0:31} \mid (RB)_{0:31}$
$RT_{32:63} \leftarrow (RA)_{32:63} \mid (RB)_{32:63}$

Each element of RA and RB is bitwise ORed. The result is placed in the corresponding element of RT.

Special Registers Altered:
None

Extended Mnemonics:

Extended mnemonics are provided for the *Vector OR* instruction to provide a 64-bit vector move instruction.

Extended:	Equivalent to:
evmr RT,RA	***evor*** RT,RA,RA

Vector OR with Complement *EVX-form*

evorc RT,RA,RB

4	RT	RA	RB	539
0	6	11	16	21 31

$RT_{0:31} \leftarrow (RA)_{0:31} \mid (\neg(RB)_{0:31})$
$RT_{32:63} \leftarrow (RA)_{32:63} \mid (\neg(RB)_{32:63})$

Each element of RA is bitwise ORed with the complement of RB. The result is placed in the corresponding element of RT.

Special Registers Altered:
None

Vector Rotate Left Word *EVX-form*

evrlw RT,RA,RB

4	RT	RA	RB	552
0	6	11	16	21 31

$nh \leftarrow (RB)_{27:31}$
$nl \leftarrow (RB)_{59:63}$
$RT_{0:31} \leftarrow ROTL((RA)_{0:31}, nh)$
$RT_{32:63} \leftarrow ROTL((RA)_{32:63}, nl)$

Each of the high and low elements of RA is rotated left by an amount specified in RB. The result is placed in RT. Rotate values for each element of RA are found in bit positions $RB_{27:31}$ and $RB_{59:63}$.

Special Registers Altered:
None

Vector Rotate Left Word Immediate EVX-form

evrlwi RT,RA,UI

4	RT	RA	UI	554
0	6	11	16	21 31

```
n ← UI
RT0:31 ← ROTL((RA)0:31, n)
RT32:63 ← ROTL((RA)32:63, n)
```

Both the high and low elements of RA are rotated left by an amount specified by UI.

Special Registers Altered:
None

Vector Select EVS-form

evsel RT,RA,RB,BFA

4	RT	RA	RB	79	BFA
0	6	11	16	21	29 31

```
ch ← CRBFA×4
cl ← CRBFA×4+1
if (ch = 1) then RT0:31 ← (RA)0:31
else RT0:31 ← (RB)0:31
if (cl = 1) then RT32:63 ← (RA)32:63
else RT32:63 ← (RB)32:63
```

If the most significant bit in the BFA field of CR is set to 1, the high-order element of RA is placed in the high-order element of RT; otherwise, the high-order element of RB is placed into the high-order element of RT. If the next most significant bit in the BFA field of CR is set to 1, the low-order element of RA is placed in the low-order element of RT, otherwise, the low-order element of RB is placed into the low-order element of RT.

Special Registers Altered:
None

Vector Round Word EVX-form

evrndw RT,RA

4	RT	RA	///	524
0	6	11	16	21 31

```
RT0:31 ← ((RA)0:31+0x00008000) & 0xFFFF0000
RT32:63 ← ((RA)32:63+0x00008000) & 0xFFFF0000
```

The 32-bit elements of RA are rounded into 16 bits. The result is placed in RT. The resulting 16 bits are placed in the most significant 16 bits of each element of RT, zeroing out the low-order 16 bits of each element.

Special Registers Altered:
None

Vector Shift Left Word EVX-form

evslw RT,RA,RB

4	RT	RA	RB	548
0	6	11	16	21 31

$nh \leftarrow (RB)_{26:31}$
$nl \leftarrow (RB)_{58:63}$
$RT_{0:31} \leftarrow SL((RA)_{0:31}, nh)$
$RT_{32:63} \leftarrow SL((RA)_{32:63}, nl)$

Each of the high and low elements of RA is shifted left by an amount specified in RB. The result is placed in RT. The separate shift amounts for each element are specified by 6 bits in RB that lie in bit positions 26:31 and 58:63.

Shift amounts from 32 to 63 give a zero result.

Special Registers Altered:
None

Vector Shift Left Word Immediate EVX-form

evslwi RT,RA,UI

4	RT	RA	UI	550
0	6	11	16	21 31

$n \leftarrow UI$
$RT_{0:31} \leftarrow SL((RA)_{0:31}, n)$
$RT_{32:63} \leftarrow SL((RA)_{32:63}, n)$

Both high and low elements of RA are shifted left by the 5-bit UI value and the results are placed in RT.

Special Registers Altered:
None

Vector Splat Fractional Immediate EVX-form

evsplatfi RT,SI

4	RT	SI	///	555
0	6	11	16	21 31

$RT_{0:31} \leftarrow SI \; || \; ^{27}0$
$RT_{32:63} \leftarrow SI \; || \; ^{27}0$

The value specified by SI is padded with trailing zeros and placed in both elements of RT. The SI ends up in bit positions $RT_{0:4}$ and $RT_{32:36}$.

Special Registers Altered:
None

Vector Splat Immediate EVX-form

evsplati RT,SI

4	RT	SI	///	553
0	6	11	16	21 31

$RT_{0:31} \leftarrow EXTS(SI)$
$RT_{32:63} \leftarrow EXTS(SI)$

The value specified by SI is sign extended and placed in both elements of RT.

Special Registers Altered:
None

Vector Shift Right Word Immediate Signed EVX-form

evsrwis RT,RA,UI

4	RT	RA	UI	547
0	6	11	16	21 31

$n \leftarrow UI$
$RT_{0:31} \leftarrow EXTS((RA)_{0:31-n})$
$RT_{32:63} \leftarrow EXTS((RA)_{32:63-n})$

Both high and low elements of RA are shifted right by the 5-bit UI value. Bits in the most significant positions vacated by the shift are filled with a copy of the sign bit.

Special Registers Altered:
None

Vector Shift Right Word Immediate Unsigned EVX-form

evsrwiu RT,RA,UI

4	RT	RA	UI	546
0	6	11	16	21 31

$n \leftarrow UI$
$RT_{0:31} \leftarrow EXTZ((RA)_{0:31-n})$
$RT_{32:63} \leftarrow EXTZ((RA)_{32:63-n})$

Both high and low elements of RA are shifted right by the 5-bit UI value; zeros are shifted into the most significant position.

Special Registers Altered:
None

Vector Shift Right Word Signed EVX-form

evsrws RT,RA,RB

4	RT	RA	RB	545
0	6	11	16	21 31

$nh \leftarrow (RB)_{26:31}$
$nl \leftarrow (RB)_{58:63}$
$RT_{0:31} \leftarrow EXTS((RA)_{0:31-nh})$
$RT_{32:63} \leftarrow EXTS((RA)_{32:63-nl})$

Both the high and low elements of RA are shifted right by an amount specified in RB. The result is placed in RT. The separate shift amounts for each element are specified by 6 bits in RB that lie in bit positions 26:31 and 58:63. The sign bits are shifted into the most significant position.

Shift amounts from 32 to 63 give a result of 32 sign bits.

Special Registers Altered:
None

Vector Shift Right Word Unsigned EVX-form

evsrwu RT,RA,RB

4	RT	RA	RB	544
0	6	11	16	21 31

$nh \leftarrow (RB)_{26:31}$
$nl \leftarrow (RB)_{58:63}$
$RT_{0:31} \leftarrow EXTZ((RA)_{0:31-nh})$
$RT_{32:63} \leftarrow EXTZ((RA)_{32:63-nl})$

Both the high and low elements of RA are shifted right by an amount specified in RB. The result is placed in RT. The separate shift amounts for each element are specified by 6 bits in RB that lie in bit positions 26:31 and 58:63. Zeros are shifted into the most significant position.

Shift amounts from 32 to 63 give a zero result.

Special Registers Altered:
None

Vector Store Double of Double EVX-form

evstdd RS,D(RA)

4	RS	RA	UI	801
0	6	11	16	21 31

```
if (RA = 0) then b ← 0
else b ← (RA)
EA ← b + EXTZ(UI×8)
MEM(EA,8) ← (RS)0:63
```

D in the instruction mnemonic is UI × 8. The contents of RS are stored as a doubleword in storage addressed by EA.

Special Registers Altered:
None

Vector Store Double of Double Indexed EVX-form

evstddx RS,RA,RB

4	RS	RA	RB	800
0	6	11	16	21 31

```
if (RA = 0) then b ← 0
else b ← (RA)
EA ← b + (RB)
MEM(EA,8) ← (RS)0:63
```

The contents of RS are stored as a doubleword in storage addressed by EA.

Special Registers Altered:
None

Vector Store Double of Four Halfwords EVX-form

evstdh RS,D(RA)

4	RS	RA	UI	805
0	6	11	16	21 31

```
if (RA = 0) then b ← 0
else b ← (RA)
EA ← b + EXTZ(UI×8)
MEM(EA,2) ← (RS)0:15
MEM(EA+2,2) ← (RS)16:31
MEM(EA+4,2) ← (RS)32:47
MEM(EA+6,2) ← (RS)48:63
```

D in the instruction mnemonic is UI × 8. The contents of RS are stored as four halfwords in storage addressed by EA.

Special Registers Altered:
None

Vector Store Double of Four Halfwords Indexed EVX-form

evstdhx RS,RA,RB

4	RS	RA	RB	804
0	6	11	16	21 31

```
if (RA = 0) then b ← 0
else b ← (RA)
EA ← b + (RB)
MEM(EA,2) ← (RS)0:15
MEM(EA+2,2) ← (RS)16:31
MEM(EA+4,2) ← (RS)32:47
MEM(EA+6,2) ← (RS)48:63
```

The contents of RS are stored as four halfwords in storage addressed by EA.

Special Registers Altered:
None

Vector Store Double of Two Words EVX-form

evstdw RS,D(RA)

4	RS	RA	UI	803
0	6	11	16	21 31

```
if (RA = 0) then b ← 0
else b ← (RA)
EA ← b + EXTZ(UI×8)
MEM(EA,4) ← (RS)0:31
MEM(EA+4,4) ← (RS)32:63
```

D in the instruction mnemonic is UI × 8. The contents of RS are stored as two words in storage addressed by EA.

Special Registers Altered:
None

Vector Store Double of Two Words Indexed EVX-form

evstdwx RS,RA,RB

4	RS	RA	RB	802
0	6	11	16	21 31

```
if (RA = 0) then b ← 0
else b ← (RA)
EA ← b + (RB)
MEM(EA,4) ← (RS)0:31
MEM(EA+4,4) ← (RS)32:63
```

The contents of RS are stored as two words in storage addressed by EA.

Special Registers Altered:
None

Vector Store Word of Two Halfwords from Even EVX-form

evstwhe RS,D(RA)

4	RS	RA	UI	817
0	6	11	16	21–31

```
if (RA = 0) then b ← 0
else b ← (RA)
EA ← b + EXTZ(UI×4)
MEM(EA,2) ← (RS)0:15
MEM(EA+2,2) ← (RS)32:47
```

D in the instruction mnemonic is UI × 4. The even halfwords from each element of RS are stored as two halfwords in storage addressed by EA.

Special Registers Altered:
None

Vector Store Word of Two Halfwords from Even Indexed EVX-form

evstwhex RS,RA,RB

4	RS	RA	RB	816
0	6	11	16	21–31

```
if (RA = 0) then b ← 0
else b ← (RA)
EA ← b + (RB)
MEM(EA,2) ← (RS)0:15
MEM(EA+2,2) ← (RS)32:47
```

The even halfwords from each element of RS are stored as two halfwords in storage addressed by EA.

Special Registers Altered:
None

Vector Store Word of Two Halfwords from Odd EVX-form

evstwho RS,D(RA)

4	RS	RA	UI	821
0	6	11	16	21–31

```
if (RA = 0) then b ← 0
else b ← (RA)
EA ← b + EXTZ(UI×4)
MEM(EA,2) ← (RS)16:31
MEM(EA+2,2) ← (RS)48:63
```

D in the instruction mnemonic is UI × 4. The odd halfwords from each element of RS are stored as two halfwords in storage addressed by EA.

Special Registers Altered:
None

Vector Store Word of Two Halfwords from Odd Indexed EVX-form

evstwhox RS,RA,RB

4	RS	RA	RB	820
0	6	11	16	21–31

```
if (RA = 0) then b ← 0
else b ← (RA)
EA ← b + (RB)
MEM(EA,2) ← (RS)16:31
MEM(EA+2,2) ← (RS)48:63
```

The odd halfwords from each element of RS are stored as two halfwords in storage addressed by EA.

Special Registers Altered:
None

Vector Store Word of Word from Even EVX-form

evstwwe RS,D(RA)

4	RS	RA	UI	825
0	6	11	16	21–31

```
if (RA = 0) then b ← 0
else b ← (RA)
EA ← b + EXTZ(UI×4)
MEM(EA,4) ← (RS)0:31
```

D in the instruction mnemonic is UI × 4. The even word of RS is stored in storage addressed by EA.

Special Registers Altered:
None

Vector Store Word of Word from Even Indexed EVX-form

evstwwex RS,RA,RB

4	RS	RA	RB	824
0	6	11	16	21–31

```
if (RA = 0) then b ← 0
else b ← (RA)
EA ← b + (RB)
MEM(EA,4) ← (RS)0:31
```

The even word of RS is stored in storage addressed by EA.

Special Registers Altered:
None

Vector Store Word of Word from Odd EVX-form

evstwwo RS,D(RA)

4	RS	RA	UI	829
0	6	11	16	21 31

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

D in the instruction mnemonic is UI × 4. The odd word of RS is stored in storage addressed by EA.

Special Registers Altered:
None

Vector Store Word of Word from Odd Indexed EVX-form

evstwwox RS,RA,RB

4	RS	RA	RB	828
0	6	11	16	21 31

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

The odd word of RS is stored in storage addressed by EA.

Special Registers Altered:
None

Vector Subtract Signed, Modulo, Integer to Accumulator Word EVX-form

evsubfsmiaaw RT,RA

4	RT	RA	///	1227
0	6	11	16	21 31

```
RT0:31 ← (ACC)0:31 - (RA)0:31
RT32:63 ← (ACC)32:63 - (RA)32:63
ACC0:63 ← (RT)0:63
```

Each word element in RA is subtracted from the corresponding element in the accumulator and the difference is placed into the corresponding RT word and into the accumulator.

Special Registers Altered:
ACC

Vector Subtract Signed, Saturate, Integer to Accumulator Word EVX-form

evsubfssiaaw RT,RA

4	RT	RA	///	1219
0	6	11	16	21 31

```
temp0:63 ← EXTS((ACC)0:31) - EXTS((RA)0:31)
ovh ← temp31 ⊕ temp32
RT0:31 ← SATURATE(ovh, temp31, 0x8000_0000,
                  0x7FFF_FFFF, temp32:63)
temp0:63 ← EXTS((ACC)32:63) - EXTS((RA)32:63)
ovl ← temp31 ⊕ temp32
RT32:63 ← SATURATE(ovl, temp31, 0x8000_0000,
                   0x7FFF_FFFF, temp32:63)
ACC0:63 ← (RT)0:63
SPEFSCROVH ← ovh
SPEFSCROV ← ovl
SPEFSCRSOVH ← SPEFSCRSOVH | ovh
SPEFSCRSOV ← SPEFSCRSOV | ovl
```

Each signed-integer word element in RA is sign-extended and subtracted from the corresponding sign-extended element in the accumulator saturating if overflow occurs, and the results are placed in RT and the accumulator.

Special Registers Altered:
ACC OV OVH SOV SOVH

Vector Subtract Unsigned, Modulo, Integer to Accumulator Word EVX-form

evsubfumiaaw RT,RA

4 (0)	RT (6)	RA (11)	/// (16)	1226 (21–31)

$RT_{0:31} \leftarrow (ACC)_{0:31} - (RA)_{0:31}$
$RT_{32:63} \leftarrow (ACC)_{32:63} - (RA)_{32:63}$
$ACC_{0:63} \leftarrow (RT)_{0:63}$

Each unsigned-integer word element in RA is subtracted from the corresponding element in the accumulator and the results are placed in RT and into the accumulator.

Special Registers Altered:
ACC

Vector Subtract Unsigned, Saturate, Integer to Accumulator Word EVX-form

evsubfusiaaw RT,RA

4 (0)	RT (6)	RA (11)	/// (16)	1218 (21–31)

$temp_{0:63} \leftarrow EXTZ((ACC)_{0:31}) - EXTZ((RA)_{0:31})$
$ovh \leftarrow temp_{31}$
$RT_{0:31} \leftarrow SATURATE(ovh, temp_{31}, 0x0000\_0000, 0x0000\_0000, temp_{32:63})$
$temp_{0:63} \leftarrow EXTS((ACC)_{32:63}) - EXTS((RA)_{32:63})$
$ovl \leftarrow temp_{31}$
$RT_{32:63} \leftarrow SATURATE(ovl, temp_{31}, 0x0000\_0000, 0x0000\_0000, temp_{32:63})$
$ACC_{0:63} \leftarrow (RT)_{0:63}$

$SPEFSCR_{OVH} \leftarrow ovh$
$SPEFSCR_{OV} \leftarrow ovl$
$SPEFSCR_{SOVH} \leftarrow SPEFSCR_{SOVH} \mid ovh$
$SPEFSCR_{SOV} \leftarrow SPEFSCR_{SOV} \mid ovl$

Each unsigned-integer word element in RA is zero-extended and subtracted from the corresponding zero-extended element in the accumulator saturating if overflow occurs, and the results are placed in RT and the accumulator.

Special Registers Altered:
ACC OV OVH SOV SOVH

Vector Subtract from Word EVX-form

evsubfw RT,RA,RB

4 (0)	RT (6)	RA (11)	RB (16)	516 (21–31)

$RT_{0:31} \leftarrow (RB)_{0:31} - (RA)_{0:31}$
$RT_{32:63} \leftarrow (RB)_{32:63} - (RA)_{32:63}$

Each signed-integer element of RA is subtracted from the corresponding element of RB and the results are placed in RT.

Special Registers Altered:
None

Vector Subtract Immediate from Word EVX-form

evsubifw RT,UI,RB

4 (0)	RT (6)	UI (11)	RB (16)	518 (21–31)

$RT_{0:31} \leftarrow (RB)_{0:31} - EXTZ(UI)$
$RT_{32:63} \leftarrow (RB)_{32:63} - EXTZ(UI)$

UI is zero-extended and subtracted from both the high and low elements of RB. Note that the same value is subtracted from both elements of the register.

Special Registers Altered:
None

Vector XOR EVX-form

evxor RT,RA,RB

4 (0)	RT (6)	RA (11)	RB (16)	534 (21–31)

$RT_{0:31} \leftarrow (RA)_{0:31} \oplus (RB)_{0:31}$
$RT_{32:63} \leftarrow (RA)_{32:63} \oplus (RB)_{32:63}$

Each element of RA and RB is exclusive-ORed. The results are placed in RT.

Special Registers Altered:
None

Chapter 7. Embedded Floating-Point [Category: SPE.Embedded Float Scalar Double] [Category: SPE.Embedded Float Scalar Single] [Category: SPE.Embedded Float Vector]

7.1 Overview

The *Embedded Floating-Point* categories require the implementation of the Signal Processing Engine (SPE) category and consist of three distinct categories:

- Embedded vector single-precision floating-point (SPE.Embedded Float Vector [SP.FV])
- Embedded scalar single-precision floating-point (SPE.Embedded Float Scalar Single [SP.FS])
- Embedded scalar double-precision floating-point (SPE.Embedded Float Scalar Double [SP.FD])

Although each of these may be implemented independently, they are defined in a single chapter because it is likely that they may be implemented together.

References to *Embedded Floating-Point* categories, *Embedded Floating-Point* instructions, or *Embedded Floating-Point* operations apply to all 3 categories.

Single-precision floating-point is handled by the SPE.Embedded Float Vector and SPE.Embedded Float Scalar Single categories; double-precision floating-point is handled by the SPE.Embedded Float Scalar Double category.

7.2 Programming Model

Embedded floating-point operations are performed in the GPRs of the processor.

The SPE.Embedded Float Vector and SPE.Embedded Float Scalar Double categories require a GPR register file with thirty-two 64-bit registers as required by the Signal Processing Engine category.

The SPE.Embedded Float Scalar Single category requires a GPR register file with thirty-two 32-bit registers. When implemented with a 64-bit register file on a 32-bit implementation, instructions in this category only use and modify bits 32:63 of the GPR. In this case, bits 0:31 of the GPR are left unchanged by the operation. For 64-bit implementations, bits 0:31 are unchanged after the operation.

Instructions in the SPE.Embedded Float Scalar Double category operate on the entire 64 bits of the GPRs.

Instructions in the SPE.Embedded Float Vector category operate on the entire 64 bits of the GPRs as well, but contain two 32-bit data items that are operated on independently of each other in a SIMD fashion. The format of both data items is the same as the format of a data item in the SPE.Embedded Float Scalar Single category. The data item contained in bits 0:31 is called the 'high word'. The data item contained in bits 32:63 is called the 'low word'.

There are no record forms of *Embedded Floating-Point* instructions. *Embedded Floating-Point Compare* instructions treat NaNs, Infinity, and Denorm as normalized numbers for the comparison calculation when default results are provided.

7.2.1 Signal Processing Embedded Floating-Point Status and Control Register (SPEFSCR)

Status and control for the *Embedded Floating-Point* categories uses the SPEFSCR. This register is defined by the Signal Processing Engine category in Section 6.3.4. Status and control bits are shared for *Embedded Floating-Point* and SPE operations. Instructions in the SPE.Embedded Float Vector category affect both the high element (bits 34:39) and low element floating-point status flags (bits 50:55). Instructions in the SPE.Embedded Float Scalar Double and SPE.Embedded Float Scalar Single categories affect only the low element floating-point status flags and leave the high element floating-point status flags undefined.

7.2.2 Floating-Point Data Formats

Single-precision floating-point data elements are 32 bits wide with 1 sign bit (*s*), 8 bits of biased exponent (*e*) and 23 bits of fraction (*f*). Double-precision floating-point data elements are 64 bits wide with 1 sign bit (*s*), 11 bits of biased exponent (*e*) and 52 bits of fraction (*f*).

In the IEEE 754 specification, floating-point values are represented in a format consisting of three explicit fields (sign field, biased exponent field, and fraction field) and an implicit hidden bit.

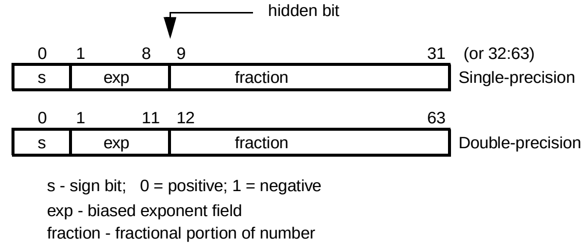


Figure 69. Floating-Point Data Format

For single-precision normalized numbers, the biased exponent value *e* lies in the range of 1 to 254 corresponding to an actual exponent value E in the range -126 to +127. For double-precision normalized numbers, the biased exponent value *e* lies in the range of 1 to 2046 corresponding to an actual exponent value E in the range -1022 to +1023. With the hidden bit implied to be '1' (for normalized numbers), the value of the number is interpreted as follows:

$$(-1)^{s} \times 2^{E} \times (1.\text{fraction})$$

where E is the unbiased exponent and 1.fraction is the mantissa (or significand) consisting of a leading '1' (the hidden bit) and a fractional part (fraction field). For the single-precision format, the maximum positive normalized number (*pmax*) is represented by the encoding `0x7F7FFFFF` which is approximately 3.4E+38 (2^{128}), and the minimum positive normalized value (*pmin*) is represented by the encoding `0x00800000` which is approximately 1.2E-38 (2^{-126}). For the double-precision format, the maximum positive normalized number (*pmax*) is represented by the encoding `0x7feFFFFF_FFFFFFFF` which is approximately 1.8E+307 (2^{1024}), and the minimum positive normalized value (*pmin*) is represented by the encoding `0x00100000_00000000` which is approximately 2.2E-308 (2^{-1022}).

Two specific values of the biased exponent are reserved (0 and 255 for single-precision; 0 and 2047 for double-precision) for encoding special values of +0, -0, +infinity, -infinity, and NaNs.

Zeros of both positive and negative sign are represented by a biased exponent value *e* of 0 and a fraction *f* which is 0.

Infinities of both positive and negative sign are represented by a maximum exponent field value (255 for single-precision, 2047 for double-precision) and a fraction which is 0.

Denormalized numbers of both positive and negative sign are represented by a biased exponent value *e* of 0 and a fraction *f,* which is nonzero. For these numbers, the hidden bit is defined by the IEEE 754 standard to be 0. This number type is not directly supported in hardware. Instead, either a software interrupt handler is invoked, or a default value is defined.

Not-a-Numbers (NaNs) are represented by a maximum exponent field value (255 for single-precision, 2047 for double-precision) and a fraction *f* which is nonzero.

Programming Note

On some implementations, operations that result in overflow or underflow are likely to take significantly longer than operations that do not. For example, these operations may cause a system error handler to be invoked; on such implementations, the system error handler updates the overflow bits appropriately.

7.2.3 Exception Conditions

7.2.3.1 Denormalized Values on Input

Any denormalized value used as an operand may be truncated by the implementation to a properly signed zero value.

7.2.3.2 Embedded Floating-Point Overflow and Underflow

Defining *pmax* to be the most positive normalized value (farthest from zero), *pmin* the smallest positive normalized value (closest to zero), *nmax* the most negative normalized value (farthest from zero) and *nmin* the smallest normalized negative value (closest to zero), an overflow is said to have occurred if the numerically correct result (*r*) of an instruction is such that $r>pmax$ or $r<nmax$. An underflow is said to have occurred if the numerically correct result of an instruction is such that $0<r<pmin$ or $nmin<r<0$. In this case, *r* may be denormalized, or may be smaller than the smallest denormalized number.

The *Embedded Floating-Point* categories do not produce +Infinity, -Infinity, NaN, or denormalized numbers. If the result of an instruction overflows and Embedded Floating-Point Overflow exceptions are disabled ($SPEFSCR_{FOVFE}$=0), *pmax* or *nmax* is generated as the result of that instruction depending upon the sign of the result. If the result of an instruction underflows and Embedded Floating-Point Underflow exceptions are disabled ($SPEFSCR_{FUNFE}$=0), +0 or -0 is generated as the result of that instruction based upon the sign of the result.

If an overflow occurs, $SPEFSCR_{FOVF\ FOVFH}$ are set appropriately, or if an underflow occurs, $SPEFSCR_{FUNF\ FUNFH}$ are set appropriately. If either Embedded Floating-Point Underflow or Embedded Floating-Point Overflow exceptions are enabled and a corresponding status bit is 1, an Embedded Floating-Point Data interrupt is taken and the destination register is not updated.

7.2.3.3 Embedded Floating-Point Invalid Operation/Input Errors

Embedded Floating-Point Invalid Operation/Input errors occur when an operand to an operation contains an invalid input value. If any of the input values are Infinity, Denorm, or NaN, or for an *Embedded Floating-Point Divide* instruction both operands are +/-0, $SPEFSCR_{FINV\ FINVH}$ are set to 1 appropriately, and $SPEFSCR_{FGH\ FXH\ FG\ FX}$ are set to 0 appropriately. If $SPEFSCR_{FINVE}$=1, an Embedded Floating-Point Data interrupt is taken and the destination register is not updated.

7.2.3.4 Embedded Floating-Point Round (Inexact)

If any result element of an *Embedded Floating-Point* instruction is inexact, or overflows but Embedded Floating-Point Overflow exceptions are disabled, or underflows but Embedded Floating-Point Underflow exceptions are disabled, and no higher priority interrupt occurs, $SPEFSCR_{FINXS}$ is set to 1. If the Embedded Floating-Point Round (Inexact) exception is enabled, an Embedded Floating-Point Round interrupt occurs. In this case, the destination register is updated with the truncated result(s). The $SPEFSCR_{FGH\ FXH\ FG\ FX}$ bits are properly updated to allow rounding to be performed in the interrupt handler.

$SPEFSCR_{FG\ FX}$ ($SPEFSCR_{FGH\ FXH}$) are set to 0 if an Embedded Floating-Point Data interrupt is taken due to overflow, underflow, or if an Embedded Floating-Point Invalid Operation/Input error is signaled for the low (high) element (regardless of $SPEFSCR_{FINVE}$).

7.2.3.5 Embedded Floating-Point Divide by Zero

If an *Embedded Floating-Point Divide* instruction executes and an Embedded Floating-Point Invalid Operation/Input error does not occur and the instruction is executed with a +/-0 divisor value and a finite normalized nonzero dividend value, an Embedded Floating-Point Divide By Zero exception occurs and $SPEFSCR_{FDBZ\ FDBZH}$ are set appropriately. If Embedded Floating-Point Divide By Zero exceptions are enabled, an Embedded Floating-Point Data interrupt is then taken and the destination register is not updated.

7.2.3.6 Default Results

Default results are generated when an Embedded Floating-Point Invalid Operation/Input Error, Embedded Floating-Point Overflow, Embedded Floating-Point Underflow, or Embedded Floating-Point Divide by Zero occurs on an *Embedded Floating-Point* operation. Default results provide a normalized value as a result of the operation. In general, Denorm results and underflows are set to 0 and overflows are saturated to the maximum representable number.

Default results produced for each operation are described in Section 7.4, “Embedded Floating-Point Results Summary”.

7.2.4 IEEE 754 Compliance

The *Embedded Floating-Point* categories require a floating-point system as defined in the ANSI/IEEE Standard 754-1985 but may rely on software support in order to conform fully with the standard. Thus, whenever an input operand of the *Embedded Floating-Point* instruction has data values that are +Infinity, -Infinity, Denormalized, NaN, or when the result of an operation produces an overflow or an underflow, an Embedded Floating-Point Data interrupt may be taken and the interrupt handler is responsible for delivering IEEE 754 compliant behavior if desired.

When Embedded Floating-Point Invalid Operation/Input Error exceptions are disabled ($SPEFSCR_{FINVE}$ = 0), default results are provided by the hardware when an Infinity, Denormalized, or NaN input is received, or for the operation 0/0. When Embedded Floating-Point Underflow exceptions are disabled ($SPEFSCR_{FUNFE}$ = 0) and the result of a floating-point operation underflows, a signed zero result is produced. The Embedded Floating-Point Round (Inexact) exception is also signaled for this condition. When Embedded Floating-Point Overflow exceptions are disabled ($SPEFSCR_{FOVFE}$ = 0) and the result of a floating-point operation overflows, a *pmax* or *nmax* result is produced. The Embedded Floating-Point Round (Inexact) exception is also signaled for this condition. An exception enable flag ($SPEFSCR_{FINXE}$) is also provided for generating an Embedded Floating-Point Round interrupt when an inexact result is produced, to allow a software handler to conform to the IEEE 754 standard. An Embedded Floating-Point Divide By Zero exception enable flag ($SPEFSCR_{FDBZE}$) is provided for generating an Embedded Floating-Point Data interrupt when a divide by zero operation is attempted to allow a software handler to conform to the IEEE 754 standard. All of these exceptions may be disabled, and the hardware will then deliver an appropriate default result.

The sign of the result of an addition operation is the sign of the source operand having the larger absolute value. If both operands have the same sign, the sign of the result is the same as the sign of the operands. This includes subtraction which is addition with the negation of the sign of the second operand. The sign of the result of an addition operation with operands of differing signs for which the result is zero is positive except when rounding to negative infinity. Thus -0 + -0 = -0, and all other cases which result in a zero value give +0 unless the rounding mode is round to negative infinity.

Programming Note

Note that when exceptions are disabled and default results computed, operations having input values that are denormalized may provide different results on different implementations. An implementation may choose to use the denormalized value or a zero value for any computation. Thus a computational operation involving a denormalized value and a normal value may return different results depending on the implementation.

7.2.4.1 Sticky Bit Handling For Exception Conditions

The SPEFSCR register defines sticky bits for retaining information about exception conditions that are detected. There are 5 sticky bits (FINXS, FINVS, FDBZS, FUNFS and FOVFS) that can be used to help provide IEEE 754 compliance. The sticky bits represent the combined ‘or’ of all the previous status bits produced from any *Embedded Floating-Point* operation since the last time software zeroed the sticky bit. The hardware will never set a sticky bit to 0.

7.3 Embedded Floating-Point Instructions

7.3.1 Load/Store Instructions

Embedded Floating-Point instructions use GPRs to hold and operate on floating-point values. The *Embedded Floating-Point* categories do not define *Load* and *Store* instructions to move the data to and from memory, but instead rely on existing instructions in Book I to load and store data.

.

7.3.2 SPE.Embedded Float Vector Instructions [Category: SPE.Embedded Float Vector]

All SPE.Embedded Float Vector instructions are single-precision. There are no vector floating-point double-precision instructions

Vector Floating-Point Single-Precision Absolute Value EVX-form

evfsabs RT,RA

4	RT	RA	///	644
0	6	11	16	21 31

$RT_{0:31} \leftarrow$ 0b0 || $(RA)_{1:31}$
$RT_{32:63} \leftarrow$ 0b0 || $(RA)_{33:63}$

The sign bit of each element in register RA is set to 0 and the results are placed into register RT.

Regardless of the value of register RA, no exceptions are taken during the execution of this instruction.

Special Registers Altered:
None

Vector Floating-Point Single-Precision Negative Absolute Value EVX-form

evfsnabs RT,RA

4	RT	RA	///	645
0	6	11	16	21 31

$RT_{0:31} \leftarrow$ 0b1 || $(RA)_{1:31}$
$RT_{32:63} \leftarrow$ 0b1 || $(RA)_{33:63}$

The sign bit of each element in register RA is set to 1 and the results are placed into register RT.

Regardless of the value of register RA, no exceptions are taken during the execution of this instruction.

Special Registers Altered:
None

Vector Floating-Point Single-Precision Negate EVX-form

evfsneg RT,RA

4	RT	RA	///	646
0	6	11	16	21 31

$RT_{0:31} \leftarrow \neg(RA)_0$ || $(RA)_{1:31}$
$RT_{32:63} \leftarrow \neg(RA)_{32}$ || $(RA)_{33:63}$

The sign bit of each element in register RA is complemented and the results are placed into register RT.

Regardless of the value of register RA, no exceptions are taken during the execution of this instruction.

Special Registers Altered:
None

Vector Floating-Point Single-Precision Add EVX-form

evfsadd RT,RA,RB

4	RT	RA	RB	640
0	6	11	16	21 31

$RT_{0:31} \leftarrow (RA)_{0:31} +_{sp} (RB)_{0:31}$
$RT_{32:63} \leftarrow (RA)_{32:63} +_{sp} (RB)_{32:63}$

Each single-precision floating-point element of register RA is added to the corresponding element of register RB and the results are stored in register RT.

If an underflow occurs, +0 (for rounding modes RN, RZ, RP) or -0 (for rounding mode RM) is stored in the corresponding element of register RT.

Special Registers Altered:
FINV FINVH FINVS
FGH FXH FG FX FINXS
FOVF FOVFH FOVFS
FUNF FUNFH FUNFS

Vector Floating-Point Single-Precision Subtract EVX-form

evfssub RT,RA,RB

4	RT	RA	RB	641
0	6	11	16	21 31

$RT_{0:31} \leftarrow (RA)_{0:31} -_{sp} (RB)_{0:31}$
$RT_{32:63} \leftarrow (RA)_{32:63} -_{sp} (RB)_{32:63}$

Each single-precision floating-point element of register RB is subtracted from the corresponding element of register RA and the results are stored in register RT.

If an underflow occurs, +0 (for rounding modes RN, RZ, RP) or -0 (for rounding mode RM) is stored in the corresponding element of register RT.

Special Registers Altered:
FINV FINVH FINVS
FGH FXH FG FX FINXS
FOVF FOVFH FOVFS
FUNF FUNFH FUNFS

Vector Floating-Point Single-Precision Multiply EVX-form

evfsmul RT,RA,RB

4	RT	RA	RB	648
0	6	11	16	21 31

$RT_{0:31} \leftarrow (RA)_{0:31} \times_{sp} (RB)_{0:31}$
$RT_{32:63} \leftarrow (RA)_{32:63} \times_{sp} (RB)_{32:63}$

Each single-precision floating-point element of register RA is multiplied with the corresponding element of register RB and the result is stored in register RT.

Special Registers Altered:
FINV FINVH FINVS
FGH FXH FG FX FINXS
FOVF FOVFH FOVFS
FUNF FUNFH FUNFS

Vector Floating-Point Single-Precision Divide EVX-form

evfsdiv RT,RA,RB

4	RT	RA	RB	649
0	6	11	16	21 31

$RT_{0:31} \leftarrow (RA)_{0:31} \div_{sp} (RB)_{0:31}$
$RT_{32:63} \leftarrow (RA)_{32:63} \div_{sp} (RB)_{32:63}$

Each single-precision floating-point element of register RA is divided by the corresponding element of register RB and the result is stored in register RT.

Special Registers Altered:
FINV FINVH FINVS
FGH FXH FG FX FINXS
FDBZ FDBZH FDBZS
FOVF FOVFH FOVFS
FUNF FUNFH FUNFS

Vector Floating-Point Single-Precision Compare Greater Than EVX-form

evfscmpgt BF,RA,RB

4	BF	//	RA	RB	652
0	6	9	11	16	21 31

```
ah ← (RA)0:31
al ← (RA)32:63
bh ← (RB)0:31
bl ←(RB)32:63
if (ah > bh) then ch ← 1
else ch ← 0
if (al > bl) then cl ← 1
else cl ← 0
CR4×BF:4×BF+3 ← ch || cl || (ch | cl) || (ch & cl)
```

Each element of register RA is compared against the corresponding element of register RB. The results of the comparisons are placed into CR field BF. If $RA_{0:31}$ is greater than $RB_{0:31}$, bit 0 of CR field BF is set to 1, otherwise it is set to 0. If $RA_{32:63}$ is greater than $RB_{32:63}$, bit 1 of CR field BF is set to 1, otherwise it is set to 0. Bit 2 of CR field BF is set to the OR of both result bits and Bit 3 of CR field BF is set to the AND of both result bits. Comparison ignores the sign of 0 (+0 = -0).

If an input error occurs and default results are generated, NaNs, Infinities, and Denorms as treated as normalized numbers, using their values of ‘*e*’ and ‘*f*’ directly.

Special Registers Altered:

FINV FINVH FINVS
FGH FXH FG FX
CR field BF

Vector Floating-Point Single-Precision Compare Less Than EVX-form

evfscmplt BF,RA,RB

4	BF	//	RA	RB	653
0	6	9	11	16	21 31

```
ah ← (RA)0:31
al ← (RA)32:63
bh ← (RB)0:31
bl ←(RB)32:63
if (ah < bh) then ch ← 1
else ch ← 0
if (al < bl) then cl ← 1
else cl ← 0
CR4×BF:4×BF+3 ← ch || cl || (ch | cl) || (ch & cl)
```

Each element of register RA is compared against the corresponding element of register RB. The results of the comparisons are placed into CR field BF. If $RA_{0:31}$ is less than $RB_{0:31}$, bit 0 of CR field BF is set to 1, otherwise it is set to 0. If $RA_{32:63}$ is less than $RB_{32:63}$, bit 1 of CR field BF is set to 1, otherwise it is set to 0. Bit 2 of CR field BF is set to the OR of both result bits and Bit 3 of CR field BF is set to the AND of both result bits. Comparison ignores the sign of 0 (+0 = -0).

If an input error occurs and default results are generated, NaNs, Infinities, and Denorms as treated as normalized numbers, using their values of ‘*e*’ and ‘*f*’ directly.

Special Registers Altered:

FINV FINVH FINVS
FGH FXH FG FX
CR field BF

Vector Floating-Point Single-Precision Compare Equal *EVX-form*

evfscmpeq BF,RA,RB

4	BF	//	RA	RB	654
0	6	9	11	16	21 31

```
ah ← (RA)0:31
al ← (RA)32:63
bh ← (RB)0:31
bl ←(RB)32:63
if (ah = bh) then ch ← 1
else ch ← 0
if (al = bl) then cl ← 1
else cl ← 0
CR4×BF:4×BF+3 ← ch || cl || (ch | cl) || (ch & cl)
```

Each element of register RA is compared against the corresponding element of register RB. The results of the comparisons are placed into CR field BF. If $RA_{0:31}$ is equal to $RB_{0:31}$, bit 0 of CR field BF is set to 1, otherwise it is set to 0. If $RA_{32:63}$ is equal to $RB_{32:63}$, bit 1 of CR field BF is set to 1, otherwise it is set to 0. Bit 2 of CR field BF is set to the OR of both result bits and Bit 3 of CR field BF is set to the AND of both result bits. Comparison ignores the sign of 0 (+0 = -0).

If an input error occurs and default results are generated, NaNs, Infinities, and Denorms as treated as normalized numbers, using their values of '*e*' and '*f*' directly.

Special Registers Altered:

FINV FINVH FINVS
FGH FXH FG FX
CR field BF

Vector Floating-Point Single-Precision Test Greater Than *EVX-form*

evfststgt BF,RA,RB

4	BF	//	RA	RB	668
0	6	9	11	16	21 31

```
ah ← (RA)0:31
al ← (RA)32:63
bh ← (RB)0:31
bl ← (RB)32:63
if (ah > bh) then ch ← 1
else ch ← 0
if (al > bl) then cl ← 1
else cl ← 0
CR4×BF:4×BF+3 ← ch || cl || (ch | cl) || (ch & cl)
```

Each element of register RA is compared against the corresponding element of register RB.The results of the comparisons are placed into CR field BF. If $RA_{0:31}$ is greater than $RB_{0:31}$, bit 0 of CR field BF is set to 1, otherwise it is set to 0. If $RA_{32:63}$ is greater than $RB_{32:63}$, bit 1 of CR field BF is set to 1, otherwise it is set to 0. Bit 2 of CR field BF is set to the OR of both result bits and Bit 3 of CR field BF is set to the AND of both result bits. Comparison ignores the sign of 0 (+0 = -0). The comparison proceeds after treating NaNs, Infinities, and Denorms as normalized numbers, using their values of '*e*' and '*f*' directly.

No exceptions are taken during the execution of ***evfststgt***.

Special Registers Altered:

CR field BF

Programming Note

In an implementation, the execution of ***evfststgt*** is likely to be faster than the execution of ***evfscmpgt***; however, if strict IEEE 754 compliance is required, the program should use ***evfscmpgt***.

Vector Floating-Point Single-Precision Test Less Than *EVX-form*

evfststlt BF,RA,RB

4	BF	//	RA	RB	669
0	6	9	11	16	21 31

```
ah ← (RA)0:31
al ← (RA)32:63
bh ← (RB)0:31
bl ← (RB)32:63
if (ah < bh) then ch ← 1
else ch ← 0
if (al < bl) then cl ← 1
else cl ← 0
CR4×BF:4×BF+3 ← ch || cl || (ch | cl) || (ch & cl)
```

Each element of register RA is compared with the corresponding element of register RB. The results of the comparisons are placed into CR field BF. If $RA_{0:31}$ is less than $RB_{0:31}$, bit 0 of CR field BF is set to 1, otherwise it is set to 0. If $RA_{32:63}$ is less than $RB_{32:63}$, bit 1 of CR field BF is set to 1, otherwise it is set to 0. Bit 2 of CR field BF is set to the OR of both result bits and Bit 3 of CR field BF is set to the AND of both result bits. Comparison ignores the sign of 0 (+0 = -0). The comparison proceeds after treating NaNs, Infinities, and Denorms as normalized numbers, using their values of '*e*' and '*f*' directly.

No exceptions are taken during the execution of ***evfststlt***.

Special Registers Altered:
CR field BF

Programming Note

In an implementation, the execution of ***evfststlt*** is likely to be faster than the execution of ***evfscmplt***; however, if strict IEEE 754 compliance is required, the program should use ***evfscmplt***.

Vector Floating-Point Single-Precision Test Equal *EVX-form*

evfststeq BF,RA,RB

4	BF	//	RA	RB	670
0	6	9	11	16	21 31

```
ah ← (RA)0:31
al ← (RA)32:63
bh ← (RB)0:31
bl ← (RB)32:63
if (ah = bh) then ch ← 1
else ch ← 0
if (al = bl) then cl ← 1
else cl ← 0
CR4×BF:4×BF+3 ← ch || cl || (ch | cl) || (ch & cl)
```

Each element of register RA is compared against the corresponding element of register RB. The results of the comparisons are placed into CR field BF. If $RA_{0:31}$ is equal to $RB_{0:31}$, bit 0 of CR field BF is set to 1, otherwise it is set to 0. If $RA_{32:63}$ is equal to $RB_{32:63}$, bit 1 of CR field BF is set to 1, otherwise it is set to 0. Bit 2 of CR field BF is set to the OR of both result bits and Bit 3 of CR field BF is set to the AND of both result bits. Comparison ignores the sign of 0 (+0 = -0). The comparison proceeds after treating NaNs, Infinities, and Denorms as normalized numbers, using their values of '*e*' and '*f*' directly.

No exceptions are taken during the execution of ***evfststeq***.

Special Registers Altered:
CR field BF

Programming Note

In an implementation, the execution of ***evfststeq*** is likely to be faster than the execution of ***evfscmpeq***; however, if strict IEEE 754 compliance is required, the program should use ***evfscmpeq***.

Vector Convert Floating-Point Single-Precision from Signed Integer EVX-form

evfscfsi RT,RB

4	RT	///	RB	657
0	6	11	16	21 31

$RT_{0:31} \leftarrow$ CnvtI32ToFP32((RB)$_{0:31}$, S, HI, I)
$RT_{32:63} \leftarrow$ CnvtI32ToFP32((RB)$_{32:63}$, S, LO, I)

Each signed integer element of register RB is converted to the nearest single-precision floating-point value using the current rounding mode and the results are placed into the corresponding element of register RT.

Special Registers Altered:
FGH FXH FG FX FINXS

Vector Convert Floating-Point Single-Precision from Unsigned Integer EVX-form

evfscfui RT,RB

4	RT	///	RB	656
0	6	11	16	21 31

$RT_{0:31} \leftarrow$ CnvtI32ToFP32((RB)$_{0:31}$, U, HI, I)
$RT_{32:63} \leftarrow$ CnvtI32ToFP32((RB)$_{32:63}$, U, LO, I)

Each unsigned integer element of register RB is converted to the nearest single-precision floating-point value using the current rounding mode and the results are placed into the corresponding elements of register RT.

Special Registers Altered:
FGH FXH FG FX FINXS

Vector Convert Floating-Point Single-Precision from Signed Fraction EVX-form

evfscfsf RT,RB

4	RT	///	RB	659
0	6	11	16	21 31

$RT_{0:31} \leftarrow$ CnvtI32ToFP32((RB)$_{0:31}$, S, HI, F)
$RT_{32:63} \leftarrow$ CnvtI32ToFP32((RB)$_{32:63}$, S, LO, F)

Each signed fractional element of register RB is converted to a single-precision floating-point value using the current rounding mode and the results are placed into the corresponding elements of register RT.

Special Registers Altered:
FGH FXH FG FX FINXS

Vector Convert Floating-Point Single-Precision from Unsigned Fraction EVX-form

evfscfuf RT,RB

4	RT	///	RB	658
0	6	11	16	21 31

$RT_{0:31} \leftarrow$ CnvtI32ToFP32((RB)$_{0:31}$, U, HI, F)
$RT_{32:63} \leftarrow$ CnvtI32ToFP32((RB)$_{32:63}$, U, LO, F)

Each unsigned fractional element of register RB is converted to a single-precision floating-point value using the current rounding mode and the results are placed into the corresponding elements of register RT.

Special Registers Altered:
FGH FXH FG FX FINXS

Vector Convert Floating-Point Single-Precision to Signed Integer EVX-form

evfsctsi RT,RB

4	RT	///	RB	661
0	6	11	16	21 31

$RT_{0:31}$ ← CnvtFP32ToI32Sat($(RB)_{0:31}$, S, HI, RND, I)
$RT_{32:63}$ ← CnvtFP32ToI32Sat($(RB)_{32:63}$, S, LO, RND, I)

Each single-precision floating-point element in register RB is converted to a signed integer using the current rounding mode and the result is saturated if it cannot be represented in a 32-bit integer. NaNs are converted as though they were zero.

Special Registers Altered:
FINV FINVH FINVS
FGH FXH FG FX FINXS

Vector Convert Floating-Point Single-Precision to Unsigned Integer EVX-form

evfsctui RT,RB

4	RT	///	RB	660
0	6	11	16	21 31

$RT_{0:31}$ ← CnvtFP32ToI32Sat($(RB)_{0:31}$, U, HI, RND, I)
$RT_{32:63}$ ← CnvtFP32ToI32Sat($(RB)_{32:63}$,U, LO, RND, I)

Each single-precision floating-point element in register RB is converted to an unsigned integer using the current rounding mode and the result is saturated if it cannot be represented in a 32-bit integer. NaNs are converted as though they were zero.

Special Registers Altered:
FINV FINVH FINVS
FGH FXH FG FX FINXS

Vector Convert Floating-Point Single-Precision to Signed Integer with Round toward Zero EVX-form

evfsctsiz RT,RB

4	RT	///	RB	666
0	6	11	16	21 31

$RT_{0:31}$ ← CnvtFP32ToI32Sat($(RB)_{0:31}$, S, HI, ZER, I)
$RT_{32:63}$ ← CnvtFP32ToI32Sat($(RB)_{32:63}$, S, LO, ZER, I)

Each single-precision floating-point element in register RB is converted to a signed integer using the rounding mode Round toward Zero and the result is saturated if it cannot be represented in a 32-bit integer. NaNs are converted as though they were zero.

Special Registers Altered:
FINV FINVH FINVS
FGH FXH FG FX FINXS

Vector Convert Floating-Point Single-Precision to Unsigned Integer with Round toward Zero EVX-form

evfsctuiz RT,RB

4	RT	///	RB	664
0	6	11	16	21 31

$RT_{0:31}$ ← CnvtFP32ToI32Sat($(RB)_{0:31}$, U, HI, ZER, I)
$RT_{32:63}$ ← CnvtFP32ToI32Sat($(RB)_{32:63}$, U, LO, ZER, I)

Each single-precision floating-point element in register RB is converted to an unsigned integer using the rounding mode Round toward Zero and the result is saturated if it cannot be represented in a 32-bit integer. NaNs are converted as though they were zero.

Special Registers Altered:
FINV FINVH FINVS
FGH FXH FG FX FINXS

Vector Convert Floating-Point Single-Precision to Signed Fraction EVX-form

evfsctsf RT,RB

4	RT	///	RB	663
0	6	11	16	21 31

```
RT0:31 ← CnvtFP32ToI32Sat((RB)0:31, S, HI, RND ,F)
RT32:63 ← CnvtFP32ToI32Sat((RB)32:63, S, LO, RND, F)
```

Each single-precision floating-point element in register RB is converted to a signed fraction using the current rounding mode and the result is saturated if it cannot be represented in a 32-bit signed fraction. NaNs are converted as though they were zero.

Special Registers Altered:
FINV FINVH FINVS
FGH FXH FG FX FINXS

Vector Convert Floating-Point Single-Precision to Unsigned Fraction EVX-form

evfsctuf RT,RB

4	RT	///	RB	662
0	6	11	16	21 31

```
RT0:31 ← CnvtFP32ToI32Sat((RB)0:31, U, HI, RND, F)
RT32:63 ← CnvtFP32ToI32Sat((RB)32:63, U, LO, RND, F)
```

Each single-precision floating-point element in register RB is converted to an unsigned fraction using the current rounding mode and the result is saturated if it cannot be represented in a 32-bit fraction. NaNs are converted as though they were zero.

Special Registers Altered:
FINV FINVH FINVS
FGH FXH FG FX FINXS

7.3.3 SPE.Embedded Float Scalar Single Instructions [Category: SPE.Embedded Float Scalar Single]

Floating-Point Single-Precision Absolute Value EVX-form

efsabs RT,RA

4	RT	RA	///	708
0	6	11	16	21 31

$RT_{32:63} \leftarrow$ 0b0 || $(RA)_{33:63}$

The sign bit of the low element of register RA is set to 0 and the result is placed into the low element of register RT.

Regardless of the value of register RA, no exceptions are taken during the execution of this instruction.

Special Registers Altered:
None

Floating-Point Single-Precision Negate EVX-form

efsneg RT,RA

4	RT	RA	///	710
0	6	11	16	21 31

$RT_{32:63} \leftarrow \neg(RA)_{32}$ || $(RA)_{33:63}$

The sign bit of the low element of register RA is complemented and the result is placed into the low element of register RT.

Regardless of the value of register RA, no exceptions are taken during the execution of this instruction.

Special Registers Altered:
None

Floating-Point Single-Precision Negative Absolute Value EVX-form

efsnabs RT,RA

4	RT	RA	///	709
0	6	11	16	21 31

$RT_{32:63} \leftarrow$ 0b1 || $(RA)_{33:63}$

The sign bit of the low element of register RA is set to 1 and the result is placed into the low element of register RT.

Regardless of the value of register RA, no exceptions are taken during the execution of this instruction.

Special Registers Altered:
None

Floating-Point Single-Precision Add EVX-form

efsadd RT,RA,RB

4	RT	RA	RB	704
0	6	11	16	21 31

$RT_{32:63} \leftarrow (RA)_{32:63} +_{sp} (RB)_{32:63}$

The low element of register RA is added to the low element of register RB and the result is stored in the low element of register RT.

If an underflow occurs, +0 (for rounding modes RN, RZ, RP) or -0 (for rounding mode RM) is stored in register RT.

Special Registers Altered:

FINV FINVS
FOVF FOVFS
FUNF FUNFS
FG FX FINXS

Floating-Point Single-Precision Subtract EVX-form

efssub RT,RA,RB

4	RT	RA	RB	705
0	6	11	16	21 31

$RT_{32:63} \leftarrow (RA)_{32:63} -_{sp} (RB)_{32:63}$

The low element of register RB is subtracted from the low element of register RA and the result is stored in the low element of register RT.

If an underflow occurs, +0 (for rounding modes RN, RZ, RP) or -0 (for rounding mode RM) is stored in register RT.

Special Registers Altered:

FINV FINVS
FOVF FOVFS
FUNF FUNFS
FG FX FINXS

Floating-Point Single-Precision Multiply EVX-form

efsmul RT,RA,RB

4	RT	RA	RB	712
0	6	11	16	21 31

$RT_{32:63} \leftarrow (RA)_{32:63} \times_{sp} (RB)_{32:63}$

The low element of register RA is multiplied by the low element of register RB and the result is stored in the low element of register RT.

Special Registers Altered:

FINV FINVS
FOVF FOVFS
FUNF FUNFS
FG FX FINXS

Floating-Point Single-Precision Divide EVX-form

efsdiv RT,RA,RB

4	RT	RA	RB	713
0	6	11	16	21 31

$RT_{32:63} \leftarrow (RA)_{32:63} \div_{sp} (RB)_{32:63}$

The low element of register RA is divided by the low element of register RB and the result is stored in the low element of register RT.

Special Registers Altered:

FINV FINVS
FG FX FINXS
FDBZ FDBZS
FOVF FOVFS
FUNF FUNFS

Floating-Point Single-Precision Compare Greater Than EVX-form

efscmpgt BF,RA,RB

4	BF	//	RA	RB	716
0	6	9	11	16	21 31

```
al ← (RA)32:63
bl ← (RB)32:63
if (al > bl) then cl ← 1
else cl ← 0
CR4×BF:4×BF+3 ← undefined || cl || undefined || undefined
```

The low element of register RA is compared against the low element of register RB. The results of the comparisons are placed into CR field BF. If $RA_{32:63}$ is greater than $RB_{32:63}$, bit 1 of CR field BF is set to 1, otherwise it is set to 0. Bits 0, 2, and 3 of CR field BF are undefined. Comparison ignores the sign of 0 (+0 = -0).

If an Input Error occurs and default results are generated, NaNs, Infinities, and Denorms are treated as normalized numbers, using their values of '*e*' and '*f*' directly.

Special Registers Altered:

FINV FINVS
FG FX
CR field BF

Floating-Point Single-Precision Compare Less Than EVX-form

efscmplt BF,RA,RB

4	BF	//	RA	RB	717
0	6	9	11	16	21 31

```
al ← (RA)32:63
bl ← (RB)32:63
if (al < bl) then cl ← 1
else cl ← 0
CR4×BF:4×BF+3 ← undefined || cl || undefined || undefined
```

The low element of register RA is compared against the low element of register RB. If $RA_{32:63}$ is less than $RB_{32:63}$, bit 1 of CR field BF is set to 1, otherwise it is set to 0. Bits 0, 2, and 3 of CR field BF are undefined. Comparison ignores the sign of 0 (+0 = -0).

If an Input Error occurs and default results are generated, NaNs, Infinities, and Denorms are treated as normalized numbers, using their values of '*e*' and '*f*' directly.

Special Registers Altered:

FINV FINVS
FG FX
CR field BF

Floating-Point Single-Precision Compare Equal EVX-form

efscmpeq BF,RA,RB

4	BF	//	RA	RB	718
0	6	9	11	16	21 31

```
al ← (RA)32:63
bl ← (RB)32:63
if (al = bl) then cl ← 1
else cl ← 0
CR4×BF:4×BF+3 ← undefined || cl || undefined || undefined
```

The low element of register RA is compared against the low element of register RB. If $RA_{32:63}$ is equal to $RB_{32:63}$, bit 1 of CR field BF is set to 1, otherwise it is set to 0. Bits 0, 2, and 3 of CR field BF are undefined. Comparison ignores the sign of 0 (+0 = -0).

If an Input Error occurs and default results are generated, NaNs, Infinities, and Denorms are treated as normalized numbers, using their values of ‘*e*’ and ‘*f*’ directly.

Special Registers Altered:

FINV FINVS
FG FX
CR field BF

Floating-Point Single-Precision Test Greater Than EVX-form

efststgt BF,RA,RB

4	BF	//	RA	RB	732
0	6	9	11	16	21 31

```
al ← (RA)32:63
bl ← (RB)32:63
if (al > bl) then cl ← 1
else cl ← 0
CR4×BF:4×BF+3 ← undefined || cl || undefined || undefined
```

The low element of register RA is compared against the low element of register RB. If $RA_{32:63}$ is greater than $RB_{32:63}$, bit 1 of CR field BF is set to 1, otherwise it is set to 0. Bits 0, 2, and 3 of CR field BF are undefined. Comparison ignores the sign of 0 (+0 = -0). The comparison proceeds after treating NaNs, Infinities, and Denorms as normalized numbers, using their values of ‘*e*’ and ‘*f*’ directly.

No exceptions are generated during the execution of ***efststgt***.

Special Registers Altered:

CR field BF

Programming Note

In an implementation, the execution of ***efststgt*** is likely to be faster than the execution of ***efscmpgt***; however, if strict IEEE 754 compliance is required, the program should use ***efscmpgt***.

Floating-Point Single-Precision Test Less Than EVX-form

efststlt BF,RA,RB

4	BF	//	RA	RB	733
0	6	9	11	16	21 31

```
al ← (RA)32:63
bl ← (RB)32:63
if (al < bl) then cl ← 1
else cl ← 0
CR4×BF:4×BF+3 ← undefined || cl || undefined || undefined
```

The low element of register RA is compared against the low element of register RB. If $RA_{32:63}$ is less than $RB_{32:63}$, bit 1 of CR field BF is set to 1, otherwise it is set to 0. Bits 0, 2, and 3 of CR field BF are undefined. Comparison ignores the sign of 0 (+0 = -0). The comparison proceeds after treating NaNs, Infinities, and Denorms as normalized numbers, using their values of '*e*' and '*f*' directly.

No exceptions are generated during the execution of ***efststlt***.

Special Registers Altered:
CR field BF

Programming Note

In an implementation, the execution of ***efststlt*** is likely to be faster than the execution of ***efscmplt***; however, if strict IEEE 754 compliance is required, the program should use ***efscmplt***.

Floating-Point Single-Precision Test Equal EVX-form

efststeq BF,RA,RB

4	BF	//	RA	RB	734
0	6	9	11	16	21 31

```
al ← (RA)32:63
bl ← (RB)32:63
if (al = bl) then cl ← 1
else cl ← 0
CR4×BF:4×BF+3 ← undefined || cl || undefined || undefined
```

The low element of register RA is compared against the low element of register RB. If $RA_{32:63}$ is equal to $RB_{32:63}$, bit 1 of CR field BF is set to 1, otherwise it is set to 0. Bits 0, 2, and 3 of CR field BF are undefined. Comparison ignores the sign of 0 (+0 = -0). The comparison proceeds after treating NaNs, Infinities, and Denorms as normalized numbers, using their values of '*e*' and '*f*' directly.

No exceptions are generated during the execution of ***efststeq***.

Special Registers Altered:
CR field BF

Programming Note

In an implementation, the execution of ***efststeq*** is likely to be faster than the execution of ***efscmpeq***; however, if strict IEEE 754 compliance is required, the program should use ***efscmpeq***.

Convert Floating-Point Single-Precision from Signed Integer *EVX-form*

efscfsi RT,RB

4	RT	///	RB	721
0	6	11	16	21 31

$RT_{32:63} \leftarrow$ CnvtI32ToFP32($(RB)_{32:63}$, S, LO, I)

The signed integer low element in register RB is converted to a single-precision floating-point value using the current rounding mode and the result is placed into the low element of register RT.

Special Registers Altered:
FINXS FG FX

Convert Floating-Point Single-Precision from Unsigned Integer *EVX-form*

efscfui RT,RB

4	RT	///	RB	720
0	6	11	16	21 31

$RT_{32:63} \leftarrow$ CnvtI32ToFP32($(RB)_{32:63}$, U, LO, I)

The unsigned integer low element in register RB is converted to a single-precision floating-point value using the current rounding mode and the result is placed into the low element of register RT.

Special Registers Altered:
FINXS FG FX

Convert Floating-Point Single-Precision from Signed Fraction *EVX-form*

efscfsf RT,RB

4	RT	///	RB	723
0	6	11	16	21 31

$RT_{32:63} \leftarrow$ CnvtI32ToFP32($(RB)_{32:63}$, S, LO, F)

The signed fractional low element in register RB is converted to a single-precision floating-point value using the current rounding mode and the result is placed into the low element of register RT.

Special Registers Altered:
FINXS FG FX

Convert Floating-Point Single-Precision from Unsigned Fraction *EVX-form*

efscfuf RT,RB

4	RT	///	RB	722
0	6	11	16	21 31

$RT_{32:63} \leftarrow$ CnvtI32ToFP32($(RB)_{32:63}$, U, LO, F)

The unsigned fractional low element in register RB is converted to a single-precision floating-point value using the current rounding mode and the result is placed into the low element of register RT.

Special Registers Altered:
FINXS FG FX

Convert Floating-Point Single-Precision to Signed Integer *EVX-form*

efsctsi RT,RB

4	RT	///	RB	725
0	6	11	16	21 31

$RT_{32:63} \leftarrow$ CnvtFP32ToI32Sat($(RB)_{32:63}$, S, LO, RND, I)

The single-precision floating-point low element in register RB is converted to a signed integer using the current rounding mode and the result is saturated if it cannot be represented in a 32-bit integer. NaNs are converted as though they were zero.

Special Registers Altered:
FINV FINVS
FINXS FG FX

Convert Floating-Point Single-Precision to Unsigned Integer *EVX-form*

efsctui RT,RB

4	RT	///	RB	724
0	6	11	16	21 31

$RT_{32:63} \leftarrow$ CnvtFP32ToI32Sat($(RB)_{32:63}$, U, LO, RND, I)

The single-precision floating-point low element in register RB is converted to an unsigned integer using the current rounding mode and the result is saturated if it cannot be represented in a 32-bit integer. NaNs are converted as though they were zero.

Special Registers Altered:
FINV FINVS
FINXS FG FX

Convert Floating-Point Single-Precision to Signed Integer with Round toward Zero EVX-form

efsctsiz RT,RB

4	RT	///	RB	730
0	6	11	16	21 31

$RT_{32:63}$ ← CnvtFP32ToI32Sat($(RB)_{32:63}$, S, LO, ZER, I)

The single-precision floating-point low element in register RB is converted to a signed integer using the rounding mode Round toward Zero and the result is saturated if it cannot be represented in a 32-bit integer. NaNs are converted as though they were zero.

Special Registers Altered:
FINV FINVS
FINXS FG FX

Convert Floating-Point Single-Precision to Unsigned Integer with Round toward Zero EVX-form

efsctuiz RT,RB

4	RT	///	RB	728
0	6	11	16	21 31

$RT_{32:63}$ ← CnvtFP32ToI32Sat($(RB)_{32:63}$, U, LO, ZER, I)

The single-precision floating-point low element in register RB is converted to an unsigned integer using the rounding mode Round toward Zero and the result is saturated if it cannot be represented in a 32-bit integer. NaNs are converted as though they were zero.

Special Registers Altered:
FINV FINVS
FINXS FG FX

Convert Floating-Point Single-Precision to Signed Fraction EVX-form

efsctsf RT,RB

4	RT	///	RB	727
0	6	11	16	21 31

$RT_{32:63}$ ← CnvtFP32ToI32Sat($(RB)_{32:63}$, S, LO, RND, F)

The single-precision floating-point low element in register RB is converted to a signed fraction using the current rounding mode and the result is saturated if it cannot be represented in a 32-bit fraction. NaNs are converted as though they were zero.

Special Registers Altered:
FINV FINVS
FINXS FG FX

Convert Floating-Point Single-Precision to Unsigned Fraction EVX-form

efsctuf RT,RB

4	RT	///	RB	726
0	6	11	16	21 31

$RT_{32:63}$ ← CnvtFP32ToI32Sat($(RB)_{32:63}$, U, LO, RND, F)

The single-precision floating-point low element in register RB is converted to an unsigned fraction using the current rounding mode and the result is saturated if it cannot be represented in a 32-bit unsigned fraction. NaNs are converted as though they were zero.

Special Registers Altered:
FINV FINVS
FINXS FG FX

7.3.4 SPE.Embedded Float Scalar Double Instructions [Category: SPE.Embedded Float Scalar Double]

Floating-Point Double-Precision Absolute Value EVX-form

efdabs RT,RA

4	RT	RA	///	740
0	6	11	16	21 31

$RT_{0:63} \leftarrow$ 0b0 || $(RA)_{1:63}$

The sign bit of register RA is set to 0 and the result is placed in register RT.

Regardless of the value of register RA, no exceptions are taken during the execution of this instruction.

Special Registers Altered:
None

Floating-Point Double-Precision Negative Absolute Value EVX-form

efdnabs RT,RA

4	RT	RA	///	741
0	6	11	16	21 31

$RT_{0:63} \leftarrow$ 0b1 || $(RA)_{1:63}$

The sign bit of register RA is set to 1 and the result is placed in register RT.

Regardless of the value of register RA, no exceptions are taken during the execution of this instruction.

Special Registers Altered:
None

Floating-Point Double-Precision Negate EVX-form

efdneg RT,RA

4	RT	RA	///	742
0	6	11	16	21 31

$RT_{0:63} \leftarrow \neg(RA)_0$ || $(RA)_{1:63}$

The sign bit of register RA is complemented and the result is placed in register RT.

Regardless of the value of register RA, no exceptions are taken during the execution of this instruction.

Special Registers Altered:
None

Floating-Point Double-Precision Add EVX-form

efdadd RT,RA,RB

4	RT	RA	RB	736
0	6	11	16	21 31

$RT_{0:63} \leftarrow (RA)_{0:63} +_{dp} (RB)_{0:63}$

RA is added to RB and the result is stored in register RT.

If an underflow occurs, +0 (for rounding modes RN, RZ, RP) or -0 (for rounding mode RM) is stored in register RT.

Special Registers Altered:

FINV FINVS
FOVF FOVFS
FUNF FUNFS
FG FX FINXS

Floating-Point Double-Precision Subtract EVX-form

efdsub RT,RA,RB

4	RT	RA	RB	737
0	6	11	16	21 31

$RT_{0:63} \leftarrow (RA)_{0:63} -_{dp} (RB)_{0:63}$

RB is subtracted from RA and the result is stored in register RT.

If an underflow occurs, +0 (for rounding modes RN, RZ, RP) or -0 (for rounding mode RM) is stored in register RT.

Special Registers Altered:

FINV FINVS
FOVF FOVFS
FUNF FUNFS
FG FX FINXS

Floating-Point Double-Precision Multiply EVX-form

efdmul RT,RA,RB

4	RT	RA	RB	744
0	6	11	16	21 31

$RT_{0:63} \leftarrow (RA)_{0:63} \times_{dp} (RB)_{0:63}$

RA is multiplied by RB and the result is stored in register RT.

Special Registers Altered:

FINV FINVS
FOVF FOVFS
FUNF FUNFS
FG FX FINXS

Floating-Point Double-Precision Divide EVX-form

efddiv RT,RA,RB

4	RT	RA	RB	745
0	6	11	16	21 31

$RT_{0:63} \leftarrow (RA)_{0:63} \div_{dp} (RB)_{0:63}$

RA is divided by RB and the result is stored in register RT.

Special Registers Altered:

FINV FINVS
FG FX FINXS
FDBZ FDBZS
FOVF FOVFS
FUNF FUNFS

Floating-Point Double-Precision Compare Greater Than EVX-form

efdcmpgt BF,RA,RB

4	BF	//	RA	RB	748
0	6	9	11	16	21 31

```
al ← (RA)0:63
bl ← (RB)0:63
if (al > bl) then cl ← 1
else cl ← 0
CR4×BF:4×BF+3 ← undefined || cl || undefined || undefined
```

RA is compared against RB. If RA is greater than RB, bit 1 of CR field BF is set to 1, otherwise it is set to 0. Bits 0, 2, and 3 of CR field BF are undefined. Comparison ignores the sign of 0 (+0 = -0).

If an input error occurs and default results are generated, NaNs, Infinities, and Denorms are treated as normalized numbers, using their values of '*e*' and '*f*' directly.

Special Registers Altered:
FINV FINVS
FG FX
CR field BF

Floating-Point Double-Precision Compare Less Than EVX-form

efdcmplt BF,RA,RB

4	BF	//	RA	RB	749
0	6	9	11	16	21 31

```
al ← (RA)0:63
bl ← (RB)0:63
if (al < bl) then cl ← 1
else cl ← 0
CR4×BF:4×BF+3 ← undefined || cl || undefined || undefined
```

RA is compared against RB. If RA is less than RB, bit 1 of CR field BF is set to 1, otherwise it is set to 0. Bits 0, 2, and 3 of CR field BF are undefined. Comparison ignores the sign of 0 (+0 = -0).

If an input error occurs and default results are generated, NaNs, Infinities, and Denorms are treated as normalized numbers, using their values of '*e*' and '*f*' directly.

Special Registers Altered:
FINV FINVS
FG FX
CR field BF

Floating-Point Double-Precision Compare Equal EVX-form

efdcmpeq BF,RA,RB

4	BF	//	RA	RB	750
0	6	9	11	16	21 31

```
al ← (RA)0:63
bl ← (RB)0:63
if (al = bl) then cl ← 1
else cl ← 0
CR4×BF:4×BF+3 ← undefined || cl || undefined || undefined
```

RA is compared against RB. If RA is equal to RB, bit 1 of CR field BF is set to 1, otherwise it is set to 0. Bits 0, 2, and 3 of CR field BF are undefined. Comparison ignores the sign of 0 (+0 = -0).

If an input error occurs and default results are generated, NaNs, Infinities, and Denorms are treated as normalized numbers, using their values of '*e*' and '*f*' directly.

Special Registers Altered:
FINV FINVS
FG FX
CR field BF

Floating-Point Double-Precision Test Greater Than EVX-form

efdtstgt BF,RA,RB

4	BF	//	RA	RB	764
0	6	9	11	16	21 31

```
al ← (RA)0:63
bl ← (RB)0:63
if (al > bl) then cl ← 1
else cl ← 0
CR4×BF:4×BF+3 ← undefined || cl || undefined || undefined
```

RA is compared against RB. If RA is greater than RB, bit 1 of CR field BF is set to 1, otherwise it is set to 0. Bits 0, 2, and 3 of CR field BF are undefined. Comparison ignores the sign of 0 (+0 = -0). The comparison proceeds after treating NaNs, Infinities, and Denorms as normalized numbers, using their values of '*e*' and '*f*' directly.

No exceptions are generated during the execution of ***efdtstgt***.

Special Registers Altered:
CR field BF

Programming Note

In an implementation, the execution of ***efdtstgt*** is likely to be faster than the execution of ***efdcmpgt***; however, if strict IEEE 754 compliance is required, the program should use ***efdcmpgt***.

Floating-Point Double-Precision Test Less Than EVX-form

efdtstlt BF,RA,RB

4	BF	//	RA	RB	765
0	6	9	11	16	21 31

```
al ← (RA)0:63
bl ← (RB)0:63
if (al < bl) then cl ← 1
else cl ← 0
CR4×BF:4×BF+3 ← undefined || cl || undefined || undefined
```

RA is compared against RB. If RA is less than RB, bit 1 of CR field BF is set to 1, otherwise it is set to 0. Bits 0, 2, and 3 of CR field BF are undefined. Comparison ignores the sign of 0 (+0 = -0). The comparison proceeds after treating NaNs, Infinities, and Denorms as normalized numbers, using their values of '*e*' and '*f*' directly.

No exceptions are generated during the execution of ***efdtstlt***.

Special Registers Altered:
CR field BF

Programming Note

In an implementation, the execution of ***efdtstlt*** is likely to be faster than the execution of ***efdcmplt***; however, if strict IEEE 754 compliance is required, the program should use ***efdcmplt***.

Floating-Point Double-Precision Test Equal EVX-form

efdtsteq BF,RA,RB

4	BF	//	RA	RB	766
0	6	9	11	16	21 31

```
al ← (RA)0:63
bl ← (RB)0:63
if (al = bl) then cl ← 1
else cl ← 0
CR4×BF:4×BF+3 ← undefined || cl || undefined || undefined
```

RA is compared against RB. If RA is equal to RB, bit 1 of CR field BF is set to 1, otherwise it is set to 0. Bits 0, 2, and 3 of CR field BF are undefined. Comparison ignores the sign of 0 (+0 = -0). The comparison proceeds after treating NaNs, Infinities, and Denorms as normalized numbers, using their values of '*e*' and '*f*' directly.

No exceptions are generated during the execution of ***efdtsteq***.

Special Registers Altered:
CR field BF

Programming Note

In an implementation, the execution of ***efdtsteq*** is likely to be faster than the execution of ***efdcmpeq***; however, if strict IEEE 754 compliance is required, the program should use ***efdcmpeq***.

Convert Floating-Point Double-Precision from Signed Integer EVX-form

efdcfsi RT,RB

4	RT	///	RB	753
0	6	11	16	21 31

```
RT0:63 ← CnvtI32ToFP64((RB)32:63, S, I)
```

The signed integer low element in register RB is converted to a double-precision floating-point value using the current rounding mode and the result is placed in register RT.

Special Registers Altered:
None

Convert Floating-Point Double-Precision from Unsigned Integer EVX-form

efdcfui RT,RB

4	RT	///	RB	752
0	6	11	16	21 31

```
RT0:63 ← CnvtI32ToFP64((RB)32:63, U, I)
```

The unsigned integer low element in register RB is converted to a double-precision floating-point value using the current rounding mode and the result is placed in register RT.

Special Registers Altered:
None

Convert Floating-Point Double-Precision from Signed Integer Doubleword EVX-form

efdcfsid RT,RB

4	RT	///	RB	739
0	6	11	16	21 31

$RT_{0:63} \leftarrow$ CnvtI64ToFP64($(RB)_{0:63}$, S)

The signed integer doubleword in register RB is converted to a double-precision floating-point value using the current rounding mode and the result is placed in register RT.

Corequisite Categories:
64-Bit

Special Registers Altered:
FINXS FG FX

Convert Floating-Point Double-Precision from Signed Fraction EVX-form

efdcfsf RT,RB

4	RT	///	RB	755
0	6	11	16	21 31

$RT_{0:63} \leftarrow$ CnvtI32ToFP64($(RB)_{32:63}$, S, F)

The signed fractional low element in register RB is converted to a double-precision floating-point value using the current rounding mode and the result is placed in register RT.

Special Registers Altered:
None

Convert Floating-Point Double-Precision from Unsigned Fraction EVX-form

efdcfuf RT,RB

4	RT	///	RB	754
0	6	11	16	21 31

$RT_{0:63} \leftarrow$ CnvtI32ToFP64($(RB)_{32:63}$, U, F)

The unsigned fractional low element in register RB is converted to a double-precision floating-point value using the current rounding mode and the result is placed in register RT.

Special Registers Altered:
None

Convert Floating-Point Double-Precision from Unsigned Integer Doubleword EVX-form

efdcfuid RT,RB

4	RT	///	RB	738
0	6	11	16	21 31

$RT_{0:63} \leftarrow$ CnvtI64ToFP64($(RB)_{0:63}$, U)

The unsigned integer doubleword in register RB is converted to a double-precision floating-point value using the current rounding mode and the result is placed in register RT.

Corequisite Categories:
64-Bit

Special Registers Altered:
FINXS FG FX

Convert Floating-Point Double-Precision to Signed Integer EVX-form

efdctsi RT,RB

4	RT	///	RB	757
0	6	11	16	21 31

$RT_{32:63} \leftarrow$ CnvtFP64ToI32Sat($(RB)_{0:63}$, S, RND, I)

The double-precision floating-point value in register RB is converted to a signed integer using the current rounding mode and the result is saturated if it cannot be represented in a 32-bit integer. NaNs are converted as though they were zero.

Special Registers Altered:
FINV FINVS
FINXS FG FX

Convert Floating-Point Double-Precision to Unsigned Integer EVX-form

efdctui RT,RB

4	RT	///	RB	756
0	6	11	16	21 31

$RT_{32:63} \leftarrow$ CnvtFP64ToI32Sat($(RB)_{0:63}$, U, RND, I)

The double-precision floating-point value in register RB is converted to an unsigned integer using the current rounding mode and the result is saturated if it cannot be represented in a 32-bit integer. NaNs are converted as though they were zero.

Special Registers Altered:
FINV FINVS
FINXS FG FX

Convert Floating-Point Double-Precision to Signed Integer Doubleword with Round toward Zero EVX-form

efdctsidz RT,RB

4	RT	///	RB	747
0	6	11	16	21 31

$RT_{0:63} \leftarrow$ CnvtFP64ToI64Sat($(RB)_{0:63}$, S, ZER)

The double-precision floating-point value in register RB is converted to a signed integer doubleword using the rounding mode Round toward Zero and the result is saturated if it cannot be represented in a 64-bit integer. NaNs are converted as though they were zero.

Corequisite Categories:
64-Bit

Special Registers Altered:
FINV FINVS
FINXS FG FX

Convert Floating-Point Double-Precision to Unsigned Integer Doubleword with Round toward Zero EVX-form

efdctuidz RT,RB

4	RT	///	RB	746
0	6	11	16	21 31

$RT_{0:63} \leftarrow$ CnvtFP64ToI64Sat($(RB)_{0:63}$, U, ZER)

The double-precision floating-point value in register RB is converted to an unsigned integer doubleword using the rounding mode Round toward Zero and the result is saturated if it cannot be represented in a 64-bit integer. NaNs are converted as though they were zero.

Corequisite Categories:
64-Bit

Special Registers Altered:
FINV FINVS
FINXS FG FX

Convert Floating-Point Double-Precision to Signed Integer with Round toward Zero EVX-form

efdctsiz RT,RB

4	RT	///	RB	762
0	6	11	16	21 31

$RT_{32:63} \leftarrow$ CnvtFP64ToI32Sat((RB)$_{0:63}$, S, ZER, I)

The double-precision floating-point value in register RB is converted to a signed integer using the rounding mode Round toward Zero and the result is saturated if it cannot be represented in a 32-bit integer. NaNs are converted as though they were zero.

Special Registers Altered:
FINV FINVS
FINXS FG FX

Convert Floating-Point Double-Precision to Signed Fraction EVX-form

efdctsf RT,RB

4	RT	///	RB	759
0	6	11	16	21 31

$RT_{32:63} \leftarrow$ CnvtFP64ToI32Sat((RB)$_{0:63}$, S, RND, F)

The double-precision floating-point value in register RB is converted to a signed fraction using the current rounding mode and the result is saturated if it cannot be represented in a 32-bit fraction. NaNs are converted as though they were zero.

Special Registers Altered:
FINV FINVS
FINXS FG FX

Convert Floating-Point Double-Precision to Unsigned Fraction EVX-form

efdctuf RT,RB

4	RT	///	RB	758
0	6	11	16	21 31

$RT_{32:63} \leftarrow$ CnvtFP64ToI32Sat((RB)$_{0:63}$, U, RND, F)

The double-precision floating-point value in register RB is converted to an unsigned fraction using the current rounding mode and the result is saturated if it cannot be represented in a 32-bit unsigned fraction. NaNs are converted as though they were zero.

Special Registers Altered:
FINV FINVS
FINXS FG FX

Convert Floating-Point Double-Precision to Unsigned Integer with Round toward Zero EVX-form

efdctuiz RT,RB

4	RT	///	RB	760
0	6	11	16	21 31

$RT_{32:63} \leftarrow$ CnvtFP64ToI32Sat((RB)$_{0:63}$, U, ZER, I)

The double-precision floating-point value in register RB is converted to an unsigned integer using the rounding mode Round toward Zero and the result is saturated if it cannot be represented in a 32-bit integer. NaNs are converted as though they were zero.

Special Registers Altered:
FINV FINVS
FINXS FG FX

Floating-Point Double-Precision convert from Single-Precision EVX-form

efdcfs RT,RB

4	RT	///	RB	751
0	6	11	16	21 31

```
FP32format f;
FP64format result;
f ← (RB)_32:63
if (f_exp = 0) & (f_frac = 0)) then
    result ← f_sign || ^63 0
else if Isa32NaNorInfinity(f) | Isa32Denorm(f) then
    SPEFSCR_FINV ← 1
    result ← f_sign || 0b11111111110 || ^52 1
else if Isa32Denorm(f) then
    SPEFSCR_FINV ← 1
    result ← f_sign || ^63 0
else
    result_sign ← f_sign
    result_exp ← f_exp - 127 + 1023
    result_frac ← f_frac || ^29 0
RT_0:63 ← result
```

The single-precision floating-point value in the low element of register RB is converted to a double-precision floating-point value and the result is placed in register RT.

Corequisite Categories:
SPE.Embedded Float Scalar Single or
SPE.Embedded Float Vector

Special Registers Altered:
FINV FINVS
FG FX

Floating-Point Single-Precision Convert from Double-Precision EVX-form

efscfd RT,RB

4	RT	///	RB	719
0	6	11	16	21 31

```
FP64format f;
FP32format result;
f ← (RB)0:63
if (fexp = 0) & (ffrac = 0)) then
    result ← fsign || 310
else if Isa64NaNorInfinity(f) then
    SPEFSCRFINV ← 1
    result ← fsign || 0b11111110 || 231
else if Isa64Denorm(f) then
    SPEFSCRFINV ← 1
    result ← fsign || 310
else
    unbias ← fexp - 1023
    if unbias > 127 then
        result ← fsign || 0b11111110 || 231
        SPEFSCRFOVF ← 1
    else if unbias < -126 then
        result ← fsign || 0b00000001 || 230
        SPEFSCRFUNF ← 1
    else
        resultsign ← fsign
        resultexp ← unbias + 127
        resultfrac ← ffrac[0:22]
        guard ← ffrac[23]
        sticky ← (ffrac[24:51] ≠ 0)
        result ← Round32(result, LO, guard,
sticky)
        SPEFSCRFG ← guard
        SPEFSCRFX ← sticky
        if guard | sticky then
            SPEFSCRFINXS ← 1
RT32:63 ← result
```

The double-precision floating-point value in register RB is converted to a single-precision floating-point value using the current rounding mode and the result is placed into the low element of register RT.

Corequisite Categories:

SPE.Embedded Float Scalar Scalar

Special Registers Altered:

FINV FINVS
FOVF FOVFS
FUNF FUNFS
FG FX FINXS

7.4 Embedded Floating-Point Results Summary

The following tables summarize the results of various types of *Embedded Floating-Point* operations on various combinations of input operands. Flag settings are performed on appropriate element flags. For all the tables the following annotation and general rules apply:

- * denotes that this status flag is set based on the results of the calculation.
- *_Calc_* denotes that the result is updated with the results of the computation.
- *max* denotes the maximum normalized number with the sign set to the computation [sign(operand A) XOR sign(operand B)].
- *amax* denotes the maximum normalized number with the sign set to the sign of Operand A.
- *bmax* denotes the maximum normalized number with the sign set to the sign of Operand B.
- *pmax* denotes the maximum normalized positive number. The encoding for single-precision is: 0x7F7FFFFF. The encoding for double-precision is: 0x7FEFFFFF_FFFFFFFF.
- *nmax* denotes the maximum normalized negative number. The encoding for single-precision is: 0xFF7FFFFF. The encoding for double-precision is: 0xFFEFFFFF_FFFFFFFF.
- *pmin* denotes the minimum normalized positive number. The encoding for single-precision is: 0x00800000. The encoding for double-precision is: 0x00100000_00000000.
- *nmin* denotes the minimum normalized negative number. The encoding for single-precision is: 0x80800000. The encoding for double-precision is: 0x80100000_00000000.
- Calculations that overflow or underflow saturate. Overflow for operations that have a floating-point result force the result to *max*. Underflow for operations that have a floating-point result force the result to zero. Overflow for operations that have a signed integer result force the result to 0x7FFFFFFF (positive) or 0x80000000 (negative). Overflow for operations that have an unsigned integer result force the result to 0xFFFFFFFF (positive) or 0x00000000 (negative).
- *1* (superscript) denotes that the sign of the result is positive when the sign of Operand A and the sign of Operand B are different, for all rounding modes except round to -infinity, where the sign of the result is then negative.
- *2* (superscript) denotes that the sign of the result is positive when the sign of Operand A and the sign of Operand B are the same, for all rounding modes except round to -infinity, where the sign of the result is then negative.
- *3* (superscript) denotes that the sign for any multiply or divide is always the result of the operation [sign(Operand A) XOR sign(Operand B)].
- *4* (superscript) denotes that if an overflow is detected, the result may be saturated.

Table 3: Embedded Floating-Point Results Summary—Add, Sub, Mul, Div

Operation	Operand A	Operand B	Result	FINV	FOVF	FUNF	FDBZ	FINX
Add								
Add	∞	∞	amax	1	0	0	0	0
Add	∞	NaN	amax	1	0	0	0	0
Add	∞	denorm	amax	1	0	0	0	0
Add	∞	zero	amax	1	0	0	0	0
Add	∞	Norm	amax	1	0	0	0	0
Add	NaN	∞	amax	1	0	0	0	0
Add	NaN	NaN	amax	1	0	0	0	0
Add	NaN	denorm	amax	1	0	0	0	0
Add	NaN	zero	amax	1	0	0	0	0
Add	NaN	norm	amax	1	0	0	0	0
Add	denorm	∞	bmax	1	0	0	0	0
Add	denorm	NaN	bmax	1	0	0	0	0
Add	denorm	denorm	zero[1]	1	0	0	0	0
Add	denorm	zero	zero[1]	1	0	0	0	0
Add	denorm	norm	operand_b[4]	1	0	0	0	0
Add	zero	∞	bmax	1	0	0	0	0
Add	zero	NaN	bmax	1	0	0	0	0
Add	zero	denorm	zero[1]	1	0	0	0	0

Table 3: Embedded Floating-Point Results Summary—Add, Sub, Mul, Div (Continued)

Operation	Operand A	Operand B	Result	FINV	FOVF	FUNF	FDBZ	FINX
Add	zero	zero	zero[1]	0	0	0	0	0
Add	zero	norm	operand_b[4]	0	0	0	0	0
Add	norm	∞	bmax	1	0	0	0	0
Add	norm	NaN	bmax	1	0	0	0	0
Add	norm	denorm	operand_a[4]	1	0	0	0	0
Add	norm	zero	operand_a[4]	0	0	0	0	0
Add	norm	norm	_Calc_	0	*	*	0	*
Subtract								
Sub	∞	∞	amax	1	0	0	0	0
Sub	∞	NaN	amax	1	0	0	0	0
Sub	∞	denorm	amax	1	0	0	0	0
Sub	∞	zero	amax	1	0	0	0	0
Sub	∞	Norm	amax	1	0	0	0	0
Sub	NaN	∞	amax	1	0	0	0	0
Sub	NaN	NaN	amax	1	0	0	0	0
Sub	NaN	denorm	amax	1	0	0	0	0
Sub	NaN	zero	amax	1	0	0	0	0
Sub	NaN	norm	amax	1	0	0	0	0
Sub	denorm	∞	-bmax	1	0	0	0	0
Sub	denorm	NaN	-bmax	1	0	0	0	0
Sub	denorm	denorm	zero[2]	1	0	0	0	0
Sub	denorm	zero	zero[2]	1	0	0	0	0
Sub	denorm	norm	-operand_b[4]	1	0	0	0	0
Sub	zero	∞	-bmax	1	0	0	0	0
Sub	zero	NaN	-bmax	1	0	0	0	0
Sub	zero	denorm	zero[2]	1	0	0	0	0
Sub	zero	zero	zero[2]	0	0	0	0	0
Sub	zero	norm	-operand_b[4]	0	0	0	0	0
Sub	norm	∞	-bmax	1	0	0	0	0
Sub	norm	NaN	-bmax	1	0	0	0	0
Sub	norm	denorm	operand_a[4]	1	0	0	0	0
Sub	norm	zero	operand_a[4]	0	0	0	0	0
Sub	norm	norm	_Calc_	0	*	*	0	*
Multiply[3]								
Mul	∞	∞	max	1	0	0	0	0
Mul	∞	NaN	max	1	0	0	0	0
Mul	∞	denorm	zero	1	0	0	0	0
Mul	∞	zero	zero	1	0	0	0	0
Mul	∞	Norm	max	1	0	0	0	0
Mul	NaN	∞	max	1	0	0	0	0
Mul	NaN	NaN	max	1	0	0	0	0
Mul	NaN	denorm	zero	1	0	0	0	0
Mul	NaN	zero	zero	1	0	0	0	0
Mul	NaN	norm	max	1	0	0	0	0

Table 3: Embedded Floating-Point Results Summary—Add, Sub, Mul, Div (Continued)

Operation	Operand A	Operand B	Result	FINV	FOVF	FUNF	FDBZ	FINX
Mul	denorm	∞	zero	1	0	0	0	0
Mul	denorm	NaN	zero	1	0	0	0	0
Mul	denorm	denorm	zero	1	0	0	0	0
Mul	denorm	zero	zero	1	0	0	0	0
Mul	denorm	norm	zero	1	0	0	0	0
Mul	zero	∞	zero	1	0	0	0	0
Mul	zero	NaN	zero	1	0	0	0	0
Mul	zero	denorm	zero	1	0	0	0	0
Mul	zero	zero	zero	0	0	0	0	0
Mul	zero	norm	zero	0	0	0	0	0
Mul	norm	∞	max	1	0	0	0	0
Mul	norm	NaN	max	1	0	0	0	0
Mul	norm	denorm	zero	1	0	0	0	0
Mul	norm	zero	zero	0	0	0	0	0
Mul	norm	norm	_Calc_	0	*	*	0	*
Divide[3]								
Div	∞	∞	zero	1	0	0	0	0
Div	∞	NaN	zero	1	0	0	0	0
Div	∞	denorm	max	1	0	0	0	0
Div	∞	zero	max	1	0	0	0	0
Div	∞	Norm	max	1	0	0	0	0
Div	NaN	∞	zero	1	0	0	0	0
Div	NaN	NaN	zero	1	0	0	0	0
Div	NaN	denorm	max	1	0	0	0	0
Div	NaN	zero	max	1	0	0	0	0
Div	NaN	norm	max	1	0	0	0	0
Div	denorm	∞	zero	1	0	0	0	0
Div	denorm	NaN	zero	1	0	0	0	0
Div	denorm	denorm	max	1	0	0	0	0
Div	denorm	zero	max	1	0	0	0	0
Div	denorm	norm	zero	1	0	0	0	0
Div	zero	∞	zero	1	0	0	0	0
Div	zero	NaN	zero	1	0	0	0	0
Div	zero	denorm	max	1	0	0	0	0
Div	zero	zero	max	1	0	0	0	0
Div	zero	norm	zero	0	0	0	0	0
Div	norm	∞	zero	1	0	0	0	0
Div	norm	NaN	zero	1	0	0	0	0
Div	norm	denorm	max	1	0	0	0	0
Div	norm	zero	max	0	0	0	1	0
Div	norm	norm	_Calc_	0	*	*	0	*

Table 4: Embedded Floating-Point Results Summary—Single Convert from Double

Operand B	efscfd result	FINV	FOVF	FUNF	FDBZ	FINX
+∞	pmax	1	0	0	0	0
-∞	nmax	1	0	0	0	0
+NaN	pmax	1	0	0	0	0
-NaN	nmax	1	0	0	0	0
+denorm	+zero	1	0	0	0	0
-denorm	-zero	1	0	0	0	0
+zero	+zero	0	0	0	0	0
-zero	-zero	0	0	0	0	0
norm	_Calc_	0	*	*	0	*

Table 5: Embedded Floating-Point Results Summary—Double Convert from Single

Operand B	efdcfs result	FINV	FOVF	FUNF	FDBZ	FINX
+∞	pmax	1	0	0	0	0
-∞	nmax	1	0	0	0	0
+NaN	pmax	1	0	0	0	0
-NaN	nmax	1	0	0	0	0
+denorm	+zero	1	0	0	0	0
-denorm	-zero	1	0	0	0	0
+zero	+zero	0	0	0	0	0
-zero	-zero	0	0	0	0	0
norm	_Calc_	0	0	0	0	0

Table 6: Embedded Floating-Point Results Summary—Convert to Unsigned

Operand B	Integer Result ctui[d][z]	Fractional Result ctuf	FINV	FOVF	FUNF	FDBZ	FINX
+∞	0xFFFF_FFFF 0xFFFF_FFFF_FFFF_FFFF	0x7FFF_FFFF	1	0	0	0	0
-∞	0	0	1	0	0	0	0
+NaN	0	0	1	0	0	0	0
-NaN	0	0	1	0	0	0	0
denorm	0	0	1	0	0	0	0
zero	0	0	0	0	0	0	0
+norm	_Calc_	_Calc_	*	0	0	0	*
-norm	_Calc_	_Calc_	*	0	0	0	*

Table 7: Embedded Floating-Point Results Summary—Convert to Signed							
Operand B	**Integer Result ctsi[d][z]**	**Fractional Result ctsf**	**FINV**	**FOVF**	**FUNF**	**FDBZ**	**FINX**
+∞	0x7FFF_FFFF 0x7FFF_FFFF_FFFF_FFFF	0x7FFF_FFFF	1	0	0	0	0
-∞	0x8000_0000 0x8000_0000_0000_0000	0x8000_0000	1	0	0	0	0
+NaN	0	0	1	0	0	0	0
-NaN	0	0	1	0	0	0	0
denorm	0	0	1	0	0	0	0
zero	0	0	0	0	0	0	0
+norm	_Calc_	_Calc_	*	0	0	0	*
-norm	_Calc_	_Calc_	*	0	0	0	*

Table 8: Embedded Floating-Point Results Summary—Convert from Unsigned							
Operand B	**Integer Source cfui**	**Fractional Source cfuf**	**FINV**	**FOVF**	**FUNF**	**FDBZ**	**FINX**
zero	zero	zero	0	0	0	0	0
norm	_Calc_	_Calc_	0	0	0	0	*

Table 9: Embedded Floating-Point Results Summary—Convert from Signed							
Operand B	**Integer Source cfsi**	**Fractional Source cfsf**	**FINV**	**FOVF**	**FUNF**	**FDBZ**	**FINX**
zero	zero	zero	0	0	0	0	0
norm	_Calc_	_Calc_	0	0	0	0	*

Table 10:Embedded Floating-Point Results Summary—*abs, *nabs, *neg								
Operand A	***abs**	***nabs**	***neg**	**FINV**	**FOVF**	**FUNF**	**FDBZ**	**FINX**
+∞	pmax | +∞	nmax | -∞	-amax | -∞	1	0	0	0	0
-∞	pmax | +∞	nmax | -∞	-amax | +∞	1	0	0	0	0
+NaN	pmax | NaN	nmax | -NaN	-amax | -NaN	1	0	0	0	0
-NaN	pmax | NaN	nmax | -NaN	-amax | +NaN	1	0	0	0	0
+denorm	+zero | +denorm	-zero | -denorm	-zero | -denorm	1	0	0	0	0
-denorm	+zero | +denorm	-zero | -denorm	+zero | +denorm	1	0	0	0	0
+zero	+zero	-zero	-zero	0	0	0	0	0
-zero	+zero	-zero	+zero	0	0	0	0	0
+norm	+norm	-norm	-norm	0	0	0	0	0
-norm	+norm	-norm	+norm	0	0	0	0	0

Chapter 8. Legacy Move Assist Instruction [Category: Legacy Move Assist]

Determine Leftmost Zero Byte X-form

dlmzb	RA,RS,RB	(Rc=0)
dlmzb.	RA,RS,RB	(Rc=1)

31	RS	RA	RB	78	Rc
0	6	11	16	21	31

```
d0:63 ← (RS)32:63 || (RB)32:63
i ← 0
x ← 0
y ← 0
do while (x<8) & (y=0)
   x ← x + 1
   if di+32:i+39 = 0 then
      y ← 1
   else
      i ← i + 8
RA ← x
XER57:63 ← x
if Rc = 1 then do
   CR35 ← SO
   if y = 1 then do
      if x<5 then CR32:34 ← 0b010
      else        CR32:34 ← 0b100
   else
      CR32:34 ← 0b001
```

The contents of bits 32:63 of register RS and the contents of bits 32:63 of register RB are concatenated to form an 8-byte operand. The operand is searched for the leftmost byte in which each bit is 0 (i.e., a null byte).

Bytes in the operand are numbered from left to right starting with 1. If a null byte is found, its byte number is placed into bits 57:63 of the XER and into register RA. Otherwise, the value `0b000_1000` is placed into both bits 57:63 of the XER and register RA.

If Rc is equal to 1, SO is copied into bit 35 of the CR and bits 32:34 of the CR are updated as follows:

- If no null byte is found, bits 32:34 of the CR are set to `0b001`.
- If the leftmost null byte is in the first 4 bytes (i.e., from register RS), bits 32:34 of the CR are set to `0b010`.
- If the leftmost null byte is in the last 4 bytes (i.e., from register RB), bits 32:34 of the CR are set to `0b100`.

Special Registers Altered:

$XER_{57:63}$
CR0 (if Rc=1)

Chapter 9. Legacy Integer Multiply-Accumulate Instructions [Category: Legacy Integer Multiply-Accumulate]

The *Legacy Integer Multiply-Accumulate* instructions with Rc=1 set the first three bits of CR Field 0 based on the 32-bit result, as described in Section 3.3.7, “Other Fixed-Point Instructions”.

The XO-form *Legacy Integer Multiply-Accumulate* instructions set SO and OV when OE=1 to reflect overflow of the 32-bit result.

Programming Note

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

Multiply Accumulate Cross Halfword to Word Modulo Signed XO-form

macchw	RT,RA,RB	(OE=0 Rc=0)
macchw.	RT,RA,RB	(OE=0 Rc=1)
macchwo	RT,RA,RB	(OE=1 Rc=0)
macchwo.	RT,RA,RB	(OE=1 Rc=1)

4	RT	RA	RB	OE	172	Rc
0	6	11	16	21	22	31

```
prod_{0:31} ← (RA)_{48:63} ×_si (RB)_{32:47}
temp_{0:32} ← prod_{0:31} + (RT)_{32:63}
RT_{32:63} ← temp_{1:32}
RT_{0:31}  ← undefined
```

The signed-integer halfword in bits 48:63 of register RA is multiplied by the signed-integer halfword in bits 32:47 of register RB.

The 32-bit signed-integer product is added to the signed-integer word in bits 32:63 of register RT.

The low-order 32 bits of the sum are placed into bits 32:63 of register RT.

The contents of bits 0:31 of register RT are undefined.

Special Registers Altered:

SO OV	(if OE=1)
CR0	(if Rc=1)

Multiply Accumulate Cross Halfword to Word Saturate Signed XO-form

macchws	RT,RA,RB	(OE=0 Rc=0)
macchws.	RT,RA,RB	(OE=0 Rc=1)
macchwso	RT,RA,RB	(OE=1 Rc=0)
macchwso.	RT,RA,RB	(OE=1 Rc=1)

4	RT	RA	RB	OE	236	Rc
0	6	11	16	21	22	31

```
prod_{0:31} ← (RA)_{48:63} ×_si (RB)_{32:47}
temp_{0:32} ← prod_{0:31} + RT_{32:63}
if temp < -2^31        then RT_{32:63} ← 0x8000_0000
else if temp > 2^31-1 then RT_{32:63} ← 0x7FFF_FFFF
else                        RT_{32:63} ← temp_{1:32}
RT_{0:31} ← undefined
```

The signed-integer halfword in bits 48:63 of register RA is multiplied by the signed-integer halfword in bits 32:47 of register RB.

The 32-bit signed-integer product is added to the signed-integer word in bits 32:63 of register RT.

If the sum is less than -2^{31}, then the value `0x8000_0000` is placed into bits 32:63 of register RT.

If the sum is greater than $2^{31}-1$, then the value `0x7FFF_FFFF` is placed into bits 32:63 of register RT.

Otherwise, the sum is placed into bits 32:63 of register RT.

The contents of bits 0:31 of register RT are undefined.

Special Registers Altered:

SO OV	(if OE=1)
CR0	(if Rc=1)

Multiply Accumulate Cross Halfword to Word Modulo Unsigned XO-form

macchwu	RT,RA,RB	(OE=0 Rc=0)
macchwu.	RT,RA,RB	(OE=0 Rc=1)
macchwuo	RT,RA,RB	(OE=1 Rc=0)
macchwuo.	RT,RA,RB	(OE=1 Rc=1)

4	RT	RA	RB	OE	140	Rc
0	6	11	16	21	22	31

$prod_{0:31} \leftarrow (RA)_{48:63} \times_{ui} (RB)_{32:47}$
$temp_{0:32} \leftarrow prod_{0:31} + (RT)_{32:63}$
$RT \leftarrow temp_{1:32}$

The unsigned-integer halfword in bits 48:63 of register RA is multiplied by the unsigned-integer halfword in bits 32:47 of register RB.

The 32-bit unsigned-integer product is added to the unsigned-integer word in bits 32:63 of register RT.

The low-order 32 bits of the sum are placed into bits 32:63 of register RT.

The contents of bits 0:31 of register RT are undefined.

Special Registers Altered:

SO OV	(if OE=1)
CR0	(if Rc=1)

Multiply Accumulate Cross Halfword to Word Saturate Unsigned XO-form

macchwsu	RT,RA,RB	(OE=0 Rc=0)
macchwsu.	RT,RA,RB	(OE=0 Rc=1)
macchwsuo	RT,RA,RB	(OE=1 Rc=0)
macchwsuo.	RT,RA,RB	(OE=1 Rc=1)

4	RT	RA	RB	OE	204	Rc
0	6	11	16	21	22	31

$prod_{0:31} \leftarrow (RA)_{48:63} \times_{ui} (RB)_{32:47}$
$temp_{0:32} \leftarrow prod_{0:31} + (RT)_{32:63}$
if temp > 2^{32}-1 then RT ← 0xFFFF_FFFF
else RT ← $temp_{1:32}$

The unsigned-integer halfword in bits 48:63 of register RA is multiplied by the unsigned-integer halfword in bits 32:47 of register RB.

The 32-bit unsigned-integer product is added to the unsigned-integer word in bits 32:63 of register RT.

If the sum is greater than 2^{32}-1, then the value 0xFFFF_FFFF is placed into bits 32:63 of register RT.

Otherwise, the sum is placed into bits 32:63 of register RT.

The contents of bits 0:31 of register RT are undefined.

Special Registers Altered:

SO OV	(if OE=1)
CR0	(if Rc=1)

Multiply Accumulate High Halfword to Word Modulo Signed XO-form

machhw	RT,RA,RB	(OE=0 Rc=0)
machhw.	RT,RA,RB	(OE=0 Rc=1)
machhwo	RT,RA,RB	(OE=1 Rc=0)
machhwo.	RT,RA,RB	(OE=1 Rc=1)

4	RT	RA	RB	OE	44	Rc
0	6	11	16	21	22	31

$prod_{0:31} \leftarrow (RA)_{32:47} \times_{si} (RB)_{32:47}$
$temp_{0:32} \leftarrow prod_{0:31} + (RT)_{32:63}$
$RT_{32:63} \leftarrow temp_{1:32}$
$RT_{0:31} \leftarrow$ undefined

The signed-integer halfword in bits 32:47 of register RA is multiplied by the signed-integer halfword in bits 32:47 of register RB.

The 32-bit signed-integer product is added to the signed-integer word in bits 32:63 of register RT.

The low-order 32 bits of the sum are placed into bits 32:63 of register RT.

The contents of bits 0:31 of register RT are undefined.

Special Registers Altered:

SO OV	(if OE=1)
CR0	(if Rc=1)

Multiply Accumulate High Halfword to Word Saturate Signed XO-form

machhws	RT,RA,RB	(OE=0 Rc=0)
machhws.	RT,RA,RB	(OE=0 Rc=1)
machhwso	RT,RA,RB	(OE=1 Rc=0)
machhwso.	RT,RA,RB	(OE=1 Rc=1)

4	RT	RA	RB	OE	108	Rc
0	6	11	16	21	22	31

$prod_{0:31} \leftarrow (RA)_{32:47} \times_{si} (RB)_{32:47}$
$temp_{0:32} \leftarrow prod_{0:31} + (RT)_{32:63}$
if temp < -2^{31} then $RT_{32:63} \leftarrow$ 0x8000_0000
else if temp > $2^{31}-1$ then $RT_{32:63} \leftarrow$ 0x7FFF_FFFF
else $RT_{32:63} \leftarrow temp_{1:32}$
$RT_{0:31} \leftarrow$ undefined

The signed-integer halfword in bits 32:47 of register RA is multiplied by the signed-integer halfword in bits 32:47 of register RB.

The 32-bit signed-integer product is added to the signed-integer word in bits 32:63 of register RT.

If the sum is less than -2^{31}, then the value `0x8000_0000` is placed into bits 32:63 of register RT.

If the sum is greater than $2^{31}-1$, then the value `0x7FFF_FFFF` is placed into bits 32:63 of register RT.

Otherwise, the sum is placed into bits 32:63 of register RT.

The contents of bits 0:31 of register RT are undefined.

Special Registers Altered:

SO OV	(if OE=1)
CR0	(if Rc=1)

Multiply Accumulate High Halfword to Word Modulo Unsigned XO-form

machhwu	RT,RA,RB	(OE=0 Rc=0)
machhwu.	RT,RA,RB	(OE=0 Rc=1)
machhwuo	RT,RA,RB	(OE=1 Rc=0)
machhwuo.	RT,RA,RB	(OE=1 Rc=1)

4	RT	RA	RB	OE	12	Rc
0	6	11	16	21	22	31

$prod_{0:31} \leftarrow (RA)_{32:47} \times_{ui} (RB)_{32:47}$
$temp_{0:32} \leftarrow prod_{0:31} + (RT)_{32:63}$
$RT_{32:63} \leftarrow temp_{1:32}$
$RT_{0:31} \leftarrow$ undefined

The unsigned-integer halfword in bits 32:47 of register RA is multiplied by the unsigned-integer halfword in bits 32:47 of register RB.

The 32-bit unsigned-integer product is added to the unsigned-integer word in bits 32:63 of register RT.

The low-order 32 bits of the sum are placed into bits 32:63 of register RT.

The contents of bits 0:31 of register RT are undefined.

Special Registers Altered:

SO OV	(if OE=1)
CR0	(if Rc=1)

Multiply Accumulate High Halfword to Word Saturate Unsigned XO-form

machhwsu	RT,RA,RB	(OE=0 Rc=0)
machhwsu.	RT,RA,RB	(OE=0 Rc=1)
machhwsuo	RT,RA,RB	(OE=1 Rc=0)
machhwsuo.	RT,RA,RB	(OE=1 Rc=1)

4	RT	RA	RB	OE	76	Rc
0	6	11	16	21	22	31

$prod_{0:31} \leftarrow (RA)_{32:47} \times_{ui} (RB)_{32:47}$
$temp_{0:32} \leftarrow prod_{0:31} + (RT)_{32:63}$
if temp > 2^{32}-1 then RT $\leftarrow$ 0xFFFF_FFFF
else RT $\leftarrow temp_{1:32}$

The unsigned-integer halfword in bits 32:47 of register RA is multiplied by the unsigned-integer halfword in bits 32:47 of register RB.

The 32-bit unsigned-integer product is added to the unsigned-integer word in bits 32:63 of register RT.

If the sum is greater than 2^{32}-1, then the value 0xFFFF_FFFF is placed into bits 32:63 of register RT.

Otherwise, the sum is placed into bits 32:63 of register RT.

The contents of bits 0:31 of register RT are undefined.

Special Registers Altered:

SO OV	(if OE=1)
CR0	(if Rc=1)

Multiply Accumulate Low Halfword to Word Modulo Signed *XO-form*

maclhw	RT,RA,RB	(OE=0 Rc=0)
maclhw.	RT,RA,RB	(OE=0 Rc=1)
maclhwo	RT,RA,RB	(OE=1 Rc=0)
maclhwo.	RT,RA,RB	(OE=1 Rc=1)

4	RT	RA	RB	OE	428	Rc
0	6	11	16	21	22	31

$prod_{0:31} \leftarrow (RA)_{48:63} \times_{si} (RB)_{48:63}$
$temp_{0:32} \leftarrow prod_{0:31} + (RT)_{32:63}$
$RT_{32:63} \leftarrow temp_{1:32}$
$RT_{0:31} \leftarrow$ undefined

The signed-integer halfword in bits 48:63 of register RA is multiplied by the signed-integer halfword in bits 48:63 of register RB.

The 32-bit signed-integer product is added to the signed-integer word in bits 32:63 of register RT.

The low-order 32 bits of the sum are placed into bits 32:63 of register RT.

The contents of bits 0:31 of register RT are undefined.

Special Registers Altered:

SO OV	(if OE=1)
CR0	(if Rc=1)

Multiply Accumulate Low Halfword to Word Saturate Signed *XO-form*

maclhws	RT,RA,RB	(OE=0 Rc=0)
maclhws.	RT,RA,RB	(OE=0 Rc=1)
maclhwso	RT,RA,RB	(OE=1 Rc=0)
maclhwso.	RT,RA,RB	(OE=1 Rc=1)

4	RT	RA	RB	OE	492	Rc
0	6	11	16	21	22	31

$prod_{0:31} \leftarrow (RA)_{48:63} \times_{si} (RB)_{48:63}$
$temp_{0:32} \leftarrow prod_{0:31} + (RT)_{32:63}$
if temp $< -2^{31}$ then $RT_{32:63} \leftarrow$ 0x8000_0000
else if temp $> 2^{31}-1$ then $RT_{32:63} \leftarrow$ 0x7FFF_FFFF
else $RT_{32:63} \leftarrow temp_{1:32}$
$RT_{0:31} \leftarrow$ undefined

The signed-integer halfword in bits 48:63 of register RA is multiplied by the signed-integer halfword in bits 48:63 of register RB.

The 32-bit signed-integer product is added to the signed-integer word in bits 32:63 of register RT.

If the sum is less than -2^{31}, then the value 0x8000_0000 is placed into bits 32:63 of register RT.

If the sum is greater than $2^{31}-1$, then the value 0x7FFF_FFFF is placed into bits 32:63 of register RT.

Otherwise, the sum is placed into bits 32:63 of register RT.

The contents of bits 0:31 of register RT are undefined.

Special Registers Altered:

SO OV	(if OE=1)
CR0	(if Rc=1)

Multiply Accumulate Low Halfword to Word Modulo Unsigned XO-form

maclhwu	RT,RA,RB	(OE=0 Rc=0)
maclhwu.	RT,RA,RB	(OE=0 Rc=1)
maclhwuo	RT,RA,RB	(OE=1 Rc=0)
maclhwuo.	RT,RA,RB	(OE=1 Rc=1)

4	RT	RA	RB	OE	396	Rc
0	6	11	16	21	22	31

$prod_{0:31} \leftarrow (RA)_{48:63} \times_{ui} (RB)_{48:63}$
$temp_{0:32} \leftarrow prod_{0:31} + (RT)_{32:63}$
$RT_{32:63} \leftarrow temp_{1:32}$
$RT_{0:31} \leftarrow$ undefined

The unsigned-integer halfword in bits 48:63 of register RA is multiplied by the unsigned-integer halfword in bits 48:63 of register RB.

The 32-bit unsigned-integer product is added to the unsigned-integer word in bits 32:63 of register RT.

The low-order 32 bits of the sum are placed into bits 32:63 of register RT.

The contents of bits 0:31 of register RT are undefined.

Special Registers Altered:

SO OV	(if OE=1)
CR0	(if Rc=1)

Multiply Accumulate Low Halfword to Word Saturate Unsigned XO-form

maclhwsu	RT,RA,RB	(OE=0 Rc=0)
maclhwsu.	RT,RA,RB	(OE=0 Rc=1)
maclhwsuo	RT,RA,RB	(OE=1 Rc=0)
maclhwsuo.	RT,RA,RB	(OE=1 Rc=1)

4	RT	RA	RB	OE	460	Rc
0	6	11	16	21	22	31

$prod_{0:31} \leftarrow (RA)_{48:63} \times_{ui} (RB)_{48:63}$
$temp_{0:32} \leftarrow prod_{0:31} + (RT)_{32:63}$
if temp > 2^{32}-1 then RT ← 0xFFFF_FFFF
else RT ← $temp_{1:32}$

The unsigned-integer halfword in bits 48:63 of register RA is multiplied by the unsigned-integer halfword in bits 48:63 of register RB.

The 32-bit unsigned-integer product is added to the unsigned-integer word in bits 32:63 of register RT.

If the sum is greater than 2^{32}-1, then the value 0xFFFF_FFFF is placed into bits 32:63 of register RT.

Otherwise, the sum is placed into bits 32:63 of register RT.

The contents of bits 0:31 of register RT are undefined.

Special Registers Altered:

SO OV	(if OE=1)
CR0	(if Rc=1)

Multiply Cross Halfword to Word Signed X-form

mulchw	RT,RA,RB	(Rc=0)
mulchw.	RT,RA,RB	(Rc=1)

4	RT	RA	RB	168	Rc
0	6	11	16	21	31

$RT_{32:63} \leftarrow (RA)_{48:63} \times_{si} (RB)_{32:47}$
$RT_{0:31} \leftarrow$ undefined

The signed-integer halfword in bits 48:63 of register RA is multiplied by the signed-integer halfword in bits 32:47 of register RB and the signed-integer word result is placed into bits 32:63 of register RT.

The contents of bits 0:31 of register RT are undefined.

Special Registers Altered:

CR0	(if Rc=1)

Multiply Cross Halfword to Word Unsigned X-form

mulchwu	RT,RA,RB	(Rc=0)
mulchwu.	RT,RA,RB	(Rc=1)

4	RT	RA	RB	136	Rc
0	6	11	16	21	31

$RT_{32:63} \leftarrow (RA)_{48:63} \times_{ui} (RB)_{32:47}$
$RT_{0:31} \leftarrow$ undefined

The unsigned-integer halfword in bits 48:63 of register RA is multiplied by the unsigned-integer halfword in bits 32:47 of register RB and the unsigned-integer word result is placed into bits 32:63 of register RT.

The contents of bits 0:31 of register RT are undefined.

Special Registers Altered:

CR0	(if Rc=1)

Multiply High Halfword to Word Signed X-form

mulhhw RT,RA,RB (Rc=0)
mulhhw. RT,RA,RB (Rc=1)

4	RT	RA	RB	40	Rc
0	6	11	16	21	31

$RT_{32:63} \leftarrow (RA)_{32:47} \times_{si} (RB)_{32:47}$
$RT_{0:31} \leftarrow$ undefined

The signed-integer halfword in bits 32:47 of register RA is multiplied by the signed-integer halfword in bits 32:47 of register RB and the signed-integer word result is placed into bits 32:63 of register RT.

The contents of bits 0:31 of register RT are undefined.

Special Registers Altered:
CR0 (if Rc=1)

Multiply High Halfword to Word Unsigned X-form

mulhhwu RT,RA,RB (Rc=0)
mulhhwu. RT,RA,RB (Rc=1)

4	RT	RA	RB	8	Rc
0	6	11	16	21	31

$RT_{32:63} \leftarrow (RA)_{32:47} \times_{ui} (RB)_{32:47}$
$RT_{0:31} \leftarrow$ undefined

The unsigned-integer halfword in bits 32:47 of register RA is multiplied by the unsigned-integer halfword in bits 32:47 of register RB and the unsigned-integer word result is placed into bits 32:63 of register RT.

The contents of bits 0:31 of register RT are undefined.

Special Registers Altered:
CR0 (if Rc=1)

Multiply Low Halfword to Word Signed X-form

mullhw RT,RA,RB (Rc=0)
mullhw. RT,RA,RB (Rc=1)

4	RT	RA	RB	424	Rc
0	6	11	16	21	31

$RT_{32:63} \leftarrow (RA)_{48:63} \times_{si} (RB)_{48:63}$
$RT_{0:31} \leftarrow$ undefined

The signed-integer halfword in bits 48:63 of register RA is multiplied by the signed-integer halfword in bits 48:63 of register RB and the signed-integer word result is placed into bits 32:63 of register RT.

The contents of bits 0:31 of register RT are undefined.

Special Registers Altered:
CR0 (if Rc=1)

Multiply Low Halfword to Word Unsigned X-form

mullhwu RT,RA,RB (Rc=0)
mullhwu. RT,RA,RB (Rc=1)

4	RT	RA	RB	392	Rc
0	6	11	16	21	31

$RT_{32:63} \leftarrow (RA)_{48:63} \times_{ui} (RB)_{48:63}$
$RT_{0:31} \leftarrow$ undefined

The unsigned-integer halfword in bits 48:63 of register RA is multiplied by the unsigned-integer halfword in bits 48:63 of register RB and the unsigned-integer word result is placed into bits 32:63 of register RT.

The contents of bits 0:31 of register RT are undefined.

Special Registers Altered:
CR0 (if Rc=1)

Negative Multiply Accumulate Cross Halfword to Word Modulo Signed *XO-form*

nmacchw	RT,RA,RB	(OE=0 Rc=0)
nmacchw.	RT,RA,RB	(OE=0 Rc=1)
nmacchwo	RT,RA,RB	(OE=1 Rc=0)
nmacchwo.	RT,RA,RB	(OE=1 Rc=1)

4	RT	RA	RB	OE	174	Rc
0	6	11	16	21	22	31

$prod_{0:31} \leftarrow (RA)_{48:63} \times_{si} (RB)_{32:47}$
$temp_{0:32} \leftarrow (RT)_{32:63} -_{si} prod_{0:31}$
$RT_{32:63} \leftarrow temp_{1:32}$
$RT_{0:31} \leftarrow$ undefined

The signed-integer halfword in bits 48:63 of register RA is multiplied by the signed-integer halfword in bits 32:47 of register RB.

The 32-bit signed-integer product is subtracted from the signed-integer word in bits 32:63 of register RT.

The low-order 32 bits of the difference are placed into bits 32:63 of register RT.

The contents of bits 0:31 of register RT are undefined.

Special Registers Altered:

SO OV	(if OE=1)
CR0	(if Rc=1)

Negative Multiply Accumulate Cross Halfword to Word Saturate Signed *XO-form*

nmacchws	RT,RA,RB	(OE=0 Rc=0)
nmacchws.	RT,RA,RB	(OE=0 Rc=1)
nmacchwso	RT,RA,RB	(OE=1 Rc=0)
nmacchwso.	RT,RA,RB	(OE=1 Rc=1)

4	RT	RA	RB	OE	238	Rc
0	6	11	16	21	22	31

$prod_{0:31} \leftarrow (RA)_{48:63} \times_{si} (RB)_{32:47}$
$temp_{0:32} \leftarrow (RT)_{32:63} -_{si} prod_{0:31}$
if $temp < -2^{31}$ then $RT_{32:63} \leftarrow$ 0x8000_0000
else if $temp > 2^{31}-1$ then $RT_{32:63} \leftarrow$ 0x7FFF_FFFF
else $RT_{32:63} \leftarrow temp_{1:32}$
$RT_{0:31} \leftarrow$ undefined

The signed-integer halfword in bits 48:63 of register RA is multiplied by the signed-integer halfword in bits 32:47 of register RB.

The 32-bit signed-integer product is subtracted from the signed-integer word in bits 32:63 of register RT.

If the difference is less than -2^{31}, then the value `0x8000_0000` is placed into bits 32:63 of register RT.

If the difference is greater than $2^{31}-1$, then the value `0x7FFF_FFFF` is placed into bits 32:63 of register RT.

Otherwise, the difference is placed into bits 32:63 of register RT.

The contents of bits 0:31 of register RT are undefined.

Special Registers Altered:

SO OV	(if OE=1)
CR0	(if Rc=1)

Negative Multiply Accumulate High Halfword to Word Modulo Signed XO-form

nmachhw	RT,RA,RB	(OE=0 Rc=0)
nmachhw.	RT,RA,RB	(OE=0 Rc=1)
nmachhwo	RT,RA,RB	(OE=1 Rc=0)
nmachhwo.	RT,RA,RB	(OE=1 Rc=1)

4	RT	RA	RB	OE	46	Rc
0	6	11	16	21	22	31

$prod_{0:31} \leftarrow (RA)_{32:47} \times_{si} (RB)_{32:47}$
$temp_{0:32} \leftarrow (RT)_{32:63} -_{si} prod_{0:31}$
$RT_{32:63} \leftarrow temp_{1:32}$
$RT_{0:31} \leftarrow$ undefined

The signed-integer halfword in bits 32:47 of register RA is multiplied by the signed-integer halfword in bits 32:47 of register RB.

The 32-bit signed-integer product is subtracted from the signed-integer word in bits 32:63 of register RT.

The low-order 32 bits of the difference are placed into bits 32:63 of register RT.

The contents of bits 0:31 of register RT are undefined.

Special Registers Altered:

SO OV	(if OE=1)
CR0	(if Rc=1)

Negative Multiply Accumulate High Halfword to Word Saturate Signed XO-form

nmachhws	RT,RA,RB	(OE=0 Rc=0)
nmachhws.	RT,RA,RB	(OE=0 Rc=1)
nmachhwso	RT,RA,RB	(OE=1 Rc=0)
nmachhwso.	RT,RA,RB	(OE=1 Rc=1)

4	RT	RA	RB	OE	110	Rc
0	6	11	16	21	22	31

$prod_{0:31} \leftarrow (RA)_{32:47} \times_{si} (RB)_{32:47}$
$temp_{0:32} \leftarrow (RT)_{32:63} -_{si} prod_{0:31}$
if temp < -2^{31} then $RT_{32:63} \leftarrow$ 0x8000_0000
else if temp > $2^{31}-1$ then $RT_{32:63} \leftarrow$ 0x7FFF_FFFF
else $RT_{32:63} \leftarrow temp_{1:32}$
$RT_{0:31} \leftarrow$ undefined

The signed-integer halfword in bits 32:47 of register RA is multiplied by the signed-integer halfword in bits 32:47 of register RB.

The 32-bit signed-integer product is subtracted from the signed-integer word in bits 32:63 of register RT.

If the difference is less than -2^{31}, then the value `0x8000_0000` is placed into bits 32:63 of register RT.

If the difference is greater than $2^{31}-1$, then the value `0x7FFF_FFFF` is placed into bits 32:63 of register RT.

Otherwise, the difference is placed into bits 32:63 of register RT.

The contents of bits 0:31 of register RT are undefined.

Special Registers Altered:

SO OV	(if OE=1)
CR0	(if Rc=1)

Negative Multiply Accumulate Low Halfword to Word Modulo Signed XO-form

nmaclhw	RT,RA,RB	(OE=0 Rc=0)
nmaclhw.	RT,RA,RB	(OE=0 Rc=1)
nmaclhwo	RT,RA,RB	(OE=1 Rc=0)
nmaclhwo.	RT,RA,RB	(OE=1 Rc=1)

4	RT	RA	RB	OE	430	Rc
0	6	11	16	21	22	31

$prod_{0:31} \leftarrow (RA)_{48:63} \times_{si} (RB)_{48:63}$
$temp_{0:32} \leftarrow (RT)_{32:63} -_{si} prod_{0:31}$
$RT_{32:63} \leftarrow temp_{1:32}$
$RT_{0:31} \leftarrow$ undefined

The signed-integer halfword in bits 48:63 of register RA is multiplied by the signed-integer halfword in bits 48:63 of register RB.

The 32-bit signed-integer product is subtracted from the signed-integer word in bits 32:63 of register RT.

The low-order 32 bits of the difference are placed into bits 32:63 of register RT.

The contents of bits 0:31 of register RT are undefined.

Special Registers Altered:

SO OV	(if OE=1)
CR0	(if Rc=1)

Negative Multiply Accumulate Low Halfword to Word Saturate Signed XO-form

nmaclhws	RT,RA,RB	(OE=0 Rc=0)
nmaclhws.	RT,RA,RB	(OE=0 Rc=1)
nmaclhwso	RT,RA,RB	(OE=1 Rc=0)
nmaclhwso.	RT,RA,RB	(OE=1 Rc=1)

4	RT	RA	RB	OE	494	Rc
0	6	11	16	21	22	31

$prod_{0:31} \leftarrow (RA)_{48:63} \times_{si} (RB)_{48:63}$
$temp_{0:32} \leftarrow (RT)_{32:63} -_{si} prod_{0:31}$
if temp < -2^{31} then $RT_{32:63} \leftarrow$ 0x8000_0000
else if temp > $2^{31}-1$ then $RT_{32:63} \leftarrow$ 0x7FFF_FFFF
else $RT_{32:63} \leftarrow temp_{1:32}$
$RT_{0:31} \leftarrow$ undefined

The signed-integer halfword in bits 48:63 of register RA is multiplied by the signed-integer halfword in bits 48:63 of register RB.

The 32-bit signed-integer product is subtracted from the signed-integer word in bits 32:63 of register RT.

If the difference is less than -2^{31}, then the value 0x8000_0000 is placed into bits 32:63 of register RT.

If the difference is greater than $2^{31}-1$, then the value 0x7FFF_FFFF is placed into bits 32:63 of register RT.

Otherwise, the difference is placed into bits 32:63 of register RT.

The contents of bits 0:31 of register RT are undefined.

Special Registers Altered:

SO OV	(if OE=1)
CR0	(if Rc=1)

Appendix A. Suggested Floating-Point Models [Category: Floating-Point]

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
```

```
            FRT_0 ← sign
            FRT_1:63 ← 0
            If sign = 0 then FPSCR_FPRF ← "+ zero"
            If sign = 1 then FPSCR_FPRF ← "- zero"
        End
    If frac_0:52 > 0 then
        Do
            If frac_0 = 1 then
                Do
                    If sign = 0 then FPSCR_FPRF ← "+ normal number"
                    If sign = 1 then FPSCR_FPRF ← "- normal number"
                End
            If frac_0 = 0 then
                Do
                    If sign = 0 then FPSCR_FPRF ← "+ denormalized number"
                    If sign = 1 then FPSCR_FPRF ← "- denormalized number"
                End
            Normalize operand:
                Do while frac_0 = 0
                    exp ← exp-1
                    frac_0:52 ← frac_1:52 || 0b0
                End
            FRT_0 ← sign
            FRT_1:11 ← exp + 1023
            FRT_12:63 ← frac_1:52
        End
    Done
```

Enabled Exponent Underflow:

```
    FPSCR_UX ← 1
    sign ← (FRB)_0
    If (FRB)_1:11 = 0 then
        Do
            exp ← -1022
            frac_0:52 ← 0b0 || (FRB)_12:63
        End
    If (FRB)_1:11 > 0 then
        Do
            exp ← (FRB)_1:11 - 1023
            frac_0:52 ← 0b1 || (FRB)_12:63
        End
    Normalize operand:
        Do while frac_0 = 0
            exp ← exp - 1
            frac_0:52 ← frac_1:52 || 0b0
        End
    Round Single(sign,exp,frac_0:52,0,0,0)
    FPSCR_XX ← FPSCR_XX | FPSCR_FI
    exp ← exp + 192
    FRT_0 ← sign
    FRT_1:11 ← exp + 1023
    FRT_12:63 ← frac_1:52
    If sign = 0 then FPSCR_FPRF ← "+ normal number"
    If sign = 1 then FPSCR_FPRF ← "- normal number"
    Done
```

Disabled Exponent Overflow:

```
    FPSCR_OX ← 1
    If FPSCR_RN = 0b00 then              /* Round to Nearest */
        Do
            If (FRB)_0 = 0 then FRT ← 0x7FF0_0000_0000_0000
            If (FRB)_0 = 1 then FRT ← 0xFFF0_0000_0000_0000
            If (FRB)_0 = 0 then FPSCR_FPRF ← "+ infinity"
            If (FRB)_0 = 1 then FPSCR_FPRF ← "- infinity"
        End
```

```
If FPSCR_RN = 0b01 then              /* Round toward Zero */
    Do
        If (FRB)_0 = 0 then FRT ← 0x47EF_FFFF_E000_0000
        If (FRB)_0 = 1 then FRT ← 0xC7EF_FFFF_E000_0000
        If (FRB)_0 = 0 then FPSCR_FPRF ← "+ normal number"
        If (FRB)_0 = 1 then FPSCR_FPRF ← "- normal number"
    End
If FPSCR_RN = 0b10 then              /* Round toward +Infinity */
    Do
        If (FRB)_0 = 0 then FRT ← 0x7FF0_0000_0000_0000
        If (FRB)_0 = 1 then FRT ← 0xC7EF_FFFF_E000_0000
        If (FRB)_0 = 0 then FPSCR_FPRF ← "+ infinity"
        If (FRB)_0 = 1 then FPSCR_FPRF ← "- normal number"
    End
If FPSCR_RN = 0b11 then              /* Round toward -Infinity */
    Do
        If (FRB)_0 = 0 then FRT ← 0x47EF_FFFF_E000_0000
        If (FRB)_0 = 1 then FRT ← 0xFFF0_0000_0000_0000
        If (FRB)_0 = 0 then FPSCR_FPRF ← "+ normal number"
        If (FRB)_0 = 1 then FPSCR_FPRF ← "- infinity"
    End
FPSCR_FR ← undefined
FPSCR_FI ← 1
FPSCR_XX ← 1
Done
```

Enabled Exponent Overflow:

```
sign ← (FRB)_0
exp ← (FRB)_1:11 - 1023
frac_0:52 ← 0b1 || (FRB)_12:63
Round Single(sign,exp,frac_0:52,0,0,0)
FPSCR_XX ← FPSCR_XX | FPSCR_FI
```

Enabled Overflow:

```
FPSCR_OX ← 1
exp ← exp - 192
FRT_0 ← sign
FRT_1:11 ← exp + 1023
FRT_12:63 ← frac_1:52
If sign = 0 then FPSCR_FPRF ← "+ normal number"
If sign = 1 then FPSCR_FPRF ← "- normal number"
Done
```

Zero Operand:

```
FRT ← (FRB)
If (FRB)_0 = 0 then FPSCR_FPRF ← "+ zero"
If (FRB)_0 = 1 then FPSCR_FPRF ← "- zero"
FPSCRFR_FI ← 0b00
Done
```

Infinity Operand:

```
FRT ← (FRB)
If (FRB)_0 = 0 then FPSCR_FPRF ← "+ infinity"
If (FRB)_0 = 1 then FPSCR_FPRF ← "- infinity"
FPSCRFR_FI ← 0b00
Done
```

QNaN Operand:

```
FRT ← (FRB)_0:34 || ^29 0
FPSCR_FPRF ← "QNaN"
FPSCR_FR FI ← 0b00
Done
```

SNaN Operand:

```
FPSCR_VXSNAN ← 1
If FPSCR_VE = 0 then
    Do
        FRT_0:11 ← (FRB)_0:11
        FRT_12 ← 1
        FRT_13:63 ← (FRB)_13:34 || 29_0
        FPSCR_FPRF ← "QNaN"
    End
FPSCR_FR FI ← 0b00
Done
```

Normal Operand:

```
sign ← (FRB)_0
exp ← (FRB)_1:11 - 1023
frac_0:52 ← 0b1 || (FRB)_12:63
Round Single(sign,exp,frac_0:52,0,0,0)
FPSCR_XX ← FPSCR_XX | FPSCR_FI
If exp > 127 and FPSCR_OE = 0 then go to Disabled Exponent Overflow
If exp > 127 and FPSCR_OE = 1 then go to Enabled Overflow
FRT_0 ← sign
FRT_1:11 ← exp + 1023
FRT_12:63 ← frac_1:52
If sign = 0 then FPSCR_FPRF ← "+ normal number"
If sign = 1 then FPSCR_FPRF ← "- normal number"
Done
```

Round Single(sign,exp,frac$_{0:52}$,G,R,X):

```
inc ← 0
lsb ← frac_23
gbit ← frac_24
rbit ← frac_25
xbit ← (frac_26:52||G||R||X)≠0
If FPSCR_RN = 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 FPSCR_RN = 0b10 then              /* Round toward + Infinity */
    Do              /* comparisons ignore u bits */
        If sign || lsb || gbit || rbit || xbit = 0b0u1uu then inc ← 1
        If sign || lsb || gbit || rbit || xbit = 0b0uu1u then inc ← 1
        If sign || lsb || gbit || rbit || xbit = 0b0uuu1 then inc ← 1
    End
If FPSCR_RN = 0b11 then              /* Round toward - Infinity */
    Do              /* comparisons ignore u bits */
        If sign || lsb || gbit || rbit || xbit = 0b1u1uu then inc ← 1
        If sign || lsb || gbit || rbit || xbit = 0b1uu1u then inc ← 1
        If sign || lsb || gbit || rbit || xbit = 0b1uuu1 then inc ← 1
    End
frac_0:23 ← frac_0:23 + inc
If carry_out = 1 then
    Do
        frac_0:23 ← 0b1 || frac_0:22
        exp ← exp + 1
    End
frac_24:52 ← 29_0
FPSCR_FR ← inc
FPSCR_FI ← gbit | rbit | xbit
Return
```

A.2 Floating-Point Convert to Integer Model

The following describes algorithmically the operation of the *Floating Convert To Integer* instructions.

```
If Floating Convert To Integer Word then
    Do
        round_mode ← FPSCR_RN
        tgt_precision ← "32-bit integer"
    End

If Floating Convert To Integer Word with round toward Zero then
    Do
        round_mode ← 0b01
        tgt_precision ← "32-bit integer"
    End

If Floating Convert To Integer Doubleword then
    Do
        round_mode ← FPSCR_RN
        tgt_precision ← "64-bit integer"
    End

If Floating Convert To Integer Doubleword with round toward Zero then
    Do
        round_mode ← 0b01
        tgt_precision ← "64-bit 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; need leading 0 for later complement */
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 frac_0:64 ← ¬frac_0:64 + 1        /* needed leading 0 for -2^64 < (FRB) < -2^63 */

If tgt_precision = "32-bit integer" and frac_0:64 > 2^31-1 then goto Large Operand
If tgt_precision = "64-bit integer" and frac_0:64 > 2^63-1 then goto Large Operand
If tgt_precision = "32-bit integer" and frac_0:64 < -2^31 then goto Large Operand
If tgt_precision = "64-bit integer" and frac_0:64 < -2^63 then goto Large Operand

FPSCR_XX ← FPSCR_XX | FPSCR_FI

If tgt_precision = "32-bit integer" then FRT ← 0xuuuu_uuuu || frac_33:64   /* u is undefined hex digit */
If tgt_precision = "64-bit integer" then FRT ← frac_1:64
FPSCR_FPRF ← undefined
Done
```

Round Integer(sign,frac$_{0:64}$,gbit,rbit,xbit,round_mode):

```
inc ← 0
If round_mode = 0b00 then          /* Round to Nearest */
    Do          /* comparisons ignore u bits */
        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          /* Round toward +Infinity */
    Do          /* comparisons ignore u bits */
        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          /* Round toward -Infinity */
    Do          /* comparisons ignore u bits */
        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
FPSCR_FR ← inc
FPSCR_FI ← gbit | rbit | xbit
Return
```

Infinity Operand:

```
FPSCR_FR FI VXCVI ← 0b001
If FPSCR_VE = 0 then Do
    If tgt_precision = "32-bit integer" then
        Do
            If sign = 0 then FRT ← 0xuuuu_uuuu_7FFF_FFFF   /* u is undefined hex digit */
            If sign = 1 then FRT ← 0xuuuu_uuuu_8000_0000   /* u is undefined hex digit */
        End
    Else
        Do
            If sign = 0 then FRT ← 0x7FFF_FFFF_FFFF_FFFF
            If sign = 1 then FRT ← 0x8000_0000_0000_0000
        End
    FPSCR_FPRF ← undefined
    End
Done
```

SNaN Operand:

```
FPSCR_FR FI VXSNAN VXCVI ← 0b0011
If FPSCR_VE = 0 then
    Do
        If tgt_precision = "32-bit integer" then FRT ← 0xuuuu_uuuu_8000_0000   /* u is undefined hex digit */
        If tgt_precision = "64-bit integer" then FRT ← 0x8000_0000_0000_0000
        FPSCR_FPRF ← undefined
    End
Done
```

QNaN Operand:

```
FPSCR_FR FI VXCVI ← 0b001
If FPSCR_VE = 0 then
    Do
        If tgt_precision = "32-bit integer" then FRT ← 0xuuuu_uuuu_8000_0000   /* u is undefined hex digit */
        If tgt_precision = "64-bit integer" then FRT ← 0x8000_0000_0000_0000
        FPSCR_FPRF ← undefined
    End
Done
```

Large Operand:

```
FPSCR_FR FI VXCVI ← 0b001
If FPSCR_VE = 0 then Do
    If tgt_precision = "32-bit integer" then
        Do
            If sign = 0 then FRT ← 0xuuuu_uuuu_7FFF_FFFF   /* u is undefined hex digit */
            If sign = 1 then FRT ← 0xuuuu_uuuu_8000_0000   /* u is undefined hex digit */
        End
    Else
        Do
            If sign = 0 then FRT ← 0x7FFF_FFFF_FFFF_FFFF
            If sign = 1 then FRT ← 0x8000_0000_0000_0000
        End
    FPSCR_FPRF ← undefined
    End
Done
```

A.3 Floating-Point Convert from Integer Model

The following describes algorithmically the operation of the *Floating Convert From Integer Doublewor*d instruction.

```
sign ← (FRB)0
exp ← 63
frac0:63 ← (FRB)

If frac0:63 = 0 then go to Zero Operand

If sign = 1 then frac0:63 ← ¬frac0:63 + 1

Do while frac0 = 0   /* do the loop 0 times if (FRB) = maximum negative integer */
      frac0:63 ← frac1:63 || 0b0
      exp ← exp - 1
End

Round Float(sign,exp,frac0:63,FPSCRRN)

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 FI ← 0b00
FPSCRFPRF ← "+ zero"
FRT ← 0x0000_0000_0000_0000
Done
```

Round Float(sign,exp,frac$_{0:63}$,round_mode):

```
inc ← 0
lsb ← frac52
gbit ← frac53
rbit ← frac54
xbit ← frac55:63 > 0
If round_mode = 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 round_mode = 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 round_mode = 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
frac0:52 ← 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 */
$frac_{0:52}$ ← 0b1 || $(FRB)_{12:63}$
gbit || rbit || xbit ← 0b000

Do i = 1, 52 - exp
    $frac_{0:52}$ || gbit || rbit || xbit ← 0b0 || $frac_{0:52}$ || gbit || (rbit | xbit)
End

Round Integer (sign, $frac_{0:52}$, gbit, rbit, xbit)

Do i = 2, 52 - exp
    $frac_{0:52}$ ← $frac_{1:52}$ || 0b0
End

If $frac_0$ = 1, then exp ← exp + 1
Else $frac_{0:52}$ ← $frac_{1:52}$ || 0b0

FRT_0 ← sign
$FRT_{1:11}$ ← exp + 1023
$FRT_{12:63}$ ← $frac_{1:52}$

If $(FRT)_0$ = 0 then $FPSCR_{FPRF}$ ← "+ normal number"
Else $FPSCR_{FPRF}$ ← "- normal number"
$FPSCR_{FR\ FI}$ ← 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 || $frac_{52}$ || gbit || rbit || xbit = 0buu1uu then inc ← 1
    End
If inst = Floating Round to Integer Plus then
    Do /* comparisons ignore u bits */
        If sign || $frac_{52}$ || gbit || rbit || xbit = 0b0u1uu then inc ← 1
        If sign || $frac_{52}$ || gbit || rbit || xbit = 0b0uu1u then inc ← 1
        If sign || $frac_{52}$ || gbit || rbit || xbit = 0b0uuu1 then inc ← 1
    End
If inst = Floating Round to Integer Minus then
    Do /* comparisons ignore u bits */
        If sign || $frac_{52}$ || gbit || rbit || xbit = 0b1u1uu then inc ← 1
        If sign || $frac_{52}$ || gbit || rbit || xbit = 0b1uu1u then inc ← 1
        If sign || $frac_{52}$ || gbit || rbit || xbit = 0b1uuu1 then inc ← 1
    End
$frac_{0:52}$ ← $frac_{0:52}$ + 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. Vector RTL Functions [Category: Vector]

```
ConvertSPtoSXWsaturate( X, Y )
   sign    = X_0
   exp_{0:7}   = X_{1:8}
   frac_{0:30} = X_{9:31} || 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 )
   if((exp+Y-127)>30) then do // large operand
         VSCR_SAT = 1
         return( (sign==1) ? 0x8000_0000 : 0x7FFF_FFFF )
   if((exp+Y-127)<0) then return(0x0000_0000) // -1.0 < value < 1.0 (value rounds to 0)
   significand_{0:31} = 0b1 || frac
   do i=1 to 31-(exp+Y-127)
         significand = significand >>_ui 1
   return( (sign==0) ? significand : (¬significand + 1) )

ConvertSPtoUXWsaturate( X, Y )
   sign    = X_0
   exp_{0:7}   = X_{1:8}
   frac_{0:30} = X_{9:31} || 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 )
   if((exp+Y-127)>31) then do // large operand
      VSCR_SAT = 1
      return( (sign==1) ? 0x0000_0000 : 0xFFFF_FFFF )
   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)
   significand_{0:31} = 0b1 || frac
   do i=1 to 31-(exp+Y-127)
      significand = significand >>_ui 1
   return( significand )

ConvertSXWtoSP( X )
   sign    = X_0
   exp_{0:7}   = 32 + 127
   frac_{0:32} = X_0 || X_{0:31}
   if( frac==0 ) return( 0x0000_0000 ) // Zero operand
   if( sign==1 ) then frac = ¬frac + 1
   do while( frac_0==0 )
      frac = frac << 1
      exp = exp - 1
   lsb = frac_23
   gbit = frac_24
   xbit = frac_{25:32}!=0
   inc = ( lsb && gbit ) | ( gbit && xbit )
   frac_{0:23} = frac_{0:23} + inc
   if( carry_out==1 ) exp = exp + 1
   return( sign || exp || frac_{1:23} )
```

```
ConvertUXWtoSP( X )
   exp0:7 = 31 + 127
   frac0:31 = X0:31
   if( frac==0 ) return( 0x0000_0000 ) // Zero Operand
   do while( frac0==0 )
      frac = frac << 1
      exp = exp - 1
   lsb = frac23
   gbit = frac24
   xbit = frac25:31!=0
   inc = ( lsb && gbit ) | ( gbit && xbit )
   frac0:23 = frac0:23 + inc
   if( carry_out==1 ) exp = exp + 1
   return( 0b0 || exp || frac1:23 )
```

Appendix C. Embedded Floating-Point RTL Functions

[Category: SPE.Embedded Float Scalar Double]
[Category: SPE.Embedded Float Scalar Single]
[Category: SPE.Embedded Float Vector]

C.1 Common Functions

```
// Check if 32-bit fp value is a NaN or Infinity
Isa32NaNorInfinity(fp)
return (fp_exp = 255)

Isa32NaN(fp)
return ((fp_exp = 255) & (fp_frac ≠ 0))

// Check if 32-bit fp value is denormalized
Isa32Denorm(fp)
return ((fp_exp = 0) & (fp_frac ≠ 0))

// Check if 64-bit fp value is a NaN or Infinity
Isa64NaNorInfinity(fp)
return (fp_exp = 2047)

Isa64NaN(fp)
return ((fp_exp = 2047) & (fp_frac ≠ 0))

// Check if 32-bit fp value is denormalized
Isa64Denorm(fp)
return ((fp_exp = 0) & (fp_frac ≠ 0))

// Signal an error in the SPEFSCR
SignalFPError(upper_lower, bits)
if (upper_lower = HI) then
   bits ← bits << 15
SPEFSCR ← SPEFSCR | bits
bits ← (FG | FX)
if (upper_lower = HI) then
   bits ← bits << 15
SPEFSCR ← SPEFSCR & ¬bits
```

```
// Round a 32-bit fp result
Round32(fp, guard, sticky)

FP32format fp;
if (SPEFSCR_FINXE = 0) then
   if (SPEFSCR_FRMC = 0b00) then    // nearest
      if (guard) then
         if (sticky | fp_frac[22]) then
            v_0:23 ← fp_frac + 1
            if v_0 then
               if (fp_exp >= 254) then
                  // overflow
                  fp ← fp_sign || 0b11111110 || ^23 1
               else
                  fp_exp ← fp_exp + 1
                  fp_frac ← v_1:23
            else
               fp_frac ← v_1:23
   else if ((SPEFSCR_FRMC & 0b10) = 0b10) then
      // infinity modes
      // implementation dependent
return fp

// Round a 64-bit fp result
Round64(fp, guard, sticky)

FP32format fp;
if (SPEFSCR_FINXE = 0) then
   if (SPEFSCR_FRMC = 0b00) then    // nearest
      if (guard) then
         if (sticky | fp_frac[51]) then
            v_0:52 ← fp_frac + 1
            if v_0 then
               if (fp_exp >= 2046) then
                  // overflow
                  fp ← fp_sign ||
                        0b11111111110 || ^52 1
               else
                  fp_exp ← fp_exp + 1
                  fp_frac ← v_1:52
            else
               fp_frac ← v_1:52
   else if ((SPEFSCR_FRMC & 0b10) = 0b10) then
      // infinity modes
      // implementation dependent
return fp
```

C.2 Convert from Single-Precision Embedded Floating-Point to Integer Word with Saturation

```
// Convert 32-bit Floating-Point to 32-bit integer
// or fractional
//   signed = S (signed) or U (unsigned)
//   upper_lower = HI (high word) or LO (low word)
//   round = RND (round) or ZER (truncate)
//   fractional = F (fractional) or I (integer)

CnvtFP32ToI32Sat(fp, signed,
upper_lower, round, fractional)

FP32format fp;
if (Isa32NaNorInfinity(fp)) then
   SignalFPError(upper_lower, FINV)
   if (Isa32NaN(fp)) then
      return 0x00000000    // all NaNs
   if (signed = S) then
      if (fp_sign = 1) then
         return 0x80000000
      else
         return 0x7fffffff
   else
      if (fp_sign = 1) then
         return 0x00000000
      else
         return 0xffffffff
if (Isa32Denorm(fp)) then
   SignalFPError(upper_lower, FINV)
   return 0x00000000    // regardless of sign
if ((signed = U) & (fp_sign = 1)) then
   SignalFPError(upper_lower, FOVF)  // overflow
   return 0x00000000
if ((fp_exp = 0) & (fp_frac = 0)) then
   return 0x00000000    // all zero values
if (fractional = I) then  // convert to integer
   max_exp ← 158
   shift ← 158 - fp_exp
   if (signed = S) then
      if ((fp_exp≠158)|(fp_frac≠0)|(fp_sign≠1)) then
         max_exp ← max_exp - 1
else  // fractional conversion
   max_exp ← 126
   shift ← 126 - fp_exp
   if (signed = S) then
      shift ← shift + 1
if (fp_exp > max_exp) then
   SignalFPError(upper_lower, FOVF)  // overflow
   if (signed = S) then
      if (fp_sign = 1) then
         return 0x80000000
      else
         return 0x7fffffff
   else
      return 0xffffffff

result ← 0b1 || fp_frac || 0b00000000  // add U bit
guard ← 0
sticky ← 0
for (n ← 0; n < shift; n ← n + 1) do
   sticky ← sticky | guard
   guard ← result & 0x00000001
   result ← result > 1
// Report sticky and guard bits
if (upper_lower = HI) then
   SPEFSCR_FGH ← guard
   SPEFSCR_FXH ← sticky
else
   SPEFSCR_FG ← guard
   SPEFSCR_FX ← sticky

if (guard | sticky) then
   SPEFSCR_FINXS ← 1
// Round the integer result
if ((round = RND) & (SPEFSCR_FINXE = 0)) then
   if (SPEFSCR_FRMC = 0b00) then    // nearest
      if (guard) then
         if (sticky | (result & 0x00000001)) then
            result ← result + 1
   else if ((SPEFSCR_FRMC & 0b10) = 0b10) then
      // infinity modes
      // implementation dependent
if (signed = S) then
   if (fp_sign = 1) then
      result ← ¬result + 1
return result
```

C.3 Convert from Double-Precision Embedded Floating-Point to Integer Word with Saturation

```
// Convert 64-bit Floating-Point to 32-bit integer
// or fractional
//   signed = S (signed) or U (unsigned)
//   round = RND (round) or ZER (truncate)
//   fractional = F (fractional) or I (integer)
```

CnvtFP64ToI32Sat(fp, signed, round, fractional)

```
FP64format fp;

if (Isa64NaNorInfinity(fp)) then
   SignalFPError(LO, FINV)
   if (Isa64NaN(fp)) then
      return 0x00000000    // all NaNs
   if (signed = S) then
      if (fp_sign = 1) then
         return 0x80000000
      else
         return 0x7fffffff
   else
      if (fp_sign = 1) then
         return 0x00000000
      else
         return 0xffffffff

if (Isa64Denorm(fp)) then
   SignalFPError(LO, FINV)
   return 0x00000000    // regardless of sign
if ((signed = U) & (fp_sign = 1)) then
   SignalFPError(LO, FOVF)  // overflow
   return 0x00000000
if ((fp_exp = 0) & (fp_frac = 0)) then
   return 0x00000000    // all zero values
if (fractional = I) then  // convert to integer
   max_exp ← 1054
   shift ← 1054 - fp_exp
   if (signed ← S) then
      if ((fp_exp≠1054)|(fp_frac≠0)|(fp_sign≠1)) then
         max_exp ← max_exp - 1
else                       // fractional conversion
   max_exp ← 1022
   shift ← 1022 - fp_exp
   if (signed = S) then
      shift ← shift + 1

if (fp_exp > max_exp) then
   SignalFPError(LO, FOVF)  // overflow
   if (signed = S) then
      if (fp_sign = 1) then
         return 0x80000000
      else
         return 0x7fffffff
   else
      return 0xffffffff

result ← 0b1 || fp_frac[0:30]  // add U to frac
guard ← fp_frac[31]
sticky ← (fp_frac[32:63] ≠ 0)
for (n ← 0; n < shift; n ← n + 1) do
   sticky ← sticky | guard
   guard ← result & 0x00000001
   result ← result > 1
// Report sticky and guard bits

SPEFSCR_FG ← guard
SPEFSCR_FX ← sticky

if (guard | sticky) then
   SPEFSCR_FINXS ← 1
// Round the result
if ((round = RND) & (SPEFSCR_FINXE = 0)) then
   if (SPEFSCR_FRMC = 0b00) then    // nearest
      if (guard) then
         if (sticky | (result & 0x00000001)) then
            result ← result + 1
   else if ((SPEFSCR_FRMC & 0b10) = 0b10) then
      // infinity modes
      // implementation dependent
if (signed = S) then
   if (fp_sign = 1) then
      result ← ¬result + 1
return result
```

C.4 Convert from Double-Precision Embedded Floating-Point to Integer Doubleword with Saturation

```
// Convert 64-bit Floating-Point to 64-bit integer
//   signed = S (signed) or U (unsigned)
//   round = RND (round) or ZER (truncate)

CnvtFP64ToI64Sat(fp, signed, round)
FP64format fp;
if (Isa64NaNorInfinity(fp)) then
   SignalFPError(LO, FINV)
   if (Isa64NaN(fp)) then
      return 0x00000000_00000000    // all NaNs
   if (signed = S) then
      if (fp_sign = 1) then
         return 0x80000000_00000000
      else
         return 0x7fffffff_ffffffff
   else
      if (fp_sign = 1) then
         return 0x00000000_00000000
      else
         return 0xffffffff_ffffffff

if (Isa64Denorm(fp)) then
   SignalFPError(LO, FINV)
   return 0x00000000_00000000

if ((signed = U) & (fp_sign = 1)) then
   SignalFPError(LO, FOVF)  // overflow
   return 0x00000000_00000000
if ((fp_exp = 0) & (fp_frac = 0)) then
   return 0x00000000_00000000    // all zero values

max_exp ← 1086
shift ← 1086 - fp_exp
if (signed = S) then
   if ((fp_exp≠1086)|(fp_frac≠0)|(fp_sign≠1)) then
      max_exp ← max_exp - 1

if (fp_exp > max_exp) then
   SignalFPError(LO, FOVF)  // overflow
   if (signed = S) then
      if (fp_sign = 1) then
         return 0x80000000_00000000
      else
         return 0x7fffffff_ffffffff
   else
      return 0xffffffff_ffffffff

result ← 0b1 || fp_frac || 0b00000000000 //add U bit
guard ← 0
sticky ← 0
for (n ← 0; n < shift; n ← n + 1) do
   sticky ← sticky | guard
   guard ← result & 0x00000000_00000001
   result ← result > 1
// Report sticky and guard bits
SPEFSCR_FG ← guard
SPEFSCR_FX ← sticky
if (guard | sticky) then
   SPEFSCR_FINXS ← 1
// Round the result
if ((round = RND) & (SPEFSCR_FINXE = 0)) then
   if (SPEFSCR_FRMC = 0b00) then    // nearest
      if (guard) then
         if (sticky | (result&0x00000000_00000001))
         then
            result ← result + 1
   else if ((SPEFSCR_FRMC & 0b10) = 0b10) then
      // infinity modes
      // implementation dependent
if (signed = S) then
   if (fp_sign = 1) then
      result ← ¬result + 1
return result
```

C.5 Convert to Single-Precision Embedded Floating-Point from Integer Word

```
// Convert from 32-bit integer or fractional to
// 32-bit Floating-Point
//   signed = S (signed) or U (unsigned)
//   round = RND (round) or ZER (truncate)
//   fractional = F (fractional) or I (integer)
CnvtI32ToFP32(v, signed, upper_lower,
fractional)
FP32format result;
result_{sign} ← 0
if (v = 0) then
   result ← 0
   if (upper_lower = HI) then
      SPEFSCR_{FGH} ← 0
      SPEFSCR_{FXH} ← 0
   else
      SPEFSCR_{FG} ← 0
      SPEFSCR_{FX} ← 0
else
   if (signed = S) then
      if (v_{0} = 1) then
         v ← ¬v + 1
         result_{sign} ← 1
   if (fractional = F) then    // frac bit align
      maxexp ← 127
      if (signed = U) then
         maxexp ← maxexp - 1
   else
      maxexp ← 158    // integer bit alignment
   sc ← 0
   while (v_{0} = 0)
      v ← v << 1
      sc ← sc + 1
   v_{0} ← 0    // clear U bit
   result_{exp} ← maxexp - sc
   guard ← v_{24}
   sticky ← (v_{25:31} ≠ 0)

   // Report sticky and guard bits
   if (upper_lower = HI) then
      SPEFSCR_{FGH} ← guard
      SPEFSCR_{FXH} ← sticky
   else
      SPEFSCR_{FG} ← guard
      SPEFSCR_{FX} ← sticky

   if (guard | sticky) then
      SPEFSCR_{FINXS} ← 1
// Round the result

   result_{frac} ← v_{1:23}
   result ← Round32(result, guard, sticky)
return result
```

C.6 Convert to Double-Precision Embedded Floating-Point from Integer Word

```
// Convert from integer or fractional to 64 bit
// Floating-Point
//   signed = S (signed) or U (unsigned)
//   fractional = F (fractional) or I (integer)
CnvtI32ToFP64(v, signed, fractional)
FP64format result;
result_{sign} ← 0
if (v = 0) then
   result ← 0
   SPEFSCR_{FG} ← 0
   SPEFSCR_{FX} ← 0
else
   if (signed = S) then
      if (v_{0} = 1) then
         v ← ¬v + 1
         result_{sign} ← 1
   if (fractional = F) then    // frac bit align
      maxexp ← 1023
      if (signed = U) then
         maxexp ← maxexp - 1
   else
      maxexp ← 1054    // integer bit align
   sc ← 0
   while (v_{0} = 0)
      v ← v << 1
      sc ← sc + 1
   v_{0} ← 0    // clear U bit
   result_{exp} ← maxexp - sc

// Report sticky and guard bits

   SPEFSCR_{FG} ← 0
   SPEFSCR_{FX} ← 0

   result_{frac} ← v_{1:31} || ^{21}0
return result
```

C.7 Convert to Double-Precision Embedded Floating-Point from Integer Doubleword

```
// Convert from 64-bit integer to 64-bit
// floating-point
//   signed = S (signed) or U (unsigned)
CnvtI64ToFP64(v, signed)
FP64format result;
result_sign ← 0
if (v = 0) then
   result ← 0
   SPEFSCR_FG ← 0
   SPEFSCR_FX ← 0
else
   if (signed = S) then
      if (v_0 = 1) then
         v ← ¬v + 1
         result_sign ← 1
   maxexp ← 1054
   sc ← 0
   while (v_0 = 0)
      v ← v << 1
      sc ← sc + 1
   v_0 ← 0    // clear U bit
   result_exp ← maxexp - sc
   guard ← v_53
   sticky ← (v_54:63 ≠ 0)

// Report sticky and guard bits

   SPEFSCR_FG ← guard
   SPEFSCR_FX ← sticky
   if (guard | sticky) then
      SPEFSCR_FINXS ← 1
// Round the result

   result_frac ← v_1:52
   result ← Round64(result, guard, sticky)

return result
```

Appendix D. 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.

D.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 D.2.2 and D.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 D.2.3 and D.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 D.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.)

D.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 D.2.2 - D.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.

D.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 D.2.2 - D.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.

D.2.2 Simple Branch Mnemonics

Instructions using one of the mnemonics in Table 11 that tests a Condition Register bit specify the corresponding bit as the first operand. The symbols defined in Section D.1 can be used in this operand.

Notice that there are no extended mnemonics for relative and absolute unconditional branches. For these the basic mnemonics ***b***, ***ba***, ***bl***, and ***bla*** should be used.

Table 11: Simple branch mnemonics

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	-

Examples

1. Decrement CTR and branch if it is still nonzero (closure of a loop controlled by a count loaded into CTR).

```
bdnz    target              (equivalent to:   bc    16,0,target)
```

2. Same as (1) but branch only if CTR is nonzero and condition in CR0 is “equal”.

```
bdnzt   eq,target           (equivalent to:   bc    8,2,target)
```

3. Same as (2), but “equal” condition is in CR5.

```
bdnzt   4×cr5+eq,target     (equivalent to:   bc    8,22,target)
```

4. Branch if bit 59 of CR is 0.

   bf 27,target (equivalent to: bc 4,27,target)

5. Same as (4), but set the Link Register. This is a form of conditional “call”.

   bfl 27,target (equivalent to: bcl 4,27,target)

D.2.3 Branch Mnemonics Incorporating Conditions

In the mnemonics defined in Table 12, the test of a bit in a Condition Register field is encoded in the mnemonic.

Instructions using the mnemonics in Table 12 specify the CR field as an optional first operand. One of the CR field symbols defined in Section D.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 12.

Table 12: Branch mnemonics incorporating conditions

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

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)

D.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-
```

D.3 Condition Register Logical Mnemonics

The *Condition Register Logical* instructions can be used to set (to 1), clear (to 0), copy, or invert a given Condition Register bit. Extended mnemonics are provided that allow these operations to be coded easily.

Table 13: Condition Register logical mnemonics

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

The symbols defined in Section D.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)

D.4 Subtract Mnemonics

D.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)
```

D.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)
```

D.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 D.1 can be used for this operand.

Note: The basic *Compare* mnemonics of Power ISA are the same as those of POWER, but the POWER instructions have three operands while the Power ISA instructions have four. The Assembler will recognize a basic *Compare* mnemonic with three operands as the POWER form, 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.)

D.5.1 Doubleword Comparisons

Table 14: Doubleword compare mnemonics

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

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)

D.5.2 Word Comparisons

Table 15: Word compare mnemonics

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

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)

D.6 Trap Mnemonics

The mnemonics defined in Table 16 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 16.

Table 16: Trap mnemonics

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

Examples

1. Trap if register Rx is not 0.

   tdnei Rx,0 (equivalent to: tdi 24,Rx,0)

2. Same as (1), but comparison is to register Ry.

   tdne Rx,Ry (equivalent to: td 24,Rx,Ry)

3. Trap if bits 32:63 of register Rx, considered as a 32-bit quantity, are logically greater than 0x7FF.

   twlgti Rx,0x7FF (equivalent to: twi 1,Rx,0x7FF)

4. Trap unconditionally.

   trap (equivalent to: tw 31,0,0)

5. Trap unconditionally with immediate parameters Rx and Ry

   tdu Rx,Ry (equivalent to: td 31,Rx,Ry)

D.7 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.

D.7.1 Operations on Doublewords

All these mnemonics can be coded with a final "." to cause the Rc bit to be set in the underlying instruction.

Table 17: Doubleword rotate and shift mnemonics

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 <= b < 64)	rldic ra,rs,n,b - n

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)

D.7.2 Operations on Words

All these mnemonics can be coded with a final "." to cause the Rc bit to be set in the underlying instruction. The operations as described above apply to the low-order 32 bits of the registers, as if the registers were 32-bit registers. The Insert operations either preserve the high-order 32 bits of the target register or place rotated data there; the other operations clear these bits.

Table 18: Word rotate and shift mnemonics

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 ≤ b < 32)	rlwinm ra,rs,n,b-n,31-n

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)

D.8 Move To/From Special Purpose Register Mnemonics

The ***mtspr*** and ***mfspr*** instructions specify a Special Purpose Register (SPR) as a numeric operand. Extended mnemonics are provided that represent the SPR in the mnemonic rather than requiring it to be coded as an operand.

Table 19: Extended mnemonics for moving to/from an SPR

Special Purpose Register	Move To SPR		Move From SPR	
	Extended	**Equivalent to**	**Extended**	**Equivalent to**
Fixed-Point Exception Register (XER)	mtxer Rx	mtspr 1,Rx	mfxer Rx	mfspr Rx,1
Link Register (LR)	mtlr Rx	mtspr 8,Rx	mflr Rx	mfspr Rx,8
Count Register (CTR)	mtctr Rx	mtspr 9,Rx	mfctr Rx	mfspr Rx,9
PPR	mtppr Rx	mtspr 896,Rx	mfppr Rx	mfspr Rx,896

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)

D.9 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)

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 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.

Appendix E. Programming Examples

E.1 Multiple-Precision Shifts

This section gives examples of how multiple-precision shifts can be programmed.

A multiple-precision shift is defined to be a shift of an N-doubleword quantity (64-bit mode) or an N-word quantity (32-bit mode), where N>1. The quantity to be shifted is contained in N registers. The shift amount is specified either by an immediate value in the instruction, or by a value in a register.

The examples shown below distinguish between the cases N=2 and N>2. If N=2, the shift amount may be in the range 0 through 127 (64-bit mode) or 0 through 63 (32-bit mode), which are the maximum ranges supported by the *Shift* instructions used. However if N>2, the shift amount must be in the range 0 through 63 (64-bit mode) or 0 through 31 (32-bit mode), in order for the examples to yield the desired result. The specific instance shown for N>2 is N=3; extending those code sequences to larger N is straightforward, as is reducing them to the case N=2 when the more stringent restriction on shift amount is met. For shifts with immediate shift amounts only the case N=3 is shown, because the more stringent restriction on shift amount is always met.

In the examples it is assumed that GPRs 2 and 3 (and 4) contain the quantity to be shifted, and that the result is to be placed into the same registers, except for the immediate left shifts in 64-bit mode for which the result is placed into GPRs 3, 4, and 5. In all cases, for both input and result, the lowest-numbered register contains the highest-order part of the data and highest-numbered register contains the lowest-order part. For non-immediate shifts, the shift amount is assumed to be in GPR 6. For immediate shifts, the shift amount is assumed to be greater than 0. GPRs 0 and 31 are used as scratch registers.

For N>2, the number of instructions required is 2N-1 (immediate shifts) or 3N-1 (non-immediate shifts).

Multiple-precision shifts in 64-bit mode [Category: 64-Bit]

Shift Left Immediate, N = 3 (shift amnt < 64)

```
rldicr   r5,r4,sh,63-sh
rldimi   r4,r3,0,sh
rldicl   r4,r4,sh,0
rldimi   r3,r2,0,sh
rldicl   r3,r3,sh,0
```

Shift Left, N = 2 (shift amnt < 128)

```
subfic   r31,r6,64
sld      r2,r2,r6
srd      r0,r3,r31
or       r2,r2,r0
addi     r31,r6,-64
sld      r0,r3,r31
or       r2,r2,r0
sld      r3,r3,r6
```

Shift Left, N = 3 (shift amnt < 64)

```
subfic   r31,r6,64
sld      r2,r2,r6
srd      r0,r3,r31
or       r2,r2,r0
sld      r3,r3,r6
srd      r0,r4,r31
or       r3,r3,r0
sld      r4,r4,r6
```

Shift Right Immediate, N = 3 (shift amnt < 64)

```
rldimi   r4,r3,0,64-sh
rldicl   r4,r4,64-sh,0
rldimi   r3,r2,0,64-sh
rldicl   r3,r3,64-sh,0
rldicl   r2,r2,64-sh,sh
```

Shift Right, N = 2 (shift amnt < 128)

```
subfic   r31,r6,64
srd      r3,r3,r6
sld      r0,r2,r31
or       r3,r3,r0
addi     r31,r6,-64
srd      r0,r2,r31
or       r3,r3,r0
srd      r2,r2,r6
```

Shift Right, N = 3 (shift amnt < 64)

```
subfic   r31,r6,64
srd      r4,r4,r6
sld      r0,r3,r31
or       r4,r4,r0
srd      r3,r3,r6
sld      r0,r2,r31
or       r3,r3,r0
srd      r2,r2,r6
```

Multiple-precision shifts in 32-bit mode

Shift Left Immediate, N = 3 (shift amnt < 32)

```
rlwinm   r2,r2,sh,0,31-sh
rlwimi   r2,r3,sh,32-sh,31
rlwinm   r3,r3,sh,0,31-sh
rlwimi   r3,r4,sh,32-sh,31
rlwinm   r4,r4,sh,0,31-sh
```

Shift Left, N = 2 (shift amnt < 64)

```
subfic   r31,r6,32
slw      r2,r2,r6
srw      r0,r3,r31
or       r2,r2,r0
addi     r31,r6,-32
slw      r0,r3,r31
or       r2,r2,r0
slw      r3,r3,r6
```

Shift Left, N = 3 (shift amnt < 32)

```
subfic   r31,r6,32
slw      r2,r2,r6
srw      r0,r3,r31
or       r2,r2,r0
slw      r3,r3,r6
srw      r0,r4,r31
or       r3,r3,r0
slw      r4,r4,r6
```

Shift Right Immediate, N = 3 (shift amnt < 32)

```
rlwinm   r4,r4,32-sh,sh,31
rlwimi   r4,r3,32-sh,0,sh-1
rlwinm   r3,r3,32-sh,sh,31
rlwimi   r3,r2,32-sh,0,sh-1
rlwinm   r2,r2,32-sh,sh,31
```

Shift Right, N = 2 (shift amnt < 64)

```
subfic   r31,r6,32
srw      r3,r3,r6
slw      r0,r2,r31
or       r3,r3,r0
addi     r31,r6,-32
srw      r0,r2,r31
or       r3,r3,r0
srw      r2,r2,r6
```

Shift Right, N = 3 (shift amnt < 32)

```
subfic   r31,r6,32
srw      r4,r4,r6
slw      r0,r3,r31
or       r4,r4,r0
srw      r3,r3,r6
slw      r0,r2,r31
or       r3,r3,r0
srw      r2,r2,r6
```

Multiple-precision shifts in 64-bit mode, continued [Category: 64-Bit]

Shift Right Algebraic Immediate, N = 3 (shift amnt < 64)

```
rldimi   r4,r3,0,64-sh
rldicl   r4,r4,64-sh,0
rldimi   r3,r2,0,64-sh
rldicl   r3,r3,64-sh,0
sradi    r2,r2,sh
```

Shift Right Algebraic, N = 2 (shift amnt < 128)

```
subfic   r31,r6,64
srd      r3,r3,r6
sld      r0,r2,r31
or       r3,r3,r0
addic.   r31,r6,-64
srad     r0,r2,r31
ble      $+8
ori      r3,r0,0
srad     r2,r2,r6
```

Shift Right Algebraic, N = 3 (shift amnt < 64)

```
subfic   r31,r6,64
srd      r4,r4,r6
sld      r0,r3,r31
or       r4,r4,r0
srd      r3,r3,r6
sld      r0,r2,r31
or       r3,r3,r0
srad     r2,r2,r6
```

Multiple-precision shifts in 32-bit mode, continued

Shift Right Algebraic Immediate, N = 3 (shift amnt < 32)

```
rlwinm   r4,r4,32-sh,sh,31
rlwimi   r4,r3,32-sh,0,sh-1
rlwinm   r3,r3,32-sh,sh,31
rlwimi   r3,r2,32-sh,0,sh-1
srawi    r2,r2,sh
```

Shift Right Algebraic, N = 2 (shift amnt < 64)

```
subfic   r31,r6,32
srw      r3,r3,r6
slw      r0,r2,r31
or       r3,r3,r0
addic.   r31,r6,-32
sraw     r0,r2,r31
ble      $+8
ori      r3,r0,0
sraw     r2,r2,r6
```

Shift Right Algebraic, N = 3 (shift amnt < 32)

```
subfic   r31,r6,32
srw      r4,r4,r6
slw      r0,r3,r31
or       r4,r4,r0
srw      r3,r3,r6
slw      r0,r2,r31
or       r3,r3,r0
sraw     r2,r2,r6
```

E.2 Floating-Point Conversions [Category: Floating-Point]

This section gives examples of how the *Floating-Point Conversion* instructions can be used to perform various conversions.

Warning: Some of the examples use the ***fsel*** instruction. Care must be taken in using ***fsel*** if IEEE compatibility is required, or if the values being tested can be NaNs or infinities; see Section E.3.4, “Notes” on page 336.

E.2.1 Conversion from Floating-Point Number to Floating-Point Integer

The full *convert to floating-point integer* function can be implemented with the sequence shown below, assuming the floating-point value to be converted is in FPR 1 and the result is returned in FPR 3.

```
mtfsb0    23          #clear VXCVI
fctid[z]  f3,f1       #convert to fx int
fcfid     f3,f3       #convert back again
mcrfs     7,5         #VXCVI to CR
bf        31,$+8      #skip if VXCVI was 0
fmr       f3,f1       #input was fp int
```

E.2.2 Conversion from Floating-Point Number to Signed Fixed-Point Integer Doubleword

The full *convert to signed fixed-point integer doubleword* function can be implemented with the sequence shown below, assuming the floating-point value to be converted is in FPR 1, the result is returned in GPR 3, and a doubleword at displacement “disp” from the address in GPR 1 can be used as scratch space.

```
fctid[z]  f2,f1         #convert to dword int
stfd      f2,disp(r1)   #store float
ld        r3,disp(r1)   #load dword
```

E.2.3 Conversion from Floating-Point Number to Unsigned Fixed-Point Integer Doubleword

The full *convert to unsigned fixed-point integer doubleword* function can be implemented with the sequence shown below, assuming the floating-point value to be converted is in FPR 1, the value 0 is in FPR 0, the value 2^{64}-2048 is in FPR 3, the value 2^{63} is in FPR 4 and GPR 4, the result is returned in GPR 3, and a doubleword at displacement “disp” from the address in GPR 1 can be used as scratch space.

```
fsel      f2,f1,f1,f0   #use 0 if < 0
fsub      f5,f3,f1      #use max if > max
fsel      f2,f5,f2,f3
fsub      f5,f2,f4      #subtract 2^63
fcmpu     cr2,f2,f4     #use diff if >= 2^63
fsel      f2,f5,f5,f2
fctid[z]  f2,f2         #convert to fx int
stfd      f2,disp(r1)   #store float
ld        r3,disp(r1)   #load dword
blt       cr2,$+8       #add 2^63 if input
add       r3,r3,r4      #  was >= 2^63
```

E.2.4 Conversion from Floating-Point Number to Signed Fixed-Point Integer Word

The full *convert to signed fixed-point integer word* function can be implemented with the sequence shown below, assuming the floating-point value to be converted is in FPR 1, the result is returned in GPR 3, and a doubleword at displacement “disp” from the address in GPR 1 can be used as scratch space.

```
fctiw[z]  f2,f1           #convert to fx int
stfd      f2,disp(r1)     #store float
lwa       r3,disp+4(r1)   #load word algebraic
```

E.2.5 Conversion from Floating-Point Number to Unsigned Fixed-Point Integer Word

The full *convert to unsigned fixed-point integer word* function can be implemented with the sequence shown below, assuming the floating-point value to be converted is in FPR 1, the value 0 is in FPR 0, the value 2^{32}-1 is in FPR 3, the result is returned in GPR 3, and a doubleword at displacement "disp" from the address in GPR 1 can be used as scratch space.

```
fsel      f2,f1,f1,f0   #use 0 if < 0
fsub      f4,f3,f1      #use max if > max
fsel      f2,f4,f2,f3
fctid[z]  f2,f2         #convert to fx int
stfd      f2,disp(r1)   #store float
lwz       r3,disp+4(r1) #load word and zero
```

E.2.6 Conversion from Signed Fixed-Point Integer Doubleword to Floating-Point Number

The full *convert from signed fixed-point integer doubleword* function, using the rounding mode specified by $FPSCR_{RN}$, can be implemented with the sequence shown below, assuming the fixed-point value to be converted is in GPR 3, the result is returned in FPR 1, and a doubleword at displacement "disp" from the address in GPR 1 can be used as scratch space.

```
std       r3,disp(r1)   #store dword
lfd       f1,disp(r1)   #load float
fcfid     f1,f1         #convert to fp int
```

E.2.7 Conversion from Unsigned Fixed-Point Integer Doubleword to Floating-Point Number

The full *convert from unsigned fixed-point integer doubleword* function, using the rounding mode specified by $FPSCR_{RN}$, can be implemented with the sequence shown below, assuming the fixed-point value to be converted is in GPR 3, the value 2^{32} is in FPR 4, the result is returned in FPR 1, and two doublewords at displacement "disp" from the address in GPR 1 can be used as scratch space.

```
rldicl    r2,r3,32,32   #isolate high half
rldicl    r0,r3,0,32    #isolate low half
std       r2,disp(r1)   #store dword both
std       r0,disp+8(r1)
lfd       f2,disp(r1)   #load float both
lfd       f1,disp+8(r1)
fcfid     f2,f2         #convert each half to
fcfid     f1,f1         #  fp int (exact result)
fmadd     f1,f4,f2,f1   #(2^32)×high + low
```

An alternative, shorter, sequence can be used if rounding according to $FSCPR_{RN}$ is desired and $FPSCR_{RN}$ specifies *Round toward +Infinity* or *Round toward -Infinity*, or if it is acceptable for the rounded answer to be either of the two representable floating-point integers nearest to the given fixed-point integer. In this case the full *convert from unsigned fixed-point integer doubleword* function can be implemented with the sequence shown below, assuming the value 2^{64} is in FPR 2.

```
std       r3,disp(r1)   #store dword
lfd       f1,disp(r1)   #load float
fcfid     f1,f1         #convert to fp int
fadd      f4,f1,f2      #add 2^64
fsel      f1,f1,f1,f4   #  if r3 < 0
```

E.2.8 Conversion from Signed Fixed-Point Integer Word to Floating-Point Number

The full *convert from signed fixed-point integer word* function can be implemented with the sequence shown below, assuming the fixed-point value to be converted is in GPR 3, the result is returned in FPR 1, and a doubleword at displacement "disp" from the address in GPR 1 can be used as scratch space. (The result is exact.)

```
extsw     r3,r3         #extend sign
std       r3,disp(r1)   #store dword
lfd       f1,disp(r1)   #load float
fcfid     f1,f1         #convert to fp int
```

E.2.9 Conversion from Unsigned Fixed-Point Integer Word to Floating-Point Number

The full *convert from unsigned fixed-point integer word* function can be implemented with the sequence shown below, assuming the fixed-point value to be converted is in GPR 3, the result is returned in FPR 1, and a doubleword at displacement "disp" from the address in GPR 1 can be used as scratch space. (The result is exact.)

```
rldicl    r0,r3,0,32    #zero-extend
std       r0,disp(r1)   #store dword
lfd       f1,disp(r1)   #load float
fcfid     f1,f1         #convert to fp int
```

E.3 Floating-Point Selection [Category: Floating-Point]

This section gives examples of how the *Floating Select* instruction can be used to implement floating-point minimum and maximum functions, and 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.

Additional examples can be found in Section E.2, "Floating-Point Conversions [Category: Floating-Point]" on page 334.

Warning: Care must be taken in using ***fsel*** if IEEE compatibility is required, or if the values being tested can be NaNs or infinities; see Section E.3.4.

E.3.1 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)

E.3.2 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)

E.3.3 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)

E.3.4 Notes

The following Notes apply to the preceding examples and to the corresponding cases using the other three arithmetic relations (<, ≤, and ≠). They should also be considered when any other use of ***fsel*** is contemplated.

In these Notes, the "optimized program" is the Power ISA program shown, and the "unoptimized program" (not shown) is the corresponding Power ISA program that uses ***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.

E.4 Vector Unaligned Storage Operations [Category: Vector]

E.4.1 Loading a Unaligned Quadword Using Permute from Big-Endian Storage

The following sequence of instructions copies the unaligned quadword storage operand into VRT.

```
# 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
perm     Vt,Vhi,Vlo,Vp # align the data
```

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".

- **processor**
  A hardware component that executes the instructions specified in a program.
- **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 operating system.
- **main storage**
  The level of storage hierarchy in which all storage state is visible to all processors and mechanisms in the system.
- **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 the section entitled "Instruction Execution Order" in 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.)
- **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 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 address translation or reference and change recording (see Book III-S).

■ **caused by, associated with**

- **caused by**

  A storage access is said to be caused by an instruction if the instruction is a *Load* or *Store* and the access (data access) is to the storage location specified by the instruction.

- **associated with**

  A storage access is said to be associated with an instruction if the access is for the purpose of fetching the instruction (instruction fetch), or is a data access caused by the instruction, or is an implicit access that occurs as a side effect of fetching or executing the instruction.

■ **prefetched instructions**
Instructions for which a copy of the instruction has been fetched from instruction storage, but the instruction has not yet been executed.

■ **uniprocessor**
A system that contains one processor.

■ **multiprocessor**
A system that contains two or more processors.

■ **shared storage multiprocessor**
A multiprocessor that contains some common storage, which all the processors in the system can access.

■ **performed**
A load or instruction fetch by a processor or mechanism (P1) is performed with respect to any processor or mechanism (P2) when the value to be returned by the load or instruction fetch can no longer be changed by a store by P2. A store by P1 is performed with respect to P2 when a load by P2 from the location accessed by the store will return the value stored (or a value stored subsequently). An instruction cache block invalidation by P1 is performed with respect to P2 when an instruction fetch by P2 will not be satisfied from the copy of the block that existed in its instruction cache when the instruction causing the invalidation was executed, 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 <S> 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.

■ **aligned storage access**
A load or store is aligned if the address of the target storage location is a multiple of the size of the transfer effected by the instruction.

1.2 Introduction

The Power ISA User Instruction Set Architecture, discussed in Book I, defines storage as a linear array of bytes indexed from 0 to a maximum of 2^{64}-1. Each byte is identified by its index, called its address, and each byte contains a value. This information is sufficient to allow the programming of applications that require no special features of any particular system environment. The Power ISA Virtual Environment Architecture, described herein, expands this simple storage model to include caches, virtual storage, and shared storage multiprocessors. The Power ISA Virtual Environment Architecture, in conjunction with services based on the Power ISA Operating Environment Architecture (see Book III) and provided by the operating system, permits explicit control of this expanded storage model. A simple model for sequential execution allows at most one storage access to be performed at a time and requires that all storage accesses appear to be performed in program order. In contrast to this simple model, the Power ISA specifies a relaxed model of storage consistency. In a multiprocessor system that allows multiple copies of a storage location, aggressive implementations of the architecture can permit intervals of time during which different copies of a storage location have different values. This chapter describes features of the Power ISA that enable programmers to write correct programs for this storage model.

1.3 Virtual Storage

The Power ISA system implements a virtual storage model for applications. This means that a combination of hardware and software can present a storage model that allows applications to exist within a “virtual” address space larger than either the effective address space or the real address space.

Each program can access 2^{64} bytes of "effective address" (EA) space, subject to limitations imposed by the operating system. In a typical Power ISA system, each program's EA space is a subset of a larger "virtual address" (VA) space managed by the operating system.

Each effective address is translated to a real address (i.e., to an address of a byte in real storage or on an I/O device) before being used to access storage. The hardware accomplishes this, using the address translation mechanism described in Book III. The operating system manages the real (physical) storage resources of the system, by setting up the tables and other information used by the hardware address translation mechanism.

In general, real storage may not be large enough to map all the virtual pages used by the currently active applications. With support provided by hardware, the operating system can attempt to use the available real pages to map a sufficient set of virtual pages of the applications. If a sufficient set is maintained, "paging" activity is minimized. If not, performance degradation is likely.

The operating system can support restricted access to virtual pages (including read/write, read only, and no access; see Book III), based on system standards (e.g., program code might be read only) and application requests.

1.4 Single-copy Atomicity

An access is *single-copy atomic*, or simply *atomic*, if it is always performed in its entirety with no visible fragmentation. Atomic accesses are thus serialized: each happens in its entirety in some order, even when that order is not specified in the program or enforced between processors.

Vector storage accesses are not guaranteed to be atomic. The following other types of single-register accesses are always atomic:

- byte accesses (all bytes are aligned on byte boundaries)
- halfword accesses aligned on halfword boundaries
- word accesses aligned on word boundaries
- doubleword accesses aligned on doubleword boundaries (64-bit implementations only; see Section 1.2 of Book III-E<E>)

No other accesses are 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 operand is unaligned
- ***lmw***, ***stmw***, ***lswi***, ***lswx***, ***stswi***, ***stswx***
- any *Cache Management* instruction

An access that is not atomic is performed as a set of smaller disjoint atomic accesses. The number and alignment of these accesses are implementation-dependent, as is the relative order in which they are performed.

The results for several combinations of loads and stores to the same or overlapping locations are described below.

1. When two processors execute 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 execute 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 execute 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 execute 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 executes an atomic store to a location, a second processor executes 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 executed simultaneously, and no other store is performed to that location, the value returned by the load is some combination of the contents of the location before the store and the contents of the location after the store.

1.5 Cache Model

A cache model in which there is one cache for instructions and another cache for data is called a "Harvard-style" cache. This is the model assumed by the Power ISA, e.g., in the descriptions of the *Cache Management* instructions in Section 3.2. 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***)
- <E> 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***)
- <E> allocate a data cache block and set the contents of that block to zeros, but perform no operation if no write access is allowed to the data cache block (***dcba***)
- 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***<S>)

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
- Endianness<E>

These attributes have meaning only when an effective address is translated by the processor performing the storage access.

<E> Additional storage control attributes may be defined for some implementations. See Section 4.8 of Book III-E for additional information.

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 ***lwarx***, ***ldarx***, ***stwcx***., and ***stdcx***. instructions may cause the system data storage error handler to be invoked if they specify a location in storage having either of these attributes.

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”, and similarly for “*Store* instruction”.

1.6.1 Write Through Required

A store to a Write Through Required storage location is performed in main storage. A Store instruction that specifies a location in Write Through Required storage may cause additional locations in main storage to be accessed. If a copy of the block containing the specified location is retained in the data cache, the store is also performed in the data cache. The store does not cause the block to be considered to be modified in the data cache.

In general, accesses caused by separate *Store* instructions that specify locations in Write Through Required storage may be combined into one access. Such combining does not occur if the *Store* instructions are separated by a ***sync***, ***eieio***<S>, or ***mbar***<E> 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*** or ***mbar***<E> instruction, or by an ***eieio***<S> instruction if the storage is also Guarded.

1.6.3 Memory Coherence Required [Category: Memory Coherence]

An access to a Memory Coherence Required storage location is performed coherently, as follows.

Memory coherence refers to the ordering of stores to a single location. Atomic stores to a given location are *coherent* if they are serialized in some order, and no processor or mechanism is able to observe any subset of those stores as occurring in a conflicting order. This serialization order is an abstract sequence of values; the physical storage location need not assume each of the values written to it. For example, a processor may update a location several times before the value is written to physical storage. The result of a store operation is not available to every processor or mechanism at the same instant, and it may be that a processor or mechanism observes only some of the values that are written to a location. However, when a location is accessed atomically and coherently by all processors and mechanisms, the sequence of values loaded from the location by any processor or mechanism during any interval of time forms a subsequence of the sequence of values that the location logically held during that interval. That is, a processor or mechanism can never load a “newer” value first and then, later, load an “older” value.

Memory coherence is managed in blocks called coherence *blocks*. Their size is implementation-dependent, but is larger than a word and is usually the size of a cache block.

For storage that is not Memory Coherence Required, software must explicitly manage memory coherence to the extent required by program correctness. The operations required to do this may be system-dependent.

Because the Memory Coherence Required attribute for a given storage location is of little use unless all processors that access the location do so coherently, in statements about Memory Coherence Required storage elsewhere in this document it is generally assumed that the storage has the Memory Coherence Required attribute for all processors that access it.

Programming Note

Operating systems that allow programs to request that storage not be Memory Coherence Required should provide services to assist in managing memory coherence for such storage, including all system-dependent aspects thereof.

In most systems the default is that all storage is Memory Coherence Required. For some applications in some systems, software management of coherence may yield better performance. In such cases, a program can request that a given unit of storage not be Memory Coherence Required, and can manage the coherence of that storage by using the ***sync*** instruction, the *Cache Management* instructions, and services provided by the operating system.

1.6.4 Guarded

A data access to a Guarded storage location is performed only if either (a) the access is caused by an instruction that is known to be required by the sequential execution model, or (b) the access is a load and the storage location is already in a cache. If the storage is also Caching Inhibited, only the storage location specified by the instruction is accessed; otherwise any storage location in the cache block containing the specified storage location may be accessed.

For the Server environment, instructions are not fetched from virtual storage that is Guarded. If the instruction addressed by the current instruction address is in such storage, the system instruction storage error handler may be invoked (see Section 6.5.5 of Book III-S).

Programming Note

In some implementations, instructions may be executed before they are known to be required by the sequential execution model. Because the results of instructions executed in this manner are discarded if it is later determined that those instructions would not have been executed in the sequential execution model, this behavior does not affect most programs.

This behavior does affect programs that access storage locations that are not “well-behaved” (e.g., a storage location that represents a control register on an I/O device that, when accessed, causes the device to perform an operation). To avoid unintended results, programs that access such storage locations should request that the storage be Guarded, and should prevent such storage locations from being in a cache (e.g., by requesting that the storage also be Caching Inhibited).

1.6.5 Endianness [Category: Embedded.Little-Endian]

The Endianness storage control attribute specifies the byte ordering (Big-Endian or Little-Endian) that is used when the storage location is accessed; see Section 1.10 of Book I.

1.6.6 Variable Length Encoded (VLE) Instructions

VLE storage is used to store VLE instructions. Instructions fetched from VLE storage are processed as VLE instructions. VLE storage must also be Big-Endian. Instructions fetched from VLE storage that is Little-Endian cause a Byte-ordering exception, and the system instruction storage error handler will be invoked.

The VLE attribute has no effect on data accesses. See Chapter 1 of Book VLE.

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 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*, ***eieio***<S>, or ***mbar***<E> 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***, ***lwsync***<S>, ***eieio***<S>, and ***mbar***<E> to order storage accesses can be found in Appendix B. "Programming Examples for Sharing Storage" on page 385.

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.

1.7.2 Storage Ordering of I/O Accesses

A “coherence domain” consists of all processors and all interfaces to main storage. Memory reads and writes initiated by mechanisms outside the coherence domain are performed within the coherence domain in the order in which they enter the coherence domain and are performed as coherent accesses.

1.7.3 Atomic Update

The *Load And Reserve* and *Store Conditiona*l instructions together permit atomic update of a shared storage location. There are word and doubleword forms of each of these instructions. Described here is the operation of the word forms ***lwarx*** and ***stwcx.***; operation of the doubleword forms ***ldarx*** and ***stdcx.*** 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 storage location specified by the ***lwarx*** that established the reservation has not been stored into by another processor or mechanism since the reservation was created. If the storage locations specified by the two instructions differ, the store is not necessarily performed.

A ***stwcx.*** that performs its store is said to “succeed”.

Examples of the use of ***lwarx*** and ***stwcx.*** are given in Appendix B. “Programming Examples for Sharing Storage” on page 385.

A successful ***stwcx.*** to a given location may complete before its store has been performed with respect to other processors and mechanisms. As a result, a subsequent load or ***lwarx*** from the given location by another processor may return a “stale” value. However, a subsequent ***lwarx*** from the given location by the other processor followed by a successful ***stwcx.*** by that processor is guaranteed to have returned the value stored by the first processor’s ***stwcx***. (in the absence of other stores to the given location).

> **Programming Note**
>
> The store caused by a successful ***stwcx.*** is ordered, by a dependence on the reservation, with respect to the load caused by the ***lwarx*** that established the reservation, such that the two storage accesses are performed in program order with respect to any processor or mechanism.

1.7.3.1 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 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). 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".)

A processor has at most one reservation at any time. A reservation is established by executing a ***lwarx*** or ***ldarx*** instruction, and is lost (or may be lost, in the case of the third, fifth, sixth and seventh item) if any of the following occur.

1. The processor holding the reservation executes another ***lwarx*** or ***ldarx***: this clears the first reservation and establishes a new one.
2. The processor holding the reservation executes any ***stwcx.*** or ***stdcx.***, regardless of whether the specified address matches the address specified by the ***lwarx*** or ***ldarx*** that established the reservation.
3. The processor holding the reservation executes a ***dcbf*** or ***dcbfl***<S> to the reservation granule: whether the reservation is lost is undefined.
4. Some other processor executes a *Store* or ***dcbz*** to the same reservation granule.
5. Some other processor executes a ***dcbtst***, ***dcbst***, ***dcbf*** (but not ***dcbfl***<S>) to the same reservation granule: whether the reservation is lost is undefined.
6. <E> Some other processor executes a ***dcba*** to the same reservation granule: the reservation is lost if the instruction causes the target block to be newly established in a data cache or to be modified; otherwise whether the reservation is lost is undefined.
7. Any processor modifies a Reference or Change bit (see Book III-S) in the same reservation granule: whether the reservation is lost is undefined.
8. Some mechanism other than a processor modifies a storage location in the same reservation granule.

For the Server environment, interrupts (see Book III-S) do not clear reservations (however, system software invoked by interrupts may clear reservations); for the Embedded environment, interrupts do not necessarily clear reservations (see Book III-E).

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 385. 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 ***stwcx.*** or ***stdcx.*** that specifies a dummy writable aligned location as part of the context switch; see Section 6.4.3 of Book III-S and Section 5.5 of Book III-E.

Programming Note

Because the reservation is lost if another processor stores anywhere in the reservation granule, lock words (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 389).

1.7.3.2 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 writing elsewhere in the reservation granule for X, as well as cancellation caused by the operating system in managing certain limited resources such as real storage. It may also include implementation-dependent causes of reservation loss.

An implementation may make a forward progress guarantee, defining the conditions under which the system as a whole makes progress. Such a guarantee must specify the possible causes of reservation loss in Case 3. While the architecture alone cannot provide such a guarantee, the characteristics listed in Cases 1 and 2 are necessary conditions for any forward progress guarantee. An implementation and operating system can build on them to provide such a guarantee.

Programming Note

The architecture does not include a “fairness guarantee”. In competing for a reservation, two processors can indefinitely lock out a third.

1.8 Instruction Storage

The instruction execution properties and requirements described in this section, including its subsections, apply only to instruction execution that is required by the sequential execution model.

In this section, including its subsections, it is assumed that all instructions for which execution is attempted are in storage that is not Caching Inhibited and (unless instruction address translation is disabled; see Book III) is not Guarded, and from which instruction fetching does not cause the system error handler to be invoked (e.g., from which instruction fetching is not prohibited by the “address translation mechanism” or the “storage protection mechanism”; see Book III).

Programming Note

The results of attempting to execute instructions from storage that does not satisfy this assumption are described in Section 1.6.2 and Section 1.6.4 of this Book and in Book III.

For each instance of executing an instruction from location X, the instruction may be fetched multiple times.

The instruction cache is not necessarily kept consistent with the data cache or with main storage. It is the responsibility of software to ensure that instruction storage is consistent with data storage when such consistency is required for program correctness.

After one or more bytes of a storage location have been modified and before an instruction located in that storage location is executed, software must execute the appropriate sequence of instructions to make instruction storage consistent with data storage. Otherwise the result of attempting to execute the instruction is boundedly undefined except as described in Section 1.8.1, “Concurrent Modification and Execution of Instructions” on page 353.

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 is not consistent with data storage or which becomes inconsistent with data storage prior to the completion of its processing. This section describes the only case in which executing this instruction under these conditions produces defined results.

In the remainder of this section the following terminology is used.

- Location X is an arbitrary word-aligned storage location.
- X_0 is the value of the contents of location X for which software has made the location X in instruction storage consistent with data storage.
- X_1, X_2, ..., X_n are the sequence of the first n values occupying location X after X_0.
- X_n is the first value of X subsequent to X_0 for which software has again made instruction storage consistent with data storage.
- The "patch class" of instructions consists of the I-form *Branch* instruction (***b***[***l***][***a***]) and the preferred no-op instruction (***ori*** 0,0,0).

If the instruction from location X is executed after the copy of location X in instruction storage is made consistent for the value X_0 and before it is made consistent for the value X_n, the results of executing the instruction are defined if and only if the following conditions are satisfied.

1. The stores that place the values X_1, ..., X_n into location X are atomic stores that modify all four bytes of location X.
2. Each X_i, $0 \le i \le n$, is a patch class instruction.
3. Location X is in storage that is Memory Coherence Required.

If these conditions are satisfied, the result of each execution of an instruction from location X will be the execution of some X_i, $0 \le i \le n$. The value of the ordinate i associated with each value executed may be different and the sequence of ordinates i associated with a sequence of values executed is not constrained, (e.g., a valid sequence of executions of the instruction at location X could be the sequence X_i, X_{i+2}, 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 a patch class instruction in the original program with a *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

It is believed that all processors that comply with versions of the architecture that precede Version 2.01 support concurrent modification and execution of instructions as described in this section if the requirements given above are satisfied, and that most such processors yield boundedly undefined results if the requirements given above are not satisfied. However, in general such support has not been verified by processor testing. Also, one such processor is known to yield undefined results in certain cases if the requirements given above are not satisfied.

Chapter 2. Effect of Operand Placement on Performance

The placement (location and alignment) of operands in storage affects relative performance of storage accesses, and may affect it significantly. The best performance is guaranteed if storage operands are aligned. In order to obtain the best performance across the widest range of implementations, the programmer should assume the performance model described in Figure 1 with respect to the placement of storage operands for the Embedded environment. For the Server environment, Figure 1 applies for Big-Endian byte ordering, and Figure 2 applies for Little-Endian byte ordering. Performance of storage accesses varies depending on the following:

1. Operand Size
2. Operand Alignment
3. Crossing no boundary
4. Crossing a cache block boundary
5. Crossing a virtual page boundary

The *Move Assist* instructions have no alignment requirements.

Operand		**Boundary Crossing**		
Size	**Byte Align.**	**None**	**Cache Block**	**Virtual Page**[2]
Integer				
8 Byte	8 4 <4	optimal good good	- good good	- good good
4 Byte	4 <4	optimal good	- good	- good
2 Byte	2 <2	optimal good	- good	- good
1 Byte	1	optimal	-	-
lmw, ***stmw***	4 <4	good poor	good poor	good poor
string		good	good	good
Float				
8 Byte	8 4 <4	optimal good poor	- good poor	- poor poor
4 Byte	4 <4	optimal poor	- poor	- poor
Vector				
any	any	optimal[3]	-	-

[1] If an instruction causes an access that is not atomic and any portion of the operand is in storage that is Write Through Required or Caching Inhibited, performance is likely to be poor.

[2] If the storage operand spans two virtual pages that have different storage control attributes or, in the Server environment, spans two segments, performance is likely to be poor.

[3] The storage operands for Vector instructions are all assumed to be aligned (see Section 5.4 of Book I).

Figure 1. Performance effects of storage operand placement

Operand		Boundary Crossing		
Size	Byte Align.	None	Cache Block	Virtual Page[2]
Integer				
8 Byte	8 4 <4	optimal poor poor	- poor poor	- poor poor
4 Byte	4 <4	optimal poor	- poor	- poor
2 Byte	2 <2	optimal poor	- poor	- poor
1 Byte	1	optimal	-	-
Float				
8 Byte	8 4 <4	optimal poor poor	- poor poor	- poor poor
4 Byte	4 <4	optimal poor	- poor	- poor
Vector				
any	any	optimal[3]	-	-

1 If an instruction causes an access that is not atomic and any portion of the operand is in storage that is Write Through Required or Caching Inhibited, performance is likely to be poor.

2 If the storage operand spans two virtual pages that have different storage control attributes or, in the Server environment, spans two segments, performance is likely to be poor.

3 The storage operands for Vector instructions are all assumed to be aligned (see Section 5.4 of Book I).

Figure 2. [Category: Server] Performance effects of storage operand placement, Little-Endian

2.1 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 Breakpoint match”, for which the corresponding rules are given in Book III).

1. A *Store* instruction that causes an atomic access and, for the Embedded environment, accesses storage that is Guarded
2. A *Load* instruction that causes an atomic access to storage that is Guarded and, for the Server environment, is also Caching Inhibited

Any other *Load* or *Store* instruction may be partially executed and then aborted after having accessed a portion of the storage operand, and then re-executed (i.e., restarted, by the processor or the operating system). If an instruction is partially executed, the contents of registers are preserved to the extent that the correct result will be produced when the instruction is re-executed. Additional restrictions on the partial execution of instructions are described in Section 6.6 of Book III-S and Section 5.7 of Book III-E.

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, or for an ***eciwx*** instruction, 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. Storage Control Instructions

3.1 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. Granule sizes for reservations
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.

3.2 Cache Management Instructions

The *Cache Management* instructions obey the sequential execution model except as described in Section 3.2.1.

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 341, storage protection, reference and change recording<S>, and the storage access ordering described in Section 1.7.1 and is treated as a *Read* (*Write*) from (to) the addressed byte with respect to debug events unless otherwise specified. (See Book III-E.)

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.

3.2.1 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 3.2), except that reference and change recording<S> 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
[Category: Embedded]

31	/	CT	RA	RB	22	/
0	6	7	11	16	21	31

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

If CT=0, this instruction provides a hint that the program will probably soon execute code from the addressed location.

If CT≠0, the operation performed by this instruction is implementation-dependent, except that the instruction is treated as a no-op for values of CT that are not implemented.

The hint is ignored if the block is Caching Inhibited.

This instruction treated as a *Load* (see Section 3.2), except that the system instruction storage error handler is not invoked.

Special Registers Altered:
None

3.2.2 Data Cache Instructions

Data Cache Block Allocate *X-form*

dcba RA,RB
[Category: Embedded]

31	///	RA	RB	758	/
0	6	11	16	21	31

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

This instruction provides a hint that the program will probably soon store into a portion of the block and the contents of the rest of the block are not meaningful to the program. The contents of the block are undefined when the instruction completes. The hint is ignored if the block is Caching Inhibited.

This instruction is treated as a *Store* (see Section 3.2) except that the instruction is treated as a no-op if execution of the instruction would cause the system data storage error handler to be invoked.

Special Registers Altered:
None

Data Cache Block Touch *X-form*

dcbt RA,RB,TH [Category: Server]
dcbt TH,RA,RB [Category: Embedded]

31	/	TH	RA	RB	278	/
0	6	7	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, or indicates the expected use thereof. A hint that the program will probably soon load from a given storage location is ignored if the location is Caching Inhibited or, for the Server environment, Guarded.

The only operation that is “caused” by the ***dcbt*** instruction is the providing of the hint. The actions (if any) taken by the processor in response to the hint are not considered to be “caused by” or “associated with” the ***dcbt*** instruction (e.g., ***dcbt*** is considered not to cause any data accesses). No means are provided by which software can synchronize these actions with the execution of the instruction stream. For example, these actions are not ordered by the memory barrier created by a ***sync*** instruction.

The ***dcbt*** instruction may complete before the operation it causes has been performed.

The nature of the hint depends, in part, on the value of the TH field, as specified below. If TH≠0b1010 this instruction is treated as a *Load* (see Section 3.2), except that the system data storage error handler is not invoked, and reference and change recording<S> 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.

Extended:	**Equivalent to:**
dcbtct RA,RB,TH	dcbt for TH values of 0b0000 - 0b0111; other TH values are invalid.
dcbtds RA,RB,TH	dcbt for TH values of 0b0000 or 0b1000 - 0b1010; other TH values are invalid.

Programming Note

New programs should avoid using the ***dcbt*** and ***dcbtst*** mnemonics; one of the extended mnemonics should be used exclusively.

<S> If the ***dcbt*** mnemonic is used with only two operands, the TH operand assumed to be 0b0000.

TH Field

For all TH field values which are not listed below, the hint provided by the instruction is undefined.

TH=0b0000

If TH=0b0000, the ***dcbt*** instruction provides a hint that the program will probably soon load from the block containing the byte addressed by EA.

TH=0b0000 - 0b0111
[Category: Cache Specification]

In addition to the hint specified above for the TH field value of 0b0000, an additional hint is provided indicating that placement of the block in the cache specified by the TH field might also improve performance. The correspondence between each value of the TH field and the cache to be specified is the same as the correspondence between each value the CT field and the cache to be specified as defined in Section 3.2. The hints corresponding to values of the TH field not supported by the implementation are undefined.

TH=0b1000 - 0b1111 [Category: Stream]

The hints provided by the ***dcbt*** instruction provide a hint regarding a sequence of contiguous data cache blocks, or indicates the expected use thereof. Such a sequence is called a “data stream”, and a ***dcbt*** instruction in which TH is set to one of these values is said to be a “data stream variant” of ***dcbt***. In the remainder of this section, “data stream” may be abbreviated to “stream”.

When, and how often, effective addresses for a data stream are translated is implementation-dependent.

The address and length of such data streams are specified in terms of aligned 128-byte units of storage; in the remainder of this instruction description, “aligned 128-byte unit of storage” is abbreviated to “unit”.

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.

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 fields, and reserved values of defined EA fields, are treated in the same manner as the corresponding cases for instruction fields (see the section entitled “Reserved Fields and Reserved Values” in Book I), except that a reserved value in a defined EA field does not make the instruction form invalid. If a defined EA field contains a reserved value, the hint provided by the instruction is undefined.

TH Description

1000 The ***dcbt*** instruction provides a hint that describes certain attributes of a data stream, and may indicate that the program will probably soon load from the stream.

The EA is interpreted as follows.

EATRUNC	D	UG	/	ID
0	57		59	60 63

Bit(s) Description

0:56 ***EATRUNC***

High-order 57 bits of effective address of first unit of data stream (i.e., the effective address of the first unit of the stream is EATRUNC || $^{7}0$)

57 ***Direction*** (D)

- 0 Subsequent units are the sequentially following units.
- 1 Subsequent units are the sequentially preceding units.

58 ***Unlimited/GO*** (UG)

- 0 No information is provided by the UG field.
- 1 The number of units in the data stream is unlimited, the program’s need for each block of the stream is not likely to be transient, and the program will probably soon load from the stream.

59 Reserved

60:63 ***Stream ID*** (ID)

Stream ID to use for this data stream

1010 The ***dcbt*** instruction provides a hint that describes certain attributes of a data stream, or indicates that the program will probably soon load from data streams that have been described using ***dcbt*** instructions in which TH_0=1 or will probably no longer load from such data streams.

The EA is interpreted as follows. If GO=1 and S$\neq$0b00 the hint provided by the instruction is undefined; the remainder of this instruction description assumes that this combination is not used.

///	GO	S	///	UNITCNT	T	U	/	ID
0	32		35	47	57		59	60 63

Bit(s) Description

0:31 Reserved

32 ***GO***

0 No information is provided by the GO field.

1 The program will probably soon load from all nascent data streams that have been completely described, and will probably no longer load from all other nascent data streams. All other fields of the EA are ignored. (“Nascent” and “completely described” are defined below.)

33:34 ***Stop*** (S)

00 No information is provided by the S field.

01 Reserved

10 The program will probably no longer load from the data stream (if any) associated with the specified stream ID. (All other fields of the EA except the ID field are ignored.)

11 The program will probably no longer load from the data streams associated with all stream IDs. (All other fields of the EA are ignored.)

35:46 Reserved

47:56 ***UNITCNT***

Number of units in data stream

57 ***Transient*** (T)

If T=1, the program’s need for each block of the data stream is likely to be transient (i.e., the time interval during which the program accesses the block 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 from which the program will probably no longer load (S=0b10)

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=0b1000 or (b) TH=0b1010 and GO=0 and S≠0b11, no hint is provided by the instruction.

The following terminology is used to describe the state of a data stream. Except as described in the paragraph after the next paragraph, the state of a data stream at a given time is determined by the most recently provided hint for the stream.

- A data stream for which only descriptive hints have been provided (by ***dcbt*** instructions with TH=0b1000 and UG=0 or with TH=0b1010 and GO=0 and S=0b00) is said to be “nascent”. A nascent data stream for which both kinds of descriptive hint have been provided (by both of the ***dcbt*** usages listed in the preceding sentence) is considered to be “completely described”.
- A data stream for which a hint has been provided (by a ***dcbt*** instruction with TH=0b1000 and UG=1 or with TH=0b1010 and GO=1) that the program will probably soon load from 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=0b1010 and S≠0b00) that the program will probably no longer load from it is considered no longer to exist.

The hint provided by a ***dcbt*** instruction with TH=0b1000 and UG=1 implicitly includes a hint that the program will probably no longer load from the data stream (if any) previously associated with the specified stream ID. The hint provided by a ***dcbt*** instruction with TH=0b1000 and UG=0 or with TH=0b1010 and GO=0 and S=0b00 implicitly includes a hint that the program will probably no longer load from the *active* data stream (if any) previously associated with the specified stream ID.

Interrupts (see Book III) cause all existing data streams to cease to exist. In addition, depending on the implementation, certain conditions and events may cause an existing data stream to cease to exist.

Programming Note

To obtain the best performance across the widest range of implementations that support the variants of ***dcbt*** in which TH_0=1, the programmer should assume the following model when using those variants.

- The processor's response to a hint that the program will probably soon load from a given data stream is to take actions that reduce the latency of loads from the first few blocks of the stream. (Such actions may include prefetching the blocks into levels of the storage hierarchy that are "near" the processor.) Thereafter, as the program loads from each successive block of the stream, the processor takes latency-reducing actions for additional blocks of the stream, pacing these actions with the program's loads (i.e., taking the actions for only a limited number of blocks ahead of the block that the program is currently loading from).

  The processor's response to a hint that the program will probably no longer load from 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 blocks of the stream.
- If the program ceases to need a given data stream before having loaded from all blocks 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 load from the stream (e.g., by executing the appropriate ***dcbt*** instruction with TH=0b1010 and S≠0b00).
- At each level of the storage hierarchy that is "near" the processor, blocks 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 a unit of a transient stream will be replaced prior to being accessed by the program.
- 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 load from 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 ***dcbt*** instructions in which TH_0=1.

- A non-transient data stream having unlimited length can be completely specified, including providing the hint that the program will probably soon load from it, using one ***dcbt*** instruction. The corresponding specification for a data stream having other attributes requires three ***dcbt*** instructions. However, one ***dcbt*** instruction with TH=0b1010 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+1 ***dcbt*** instructions. (There is no need to execute a ***dcbt*** instruction with TH=0b1010 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*** instructions that describe data streams suffices.)
- If it is desired that the hint provided by a given ***dcbt*** instruction be provided in program order with respect to the hint provided by another ***dcbt*** instruction, the two ***dcbt*** instructions must be separated by an ***eieio***<S> (or ***sync***) instruction. For example, if a ***dcbt*** instruction with TH=0b1010 and GO=1 is intended to indicate that the program will probably soon load from nascent data streams described (completely) by preceding ***dcbt*** instructions, and is intended *not* to indicate that the program will probably soon load from nascent data streams described (completely) by *following* ***dcbt*** instructions, an ***eieio***<S> (or ***sync***) instruction must separate the ***dcbt*** instruction with GO=1 from the preceding ***dcbt*** instructions, and another ***eieio***<S> (or ***sync***) instruction must separate that ***dcbt*** instruction from the following ***dcbt*** instructions.
- In practice, the second ***eieio***<S> (or ***sync***) described above can sometimes be omitted. For example, if the program consists of an outer loop that contains the ***dcbt*** instructions and an inner loop that contains the *Load* instructions that load from the data streams, the characteristics of the inner loop and of the implementation's branch prediction mechanisms may make it highly unlikely that hints corresponding to a given iteration of the outer loop will be provided out of program order with respect to hints corresponding to the previous iteration of the outer loop. (Also, any providing of hints out of program order affects only performance, not program correctness.)
- To mitigate the effects of interrupts on data streams, it may be desirable to specify a given "logical" data stream as a sequence of shorter, component data streams. Similar considerations apply to conditions and events that, depending on the implementation, may cause an existing data stream to cease to exist.
- 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.

Data Cache Block Touch for Store X-form

dcbtst RA,RB [Category: Server]
dcbtst TH,RA,RB [Category: Embedded]

31	/	TH	RA	RB	246	/
0	6	7	11	16	21	31

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

The ***dcbtst*** instruction provides a hint that the program will probably soon store to the block containing the byte addressed by EA. If the Cache Specification category is supported, the nature of the hint depends on the value of the TH field, as specified below. If the Cache Specification category is not supported, the TH field is treated as a reserved field.

The hint is ignored if the block is in a storage location that is Caching Inhibited or, for the Server environment, 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.

This instruction is treated as a *Load* (see Section 3.2), except that the system data storage error handler is not invoked, and reference and change recording<S> need not be done.

TH Field [Category: Cache Specification]

For all TH field values which the are not listed below, the hint provided by the instruction is undefined.

TH=0b0000 - 0b0111 [Category: Cache Specification]

In addition to the hint provided if the Cache Specification category is not supported, a hint is provided indicating that placement of the block in the cache specified by the TH field might also improve performance. The correspondence between each value of the TH field and the cache to be specified is the same as the correspondence between each value of the CT field and the cache to be specified as defined in Section 3.2. The hints corresponding to values of the TH field not supported by the implementation are undefined.

Special Registers Altered:

None

Extended Mnemonic:

An extended mnemonic is 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.
.

Extended:	**Equivalent to:**
dcbtstct RA,RB,TH	dcbt for TH values of 0b0000 or 0b0000 - 0b0111; other TH values are invalid for this extended mnemonic.

Programming Note

See the Programming Notes for the ***dcbt*** instruction.

Programming Note

The processor's response to the hint provided by ***dcbt*** or ***dcbtst*** is to take actions that reduce the latency of subsequent loads or stores that access the specified block. (Such actions may include prefetching the block into levels of the storage hierarchy that are "near" the processor.)

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.

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 3.2).

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 5.9.1 of Book III-S and Section 4.9.1 of Book III-E. for additional information about ***dcbz***.

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 3.2), except that reference and change recording<S> need not be done, and it is treated as a *Write* with respect to debug events.

Special Registers Altered:
None

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).

Data Cache Block Flush *X-form*

dcbf RA,RB,L

31	///	L	RA	RB	86	/
0	6	10	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") [Category: Server Phased-In]

The L=1 form of the ***dcbf*** instruction permits a program to limit the scope of the "flush" operation to the data cache of a single processor. If the block containing the byte addressed by EA 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.

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. If L=1, the function of this instruction is also independent of whether the block containing the byte addressed by EA is in storage that is Memory Coherence Required.

This instruction is treated as a *Load* (see Section 3.2), except that reference and change recording<S> need not be done, and it is treated as a *Write* with respect to debug events.

Special Registers Altered:

None

Extended Mnemonics:

Extended mnemonics are provided for the *Data Cache Block Flush* instruction so that it can be coded with the L value as part of the mnemonic rather than as a numeric operand. These are shown as examples with the instruction. See Appendix A. "Assembler Extended Mnemonics" on page 383. The extended mnemonics are shown below.

Extended:	Equivalent to:
dcbf RA,RB	dcbf RA,RB,0
dcbfl RA,RB<S>	dcbf RA,RB,1

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***<S>" is used for L=1.

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 [Category: Server]

dcbf with L=1 can be used to cause a block that will not be reused soon to be removed from the processor's data cache, and thereby potentially to cause that data cache to be used more efficiently.

Programming Note [Category: Server]

The functions provided by ***dcbf*** with L=1 are identical to those that would be provided if L were 0 and the specified block were in storage that is not Memory Coherence Required.

3.2.2.1 Obsolete Data Cache Instructions [Category: Vector.Phased-Out]

The *Data Stream Touch* (***dst***), *Data Stream Touch for Store* (***dstst***), and *Data Stream Stop* (***dss***) instructions (primary opcode 31, extended opcodes 342, 374, and 822 respectively), which were proposed for addition to the Power ISA and were implemented by some processors, may be treated as no-ops (rather than as illegal instructions).

The treatment of these instructions (no-op or illegal instruction) is independent of whether other Vector instructions are available (i.e., is independent of the contents of MSR_{VEC}<S> (see Book III-S) or MSR_{SPV} (see Book III-E).

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***.

3.3 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.

3.3.1 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

3.3.2 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 1.7.3, “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 other processors or mechanisms. If the specified location is in storage that is Write Through Required or Caching Inhibited, the system data storage error handler or the system alignment error handler is invoked for the Server environment and may be invoked for the Embedded environment.

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.

Load Word And Reserve Indexed X-form

lwarx RT,RA,RB

31	RT	RA	RB	20	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b +(RB)
RESERVE ← 1
RESERVE_ADDR ← real_addr(EA)
RT ← 320 || MEM(EA, 4)
```

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

This instruction creates a reservation for use by a *Store Word Conditional* instruction. An address computed from the EA as described in Section 1.7.3.1 is associated with the reservation, and replaces any address previously associated with the reservation.

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

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_ADDR = real_addr(EA) then
    MEM(EA, 4) ← (RS)32:63
    CR0 ← 0b00 || 0b1 || XERSO
   else
    u1 ← undefined 1-bit value
    if u1 then
      MEM(EA, 4) ← (RS)32:63
    u2 ← undefined 1-bit value
    CR0 ← 0b00 || u2 || XERSO
   RESERVE ← 0
else
   CR0 ← 0b00 || 0b0 || XERSO
```

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

If a reservation exists and the storage location specified by the ***stwcx.*** is the same as the location specified by the *Load And Reserve* instruction that established the reservation, $(RS)_{32:63}$ are stored into the word in storage addressed by EA and the reservation is cleared.

If a reservation exists but the storage location specified by the ***stwcx.*** is not the same as the location specified by the *Load And Reserve* instruction that established the reservation, the reservation is cleared, and it is undefined whether $(RS)_{32:63}$ are stored into the word in storage addressed by EA.

If a reservation does not exist, the instruction completes without altering storage.

CR Field 0 is set as follows. n is a 1-bit value that indicates whether the store was performed, except that if a reservation exists but the storage location specified by the ***stwcx.*** is not the same as the location specified by the *Load And Reserve* instruction that established the reservation the value of n is undefined.

$$CR0_{LT\ GT\ EQ\ SO} = 0b00 \,||\, n \,||\, XER_{SO}$$

EA must be a multiple of 4. If it is not, either the system alignment error handler is invoked or the results are boundedly undefined.

Special Registers Altered:

CR0

3.3.2.1 64-Bit Load and Reserve and Store Conditional Instructions [Category: 64-Bit]

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_ADDR = real_addr(EA) then
    MEM(EA, 8) ← (RS)
    CR0 ← 0b00 || 0b1 || XER_SO
  else
    u1 ← undefined 1-bit value
    if u1 then
      MEM(EA, 8) ← (RS)
    u2 ← undefined 1-bit value
    CR0 ← 0b00 || u2 || XER_SO
  RESERVE ← 0
else
  CR0 ← 0b00 || 0b0 || XER_SO
```

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

If a reservation exists and the storage location specified by the ***stdcx.*** is the same as the location specified by the *Load And Reserve* instruction that established the reservation, (RS) is stored into the doubleword in storage addressed by EA and the reservation is cleared.

If a reservation exists but the storage location specified by the ***stdcx.*** is not the same as the location specified by the *Load And Reserve* instruction that established the reservation, the reservation is cleared, and it is undefined whether (RS) is stored into the doubleword in storage addressed by EA.

If a reservation does not exist, the instruction completes without altering storage.

CR Field 0 is set as follows. n is a 1-bit value that indicates whether the store was performed, except that if a reservation exists but the storage location specified by the ***stdcx.*** is not the same as the location specified by the *Load And Reserve* instruction that established the reservation the value of n is undefined.

$$CR0_{LT\ GT\ EQ\ SO} = 0b00 \ || \ n \ || \ XER_{SO}$$

EA must be a multiple of 8. If it is not, either the system alignment error handler is invoked or the results are boundedly undefined.

Special Registers Altered:

CR0

Load Doubleword And Reserve Indexed X-form

ldarx RT,RA,RB

31	RT	RA	RB	84	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b +(RB)
RESERVE ← 1
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 *Store Doubleword Conditional* instruction. An address computed from the EA as described in Section 1.7.3.1 is associated with the reservation, and replaces any address previously associated with the reservation.

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

3.3.3 Memory Barrier Instructions

The *Memory Barrier* instructions can be used to control the order in which storage accesses are performed. Additional information about these instructions and about related aspects of storage management can be found in Book III.

Extended mnemonics for Synchronize

Extended mnemonics are provided for the *Synchronize* instruction so that it can be supported by assemblers that recognize only the ***msync***<E> mnemonic and so that it can be coded with the L value as part of the mnemonic rather than as a numeric operand. These are shown as examples with the instruction. See Appendix A. “Assembler Extended Mnemonics” on page 383.

Synchronize *X-form*

sync L

31	///	L	///	///	598	/
0	6	9	11	16	21	31

The ***sync*** instruction creates a memory barrier (see Section 1.7.1). The set of storage accesses that is ordered by the memory barrier depends on the value of the L field.

L=0 (“heavyweight sync”)

The memory barrier provides an ordering function for the storage accesses associated with all instructions that are executed by the processor executing the ***sync*** instruction. The applicable pairs are all pairs a_i,b_j in which b_j is a data access, 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”)<S>

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 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<S>

The set of storage accesses that is ordered by the memory barrier is described in Section 5.9.2 of Book III-S and Section 4.9.3 of Book III-E, as are additional properties of the ***sync*** instruction with L=2.

The ordering done by the memory barrier is cumulative.

If L=0 (or L=2<S>), 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<S> (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 359).

  The cumulative properties of the 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.
- The ***sync*** instruction provides an ordering function for the operations caused by ***dcbt*** instructions with TH_0=1.

The value L=3 is reserved.

The ***sync*** instruction may complete before storage accesses associated with instructions preceding the ***sync*** instruction have been performed. The ***sync*** instruction may complete before operations caused by ***dcbt*** instructions with TH_0=1 preceding the ***sync*** instruction have been performed.

Special Registers Altered:

None

Extended Mnemonics:

Extended mnemonics for *Synchronize*:

Extended:	**Equivalent to:**
sync	sync 0
msync<E>	sync 0
lwsync<S>	sync 1
ptesync<S>	sync 2

Except in the ***sync*** instruction description in this section, references to "***sync***" in Books I-III imply L=0 unless otherwise stated or obvious from context; the appropriate extended mnemonics are used when other L values are intended.

Programming Note

Section 1.8 contains a detailed description of how to modify instructions such that a well-defined result is obtained.

Programming Note

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

Programming Note

The ***sync*** instruction can be used to ensure that all stores into a data structure, caused by *Store* instructions executed in a "critical section" of a program, will be performed with respect to another processor before the store that releases the lock is performed with respect to that processor; see Section B.2, "Lock Acquisition and Release, and Related Techniques" on page 387.

The memory barrier created by a ***sync*** instruction with L=0 or L=1 does not order implicit storage accesses. The memory barrier created by a ***sync*** instruction with any L value does not order instruction fetches.

(The memory barrier created by a ***sync*** instruction with L=0 – or L=2<S>; see Book III – *appears* to order instruction fetches for instructions preceding the ***sync*** instruction with respect to data accesses caused by instructions following the ***sync*** instruction. However, this ordering is a consequence of the first "additional property" of ***sync*** with L=0, not a property of the memory barrier.)

In order to obtain the best performance across the widest range of implementations, the programmer should use the ***sync*** instruction with L=1, or the ***eieio***<S> or ***mbar***<E> instruction, if any of these is sufficient for his needs; otherwise he should use ***sync*** with L=0. ***sync*** with L=2<S> should not be used by application programs.

Programming Note

The functions provided by ***sync*** with L=1 are a strict subset of those provided by ***sync*** with L=0. (The functions provided by ***sync*** with L=2<S> are a strict superset of those provided by ***sync*** with L=0; see Book III.)

Enforce In-order Execution of I/O X-form

eieio
[Category: Server]

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*, ***dcbz***, ***eciwx***, and ***ecowx*** 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 an ***eciwx*** instruction is ordered as a load, and the storage access caused by a ***dcbz*** or ***ecowx*** 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 ***dcbt*** instructions with TH_0 = 1 are ordered by ***eieio*** as a third set of operations, and the operations caused by ***tlbie***<S> and ***tlbsync*** instructions (see Book III-S) are ordered by ***eieio*** as a fourth set of operations.

Each of the four sets of storage accesses or operations is ordered independently of the other three 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 three sets is not cumulative.

The ***eieio*** instruction may complete before storage accesses or operations associated with instructions preceding the ***eieio*** instruction have been performed.

Special Registers Altered:
None

Memory Barrier X-form

mbar MO
[Category: Embedded]

31	MO	///	///	854	/
0	6	11	16	21	31

When MO=0, the ***mbar*** instruction creates a cumulative memory barrier (see Section 1.7.1, “Storage Access Ordering”), which provides an ordering function for the storage accesses executed by the processor executing the ***mbar*** instruction.

When MO≠0, an implementation may support the ***mbar*** instruction ordering a particular subset of storage accesses. An implementation may also support multiple, non-zero values of MO that each specify a different subset of storage accesses that are ordered by the ***mbar*** instruction. Which subsets of storage accesses that are ordered and which values of MO that specify these subsets is implementation-dependent.

The ***mbar*** instruction may complete before storage accesses associated with instructions preceding the ***mbar*** instruction have been performed. The ***mbar*** instruction may complete before operations caused by ***dcbt*** instructions having TH_0=1 preceding the ***mbar*** instruction have been performed.

Special Registers Altered:
None

Programming Note

The ***eieio***<S> and ***mbar***<E> instructions are intended for use in doing memory-mapped I/O, and in preventing load/store combining operations in main storage (see Section 1.6, “Storage Control Attributes” on page 344).

Because 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 347), ***eieio***<S> or ***mbar***<E> is needed for such storage only when loads must be ordered with respect to stores or with respect to other loads, or when load/store combining operations must be prevented.

For the ***eieio***<S> 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***<S> or ***mbar***<E>, the ***sync*** instruction must be used, with the appropriate value in the L field.

Programming Note

The functions provided by ***eieio***<S> and ***mbar***<E> are a strict subset of those provided by ***sync*** with L=0. The functions provided by ***eieio***<S> for its second set are a strict subset of those provided by ***sync*** with L=1.

Since ***eieio***<S> and ***mbar***<E>share the same opcode, software designed for both server and embedded environments must assume that only the ***eieio***<S> functionality applies since the functions provided by ***eieio*** are a subset of those provided by ***mbar***.

3.3.4 Wait Instruction

Wait ***X-form***

wait
[Category: Wait]

31	///	///	///	62	/
0	6	11	16	21	31

The ***wait*** instruction provides an ordering function for the effects of all instructions executed by the processor executing the **wait** instruction. Executing a ***wait*** instruction ensures that all instructions have completed before the ***wait*** instruction completes, and that no subsequent instructions are initiated until an interrupt occurs. The ***wait*** instruction also causes any prefetched instructions to be discarded and instruction fetching is suspended until an interrupt occurs.

Once the ***wait*** instruction has completed, the NIA will point to the next sequential instruction.

Special Registers Altered:
None

Programming Note

The ***wait*** instruction can be used in verification test cases to signal the end of a test case. The encoding for the instruction is the same in both Big-Endian and Little-Endian modes.

Programming Note

The ***wait*** instruction may be useful as the primary instruction of an “idle process” or the completion of processing for a cooperative thread. Note that ***wait*** updates the NIA so that an interrupt that awakens a ***wait*** instruction will return to the instruction after the ***wait***.

Chapter 4. Time Base

4.1 Time Base Overview

The time base facilities include a Time Base and an Alternate Time Base which is category: Alternate Time Base. The Alternate Time Base is analogous to the Time Base except that it may count at a different frequency and is not writable.

4.2 Time Base

The Time Base (TB) is a 64-bit register (see Figure 3) containing a 64-bit unsigned integer that is incremented periodically. Each increment adds 1 to the low-order bit (bit 63). The frequency at which the integer is updated is implementation-dependent.

TBU	TBL
0	32 63

Field	Description
TBU	Upper 32 bits of Time Base
TBL	Lower 32 bits of Time Base

Figure 3. Time Base

The Time Base 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 explicit indication (such as an interrupt; see Book III) that this has occurred.

The period of the Time Base depends on the driving frequency. As an order of magnitude example, suppose that the CPU clock is 1 GHz and that the Time Base is driven by this frequency divided by 32. Then the period of the Time Base would be

$$T_{TB} = \frac{2^{64} \times 32}{1\text{GHz}} = 5.90 \times 10^{11} \text{ seconds}$$

which is approximately 18,700 years.

The Power ISA AS 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 AS 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.

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.

4.2.1 Time Base Instructions

Move From Time Base *XFX-form*

mftb RT,TBR
[Category: Server.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 3.3.14 of Book I.

Special Registers Altered:

None.

Extended Mnemonics:

Extended mnemonics for *Move From Time Base*:

Extended:	Equivalent to:
mftb Rx	mftb Rx,268 mfspr Rx,268
mftbu Rx	mftb Rx,269 mfspr Rx,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).

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 an Illegal Instruction type Program interrupt and thereby permits the operating system to emulate the Time Base.)

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 is incremented at a constant rate of once for every 32 cycles of a 1 GHz CPU instruction clock. 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

  $$\frac{1\ \text{GHz}}{32} = 31{,}250{,}000$$

  which is the number of times the Time Base is updated each second.
- The integer constant *ns_adj* contains the value

  $$\frac{1{,}000{,}000{,}000}{31{,}250{,}000} = 32$$

  which is the number of nanoseconds per tick of the Time Base.

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
divd   Rz,Ry,Rx# Rz = whole seconds
stw    Rz,posix_sec
mulld  Rz,Rz,Rx# Rz = quotient × divisor
sub    Rz,Ry,Rz# Rz = excess ticks
```

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.

```
   lwz    Rx,ns_adj
   mulld  Rz,Rz,Rx# Rz = excess nanoseconds
   stw    Rz,posix_ns
```

For the Embedded environment when the processor is in 32-bit mode, it is not possible to read the Time Base using a single instruction. Instead, two instructions must be used, one of which reads TBL and the other of which reads TBU. Because of the possibility of a carry from TBL to TBU occurring between the two reads, a sequence such as the following must be used to read the Time Base.

```
loop:
   mfspr  Rx,TBU  # load from TBU
   mfspr  Ry,TB   # load from TB
   mfspr  Rz,TBU  # load from TBU
   cmp    cr0,0,Rx,Rz# check if 'old'='new'
   bne    loop    #branch if carry occurred
```

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.

4.3 Alternate Time Base [Category: Alternate Time Base]

The Alternate Time Base (ATB) is a 64-bit register (see Figure 3) containing a 64-bit unsigned integer that is incremented periodically. The frequency at which the integer is updated is implementation-dependent.

ATBU		ATBL	
0	32		63

Figure 4. Alternate Time Base

The ATBL register is an aliased name for the ATB.

The Alternate Time Base 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 explicit indication (such as an interrupt; see Book III) that this has occurred.

The Alternate Time Base is accessible in both user and supervisor mode. The counter can be read by executing a ***mfspr*** instruction specifying the ATB (or ATBL) register, but cannot be written. A second SPR register ATBU, is defined that accesses only the upper 32 bits of the counter. Thus the upper 32 bits of the counter may be read into a register by reading the ATBU register.

The effect of entering a power-savings mode or of processor frequency changes on counting in the Alternate Time Base is implementation-dependent.

Chapter 5. External Control [Category: External Control]

The External Control category of facilities and instructions permits a program to communicate with a special-purpose device. Two instructions are provided, both of which must be implemented if the facility is provided.

- *External Control In Word Indexed* (***eciwx***), which does the following:
  - Computes an effective address (EA) like most X-form instructions
  - Validates the EA as would be done for a load from that address
  - Translates the EA to a real address
  - Transmits the real address to the device
  - Accepts a word of data from the device and places it into a General Purpose Register
- *External Control Out Word Indexed* (***ecowx***), which does the following:
  - Computes an effective address (EA) like most X-form instructions
  - Validates the EA as would be done for a store to that address
  - Translates the EA to a real address
  - Transmits the real address and a word of data from a General Purpose Register to the device

Permission to execute these instructions and identification of the target device are controlled by two fields, called the E bit and the RID field respectively. If attempt is made to execute either of these instructions when E=0 the system data storage error handler is invoked. The location of these fields is described in Book III.

The storage access caused by ***eciwx*** and ***ecowx*** is performed as though the specified storage location is Caching Inhibited and Guarded, and is neither Write Through Required nor Memory Coherence Required.

Interpretation of the real address transmitted by ***eciwx*** and ***ecowx*** and of the 32-bit value transmitted by ***ecowx*** is up to the target device, and is not specified by the Power ISA. See the System Architecture documentation for a given Power ISA system for details on how the External Control facility can be used with devices on that system.

Example

An example of a device designed to be used with the External Control facility might be a graphics adapter. The ***ecowx*** instruction might be used to send the device the translated real address of a buffer containing graphics data, and the word transmitted from the General Purpose Register might be control information that tells the adapter what operation to perform on the data in the buffer. The ***eciwx*** instruction might be used to load status information from the adapter.

A device designed to be used with the External Control facility may also recognize events that indicate that the address translation being used by the processor has changed. In this case the operating system need not “pin” the area of storage identified by an ***eciwx*** or ***ecowx*** instruction (i.e., need not protect it from being paged out).

5.1 External Access Instructions

In the instruction descriptions the statements "this instruction is treated as a *Load*" and "this instruction is treated as a *Store*" have the same meanings as for the *Cache Management* instructions; see Section 3.2.

External Control In Word Indexed X-form

eciwx RT,RA,RB

31	RT	RA	RB	310	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + (RB)
raddr ← address translation of EA
send load word request for raddr to
   device identified by RID
RT ← 320 || word from device
```

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

A load word request for the real address corresponding to EA is sent to the device identified by RID, bypassing the cache. The word returned by the device is placed into $RT_{32:63}$. $RT_{0:31}$ are set to 0.

The E bit must be 1. If it is not, the data storage error handler is invoked.

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.

This instruction is treated as a *Load*.

See Book III-S for additional information about this instruction.

Special Registers Altered:
None

Programming Note

The ***eieio***<S> or ***mbar***<E> instruction can be used to ensure that the storage accesses caused by ***eciwx*** and ***ecowx*** are performed in program order with respect to other Caching Inhibited and Guarded storage accesses.

External Control Out Word Indexed X-form

ecowx RS,RA,RB

31	RS	RA	RB	438	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + (RB)
raddr ← address translation of EA
send store word request for raddr to
  device identified by RID
send (RS)32:63 to device
```

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

A store word request for the real address corresponding to EA and the contents of $RS_{32:63}$ are sent to the device identified by RID, bypassing the cache.

The E bit must be 1. If it is not, the data storage error handler is invoked.

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.

This instruction is treated as a *Store*, except that its storage access is not performed in program order with respect to accesses to other Caching Inhibited and Guarded storage locations unless software explicitly imposes that order.

See Book III-S for additional information about this instruction.

Special Registers Altered:
None

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 Flush Mnemonics

The L field in the *Data Cache Block Flush* instruction controls the scope of the flush function performed by the instruction. Extended mnemonics are provided that represent the L value in the mnemonic rather than requiring it to be coded as a numeric operand.

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<S> RA,RB	(equivalent to: dcbfl RA,RB,1)

A.2 Synchronize Mnemonics

The L field in the *Synchronize* instruction controls the scope of the synchronization function performed by the instruction. Extended mnemonics are provided that represent the L value in the mnemonic rather than requiring it to be coded as a numeric operand. Two extended mnemonics are provided for the L=0 value in order to support assemblers that do not recognize the ***sync*** mnemonic.

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

sync	(equivalent to:	sync	0)
msync<E>	(equivalent to:	sync	0)
lwsync<S>	(equivalent to:	sync	1)
ptesync<S>	(equivalent to:	sync	2)

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 D of Book I.

Many of the examples use the *Load And Reserve* and *Store Conditional* instructions, in a sequence that begins with a *Load And Reserve* instruction and ends with a *Store Conditional* instruction (specifying the same storage location as the *Load Conditional*) followed by a *Branch Conditional* instruction that tests whether the *Store Conditional* instruction succeeded.

In these examples it is assumed that contention for the shared resource is low; the conditional branches are optimized for this case by using "+" and "-" suffixes appropriately.

The examples deal with words; they can be used for doublewords by changing all word-specific mnemonics to the corresponding doubleword-specific mnemonics (e.g., ***lwarx*** to ***ldarx***, ***cmpw*** to ***cmpd***).

In this appendix it is assumed that all shared storage locations are in storage that is Memory Coherence Required, and that the storage locations specified by *Load And Reserve* and *Store Conditional* instructions are in storage that is neither Write Through Required nor Caching Inhibited.

B.1 Atomic Update Primitives

This section gives examples of how the *Load And Reserve* and *Store Conditional* instructions can be used to emulate atomic read/modify/write operations.

An atomic read/modify/write operation reads a storage location and writes its next value, which may be a function of its current value, all as a single atomic operation. The examples shown provide the effect of an atomic read/modify/write operation, but use several instructions rather than a single atomic instruction.

Fetch and No-op

The "Fetch and No-op" primitive atomically loads the current value in a word in storage.

In this example it is assumed that the address of the word to be loaded is in GPR 3 and the data loaded are returned in GPR 4.

```
loop:
   lwarx  r4,0,r3 #load and reserve
   stwcx. r4,0,r3 #store old value if
                  #  still reserved
   bne-   loop    #loop if lost reservation
```

Note:

1. The ***stwcx.***, if it succeeds, stores to the target location the same value that was loaded by the preceding ***lwarx***. While the store is redundant with respect to the value in the location, its success ensures that the value loaded by the ***lwarx*** is still the current value at the time the ***stwcx.*** is executed.

Fetch and Store

The "Fetch and Store" primitive atomically loads and replaces a word in storage.

In this example it is assumed that the address of the word to be loaded and replaced is in GPR 3, the new value is in GPR 4, and the old value is returned in GPR 5.

```
loop:
   lwarx  r5,0,r3 #load and reserve
   stwcx. r4,0,r3 #store new value if
                  #  still reserved
   bne-   loop    loop if lost reservation
```

Fetch and Add

The “Fetch and Add” primitive atomically increments a word in storage.

In this example it is assumed that the address of the word to be incremented is in GPR 3, the increment is in GPR 4, and the old value is returned in GPR 5.

```
loop:
   lwarx  r5,0,r3 #load and reserve
   add    r0,r4,r5#increment word
   stwcx. r0,0,r3 #store new value if still res’ved
   bne-   loop    #loop if lost reservation
```

Fetch and AND

The “Fetch and AND” primitive atomically ANDs a value into a word in storage.

In this example it is assumed that the address of the word to be ANDed is in GPR 3, the value to AND into it is in GPR 4, and the old value is returned in GPR 5.

```
loop:
   lwarx  r5,0,r3 #load and reserve
   and    r0,r4,r5#AND word
   stwcx. r0,0,r3 #store new value if still res’ved
   bne-   loop    #loop if lost reservation
```

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 res’ved
   bne-   loop    #loop if lost reservation
exit:
   mr     r4,r6   #return value from storage
```

Notes:

1. The semantics given for “Compare and Swap” above are based on those of the IBM System/370 Compare and Swap instruction. Other architectures may define a Compare and Swap instruction differently.
2. “Compare and Swap” is shown primarily for pedagogical reasons. It is useful on machines that lack the better synchronization facilities provided by ***lwarx*** and ***stwcx.***. A major weakness of a System/370-style Compare and Swap instruction is that, although the instruction itself is atomic, it checks only that the old and current values of the word being tested are equal, with the result that programs that use such a Compare and Swap to control a shared resource can err if the word has been modified and the old value subsequently restored. The sequence shown above has the same weakness.
3. In some applications the second ***bne-*** instruction and/or the ***mr*** instruction can be omitted. The ***bne-*** is needed only if the application requires that if the EQ bit of CR Field 0 on exit indicates “not equal” then (r4) and (r6) are in fact not equal. The ***mr*** is needed only if the application requires that if the comparands are not equal then the word from storage is loaded into the register with which it was compared (rather than into a third register). If either or both of these instructions is omitted, the resulting Compare and Swap does not obey System/370 semantics.

B.2 Lock Acquisition and Release, and Related Techniques

This section gives examples of how dependencies and the *Synchronization* instructions can be used to implement locks, import and export barriers, and similar constructs.

B.2.1 Lock Acquisition and Import Barriers

An “import barrier” is an instruction or sequence of instructions that prevents storage accesses caused by instructions following the barrier from being performed before storage accesses that acquire a lock have been performed. An import barrier can be used to ensure that a shared data structure protected by a lock is not accessed until the lock has been acquired. A ***sync*** instruction can be used as an import barrier, but the approaches shown below will generally yield better performance because they order only the relevant storage accesses.

B.2.1.1 Acquire Lock and Import Shared Storage

If ***lwarx*** and ***stwcx.*** instructions are used to obtain the lock, an import barrier can be constructed by placing an ***isync*** instruction immediately following the loop containing the ***lwarx*** and ***stwcx.***. The following example uses the “Compare and Swap” primitive to acquire the lock.

In this example it is assumed that the address of the lock is in GPR 3, the value indicating that the lock is free is in GPR 4, the value to which the lock should be set is in GPR 5, the old value of the lock is returned in GPR 6, and the address of the shared data structure is in GPR 9.

```
loop:
   lwarx  r6,0,r3 #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 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***<S> instruction can be used instead of the ***isync*** instruction. If ***lwsync***<S> 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***<S> 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 <S>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***<S> 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<S> 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

1. To increase the likelihood that forward progress is made, it is important that looping on ***lwarx***/***stwcx.*** pairs be minimized. For example, in the "Test and Set" sequence shown in Section B.1, this is achieved by testing the old value before attempting the store; were the order reversed, more ***stwcx.*** instructions might be executed, and reservations might more often be lost between the ***lwarx*** and the ***stwcx.***

2. The manner in which ***lwarx*** and ***stwcx.*** are communicated to other processors and mechanisms, and between levels of the storage hierarchy within a given processor, is implementation-dependent. In some implementations performance may be improved by minimizing looping on a ***lwarx*** instruction that fails to return a desired value. For example, in the "Test and Set" sequence shown in Section B.1, if the programmer wishes to stay in the loop until the word loaded is zero, he could change the "bne- exit" to "bne- loop". However, in some implementations better performance may be obtained by using an ordinary Load instruction to do the initial checking of the value, as follows.

```
loop:
 lwz   r5,0(r3)#load the word
 cmpwi r5,0    #loop back if word
 bne-  loop    #  not equal to 0
 lwarx r5,0,r3 #try again, reserving
 cmpwi r5,0    #  (likely to succeed)
 bne-  loop
 stwcx.r4,0,r3 #try to store non-0
 bne-  loop    #loop if lost reserv'n
```

3. In a multiprocessor, livelock is possible if there is a *Store* instruction (or any other instruction that can clear another processor's reservation; see Section 1.7.3.1) 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-S:

Power ISA Operating Environment Architecture - Server Environment

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”, “Data Segment interrupt”, or “Hypervisor 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 “Illegal Instruction type Program interrupt”
- For “system instruction storage error handler” substitute “Instruction Storage interrupt”, “Hypervisor Instruction Storage interrupt”, “Instruction Segment interrupt”, or “Hypervisor 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”.
- For “system trap handler” substitute “Trap type Program interrupt”.

1.2.1 Definitions and Notation

The definitions and notation given in Book I are augmented by the following.

- **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 processor 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 SDR1, of certain lookaside buffers, such as the SLB and TLB, and of the Page Table.
- **exception**
  An error, unusual condition, or external signal, that may set a status bit and may or may not cause an interrupt, depending upon whether the corresponding interrupt is enabled.
- **interrupt**
  The act of changing the machine state in response to an exception, as described in Chapter 6. “Interrupts” on page 459.
- **trap interrupt**
  An interrupt that results from execution of a *Trap* instruction.
- Additional exceptions to the rule that the processor obeys the sequential execution model, beyond those described in the section entitled “Instruction Fetching” in Book I, are the following.
  - A System Reset or Machine Check interrupt may occur. The determination of whether an

instruction is required by the sequential execution model is not affected by the potential occurrence of a System Reset or Machine Check interrupt. (The determination is affected by the potential occurrence of any other kind of interrupt.)

- A context-altering instruction is executed (Chapter 10. "Synchronization Requirements for Context Alterations" on page 489). The context alteration need not take effect until the required subsequent synchronizing operation has occurred.
- A Reference and Change bit is updated by the processor. The update need not be performed with respect to that processor until the required subsequent synchronizing operation has occurred.

- **"must"**
  If hypervisor software violates a rule that is stated using the word "must" (e.g., "this field must be set to 0"), and the rule pertains to the contents of a hypervisor resource, to executing an instruction that can be executed only in hypervisor state, or to accessing storage in real addressing mode, the results are undefined, and may include altering resources belonging to other partitions, causing the system to "hang", etc.
- **hardware**
  Any combination of hard-wired implementation, emulation assist, or interrupt for software assistance. In the last case, the interrupt may be to an architected location or to an implementation-dependent location. Any use of emulation assists or interrupts to implement the architecture is implementation-dependent.
- **privileged state** and **supervisor mode**
  Used interchangeably to refer to a processor state in which privileged facilities are available.
- **problem state** and **user mode**
  Used interchangeably to refer to a processor 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).

Some fields of certain architected storage tables may be written to automatically by the processor, e.g., Reference and Change bits in the Page Table. When the processor writes to such a table, the following rules are obeyed.

- Unless otherwise stated, no defined field other than the one(s) the processor is specifically updating are modified.
- Contents of reserved fields are either preserved by the processor or written as zero.

Programming Note

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.

1.3 General Systems Overview

The processor or processor unit 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 Processor
- instructions executed in the Fixed-Point Processor
- instructions executed in the Floating-Point Processor
- instructions executed in the Vector Processor

Almost all instructions executed in the Branch Processor, Fixed-Point Processor, Floating-Point Processor, and Vector Processor 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 of the processor, control of processor 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 processor is in privileged but non-hypervisor state (see Chapter 2), or an attempt to execute a hypervisor-only instruction (e.g., ***tlbie***) when the processor is in privileged but non-hypervisor state

- the execution of a traced instruction (Trace interrupt)
- the execution of a Vector instruction when the vector processor is unavailable (Vector Unavailable interrupt)

1.5 Synchronization

The synchronization described in this section refers to the state of the processor 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 6.4).

1. The operation causes instruction dispatching (the issuance of instructions by the instruction fetching mechanism to any instruction execution mechanism) to be halted.
2. The operation is not initiated or, in the case of ***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 6.8).
5. The operation ensures that the instructions that follow the operation will be fetched and executed in the context established by the operation. (This requirement dictates that any prefetched instructions be discarded and that any effects and side effects of executing them out-of-order also be discarded, except as described in Section 5.5, “Performing Operations Out-of-Order”.)

Programming Note

A context synchronizing operation is necessarily execution synchronizing; see Section 1.5.2.

Unlike the *Synchronize* instruction, a context synchronizing operation does not affect the order in which storage accesses are performed.

Item 2 permits a choice only for ***isync*** (and ***sync*** and ***ptesync***; see Section 1.5.2) because all other execution synchronizing operations also alter context.

1.5.2 Execution Synchronization

An instruction is *execution synchronizing* if it satisfies items 2 and 3 of the definition of context synchronization (see Section 1.5.1). ***sync*** and ***ptesync*** are treated like ***isync*** with respect to item 2 (i.e., the conditions described in item 2 apply to the completion of ***sync*** and ***ptesync***). Examples of execution synchronizing instructions include ***sync***, ***ptesync***, and ***mtmsrd***.

An instruction is *execution synchronizing* if it satisfies items 2 and 3 of the definition of context synchronization (see Section 1.5.1). ***sync*** and ***ptesync*** are treated like ***isync*** with respect to item 2. The execution synchronizing instructions are ***sync***, ***ptesync***, the ***mtmsr***[***d***]<S> instructions with L=1, and all context synchronizing instructions.

Programming Note

All context synchronizing instructions are execution synchronizing.

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)

2.1 Overview

The Logical Partitioning (LPAR) facility permits processors and portions of real storage to be assigned to logical collections called *partitions*, such that a program executing on a processor in one partition cannot interfere with any program executing on a processor in a different partition. This isolation can be provided for both problem state and privileged 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 processor is assigned to one partition at any given time. A processor 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 processors 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 layout of the Logical Partitioning Control Register (LPCR) is shown in Figure 1 below.

VC	/	VRMASD	//	RMLS	ILE	//	LPES	RMI	HDICE
0	3	12	17	34	38	39	60	62	63

Figure 1. Logical Partitioning Control Register

The contents of the LPCR control a number of aspects of the operation of the processor with respect to a logical partition. Below are shown the bit definitions for the LPCR.

Bit Description

0:2 ***Virtualization Control*** (VC)

Controls the virtualization of partition memory. This field contains two subfields, VPM and ISL.

0:1 ***Virtualized Partition Memory*** (VPM)

This field controls whether VPM mode is enabled as specified below. (See Section 5.7.3.4, “Virtual Real Mode Addressing Mechanism” and Section 5.7.2, “Virtualized Partition Memory (VPM) Mode” for additional information on VPM mode.)

Bit Description

0 This bit controls whether VPM mode is enabled when address translation is disabled
0 - VPM mode disabled
1 - VPM mode enabled

1 This bit controls whether VPM mode is enabled when address translation is enabled
0 - VPM mode disabled
1 - VPM mode enabled

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 and the processor is not in hypervisor state, address translation is performed as if the contents of $SLB_{L||LP}$ 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***, ***slbia***, ***tlbia***, ***tlbie***, and ***slbie***.

3:11 Reserved

12:16 ***Virtual Real Mode Area Segment Descriptor*** (VRMASD)

When address translation is disabled and VPM_0=1, the contents of this field specify the L and LP fields of the segment descriptor that apply for storage references to the virtualized real mode area (VRMA). See Section 5.7.3.4, “Virtual Real Mode Addressing Mechanism” for additional information. The definitions and allowed values of the L and LP fields are the same as for the corresponding fields in the segment descriptor. (See Section 5.7.7.) If VPM_0=0 or address translation is enabled, the setting of the VRMASD has no effect.

Bit Description

0 ***Virtual Page Size Selector Bit 0*** (L)
1:2 Reserved
3:4 ***Virtual Page Size Selector Bits 1:2*** (LP)

Programming Note

17:33 Reserved

34:37 ***Real Mode Limit Selector*** (RMLS)

The RMLS field specifies the largest effective address that can be used by partition software when address translation is disabled. The valid RMLS values are implementation-dependent, and each value corresponds to a maximum effective address of 2^m, where m has a minimum value of 12 and a maximum value equal to the number of bits in the real address size supported by the implementation.

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 6.5), to establish the Endian mode for the interrupt handler.

39:59 Reserved

60:61 ***Logical Partitioning Environment Selector*** (LPES)

Three of the four LPES values are supported. The 0b10 value is reserved.

60 $LPES_0$

Controls whether External interrupts set MSR_{HV} to 1 or leave it unchanged.

61 $LPES_1$

Controls how storage is accessed when address translation is disabled, and whether a subset of interrupts set MSR_{HV} to 1.

Programming Note

$LPES_1$=0 provides an environment in which only the hypervisor can run with address translation disabled and in which all interrupts invoke the hypervisor. This value (along with MSR_{HV}=1) can also be used in a system that is not partitioned, to permit the operating system to access all system resources.

62 ***Real Mode Caching Inhibited Bit*** (RMI)

The RMI bit affects the manner in which storage accesses are performed in hypervisor state when address translation is disabled (see Section 5.7.3.3 on page 424).

Programming Note

Because in real addressing mode all storage is not Caching Inhibited (unless the Real Mode Caching Inhibited bit is 1), software should not map a Caching Inhibited virtual page to storage that is treated as non-Guarded in real addressing mode. Doing so could permit storage locations in the virtual page to be copied into the cache, which could lead to violations of the requirement given in Section 5.8.2.2 on page 441 for changing the value of the I bit. See also Section 5.7.3.3.1 on page 424.

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 6.5.12 on page 473.

See Section 5.7.3 on page 422 (including subsections) and Section 5.7.9 on page 437 for a description of how storage accesses are affected by the setting of $LPES_1$, RMLS, and RMI. See Section 6.5 on page 466 for a description of how the setting of $LPES_{0:1}$ affects the processing of interrupts.

2.3 Real Mode Offset Register (RMOR)

The layout of the Real Mode Offset Register (RMOR) is shown in Figure 2 below.

//	RMO
0	4 ... 63

Bits	Name	Description
4:63	RMO	Real Mode Offset

Figure 2. Real Mode Offset Register

All other fields are reserved.

The supported RMO values are the non-negative multiples of 2^s, where 2^s is the smallest implementation-dependent limit value representable by the contents of the Real Mode Limit Selector field of the LPCR.

The contents of the RMOR affect how some storage accesses are performed as described in Section 5.7.3 on page 422 and Section 5.7.4 on page 426.

2.4 Hypervisor Real Mode Offset Register (HRMOR)

The layout of the Hypervisor Real Mode Offset Register (HRMOR) is shown in Figure 3 below.

//	HRMO
0	4 ... 63

Bits	Name	Description
4:63	HRMO	Real Mode Offset

Figure 3. 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 $12 \le r \le 26$.

The contents of the HRMOR affect how some storage accesses are performed as described in Section 5.7.3 on page 422 and Section 5.7.4 on page 426.

2.5 Logical Partition Identification Register (LPIDR)

The layout of the Logical Partition Identification Register (LPIDR) is shown in Figure 4 below.

LPID
32 ... 63

Bits	Name	Description
32:63	LPID	Logical Partition Identifier

Figure 4. Logical Partition Identification Register

The contents of the LPIDR identify the partition to which the processor is assigned, affecting operations necessary to manage the coherency of some translation lookaside buffers (see Section 5.10.1 on page 454 and Chapter 10 on page 489).

The supported LPID values consist of all non-negative values that are less than an implementation-dependent power of 2, 2^q, where $2^q \geq$ (the maximum number of processors in a system) $\times$ 2.

Programming Note

On some implementations, software must prevent the execution of a ***tlbie*** instruction on any processor for which the contents of the LPIDR is the same as on the processor on which the LPIDR is being modified or is the same as the new value being written to the LPIDR. This restriction can be met with less effort if one partition identity is used only on processors on which no ***tlbie*** instruction is ever executed. This partition can be thought of as the transfer partition used exclusively to move a processor from one partition to another.

2.6 Other Hypervisor Resources

In addition to the resources described above, the following resources are hypervisor resources, accessible to software only when the processor is in hypervisor state.

- All implementation-specific resources, including implementation-specific 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

- DABR, DABRX, EAR (if implemented), HDAR, HDSISR, Hypervisor Decrementer, PIR, PURR, SDR1, and Time Base. (Note: Although the Time Base and the PURR can be altered only by a hypervisor program, the Time Base can be read by all programs and the PURR can be read when the processor 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). Whether 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) is ignored (i.e., treated as a no-op) or causes a Privileged Instruction type Program interrupt is implementation-dependent. An attempt to modify MSR_{ME} in privileged but non-hypervisor state is ignored (i.e., the bit is not changed).

The ***tlbie***, ***tlbiel***, ***tlbia***, and ***tlbsync*** instructions can be executed only in hypervisor state; see the descriptions of these instructions on pages 450 and 453.

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}.

2.7 Sharing Hypervisor Resources

Some hypervisor resources may be shared among processors. 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 processor. The following resources may be shared among processors.

- RMOR (see Section 2.3.)
- HRMOR (see Section 2.4.)
- LPIDR (see Section 2.5.)
- PVR (see Section 4.3.1.)
- SDR1 (see Section 5.7.7.2.)
- Time Base (see Section 7.2.)
- Hypervisor Decrementer (see Section 7.4.)
- certain implementation-specific registers

The set of resources that are shared is implementation-dependent.

Processors that share any of the resources listed above, with the exception of the PIR and the HRMOR, must be in the same partition.

For each field of the LPCR except the RMI field and the HDICE field, software must ensure that the contents of the field are identical among all processors that are in the same partition and are in a state such that the contents of the field could have side effects. (E.g., software must ensure that the contents of $LPCR_{LPES}$ are identical among all processors that are in the same partition and are not in hypervisor state.) For the HDICE field, software must ensure that the contents of the field are identical among all processors that share the Hypervisor Decrementer and are in a state such that the contents of the field could have side effects. (There are no identity requirements for the RMI field.)

2.8 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 set MSR_{HV} to 1 (see Section 6.5), to establish the Endian mode for the interrupt handler. The HILE bit is set, by an implementation-dependent method, during system initialization, and cannot be modified after system initialization.

The contents of the HILE bit must be the same for all processors under the control of a given instance of the hypervisor; otherwise all results are undefined.

Chapter 3. Branch Processor

3.1 Branch Processor Overview

This chapter describes the details concerning the registers and the privileged instructions implemented in the Branch Processor that are not covered in Book I.

3.2 Branch Processor Registers

3.2.1 Machine State Register

The Machine State Register (MSR) is a 64-bit register. This register defines the state of the processor. On interrupt, the MSR bits are altered in accordance with Figure 37 on page 466. The MSR can also be modified by the ***mtmsr***[***d***], ***rfid***, and ***hrfid*** instructions. It can be read by the ***mfmsr*** instruction.

MSR	
0	63

Figure 5. Machine State Register

Below are shown the bit definitions for the Machine State Register.

Bit **Description**

0 ***Sixty-Four-Bit Mode*** (SF)

- 0 The processor is in 32-bit mode.
- 1 The processor is in 64-bit mode.

1:2 Reserved

3 ***Hypervisor State*** (HV)

- **0** The processor is not in hypervisor state.
- **1** If MSR_{PR}=0 the processor is in hypervisor state; otherwise the processor is not in hypervisor state.

Programming Note

The privilege state of the processor is determined by MSR_{HV} and MSR_{PR}, as follows.

HV	PR	
0	0	privileged
0	1	problem
1	0	privileged and hypervisor
1	1	problem

MSR_{HV} can be set to 1 only by the *System Call* instruction and some interrupts. It can be set to 0 only by ***rfid*** and ***hrfid***.

4:37 Reserved

38 ***Vector Available*** (VEC) [Category: Vector]

- **0** The processor cannot execute any vector instructions, including vector loads, stores, and moves.
- **1** The processor can execute vector instructions.

39:46 Reserved

47 Reserved

48 ***External Interrupt Enable*** (EE)

- 0 External and Decrementer interrupts are disabled.
- 1 External and Decrementer interrupts are enabled.

This bit also affects whether Hypervisor Decrementer interrupts are enabled; Section 6.5.12 on page 473.

49 ***Problem State*** (PR)

- 0 The processor is in privileged state.
- 1 The processor 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.

50 ***Floating-Point Available*** (FP)
[Category: Floating-Point]

0 The processor cannot execute any floating-point instructions, including floating-point loads, stores, and moves.
1 The processor can execute floating-point instructions.

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 Chapter 2., “Logical Partitioning (LPAR)”, on page 397.

> **Programming Note**
>
> The only instructions that can alter MSR_{ME} are ***rfid*** and ***hrfid***.

52 ***Floating-Point Exception Mode 0*** (FE0)
[Category: Floating-Point]

See below.

53 ***Single-Step Trace Enable*** (SE)
[Category: Trace]

0 The processor executes instructions normally.
1 The processor generates a Single-Step type Trace interrupt after successfully completing the execution of the next instruction, unless that instruction is ***hrfid*** or ***rfid***, which are never traced. Successful completion means that the instruction caused no other interrupt.

54 ***Branch Trace Enable*** (BE)
[Category: Trace]

0 The processor executes branch instructions normally.
1 The processor generates a Branch type Trace interrupt after completing the execution of a branch instruction, whether or not the branch is taken.

Branch tracing need not be supported on all implementations that support the Trace category. If the function is not implemented, this bit is treated as reserved.

55 ***Floating-Point Exception Mode 1*** (FE1)
[Category: Floating-Point]

See below.

56:57 Reserved

58 ***Instruction Relocate*** (IR)

0 Instruction address translation is disabled.
1 Instruction address translation is enabled.

> **Programming Note**
>
> See the Programming Note in the definition of MSR_{PR}.

59 ***Data Relocate*** (DR)

0 Data address translation is disabled. Effective Address Overflow (EAO) (see Book I) does not occur.
1 Data address translation is enabled. EAO causes a Data Storage interrupt.

> **Programming Note**
>
> See the Programming Note in the definition of MSR_{PR}.

60 Reserved

61 ***Performance Monitor Mark*** (PMM)
[Category: Server.Performance Monitor]

See Appendix B of Book III-S.

62 ***Recoverable Interrupt*** (RI)

0 Interrupt is not recoverable.
1 Interrupt is recoverable.

Additional information about the use of this bit is given in Sections 6.4.3, “Interrupt Processing” on page 463, 6.5.1, “System Reset Interrupt” on page 466, and 6.5.2, “Machine Check Interrupt” on page 467.

63 ***Little-Endian Mode*** (LE)

0 The processor is in Big-Endian mode.
1 The processor is in Little-Endian mode.

> **Programming Note**
>
> The only instructions that can alter MSR_{LE} are ***rfid*** and ***hrfid***.

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

3.3 Branch Processor Instructions

3.3.1 System Linkage Instructions

These instructions provide the means by which a program can call upon the system to perform a service, and by which the system can return from performing a service or from processing an interrupt.

The *System Call* instruction is described in Book I, but only at the level required by an application programmer. A complete description of this instruction appears below.

System Call — *SC-form*

sc LEV

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

$SRR0 \leftarrow_{iea} CIA + 4$
$SRR1_{33:36\ 42:47} \leftarrow 0$
$SRR1_{0:32\ 37:41\ 48:63} \leftarrow MSR_{0:32\ 37:41\ 48:63}$
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 6.5, “Interrupt Definitions” on page 466. The setting of the MSR is affected by the contents of the LEV field. LEV values greater than 1 are reserved. Bits 0:5 of the LEV field (instruction bits 20:25) are treated as a reserved field.

The interrupt causes the next instruction to be fetched from effective address 0x0000_0000_0000_0C00.

This instruction is context synchronizing.

Special Registers Altered:
SRR0 SRR1 MSR

Programming Note

If LEV=1 the hypervisor is invoked.

If $LPES_1$=1, executing this instruction with LEV=1 is the only way that executing an instruction can cause hypervisor state to be entered.

Because this instruction is not privileged, it is possible for application software to invoke the hypervisor. However, such invocation should be considered a programming error.

Programming Note

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

Return From Interrupt Doubleword XL-form

rfid

19	///	///	///	18	/
0	6	11	16	21	31

$MSR_{51} \leftarrow (MSR_3 \ \& \ SRR1_{51}) \ | \ ((\neg MSR_3) \ \& \ MSR_{51})$
$MSR_3 \leftarrow MSR_3 \ \& \ SRR1_3$
$MSR_{48} \leftarrow SRR1_{48} \ | \ SRR1_{49}$
$MSR_{58} \leftarrow SRR1_{58} \ | \ SRR1_{49}$
$MSR_{59} \leftarrow SRR1_{59} \ | \ SRR1_{49}$
$MSR_{0:2\ 4:32\ 37:41\ 49:50\ 52:57\ 60:63} \leftarrow SRR1_{0:2\ 4:32\ 37:41\ 49:50\ 52:57\ 60:63}$
$NIA \leftarrow_{iea} SRR0_{0:61} \ || \ 0b00$

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 ORing bits 58 and 49 of SRR1 is placed into MSR_{58}. The result of ORing bits 59 and 49 of SRR1 is placed into MSR_{59}. Bits 0:2, 4:32, 37:41, 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 or HSRR0 by the interrupt processing mechanism (see Section 6.4.3) is the address of the instruction that would have been executed next had the interrupt not occurred.

This instruction is privileged and context synchronizing.

Special Registers Altered:
MSR

Programming Note

If this instruction sets MSR_{PR} to 1, it also sets MSR_{EE}, MSR_{IR}, and MSR_{DR} to 1.

Hypervisor Return From Interrupt Doubleword XL-form

hrfid

19	///	///	///	274	/
0	6	11	16	21	31

$MSR_{48} \leftarrow HSRR1_{48} \ | \ HSRR1_{49}$
$MSR_{58} \leftarrow HSRR1_{58} \ | \ HSRR1_{49}$
$MSR_{59} \leftarrow HSRR1_{59} \ | \ HSRR1_{49}$
$MSR_{0:32\ 37:41\ 49:57\ 60:63} \leftarrow HSRR1_{0:32\ 37:41\ 49:57\ 60:63}$
$NIA \leftarrow_{iea} HSRR0_{0:61} \ || \ 0b00$

The result of ORing bits 48 and 49 of HSRR1 is placed into MSR_{48}. The result of ORing bits 58 and 49 of HSRR1 is placed into MSR_{58}. The result of ORing bits 59 and 49 of HSRR1 is placed into MSR_{59}. Bits 0:32, 37:41, 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 or HSRR0 by the interrupt processing mechanism (see Section 6.4.3) is the address of the instruction that would have been executed next had the interrupt not occurred.

This instruction is privileged and context synchronizing, and can be executed only in hypervisor state. If it is executed in privileged but non-hypervisor state either a Privileged Instruction type Program interrupt occurs or the results are boundedly undefined.

Special Registers Altered:
MSR

Programming Note

If this instruction sets MSR_{PR} to 1, it also sets MSR_{EE}, MSR_{IR}, and MSR_{DR} to 1.

Chapter 4. Fixed-Point Processor

4.1 Fixed-Point Processor Overview

This chapter describes the details concerning the registers and the privileged instructions implemented in the Fixed-Point Processor that are not covered in Book I.

4.2 Special Purpose Registers

Special Purpose Registers (SPRs) are read and written using the ***mfspr*** (page 414) and ***mtspr*** (page 413) instructions. Most SPRs are defined in other chapters of this book; see the index to locate those definitions.

4.3 Fixed-Point Processor Registers

4.3.1 Processor Version Register

The Processor Version Register (PVR) is a 32-bit read-only register that contains a value identifying the version and revision level of the processor. 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.

Version	Revision
32	48 63

Figure 6. Processor Version Register

The PVR distinguishes between processors that differ in attributes that may affect software. It contains two fields.

Version A 16-bit number that identifies the version of the processor. Different version numbers indicate major differences between processors, such as which categories are supported.

Revision A 16-bit number that distinguishes between implementations of the version. Different revision numbers indicate minor differences between processors having the same version number, such as clock rate and Engineering Change level.

Version numbers are assigned by the Power ISA process. Revision numbers are assigned by an implementation-defined process.

4.3.2 Processor Identification Register

The Processor Identification Register (PIR) is a 32-bit register that contains a value that can be used to distinguish the processor from other processors 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, if provided, is implementation-dependent.

PROCID
32 … 63

Bits	Name	Description
0:31	PROCID	Processor ID

Figure 7. Processor Identification Register

The means by which the PIR is initialized are implementation-dependent.

The PIR is a hypervisor resource; see Chapter 2.

4.3.3 Control Register

The Control Register (CTRL) is a 32-bit register that controls an external I/O pin. This signal may be used for the following:

- driving the RUN Light on a system operator panel
- External interrupt routing
- Performance Monitor Counter incrementing (see Appendix B)

///	RUN
32	63

Bit	Name	Description
63	RUN	Run state bit

All other fields are implementation-dependent.

Figure 8. Control Register

The CTRL RUN can be used by the operating system to indicate when the processor is doing useful work.

The contents of the CTRL can be written by the ***mtspr*** instruction and read by the ***mfspr*** instruction. Write access to the CTRL is privileged. Reads can be performed in privileged or problem state.

4.3.4 Program Priority Register

The Program Priority Register (PPR) is a 64-bit register that controls the program's priority. The layout of the PPR is shown in Figure 9. A subset of the PRI values may be set by problem state programs (see Section 3.2.3 of Book I).

///	PRI	///	imp-specific
0	11	14	44 … 63

Bit(s) Description

11:13 ***Program Priority*** (PRI)

001 very low
010 low
011 medium low
100 medium (normal)
101 medium high
110 high
111 very high

44:63 Implementation-specific

Figure 9. Program Priority Register

4.3.5 Software-use SPRs

Software-use SPRs are 64-bit registers provided for use by software.

SPRG0
SPRG1
SPRG2
SPRG3

0 63

Figure 10. 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 processor. One or more of the registers is likely to be needed by non-hypervisor interrupt handler programs (e.g., as scratch registers and/or pointers to per processor 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.

HSPRG0
HSPRG1

0 63

Figure 11. 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 processor. One or more of the registers is likely to be needed by hypervisor interrupt handler programs (e.g., as scratch registers and/or pointers to per processor save areas).

4.4 Fixed-Point Processor Instructions

4.4.1 Fixed-Point Storage Access Instructions [Category: Load/Store Quadword]

Load Quadword *DQ-form*

lq RT,DQ(RA)

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

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + EXTS(DQ || 0b0000)
RT ← MEM(EA, 8)
GPR(RT+1) ← MEM(EA+8, 8)
```

Let the effective address (EA) be the sum (RA|0)+(DQ||0b0000). The quadword in storage addressed by EA is loaded into registers RT and RT+1, in increasing order of storage address and register number.

EA must be a multiple of 16. If it is not, an Alignment interrupt occurs.

If RT is odd or RT=RA, the instruction form is invalid. If RT=RA, an attempt to execute this instruction causes an Illegal Instruction type Program interrupt. (The RT=RA case includes the case of RT=RA=0.)

This instruction is not supported in Little-Endian mode. Execution of this instruction in Little-Endian mode causes either an Alignment interrupt or the results are boundedly undefined.

This instruction is privileged.

Special Registers Altered:
None

Store Quadword *DS-form*

stq RS,DS(RA)

62	RS	RA	DS	2
0	6	11	16	30 31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + EXTS(DS || 0b00)
MEM(EA, 8) ← RS
MEM(EA+8, 8) ← GPR(RS+1)
```

Let the effective address (EA) be the sum (RA|0)+(DS||0b00). (RS) and (RS+1) are stored into the quadword in storage addressed by EA, in increasing order of storage address and register number.

EA must be a multiple of 16. If it is not, an Alignment interrupt occurs.

If RS is odd, the instruction form is invalid.

This instruction is not supported in Little-Endian mode. Execution of this instruction in Little-Endian mode causes either an Alignment interrupt or the results are boundedly undefined.

This instruction is privileged.

Special Registers Altered:
None

4.4.2 OR Instruction

or Rx,Rx,Rx can be used to set PPR_{PRI} (see Section 4.3.4) as shown in Figure 12. PPR_{PRI} remains unchanged if the privilege state of the processor executing the instruction is lower than the privilege indicated in the figure. (The encodings available to application programs are also shown in Book I.)

Rx	PPR_{PRI}	Priority	Privileged
31	001	very low	yes
1	010	low	no
6	011	medium low	no
2	100	medium (normal)	no
5	101	medium high	yes
3	110	high	yes
7	111	very high	hypv

Figure 12. Priority levels for *or Rx,Rx,Rx*

4.4.3 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 below, the “defined” SPR numbers are the SPR numbers shown in the figure for the instruction and the implementation-specific SPR numbers that are implemented, and similarly for “defined” registers.

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. See Appendix A, “Assembler Extended Mnemonics” on page 493.

decimal	SPR[1]		Register Name	Privileged		Length (bits)	Cat[2]
	$spr_{5:9}$	$spr_{0:4}$		mtspr	mfspr		
1	00000	00001	XER	no	no	64	B
8	00000	01000	LR	no	no	64	B
9	00000	01001	CTR	no	no	64	B
18	00000	10010	DSISR	yes	yes	32	S
19	00000	10011	DAR	yes	yes	64	S
22	00000	10110	DEC	yes	yes	32	B
25	00000	11001	SDR1	hypv[3]	yes	64	S
26	00000	11010	SRR0	yes	yes	64	B
27	00000	11011	SRR1	yes	yes	64	B
29	00000	11101	AMR	yes	yes	64	S
136	00100	01000	CTRL	-	no	32	S
152	00100	11000	CTRL	yes	-	32	S
256	01000	00000	VRSAVE	no	no	32	V
259	01000	00011	SPRG3	-	no	64	B
268	01000	01100	TB	-	no	64	B
269	01000	01100	TBU	-	no	32	B
272-275	01000	100xx	SPRG[0-3]	yes	yes	64	B
282	01000	11010	EAR	hypv[3]	yes	32	EC
284	01000	11100	TBL	hypv[3]	-	32	B
285	01000	11101	TBU	hypv[3]	-	32	B
286	01000	11110	TBU40	hypv	-	64	S
287	01000	11111	PVR	-	yes	32	B
304	01001	10000	HSPRG0	hypv[3]	hypv[3]	64	S
305	01001	10001	HSPRG1	hypv[3]	hypv[3]	64	S
306	01001	10010	HDSISR	hypv[3]	hypv[3]	32	B
307	01001	10011	HDAR	hypv[3]	hypv[3]	64	B
309	01001	10101	PURR	hypv[3]	yes	64	S
310	01001	10110	HDEC	hypv[3]	yes	32	S
312	01001	11000	RMOR	hypv[3]	hypv[3]	64	S
313	01001	11001	HRMOR	hypv[3]	hypv[3]	64	S
314	01001	11010	HSRR0	hypv[3]	hypv[3]	64	S
315	01001	11011	HSRR1	hypv[3]	hypv[3]	64	S
318	01001	11110	LPCR	hypv[3]	hypv[3]	64	S
319	01001	11111	LPIDR	hypv[3]	hypv[3]	32	S
768-783	11000	0xxxx	perf_mon	-	no	64	S.PM
784-799	11000	1xxxx	perf_mon	yes	yes	64	S.PM
896	11100	00000	PPR	no	no	64	S
1013	11111	10101	DABR	hypv[3]	yes	64	S
1015	11111	10111	DABRX	hypv[3]	yes	64	S
1023	11111	11111	PIR	-	yes	32	S

- 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] See Section 1.3.5 of Book I.

[3] This register is a hypervisor resource, and can be modified by this instruction only in hypervisor state (see Chapter 2).

All SPR numbers that are not shown above and are not implementation-specific are reserved.

Figure 13. SPR encodings

Move To Special Purpose Register XFX-form

mtspr SPR,RS

31	RS	spr	467	/
0	6	11	21	31

```
n ← spr5:9 || spr0:4
if length(SPR(n)) = 64 then
  SPR(n) ← (RS)
else
  SPR(n) ← (RS)32:63
```

The SPR field denotes a Special Purpose Register, encoded as shown in Figure 13. The contents of register RS are placed into the designated Special Purpose Register. For Special Purpose Registers that are 32 bits long, the low-order 32 bits of RS are placed into the SPR.

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 a defined and privileged register when MSR_{PR}=1 causes a Privileged Instruction type Program interrupt. Execution of this instruction specifying a hypervisor resource when $MSR_{HV\ PR}$ = 0b00 either has no effect or causes a Privileged Instruction type Program interrupt (Chapter 2., “Logical Partitioning (LPAR)”, on page 397).

Execution of this instruction specifying an SPR number that is not defined for the implementation causes either an Illegal Instruction type Program interrupt or one of the following.

- if spr_0=0: boundedly undefined results
- if spr_0=1:
  - if MSR_{PR}=1: Privileged Instruction type Program interrupt
  - if MSR_{PR}=0 and MSR_{HV}=0: boundedly undefined results
  - if MSR_{PR}=0 and MSR_{HV}=1: undefined results

If the SPR number is set to a value that is shown in Figure 13 but corresponds to an optional Special Purpose Register that is not provided by the implementation, the effect of executing this instruction is the same as if the SPR number were reserved.

Special Registers Altered:

See Figure 13

Programming Note

For a discussion of software synchronization requirements when altering certain Special Purpose Registers, see Chapter 10. “Synchronization Requirements for Context Alterations” on page 489.

Move From Special Purpose Register XFX-form

mfspr RT,SPR

31	RT	spr	339	/
0	6	11	21	31

```
n ← spr5:9 || spr0:4
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 Figure 13. 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.

spr_0=1 if and only if reading the register is privileged. Execution of this instruction specifying a defined and privileged register when MSR_{PR}=1 causes a Privileged Instruction type Program interrupt.

Execution of this instruction specifying an SPR number that is not defined for the implementation causes either an Illegal Instruction type Program interrupt or one of the following.

- if spr_0=0: boundedly undefined results
- if spr_0=1:
  - if MSR_{PR}=1: Privileged Instruction type Program interrupt
  - if MSR_{PR}=0: boundedly undefined results

If the SPR field contains a value that is shown in Figure 13 but corresponds to an optional Special Purpose Register that is not provided by the implementation, the effect of executing this instruction is the same as if the SPR number were reserved.

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
   MSR59 ← (RS)59 | (RS)49
   MSR32:47 49:50 52:57 60:62 ←(RS)32:47 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 ORing bits 58 and 49 of register RS is placed into MSR_{58}. The result of ORing bits 59 and 49 of register RS is placed into MSR_{59}. Bits 32:47, 49:50, 52:57, and 60:62 of register RS are placed into the corresponding bits of the MSR.

L=1:

Bits 48 and 62 of register RS are placed into the corresponding bits of the MSR. The remaining bits of the MSR are unchanged.

This instruction is privileged.

If L=0 this instruction is context synchronizing. If L=1 this instruction is execution synchronizing; in addition, the alterations of the EE and RI bits take effect as soon as the instruction completes.

Special Registers Altered:
MSR

Except in the ***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.

This instruction does not alter MSR_{ME} or MSR_{LE}. (This instruction does not alter MSR_{HV} because it does not alter any of the high-order 32 bits of the MSR.)

If the only MSR bits to be altered are $MSR_{EE\ RI}$, to obtain the best performance L=1 should be used.

Programming Note

If MSR_{EE}=0 and an External or Decrementer exception is pending, executing an ***mtmsr*** instruction that sets MSR_{EE} to 1 will cause the External or Decrementer interrupt to occur before the next instruction is executed, if no higher priority exception exists (see Section 6.8, "Interrupt Priorities" on page 479). 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 10.

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
   MSR59 ← (RS)59 | (RS)49
   MSR0:2 4:47 49:50 52:57 60:62 ← (RS)0:2 4:47 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 ORing bits 58 and 49 of register RS is placed into MSR_{58}. The result of ORing bits 59 and 49 of register RS is placed into MSR_{59}. Bits 0:2, 4:47, 49:50, 52:57, and 60:62 of register RS are placed into the corresponding bits of the MSR.

L=1:

Bits 48 and 62 of register RS are placed into the corresponding bits of the MSR. The remaining bits of the MSR are unchanged.

This instruction is privileged.

If L=0 this instruction is context synchronizing. If L=1 this instruction is execution synchronizing; in addition, the alterations of the EE and RI bits take effect as soon as the instruction completes.

Special Registers Altered:
MSR

Except in the ***mtmsrd*** instruction description in this section, references to "***mtmsrd***" in this document imply either L value unless otherwise stated or obvious from context (e.g., a reference to an ***mtmsrd*** instruction that modifies an MSR bit other than the EE or RI bit implies L=0).

Programming Note

If this instruction sets MSR_{PR} to 1, it also sets MSR_{EE}, MSR_{IR}, and MSR_{DR} to 1.

This instruction does not alter MSR_{LE}, MSR_{ME} or MSR_{HV}.

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 or Decrementer exception is pending, executing an ***mtmsrd*** instruction that sets MSR_{EE} to 1 will cause the External or Decrementer interrupt to occur before the next instruction is executed, if no higher priority exception exists (see Section 6.8, "Interrupt Priorities" on page 479). 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 10.

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

31	RT	///	///	83	/
0	6	11	16	21	31

```
RT ← MSR
```

The contents of the MSR are placed into register RT.

This instruction is privileged.

Special Registers Altered:
None

Chapter 5. Storage Control

5.1 Overview

A program references storage using the effective address computed by the processor when it executes a *Load*, *Store*, *Branch*, or *Cache Management* instruction, or when it fetches the next sequential instruction.

The effective address is translated to a real address according to procedures described in Section 5.7.3, in Section 5.7.5 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.

5.2 Storage Exceptions

A *storage exception* results when the sequential execution model requires that a storage access be performed but the access is not permitted (e.g., is not permitted by the storage protection mechanism), the access cannot be performed because the effective address cannot be translated to a real address, or the access matches some tracking mechanism criteria (e.g., Data Address Breakpoint).

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 2.1 of Book II, and Section 6.6 in this Book.

5.3 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 5.7.3.

MSR_{IR}=1

The effective address of the instruction is translated by the Address Translation mechanism described beginning in Section 5.7.5.

5.3.1 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, and possibly other implementation-dependent registers), may have the side effect of changing the addresses, effective or real, from which the current instruction stream is being fetched. This side effect is called an *implicit branch*. For example, an ***mtmsrd*** instruction that changes the value of MSR_{SF} may change the effective addresses from which the current instruction stream is being fetched. The MSR bits and System Registers (excluding implementation-dependent registers) for which alteration can cause an implicit branch are indicated as such in Chapter 10. "Synchronization Requirements for Context Alterations" on page 489. Implicit branches are not supported by the Power ISA. If an implicit branch occurs, the results are boundedly undefined.

5.3.2 Address Wrapping Combined with Changing MSR Bit SF

If the current instruction is at effective address 2^{32} - 4 and is an ***mtmsrd*** instruction that changes the contents of MSR_{SF}, the effective address of the next sequential instruction is undefined.

Programming Note

In the case described in the preceding paragraph, if an interrupt occurs before the next sequential instruction is executed, the contents of SRR0, or HSRR0, as appropriate to the interrupt, are undefined.

5.4 Data Access

Data accesses are controlled by MSR_{DR}.

MSR_{DR}=0

The effective address of the data is interpreted as described in Section 5.7.3.

MSR_{DR}=1

The effective address of the data is translated by the Address Translation mechanism described in Section 5.7.5.

5.5 Performing Operations Out-of-Order

An operation is said to be performed "in-order" if, at the time that it is performed, it is known to be required by the sequential execution model. An operation is said to be performed "out-of-order" if, at the time that it is performed, it is not known to be required by the sequential execution model.

Operations are performed out-of-order by the processor 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, the processor performs operations out-of-order when it has resources 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, the processor abandons any results of the operation (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 5.8.1.1.

The only permitted side effects of performing an operation out-of-order are the following.

- A Machine Check or Checkstop that could be caused by in-order execution may occur out-of-order, except as described in Section 5.7.3.3.1 for the Real Mode Storage Control facility.
- On implementations which support Reference and Change bits, these bits may be set as described in Section 5.7.8.
- 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.

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 or made accessible to a partition using the Offset Real Mode Address mechanism (see Section 5.7.3.3) does not exist.

5.6 Invalid Real Address

A storage access (including an access that is performed out-of-order; see Section 5.5) may cause a Machine Check if the accessed storage location contains an uncorrectable error or does not exist.

In the case that the accessed storage location does not exist, the Checkstop state may be entered. See Section 6.5.2 on page 467.

5.7 Storage Addressing

Storage Control Overview

- Real address space size is 2^m bytes, m≤60; see Note 1.
- Real page size is 2^{12} bytes (4 KB).
- Effective address space size is 2^{64} bytes.
- An effective address is translated to a virtual address via the Segment Lookaside Buffer (SLB).
  - Virtual address space size is 2^n bytes, 65≤n≤78; see Note 2.
  - Segment size is 2^s bytes, s=28 or 40.
  - 2^{n-40} ≤ number of virtual segments ≤ 2^{n-28}; see Note 2.
  - 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 4KB, 64 KB, and an implementation-dependent number of other sizes are supported; see Note 3.
  - Segments contain pages of a single size or a mixture of 4KB and 64KB pages
- A virtual address is translated to a real address via the Page Table.

Notes:

1. The value of m is implementation-dependent (subject to the maximum given above). When used to address storage, the high-order 60-m bits of the "60-bit" real address must be zeros.
2. 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.
3. The supported values of p for the larger virtual page sizes are implementation-dependent (subject to the limitations given above).

5.7.1 32-Bit Mode

The computation of the 64-bit effective address is independent of whether the processor is in 32-bit mode or 64-bit mode. In 32-bit mode (MSR_{SF}=0), the high-order 32 bits of the 64-bit effective address are treated as zeros for the purpose of addressing storage. This applies to both data accesses and instruction fetches. It applies independent of whether address translation is enabled or disabled. This truncation of the effective address is the only respect in which storage accesses in 32-bit mode differ from those in 64-bit mode.

Programming Note

Treating the high-order 32 bits of the effective address as zeros effectively truncates the 64-bit effective address to a 32-bit effective address such as would have been generated on a 32-bit implementation of the Power ISA. Thus, for example, the ESID in 32-bit mode is the high-order four bits of this truncated effective address; the ESID thus lies in the range 0-15. When address translation is enabled, these four bits would select a Segment Register on a 32-bit implementation of the Power ISA. The SLB entries that translate these 16 ESIDs can be used to emulate these Segment Registers.

5.7.2 Virtualized Partition Memory (VPM) Mode

VPM mode enables the hypervisor to reassign all or part of a partition's memory transparently so that the reassignment is not visible to the partition. When this is done, the partition's memory is said to be "virtualized." The VPM field in the LPCR enables VPM mode separately when address translation is enabled and when translation is disabled.

If the processor is not in hypervisor state, and either address translation is enabled and VPM_1=1, or address translation is disabled and VPM_0=1, 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 processor in hypervisor state, they permit the hypervisor to handle the condition if appropriate (e.g., to restore the contents of a page that was reassigned), and to reflect it to the operating system's Data Storage or Instruction Storage interrupt handler otherwise.

When address translation is enabled, VPM mode has no effect on address translation. When address translation is disabled, addressing is controlled as specified in Section 5.7.3.

5.7.3 Real And Virtual Real Addressing Modes

When a storage access is an instruction fetch performed when instruction address translation is disabled, or if the access is a data access and data address translation is disabled, it is said to be performed in "real addressing mode" if VPM_0=0 and the processor is not in hypervisor state. If the processor is in hypervisor state, the access is said to be performed in "hypervisor real addressing mode" regardless of the value of VPM_0. If the processor is not in hypervisor state and VPM_0=1, the access is said to be performed in "virtual real addressing mode." Storage accesses in real, hypervisor real, and virtual real addressing modes are performed in a manner that depends on the contents of MSR_{HV}, LPES, VPM, VRMASD, HRMOR, RMLS, and RMOR (see Chapter 2), and bit 0 of the effective address (EA_0) as described below. Bit 1 of the effective address is ignored.

MSR_{HV}=1

- If EA_0=0, the Hypervisor Offset Real Mode Address mechanism, described in Section 5.7.3.1, 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 $LPES_1$=0, the access causes a storage exception as described in Section 5.7.9.3.
- If $LPES_1$=1 and VPM_0=0, the Offset Real Mode Address mechanism, described in Section 5.7.3.2, controls the access.
- If $LPES_1$=1 and VPM_0=1, the Virtual Real Mode Addressing mechanism, described in Section 5.7.3.4, controls the access.

5.7.3.1 Hypervisor Offset Real Mode Address

If MSR_{HV} = 1 and EA_0 = 0, the access is controlled by the contents of the Hypervisor Real Mode Offset Register, as follows.

Hypervisor Real Mode Offset Register (HRMOR)

Bits 4:63 of the effective address for the access are ORed with the 60-bit offset represented by the contents of the HRMOR, and the 60-bit result is used as the real address for the access. The supported offset values are all values of the form $i\times2^r$, where $0 \le i < 2^j$, and j and r are implementation-dependent values having the properties that $12 \le r \le 26$ (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, $EA_{4:63-m}$ and $HRMOR_{0:59-m}$ must be zeros.

Software must ensure that altering the HRMOR does not cause an implicit branch.

5.7.3.2 Offset Real Mode Address

If VPM_0=0, MSR_{HV}=0, and $LPES_1$=1, the access is controlled by the contents of the Real Mode Limit Selector and Real Mode Offset Register, as specified below, and the set of storage locations accessible by code is referred to as the Real Mode Area (RMA).

Real Mode Limit Selector (RMLS)

If bits 4:63 of effective address for the access are greater than or equal to the value (limit) represented by the contents of the RMLR, the access causes a storage exception (see Section 5.7.9.3). In this comparison, if m<60, bits 4:63-m of the effective address may be ignored (i.e., treated as if they were zeros), where the real address size supported by the implementation is m bits. The supported limit values are of the form 2^j, where $12 \le j \le 60$. Subject to the preceding sentence, the number and values of the limits supported are implementation-dependent.

Real Mode Offset Register (RMOR)

If the access is permitted by the RMLR, bits 4:63 of the effective address for the access are ORed with the 60-bit offset represented by the contents of the RMOR, and the low-order m bits of the 60-bit result are used as the real address for the access. The supported offset values are all values of the form $i\times2^s$, where $0 \le i < 2^k$, and k and s are implementation-dependent values having the properties that 2^s is the minimum limit value supported by the implementation (i.e., the minimum value representable by the contents of the RMLR) and k+s = m.

Programming Note

The offset specified by the RMOR should be a nonzero multiple of the limit specified by the RMLS. If these registers are set thus, ORing the effective address with the offset produces a result that is equivalent to adding the effective address and the offset. (The offset must not be zero, because real page 0 contains the fixed interrupt vectors and real pages 1 and 2 may be used for implementation-specific purposes; see Section 5.7.4, "Address Ranges Having Defined Uses" on page 426.)

5.7.3.3 Storage Control Attributes for Accesses in Real and Hypervisor Real Addressing Modes

Storage accesses in hypervisor real addressing mode are performed as though all of storage had the following storage control attributes, except as modified by the Real Mode Storage Control facility (see Section 5.7.3.3.1). (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 if the Real Mode Caching Inhibited bit is set to 0; Caching Inhibited, for data accesses if the Real Mode Caching Inhibited bit is set to 1
- Memory Coherence Required, for data accesses
- Guarded

Storage accesses in real addressing mode are performed as though all of storage had the following storage control attributes. (Such accesses use the Offset Real Mode Address mechanism.)

- not Write Through Required
- not Caching Inhibited
- Memory Coherence Required, for data accesses
- not Guarded

Additionally, storage accesses in real or hypervisor real addressing modes are performed as though all storage was not No-execute.

Software must ensure that any data storage location that is accessed with the Real Mode Caching Inhibited bit set to 1 is not in the caches.

Software must ensure that the Real Mode Caching Inhibited bit contains 0 whenever data address translation is enabled and whenever the processor is not in hypervisor state.

Programming Note

Because storage accesses in real addressing mode and hypervisor real addressing mode 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).

The Real Mode Caching Inhibited bit 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 bit should normally contain 0. Software would set the bit to 1 just before accessing the control register, access the control register as needed, and then set the bit back to 0.

5.7.3.3.1 Hypervisor Real Mode Storage Control

The Hypervisor Real Mode Storage Control facility provides a means of specifying portions of real storage that are treated as non-Guarded in hypervisor real addressing mode ($MSR_{HV\ PR}$=0b10, and MSR_{IR}=0 or MSR_{DR}=0, as appropriate for the type of access). The remaining portions are treated as Guarded in hypervisor real addressing mode. The means is a hypervisor resource (see Chapter 2), and may also be system-specific.

If the Real Mode Caching Inhibited (RMI) bit is set to 1, it is undefined whether a given data access to a storage location that is treated as non-Guarded in hypervisor real addressing mode is treated as Caching Inhibited or as not Caching Inhibited. If the access is treated as Caching Inhibited and is performed out-of-order, the access cannot cause a Machine Check or Checkstop to occur out-of-order due to violation of the requirements given in Section 5.8.2.2 for changing the value of the effective I bit. (Recall that software must ensure that RMI = 0 when the processor is not in hypervisor real addressing mode; see Section 5.7.3.3.)

The facility does not apply to implicit accesses to the Page Table by the processor in performing address translation or in recording reference and change information. These accesses are performed as described in Section 5.7.3.3.

Programming Note

The preceding capability can be used to improve the performance of hypervisor software that runs in hypervisor real addressing mode, by causing accesses to instructions and data that occupy well-behaved storage to be treated as non-Guarded. See also the second paragraph of the Programming Note in Section 5.7.3.3.

If RMI=1, the statement in Section 5.5, that non-Guarded storage locations may be fetched out-of-order into a cache only if they could be fetched into that cache by in-order execution does not preclude the out-of-order fetching into the data cache of storage locations that are treated as non-Guarded in hypervisor real addressing mode, because the effective RMI value that could be used for an in-order data access to such a storage location is undefined and hence could be 0.

5.7.3.4 Virtual Real Mode Addressing Mechanism

If VPM_0=1, MSR_{HV}=0, $LPES_1$=1, 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 14 instead of the entry in the SLB corresponding to the ESID, bits 0:3 of the effective address are ignored (i.e. treated as if they were 0s), bits 4:63-m of the effective address may be ignored (where the real address size supported by the implementation is m bits), 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	0x0_01FF_FFFF
K_s	0
K_p	undefined
N	0
L	$VRMASD_L$
C	0
LP	$VRMASD_{LP}$

Figure 14. SLBE for VRMA

If the effective address is not less than 1 TB, a Hypervisor Data Segment or Hypervisor Instruction Segment interrupt may occur.

Programming Note

The C bit in Figure 14 is set to 0 because the implementation-dependent lookaside information associated with the VRMA is expected to be long-lived. See Section 5.9.3.1.

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 14.

Programming Note

All accesses to the RMA are considered not Guarded. The G bit of the associated Page Table Entry determines whether an access to the VRMA is Guarded. Therefore, if an instruction is fetched from the VRMA, a Hypervisor Instruction Storage interrupt will result if G=1 in the associated Page Table Entry.

5.7.3.5 Storage Control Attributes for Implicit Storage Accesses

Implicit accesses to the Page Table by the processor in performing address translation and in recording reference and change information are performed as though the storage occupied by the Page Table had the following storage control attributes.

- not Write Through Required
- not Caching Inhibited
- Memory Coherence Required
- not Guarded

The definition of “performed” given in Book II applies also to these implicit accesses; accesses for performing address translation are considered to be loads in this respect, and accesses for recording reference and change information are considered to be stores. These implicit accesses are ordered by the ***ptesync*** instruction as described in Section 5.9.2.

5.7.4 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 address specified by the HRMOR and RMOR are used similarly to the page for the fixed interrupt vectors.

- Page Table

  A contiguous sequence of real pages beginning at the real address specified by SDR1 contains the Page Table.

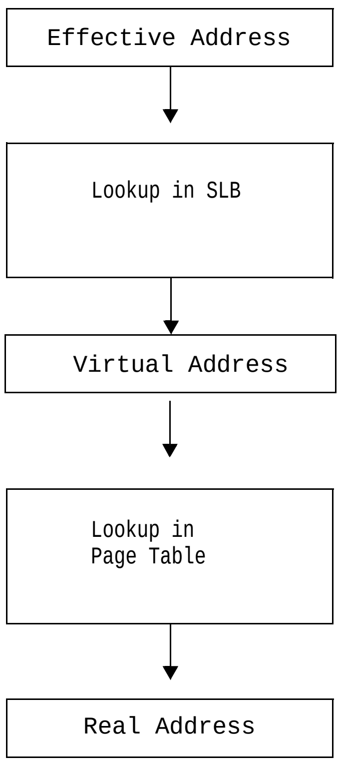
5.7.5 Address Translation Overview

The effective address (EA) is the address generated by the processor 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.

The first step in address translation is to convert the effective address to a virtual address (VA), as described in Section 5.7.6. The second step, conversion of the virtual address to a real address (RA), is described in Section 5.7.7.

If the effective address cannot be translated, a storage exception (see Section 5.2) occurs.

Figure gives an overview of the address translation process.



Address translation overview

5.7.6 Virtual Address Generation

Conversion of a 64-bit effective address to a virtual address is done by searching the Segment Lookaside Buffer (SLB) as shown in Figure 15.

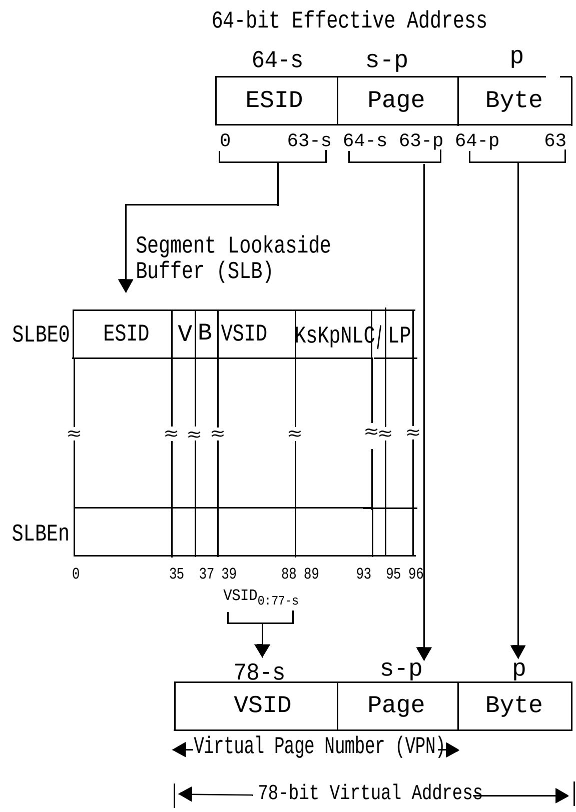


Figure 15. Translation of 64-bit effective address to 78 bit virtual address

5.7.6.1 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 contents of the SLB are managed by software, using the instructions described in Section 5.9.3.1. See Chapter 10. “Synchronization Requirements for Context Alterations” on page 489 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 16 shows the layout of an SLB entry

.

ESID	V	B	VSID	K_sK_pNLC	/	LP
0	36	37	39	89	94	95 96

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 5.7.9.2)
90	K_p	Problem state storage key (See Section 5.7.9.2.)
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 16. SLB Entry

Instructions cannot be executed from a No-execute (N=1) segment.

The L and LP bits specify the page size or sizes that the segment may contain as shown in Figure 17. A Mixed Page Size (MPS) segment is a segment that may contain 4 KB pages, 64 KB pages, or a mixture of both. A Uniform Page Size (UPS) segment is a segment that must contain pages of only a single size.

$SLBE_{L\|\|LP}$	Segment Type	Virtual Page Size(s)
0b000	MPS	4 KB, 64 KB if $PTE_{L\ LP}$ specifies 64 KB page in MPS segment, or both sizes
0b101	UPS	64 KB if $PTE_{L\ LP}$ specifies 64 KB page in UPS segment
additional values[1]	UPS	2^p bytes, where p > 12 and may differ among $SLB_{L\|\|LP}$ values

[1] The "additional values" of $SLB_{L||LP}$ are implementation-dependent, as are the corresponding virtual page sizes.

Figure 17. $SLBL_{L||LP}$ Encoding

For each SLB entry, software must ensure the following requirements are satisfied.

- L||LP contains a value supported by the implementation.
- The 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 processor.

The Class field is used in conjunction with the ***slbie*** instruction (see Section 5.9.3.1).

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

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 10 Note 11.

5.7.6.2 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. If the value of the virtual page size selector field in the matching SLBE is 0b000, then the value of p is the value specified in the PTE used to translate the virtual address (see Section 5.7.7.1); otherwise the value of p is the value specified in the virtual page size selector field in the matching SLBE. If $SLBE_N$ = 1, the N (No-execute) value used for the storage access is 1.

If the SLB search fails, a *segment fault* occurs. This is an Instruction Segment exception or a Data Segment exception, depending on whether the effective address is for an instruction fetch or for a data access.

5.7.7 Virtual to Real Translation

Conversion of a 78-bit virtual address to a real address is done by searching the Page Table as shown in Figure 18.

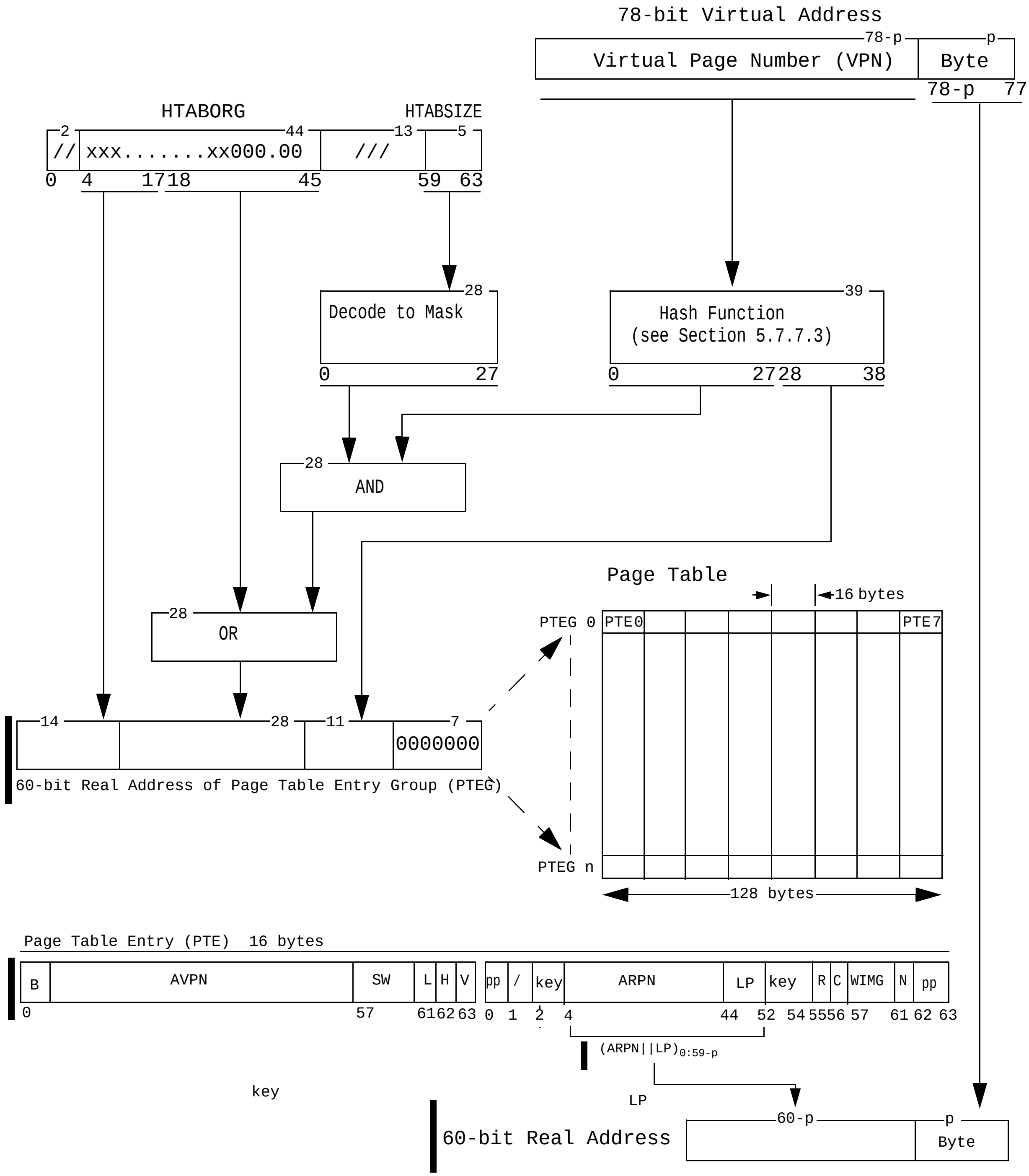


Figure 18. Translation of 78-bit virtual address to 60-bit real address

5.7.7.1 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 0:50 of the address of the first byte in the real page. The HTAB's size must be a multiple of 4 KB, its starting address must be a multiple of its size, and it must be located in storage having the storage control attributes that are used for implicit accesses to it (see Section 5.7.3.3).

The HTAB contains Page Table Entry Groups (PTEGs). A PTEG contains 8 Page Table Entries (PTEs) of 16 bytes each; each PTEG is thus 128 bytes long. PTEGs are entry points for searches of the Page Table.

See Section 5.10 for the rules that software must follow when updating the Page Table.

Programming Note

The Page Table must be treated as a hypervisor resource (see Chapter 2), and therefore must be placed in real storage to which only the hypervisor has write access. Moreover, the contents of the Page Table must be such that non-hypervisor software cannot modify storage that contains hypervisor programs or data.

Page Table Entry

Each Page Table Entry (PTE) maps one VPN to one RPN. Figure 19 shows the layout of a PTE. This layout is independent of the Endian mode of the processor.

0	57	61	62	63
B AVPN	SW	L	H	V

pp	/	key	ARPN	LP	key	R	C	WIMG	N	pp
0	1	2	4	44	52	55	56	57	61	62 63

Dword	Bit(s)	Name	Description
0	0:1	B	Segment Size 0b00 - 256 MB 0b01 - 1 TB 0b10 - reserved 0b11 - reserved
	2:56	AVPN	Abbreviated Virtual Page Number
	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
	4:43	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 19. Page Table Entry

If p≤23, the Abbreviated Virtual Page Number (AVPN) field contains bits 0:54 of the VPN. Otherwise bits 0:77-p of the AVPN field contain bits 0:77-p of the VPN, and bits 78-p:54 of the AVPN field must be zeros and are ignored by the processor.

Programming Note

If p≤23, the AVPN field omits the low-order 23-p bits of the VPN. These bits are not needed in the PTE, because the low-order 11 bits of the VPN are always used in selecting the PTEGs to be searched (see Section 5.7.7.3).

On implementations that support a virtual address size of only n bits, n<78, bits 0:77-n of the AVPN field must be zeros.

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.

If PTE_L=0, the virtual page size is 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 virtual page size is specified by PTE_{LP}. In this case, the contents of PTE_{LP} have the format shown in Figure 20. Bits labelled “r” are bits of the real page number. The page size specified by the non-r bits of PTE_{LP} is implementation-dependent.

```
r r r r _r r r 0
r r r r _r r 0 1
r r r r _r 0 1 1
r r r r _0 1 1 1
r r r 0_1 1 1 1
r r 0 1_1 1 1 1
r 0 1 1_1 1 1 1
0 1 1 1_1 1 1 1
```

Figure 20. Format of PTE_{LP}

There are at least 2 formats of PTE_{LP} that specify a 64 KB page. One format specifies a 64 KB page contained in an MPS segment, and another specifies a 64 K page contained in a Uniform segment.

If L=1, the 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 processor are treated as reserved values for that processor.

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 processor.

Programming Note

The 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

The processor often has implementation-dependent lookaside buffers (e.g. TLBs and ERATs) used to cache translations of recently used storage addresses. Mapping virtual storage to large pages may increase the effectiveness of such lookaside buffers, improving performance, because it is possible for such buffers to translate a larger range of addresses, reducing the frequency that the Page Table must be searched to translate an address.

Instructions cannot be executed from a No-execute (N=1) page.

Page Table Size

The number of entries in the Page Table directly affects performance because it influences the hit ratio in the Page Table and thus the rate of page faults. If the table is too small, it is possible that not all the virtual pages that actually have real pages assigned can be mapped via the Page Table. This can happen if too many hash collisions occur and there are more than 16 entries for the same primary/secondary pair of PTEGs. While this situation cannot be guaranteed not to occur for any size Page Table, making the Page Table larger than the minimum size (see Section 5.7.7.2) will reduce the frequency of occurrence of such collisions.

Programming Note

If large pages are not used, it is recommended that the number of PTEGs in the Page Table be at least half the number of real pages to be accessed. For example, if the amount of real storage to be accessed is 2^{31} bytes (2 GB), then we have $2^{31-12}=2^{19}$ real pages. The minimum recommended Page Table size would be 2^{18} PTEGs, or 2^{25} bytes (32 MB).

5.7.7.2 Storage Description Register 1

The Storage Description Register 1 (SDR1) register is shown in Figure 21.

//	HTABORG	///	HTABSIZE
0	4	46	59 63

Bits	Name	Description
4:45	HTABORG	Real address of Page Table
59:63	HTABSIZE	Encoded size of Page Table

All other fields are reserved.

Figure 21. SDR1

SDR1 is a hypervisor resource; see Chapter 2.

The HTABORG field in SDR1 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 a minimum. At least 11 bits from the hash function (see Figure 18) 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 $18 \le n \le 46$. As the table size is increased, more bits are used from the hash to index into the table and the value in HTABORG must have more of its low-order bits equal to 0.

The HTABSIZE field in SDR1 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 18). The number of low-order 0 bits in HTABORG must be greater than or equal to the value in HTABSIZE.

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 5.7.7.3) 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 and the value in HTABORG must have its low-order 3 bits (bits 43:45 of SDR1) equal to 0. This means that the Page Table must begin on a $2^{3+11+7} = 2^{21}$ = 2 MB boundary.

5.7.7.3 Page Table Search

When the hardware searches the Page Table, the accesses are performed as described in Section 5.7.3.3.

An outline of the HTAB search process is shown in Figure 18. Up to two hash functions are used to locate a PTE that may translate the given virtual address.

A 39-bit hash value is computed from the VPN. The value of s is the value specified in the SLBE that was used to generate the virtual address; the value of p used when computing the hash function is 12 if $SLBE_{L||LP}$ =0b000, otherwise the value of p is the value specified in the SLBE.

1. **Primary Hash:**

   If s=28, the hash value is computed by Exclusive ORing $VPN_{11:49}$ with $(^{11+p}0||VPN_{50:77-p})$

   If s=40, the hash value is computed by Exclusive ORing the following three quantities: $(VPN_{24:37} ||^{25}0)$, $(0||VPN_{0:37})$, and $(^{p-1}0||VPN_{38:77-p})$

   The 60-bit real address of a PTEG is formed by concatenating the following values:
   - Bits 4:17 of SDR1 (the high-order 14 bits of HTABORG).
   - Bits 0:27 of the 39-bit hash value ANDed with the mask generated from bits 59:63 of SDR1 (HTABSIZE) and then ORed with bits 18:45 of SDR1 (the low-order 28 bits of 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 s=28, the hash value is computed by taking the ones complement of the Exclusive OR of $VPN_{11:49}$ with $(^{11+p}0||VPN_{50:77-p})$

   If s=40, the hash value is computed by taking the ones complement of the Exclusive OR of the following three quantities: $(VPN_{24:37} \; ||^{25}0)$, $(0||VPN_{0:37})$, and $(^{p-1}0||VPN_{38:77-p})$

   The 60-bit real address of a PTEG is formed by concatenating the following values:
   - Bits 4:17 of SDR1 (the high-order 14 bits of HTABORG).
   - Bits 0:27 of the 39-bit hash value ANDed with the mask generated from bits 59:63 of SDR1 (HTABSIZE) and then ORed with bits 18:45 of SDR1 (the low-order 28 bits of HTABORG).
   - Bits 28:38 of the 39-bit hash value.
   - Seven 0-bits.

   This operation identifies the “secondary PTEG”.

3. As many as 16 PTEs in the two identified PTEGs are tested to determine if any translate the given virtual address. Let q = minimum(54, 77-p). 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=$SLBE_B$
   - $PTE_{AVPN[0:q]}$=$VA_{0:q}$
   - if PTE_L=0 then $SLBE_{L||LP}$ =0b000
     else PTE_{LP} specifies a page size
     specified by $SLBE_{L||LP}$

   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 Page Table search succeeds, the real address (RA) is formed by concatenating bits 0:59-p of (ARPN||LP) from the matching PTE with bits 64-p:63 of the effective address (the byte offset), where the p value is the value specified by $PTE_{L\ LP}$.

$$RA=(ARPN \; || \; LP)_{0:59-p} \; || \; EA_{64-p:63}$$

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.

Programming Note

For segments that may contain a mixture of 4 KB and 64 KB pages (i.e. $SLBE_{L||LP}$ = 0b000), the value of p used when searching the Page Table to identify the PTEGs is specified to be 12. Since the segment may contain pages of size 4KB and 64 KB, the processor searches for PTEs specifying pages of either size, and the real address is formed using a value of p specified by the matching PTE.

If the Page Table search fails, a *page fault* occurs. This is an Instruction Storage exception or a Data Storage exception, depending on whether the effective address is for an instruction fetch or for a data access. The N value used for the storage access is the N bit from the SLB entry that was used to translate the effective address.

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.

Translation Lookaside Buffer

Conceptually, the Page Table is searched by the address relocation hardware to translate every reference. For performance reasons, the hardware usually keeps a Translation Lookaside Buffer (TLB) that holds PTEs that have recently been used. The TLB is searched prior to searching the Page Table. As a consequence, when software makes changes to the Page Table it must perform the appropriate TLB invalidate operations to maintain the consistency of the TLB with the Page Table (see Section 5.10).

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*** or ***tlbia*** instruction to ensure that the TLB no longer contains a mapping for a particular virtual page.

5.7.8 Reference and Change Recording

If address translation is enabled, Reference (R) and Change (C) bits are maintained in the Page Table Entry that is used to translate the virtual address. 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.

Reference and Change bits are set by the processor as described below. Setting the bits need not be atomic with respect to performing the access that caused the bits to be updated. 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, as determined by examining either the Page Table Entry or any corresponding lookaside information (e.g., TLB) maintained by the processor.

Reference Bit

The Reference bit is set to 1 if the corresponding access (load, store, 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.

Change Bit

The Change bit is set to 1 if a *Store* instruction is executed and the store is performed. Otherwise the Change bit may be set to 1 if a *Store* instruction is executed and the store is permitted by the storage protection mechanism and, if the *Store* instruction is executed out-of-order, the instruction would be required by the sequential execution model in the absence of the following kinds of interrupts:

- system-caused interrupts (see Section 6.4 on page 462)
- Floating-Point Enabled Exception type Program interrupts when the processor is in an Imprecise mode

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.***) is executed, but no store is performed.
- A *Store String Word Indexed* instruction (***stswx***) is executed, but the length is zero.
- The *Store* instruction causes a Data Storage exception (for which setting the Change bit is not prohibited).
- 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 processor may search the Page Table for the purpose of setting the Change bit and then re-execute the instruction. When reexecuting the instruction, the processor may search the Page Table a second time. If the Page Table Entry has meanwhile been altered, by a program executing on another processor, the second search may obtain the new contents, which may preclude the store.
- A system-caused interrupt occurs before the store has been performed.

Figure 22 on page 436 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.

In the figure, the “*Load*-type” instructions are the *Load* instructions described in Books I and II, ***eciwx***, and the *Cache Management* instructions that are treated as *Loads*. The “*Store*-type” instructions are the *Store* instructions described in Books I and II, ***ecowx***, and the *Cache Management* instructions that are treated as *Stores*. The “ordinary” *Load* and *Store* instructions are those described in Books I and II. “set” means “set to 1”.

When the processor updates the Reference and Change bits in the Page Table Entry, the accesses are performed as described in Section 5.7.3.3, “Storage Control Attributes for Accesses in Real and Hypervisor Real Addressing Modes” on page 424. The accesses may be performed using operations equivalent to a store to a byte, halfword, word, or doubleword, and are not necessarily performed as an atomic read/modify/write of the affected bytes.

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 processor executing the ***sync*** instruction, before the ***sync*** instruction is executed will be performed with respect to that processor before the ***sync*** instruction's memory barrier is created. There are additional requirements for synchronizing Reference and Change bit updates in multiprocessor systems; see Section 5.10, "Page Table Update Synchronization Requirements" on page 454.

Programming Note

Because the ***sync*** instruction is execution synchronizing, the set of Reference and Change bit updates that are performed with respect to the processor 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, the Reference and Change bits in the associated Page Table Entry are set as for ordinary loads and stores. See Section 5.10 for the rules software must follow when updating Reference and Change bits.

Status of Access	**R**	**C**
Storage protection violation	Acc[1]	No
Out-of-order I-fetch or *Load*-type insn	Acc	No
Out-of-order *Store*-type insn		
Would be required by the sequential execution model in the absence of system-caused or imprecise interrupts[3]	Acc	Acc[1 2]
All other cases	Acc	No
In-order *Load*-type or *Store*-type insn, access not performed		
Load-type insn	Acc	No
Store-type insn	Acc	Acc[2]
Other in-order access		
I-fetch	Yes	No
Ordinary *Load*, ***eciwx***	Yes	No
Other ordinary *Store*, ***ecowx***, ***dcbz***	Yes	Yes
icbi, ***dcbt***, ***dcbtst***, ***dcbst***, ***dcbf***[***l***]	Acc	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 $MSR_{FE0\ FE1}$.

Figure 22. Setting the Reference and Change bits

5.7.9 Storage and Virtual Page Class Key Protection

The storage and virtual page class key protection mechanism provides a means for selectively granting instruction fetch access, granting read access, granting read/write access, and prohibiting access to areas of storage based on a number of control criteria.

The operation of the protection mechanism depends on one or more of the following conditions.

- the state of MSR bits HV, IR,DR, PR
- the value of the key bits in the associated SLB entry
- the values of the page protection and key bits in the associated PTE
- the contents of the Authority Mask Register

When translation is enabled for an access, the access is permitted if and only if the access is permitted by the virtual page class key protection (see Section 5.7.9.1) and the storage protection mechanism (see Section 5.7.9.2). If an instruction fetch is not permitted, an Instruction Storage exception is generated. If a data access is not permitted, a Data Storage exception is generated. (See Section 5.2)

Unless otherwise indicated, references to “storage protection mechanism” or “protection mechanism” throughout the Books refer to both the Storage Protection mechansm and the Virtual Page Class Key Protection mechanism.

When address translation is enabled, a *protection domain* is a range of unmapped effective addresses, a virtual page, or a segment. When address translation is disabled and $LPES_1$=1 there are two protection domains: the set of effective addresses that are less than the value specified by the RMLS, and all other effective addresses. When address translation is disabled and $LPES_1$=0 the entire effective address space comprises a single protection domain. A *protection boundary* is a boundary between protection domains.

5.7.9.1 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 access permissions for each class quickly by modifying the Authority Mask Register (AMR) shown in Figure 23. The access permissions associated with the Virtual Page Class Key protection mechanism apply only to load and store operations when address translation is enabled. The Virtual Page Class Key protection mechanism has no effect on instruction fetches.

Key0	Key1	Key2	. . .	Key29	Key30	Key31
0	2	4	6	58	60	62

Figure 23. Authority Mask Register (AMR)

The contents of the AMR are as follows.

Bit	Description
0:1	Access mask for class number 0
2:3	Access mask for class number 1
...	
2n:2n+1	Access mask for class number n
...	
62:63	Access mask for class number 31

The access mask for each class defines the access permissions used in conjunction with load and store operations corresponding to page table entries containing 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 of the access mask corresponding to class number n.

- An access caused by a Store instruction is permitted if AMR_{2n}=0b0; otherwise the access is not permitted.
- An access caused by a Load instruction is permitted if AMR_{2n+1}=0b0; otherwise the access is not permitted.

Programming Note

If 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.

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 processor 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 processors that comply with versions of the architecture that precede Version 2.04 (and hence implement the Data Address Compare mechanism) and on processors 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 AMR has the same SPR number as 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 implemented by any processor that implements 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 mechanism permit all accesses. Do not use PTE_{KEY} =31.
- When a Data Storage interrupt occurs, if $DSISR_{42}$=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 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.)

5.7.9.2 Storage Protection, Address Translation Enabled

When address translation is enabled, the protection mechanism is controlled both by virtual page class key protection (see Section 5.7.9.1) and 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 5.7.6.1 and 5.7.7.3)
  - 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:

   $$Key \leftarrow (K_p \ \& \ MSR_{PR}) \ | \ (K_s \ \& \ \neg MSR_{PR})$$

   Using the computed Key, Figure 24 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 24. PP bit protection states, address translation enabled

5.7.9.3 Storage Protection, Address Translation Disabled

When address translation is disabled, the protection mechanism is controlled by the following (see Chapter 2 and Section 5.7.3, "Real And Virtual Real Addressing Modes").

- $LPES_1$, which distinguishes between the two modes of accessing storage using the LPAR facility
- MSR_{HV}, which distinguishes between hypervisor state and other privilege states
- RMLS, which specifies the real mode limit value

Using the above values, Figure 25 is applied. The access is permitted for any entry in the figure except "no access".

$LPES_1$	HV	Access Authority
0	0	no access
0	1	read/write
1	0	read/write or no access[1]
1	1	read/write
[1] If VPM_0=1, the access authority is read/write. If VPM_0=0 and the effective address for the access is less than the value specified by the RMLS, the access authority is read/write; otherwise the access is not permitted.		

Figure 25. Protection states, address translation disabled

Programming Note

The comparison described in note 1 in Figure 25 ignores bits 0:3 of the effective address and may ignore bits 4:63-m; see Section 5.7.3.

5.8 Storage Control Attributes

This section describes aspects of the storage control attributes that are relevant only to privileged software programmers. The rest of the description of storage control attributes may be found in Section 1.6 of Book II and subsections.

5.8.1 Guarded Storage

Storage is said to be "well-behaved" if the corresponding real storage exists and is not defective, and if the effects of a single access to it are indistinguishable from the effects of multiple identical accesses to it. Data and instructions can be fetched out-of-order from well-behaved storage without causing undesired side effects.

Storage is said to be Guarded if any of the following conditions is satisfied.

- MSR bit IR or DR is 1 for instruction fetches or data accesses respectively, and the G bit is 1 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 outside the range(s) specified by the Real Mode Storage Control facility (see Section 5.7.3.3.1).

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, 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 Data Storage exception for reasons other than Data Address Breakpoint 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 Breakpoint match are given in Section 8.1.1.)

5.8.1.1 Out-of-Order Accesses to Guarded Storage

In general, Guarded storage is not accessed out-of-order. The only exceptions to this rule are the following.

Load Instruction

If a copy of any byte of the storage operand is in a cache then that byte may be accessed in the cache or in main storage.

Instruction Fetch

If $MSR_{HV\ IR}$=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 Real Mode Storage Control facility specifies that this real page is not Guarded).

5.8.2 Storage Control Bits

When address translation is enabled, 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 26.

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] [Category: Memory Coherence] 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 26. Storage control bits

When address translation is enabled, instructions are not fetched from storage for which the G bit in the Page Table Entry is set to 1; see Section 5.7.9.

When address translation is disabled, the storage control attributes are implicit; see Section 5.7.3.3.

In Section 5.8.2.1 and 5.8.2.2, ”access” includes accesses that are performed out-of-order, and references to W, I, M, and G bits include the values of those bits that are implied when address translation is disabled.

Programming Note

In a uniprocessor system in which only the processor has caches, correct coherent execution does not require the processor to access storage as Memory Coherence Required, and accessing storage as not Memory Coherence Required may give better performance.

5.8.2.1 Storage Control Bit Restrictions

All combinations of W, I, M, and G values are permitted except those for which both W and I are 1.

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 I bit must be the same for all accesses to a given real page.

At any given time, the value of the W bit must be the same for all accesses to a given real page.

5.8.2.2 Altering the Storage Control Bits

When changing the value of the 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***] and ***icbi*** before permitting any other accesses to the page.

When changing the value of the W bit for a given real page from 0 to 1, software must ensure that no processor 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***]

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 processors 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 processor 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 to 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 processor to flush all storage locations accessed with the old M value before permitting the locations to be accessed with the new M value.

Additional requirements for changing the storage control bits in the Page Table are given in Section 5.10.

5.9 Storage Control Instructions

5.9.1 Cache Management Instructions

This section describes aspects of cache management that are relevant only to privileged software programmers.

For a ***dcbz*** instruction that causes the target block to be newly established in the data cache without being fetched from main storage, the processor need not verify that the associated real address is valid. The existence of a data cache block that is associated with an invalid real address (see Section 5.6) can cause a delayed Machine Check interrupt or a delayed Checkstop.

Each implementation provides an efficient means by which software can ensure that all blocks that are considered to be modified in the data cache have been copied to main storage before the processor enters any power conserving mode in which data cache contents are not maintained.

5.9.2 Synchronize Instruction

The *Synchronize* instruction is described in Section 3.3.3 of Book II, but only at the level required by an application programmer (***sync*** with L=0 or L=1). This section describes properties of the instruction that are relevant only to operating system and hypervisor software programmers. This variant of the *Synchronize* instruction is designated the Page Table Entry ***sync*** and is specified by the extended mnemonic ***ptesync*** (equivalent to ***sync*** with L=2).

The ***ptesync*** instruction has all of the properties of ***sync*** with L=0 and also the following additional properties.

- The memory barrier created by the ***ptesync*** instruction provides an ordering function for the storage accesses associated with all instructions that are executed by the processor 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 processor executing the ***ptesync*** instruction, before the ***ptesync*** instruction is executed. The applicable pairs are all pairs a_i,b_j in which b_j is a data access and a_i is not an instruction fetch.
- The ***ptesync*** instruction causes all Reference and Change bit updates associated with address translations that were performed, by the processor executing the ***ptesync*** instruction, before the ***ptesync*** instruction is executed, to be performed with respect to that processor before the ***ptesync*** instruction's memory barrier is created.
- The ***ptesync*** instruction provides an ordering function for all stores to the Page Table caused by *Store* instructions preceding the ***ptesync*** instruction with respect to searches of the Page Table that are performed, by the processor executing the ***ptesync*** instruction, after the ***ptesync*** instruction completes. Executing a ***ptesync*** instruction ensures that all such stores will be performed, with respect to the processor executing the ***ptesync*** instruction, before any implicit accesses to the affected Page Table Entries, by such Page Table searches, are performed with respect to that processor.
- In conjunction with the ***tlbie*** and ***tlbsync*** instructions, the ***ptesync*** instruction provides an ordering function for TLB invalidations and related storage accesses on other processors as described in the ***tlbsync*** instruction description on page 453.

Programming Note

For instructions following a ***ptesync*** instruction, the memory barrier need not order implicit storage accesses for purposes of address translation and reference and change recording.

The functions performed by the ***ptesync*** instruction may take a significant amount of time to complete, so this form of the instruction should be used only if the functions listed above are needed. Otherwise ***sync*** with L=0 should be used (or ***sync*** with L=1, or ***eieio***, if appropriate).

Section 5.10, "Page Table Update Synchronization Requirements" on page 454 gives examples of uses of ***ptesync***.

5.9.3 Lookaside Buffer Management

All implementations have a Segment Lookaside Buffer (SLB). For performance reasons, most implementations also have implementation-specific lookaside information that is used in address translation. This lookaside information may be: 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; etc.; or any combination of these. Lookaside information, including the SLB, is managed using the instructions described in the subsections of this section.

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 implementation-specific lookaside information; exceptions to this rule are described in Section 5.10.1.2.

The effects of the ***slbie***, ***slbia***, and *TLB Management* instructions on address translations, as specified in Sections 5.9.3.1 and 5.9.3.3 for the SLB and TLB respectively, 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 all implementation-specific lookaside information that is derived from SLB entries and PTEs respectively.

The ***tlbia*** instruction is optional. However, 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, RMOR, and HRMOR, when software alters the contents of these registers it must also invalidate the corresponding implementation-specific lookaside information.

All implementations that have such lookaside information provide a means by which software can invalidate all such lookaside information.

For simplicity, elsewhere in the Books it is assumed that the TLB exists.

Programming Note

Because the instructions used to manage implementation-specific lookaside information that is derived from PTEs may be changed in a future version of the architecture, it is recommended that software "encapsulate" uses of the *TLB Management* instructions into subroutines.

Programming Note

The function of all the instructions described in Sections 5.9.3.1 - 5.9.3.3 is independent of whether address translation is enabled or disabled.

For a discussion of software synchronization requirements when invalidating SLB and TLB entries, see Chapter 10.

5.9.3.1 SLB Management Instructions

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 state 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 10.

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
   translation of esid ← undefined
```

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 16 on page 428).

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, the results of translating effective addresses that would have been translated by that SLB entry are undefined, 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. 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 processor ignores the contents of RB listed below and software must set them to 0s.

- $(RB)_{37}$
- $(RB)_{39:63}$
- 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 processors.

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 processor 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 processors maintain in an Effective to Real Address Translation (ERAT) lookaside buffer.)

The recommended use of the Class field is to use the 0 value to indicate that the SLB entry contains a translation that is expected to be long-lived and the 1 value to indicate the SLB entry contains a translation that is expected to be short lived. If this is done and the processor invalidates certain implementation-specific lookaside information based only on the specified class value, an ***slbie*** instruction that invalidates a short-lived translation will preserve such lookaside information for long-lived translations.

The *Move To Segment Register* instructions (see Section 5.9.3.2.1) create SLB entries in which the Class value is 0.

Programming Note

The B value in register RB may be needed for invalidating ERAT entries corresponding to the translation being invalidated.

SLB Invalidate All — *X-form*

slbia

31	///	///	///	498	/
0	6	11	16	21	31

```
for each SLB entry except SLB entry 0
   SLBE_V ← 0
   all other fields of SLBE ← undefined
```

For all SLB entries except SLB entry 0, the V bit in the entry is set to 0, making the entry invalid, and the remaining fields of the entry are set to undefined values. SLB entry 0 is not altered.

This instruction is privileged.

Special Registers Altered:
None

Programming Note

slbia does not affect SLBs on other processors.

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.)

SLB Move To Entry — *X-form*

slbmte RS,RB

31	RS	///	RB	402	/
0	6	11	16	21	31

The SLB entry specified by bits 52:63 of register RB is loaded from register RS and from the remainder of register RB. The contents of these registers are interpreted as shown in Figure 27.

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

Figure 27. GPR contents for slbmte

On implementations that support a virtual address size of only n bits, n<78, $(RS)_{0:77-n}$ must be zeros.

$(RS)_{57}$ and $(RS)_{60:63}$ must be ignored by the processor.

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.

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 cannot be used to invalidate an SLB entry.

This instruction is privileged.

Special Registers Altered:
None

Programming Note

The reason ***slbmte*** cannot be used to invalidate an SLB entry is that it does not necessarily affect implementation-specific address translation lookaside information. ***slbie*** (or ***slbia***) must be used for this purpose.

SLB Move From Entry VSID X-form

slbmfev RT,RB

31	RT	///	RB	851	/
0	6	11	16	21	31

If the SLB entry specified by bits 52:63 of register RB is valid (V=1), the contents of the B, VSID, K_s, K_p, N, L, C, and LP fields of the entry are placed into register RT. The contents of these registers are interpreted as shown in Figure 28.

RT

B	VSID	K_sK_pNLC	0	LP	0s
0	2	52	57	58	60 63

RB

0s	index
0	52 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
$RB_{0:51}$	must be 0x0_0000_0000_0000
$RB_{52:63}$	index, which selects the SLB entry

Figure 28. GPR contents for slbmfev

On implementations that support a virtual address size of only n bits, n<78, $RT_{0:77-n}$ 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.

This instruction is privileged.

Special Registers Altered:
None

SLB Move From Entry ESID X-form

slbmfee RT,RB

31	RT	///	RB	915	/
0	6	11	16	21	31

If the SLB entry specified by bits 52:63 of register RB is valid (V=1), the contents of the ESID and V fields of the entry are placed into register RT. The contents of these registers are interpreted as shown in Figure 29.

RT

ESID	V	0s
0	36	37 63

RB

0s	index
0	52 63

$RT_{0:35}$	ESID
RT_{36}	V
$RT_{37:63}$	set to 0b000 || 0x00_0000
$RB_{0:51}$	must be 0x0_0000_0000_0000
$RB_{52:63}$	index, which selects the SLB entry

Figure 29. GPR contents for slbmfee

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.

This instruction is privileged.

Special Registers Altered:
None

5.9.3.2 Bridge to SLB Architecture [Category:Server.Phased-Out]

The facility described in this section can be used to ease the transition to the current Power ISA software-managed Segment Lookaside Buffer (SLB) architecture, from the Segment Register architecture provided by 32-bit PowerPC implementations. A complete description of the Segment Register architecture may be found in "Segmented Address Translation, 32-Bit Implementations," Section 4.5, Book III of Version 1.10 of the PowerPC architecture, referenced in the introduction to this architecture.

The facility permits the operating system to continue to use the 32-bit PowerPC implementation's *Segment Register Manipulation* instructions.

5.9.3.2.1 Segment Register Manipulation Instructions

The instructions described in this section -- ***mtsr***, ***mtsrin***, ***mfsr***, and ***mfsrin*** -- allow software to associate effective segments 0 through 15 with any of virtual segments 0 through 2^{27}-1. SLB entries 0:15 serve as virtual Segment Registers, with SLB entry i used to emulate Segment Register i. The ***mtsr*** and ***mtsrin*** instructions move 32 bits from a selected GPR to a selected SLB entry. The ***mfsr*** and ***mfsrin*** instructions move 32 bits from a selected SLB entry to a selected GPR.

The contents of the GPRs used by the instructions described in this section are shown in Figure 30. Fields shown as zeros must be zero for the *Move To Segment Register* instructions. Fields shown as hyphens are ignored. Fields shown as periods are ignored by the *Move To Segment Register* instructions and set to zero by the *Move From Segment Register* instructions. Fields shown as colons are ignored by the *Move To Segment Register* instructions and set to undefined values by the *Move From Segment Register* instructions.

RS/RT

: : :	.	K_sK_pN	0	$VSID_{23:49}$
0	32	33	36	37 63

RB

- - -	ESID	- - -
0	32	36 63

Figure 30. GPR contents for *mtsr*, *mtsrin*, *mfsr*, and *mfsrin*

Programming Note

The "Segment Register" format used by the instructions described in this section corresponds to the low-order 32 bits of RS and RT shown in the figure. This format is essentially the same as that for the Segment Registers of 32-bit PowerPC implementations. The only differences are the following.

- Bit 36 corresponds to a reserved bit in Segment Registers. Software must supply 0 for the bit because it corresponds to the L bit in SLB entries, and large pages are not supported for SLB entries created by the *Move To Segment Register* instructions.
- VSID bits 23:25 correspond to reserved bits in Segment Registers. Software can use these extra VSID bits to create VSIDs that are larger than those supported by the *Segment Register Manipulation* instructions of 32-bit PowerPC implementations.

Bit 32 of RS and RT corresponds to the T (direct-store) bit of early 32-bit PowerPC implementations. No corresponding bit exists in SLB entries.

Programming Note

The Programming Note in the introduction to Section 5.9.3.1 applies also to the *Segment Register Manipulation* instructions described in this section, and to any combination of the instructions described in the two sections, except as specified below for ***mfsr*** and ***mfsrin***.

The requirement that the SLB contain at most one entry that translates a given effective address (see Section 5.7.6.1) applies to SLB entries created by ***mtsr*** and ***mtsrin***. This requirement is satisfied naturally if only ***mtsr*** and ***mtsrin*** are used to create SLB entries for a given ESID, because for these instructions the association between SLB entries and ESID values is fixed (SLB entry i is used for ESID i). However, care must be taken if ***slbmte*** is also used to create SLB entries for the ESID, because for ***slbmte*** the association between SLB entries and ESID values is specified by software.

Move To Segment Register X-form

mtsr SR,RS

31	RS	/	SR	///	210	/
0	6	11	12	16	21	31

The SLB entry specified by SR is loaded from register RS, as follows.

SLBE Bit(s)	Set to	SLB Field(s)
0:31	0x0000_0000	$ESID_{0:31}$
32:35	SR	$ESID_{32:35}$
36	0b1	V
37:38	0b00	B
39:61	0b000||0x0_0000	$VSID_{0:22}$
62:88	$(RS)_{37:63}$	$VSID_{23:49}$
89:91	$(RS)_{33:35}$	K_sK_pN
92	$(RS)_{36}$	L ($(RS)_{36}$ must be 0b0)
93	0b0	C
94	0b0	reserved
95:96	0b00	LP

MSR_{SF} must be 0 when this instruction is executed; otherwise the results are boundedly undefined.

This instruction is privileged.

Special Registers Altered:

None

Move To Segment Register Indirect X-form

mtsrin RS,RB

31	RS	///	RB	242	/
0	6	11	16	21	31

The SLB entry specified by $(RB)_{32:35}$ is loaded from register RS, as follows.

SLBE Bit(s)	Set to	SLB Field(s)
0:31	0x0000_0000	$ESID_{0:31}$
32:35	$(RB)_{32:35}$	$ESID_{32:35}$
36	0b1	V
37:38	0b00	B
39:61	0b000||0x0_0000	$VSID_{0:22}$
62:88	$(RS)_{37:63}$	$VSID_{23:49}$
89:91	$(RS)_{33:35}$	K_sK_pN
92	$(RS)_{36}$	L ($(RS)_{36}$ must be 0b0)
93	0b0	C
94	0b0	reserved
95:96	0b00	LP

MSR_{SF} must be 0 when this instruction is executed; otherwise the results are boundedly undefined.

This instruction is privileged.

Special Registers Altered:

None

Move From Segment Register X-form

mfsr RT,SR

31	RT	/	SR	///	595	/
0	6	11	12	16	21	31

The contents of the low-order 27 bits of the VSID field and the contents of the K_s, K_p, N, and L fields of the SLB entry specified by SR are placed into register RT as follows.

SLBE Bit(s)	Copied to	SLB Field(s)
62:88	$RT_{37:63}$	$VSID_{23:49}$
89:91	$RT_{33:35}$	K_sK_pN
92	RT_{36}	L ($SLBE_L$ must be 0b0)

RT_{32} is set to 0. The contents of $RT_{0:31}$ are undefined.

MSR_{SF} must be 0 when this instruction is executed; otherwise the results are boundedly undefined.

This instruction must be used only to read an SLB entry that was, or could have been, created by ***mtsr*** or ***mtsrin*** and has not subsequently been invalidated (i.e., an SLB entry in which ESID<16, V=1, VSID<2^{27}, L=0, and C=0). If the SLB entry is invalid (V=0), $RT_{33:63}$ are set to 0. Otherwise the contents of register RT are undefined.

This instruction is privileged.

Special Registers Altered:
None

Move From Segment Register Indirect X-form

mfsrin RT,RB

31	RT	///	RB	659	/
0	6	11	16	21	31

The contents of the low-order 27 bits of the VSID field and the contents of the K_s, K_p, N, and L fields of the SLB entry specified by $(RB)_{32:35}$ are placed into register RT as follows.

SLBE Bit(s)	Copied to	SLB Field(s)
62:88	$RT_{37:63}$	$VSID_{23:49}$
89:91	$RT_{33:35}$	K_sK_pN
92	RT_{36}	L ($SLBE_L$ must be 0b0)

RT_{32} is set to 0. The contents of $RT_{0:31}$ are undefined.

MSR_{SF} must be 0 when this instruction is executed; otherwise the results are boundedly undefined.

This instruction must be used only to read an SLB entry that was, or could have been, created by ***mtsr*** or ***mtsrin*** and has not subsequently been invalidated (i.e., an SLB entry in which ESID<16, V=1, VSID<2^{27}, L=0, and C=0). If the SLB entry is invalid (V=0), $RT_{33:63}$ are set to 0. Otherwise the contents of register RT are undefined.

This instruction is privileged.

Special Registers Altered:
None

5.9.3.3 TLB Management Instructions

TLB Invalidate Entry **_X-form_**

tlbie RB,L
[Category: Server]

31	///	L	///	RB	306	/
0	6	10	11	16	21	31

```
if L = 0
  then
    p = 12
    if (RB)56=0
      then pg_size ← 4 KB
      else pg_size ← 64 KB
    else
      pg_size ← page size specified in (RB)44:51
      p ← log_base_2(pg_size)
sg_size ←  segment size specified in (RB)54:55
for each processor in the partition
  for each TLB entry
    if (entry_VA14:77-p = (RB)0:63-p) &
       (entry_sg_size = sg_size) &
       (entry_pg_size = pg_size)
    then TLB entry ← invalid
```

The operation performed by this instruction is based upon the contents of RB and the L field. The contents of RB are shown below, where L is the L field in the instruction.

L=0:

VPN	0s	B	AP	0s
0	52	54	56	57 63

L=1:

VPN	LP	0s	B	0s
0	44	52	54	56 63

If the L field of the instruction contains 0, RB_{56} (AP - Admixed Page size field) must be set to 0 if the page size specified by the PTE that was used to create the TLB entry to be invalidated is 4 KB and must be set to 1 if the page size specified by the PTE that was used to create the TLB entry to be invalidated is 64 KB. The VPN field in register RB must contain bits 14:65 of the virtual address translated by the TLB entry to be invalidated.

If the L field in the instruction contains 1, the following rules apply, 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.

- The page size is specified in the LP field in register RB, where the relationship between $(RB)_{LP}$ and the page size is the same as the relationship between PTE_{LP} and the page size (see Figure 6). 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.
- $(RB)_{0:43+c}$ must contain bits 14:77-p of the virtual address translated by the TLB to be invalidated, followed by p+c-20 zeros which must be ignored by the processor.

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 PTE_B used to create the TLB entry to be invalidated.

$RB_{52:53}$, RB_{56} (when the L field of the instruction is 1), and $RB_{57:63}$ must be set to zeros and must be ignored by the processor.

All TLB entries that have all of the following properties are made invalid on all processors that are in the same partition as the processor executing the ***tlbie*** instruction.

- The entry translates a virtual address for which $VA_{14:77-p}$ is equal to $(RB)_{0:63-p}$.
- The segment size of the entry is the same as the segment size specified in $(RB)_{54:55}$.
- Either of the following is true:
  - The L field in the instruction is 0, and either the page size of the entry is 4KB and $(RB)_{56}$=0, or the page size of the entry is 64KB and $(RB)_{56}$ =1.
  - The L field of the instruction is 1, and the page size of the entry matches the page size specified in $(RB)_{44:51}$.

Additional TLB entries may also be made invalid on any processor that is in the same partition as the processor executing the ***tlbie*** instruction.

MSR_{SF} must be 1 when this instruction is executed; otherwise the results are undefined.

The operation performed by this instruction is ordered by the ***eieio*** (or ***sync*** or ***ptesync***) instruction with respect to a subsequent ***tlbsync*** instruction executed by the processor 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 three sets that ***eieio*** orders.

This instruction is privileged, and can be executed only in hypervisor state. If it is executed in privileged but non-hypervisor state either a privileged Instruction type Program interrupt occurs or the results are boundedly undefined.

See Section 5.10, “Page Table Update Synchronization Requirements” for a description of other requirements associated with the use of this instruction.

Special Registers Altered:

None

Programming Note

For ***tlbie***[***l***] instructions in which L=0, the AP value in RB is provided to make it easier for the processor to locate address translations, in lookaside buffers, corresponding to the address translation being invalidated.

TLB Invalidate Entry Local — *X-form*

tlbiel RB,L
[Category: Server]

31	///	L	///	RB	274	/
0	6	10	11	16	21	31

```
if L = 0
   then
   p = 12
   if (RB)56=0
      then pg_size ← 4 KB
      else pg_size ← 64 KB
   else
      pg_size ← page size specified in (RB)44:51
      p ← log_base_2(pg_size)
sg_size ←  segment size specified in (RB)54:55
for each TLB entry
  if (entry_VA14:77-p = (RB)0:63-p) &
     (entry_sg_size = segment_size)
     (entry_pg_size = pg_size)
  then TLB entry ← invalid
```

The operation performed by this instruction is based upon the contents of RB and the L field. The contents of RB are shown below, where L is the L field in the instruction.

L=0:

VPN	0s	B	AP	0s
0	52	54	56	57 63

L=1:

VPN	LP	0s	B	0s
0	44	52	54	56 63

If the L field of the instruction contains 0, RB_{56} (AP - Admixed Page size field) must be set to 0 if the page size specified by the PTE that was used to create the TLB entry to be invalidated is 4 KB and must be set to 1 if the page size specified by the PTE that was used to create the TLB entry to be invalidated is 64 KB. The VPN field in register RB must contain bits 14:65 of the virtual address translated by the TLB entry to be invalidated.

If the L field in the instruction contains 1, the following rules apply, 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.

- The page size is specified in the LP field in register RB, where the relationship between $(RB)_{LP}$ and the page size is the same as the relationship between PTE_{LP} and the page size (see Figure 6). 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.
- $(RB)_{0:43+c}$ must contain bits 14:77-p of the virtual address translated by the TLB to be invalidated, followed by p+c-20 zeros which must be ignored by the processor.

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 PTE_B used to create the TLB entry to be invalidated.

$RB_{52:53}$, RB_{56} (when the L field of the instruction is 1), and RB $_{57:63}$ must be set to 0s and must be ignored by the processor.

All TLB entries that have all of the following properties are made invalid on the processor executing the ***tlbiel*** instruction.

- The entry translates a virtual address for which $VA_{14:77-p}$ is equal to $(RB)_{0:63-p}$.
- The segment size of the entry is the same as the segment size specified in $(RB)_{54:55}$.
- Either of the following is true:
  - The L field in the instruction is 0, and either the page size of the entry is 4KB and $(RB)_{56}$=0, or the page size of the entry is 64KB and $(RB)_{56}$ =1.
  - The L field of the instruction is 1, and the page size of the entry matches the page size specified in $(RB)_{44:51}$.

Only TLB entries on the processor executing the ***tlbiel*** instruction are affected.

MSR_{SF} must be 1 when this instruction is executed; otherwise the results are undefined.

This instruction is privileged, and can be executed only in hypervisor state. If it is executed in privileged but non-hypervisor state either a Privileged Instruction type Program interrupt occurs or the results are boundedly undefined.

See Section 5.10, "Page Table Update Synchronization Requirements" on page 454 for a description of other requirements associated with the use of this instruction.

Special Registers Altered:
None

Programming Note

The primary use of this instruction by hypervisor state code is to invalidate TLB entries prior to reassigning a processor to a new logical partition.

tlbiel may be executed on a given processor even if the sequence ***tlbie*** - ***eieio*** - ***tlbsync*** - ***ptesync*** is concurrently being executed on another processor.

See also the Programming Note with the description of the ***tlbie*** instruction.

TLB Invalidate All — *X-form*

tlbia

31	///	///	///	370	/
0	6	11	16	21	31

```
all TLB entries ← invalid
```

All TLB entries are made invalid on the processor executing the ***tlbia*** instruction.

This instruction is privileged, and can be executed only in hypervisor state. If it is executed in privileged but non-hypervisor state either a Privileged instruction type Program interrupt occurs or the results are boundedly undefined.

This instruction is optional, and need not be implemented.

Special Registers Altered:
None

Programming Note

tlbia does not affect TLBs on other processors.

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 processor executing the ***tlbsync*** instruction, with respect to the memory barrier created by a subsequent ***ptesync*** instruction executed by the same processor. 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 processor before any data accesses caused by instructions following the ***ptesync*** instruction are performed with respect to that processor.
- All storage accesses by other processors 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 processors using the translations being invalidated, will have been performed with respect to the processor executing the ***ptesync*** instruction, to the extent required by the associated Memory Coherence Required attributes, before the ***ptesync*** instruction's memory barrier is created.

The operation performed by this instruction is ordered by the ***eieio*** (or ***sync*** or ***ptesync***) instruction with respect to preceding ***tlbie*** instructions executed by the processor 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 and can be executed only in hypervisor state. If it is executed in privileged but non-hypervisor state either a Privileged Instruction type Program interrupt occurs or the results are boundedly undefined.

See Section 5.10 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***.

5.10 Page Table Update Synchronization Requirements

This section describes rules that software must follow when updating the Page Table, and includes suggested sequences of operations for some representative cases.

In the sequences of operations shown in the following subsections, any alteration of a Page Table Entry (PTE) 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).

Stores are not performed out-of-order, as described in Section 5.5, “Performing Operations Out-of-Order” on page 420. Moreover, address translations associated with instructions preceding the corresponding Store instructions are not performed again after the stores have been performed. (These address translations must have been performed before the store was determined to be required by the sequential execution model, because they might have caused an exception.) As a result, an update to a PTE need not be preceded by a context synchronizing operation.

All of the sequences require a context synchronizing operation after the sequence if the new contents of the PTE are to be used for address translations associated with subsequent instructions.

As noted in the description of the *Synchronize* instruction in Section 3.3.3 of Book II, address translation associated with instructions which occur in program order subsequent to the *Synchronize* (and this includes the ***ptesync*** variant) may actually 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*** or ***hrfid*** instruction that returns from the interrupt handler may provide the required context synchronization.

Page Table Entries must not be changed in a manner that causes an implicit branch.

5.10.1 Page Table Updates

TLBs are non-coherent caches of the HTAB. TLB entries must be invalidated explicitly with one of the *TLB Invalidate* instructions.

Unsynchronized lookups in the HTAB continue even while it is being modified. Any processor, including a processor on which software is modifying the HTAB, may look in the HTAB at any time in an attempt to translate a virtual address. When modifying a PTE, software must ensure that the PTE’s Valid bit is 0 if the PTE is inconsistent (e.g., if the RPN field is not correct for the current AVPN field).

Updates of Reference and Change bits by the processor 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 processor update of these bits and to avoid having the value written by a *Store* instruction overwritten by a processor update.

Before permitting one or more ***tlbie*** instructions to be executed on a given processor in a given partition software must ensure that no other processor will execute a “conflicting instruction” until after the following sequence of instructions has been executed on the given processor.

the ***tlbie*** instruction(s)
eieio
tlbsync
ptesync

The “conflicting instructions” in this case are the following.

- a ***tlbie*** or ***tlbsync*** instruction, if executed on another processor in the given partition
- an ***mtspr*** instruction that modifies the LPIDR, if the modification has either of the following properties.
  - The old LPID value (i.e., the contents of the LPIDR just before the ***mtspr*** instruction is executed) is the value that identifies the given partition
  - The new LPID value (i.e., the value specified by the ***mtspr*** instruction) is the value that identifies the given partition

Other instructions (excluding ***mtspr*** instructions that modify the LPIDR as described above, and excluding ***tlbie*** instructions except as shown) may be interleaved with the instruction sequence shown above, but the instructions in the sequence must appear in the order shown. On uniprocessor systems, the ***eieio*** and ***tlbsync*** instructions can be omitted. Other instructions may be interleaved with this sequence of instructions, but these instructions must appear in the order shown.

Programming Note

The ***eieio*** instruction prevents the reordering of ***tlbie*** instructions previously executed by the processor with respect to the subsequent ***tlbsync*** instruction. The ***tlbsync*** instruction and the subsequent ***ptesync*** instruction together ensure that all storage accesses 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 using the translations being invalidated, will be performed with respect to any processor 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 processor or mechanism.

The requirements specified above for ***tlbie*** instructions apply also to ***tlbsync*** instructions, except that the “sequence of instructions” consists solely of the ***tlbsync*** instruction(s) followed by a ***ptesync*** instruction.

Before permitting an ***mtspr*** instruction that modifies the LPIDR to be executed on a given processor, software must ensure that no other processor will execute a “conflicting instruction” until after the ***mtspr*** instruction followed by a context synchronizing instruction have been executed on the given processor (a context synchronizing event can be used instead of the context synchronizing instruction; see Chapter 10).

The “conflicting instructions” in this case are the following.

- a ***tlbie*** or ***tlbsync*** instruction, if executed on a processor in either of the following partitions
  - the partition identified by the old LPID value
  - the partition identified by the new LPID value

Programming Note

The restrictions specified above regarding modifying the LPIDR apply even on uniprocessor systems, and even if the new LPID value is equal to the old LPID value.

Similarly, when a ***tlbsync*** instruction has been executed by a processor in a given partition, a ***ptesync*** instruction must be executed by that processor before a ***tlbie*** or ***tlbsync*** instruction is executed by another processor in that partition.

The sequences of operations shown in the following subsections assume a multiprocessor environment. In a uniprocessor environment the ***tlbsync*** must be omitted, and the ***eieio*** that separates the ***tlbie*** from the ***tlbsync*** can be omitted. In a multiprocessor environment, when ***tlbiel*** is used instead of ***tlbie*** in a Page Table update, the synchronization requirements are the same as when ***tlbie*** is used in a uniprocessor environment.

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 processor, 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 processor returns the chosen value from location X, the sequence’s stores to the Page Table have been performed with respect to that other processor. 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.)

This Note assumes that the Page Table and location X are in storage that is Memory Coherence Required.

5.10.1.1 Adding a Page Table Entry

This is the simplest Page Table case. The Valid 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

```
PTE_ARPN,LP,AC,R,C,WIMG,N,PP ← new values
eieio     /* order 1st update before 2nd */
PTE_B,AVPN,SW,L,H,V ← new values (V=1)
ptesync   /* order updates before next
             Page Table search and before
             next data access.           */
```

5.10.1.2 Modifying a Page Table Entry

General Case

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 PTE, maintain a consistent state, ensure that the translation instantiated by the old entry is no longer available, and ensure that a subsequent reference to the virtual address translated by the new entry will use the correct real address and associated attributes. (The sequence is equivalent to deleting the PTE and then adding a new one; see Sections 5.10.1.1 and 5.10.1.3.)

```
PTE_V ← 0  /* (other fields don't matter)*/
ptesync    /* order update before tlbie and
              before next Page Table search */
tlbie(old_B,old_VA_14:77-p,old_L,old_LP,old_AP)
           /*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,AVPN,SW,L,H,V ← new values (V=1)
ptesync    /* order 2nd and 3rd updates before
              next Page Table search and
              before next data access      */
```

Resetting the Reference Bit

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.

```
oldR ← PTE_R /* get old R                    */
if oldR = 1 then
  PTE_R ← 0       /* store byte (R=0, other bits
                     unchanged)               */
  tlbie(B,VA_14:77-p,L,LP,AP) /* 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      */
```

Modifying the SW field

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 processor and doubleword 0 of the PTE is not modified by the processor.

```
loop: ldarx r1 ← PTE_dwd_0 /* load dwd 0 of PTE */
      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 ***lwarx***/***stwcx.*** pair (specifying the low-order word of doubleword 0 of the PTE) can be used instead of the ***ldarx*** /***stdcx.*** pair shown above.

Modifying the Virtual Address

If the virtual address translated by a valid PTE is to be modified and the new virtual address hashes to the same two PTEGs as does the old virtual address, the following sequence can be used to modify the PTE, maintain a consistent state, ensure that the translation instantiated by the old entry is no longer available, and ensure that a subsequent reference to the virtual address translated by the new entry will use the correct real address and associated attributes.

```
PTE_AVPN,SW,L,H,V ← new values (V=1)
ptesync   /* order update before tlbie and
             before next Page Table search      */
tlbie(old_B,old_VA_14:77-p,old_L,old_LP,old_AP)
          /*invalidate old translation*/
eieio     /* order tlbie before tlbsync         */
tlbsync   /* order tlbie before ptesync         */
ptesync   /* order tlbie, tlbsync, and update
             before next data access            */
```

5.10.1.3 Deleting a Page Table Entry

The following sequence can be used to ensure that the translation instantiated by an existing entry is no longer available.

```
PTE_V ← 0  /* (other fields don’t matter)         */
ptesync    /* order update before tlbie and
              before next Page Table search  */
tlbie(old_B,old_VA_14:77-p,old_L,old_LP,old_AP)
           /*invalidate old translation*/
eieio      /* order tlbie before tlbsync          */
tlbsync    /* order tlbie before ptesync          */
ptesync    /* order tlbie, tlbsync, and update
              before next data access             */
```

Chapter 6. Interrupts

6.1 Overview

The Power ISA provides an interrupt mechanism to allow the processor to change state as a result of external signals, errors, or unusual conditions arising in the execution of instructions.

System Reset and Machine Check interrupts are not ordered. All other interrupts are ordered such that only one interrupt is reported, and when it is processed (taken) no program state is lost. Since Save/Restore Registers SRR0 and SRR1 are serially reusable resources used by most interrupts, program state may be lost when an unordered interrupt is taken.

6.2 Interrupt Registers

6.2.1 Machine Status Save/ Restore Registers

When various interrupts occur, the state of the machine is saved in the Machine Status Save/Restore registers

(SRR0 and SRR1). Section 6.5 describes which registers are altered by each interrupt.

SRR0	//
0	62 63

SRR1	
0	63

Figure 31. 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 or, for SRR1 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 SRR1 (including implementation-dependent setting, e.g. by the Machine Check interrupt or by implementation-specific interrupts).

6.2.2 Hypervisor Machine Status Save/Restore Registers

When various interrupts occur, the state of the machine is saved in the Hypervisor Machine Status Save/Restore registers (HSRR0 and HSRR1). Section 6.5 describes which registers are altered by each interrupt.

HSRR0	//
0	62 63

HSRR1	
0	63

Figure 32. 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 or, 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, may have the side effect of modifying HSRR0 and HSRR1; see Section 6.4.4.

6.2.3 Data Address Register

The Data Address Register (DAR) is a 64-bit register that is set by the Machine Check, Data Storage, Data Segment, and Alignment interrupts; see Sections 6.5.2, 6.5.3, 6.5.4, and 6.5.8. In general, when one of these interrupts occurs the DAR is set to an effective address associated with the storage access that caused the interrupt, with the high-order 32 bits of the DAR set to 0 if the interrupt occurs in 32-bit mode.

DAR	
0	63

Figure 33. Data Address Register

6.2.4 Hypervisor Data Address Register

The Hypervisor Data Address Register (HDAR) is a 64-bit register that is set by the Hypervisor Data Storage and Hypervisor Data Segment interrupts; see Section 6.5.15 and Section 6.5.17. In general, when one of these interrupts 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.

HDAR	
0	63

Figure 34. Hypervisor Data Address Register

6.2.5 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, Data Segment, and Alignment interrupts; see Sections 6.5.2, 6.5.3, 6.5.4, and 6.5.8. In general, when one of these interrupts occurs the DSISR is set to indicate the cause of the interrupt.

DSISR	
32	63

Figure 35. 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 (including implementation-dependent setting, e.g. by the Machine Check interrupt or by implementation-specific interrupts).

6.2.6 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.

HDSISR	
32	63

Figure 36. Hypervisor Data Storage Interrupt Status Register

6.3 Interrupt Synchronization

When an interrupt occurs, SRR0 or HSRR0 is set to point to an instruction such that all preceding instructions have completed execution, no subsequent instruction has begun execution, and the instruction addressed by SRR0 or HSRR0 may or may not have completed execution, depending on the interrupt type.

With the exception of System Reset and Machine Check interrupts, all interrupts are context synchronizing as defined in Section 1.5.1. System Reset and Machine Check interrupts are context synchronizing if they are recoverable (i.e., if bit 62 of SRR1 is set to 1 by the interrupt). If a System Reset or Machine Check interrupt is not recoverable (i.e., if bit 62 of SRR1 is set to 0 by the interrupt), it acts like a context synchronizing operation with respect to subsequent instructions. That is, a non-recoverable System Reset or Machine Check interrupt need not satisfy items 1 through 3 of Section 1.5.1, but does satisfy items 4 and 5.

6.4 Interrupt Classes

Interrupts are classified by whether they are directly caused by the execution of an instruction or are caused by some other system exception. Those that are "system-caused" are:

- System Reset
- Machine Check
- External
- Decrementer
- Hypervisor Decrementer

External, Decrementer, and Hypervisor Decrementer interrupts are maskable interrupts. Therefore, software may delay the generation of these interrupts. System Reset and Machine Check interrupts are not maskable.

"Instruction-caused" interrupts are further divided into two classes, *precise* and *imprecise*.

6.4.1 Precise Interrupt

Except for the Imprecise Mode Floating-Point Enabled Exception type Program interrupt, all instruction-caused interrupts are precise.

When the fetching or execution of an instruction causes a precise interrupt, the following conditions exist at the interrupt point.

1. SRR0 addresses either the instruction causing the exception or the immediately following instruction. Which instruction is addressed can be determined from the interrupt type and status bits.
2. An interrupt is generated such that all instructions preceding the instruction causing the exception appear to have completed with respect to the executing processor.
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.

6.4.2 Imprecise Interrupt

This architecture defines one imprecise interrupt, the Imprecise Mode Floating-Point Enabled Exception type Program interrupt.

When an Imprecise Mode Floating-Point Enabled Exception type Program interrupt occurs, the following conditions exist at the interrupt point.

1. SRR0 addresses either the instruction causing the exception or some instruction following that instruction; see Section 6.5.9, "Program Interrupt" on page 471.
2. An interrupt is generated such that all instructions preceding the instruction addressed by SRR0 appear to have completed with respect to the executing processor.
3. The instruction addressed by SRR0 may appear not to have begun execution (except, in some cases, for causing the interrupt to occur), may have been partially executed, or may have completed; see Section 6.5.9.
4. No instruction following the instruction addressed by SRR0 appears to have begun execution.

All Floating-Point Enabled Exception type Program interrupts are maskable using the MSR bits FE0 and FE1. Although these interrupts are maskable, they differ significantly from the other maskable interrupts in that the masking of these interrupts is usually controlled by the application program, whereas the masking of all other maskable interrupts is controlled by either the operating system or the hypervisor.

6.4.3 Interrupt Processing

Associated with each kind of interrupt is an *interrupt vector*, which contains the initial sequence of instructions that is executed when the corresponding interrupt occurs.

Interrupt processing consists of saving a small part of the processor's state in certain registers, identifying the cause of the interrupt in other registers, and continuing execution at the corresponding interrupt vector location. When an exception exists that will cause an interrupt to be generated and it has been determined that the interrupt will occur, the following actions are performed. The handling of Machine Check interrupts (see Section 6.5.2) differs from the description given below in several respects.

1. SRR0 or HSRR0 is loaded with an instruction address that depends on the type of interrupt; see the specific interrupt description for details.
2. Bits 33:36 and 42:47 of SRR1 or HSRR1 are loaded with information specific to the interrupt type.
3. Bits 0:32, 37:41, and 48:63 of SRR1 or HSRR1 are loaded with a copy of the corresponding bits of the MSR.
4. The MSR is set as shown in Figure 37 on page 466. In particular, MSR bits IR and DR are set to 0, disabling relocation, 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 38 on page 466.

Interrupts do not clear reservations obtained with ***lwarx*** or ***ldarx***.

Programming Note

In general, when an interrupt occurs, the following instructions should be executed by the operating system before dispatching a "new" program.

- ***stwcx.*** or ***stdcx.***, to clear the reservation if one is outstanding, to ensure that a ***lwarx*** or ***ldarx*** in the interrupted program is not paired with a ***stwcx.*** or ***stdcx.*** in the "new" program.
- ***sync***, to ensure that all storage accesses caused by the interrupted program will be performed with respect to another processor before the program is resumed on that other processor.
- ***isync*** or ***rfid***, to ensure that the instructions in the "new" program execute in the "new" context.

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, processor 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: Illegal Instruction type Program interrupt caused by an instruction that has been phased out of the architecture but is still used by some programs that the operating system supports, or by an instruction that is in a category that the implementation does not support but is used by some programs that the operating system supports.

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 processor 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 an Illegal Instruction type Program 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 processor generated the Program interrupt as the result of executing a valid instruction.

Programming Note

In order to handle Machine Check and System Reset interrupts correctly, the operating system should manage MSR_{RI} as follows.

- In the Machine Check and System Reset interrupt handlers, interpret SRR1 bit 62 (where MSR_{RI} is placed) as:
  - 0: interrupt is not recoverable
  - 1: interrupt is recoverable
- In each interrupt handler, when enough state has been saved that a Machine Check or System Reset interrupt can be recovered from, set MSR_{RI} to 1.
- In each 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***.

For interrupts that set the SRRs other than Machine Check or System Reset, MSR_{RI} can be managed similarly when these interrupts occur within interrupt handlers for other interrupts that set the SRRs.

This Note does not apply to interrupts that set the HSRRs because these interrupts put the processor into hypervisor state, and either do not occur or can be prevented from occurring within interrupt handlers for other interrupts that set the HSRRs.

6.4.4 Implicit alteration of HSRR0 and HSRR1

Executing some of the more complex instructions may have the side effect of altering the contents of HSRR0 and HSRR1. The instructions listed below are guaranteed not to have this side effect. Any omission of instruction suffixes is significant; e.g., ***add*** is listed but ***add.*** is excluded.

1. *Branch* instructions

   b[***l***][***a***], ***bc***[***l***][***a***], ***bclr***[***l***], ***bcctr***[***l***]

2. *Fixed-Point Load* and *Store* Instructions

   lbz, ***lbzx***, ***lhz***, ***lhzx***, ***lwz***, ***lwzx***, ***ld***<64>, ***ldx***<64>, ***stb***, ***stbx***, ***sth***, ***sthx***, ***stw***, ***stwx***, ***std***<64>, ***stdx***<64>

   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_{DR}=0.

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<64>, ***rldicr***<64>, ***rldic***<64>, ***rlwinm***, ***rldcl***<64>, ***rldcr***<64>, ***rlwnm***, ***rldimi***<64>, ***rlwimi***, ***sld***<64>, ***slw***, ***srd***<64>, ***srw***

7. Other instructions

   isync

   rfid, ***hrfid***

   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 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_{IR}=0.

6.5 Interrupt Definitions

Figure 37 shows all the types of interrupts and the values assigned to the MSR for each. Figure 38 shows the effective address of the interrupt vector for each interrupt type. (Section 5.7.4 on page 426 summarizes all architecturally defined uses of effective addresses, including those implied by Figure 38.)

Interrupt Type	MSR Bit IR	DR	FE0	FE1	EE	RI	ME	HV
System Reset	0	0	0	0	0	0	-	1
Machine Check	0	0	0	0	0	0	0	1
Data Storage	0	0	0	0	0	0	-	m
Data Segment	0	0	0	0	0	0	-	m
Instruction Storage	0	0	0	0	0	0	-	m
Instruction Segment	0	0	0	0	0	0	-	m
External	0	0	0	0	0	0	-	e
Alignment	0	0	0	0	0	0	-	m
Program	0	0	0	0	0	0	-	m
FP Unavailable	0	0	0	0	0	0	-	m
Decrementer	0	0	0	0	0	0	-	m
Hypervisor Decrem'er	0	0	0	0	0	-	-	1
System Call	0	0	0	0	0	0	-	s
Trace	0	0	0	0	0	0	-	m
Hypervisor Data Stg.	0	0	0	0	0	-	-	1
Hypervisor Instr. Stg.	0	0	0	0	0	-	-	1
Hypervisor Instr. Seg.	0	0	0	0	0	-	-	1
Hypervisor Data Seg.	0	0	0	0	0	-	-	1
Performance Monitor	0	0	0	0	0	0	-	m
Vector Unavailable[1]	0	0	0	0	0	0	-	m

0 bit is set to 0
1 bit is set to 1
- bit is not altered
m if $LPES_1$=0, set to 1; otherwise not altered
e if $LPES_0$=0, set to 1; otherwise not altered
s if LEV=1 or LPES/$LPES_1$=0, set to 1; otherwise not altered

Settings for Other Bits

Bits BE, FP, PMM, PR, SE, and VEC[1] are set to 0.

If the interrupt results in HV being equal to 1, the LE bit is copied from the HILE bit; otherwise the LE bit is copied from the $LPCR_{ILE}$ bit.

The SF bit is set to 1.

Reserved bits are set as if written as 0.

[1] Category: Vector

Figure 37. MSR setting due to interrupt

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	Reserved
00..0000_0B00	Reserved
00..0000_0C00	System Call
00..0000_0D00	Trace
00..0000_0E00	Hypervisor Data Storage
00..0000_0E10	Hypervisor Instruction Storage
00..0000_0E20	Hypervisor Data Segment
00..0000_0E30	Hypervisor Instruction Segment
00..0000_0E40	Reserved
. . .	. . .
00..0000_0EFF	Reserved
00..0000_0F00	Performance Monitor
00..0000_0F10	Reserved
00..0000_0F20	Vector Unavailable[3]
00..0000_0F30	Reserved
. . .	. . .
00..0000_0FFF	Reserved

[1] The values in the Effective Address column are interpreted as follows.
- 00...0000_nnnn *means* 0x0000_0000_0000_nnnn

[2] Effective addresses 0x0000_0000_0000_0000 through 0x0000_0000_0000_00FF are used by software and will not be assigned as interrupt vectors.

[3] Category: Vector.

Figure 38. 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 38 risks incompatibility with future implementations.

6.5.1 System Reset Interrupt

If a System Reset exception causes an interrupt that is not context synchronizing or causes the loss of a

Machine Check exception or an External exception, or if the state of the processor has been corrupted, the interrupt is not recoverable.

The following registers are set:

SRR0 Set to the effective address of the instruction that the processor would have attempted to execute next if no interrupt conditions were present.

SRR1
- **33:36** Set to 0.
- **42:44** Set to an implementation-dependent value.
- **45:47** Set to 0.
- **62** Loaded from bit 62 of the MSR if the processor is in a recoverable state; otherwise set to 0.
- **Others** Loaded from the MSR.

MSR See Figure 37 on page 466.

Execution resumes at effective address 0x0000_0000_0000_0100.

Each implementation provides an implementation-dependent means for software to distinguish power-on Reset from other types of System Reset.

6.5.2 Machine Check Interrupt

The causes of Machine Check interrupts are implementation-dependent. For example, a Machine Check interrupt may be caused by a reference to a storage location that contains an uncorrectable error or does not exist (see Section 5.6), or by an error in the storage subsystem.

Machine Check interrupts are enabled when MSR_{ME}=1. If MSR_{ME}=0 and a Machine Check occurs, the processor enters the Checkstop state. The Checkstop state may also be entered if an access is attempted to a storage location that does not exist (see Section 5.6).

Disabled Machine Check (Checkstop State)

When a processor is in Checkstop state, instruction processing is suspended and generally cannot be restarted without resetting the processor. Some implementations may preserve some or all of the internal state of the processor 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 an External exception, or if the state of the processor has been corrupted, the interrupt is not recoverable.

In some systems, the operating system may attempt to identify and log the cause of the Machine Check.

The following registers are set:

SRR0 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. The details are implementation-dependent.

SRR1
- **62** Loaded from bit 62 of the MSR if the processor is in a recoverable state; otherwise set to 0.
- **Others** Set to an implementation-dependent value.

MSR See Figure 37.

DSISR Set to an implementation-dependent value.

DAR 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.

6.5.3 Data Storage Interrupt

A Data Storage interrupt occurs when no higher priority exception exists, the value of the expression

$(MSR_{HV\ PR} = 0b10) \mid (\neg VPM_0 \ \&\ \neg MSR_{DR})$

$\mid (\neg VPM_1 \ \&\ MSR_{DR})$

is 1, and a data access cannot be performed for any of the following reasons.

- Data address translation is enabled (MSR_{DR}=1) and the virtual address of any byte of the storage location specified by a *Load*, *Store*, ***icbi***, ***dcbz***, ***dcbst***, ***dcbf***[***l***], ***eciwx***, or ***ecowx*** instruction cannot be translated to a real address.
- The effective address specified by a ***lq***, ***stq***, ***lwarx***, ***ldarx***, ***stwcx.***, or ***stdcx.*** instruction refers to storage that is Write Through Required or Caching Inhibited.
- The access violates storage protection.
- A Data Address Breakpoint match occurs.
- Execution of an ***eciwx*** or ***ecowx*** instruction is disallowed because EAR_E=0.

If a ***stwcx.*** or ***stdcx.*** would not perform its store in the absence of a Data Storage interrupt, and either (a) the specified effective address refers to storage that is Write Through Required or Caching Inhibited, or (b) a non-conditional *Store* to the specified effective address would cause a Data Storage interrupt, it is implementation-dependent whether a Data Storage interrupt occurs.

If the contents of the XER specifies a length of zero bytes for a *Move Assist* instruction, a Data Storage interrupt does not occur for reasons of address translation, or storage protection. If such an instruction causes a Data Storage interrupt for other reasons, the setting of the DSISR and DAR reflects only these other reasons listed in the preceding sentence. (E.g., if such an instruction causes a storage protection violation and a Data Address Breakpoint match, the DSISR and DAR are set as if the storage protection violation did not occur.)

The following registers are set:

SRR0 Set to the effective address of the instruction that caused the interrupt.

SRR1
- **33:36** Set to 0.
- **42:47** Set to 0.
- **Others** Loaded from the MSR.

MSR See Figure 37.

DSISR
- **32** Set to 0.
- **33** Set to 1 if MSR_{DR}=1 and the translation for an attempted access is not found in the primary PTEG or in the secondary PTEG; otherwise set to 0.
- **34:35** Set to 0.
- **36** Set to 1 if the access is not permitted by Figure 24 or 25, as appropriate; otherwise set to 0.
- **37** Set to 1 if the access is due to a ***lq***, ***stq***, ***lwarx***, ***ldarx***, ***stwcx.***, or ***stdcx.*** instruction that addresses storage that is Write Through Required or Caching Inhibited; otherwise set to 0.
- **38** Set to 1 for a *Store*, ***dcbz***, or ***ecowx*** instruction; otherwise set to 0.
- **39:40** Set to 0.
- **41** Set to 1 if a Data Address Breakpoint 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 execution of an ***eciwx*** or ***ecowx*** instruction is attempted when EAR_E=0; otherwise set to 0.
- **44:63** Set to 0.

DAR Set to the effective address of a storage element as described in the following list. The list should be read from the top down; the DAR is set as described by the first item that corresponds to an exception that is reported in the DSISR. For example, if a *Load* instruction causes a storage protection violation and a Data Address Breakpoint 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.

- a Data Storage exception occurs for reasons other than a Data Address Breakpoint match or, for ***eciwx*** and ***ecowx***, EAR_E=0
  - 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*, *Store*, ***eciwx***, or ***ecowx*** instruction ("first" refers to address order; see Section 6.7)
- undefined, for a Data Address Breakpoint match, or if ***eciwx*** or ***ecowx*** is executed when EAR_E=0

For the cases in which the DAR is specified above to be set to a defined value, if the interrupt occurs in 32-bit mode the high-order 32 bits of the DAR are set to 0.

If multiple Data Storage exceptions occur for a given effective address, any one or more of the bits corresponding to these exceptions may be set to 1 in the DSISR.

Execution resumes at effective address 0x0000_0000_0000_0300.

6.5.4 Data Segment Interrupt

A Data Segment interrupt occurs when no higher priority exception exists and a data access cannot be performed because data address translation is enabled and the effective address of any byte of the storage location specified by a *Load*, *Store*, ***icbi***, ***dcbz***, ***dcbst***, ***dcbf***[***l***] ***eciwx***, or ***ecowx*** instruction cannot be translated to a virtual address.

If a ***stwcx.*** or ***stdcx.*** 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 a *Move Assist* instruction has a length of zero (in the XER), a Data Segment interrupt does not occur, regardless of the effective address.

The following registers are set:

SRR0 Set to the effective address of the instruction that caused the interrupt.

SRR1
- **33:36** Set to 0.

42:47 Set to 0.
Others Loaded from the MSR.

MSR See Figure 37.

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 Data Segment interrupt, for a *Cache Management* instruction
- a byte in the first aligned doubleword for which access was attempted in the segment that caused the Data Segment interrupt, for a *Load*, *Store*, ***eciwx***, or ***ecowx*** instruction ("first" refers to address order; see Section 6.7)

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.

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).

6.5.5 Instruction Storage Interrupt

An Instruction Storage interrupt occurs when no higher priority exception exists, the value of the expression

$$(MSR_{HV\ PR} = 0b10) \mid (\neg VPM_0 \ \& \ \neg MSR_{IR})$$
$$\mid (\neg VPM_1 \ \& \ MSR_{IR})$$

is 1, and the next instruction to be executed cannot be fetched for any of the following reasons.

- Instruction address translation is enabled and the virtual address cannot be translated to a real address.
- The fetch access violates storage protection.

The following registers are set:

SRR0 Set to the effective address of the instruction that the processor 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 MSR_{IR}=1 and the translation for an attempted access is not found in the primary PTEG or in the secondary PTEG; otherwise set to 0.
34 Set to 0.
35 Set to 1 if the access is to No-execute or Guarded storage; otherwise set to 0.
36 Set to 1 if the access is not permitted by Figure 24, or 25, as appropriate; otherwise set to 0.

Programming Note

Storage protection violations for the Data Storage Interrupt are reported in $DSISR_{36}$ and $DSISR_{42}$, whereas storage protection violations for the Instruction Storage Interrupt are reported in $SRR1_{35}$ and $SRR1_{36}$.

42:47 Set to 0.

Others Loaded from the MSR.

MSR See Figure 37.

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.

6.5.6 Instruction Segment Interrupt

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.

The following registers are set:

SRR0 Set to the effective address of the instruction that the processor 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 37 on page 466.

Execution resumes at effective address 0x0000_0000_0000_0480.

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).

6.5.7 External Interrupt

An External interrupt occurs when no higher priority exception exists, an External exception exists, and MSR_{EE}=1. The occurrence of the interrupt does not cause the exception to cease to exist.

The following registers are set:

SRR0 Set to the effective address of the instruction that the processor 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 37.

Execution resumes at effective address 0x0000_0000_0000_0500.

6.5.8 Alignment Interrupt

An Alignment interrupt occurs when no higher priority exception exists and a data access cannot be performed for any of the following reasons.

- The operand of a floating-point *Load* or *Store* is not word-aligned, or crosses a virtual page boundary.
- The operand of ***lq***, ***stq***, ***lmw***, ***stmw***, ***lwarx***, ***ldarx***, ***stwcx.***, ***stdcx.***, ***eciwx***, or ***ecowx*** is not aligned.
- The operand of a single-register *Load* or *Store* is not aligned and the processor is in Little-Endian mode.
- The instruction is ***lq***, ***stq***, ***lmw***, ***stmw***, ***lswi***, ***lswx***, ***stswi***, or ***stswx***, and the operand is in storage that is Write Through Required or Caching Inhibited, or the processor is in Little-Endian mode.
- The operand of a *Load* or *Store* crosses a segment boundary, or crosses a boundary between virtual pages that have different storage control attributes.
- The operand of a *Load* or *Store* is not aligned and is in storage that is Write Through Required or Caching Inhibited.
- The operand of ***dcbz***, ***lwarx***, ***ldarx***, ***stwcx.***, or ***stdcx.*** is in storage that is Write Through Required or Caching Inhibited.

If a ***stwcx.*** or ***stdcx.*** would not perform its store in the absence of an Alignment interrupt and the specified effective address refers to storage that is Write Through Required or Caching Inhibited, it is implementation-dependent whether an Alignment interrupt occurs.

Setting the DSISR and DAR as described below is optional for implementations on which Alignment interrupts occur rarely, if ever, for cases that the Alignment interrupt handler emulates. For such implementations, if the DSISR and DAR are not set as described below they are set to undefined values.

The following registers are set:

SRR0 Set to the effective address of the instruction that caused the interrupt.

SRR1
- **33:36** Set to 0.
- **42:47** Set to 0.
- **Others** Loaded from the MSR.

MSR See Figure 37.

DSISR
- **32:43** Set to 0.
- **44:45** Set to bits 30:31 of the instruction if DS-form. Set to 0b00 if D-, or X-form.
- **46** Set to 0.
- **47:48** Set to bits 29:30 of the instruction if X-form. Set to 0b00 if D- or DS-form.
- **49** Set to bit 25 of the instruction if X-form. Set to bit 5 of the instruction if D- or DS-form.
- **50:53** Set to bits 21:24 of the instruction if X-form. Set to bits 1:4 of the instruction if D- or DS-form.
- **54:58** Set to bits 6:10 of the instruction (RT/RS/FRT/FRS), except undefined for ***dcbz***.
- **59:63** Set to bits 11:15 of the instruction (RA) for update form instructions; set to either bits 11:15 of the instruction or to any register number not in the range of registers to be loaded for a valid form ***lmw***, a valid form ***lswi***, or a valid form ***lswx*** for which neither RA nor RB is in the range of registers to be loaded; otherwise undefined.

DAR Set to the effective address computed by the instruction, except that if the interrupt occurs in 32-bit mode the high-order 32 bits of the DAR are set to 0.

For an X-form *Load* or *Store*, it is acceptable for the processor to set the DSISR to the same value that would have resulted if the corresponding D- or DS-form instruction had caused the interrupt. Similarly, for a D- or DS-form *Load* or *Store*, it is acceptable for the processor to set the DSISR to the value that would have resulted for the corresponding X-form instruction. For example, an unaligned ***lwax*** (that crosses a protection boundary) would normally, following the description above, cause the DSISR to be set to binary:

```
000000000000 00 0 01 0 0101 ttttt ?????
```

where “ttttt” denotes the RT field, and “?????” denotes an undefined 5-bit value. However, it is acceptable if it causes the DSISR to be set as for ***lwa***, which is

```
000000000000 10 0 00 0 1101 ttttt ?????
```

If there is no corresponding alternative form instruction (e.g., for ***lwaux***), the value described above is set in the DSISR.

The instruction pairs that may use the same DSISR value are.

lhz/lhzx	lhzu/lhzux	lha/lhax	lhau/lhaux
lwz/lwzx	lwzu/lwzux	lwa/lwax	
ld/ldx	ldu/ldux		
lsth/sthx	sthu/sthux	stw/stwx	stwu/stwux
std/stdx	stdu/stdux		
lfs/lfsx	lfsu/lfsux	lfd/lfdx	lfdu/lfdux
stfs/stfsx	stfsu/stfsux	stfd/stfdx	stfdu/stfdux

Execution resumes at effective address 0x0000_0000_0000_0600.

Programming Note

The architecture does not support the use of an unaligned effective address by ***lwarx***, ***ldarx***, ***stwcx.***, ***stdcx.***, ***eciwx***, and ***ecowx***. If an Alignment interrupt occurs because one of these instructions specifies an unaligned effective address, the Alignment interrupt handler must not attempt to simulate the instruction, but instead should treat the instruction as a programming error.

6.5.9 Program Interrupt

A Program interrupt occurs when no higher priority exception exists and one of the following exceptions arises during execution of an instruction:

Floating-Point Enabled Exception

A Floating-Point Enabled Exception type Program interrupt is generated when the value of the expression

$$(MSR_{FE0} \mid MSR_{FE1}) \ \& \ FPSCR_{FEX}$$

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.

Illegal Instruction

An Illegal Instruction type Program 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.

An Illegal Instruction type Program interrupt may be generated when execution is attempted of any of the following kinds of instruction.

- an instruction that is in invalid form
- an ***lswx*** instruction for which RA or RB is in the range of registers to be loaded
- an ***mtspr*** or ***mfspr*** instruction with an SPR field that does not contain one of the defined values

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 one of the defined values having spr_0=1. It may be generated when execution is attempted of an ***mtspr*** or ***mfspr*** instruction with an SPR field that does not contain one of the defined values but has spr_0=1.

The following applies if the instruction is executed when $MSR_{HV\ PR}$ = 0b00.

A Privileged Instruction type Program interrupt may be generated when execution is attempted of an ***mtspr*** instruction with an SPR field that designates a hypervisor resource, or when execution of a ***tlbie***, ***tlbiel***, ***tlbia***, or ***tlbsync*** instruction is attempted.

Programming Note

These are the only cases in which a Privileged Instruction type Program interrupt can be generated when MSR_{PR}=0. They can be distinguished from other causes of Privileged Instruction type Program interrupts by examining $SRR1_{49}$ (the bit in which MSR_{PR} was saved by the interrupt).

Trap

A Trap type Program interrupt is generated when any of the conditions specified in a *Trap* instruction is met.

The following registers are set:

SRR0 For all Program interrupts except a Floating-Point Enabled Exception type Program interrupt, set to the effective address of the instruction that caused the corresponding exception.

For a Floating-Point Enabled Exception type Program interrupt, set as described in the following list.

- If $MSR_{FE0\ FE1}$ = 0b00, $FPSCR_{FEX}$ = 1, and an instruction is executed that changes $MSR_{FE0\ FE1}$ to a nonzero value, set to the effective address of the instruction that the processor 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 $MSR_{FE0\ FE}$ = 0b11, set to the effective address of the instruction that caused the Floating-Point Enabled Exception.
- If $MSR_{FE0\ FE}$ = 0b01 or 0b10, set to the effective address of the first instruction that caused a Floating-Point Enabled Exception since the most recent time $FPSCR_{FEX}$ 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

33:36 Set to 0.
42 Set to 0.
43 Set to 1 for a Floating-Point Enabled Exception type Program interrupt; otherwise set to 0.
44 Set to 1 for an Illegal Instruction type Program interrupt; otherwise set to 0.
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.

Only one of bits 43:46 can be set to 1.

MSR See Figure 37 on page 466.

Execution resumes at effective address 0x0000_0000_0000_0700.

6.5.10 Floating-Point Unavailable Interrupt

A Floating-Point Unavailable interrupt occurs when no higher priority exception exists, an attempt is made to execute a floating-point instruction (including floating-point loads, stores, and moves), and MSR_{FP}=0.

The following registers are set:

SRR0 Set to the effective address of the instruction that caused the interrupt.

SRR1

33:36 Set to 0.
42:47 Set to 0.
Others Loaded from the MSR.

MSR See Figure 37 on page 466.

Execution resumes at effective address 0x0000_0000_0000_0800.

6.5.11 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 processor 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 37 on page 466.

Execution resumes at effective address 0x0000_0000_0000_0900.

6.5.12 Hypervisor Decrementer Interrupt

A Hypervisor Decrementer interrupt occurs when no higher priority exception exists, a Hypervisor Decrementer exception exists, and the value of the following expression is 1.

$(MSR_{EE} \mid \neg(MSR_{HV}) \mid MSR_{PR})$ & HDICE

The following registers are set:

HSRR0 Set to the effective address of the instruction that the processor 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 37 on page 466.

Execution resumes at effective address 0x0000_0000_0000_0980.

> **Programming Note**
>
> Because the value of MSR_{EE} is always 1 when the processor is in problem state, the simpler expression
>
> $(MSR_{EE} \mid \neg(MSR_{HV}))$ & HDICE
>
> is equivalent to the expression given above.

6.5.13 System Call Interrupt

A System Call interrupt occurs when a *System Call* instruction is executed.

The following registers are set:

SRR0 Set to the effective address of the instruction following the System Call instruction.

SRR1
- **33:36** Set to 0.
- **42:47** Set to 0.
- **Others** Loaded from the MSR.

MSR See Figure 37 on page 466.

Execution resumes at effective address 0x0000_0000_0000_0C00.

> **Programming Note**
>
> An attempt to execute an ***sc*** instruction with LEV=1 in problem state should be treated as a programming error.

6.5.14 Trace Interrupt [Category: Trace]

A Trace interrupt occurs when no higher priority exception exists and either MSR_{SE}=1 and any instruction except ***rfid*** or ***hrfid,*** is successfully completed, or MSR_{BE}=1 and a *Branch* instruction is completed. Successful completion means that the instruction caused no other interrupt. Thus a Trace interrupt never occurs for a *System Call* instruction, or for a *Trap* instruction that traps. The instruction that causes a Trace interrupt is called the “traced instruction”.

When a Trace interrupt occurs, the following registers are set:

SRR0 Set to the effective address of the instruction that the processor would have attempted to execute next if no interrupt conditions were present.

SRR1
- **33:36 and 42:47** Set to an implementation-dependent value.
- **Others** Loaded from the MSR.

MSR See Figure 37 on page 466.

Execution resumes at effective address 0x0000_0000_0000_0D00.

Extensions to the Trace facility are described in Appendix C.

> **Programming Note**
>
> The following instructions are not traced.
>
> - ***rfid***
> - ***hrfid***
> - ***sc***, and *Trap* instructions that trap
> - other instructions that cause interrupts (other than Trace interrupts)
> - the first instructions of any interrupt handler
> - instructions that are emulated by software
>
> In general, interrupt handlers can achieve the effect of tracing these instructions.

6.5.15 Hypervisor Data Storage Interrupt

A Hypervisor Data Storage interrupt occurs when the processor is not in hypervisor state, no higher priority exception exists, the value of the expression

$(VPM_0 \,\&\, \neg MSR_{DR}) \mid (VPM_1 \,\&\, MSR_{DR})$

is 1, and a data access cannot be performed for any of the following reasons.

- Data address translation is enabled (MSR_{DR}=1) and the virtual address of any byte of the storage location specified by a *Load*, *Store*, ***icbi***, ***dcbz***,

dcbst, ***dcbf***[***l***], ***eciwx***, or ***ecowx*** instruction cannot be translated to a real address.

- Data address translation is disabled (MSR_{DR}=0), $LPES_1$ =1, and the virtual address of any byte of the storage location specified by a *Load*, *Store*, ***icbi***, ***dcbz***, ***dcbst***, ***dcbf***[***l***], ***eciwx***, or ***ecowx*** instruction cannot be translated to a real address by means of the virtual real addressing mechanism.
- The effective address specified by a ***lwarx***, ***ldarx***, ***stwcx.***, or ***stdcx.*** instruction refers to storage that is Write Through Required or Caching Inhibited.
- The access violates storage protection.
- A Data Address Compare match or a Data Address Breakpoint match occurs.
- Execution of an ***eciwx*** or ***ecowx*** instruction is disallowed because EA_{RE}=0.

If a ***stwcx.*** or ***stdcx.*** 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 contents of the XER specifies a length of zero bytes for a *Move Assist* instruction, a Hypervisor Data Storage interrupt does not occur for reasons of address translation, or storage protection. If such an instruction causes a Hypervisor Data Storage interrupt for other reasons, the setting of the HDSISR and HDAR reflects only these other reasons listed in the preceding sentence. (E.g., if such an instruction causes a storage protection violation and a Data Address Breakpoint match, the HDSISR and HDAR are set as if the storage protection violation did not occur.)

The following registers are set:

HSRR0 Set to the effective address of the instruction that caused the interrupt.

HSRR1
- **33:36** Set to 0.
- **42:47** Set to 0.
- **Others** Loaded from the MSR.

MSR See Figure 37.

HDSISR
- **32** Set to 0.
- **33** Set to 1 if the value of the expression $(MSR_{DR}) \mid ((\neg MSR_{DR} \;\&\; VPM_0) \;\&\; LPES_1)$ is 1 and the translation for an attempted access is not found in the primary PTEG or in the secondary PTEG; otherwise set to 0.
- **34:35** Set to 0.
- **36** Set to 1 if the access is not permitted by the storage protection mechanism; otherwise set to 0.
- **37** Set to 1 if the access is due to a ***lq***, ***stq***, ***lwarx***, ***ldarx***, ***stwcx.***, or ***stdcx.*** instruction that addresses storage that is Write Through Required or Caching Inhibited; otherwise set to 0.
- **38** Set to 1 for a *Store*, ***dcbz***, or ***ecowx*** instruction; otherwise set to 0.
- **39:40** Set to 0.
- **41** Set to 1 if a Data Address Compare match or a Data Address Breakpoint match occurs; otherwise set to 0.
- **42** Set to 0.
- **43** Set to 1 if execution of an ***eciwx*** or ***ecowx*** instruction is attempted when EAR_E=0; otherwise set to 0.
- **44:63** Set to 0.

HDAR Set to the effective address of a storage element as described in the following list. The list should be read from the top down; the 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* instruction causes a storage protection violation and a Data Address Breakpoint 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.

- a Data Storage exception occurs for reasons other than a Data Address Breakpoint match or, for ***eciwx*** and ***ecowx***, EAR_E=0
  - 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*, *Store*, ***eciwx***, or ***ecowx*** instruction ("first" refers to address order; see Section 6.7)
- undefined, for a Data Address Breakpoint match, or if ***eciwx*** or ***ecowx*** is executed when EAR_E=0

For the cases in which the HDAR is specified above to be set to a defined value, if the interrupt occurs in 32-bit mode the high-order 32 bits of the DAR are set to 0.

If multiple 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.

Execution resumes at effective address 0x0000_0000_0000_0E00.

6.5.16 Hypervisor Instruction Storage Interrupt

A Hypervisor Instruction Storage interrupt occurs when the processor is not in hypervisor state, no higher priority exception exists, the value of the expression

$(VPM_0 \ \& \ \neg MSR_{IR}) \ | \ (VPM_1 \ \& \ MSR_{IR})$

is 1, and the next instruction to be executed cannot be fetched for any of the following reasons.

- Instruction address translation is enabled (MSR_{IR}=1) and the virtual address cannot be translated to a real address.
- Instruction address translation is disabled (MSR_{IR}=0), $LPES_1$ =1, and the virtual address cannot be translated to a real address by means of the virtual real addressing mechanism.
- The fetch access violates storage protection.

The following registers are set:

HSRR0 Set to the effective address of the instruction that the processor 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 value of the expression $(MSR_{IR}) \ | \ ((\neg MSR_{IR} \ \& \ VPM_0) \ \& \ LPES_1)$ is 1 and the translation for an attempted access is not found in the primary PTEG or in the secondary PTEG; otherwise set to 0.

34 Set to 0.

35 Set to 1 if the access is to No-execute or Guarded storage; otherwise set to 0.

36 Set to 1 if the access is not permitted by Figure 24; otherwise set to 0.

> **Programming Note**
>
> Storage protection violations for the Hypervisor Data Storage Interrupt are reported in $HDSISR_{36}$, whereas storage protection violations for the Hypervisor Instruction Storage Interrupt are reported in $HSRR1_{35}$ and $HSRR1_{36}$.

42:46 Set to 0.

47 Set to 0.

Others Loaded from the MSR.

MSR See Figure 37.

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 HSRR1.

Execution resumes at effective address 0x0000_0000_0000_0E10.

6.5.17 Hypervisor Data Segment Interrupt

A Hypervisor Data Segment interrupt may occur when the processor is not in hypervisor state, data address translation is disabled (MSR_{DR}=0), VPM_0=1, $LPES_1$=1, no higher priority exception exists, the effective address of any byte of the storage location specified by a *Load*, *Store*, ***icbi***, ***dcbz***, ***dcbst***, ***dcbf[l]*** ***eciwx***, or ***ecowx*** instruction is beyond the 1 TB VRMA.

If a ***stwcx.*** or ***stdcx.*** would not perform its store in the absence of a Hypervisor Data Segment interrupt, and a non-conditional *Store* to the specified effective address would cause a Hypervisor Data Segment interrupt, it is implementation-dependent whether a Hypervisor Data Segment interrupt occurs.

If a *Move Assist* instruction has a length of zero (in the XER), a Hypervisor Data Segment interrupt does not occur, regardless of the effective address.

The following registers are set:

HSRR0 Set to the effective address of the instruction that caused the interrupt.

HSRR1

33:36 Set to 0.

42:47 Set to 0.

Others Loaded from the MSR.

MSR See Figure 37.

HDSISR Set to an undefined value.

HDAR Set to the effective address of a storage element as described in the following list.

- a byte in the block that caused the Hypervisor Data Segment interrupt, for a *Cache Management* instruction
- a byte in the first aligned doubleword for which access was attempted in the segment that caused the Hypervisor Data Segment interrupt, for a *Load*, *Store*, ***eciwx***, or ***ecowx*** instruction ("first" refers to address order; see Section 6.7)

Execution resumes at effective address 0x0000_0000_0000_0E20.

6.5.18 Hypervisor Instruction Segment Interrupt

A Hypervisor Instruction Segment interrupt may occur when the processor is not in hypervisor state, instruction address translation is disabled (MSR_{IR}=0), VPM_0=1, $LPES_1$=1, no higher priority exception exists, and the effective address of any byte of the instruction is beyond the 1 TB VRMA.

The following registers are set:

HSRR0 Set to the effective address of the instruction that the processor 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
- **33:36** Set to 0.
- **42:47** Set to 0.
- **Others** Loaded from the MSR.

MSR See Figure 37 on page 466.

Execution resumes at effective address 0x0000_0000_0000_03E0.

6.5.19 Performance Monitor Interrupt [Category: Server.Performance Monitor]

The Performance Monitor interrupt is part of the Performance Monitor facility; see Appendix C. If the Performance Monitor facility is not implemented or does not use this interrupt, the corresponding interrupt vector (see Figure 38 on page 466) is treated as reserved.

6.5.20 Vector Unavailable Interrupt [Category: Vector]

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
- **33:36** Set to 0.
- **42:47** Set to 0.
- **Others** Loaded from the MSR.

MSR See Figure 37 on page 466.

Execution resumes at effective address 0x0000_0000_0000_0F20.

6.6 Partially Executed Instructions

If a Data Storage, Data Segment, Alignment, system-caused, or imprecise exception occurs while a *Load* or *Store* instruction is executing, the instruction may be aborted. In such cases the instruction is not completed, but may have been partially executed in the following respects.

- Some of the bytes of the storage operand may have been accessed, except that if access to a given byte of the storage operand would violate storage protection, that byte is neither copied to a register by a *Load* instruction nor modified by a *Store* instruction. Also, the rules for storage accesses given in Section 5.8.1, “Guarded Storage” and in Section 2.1 of Book II are obeyed.
- Some registers may have been altered as described in the Book II section cited above.
- Reference and Change bits may have been updated as described in Section 5.7.8.
- For a ***stwcx.*** or ***stdcx.*** instruction that is executed in-order, CR0 may have been set to an undefined value and the reservation may have been cleared.
- For an ***lq*** instruction that is executed in-order, the TGCC may have been set to an undefined value.

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 processor, 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 or one item in the list applies.

- For a fixed-point *Load* instruction that is not a multiple or string form, or for an ***eciwx*** instruction, if RT=RA or RT=RB then the contents of register RT are not altered.
- For an ***lq*** instruction, if RT+1 = RA then the contents of register RT+1 are not altered.
- For an update form *Load* or *Store* instruction, the contents of register RA are not altered.

6.7 Exception Ordering

Since multiple exceptions can exist at the same time and the architecture does not provide for reporting more than one interrupt at a time, the generation of more than one interrupt is prohibited. Some exceptions, such as the 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, Hypervisor 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).

6.7.1 Unordered Exceptions

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.

1. System Reset
2. Machine Check

6.7.2 Ordered Exceptions

The exceptions listed here are ordered with respect to the state of the interrupt processing mechanism. In the following list, the hypervisor forms of the Data Storage, Instruction Storage, Data Segment, and Instruction Segment 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.

System-Caused or Imprecise

1. Program
   - Imprecise Mode Floating-Point Enabled Exception
2. External and [Hypervisor] Decrementer

Instruction-Caused and Precise

1. [Hypervisor] Instruction Segment
2. [Hypervisor] Instruction Storage
3. Program
   - Illegal Instruction
   - Privileged Instruction
4. Function-Dependent
   4.a Fixed-Point and Branch
      1a Program
         - Trap
      1b System Call
      1c [Hypervisor] Data Storage, [Hypervisor] Data Segment, or Alignment
      2 Trace
   4.b Floating-Point
      1 FP Unavailable
      2a Program
         - Precise Mode Floating-Pt Enabled Excep'n
      2b [Hypervisor] Data Storage, [Hypervisor] Data Segment, or Alignment
      3 Trace
   4.c Vector
      1 Vector Unavailable
      2a [Hypervisor] Data Storage, [Hypervisor] Data Segment, or Alignment
      3 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 External, Decrementer, and Hypervisor Decrementer interrupts have equal ordering. Similarly, where Data Storage, Data Segment, and Alignment exceptions are listed in the same item they have equal ordering.

Even on processors that are capable of executing several instructions simultaneously, or out of order, instruction-caused interrupts (precise and imprecise) occur in program order.

6.8 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. In the following list, the hypervisor forms of the Data Storage, Instruction Storage, Data Segment, and Instruction Segment exceptions can be substituted for the non-hypervisor forms since the hypervisor forms cannot occur simultaneously and have the same priority.

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

   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.

   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-Dependent

   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, [Hypervisor] Data Segment, and Alignment exceptions are listed in the same item they have equal priority (i.e., the processor 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:
         - Program - Illegal Instruction
         - Program - Privileged Instruction
      b. [Hypervisor] Data Storage, [Hypervisor] Data Segment, or Alignment
      c. Trace

   B. Floating-Point Loads and Stores
      a. Program - Illegal Instruction
      b. Floating-Point Unavailable
      c. [Hypervisor] Data Storage, [Hypervisor] Data Segment, or Alignment
      d. Trace

   C. Vector Loads and Stores
      a. Program - Illegal Instruction
      b. Vector Unavailable
      c. [Hypervisor] Data Storage, [Hypervisor] Data Segment, or Alignment
      d. Trace

   D. Other Floating-Point Instructions
      a. Floating-Point Unavailable
      b. Program - Precise Mode Floating-Point Enabled Exception
      c. Trace

   E. Other Vector Instructions
      a. Vector Unavailable
      b. Trace

   F. ***rfid, hrfid*** and ***mtmsr***[***d***]
      a. Program - Privileged Instruction
      b. Program - Floating-Point Enabled Exception
      c. Trace, for ***mtmsr***[***d***] only

   G. Other Instructions
      a. These exceptions are mutually exclusive and have the same priority:
         - Program - Trap
         - System Call
         - Program - Privileged Instruction
         - Program - Illegal Instruction
      b. Trace

   H. [Hypervisor] Instruction Storage and [Hypervisor] Instruction Segment

      These exceptions have the lowest priority in this category. They are recognized only when all instructions prior to the instruction causing one of these exceptions appear to have completed and that instruction is the next instruction to be executed. The two exceptions are mutually exclusive.

      The priority of these exceptions is specified for completeness and to ensure that they are not given more favorable treatment. It is acceptable for an implementation to treat these exceptions as though they had a lower priority.

4. Program - Imprecise Mode Floating-Point Enabled Exception

   This exception is the fourth highest priority exception. When this exception is created, the interrupt mechanism waits for all other possible exceptions to be reported. It then generates this interrupt if no higher priority exception exists when the interrupt is to be generated.

5. External and [Hypervisor] Decrementer

   These exceptions are the lowest priority exceptions. All have equal priority (i.e., the processor may generate any one of these 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.

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., illegal 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). Similarly, if the Real Mode Area is virtualized and there is no PTE for the page containing the interrupt vectors, every attempt to execute the first instruction of the OS's Instruction Storage interrupt handler would cause a Hypervisor Instruction Storage interrupt; if the Hypervisor Instruction Storage interrupt handler returns to the OS's Instruction Storage interrupt handler without the relevant PTE having been created, another Hypervisor Instruction Storage interrupt would occur immediately. The looping caused by these and similar cases is terminated by the occurrence of a System Reset or Hypervisor Decrementer interrupt.

Chapter 7. Timer Facilities

7.1 Overview

The Time Base, Decrementer, Hypervisor Decrementer, and the Processor Utilization of Resources Register, provide timing functions for the system. The remainder of this section describes these registers and related facilities.

7.2 Time Base (TB)

The Time Base (TB) is a 64-bit register (see Figure 39) containing a 64-bit unsigned integer that is incremented periodically. Each increment adds 1 to the low-order bit (bit 63). The frequency at which the integer is updated is implementation-dependent.

0	39	
TBU40		///
TBU	TBL	
0	32	63

Field	Description
TBU40	Upper 40 bits of Time Base
TBU	Upper 32 bits of Time Base
TBL	Lower 32 bits of Time Base

Figure 39. 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 39. 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.

The Time Base 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 period of the Time Base depends on the driving frequency. As an order of magnitude example, suppose that the CPU clock is 1 GHz and that the Time Base is driven by this frequency divided by 32. Then the period of the Time Base would be

$$T_{TB} = \frac{2^{64} \times 32}{1\ \text{GHz}} = 5.90 \times 10^{11}\ \text{seconds}$$

which is approximately 18,700 years.

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 privileged and, in implementations of the Server environment, must be accessible only in hypervisor state ($MSR_{HV\ PR}$ = 0b10). There must be a method for getting all processors' Time Bases to start incrementing with values that are identical or almost identical in all processors.

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.

See the description of the Time Base in Chapter 4 of Book II for ways to compute time of day in POSIX format from the Time Base.

7.2.1 Writing the Time Base

Writing the Time Base is privileged, and can be done only in hypervisor state. Reading the Time Base is not privileged; it is discussed in Chapter 4 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; see Appendix A, "Assembler Extended Mnemonics" on page 493.

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
mttbu40Rx      # 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.

7.3 Decrementer

The Decrementer (DEC) is a 32-bit decrementing counter that provides a mechanism for causing a Decrementer interrupt after a programmable delay. The contents of the Decrementer are treated as a signed integer.

DEC	
32	63

Figure 40. Decrementer

The Decrementer is driven by the same frequency as the Time Base. The period of the Decrementer will depend on the driving frequency, but if the same values are used as given above for the Time Base (see Section 7.2), and if the Time Base update frequency is constant, the period would be

$$T_{DEC} = \frac{2^{32} \times 32}{1\ \text{GHz}} = 137 \text{ seconds.}$$

The Decrementer counts down.

When the contents of DEC_{32} change from 0 to 1, a Decrementer exception will come into existence within a reasonable period or time. When the contents of DEC_{32} change from 1 to 0, an existing 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.

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 processor or of modification of the Decrementer caused by execution of an ***mtspr*** instruction.

The operation of the Decrementer satisfies the following constraints.

1. The operation of the Time Base and the Decrementer is coherent, i.e., the counters are driven by the same fundamental time base.
2. Loading a GPR from the Decrementer has no effect on the accuracy of the Time Base.
3. 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.

7.3.1 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 (see Appendix A, “Assembler Extended Mnemonics” on page 493), the Decrementer can be written from GPR Rx using:

```
mtdec  Rx
```

The Decrementer can be read into GPR Rx using:

```
mfdec  Rx
```

Copying the Decrementer to a GPR has no effect on the Decrementer contents or on the interrupt mechanism.

7.4 Hypervisor Decrementer

The Hypervisor Decrementer (HDEC) is a 32-bit decrementing counter that provides a mechanism for causing a Hypervisor Decrementer interrupt after a programmable delay. The contents of the Decrementer are treated as a signed integer.

HDEC
32 … 63

Figure 41. Hypervisor Decrementer

The Hypervisor Decrementer is a hypervisor resource; see Chapter 2.

The Hypervisor Decrementer is driven by the same frequency as the Time Base. The period of the Hypervisor Decrementer will depend on the driving frequency, but if the same values are used as given above for the Time Base (see Section 7.2), and if the Time Base update frequency is constant, the period would be

$$T_{DEC} = \frac{2^{32} \times 32}{1\text{ GHz}} = 137\text{ seconds.}$$

When the contents of $HDEC_{32}$ change from 0 to 1, a Hypervisor Decrementer exception will come into existence within a reasonable period or time. When the contents of $HDEC_{32}$ change from 1 to 0, an 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.

The preceding paragraph applies regardless of whether the change in the contents of $HDEC_{32}$ is the result of decrementation of the Hypervisor Decrementer by the processor or of modification of the Hypervisor Decrementer caused by execution of an ***mtspr*** instruction.

The operation of the Hypervisor Decrementer satisfies the following constraints.

1. The operation of the Time Base and the Hypervisor Decrementer is coherent, i.e., the counters are driven by the same fundamental time base.
2. Loading a GPR from the Hypervisor Decrementer has no effect on the accuracy of the Hypervisor Decrementer.
3. 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.

7.5 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 processor. The contents of the PURR are treated as a 64-bit unsigned integer.

PURR
0 … 63

Figure 42. 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 processor with respect to other processors that share those resources monitored by the PURR.

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 processor during the interval T_{ab}. For the set {S} of processors that share the resources monitored by the PURR, the sum of the usage estimates for all the processors in the set is 1.0.

The definition of the set of processors 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 of resource use accounting, program dispatching, etc.

Because the rate at which the PURR accumulates resource usage estimates is dependent on the frequency at which the Time Base is incremented, the interpretation of the contents of the PURR must be adjusted if the frequency at which the Time Base is incremented is altered.

Chapter 8. Debug Facilities

8.1 Overview

Processors provide debug facilities to enable hardware and software debug functions, such as instructions and data breakpoints and program single stepping. The debug facilities consist of a data address breakpoint register (DABR), a data address breakpoint register extension (DABRX) (see Section 8.1.1) and an associated interrupt (see Section 6.5.3).

The ***mfspr*** and ***mtspr*** instructions (see Section 4.4.3) provide access to the registers of the debug facilities.

In addition to the facilities described here, implementations will typically include debug facilities, modes, and access mechanisms which are implementation-specific. For example, implementations will typically provide access to the debug facilities via a dedicated interface such as the IEEE 1149.1 Test Access Port (JTAG).

8.1.1 Data Address Breakpoint

The Data Address Breakpoint mechanism provides a means of detecting load and store accesses to a designated doubleword. The address comparison is done on an effective address (EA).

The Data Address Breakpoint mechanism is controlled by the Data Address Breakpoint Register (DABR), shown in Figure 43, and the Data Address Breakpoint Register Extension (DABRX), shown in Figure 44.

DAB	BT	DW	DR
0	61	62	63

Bit(s)	Name	Description
0:60	DAB	Data Address Breakpoint
61	BT	Breakpoint Translation
62	DW	Data Write
63	DR	Data Read

Figure 43. Data Address Breakpoint Register

///	BTI	PRIVM
0	60	61 63

Bit(s)	Name	Description
60	BTI	Breakpoint Translation Ignore
61:63	PRIVM	Privilege Mask
61	HYP	Hypervisor state
62	PNH	Privileged but Non-Hypervisor state
63	PRO	Problem state

All other fields are reserved.

Figure 44. Data Address Breakpoint Register Extension

The DABR and DABRX are hypervisor resources; see Section 2.6 on page 399.

The supported PRIVM values are 0b000, 0b001, 0b010, 0b011, 0b100, and 0b111. If the PRIVM field does not contain one of the supported values, then whether a match occurs for a given storage access is undefined. Elsewhere in this section it is assumed that the PRIVM field contains one of the supported values.

Programming Note

PRIVM value 0b000 causes matches not to occur regardless of the contents of other DABR and DABRX 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, as described in a subsequent Programming Note.)

A Data Address Breakpoint match occurs for a Load or Store instruction if, for any byte accessed, all of the following conditions are satisfied.

- $EA_{0:60} = DABR_{DAB}$
- $(MSR_{DR} = DABR_{BT})$ | $DABRX_{BTI}$
- if the processor is in
  - hypervisor state and $DABRX_{HYP} = 1$ or
  - privileged but non-hypervisor state and $DABRX_{PNH} = 1$ or
  - problem state and $DABRX_{PR} = 1$
- the instruction is a *Store* and $DABR_{DW} = 1$, or the instruction is a *Load* and $DABR_{DR} = 1$.

In 32-bit mode the high-order 32 bits of the EA are treated as zeros for the purpose of detecting a match.

If the above conditions are satisfied, a match also occurs for ***eciwx*** and ***ecowx***. For the purpose of determining whether a match occurs, ***eciwx*** is treated as a *Load*, and ***ecowx*** is treated as a *Store*.

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 a *Load/Store String* of zero length.
- 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 Breakpoint match causes a Data Storage exception (see Section 6.5.3, “Data Storage Interrupt” on page 467). If a match occurs, some or all of the bytes of the storage operand may have been accessed; however, if a *Store* or ***ecowx*** 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
- ***ecowx***

Programming Note

The Data Address Breakpoint mechanism does not apply to instruction fetches.

Programming Note

Before setting a breakpoint requested by the operating system, the hypervisor must verify that the requested contents of the DABR and DABRX cannot cause the hypervisor to receive a Data Storage interrupt that it is not prepared to handle, or that it intrinsically cannot handle (e.g., the EA is in the range of EAs at which the hypervisor's Data Storage interrupt handler saves registers, $DABR_{BT}$ || $DABRX_{BTI} \neq 0b10$, $DABR_{DW} = 1$, and $DABRX_{HYP} = 1$).

Programming Note

Processors that comply with versions of the architecture that precede Version 2.02 do not provide the DABRX. Forward compatibility for software that was written for such processors (and uses the Data Address Breakpoint facility) can be obtained by setting $DABRX_{60:63}$ to 0b0111.

Chapter 9. External Control [Category: External Control]

The External Control facility permits a program to communicate with a special-purpose device. The facility consists of a Special Purpose Register, called EAR, and two instructions, called *External Control In Word Indexed* (***eciwx***) and *External Control Out Word Indexed* (***ecowx***).

This facility must provide a means of synchronizing the devices with the processor to prevent the use of an address by the device when the translation that produced that address is being invalidated.

9.1 External Access Register

This 32-bit Special Purpose Register controls access to the External Control facility and, for external control operations that are permitted, identifies the target device.

E	///	RID
32	33	58 63

Bit(s)	Name	Description
32	E	Enable bit
58:63	RID	Resource ID

All other fields are reserved.

Figure 45. External Access Register

The EAR is a hypervisor resource; see Chapter 2.

The high-order bits of the RID field that correspond to bits of the Resource ID beyond the width of the Resource ID supported by the implementation are treated as reserved bits.

Programming Note

The hypervisor can use the EAR to control which programs are allowed to execute *External Access* instructions, when they are allowed to do so, and which devices they are allowed to communicate with using these instructions.

9.2 External Access Instructions

The *External Access* instructions, *External Control In Word Indexed* (***eciwx***) and *External Control Out Word Indexed* (***ecowx***), are described in Book II. Additional information about them is given below.

If attempt is made to execute either of these instructions when EAR_E=0, a Data Storage interrupt occurs with bit 43 of the DSISR set to 1.

The instructions are supported whenever MSR_{DR}=1. If either instruction is executed when MSR_{DR}=0 (real addressing mode), the results are boundedly undefined.

Chapter 10. 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 1 (for data access) and Table 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.

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 2, because all instructions preceding the context-altering instruction are fetched and decoded before the context-altering instruction is executed (the processor must determine whether any of these preceding instructions are context synchronizing).

Unless otherwise stated, the material in this chapter assumes a uniprocessor environment.

Instruction or Event	Required Before	Required After	Notes
interrupt	none	none	
rfid	none	none	
hrfid	none	none	
sc	none	none	
Trap	none	none	
mtmsrd (SF)	none	none	
mtmsr[***d***] (PR)	none	none	
mtmsr[***d***] (DR)	none	none	
mtsr[***in***]	CSI	CSI	
mtspr (SDR1)	***ptesync***	CSI	3,4
mtspr (AMR)	CSI	CSI	
mtspr (EAR)	CSI	CSI	
mtspr (RMOR)	CSI	CSI	13
mtspr (HRMOR)	CSI	CSI	13
mtspr (LPCR)	CSI	CSI	13
mtspr (DABR)	--	--	2
mtspr (DABRX)	--	--	2
slbie	CSI	CSI	
slbia	CSI	CSI	
slbmte	CSI	CSI	11
tlbie	CSI	CSI	5,7
tlbiel	CSI	***ptesync***	5
tlbia	CSI	CSI	5
Store(PTE)	none	{***ptesync***, CSI}	6,7

Table 1: Synchronization requirements for data access

Instruction or Event	Required Before	Required After	Notes
interrupt	none	none	
rfid	none	none	
hrfid	none	none	
sc	none	none	
Trap	none	none	
mtmsrd (SF)	none	none	8
mtmsr[***d***] (EE)	none	none	1
mtmsr[***d***] (PR)	none	none	9
mtmsr[***d***] (FP)	none	none	
mtmsr[***d***](FE0,FE1)	none	none	
mtmsr[***d***] (SE, BE)	none	none	
mtmsr[***d***] (IR)	none	none	9
mtmsr[***d***] (RI)	none	none	
mtsr[***in***]	none	CSI	9
mtspr (DEC)	none	none	10
mtspr (SDR1)	***ptesync***	CSI	3,4
mtspr (CTRL)	none	none	
mtspr (HDEC)	none	none	10
mtspr (RMOR)	none	CSI	13
mtspr (HRMOR)	none	CSI	9,13
mtspr (LPCR)	none	CSI	13
mtspr (LPIDR)	CSI	CSI	7,12
slbie	none	CSI	
slbia	none	CSI	
slbmte	none	CSI	9,11
tlbie	none	CSI	5,7
tlbiel	none	CSI	5
tlbia	none	CSI	5
Store(PTE)	none	{***ptesync***, CSI}	6,7

Table 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 nor a Decrementer 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, 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 processor 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. SDR1 must not be altered when MSR_{DR}=1 or MSR_{IR}=1; if it is, the results are undefined.
4. A ***ptesync*** instruction is required before the ***mtspr*** instruction because (a) SDR1 identifies the Page Table and thereby the location of Reference and Change bits, and (b) on some implementations, use of SDR1 to update Reference and Change bits may be independent of translating the virtual address. (For example, an implementation might identify the PTE in which to update the Reference and Change bits in terms of its offset in the Page Table, instead of its real address, and then add the Page Table address from SDR1 to the offset to determine the real address at which to update the bits.) To ensure that Reference and Change bits are updated in the correct Page Table, SDR1 must not be altered until all Reference and Change bit updates associated with address translations that were performed, by the processor executing the ***mtspr*** instruction, before the ***mtspr*** instruction is executed have been performed with respect to that processor. A ***ptesync*** instruction guarantees this synchronization of Reference and Change bit updates, while neither a context synchronizing operation nor the instruction fetching mechanism does so.
5. For data accesses, the context synchronizing instruction before the ***tlbie***, ***tlbiel***, or ***tlbia*** 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 ***tlbie***, ***tlbiel***, or ***tlbia*** instruction ensures that storage accesses associated with instructions following the context synchronizing instruction will not use the TLB entry(s) being invalidated.

   (If it is necessary to order storage accesses associated with preceding instructions, or Reference and Change bit updates associated with preceding address translations, with respect to subsequent data accesses, a ***ptesync*** instruction must also be used, either before or after the ***tlbie***, ***tlbiel***, or ***tlbia*** instruction. These effects of the ***ptesync*** instruction are described in the last paragraph of Note 8.)
6. The notation "{***ptesync***,CSI}" denotes an instruction sequence. Other instructions may be interleaved with this sequence, but these instructions must appear in the order shown.

   No software synchronization is required before the *Store* instruction because (a) stores are not performed out-of-order and (b) address translations associated with instructions preceding the *Store* instruction are not performed again after the store has been performed (see Section 5.5). These properties ensure that all address translations associated with instructions preceding the *Store* instruction will be performed using the old contents of the PTE.

   The ***ptesync*** instruction after the *Store* instruction ensures that all searches of the Page Table that are performed after the ***ptesync*** instruction completes will use the value stored (or a value stored subsequently). The context synchronizing instruction after the ***ptesync*** instruction ensures that any address translations associated with instructions following the context synchronizing instruction that were performed using the old contents of the PTE will be discarded, with the result that these address translations will be performed again and, if there is no corresponding entry in any implementation-specific address translation lookaside information, will use the value stored (or a value stored subsequently).

   The ***ptesync*** instruction also ensures that all storage accesses associated with instructions preceding the ***ptesync*** instruction, and all Reference and Change bit updates associated with additional address translations that were performed, by the processor executing the ***ptesync*** instruction, before the ***ptesync*** instruction is executed, will be performed with respect to any processor 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 processor or mechanism.

7. There are additional software synchronization requirements for this instruction in multiprocessor environments (e.g., it may be necessary to invalidate one or more TLB entries on all processors in the multiprocessor system and to be able to determine that the invalidations have completed and that all side effects of the invalidations have taken effect).

   Section 5.10 gives examples of using ***tlbie***, *Store*, and related instructions to maintain the Page Table, in both multiprocessor and uniprocessor environments.

   Programming Note

   In a multiprocessor system, if software locking is used to help ensure that the requirements described in Section 5.10 are satisfied, the ***lwsync*** instruction near the end of the lock acquisition sequence (see Section B.2.1.1 of Book II) may naturally provide the context synchronization that is required before the alteration.

8. 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 5.3.2 on page 420).

9. 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.

10. The elapsed time between the contents of the Decrementer or Hypervisor Decrementer becoming negative and the signaling of the corresponding exception is not defined.

11. 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.

12. The context synchronizing instruction before the ***mtspr*** instruction ensures that the LPIDR is not altered out-of-order. (Out-of-order alteration of the LPIDR could permit the requirements described in Section 5.10.1 to be violated. For the same reason, such a context synchronizing instruction may be needed even if the new LPID value is equal to the old LPID value.)

    See also Chapter 2. “Logical Partitioning (LPAR)” on page 397 regarding moving a processor from one partition to another.

13. When the RMOR or HRMOR is modified, or the VC, VRMASD, RMLS, $LPES_1$, or RMI fields of the LPCR are modified, software must invalidate all implementation-specific lookaside information used in address translation that depends on values stored in these registers. All implementations provide a means by which software can do this.

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 III.

Assemblers should provide the extended mnemonics and symbols listed here, and may provide others.

A.1 Move To/From Special Purpose Register Mnemonics

This section defines extended mnemonics for the ***mtspr*** and ***mfspr*** instructions, including the Special Purpose Registers (SPRs) defined in Book I and certain privileged SPRs, 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).

Programming Note

The extended mnemonics in Table 3 for SPRs associated with the Performance Monitor facility are based on the definitions in Appendix B.

Other versions of Performance Monitor facilities used different sets of SPR numbers (all 32-bit PowerPC processors used a different set, and some early Power ISA processors used yet a different set).

Table 3: Extended mnemonics for moving to/from an SPR

Special Purpose Register	Move To SPR		Move From SPR[1]	
	Extended	Equivalent to	Extended	Equivalent to
Fixed-Point Exception Register	mtxer Rx	mtspr 1,Rx	mfxer Rx	mfspr Rx,1
Link Register	mtlr Rx	mtspr 8,Rx	mflr Rx	mfspr Rx,8
Count Register	mtctr Rx	mtspr 9,Rx	mfctr Rx	mfspr Rx,9
Data Storage Interrupt Status Register	mtdsisr Rx	mtspr 18,Rx	mfdsisr Rx	mfspr Rx,18
Data Address Register	mtdar Rx	mtspr 19,Rx	mfdar Rx	mfspr Rx,19
Decrementer	mtdec Rx	mtspr 22,Rx	mfdec Rx	mfspr Rx,22
Storage Description Register 1	mtsdr1 Rx	mtspr 25,Rx	mfsdr1 Rx	mfspr Rx,25
Save/Restore Register 0	mtsrr0 Rx	mtspr 26,Rx	mfsrr0 Rx	mfspr Rx,26
Save/Restore Register 1	mtsrr1 Rx	mtspr 27,Rx	mfsrr1 Rx	mfspr Rx,27
AMR	mtamr Rx	mtspr 29,Rx	mfamr Rx	mfspr Rx,29
CTRL	mtctrl Rx	mtspr 152,Rx	mfctrl Rx	mfspr Rx,136
Special Purpose Registers G0 through G3	mtsprg n,Rx	mtspr 272+n,Rx	mfsprg Rx,n	mfspr Rx,272+n
Time Base [Lower]	mttbl Rx	mtspr 284,Rx	mftb Rx	mftb Rx,268[1] mfspr Rx,268
Time Base Upper	mttbu Rx	mtspr 285,Rx	mftbu Rx	mftb Rx,269[1] mfspr Rx,269
Time Base Upper 40	mttbu40 Rx	mtspr 286,Rx	-	-
Processor Version Register	-	-	mfpvr Rx	mfspr Rx,287
MMCRA	mtmmcra Rx	mtspr 786,Rx	mfmmcra Rx	mfspr Rx,770
PMC1	mtpmc1 Rx	mtspr 787,Rx	mfpmc1 Rx	mfspr Rx,771
PMC2	mtpmc2 Rx	mtspr 788,Rx	mfpmc2 Rx	mfspr Rx,772
PMC3	mtpmc3 Rx	mtspr 789,Rx	mfpmc3 Rx	mfspr Rx,773
PMC4	mtpmc4 Rx	mtspr 790,Rx	mfpmc4 Rx	mfspr Rx,774
PMC5	mtpmc5 Rx	mtspr 791,Rx	mfpmc5 Rx	mfspr Rx,775
PMC6	mtpmc6 Rx	mtspr 792,Rx	mfpmc6 Rx	mfspr Rx,776
MMCR0	mtmmcr0 Rx	mtspr 795,Rx	mfmmcr0 Rx	mfspr Rx,779
MMCR1	mtmmcr1 Rx	mtspr 798,Rx	mfmmcr1 Rx	mfspr Rx,782
PPR	mtppr Rx	mtspr 896, Rx	mfppr Rx	mfspr Rx, 896
Processor Identification Register	-	-	mfpir Rx	mfspr Rx,1023

[1] The ***mftb*** instruction is Category: Server.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 4.2.1 of Book II).

Appendix B. Example Performance Monitor

Note

This Appendix describes an example implementation of a Performance Monitor. A subset of these requirements are being considered for inclusion in the Architecture as part of Category: Server.Performance Monitor.

A Performance Monitor facility provides a means of collecting information about program and system performance.

The resources (e.g., SPR numbers) that a Performance Monitor facility may use are identified elsewhere in this Book. All other aspects of any Performance Monitor facility are implementation-dependent.

This appendix provides an example of a Performance Monitor facility. It is only an example; implementations may provide all, some, or none of the features described here, or may provide features that are similar to those described here but differ in detail.

Programming Note

Because the features provided by a Performance Monitor facility are implementation-dependent, operating systems should provide services that support the useful performance monitoring functions in a generic fashion. Application programs should use these services, and should not depend on the features provided by a particular implementation.

The example Performance Monitor facility consists of the following features (described in detail in subsequent sections).

- one MSR bit
  - PMM (Performance Monitor Mark), which can be used to select one or more programs for monitoring
- SPRs
  - PMC1 - PMC6 (Performance Monitor Counter registers 1 - 6), which count events
  - MMCR0, MMCR1, and MMCRA (Monitor Mode Control Registers 0, 1, and A), which control the Performance Monitor facility
  - SIAR and SDAR (Sampled Instruction Address Register and Sampled Data Address Register), which contain the address of the “sampled instruction” and of the “sampled data”
- the Performance Monitor interrupt, which can be caused by monitored conditions and events

The minimal subset of the features that makes the resulting Performance Monitor useful to software consists of MSR_{PMM}, PMC1, PMC2, PMC3, PMC4, MMCR0, MMCR1, and MMCRA and certain bits and fields of these three Monitor Mode Control Registers, and the Performance Monitor Interrupt. These features are identified as the “basic” features below. The remaining features (the remaining SPRs, and the remaining bits and fields in the three Monitor Mode Control Registers) are considered “extensions”.

The events that can be counted in the PMCs as well as the code that identifies each event are implementation-dependent. The events and codes may vary between PMCs, as well as between implementations. For the programmable PMCs, the event to be counted is selected by specifying the appropriate code in the MMCR “Selector” field for the PMC. Some events may include operations that are performed out-of-order.

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 an MMCR field). 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 processor (e.g., incrementing a PMC) or explicitly by software (***mtspr***).
- A condition or event is enabled if the corresponding “Enable” bit in an MMCR 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

an MMCR. A single Performance Monitor alert may reflect multiple enabled conditions and events.

- A Performance Monitor alert causes a Performance Monitor exception.

  The exception effects of the Performance Monitor are said to be consistent with the contents of $MMCR0_{PMAO}$ if one of the following statements is true. ($MMCR0_{PMAO}$ reflects the occurrence of Performance Monitor alerts; see the definition of that bit in Section B.2.2.)
  - $MMCR0_{PMAO}$=0 and a Performance Monitor exception does not exist.
  - $MMCR0_{PMAO}$=1 and a Performance Monitor exception exists.

  A context synchronizing instruction or event that occurs when $MMCR0_{PMAO}$=0 ensures that the exception effects of the Performance Monitor are consistent with the contents of $MMCR0_{PMAO}$.

  Even without software synchronization, when the contents of $MMCR0_{PMAO}$ change, the exception effects of the Performance Monitor become consistent with the new contents of $MMCR0_{PMAO}$ sufficiently soon that the Performance Monitor facility is useful to software for its intended purposes.

- A Performance Monitor exception causes a Performance Monitor interrupt when MSR_{EE}=1.

Programming Note

The Performance Monitor can be effectively disabled (i.e., put into a state in which Performance Monitor SPRs are not altered and Performance Monitor interrupts do not occur) by setting MMCR0 to 0x0000_0000_8000_0000.

B.1 PMM Bit of the Machine State Register

The Performance Monitor uses MSR bit PMM, which is defined as follows.

Bit **Description**

61 ***Performance Monitor Mark*** (PMM)

This bit is a basic feature.

This bit contains the Performance Monitor "mark" (0 or 1).

Programming Note

Software can use this bit as a process-specific marker which, in conjunction with $MMCR0_{FCM0\ FCM1}$ (see Section B.2.2), 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.

- 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
- 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

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 37 on page 466), 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.

B.2 Special Purpose Registers

The Performance Monitor SPRs count events, control the operation of the Performance Monitor, and provide associated information.

The Performance Monitor SPRs can be read and written using the ***mfspr*** and ***mtspr*** instructions (see Section 4.4.3, "Move To/From System Register Instructions" on page 411). The Performance Monitor SPR numbers are shown in Figure 46. Writing any of the Performance Monitor SPRs is privileged. Reading any of the Performance Monitor SPRs is *not* privileged (however, the privileged SPR numbers used to write the SPRs can also be used to read them; see the figure).

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 SPRs 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 SPRs. 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 SPRs. Thus the contents obtained by reading a Performance Monitor SPR 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 processor is such that the SPR is subject to change by the processor 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 SPR other than SIAR or SDAR, the change is not guaranteed to have taken effect until after a subsequent context synchronizing instruction has been executed (see Chapter 10. “Synchronization Requirements for Context Alterations” on page 489).

Programming Note

Depending on the events being monitored, the contents of Performance Monitor SPRs may be affected by aspects of the runtime environment (e.g., cache contents) that are not directly attributable to the programs being monitored.

decimal	SPR[1,2] $spr_{5:9}$ $spr_{0:4}$	Register Name	Privileged
770,786	11000 n0010	MMCRA	no,yes
771,787	11000 n0011	PMC1	no,yes
772,788	11000 n0100	PMC2	no,yes
773,789	11000 n0101	PMC3	no,yes
774,790	11000 n0110	PMC4	no,yes
775,791	11000 n0111	PMC5	no,yes
776,792	11000 n1000	PMC6	no,yes
779,795	11000 n1011	MMCR0	no,yes
780,796	11000 n1100	SIAR	no,yes
781,797	11000 n1101	SDAR	no,yes
782,798	11000 n1110	MMCR1	no,yes

1 Note that the order of the two 5-bit halves of the SPR number is reversed.
2 For ***mtspr***, n must be 1. For ***mfspr***, reading the SPR is privileged if and only if n=1.

Figure 46. Performance Monitor SPR encodings for mtspr and mfspr

B.2.1 Performance Monitor Counter Registers

The six Performance Monitor Counter registers, PMC1 through PMC6, are 32-bit registers that count events.

	PMC1	
	PMC2	
	PMC3	
	PMC4	
	PMC5	
	PMC6	
32		63

Figure 47. Performance Monitor Counter registers

PMC1, PMC2, PMC3, and PMC4 are basic features. PMC5 and PMC6 are not programmable. PMC5 counts instructions completed and PMC6 counts cycles.

Normally each PMC is incremented each processor cycle by the number of times the corresponding event occurred in that cycle. Other modes of incrementing may also be provided (e.g., see the description of MMCR1 bits PMC1HIST and PMCjHIST).

“PMCj” is used as an abbreviation for “PMCi, i > 1”.

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 interrupt to occur, etc.

B.2.2 Monitor Mode Control Register 0

Monitor Mode Control Register 0 (MMCR0) is a 64-bit register. This register, along with MMCR1 and

MMCRA, controls the operation of the Performance Monitor.

MMCR0
0 ... 63

Figure 48. Monitor Mode Control Register 0

MMCR0 is a basic feature. Within MMCR0, some of the bits and fields are basic features and some are extensions. The basic bits and fields are identified as such, below.

Some bits of MMCR0 are altered by the processor when various events occur, as described below.

The bit definitions of MMCR0 are as follows. MMCR0 bits that are not implemented are treated as reserved.

Bit(s) **Description**

0:31 Reserved

32 ***Freeze Counters*** (FC)

This bit is a basic feature.

0 The PMCs are incremented (if permitted by other MMCR bits).
1 The PMCs are not incremented.

The processor sets this bit to 1 when an enabled condition or event occurs and $MMCR0_{FCECE}$=1.

33 ***Freeze Counters in Privileged State*** (FCS)

This bit is a basic feature.

0 The PMCs are incremented (if permitted by other MMCR bits).
1 The PMCs are not incremented if $MSR_{HV\ PR}$=0b00.

34 ***Freeze Counters in Problem State*** (FCP)

This bit is a basic feature.

0 The PMCs are incremented (if permitted by other MMCR bits).
1 The PMCs are not incremented if MSR_{PR}=1.

35 ***Freeze Counters while Mark = 1*** (FCM1)

This bit is a basic feature.

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)

This bit is a basic feature.

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)

This bit is a basic feature.

0 Performance Monitor alerts are disabled.
1 Performance Monitor alerts are enabled 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 interrupts.

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 63 is selected.
01 Time Base bit 55 is selected.
10 Time Base bit 51 is selected.
11 Time Base bit 47 is selected.

Programming Note

Time Base transition events can be used to collect information about processor activity, as revealed by event counts in PMCs and by addresses in SIAR and SDAR, at periodic intervals.

In multiprocessor systems in which the Time Base registers are synchronized among the processors, Time Base transition events can be used to correlate the Performance Monitor data obtained by the several processors. For this use, software must specify the same TBSEL value for all the processors 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.

42:47 Reserved

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

51:52 Setting is implementation-dependent.

53:55 Reserved

56 ***Performance Monitor Alert Occurred*** (PMAO)

This bit is a basic feature.

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 processor 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 to simulate the occurrence of a Performance Monitor alert.
>
> Software should set this bit to 0 after handling the Performance Monitor alert.

57 Setting is implementation-dependent.

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 in Wait State*** (FCWAIT)

This bit is a basic feature.

- 0 The PMCs are incremented (if permitted by other MMCR bits).
- 1 The PMCs are not incremented if $CTRL_{31}$=0. Software is expected to set $CTRL_{31}$=0 when it is in a “wait state”, i.e, when there is no process ready to run.

Only Branch Unit type of events do not increment if $CTRL_{31}$=0. Other units continue to count.

63 ***Freeze Counters in Hypervisor State*** (FCH)

This bit is a basic feature.

- 0 The PMCs are incremented (if permitted by other MMCR bits).
- 1 The PMCs are not incremented if $MSR_{HV\ PR}$=0b10.

B.2.3 Monitor Mode Control Register 1

Monitor Mode Control Register 1 (MMCR1) is a 64-bit register. This register, along with MMCR0 and MMCRA, controls the operation of the Performance Monitor.

MMCR1
0 ... 63

Figure 49. Monitor Mode Control Register 1

MMCR1 is a basic feature. Within MMCR1, some of the bits and fields are basic features and some are extensions. The basic bits and fields are identified as such, below.

Some bits of MMCR1 are altered by the processor when various events occur, as described below.

The bit definitions of MMCR1 are as follows. MMCR1 bits that are not implemented are treated as reserved.

Bit(s) Description

0:31 Implementation-Dependent Use

These bits have implementation-dependent uses (e.g., extended event selection).

32:39 ***PMC1 Selector*** (PMC3SEL)
40:47 ***PMC2 Selector*** (PMC4SEL)
48:55 ***PMC3 Selector*** (PMC5SEL)
56:63 ***PMC4 Selector*** (PMC6SEL)

Each of these fields contains a code that identifies the event to be counted by PMCs 1 through 4 respectively.

PMC Selectors are basic features.

> **Compatibility Note**
>
> In versions of the architecture that precede Version 2.02 the PMC Selector Fields were six bits long, and were split between MMCR0 and MMCR1. PMC1-8 were all programmable.
>
> If more programmable PMCs are implemented in the future, additional MMCRs may be defined to cover the additional selectors.

B.2.4 Monitor Mode Control Register A

Monitor Mode Control Register A (MMCRA) is a 64-bit register. This register, along with MMCR0 and MMCR1, controls the operation of the Performance Monitor.

MMCRA
0 ... 63

Figure 50. Monitor Mode Control Register A

MMCRA is a basic feature. Within MMCRA, some of the bits and fields are basic features and some are extensions. The basic bits and fields are identified as such, below.

Some bits of MMCRA are altered by the processor when various events occur, as described below.

The bit definitions of MMCRA are as follows. MMCRA bits that are not implemented are treated as reserved.

Bit(s) Description

0:31 Reserved

32 ***Contents of SIAR and SDAR Are Related*** (CSSR)

Set to 1 by the processor if the contents of SIAR and SDAR are associated with the same instruction; otherwise set to 0.

33:34 Setting is implementation-dependent.

35 ***Sampled*** MSR_{HV} (SAMPHV)

Value of MSR_{HV} when the Performance Monitor Alert occurred.

36 ***Sampled*** MSR_{PR} (SAMPPR)

Value of MSR_{PR} when the Performance Monitor Alert occurred.

37:47 Setting is implementation-dependent.

48:53 ***Threshold*** (THRESHOLD)

This field contains a “threshold value”, which is a value such that only events that exceed the value are counted. The events to which a threshold value can apply are implementation-dependent, as are the dimension of the threshold (e.g., duration in cycles) and the granularity with which the threshold value is interpreted.

Programming Note

By varying the threshold value, software can obtain a profile of the characteristics of the events subject to the threshold. For example, if PMC1 counts the number of cache misses for which the duration exceeds the threshold value, then software can obtain the distribution of cache miss durations for a given program by monitoring the program repeatedly using a different threshold value each time.

54:59 Reserved for implementation-specific use.

60:62 Reserved

63 Setting is implementation-dependent.

B.2.5 Sampled Instruction Address Register

The Sampled Instruction Address Register (SIAR) is a 64-bit register. It contains the address of the “sampled instruction” when a Performance Monitor alert occurs.

SIAR
0 ... 63

Figure 51. Sampled Instruction Address Register

When a Performance Monitor alert occurs, SIAR is set to 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. This instruction is called the “sampled instruction”.

The contents of SIAR may be altered by the processor if and only if $MMCR0_{PMAE}$=1. Thus after the Performance Monitor alert occurs, the contents of SIAR are not altered by the processor 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.

See Section B.4 regarding the effects of the Trace facility on SIAR.

Programming Note

If the Performance Monitor alert causes a Performance Monitor interrupt, the value of $MSR_{HV\ PR}$ that was in effect when the sampled instruction was being executed is reported in MMCRA.

B.2.6 Sampled Data Address Register

The Sampled Data Address Register (SDAR) is a 64-bit register. It contains the address of the “sampled data” when a Performance Monitor alert occurs.

SDAR
0 ... 63

Figure 52. Sampled Data Address Register

When a Performance Monitor alert occurs, SDAR is set to 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 is called the “sampled data”. The sampled data may be, but need not be, the storage operand (if any) of the sampled instruction (see Section B.2.5).

The contents of SDAR may be altered by the processor if and only if $MMCR0_{PMAE}$=1. Thus after the Performance Monitor alert occurs, the contents of SDAR are not altered by the processor 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.

See Section B.4 regarding the effects of the Trace facility on SDAR.

Programming Note

If the Performance Monitor alert causes a Performance Monitor interrupt, MMCRA indicates whether the sampled data is the storage operand of the sampled instruction.

B.3 Performance Monitor Interrupt

The Performance Monitor interrupt is a system caused interrupt (Section 6.4). It is masked by MSR_{EE} in the same manner that External and Decrementer interrupts are.

The Performance Monitor interrupt is a basic feature.

A Performance Monitor interrupt occurs when no higher priority exception exists, a Performance Monitor exception exists, and MSR_{EE}=1.

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 the processor would have attempted to execute next if no interrupt conditions were present.

SRR1

33:36 and 42:47 Implementation-specific.

Others Loaded from the MSR.

MSR See Figure 37 on page 466.

SIAR Set to the effective address of the "sampled instruction" (see Section B.2.5).

SDAR Set to the effective address of the "sampled data" (see Section B.2.6).

Execution resumes at effective address 0x0000_0000_0000_0F00.

In general, statements about External and Decrementer interrupts elsewhere in this Book apply also to the Performance Monitor interrupt; for example, if a Performance Monitor exception exists when an ***mtmsrd***[***d***] instruction is executed that changes MSR_{EE} from 0 to 1, the Performance Monitor interrupt will occur before the next instruction is executed (if no higher priority exception exists).

The priority of the Performance Monitor exception is equal to that of the External, Decrementer, and Hypervisor Decrementer exceptions (i.e., the processor may generate any one of the four interrupts for which an exception exists) (see Section 6.7.2, "Ordered Exceptions" on page 478 and Section 6.8, "Interrupt Priorities" on page 479).

B.4 Interaction with the Trace Facility

If the Trace facility includes setting SIAR and SDAR (see Appendix C, "Example Trace Extensions" on page 503), and tracing is active (MSR_{SE}=1 or MSR_{BE}=1), the contents of SIAR and SDAR as used by the Performance Monitor facility are undefined and may change even when $MMCR0_{PMAE}$=0.

Programming Note

A potential combined use of the Trace and Performance Monitor facilities is to trace the control flow of a program and simultaneously count events for that program.

Appendix C. Example Trace Extensions

> **Note**
>
> This Appendix describes an example implementation of Trace Extensions. A subset of these requirements are being considered for inclusion in the Architecture as part of Category: Trace.

This appendix provides an example of extensions that may be added to the Trace facility described in Section 6.5.14, “Trace Interrupt [Category: Trace]” on page 473. It is only an example; implementations may provide all, some, or none of the features described here, or may provide features that are similar to those described here but differ in detail.

The extensions consist of the following features (described in detail below).

- use of $MSR_{SE\ BE}$=0b11 to specify new causes of Trace interrupts
- specification of how certain SRR1 bits are set when a Trace interrupt occurs
- setting of SIAR and SDAR (see Appendix B, “Example Performance Monitor” on page 495) when a Trace interrupt occurs

$MSR_{SE\ BE}$ = 0b11

If $MSR_{SE\ BE}$=0b11, the processor generates a Trace exception under the conditions described in Section 6.5.14 for $MSR_{SE\ BE}$=0b01, and also after successfully completing the execution of any instruction that would cause at least one of SRR1 bits 33:36, 42, and 44:46 to be set to 1 (see below) if the instruction were executed when $MSR_{SE\ BE}$=0b10.

This overrides the implicit statement in Section 6.5.14 that the effects of $MSR_{SE\ BE}$=0b11 are the same as those of $MSR_{SE\ BE}$=0b10.

SRR1

When a Trace interrupt occurs, the SRR1 bits that are not loaded from the MSR are set as follows instead of as described in Section 6.5.14.

33 Set to 1 if the traced instruction is ***icbi***; otherwise set to 0.

34 Set to 1 if the traced instruction is ***dcbt***, ***dcbtst***, ***dcbz***, ***dcbst***, ***dcbf***[***l***]; otherwise set to 0.

35 Set to 1 if the traced instruction is a Load instruction or ***eciwx***; may be set to 1 if the traced instruction is ***icbi***, ***dcbt***, ***dcbtst***, ***dcbst***, ***dcbf***[***l***]; otherwise set to 0.

36 Set to 1 if the traced instruction is a *Store* instruction, ***dcbz***, or ***ecowx***; otherwise set to 0.

42 Set to 1 if the traced instruction is ***lswx*** or ***stswx***; otherwise set to 0.

43 Implementation-dependent.

44 Set to 1 if the traced instruction is a *Branch* instruction and the branch is taken; otherwise set to 0.

45 Set to 1 if the traced instruction is ***eciwx*** or ***ecowx***; otherwise set to 0.

46 Set to 1 if the traced instruction is ***lwarx***, ***ldarx***, ***stwcx.***, or ***stdcx.***; otherwise set to 0.

47 Implementation-dependent.

SIAR and SDAR

If the Performance Monitor facility is implemented and includes SIAR and SDAR (see Appendix B), the following additional registers are set when a Trace interrupt occurs:

SIAR Set to the effective address of the traced instruction.

SDAR Set to the effective address of the storage operand (if any) of the traced instruction; otherwise undefined.

If the state of the Performance Monitor is such that the Performance Monitor may be altering these registers (i.e., if $MMCR0_{PMAE}$=1), the contents of SIAR and SDAR as used by the Trace facility are undefined and may change even when no Trace interrupt occurs.

Appendix D. Interpretation of the DSISR as Set by an Alignment Interrupt

For most causes of Alignment interrupt, the interrupt handler will emulate the interrupting instruction. To do this, it needs the following characteristics of the interrupting instruction:

Load or store
Length (halfword, word, doubleword)
String, multiple, or elementary
Fixed-point or floating-point
Update or non-update
Byte reverse or not
Is it ***dcbz***?

The Power ISA optionally provides this information by setting bits in the DSISR that identify the interrupting instruction type. It is not necessary for the interrupt handler to load the interrupting instruction from storage. The mapping is unique except for a few exceptions that are discussed below. The near-uniqueness depends on the fact that many instructions, such as the fixed- and floating-point arithmetic instructions and the one-byte loads and stores, cannot cause an Alignment interrupt.

See Section 6.5.8 for a description of how the opcode and extended opcode are mapped to a DSISR value for an X-, D-, or DS-form instruction that causes an Alignment interrupt.

The table on the next page shows the inverse mapping: how the DSISR bits identify the interrupting instruction. The following notes are cited in the table.

1. The instructions ***lwz*** and ***lwarx*** give the same DSISR bits (all zero). But if ***lwarx*** causes an Alignment interrupt, it should not be emulated. It is adequate for the Alignment interrupt handler simply to treat the instruction as if it were ***lwz***. The emulator must use the address in the DAR, rather than compute it from RA/RB/D, because ***lwz*** and ***lwarx*** have different instruction formats.

   If opcode 0 (“Illegal or Reserved”) can cause an Alignment interrupt, it will be indistinguishable to the interrupt handler from ***lwarx*** and ***lwz***.

2. These are distinguished by DSISR bits 44:45, which are not shown in the table.

The interrupt handler has no need to distinguish between an X-form instruction and the corresponding D- or DS-form instruction if one exists, and vice versa. Therefore two such instructions may yield the same DSISR value (all 32 bits). For example, ***stw*** and ***stwx*** may both yield either the DSISR value shown in the following table for ***stw***, or that shown for ***stwx***.

If DSISR 47:53 is:	then it is either X-form opcode:	or D/DS-form opcode:	so the instruction is:
00 0 0000	00000xxx00	x00000	lwarx,lwz,reserved(1)
00 0 0001	00010xxx00	x00010	ldarx
00 0 0010	00100xxx00	x00100	stw
00 0 0011	00110xxx00	x00110	-
00 0 0100	01000xxx00	x01000	lhz
00 0 0101	01010xxx00	x01010	lha
00 0 0110	01100xxx00	x01100	sth
00 0 0111	01110xxx00	x01110	lmw
00 0 1000	10000xxx00	x10000	lfs
00 0 1001	10010xxx00	x10010	lfd
00 0 1010	10100xxx00	x10100	stfs
00 0 1011	10110xxx00	x10110	stfd
00 0 1100	11000xxx00	x11000	-
00 0 1101	11010xxx00	x11010	ld, ldu, lwa (2)
00 0 1110	11100xxx00	x11100	-
00 0 1111	11110xxx00	x11110	std, stdu (2)
00 1 0000	00001xxx00	x00001	lwzu
00 1 0001	00011xxx00	x00011	-
00 1 0010	00101xxx00	x00101	stwu
00 1 0011	00111xxx00	x00111	-
00 1 0100	01001xxx00	x01001	lhzu
00 1 0101	01011xxx00	x01011	lhau
00 1 0110	01101xxx00	x01101	sthu
00 1 0111	01111xxx00	x01111	stmw
00 1 1000	10001xxx00	x10001	lfsu
00 1 1001	10011xxx00	x10011	lfdu
00 1 1010	10101xxx00	x10101	stfsu
00 1 1011	10111xxx00	x10111	stfdu
00 1 1100	11001xxx00	x11001	-
00 1 1101	11011xxx00	x11011	-
00 1 1110	11101xxx00	x11101	-
00 1 1111	11111xxx00	x11111	-
01 0 0000	00000xxx01		ldx
01 0 0001	00010xxx01		-
01 0 0010	00100xxx01		stdx
01 0 0011	00110xxx01		-
01 0 0100	01000xxx01		-
01 0 0101	01010xxx01		lwax
01 0 0110	01100xxx01		-
01 0 0111	01110xxx01		-
01 0 1000	10000xxx01		lswx
01 0 1001	10010xxx01		lswi
01 0 1010	10100xxx01		stswx
01 0 1011	10110xxx01		stswi
01 0 1100	11000xxx01		-
01 0 1101	11010xxx01		-
01 0 1110	11100xxx01		-
01 0 1111	11110xxx01		-
01 1 0000	00001xxx01		ldux
01 1 0001	00011xxx01		-
01 1 0010	00101xxx01		stdux
01 1 0011	00111xxx01		-
01 1 0100	01001xxx01		-
01 1 0101	01011xxx01		lwaux
01 1 0110	01101xxx01		-
01 1 0111	01111xxx01		-
01 1 1000	10001xxx01		-
01 1 1001	10011xxx01		-
01 1 1010	10101xxx01		-
01 1 1011	10111xxx01		-
01 1 1100	11001xxx01		-
01 1 1101	11011xxx01		-
01 1 1110	11101xxx01		-
01 1 1111	11111xxx01		-
10 0 0000	00000xxx10		-
10 0 0001	00010xxx10		-
10 0 0010	00100xxx10		stwcx.
10 0 0011	00110xxx10		stdcx.
10 0 0100	01000xxx10		-
10 0 0101	01010xxx10		-
10 0 0110	01100xxx10		-
10 0 0111	01110xxx10		-
10 0 1000	10000xxx10		lwbrx
10 0 1001	10010xxx10		-
10 0 1010	10100xxx10		stwbrx
10 0 1011	10110xxx10		-
10 0 1100	11000xxx10		lhbrx
10 0 1101	11010xxx10		-
10 0 1110	11100xxx10		sthbrx
10 0 1111	11110xxx10		-
10 1 0000	00001xxx10		-
10 1 0001	00011xxx10		-
10 1 0010	00101xxx10		-
10 1 0011	00111xxx10		-
10 1 0100	01001xxx10		eciwx
10 1 0101	01011xxx10		-
10 1 0110	01101xxx10		ecowx
10 1 0111	01111xxx10		-
10 1 1000	10001xxx10		-
10 1 1001	10011xxx10		-
10 1 1010	10101xxx10		-
10 1 1011	10111xxx10		-
10 1 1100	11001xxx10		-
10 1 1101	11011xxx10		-
10 1 1110	11101xxx10		-
10 1 1111	11111xxx10		dcbz
11 0 0000	00000xxx11		lwzx
11 0 0001	00010xxx11		-
11 0 0010	00100xxx11		stwx
11 0 0011	00110xxx11		-
11 0 0100	01000xxx11		lhzx
11 0 0101	01010xxx11		lhax
11 0 0110	01100xxx11		sthx
11 0 0111	01110xxx11		-
11 0 1000	10000xxx11		lfsx
11 0 1001	10010xxx11		lfdx
11 0 1010	10100xxx11		stfsx
11 0 1011	10110xxx11		stfdx
11 0 1100	11000xxx11		-
11 0 1101	11010xxx11		-
11 0 1110	11100xxx11		-
11 0 1111	11110xxx11		stfiwx
11 1 0000	00001xxx11		lwzux
11 1 0001	00011xxx11		-
11 1 0010	00101xxx11		stwux
11 1 0011	00111xxx11		-
11 1 0100	01001xxx11		lhzux
11 1 0101	01011xxx11		lhaux
11 1 0110	01101xxx11		sthux
11 1 0111	01111xxx11		-
11 1 1000	10001xxx11		lfsux
11 1 1001	10011xxx11		lfdux
11 1 1010	10101xxx11		stfsux
11 1 1011	10111xxx11		stfdux
11 1 1100	11001xxx11		-
11 1 1101	11011xxx11		-
11 1 1110	11101xxx11		-
11 1 1111	11111xxx11		-

Book III-E:

Power ISA Operating Environment Architecture - Embedded Environment

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 32-Bit Implementations

Though the specifications in this document assume a 64-bit implementation, 32-bit implementations are permitted as described in Appendix C, “Guidelines for 64-bit Implementations in 32-bit Mode and 32-bit Implementations” on page 637.

1.3 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 auxiliary processor enabled exception error handler” substitute “Auxiliary Processor Enabled Exception type Program interrupt”,
- For “system data storage error handler” substitute “Data Storage interrupt” or Data TLB Error 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 “Illegal Instruction exception type Program interrupt” or “Unimplemented Operation exception type Program interrupt”, as appropriate.
- For “system instruction storage error handler” substitute “Instruction Storage interrupt” or “Instruction TLB Error”, as appropriate.
- For “system privileged instruction error handler” substitute “Privileged Instruction exception type Program interrupt”.
- For “system service program” substitute “System Call interrupt”.
- For “system trap handler” substitute “Trap type Program interrupt”.

1.3.1 Definitions and Notation

The definitions and notation given in Book I are augmented by the following.

- real page

  A unit of real storage that is aligned at a boundary that is a multiple of its size. The real page size may range from 1KB to 1TB.

- context of a program

  The processor 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, of certain lookaside buffers, such as the TLB, and of other resources.

- exception

  An error, unusual condition, or external signal, that may set a status bit and may or may not cause an interrupt, depending upon whether the corresponding interrupt is enabled.

- interrupt

  The act of changing the machine state in response to an exception, as described in Chapter 5. “Interrupts and Exceptions” on page 563.

- trap interrupt

  An interrupt that results from execution of a *Trap* instruction.

- Additional exceptions to the rule that the processor obeys the sequential execution model, beyond those described in the section entitled “Instruction Fetching” in Book I, are the following.
  - A System Reset or Machine Check interrupt may occur. The determination of whether an instruction is required by the sequential execution model is not affected by the potential occurrence of a System Reset or Machine Check interrupt. (The determination is affected by the potential occurrence of any other kind of interrupt.)
  - A context-altering instruction is executed (Chapter 10. “Synchronization Requirements for Context Alterations” on page 625). The context alteration need not take effect until the required subsequent synchronizing operation has occurred.

- 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.

- /, //, ///, ... 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.

1.3.2 Reserved Fields

Some fields of certain architected registers may be written to automatically by the processor, e.g., Reserved bits in System Registers. When the processor writes to such a register, the following rules are obeyed.

- Unless otherwise stated, no defined field other than the one(s) the processor is specifically updating are modified.
- Contents of reserved fields are either preserved by the processor or written as zero.

The reader should be aware that reading and writing of some of these registers (e.g., the MSR) can occur as a side effect of processing an interrupt and of returning from an interrupt, as well as when requested explicitly by the appropriate instruction (e.g., ***mtmsr*** instruction).

1.4 General Systems Overview

The processor or processor unit 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 Processor
- instructions executed in the Fixed-Point Processor
- instructions executed in the Floating-Point Processor
- instructions executed in the Vector Processor
- instructions executed in an Auxiliary Processor
- other instructions executed by the processor

Almost all instructions executed in the Branch Processor, Fixed-Point Processor, Floating-Point Processor, and Vector Processor are nonprivileged and are described in Book I. Book I may describe additional nonprivileged instructions (e.g., Book II describes some nonprivileged instructions for cache management). Instructions executed in an Auxiliary Processor are implementation-dependent. Instructions related to the supervisor mode, control of processor resources, control of the storage hierarchy, and all other privileged instructions are described here or are implementation-dependent.

1.5 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)
- execution of an instruction that causes a debug event (Debug interrupt).
- the execution of an auxiliary processor instruction when the auxiliary processor instruction is unavailable (Auxiliary Processor Unavailable interrupt)
- the execution of a Vector, SPE, or Embedded Floating-Point instruction when MSR_{SPV}=0 (SPE/Embedded Floating-Point/Vector Unavailable interrupt)

1.6 Synchronization

The synchronization described in this section refers to the state of the processor that is performing the synchronization.

1.6.1 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 include the ***isync*** instruction, the *System Linkage* instructions, the ***mtmsr*** instruction, and most interrupts (see Section 5.1).

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 ***dnh*** [Category: Embedded.Enhanced Debug], ***isync*** and ***wait*** [Category: Wait], 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.
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 5.9, “Exception Priorities” on page 591).
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 4.5, “Performing Operations Out-of-Order”.)

Programming Note

A context synchronizing operation is necessarily execution synchronizing; see Section 1.6.2.

Unlike the *Synchronize* instruction, a context synchronizing operation does not affect the order in which storage accesses are performed.

Item 2 permits a choice only for ***isync*** (and ***sync***; see Section 1.6.2) because all other execution synchronizing operations also alter context.

1.6.2 Execution Synchronization

An instruction is *execution synchronizing* if it satisfies items 2 and 3 of the definition of context synchronization (see Section 1.6.1). ***sync*** is treated like ***isync*** with respect to item 2. The execution synchronizing instructions are ***sync***, ***mtmsr*** and all context synchronizing instructions.

Programming Note

All context synchronizing instructions are execution synchronizing.

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. Branch Processor

2.1 Branch Processor Overview

This chapter describes the details concerning the registers and the privileged instructions implemented in the Branch Processor that are not covered in Book I.

2.2 Branch Processor Registers

2.2.1 Machine State Register

The MSR (MSR) is a 32-bit register. MSR bits are numbered 32 (most-significant bit) to 63 (least-significant bit). This register defines the state of the processor. The MSR can also be modified by the ***mtmsr***, ***rfi***, ***rfci***, ***rfdi*** [Category: Embedded.Enhanced Debug], ***rfmci,*** ***wrtee*** and ***wrteei*** instructions and interrupts. It can be read by the ***mfmsr*** instruction.

MSR	
32	63

Figure 1. Machine State Register

Below are shown the bit definitions for the Machine State Register.

Bit **Description**

32 ***Computation Mode*** (CM)

- 0 The processor runs in 32-bit mode.
- 1 The processor runs in 64-bit mode.

33 ***Interrupt Computation Mode*** (ICM)

On interrupt this bit is copied to MSR_{CM}, selecting 32-bit or 64-bit mode for interrupt handling.

- 0 MSR_{CM} is set to 0 (32-bit mode) when an interrupt occurs.
- 1 MSR_{CM} is set to 1 (64-bit mode) when an interrupt occurs.

34:36 Implementation-dependent

37 ***User Cache Locking Enable*** (UCLE)
[Category: Embedded Cache Locking.User Mode]

- 0 *Cache Locking* instructions are privileged.
- 1 *Cache Locking* instructions can be executed in user mode (MSR_{PR}=1).

If category Embedded Cache Locking.User Mode is not supported, this bit is treated as reserved.

38 ***SP/Embedded Floating-Point/Vector Available*** (SPV)

[Category: Signal Processing]:
- 0 The processor cannot execute any SP instructions except for the ***brinc*** instruction.
- 1 The processor can execute all SP instructions.

[Category: Vector]:
- 0 The processor cannot execute any Vector instruction.
- 1 The processor can execute Vector instructions.

39:44 Reserved

45 ***Wait State Enable*** (WE)

- 0 The processor is not in wait state and continues processing
- 1 The processor enters the wait state by ceasing to execute instructions and entering low power mode. The details of how the wait state is entered and exited, and how the processor behaves while in the wait state, are implementation-dependent.

46 ***Critical Enable*** (CE)

0 Critical Input, Watchdog Timer, and Processor Doorbell Critical interrupts are disabled
1 Critical Input, Watchdog Timer, and Processor Doorbell Critical interrupts are enabled

47 Reserved

48 ***External Enable*** (EE)

0 External Input, Decrementer, Fixed-Interval Timer, Processor Doorbell, and Embedded Performance Monitor [Category:E.PM] interrupts are disabled.
1 External Input, Decrementer, Fixed-Interval Timer, Processor Doorbell, and Embedded Performance Monitor [Category:E.PM] interrupts are enabled.

49 ***Problem State*** (PR)

0 The processor is in supervisor mode, can execute any instruction, and can access any resource (e.g. GPRs, SPRs, MSR, etc.).
1 The processor is in user mode, cannot execute any privileged instruction, and cannot access any privileged resource.

MSR_{PR} also affects storage access control, as described in Section 6.2.4.

50 ***Floating-Point Available*** (FP) [Category: Floating-Point]

0 The processor cannot execute any floating-point instructions, including floating-point loads, stores and moves.
1 The processor can execute floating-point instructions.

51 ***Machine Check Enable*** (ME)

0 Machine Check interrupts are disabled.
1 Machine Check interrupts are enabled.

52 ***Floating-Point Exception Mode 0*** (FE0) [Category: Floating-Point]

(See below)

53 Implementation-dependent

54 ***Debug Interrupt Enable*** (DE)

0 Debug interrupts are disabled
1 Debug interrupts are enabled if $DBCR0_{IDM}$=1

55 ***Floating-Point Exception Mode*** 1 (FE1) [Category: Floating-Point]

(See below)

56 Reserved

57 Reserved

58 ***Instruction Address Space*** (IS)

0 The processor directs all instruction fetches to address space 0 (TS=0 in the relevant TLB entry).
1 The processor directs all instruction fetches to address space 1 (TS=1 in the relevant TLB entry).

59 ***Data Address Space*** (DS)

0 The processor directs all data storage accesses to address space 0 (TS=0 in the relevant TLB entry).
1 The processor directs all data storage accesses to address space 1 (TS=1 in the relevant TLB entry).

60 Implementation-dependent

61 ***Performance Monitor Mark*** (PMM) [Category: Embedded.Performance Monitor]

0 Disable statistics gathering on marked processes.
1 Enable statistics gathering on marked processes

See Appendix E for additional information.

62 Reserved

63 Reserved

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

See Section 6.3, “Processor State After Reset” on page 595 for the initial state of the MSR.

Programming Note

A Machine State Register bit that is reserved may be altered by ***rfi***/***rfci/rfmci/rfdi*** [Category:Embedded.Enhanced Debug].

2.3 Branch Processor Instructions

2.4 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

17	///	///	///	///	//	1	/
0	6	11	16	20	27	30	31

```
SRR0 ←iea CIA + 4
SRR1 ← MSR
NIA ← IVPR0:47 || IVOR848:59 || 0b0000
MSR ← new_value (see below)
```

The effective address of the instruction following the *System Call* instruction is placed into SRR0. The contents of the MSR are copied into SRR1.

Then a System Call interrupt is generated. The interrupt causes the MSR to be set as described in Section 5.6 on page 574.

The interrupt causes the next instruction to be fetched from effective address

$IVPR_{0:47}||IVOR8_{48:59}||0b0000$.

This instruction is context synchronizing.

Special Registers Altered:
SRR0 SRR1 MSR

Return From Interrupt — *XL-form*

rfi

19	///	///	///	50	/
0	6	11	16	21	31

```
MSR ← SRR1
NIA ←iea SRR00:61 || 0b00
```

The ***rfi*** instruction is used to return from a base class interrupt, or as a means of simultaneously establishing a new context and synchronizing on that new context.

The contents of SRR1 are placed into 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 $SRR0_{0:61}||0b00$. (Note: VLE behavior may be different; see Book VLE.) 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 the applicable save/restore register 0 by the interrupt processing mechanism (see Section 5.6 on page 574) is the address of the instruction that would have been executed next had the interrupt not occurred (i.e. the address in SRR0 at the time of the execution of the ***rfi***).

This instruction is privileged and context synchronizing.

Special Registers Altered:
MSR

Return From Critical Interrupt XL-form

rfci

19	///	///	///	51	/
0	6	11	16	21	31

```
MSR ← CSRR1
NIA ←iea CSRR0_{0:61} || 0b00
```

The ***rfci*** instruction is used to return from a critical class interrupt, or as a means of establishing a new context and synchronizing on that new context simultaneously.

The contents of CSRR1 are placed into 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 $CSRR0_{0:61}$||0b00. (Note: VLE behavior may be different; see Book VLE.) If the new MSR value enables one or more pending exceptions, the interrupt associated with the highest priority pending exception is generated; in this case the value placed into SRR0 or CSRR0 by the interrupt processing mechanism (see Section 5.6 on page 574) is the address of the instruction that would have been executed next had the interrupt not occurred (i.e. the address in CSRR0 at the time of the execution of the ***rfci***).

This instruction is privileged and context synchronizing.

Special Registers Altered:
MSR

Return From Debug Interrupt X-form

rfdi
[Category: Embedded.Enhanced Debug]

19	///	///	///	39	/
0	6	11	16	21	31

```
MSR ← DSRR1
NIA ←iea DSRR0_{0:61} || 0b00
```

The ***rfdi*** instruction is used to return from a Debug interrupt, or as a means of establishing a new context and synchronizing on that new context simultaneously.

The contents of DSRR1 are placed into 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 $DSRR0_{0:61}$||0b00. (Note: VLE behavior may be different; see Book VLE.) 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, CSRR0, or DSRR0 by the interrupt processing mechanism is the address of the instruction that would have been executed next had the interrupt not occurred (i.e. the address in DSRR0 at the time of the execution of the ***rfdi***).

This instruction is privileged and context synchronizing.

Special Registers Altered:
MSR

Return From Machine Check Interrupt *XL-form*

rfmci

19	///	///	///	38	/
0	6	11	16	21	31

```
MSR ← MCSRR1
NIA ←iea MCSRR0_{0:61} || 0b00
```

The ***rfmci*** instruction is used to return from a Machine Check class interrupt, or as a means of establishing a new context and synchronizing on that new context simultaneously.

The contents of MCSRR1 are placed into 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 $MCSRR0_{0:61}$||0b00. (Note: VLE behavior may be different; see Book VLE.) 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, CSRR0, MCSRR0, or DSRR0 [Category: Embedded.Enhanced Debug] by the interrupt processing mechanism (see Section 5.6 on page 574) is the address of the instruction that would have been executed next had the interrupt not occurred (i.e. the address in MCSRR0 at the time of the execution of the ***rfmci***).

This instruction is privileged and context synchronizing.

Special Registers Altered:

MSR

Chapter 3. Fixed-Point Processor

3.1 Fixed-Point Processor Overview

This chapter describes the details concerning the registers and the privileged instructions implemented in the Fixed-Point Processor that are not covered in Book I.

3.2 Special Purpose Registers

Special Purpose Registers (SPRs) are read and written using the ***mfspr*** (page 526) and ***mtspr*** (page 524) instructions. Most SPRs are defined in other chapters of this book; see the index to locate those definitions.

3.3 Fixed-Point Processor Registers

3.3.1 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 processor. 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.

Version	Revision
32	48 63

Figure 2. Processor Version Register

The PVR distinguishes between processors that differ in attributes that may affect software. It contains two fields.

Version A 16-bit number that identifies the version of the processor. Different version numbers indicate major differences between processors, such as which optional facilities and instructions are supported.

Revision A 16-bit number that distinguishes between implementations of the version. Different revision numbers indicate minor differences between processors having the same version number, such as clock rate and Engineering Change level.

Version numbers are assigned by the Power ISA Architecture process. Revision numbers are assigned by an implementation-defined process.

3.3.2 Processor Identification Register

The Processor Identification Register (PIR) is a 32-bit register that contains a value that can be used to distinguish the processor from other processors 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, if provided, is implementation-dependent.

PROCID	
32	63

Bits	Name	Description
32:63	PROCID	Processor ID

Figure 3. Processor Identification Register

The means by which the PIR is initialized are implementation-dependent.

3.3.3 Software-use SPRs

Software-use SPRs are 64-bit registers provided for use by software.

SPRG0	
SPRG1	
SPRG2	
SPRG3	
SPRG4	
SPRG5	
SPRG6	
SPRG7	
SPRG8	
SPRG9 [Category: Embedded.Enhanced Debug]	
0	63

Figure 4. Special Purpose Registers

Programming Note

USPRG0 was made a 32-bit register and renamed to VRSAVE; see Book I, Section 5.3.3.

SPRG0 through SPRG2

These 64-bit registers can be accessed only in supervisor mode.

SPRG3

This 64-bit register can be read in supervisor mode and can be written only in supervisor mode. It is implementation-dependent whether or not this register can be read in user mode.

SPRG4 through SPRG7

These 64-bit registers can be written only in supervisor mode. These registers can be read in supervisor and user modes.

SPRG8 through SPRG9

These 64-bit registers can be accessed only in supervisor mode.

The contents of SPRGi can be read using ***mfspr*** and written into SPRGi using ***mtspr***.

3.3.4 External Process ID Registers [Category: Embedded.External PID]

The External Process ID Registers provide capabilities for loading and storing General Purpose Registers and performing cache management operations using a supplied context other than the context normally used by the programming model.

Two SPRs describe the context for loading and storing using external contexts. The External Process ID Load Context (EPLC) Register provides the context for External Process ID *Load* instructions, and the External Process ID Store Context (EPSC) Register provides the context for External Process ID *Store* instructions. Each of these registers contains a PR (privilege) bit, an AS (address space) bit, and a Process ID. Changes to the EPLC or the EPSC Register require that a context synchronizing operation be performed prior to using any External Process ID instructions that use these registers.

External Process ID instructions that use the context provided by the EPLC register include ***lbepx***, ***lhepx***, ***lwepx***, ***ldepx***, ***dcbtep***, ***dcbtstep***, ***dcbfep***, ***dcbstep***, ***icbiep***, ***lfdepx***, ***evlddepx***, ***lvepx***, and ***lvepxl*** and those that use the context provided by the EPSC register include ***stbepx***, ***sthepx***, ***stwepx***, ***stdepx***, ***dcbzep***, ***stfdepx***, ***evstddepx***, ***stvepx***, and ***stvepxl***. Instruction definitions appear in Section 3.4.2.

System software configures the EPLC register to reflect the Process ID, AS, and PR state from the context that it wishes to perform loads from and configures the EPSC register to reflect the Process ID, AS, and PR state from the context it wishes to perform stores to. Software then issues External Process ID instructions to manipulate data as required.

When the processor executes an External Process ID *Load* instruction, it uses the context information in the EPLC Register instead of the normal context with respect to address translation and storage access control. $EPLC_{EPR}$ is used in place of MSR_{PR}, $EPLC_{EAS}$ is used in place of MSR_{DS}, and $EPLC_{EPID}$ is used in place of any Process ID registers implemented by the processor. Similarly, when the processor executes an External Process ID *Store* instruction, it uses the context information in the EPSC Register instead of the normal context with respect to address translation and storage access control. $EPSC_{EPR}$ is used in place of MSR_{PR}, $EPSC_{EAS}$ is used in place of MSR_{DS}, and $EPSC_{EPID}$ is used in place of all Process ID registers implemented by the processor. Translation occurs using the new substituted values.

If the TLB lookup is successful, the storage access control mechanism grants or denies the access using context information from $EPLC_{EPR}$ or $EPSC_{EPR}$ for loads and stores respectively. If access is not granted, a Data Storage interrupt occurs, and the ESR_{EPID} bit is set to 1. If the operation was a *Store*, the ESR_{ST} bit is also set to 1.

3.3.4.1 External Process ID Load Context (EPLC) Register

The EPLC register contains fields to provide the context for External Process ID load instructions.

EPLC	
32	63

Figure 5. External Process ID Load Context Register

These bits are interpreted as follows:

Bit	Definition
0	***External Load Context PR Bit*** (EPR) Used in place of MSR_{PR} by the storage access control mechanism when an External Process ID *Load* instruction is executed. 0 Supervisor mode 1 User mode
1	***External Load Context AS Bit*** (EAS) Used in place of MSR_{DS} for translation when an External Process ID *Load* instruction is executed. 0 Address space 0 1 Address space 1
2:17	Reserved
18:31	***External Load Context Process ID Value*** (EPID) Used in place of all Process ID register values for translation when an external Process ID *Load* instruction is executed.

3.3.4.2 External Process ID Store Context (EPSC) Register

The EPSC register contains fields to provide the context for External Process ID Store instructions. The field encoding is the same as the EPLC Register.

EPSC	
32	63

Figure 6. External Process ID Store Context Register

These bits are interpreted as follows:

Bits Definition

0 ***External Store Context PR Bit*** (EPR)
Used in place of MSR_{PR} by the storage access control mechanism when an External Process ID *Store* instruction is executed.

- 0 Supervisor mode
- 1 User mode

1 ***External Store Context AS Bit*** (EAS)
Used in place of MSR_{DS} for translation when an External Process ID *Store* instruction is executed.

- 0 Address space 0
- 1 Address space 1

2:17 Reserved

18:31 ***External Store Context Process ID Value*** (EPID)
Used in place of all Process ID register values for translation when an external PID *Store* instruction is executed.

3.4 Fixed-Point Processor Instructions

3.4.1 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 supervisor mode. 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 below, the “defined” SPR numbers are the SPR numbers shown in Figure 7 and the implementation-specific SPR numbers that are implemented, and similarly for “defined” registers.

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; see Appendix B.

SPR Numbers

decimal	SPR[1] $spr_{5:9}$ $spr_{0:4}$	Register Name	Privileged		Length (bits)	Cat[2]
			mtspr	mfspr		
1	00000 00001	XER	no	no	64	B
8	00000 01000	LR	no	no	64	B
9	00000 01001	CTR	no	no	64	B
22	00000 10110	DEC	yes	yes	32	B
26	00000 11010	SRR0	yes	yes	64	B
27	00000 11011	SRR1	yes	yes	64	B
48	00001 10000	PID	yes	yes	32	E
54	00001 10110	DECAR	yes	yes	32	E
58	00001 11010	CSRR0	yes	yes	64	E
59	00001 11011	CSRR1	yes	yes	32	E
61	00001 11101	DEAR	yes	yes	64	E
62	00001 11110	ESR	yes	yes	32	E
63	00001 11111	IVPR	yes	yes	64	E
256	01000 00000	VRSAVE	no	no	32	E,V
259	01000 00011	SPRG3	-	no	64	B
260-263	01000 001xx	SPRG[4-7]	-	no	64	E
268	01000 01100	TB	-	no	64	B
269	01000 01101	TBU	-	no	32[5]	B
272-275	01000 100xx	SPRG[0-3]	yes	yes	64	B
276-279	01000 101xx	SPRG[4-7]	yes	yes	64	E
282	01000 11010	EAR	yes	yes	32	EC
284	01000 11100	TBL	yes	-	32	B
285	01000 11101	TBU	yes	-	32	B
286	01000 11110	PIR	-	yes	32	E
287	01000 11111	PVR	-	yes	32	B
304	01001 10000	DBSR	yes[3]	yes	32	E
308	01001 10100	DBCR0	yes	yes	32	E
309	01001 10101	DBCR1	yes	yes	32	E
310	01001 10110	DBCR2	yes	yes	32	E
312	01001 11000	IAC1	yes	yes	64	E
313	01001 11001	IAC2	yes	yes	64	E
314	01001 11010	IAC3	yes	yes	64	E
315	01001 11011	IAC4	yes	yes	64	E
316	01001 11100	DAC1	yes	yes	64	E
317	01001 11101	DAC2	yes	yes	64	E
318	01001 11110	DVC1	yes	yes	64	E
319	01001 11111	DVC2	yes	yes	64	E
336	01010 10000	TSR	yes[3]	yes	32	E

decimal	SPR[1] spr$_{5:9}$ spr$_{0:4}$	Register Name	Privileged mtspr	Privileged mfspr	Length (bits)	Cat[2]
340	01010 10100	TCR	yes	yes	32	E
400-415	01100 1xxxx	IVOR[0-15]	yes	yes	32	E
512	10000 00000	SPEFSCR	no	no	32	SPE
526	10000 01110	ATB/ATBL	-	no	64	ATB
527	10000 01111	ATBU	-	no	32	ATB
528	10000 10000	IVOR32	yes	yes	32	SPE
529	10000 10001	IVOR33	yes	yes	32	SPE
530	10000 10010	IVOR34	yes	yes	32	SPE
531	10000 10011	IVOR35	yes	yes	32	E.PM
532	10000 10100	IVOR36	yes	yes	32	E.PC
533	10000 10101	IVOR37	yes	yes	32	E.PC
570	10001 11010	MCSRR0	yes	yes	64	E
571	10001 11011	MCSRR1	yes	yes	32	E
572	10001 11100	MCSR	yes	yes	64	E
574	10001 11110	DSRR0	yes	yes	64	E.ED
575	10001 11111	DSRR1	yes	yes	32	E.ED
604	10010 11100	SPRG8	yes	yes	64	E
605	10010 11101	SPRG9	yes	yes	64	E.ED
624	10011 10000	MAS0	yes	yes	32	E.MF
625	10011 10001	MAS1	yes	yes	32	E.MF
626	10011 10010	MAS2	yes	yes	64	E.MF
627	10011 10011	MAS3	yes	yes	32	E.MF
628	10011 10100	MAS4	yes	yes	32	E.MF
630	10011 10110	MAS6	yes	yes	32	E.MF
633	10011 11001	PID1	yes	yes	32	E.MF
634	10011 11010	PID2	yes	yes	32	E.MF
688-691	10101 100xx	TLB[0-3]CFG	yes	yes	32	E.MF
702	10101 11110	EPR	-	yes	32	EXP
924	11100 11100	DCBTRL	-[4]	yes	32	E.CD
925	11100 11101	DCBTRH	-[4]	yes	32	E.CD
926	11100 11110	ICBTRL	-[5]	yes	32	E.CD
927	11100 11111	ICDBTRH	-[5]	yes	32	E.CD
944	11101 10000	MAS7	yes	yes	32	E.MF
947	11101 10011	EPLC	yes	yes	32	E.PD
948	11101 10100	EPSC	yes	yes	32	E.PD
979	11110 10011	ICBDR	-[5]	yes	32	E.CD
1012	11111 10100	MMUCSR0	yes	yes	32	E.MF
1015	11111 10111	MMUCFG	yes	yes	32	E.MF

\- 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] See Section 1.3.5 of Book I.
[3] This register cannot be directly written to. Instead, bits in the register corresponding to 1 bits in (RS) can be cleared using *mtspr SPR,RS*.
[4] The register can be written by the ***dcread*** instruction.
[5] The register can be written by the ***icread*** instruction.

All SPR numbers that are not shown above and are not implementation-specific are reserved.

Figure 7. Embedded SPR List

Move To Special Purpose Register XFX-form

mtspr SPR,RS

31	RS	spr	467	/
0	6	11	21	31

```
n ← spr5:9 || spr0:4
if length(SPR(n)) = 64 then
  SPR(n) ← (RS)
else
  SPR(n) ← (RS)32:63
```

The SPR field denotes a Special Purpose Register, encoded as shown in Figure 7. The contents of register RS are placed into the designated Special Purpose Register. For Special Purpose Registers that are 32 bits long, the low-order 32 bits of RS are placed into the SPR.

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 a defined and privileged register when MSR_{PR}=1 causes a Privileged Instruction type Program interrupt.

Execution of this instruction specifying an SPR number that is not defined for the implementation causes either an Illegal Instruction type Program interrupt or one of the following.

- if spr_0=0: boundedly undefined results
- if spr_0=1:
  - if MSR_{PR}=1: Privileged Instruction type Program interrupt; if MSR_{PR}=0: boundedly undefined results

If the SPR number is set to a value that is shown in Figure 7 but corresponds to an optional Special Purpose Register that is not provided by the implementation, the effect of executing this instruction is the same as if the SPR number were reserved.

Special Registers Altered:

See Figure 7

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.

Programming Note

For a discussion of software synchronization requirements when altering certain Special Purpose Registers, see Chapter 10. “Synchronization Requirements for Context Alterations” on page 625.

Move From Special Purpose Register XFX-form

mfspr RT,SPR

31	RT	spr	339	/
0	6	11	21	31

```
n ← spr5:9 || spr0:4
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 Figure 7. 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.

spr_0=1 if and only if reading the register is privileged. Execution of this instruction specifying a defined and privileged register when MSR_{PR}=1 causes a Privileged Instruction type Program interrupt.

Execution of this instruction specifying an SPR number that is not defined for the implementation causes either an Illegal Instruction type Program interrupt or one of the following.

- if spr_0=0: boundedly undefined results
- if spr_0=1:
  - if MSR_{PR}=1: Privileged Instruction type Program interrupt
  - if MSR_{PR}=0: boundedly undefined results

If the SPR field contains a value that is shown in Figure 7 but corresponds to an optional Special Purpose Register that is not provided by the implementation, the effect of executing this instruction is the same as if the SPR number were reserved.

Special Registers Altered:
None

Note
See the Notes that appear with ***mtspr***.

Move To Device Control Register XFX-form

mtdcr DCRN,RS

31	RS	dcr	451	/
0	6	11	21	31

```
DCRN ← dcr0:4 || dcr5:9
DCR(DCRN) ← (RS)
```

Let DCRN denote a Device Control Register. (The supported Device Control Registers are implementation-dependent.)

The contents of register RS are placed into the designated Device Control Register. For 32-bit Device Control Registers, the contents of bits 32:63 of (RS) are placed into the Device Control Register.

This instruction is privileged.

Special Registers Altered:
Implementation-dependent.

Move To Device Control Register Indexed X-form

mtdcrx RA,RS

31	RS	RA	///	387	/
0	6	11	16	21	31

```
DCRN ← (RA)
DCR(DCRN) ← (RS)
```

Let the contents of register RA denote a Device Control Register. (The supported Device Control Registers supported are implementation-dependent.)

The contents of register RS are placed into the designated Device Control Register. For 32-bit Device Control Registers, the contents of $RS_{32:63}$ are placed into the Device Control Register.

The specification of Device Control Registers using ***mtdcrx***, ***mtdcrux*** (see Book I), and ***mtdcr*** is implementation-dependent. For example, *mtdcr 105,r2* and *mtdcrux r1,r2* (where register r1 contains the value 105) may not produce identical results on an implementation.

This instruction is privileged.

Special Registers Altered:
Implementation-dependent.

Move From Device Control Register XFX-form

mfdcr RT,DCRN

31	RT	dcr	323	/
0	6	11	21	31

```
DCRN ← dcr_{0:4} || dcr_{5:9}
RT ← DCR(DCRN)
```

Let DCRN denote a Device Control Register. (The supported Device Control Registers are implementation-dependent.)

The contents of the designated Device Control Register are placed into register RT. For 32-bit Device Control Registers, the contents of the Device Control Register are placed into bits 32:63 of RT. Bits 0:31 of RT are set to 0.

This instruction is privileged.

Special Registers Altered:
Implementation-dependent.

Move From Device Control Register Indexed X-form

mfdcrx RT,RA

31	RT	RA	///	259	/
0	6	11	16	21	31

```
DCRN ← (RA)
RT ← DCR(DCRN)
```

Let the contents of register RA denote a Device Control Register (the supported Device Control Registers are implementation-dependent.)

The contents of the designated Device Control Register are placed into register RT. For 32-bit Device Control Registers, the contents of bits 32:63 of the designated Device Control Register are placed into RT. Bits 0:31 of RT are set to 0.

The specification of Device Control Registers using ***mfdcrx*** and ***mfdcrux*** (see Book I) compared to the specification of Device Control Registers using ***mfdcr*** is implementation-dependent. For example, *mfdcr r2,105* and *mfdcrx r2,r1* (where register r1 contains the value 105) may not produce identical results on an implementation or between implementations. Also, accessing privileged Device Control Registers with ***mfdcrux*** when the processor is in supervisor mode is implementation-dependent.

This instruction is privileged.

Special Registers Altered:
Implementation-dependent.

Move To Machine State Register X-form

mtmsr RS

31	RS	///	///	146	/
0	6	11	16	21	31

```
newmsr ← (RS)_{32:63}
if MSR_{CM} = 0 & newmsr_{CM} = 1 then NIA_{0:31} ← 0
MSR ← newmsr
```

The contents of register $RS_{32:63}$ are placed into the MSR. If the processor is changing from 32-bit mode to 64-bit mode, the next instruction is fetched from $^{32}0||NIA_{32:63}$.

This instruction is privileged and execution synchronizing.

In addition, alterations to the EE or CE bits are effective as soon as the instruction completes. Thus if MSR_{EE}=0 and an External interrupt is pending, executing an ***mtmsr*** that sets MSR_{EE} to 1 will cause the External interrupt to be taken before the next instruction is executed, if no higher priority exception exists. Likewise, if MSR_{CE}=0 and a Critical Input interrupt is pending, executing an ***mtmsr*** that sets MSR_{CE} to 1 will cause the Critical Input interrupt to be taken before the next instruction is executed if no higher priority exception exists. (See Section 5.6 on page 574).

Special Registers Altered:
MSR

Programming Note

For a discussion of software synchronization requirements when altering certain MSR bits please refer to Chapter 10.

Move From Machine State Register X-form

mfmsr RT

31	RT	///	///	83	/
0	6	11	16	21	31

```
RT ← ³²0 || MSR
```

The contents of the MSR are placed into bits 32:63 of register RT and bits 0:31 of RT are set to 0.

This instruction is privileged.

Special Registers Altered:
None

Write MSR External Enable X-form

wrtee RS

31	RS	///	///	131	/
0	6	11	16	21	31

$MSR_{EE} \leftarrow (RS)_{48}$

The content of $(RS)_{48}$ is placed into MSR_{EE}.

Alteration of the MSR_{EE} bit is effective as soon as the instruction completes. Thus if MSR_{EE}=0 and an External interrupt is pending, executing a wrtee instruction that sets MSR_{EE} to 1 will cause the External interrupt to occur before the next instruction is executed, if no higher priority exception exists (Section 5.9, “Exception Priorities” on page 591).

This instruction is privileged.

Special Registers Altered:
MSR

Write MSR External Enable Immediate X-form

wrteei

31	///	///	E	///	163	/
0	6	11	16	17	21	31

$MSR_{EE} \leftarrow E$

The value specified in the E field is placed into MSR_{EE}.

Alteration of the MSR_{EE} bit is effective as soon as the instruction completes. Thus if MSR_{EE}=0 and an External interrupt is pending, executing a wrtee instruction that sets MSR_{EE} to 1 will cause the External interrupt to occur before the next instruction is executed, if no higher priority exception exists (Section 5.9, “Exception Priorities” on page 591).

This instruction is privileged.

Special Registers Altered:
MSR

Programming Note

wrtee and ***wrteei*** are used to provide atomic update of MSR_{EE}. Typical usage is:

```
mfmsr    Rn          #save EE in (Rn)48
wrteei   0           #turn off EE
mfmsr    Rn          #save EE in (Rn)48
wrteei   0           #turn off EE
  :      :           :
  :                  #code with EE disabled
wrtee    Rn   #restore EE without altering
              #other MSR bits that might
              #have changed
```

3.4.2 External Process ID Instructions [Category: Embedded.External PID]

External Process ID instructions provide capabilities for loading and storing General Purpose Registers and performing cache management operations using a supplied context other than the context normally used by translation.

The EPLC and EPSC registers provide external contexts for performing loads and stores. The EPLC and the EPSC registers are described in Section 3.3.4.

If an Alignment interrupt, Data Storage interrupt, or a Data TLB Error interrupt, occurs while attempting to execute an External Process ID instruction, ESR_{EPID} is set to 1 indicating that the instruction causing the interrupt was an External Process ID instruction; any other applicable ESR bits are also set.

Load Byte by External Process ID Indexed X-form

lbepx RT,RA,RB

31	RT	RA	RB	95	/
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.

For ***lbepx***, the normal translation mechanism is not used. The contents of the EPLC register are used to provide the context in which translation occurs. The following substitutions are made for just the translation and access control process:

- $EPLC_{EPR}$ is used in place of MSR_{PR}
- $EPLC_{EAS}$ is used in place of MSR_{DS}
- $EPLC_{EPID}$ is used in place of all Process ID registers.

This instruction is privileged.

Special Registers Altered:

None

Programming Note

This instruction behaves identically to a ***lbzx*** instruction except for using the EPLC register to provide the translation context.

Load Halfword by External Process ID Indexed X-form

lhepx RT,RA,RB

31	RT	RA	RB	287	/
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.

For ***lhepx***, the normal translation mechanism is not used. The contents of the EPLC register are used to provide the context in which translation occurs. The following substitutions are made for just the translation and access control process:

- $EPLC_{EPR}$ is used in place of MSR_{PR}
- $EPLC_{EAS}$ is used in place of MSR_{DS}
- $EPLC_{EPID}$ is used in place of all Process ID registers.

This instruction is privileged.

Special Registers Altered:

None

Programming Note

This instruction behaves identically to a ***lhzx*** instruction except for using the EPLC register to provide the translation context.

Load Word by External Process ID Indexed *X-form*

lwepx RT,RA,RB

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

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + (RB)
RT ← 320 || MEM(EA,4)
```

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

For ***lwepx***, the normal translation mechanism is not used. The contents of the EPLC register are used to provide the context in which translation occurs. The following substitutions are made for just the translation and access control process:

$EPLC_{EPR}$ is used in place of MSR_{PR}
$EPLC_{EAS}$ is used in place of MSR_{DS}
$EPLC_{EPID}$ is used in place of all Process ID registers.

This instruction is privileged.

Special Registers Altered:
None

Programming Note

This instruction behaves identically to a ***lwzx*** instruction except for using the EPLC register to provide the translation context.

Load Doubleword by External Process ID Indexed *X-form*

ldepx RT,RA,RB

31	RT	RA	RB	29	/
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.

For ***ldepx***, the normal translation mechanism is not used. The contents of the EPLC register are used to provide the context in which translation occurs. The following substitutions are made for just the translation and access control process:

$EPLC_{EPR}$ is used in place of MSR_{PR}
$EPLC_{EAS}$ is used in place of MSR_{DS}
$EPLC_{EPID}$ is used in place of all Process ID registers.

This instruction is privileged.

Corequisite Categories:
64-Bit

Special Registers Altered:
None

Programming Note

This instruction behaves identically to a ***ldx*** instruction except for using the EPLC register to provide the translation context.

Store Byte by External Process ID Indexed X-form

stbepx RS,RA,RB

31	RS	RA	RB	223	/
0	6	11	16	21	31

```
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.

For ***stbepx***, the normal translation mechanism is not used. The contents of the EPSC register are used to provide the context in which translation occurs. The following substitutions are made for just the translation and access control process:

$EPSC_{EPR}$ is used in place of MSR_{PR}
$EPSC_{EAS}$ is used in place of MSR_{DS}
$EPSC_{EPID}$ is used in place of all Process ID registers.

This instruction is privileged.

Special Registers Altered:
None

Programming Note

This instruction behaves identically to a ***stbx*** instruction except for using the EPSC register to provide the translation context.

Store Halfword by External Process ID Indexed X-form

sthepx RS,RA,RB

31	RS	RA	RB	415	/
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.

For ***sthepx***, the normal translation mechanism is not used. The contents of the EPSC register are used to provide the context in which translation occurs. The following substitutions are made for just the translation and access control process:

$EPSC_{EPR}$ is used in place of MSR_{PR}
$EPSC_{EAS}$ is used in place of MSR_{DS}
$EPSC_{EPID}$ is used in place of all Process ID registers.

This instruction is privileged.

Special Registers Altered:
None

Programming Note

This instruction behaves identically to a ***sthx*** instruction except for using the EPSC register to provide the translation context.

Store Word by External Process ID Indexed *X-form*

stwepx RS,RA,RB

31	RS	RA	RB	159	/
0	6	11	16	21	31

```
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.

For ***stwepx***, the normal translation mechanism is not used. The contents of the EPSC register are used to provide the context in which translation occurs. The following substitutions are made for just the translation and access control process:

$EPSC_{EPR}$ is used in place of MSR_{PR}
$EPSC_{EAS}$ is used in place of MSR_{DS}
$EPSC_{EPID}$ is used in place of all Process ID registers.

This instruction is privileged.

Special Registers Altered:
None

Programming Note

This instruction behaves identically to a ***stwx*** instruction except for using the EPSC register to provide the translation context.

Store Doubleword by External Process ID Indexed *X-form*

stdepx RS,RA,RB

31	RS	RA	RB	157	/
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.

For ***stdepx***, the normal translation mechanism is not used. The contents of the EPSC register are used to provide the context in which translation occurs. The following substitutions are made for just the translation and access control process:

$EPSC_{EPR}$ is used in place of MSR_{PR}
$EPSC_{EAS}$ is used in place of MSR_{DS}
$EPSC_{EPID}$ is used in place of all Process ID registers.

This instruction is privileged.

Corequisite Categories:
64-Bit

Special Registers Altered:
None

Programming Note

This instruction behaves identically to a ***stdx*** instruction except for using the EPSC register to provide the translation context.

Data Cache Block Store by External PID X-form

dcbstep RA,RB

31	///	RA	RB	63	/
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, 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, then those locations are written to main storage. Additional locations in the block may be written to main storage. The block ceases to be considered 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 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.

The instruction is treated as a *Load* with respect to translation, memory protection, and is treated as a *Write* with respect to debug events.

This instruction is privileged.

For ***dcbstep***, the normal translation mechanism is not used. The contents of the EPLC register are used to provide the context in which translation occurs. The following substitutions are made for just the translation and access control process:

$EPLC_{EPR}$ is used in place of MSR_{PR}
$EPLC_{EAS}$ is used in place of MSR_{DS}
$EPLC_{EPID}$ is used in place of all Process ID registers

Special Registers Altered:

None

Programming Note

This instruction behaves identically to a ***dcbst*** instruction except for using the EPLC register to provide the translation context.

Data Cache Block Touch by External PID X-form

dcbtep TH,RA,RB

31	/	TH	RA	RB	319	/
0	6	7	11	16	21	31

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

The ***dcbtep*** instruction provides a hint that describes a block or data stream, or indicates the expected use thereof. 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 ***dcbtep*** 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 ***dcbtep*** instruction (e.g., ***dcbtep*** is considered not to cause any data accesses). No means are provided by which software can synchronize these actions with the execution of the instruction stream. For example, these actions are not ordered by the memory barrier created by a ***sync*** instruction.

The ***dcbtep*** 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 in the ***dcbt*** instruction in Section 3.2.2 of Book II.

The instruction is treated as a *Load*, except that no interrupt occurs if a protection violation occurs.

The instruction is privileged.

The normal address translation mechanism is not used. The contents of the EPLC register are used to provide the context in which translation occurs. The following substitutions are made for just the translation and access control process:

$EPLC_{EPR}$ is used in place of MSR_{PR}
$EPLC_{EAS}$ is used in place of MSR_{DS}
$EPLC_{EPID}$ is used in place of all Process ID registers.

Special Registers Altered:

None

Extended Mnemonics:

Extended mnemonics are provided for the *Data Cache Block Touch by External PID* instruction so that it can be coded with the TH value as the last operand for all categories. .

Extended:	**Equivalent to:**
dcbtctep RA,RB,TH	dcbtep for TH values of 0b0000 - 0b0111; other TH values are invalid.

Extended:	Equivalent to:
dcbtdsep RA,RB,TH	dcbtep for TH values of 0b0000 or 0b1000 - 0b1010; other TH values are invalid.

Programming Note

This instruction behaves identically to a ***dcbt*** instruction except for using the EPLC register to provide the translation context.

Data Cache Block Flush by External PID X-form

dcbfep RA,RB

31	///	RA	RB	127	/
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, 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, then those locations are written to main storage. Additional locations in the block may also be written to main storage. The block is invalidated in the data cache of all processors.

If the block containing the byte addressed by EA is in storage that is not Memory Coherence Required, a block containing the byte addressed by EA is in the data cache of this processor, and any locations in the block are considered to be modified there, then those locations are written to main storage. Additional locations in the block may also be written to main storage. The block is invalidated in the data cache of this processor.

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.

The instruction is treated as a *Load* with respect to translation, memory protection, and is treated as a *Write* with respect to debug events.

This instruction is privileged.

The normal translation mechanism is not used. The contents of the EPLC register are used to provide the context in which translation occurs. The following substitutions are made for just the translation and access control process:

$EPLC_{EPR}$ is used in place of MSR_{PR}
$EPLC_{EAS}$ is used in place of MSR_{DS}
$EPLC_{EPID}$ is used in place of all Process ID registers

Special Registers Altered:

None

Programming Note

This instruction behaves identically to a ***dcbf*** instruction except for using the EPLC register to provide the translation context.

Data Cache Block Touch for Store by External PID *X-form*

dcbtstep TH,RA,RB

31	/	TH	RA	RB	255	/
0	6	7	11	16	21	31

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

The ***dcbtstep*** instruction provides a hint that the program will probably soon store to the block containing the byte addressed by EA. If the Cache Specification category is supported, the nature of the hint depends on the value of the TH field, as specified in Section 3.2.2 of Book II. If the Cache Specification category is not supported, the TH field is treated as a reserved field.

If the block is in a storage location that is Caching Inhibited or Guarded, then the hint is ignored.

The only operation that is "caused" by the ***dcbtstep*** 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 ***dcbtstep*** instruction (e.g., ***dcbtstep*** is considered not to cause any data accesses). No means are provided by which software can synchronize these actions with the execution of the instruction stream. For example, these actions are not ordered by the memory barrier created by a ***sync*** instruction.

The ***dcbtstep*** instruction may complete before the operation it causes has been performed.

The instruction is treated as a *Load*, except that no interrupt occurs if a protection violation occurs.

The instruction is privileged.

The normal address translation mechanism is not used. The contents of the EPLC register are used to provide the context in which translation occurs. The following substitutions are made for just the translation and access control process:

$EPLC_{EPR}$ is used in place of MSR_{PR}
$EPLC_{EAS}$ is used in place of MSR_{DS}
$EPLC_{EPID}$ is used in place of all Process ID registers.

Special Registers Altered:

None

Extended Mnemonics:

Extended mnemonics are provided for the *Data Cache Block Touch for Store by External PID* instruction so that it can be coded with the TH value as the last operand for all categories. .

Extended:	**Equivalent to:**
dcbtstctep RA,RB,TH	dcbtstep for TH values of 0b0000 - 0b0111; other TH values are invalid.

Programming Note

This instruction behaves identically to a ***dcbtst*** instruction except for using the EPLC register to provide the translation context.

Instruction Cache Block Invalidate by External PID *X-form*

icbiep RA,RB

31	///	RA	RB	991	/
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 processor, 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 a block containing the byte addressed by EA 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.

The instruction is treated as a *Load*.

This instruction is privileged.

For ***icbiep***, the normal translation mechanism is not used. The contents of the EPLC register are used to provide the context in which translation occurs. The following substitutions are made for just the translation and access control process:

$EPLC_{EPR}$ is used in place of MSR_{PR}
$EPLC_{EAS}$ is used in place of MSR_{DS}
$EPLC_{EPID}$ is used in place of all Process ID registers

Special Registers Altered:

None

Programming Note

This instruction behaves identically to an ***icbi*** instruction except for using the EPLC register to provide the translation context.

Data Cache Block set to Zero by External PID *X-form*

dcbzep RA,RB

31	///	RA	RB	1023	/
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*.

This instruction is privileged.

The normal translation mechanism is not used. The contents of the EPSC register are used to provide the context in which translation occurs. The following substitutions are made for just the translation and access control process:

$EPSC_{EPR}$ is used in place of MSR_{PR}
$EPSC_{EAS}$ is used in place of MSR_{DS}
$EPSC_{EPID}$ is used in place of all Process ID registers

Special Registers Altered:

None

Programming Note

See the Programming Notes for the ***dcbz*** instruction.

Programming Note

This instruction behaves identically to a ***dcbz*** instruction except for using the EPSC register to provide the translation context.

Load Floating-Point Double by External Process ID Indexed X-form

lfdepx FRT,RA,RB

31	FRT	RA	RB	607	/
0	6	11	16	21	31

```
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 FRT.

For ***lfdepx***, the normal translation mechanism is not used. The contents of the EPLC register are used to provide the context in which translation occurs. The following substitutions are made for just the translation and access control process:

$EPLC_{EPR}$ is used in place of MSR_{PR}
$EPLC_{EAS}$ is used in place of MSR_{DS}
$EPLC_{EPID}$ is used in place of all Process ID registers

This instruction is privileged.

An attempt to execute ***lfdepx*** while MSR_{FP}=0 will cause a Floating-Point Unavailable interrupt.

Corequisite Categories:
Floating-Point

Special Registers Altered:
None

Programming Note

This instruction behaves identically to a ***lfdx*** instruction except for using the EPLC register to provide the translation context.

Store Floating-Point Double by External Process ID Indexed X-form

stfdepx FRS,RA,RB

31	FRS	RA	RB	735	/
0	6	11	16	21	31

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

For ***stfdepx***, the normal translation mechanism is not used. The contents of the EPSC register are used to provide the context in which translation occurs. The following substitutions are made for just the translation and access control process:

$EPSC_{EPR}$ is used in place of MSR_{PR}
$EPSC_{EAS}$ is used in place of MSR_{DS}
$EPSC_{EPID}$ is used in place of all Process ID registers

This instruction is privileged.

An attempt to execute ***stfdepx*** while MSR_{FP}=0 will cause a Floating-Point Unavailable interrupt.

Corequisite Categories:
Floating-Point

Special Registers Altered:
None

Programming Note

This instruction behaves identically to a ***stfdx*** instruction except for using the EPSC register to provide the translation context.

Vector Load Doubleword into Doubleword by External Process ID Indexed EVX-form

evlddepx RT,RA,RB

31	RT	RA	RB	285
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.

For ***evlddepx***, the normal translation mechanism is not used. The contents of the EPLC register are used to provide the context in which translation occurs. The following substitutions are made for just the translation and access control process:

$EPLC_{EPR}$ is used in place of MSR_{PR}
$EPLC_{EAS}$ is used in place of MSR_{DS}
$EPLC_{EPID}$ is used in place of all Process ID registers

This instruction is privileged.

An attempt to execute ***evlddepx*** while MSR_{SPV}=0 will cause an SPE Unavailable interrupt.

Corequisite Categories:
Signal Processing Engine

Special Registers Altered:
None

Programming Note

This instruction behaves identically to a ***evlddx*** instruction except for using the EPLC register to provide the translation context.

Vector Store Doubleword into Doubleword by External Process ID Indexed EVX-form

evstddepx RS,RA,RB

31	RS	RA	RB	413
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.

For ***evstddepx***, the normal translation mechanism is not used. The contents of the EPSC register are used to provide the context in which translation occurs. The following substitutions are made for just the translation and access control process:

$EPSC_{EPR}$ is used in place of MSR_{PR}
$EPSC_{EAS}$ is used in place of MSR_{DS}
$EPSC_{EPID}$ is used in place of all Process ID registers

This instruction is privileged.

An attempt to execute ***evstddepx*** while MSR_{SPV}=0 will cause an SPE Unavailable interrupt.

Corequisite Categories:
Signal Processing Engine

Special Registers Altered:
None

Programming Note

This instruction behaves identically to a ***evstddx*** instruction except for using the EPSC register to provide the translation context.

Load Vector by External Process ID Indexed *X-form*

lvepx VRT,RA,RB

31	VRT	RA	RB	295	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + (RB)
VRT ← MEM(EA & 0xFFFF_FFFF_FFFF_FFF0, 16)
```

Let the effective address (EA) be the sum (RA|0)+(RB). The quadword in storage addressed by the result of EA ANDed with 0xFFFF_FFFF_FFFF_FFF0 is loaded into VRT.

For ***lvepx***, the normal translation mechanism is not used. The contents of the EPLC register are used to provide the context in which translation occurs. The following substitutions are made for just the translation and access control process:

$EPLC_{EPR}$ is used in place of MSR_{PR}
$EPLC_{EAS}$ is used in place of MSR_{DS}
$EPLC_{EPID}$ is used in place of all Process ID registers

This instruction is privileged.

An attempt to execute ***lvepx*** while MSR_{SPV}=0 will cause a Vector Unavailable interrupt.

Corequisite Categories:
Vector

Special Registers Altered:
None

Programming Note

This instruction behaves identically to a ***lvx*** instruction except for using the EPLC register to provide the translation context.

Load Vector by External Process ID Indexed LRU *X-form*

lvepxl VRT,RA,RB

31	VRT	RA	RB	263	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + (RB)
VRT ← MEM(EA & 0xFFFF_FFFF_FFFF_FFF0, 16)
mark_as_not_likely_to_be_needed_again_anytime_soon
( EA )
```

Let the effective address (EA) be the sum (RA|0)+(RB). The quadword in storage addressed by the result of EA ANDed with 0xFFFF_FFFF_FFFF_FFF0 is loaded into VRT.

lvepxl provides a hint that the quadword in storage addressed by EA will probably not be needed again by the program in the near future.

For ***lvepxl***, the normal translation mechanism is not used. The contents of the EPLC register are used to provide the context in which translation occurs. The following substitutions are made for just the translation and access control process:

$EPLC_{EPR}$ is used in place of MSR_{PR}
$EPLC_{EAS}$ is used in place of MSR_{DS}
$EPLC_{EPID}$ is used in place of all Process ID registers

This instruction is privileged.

An attempt to execute ***lvepxl*** while MSR_{SPV}=0 will cause a Vector Unavailable interrupt.

Corequisite Categories:
Vector

Special Registers Altered:
None

Programming Note

See the Programming Notes for the ***lvxl*** instruction in Section 5.7.2 of Book I.

Programming Note

This instruction behaves identically to a ***lvxl*** instruction except for using the EPLC register to provide the translation context.

Store Vector by External Process ID Indexed — *X-form*

stvepx VRS,RA,RB

31	VRS	RA	RB	807	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + (RB)
MEM(EA & 0xFFFF_FFFF_FFFF_FFF0, 16) ← (VRS)
```

Let the effective address (EA) be the sum (RA|0)+(RB). The contents of VRS are stored into the quadword in storage addressed by the result of EA ANDed with 0xFFFF_FFFF_FFFF_FFF0.

For ***stvepx***, the normal translation mechanism is not used. The contents of the EPSC register are used to provide the context in which translation occurs. The following substitutions are made for just the translation and access control process:

- $EPSC_{EPR}$ is used in place of MSR_{PR}
- $EPSC_{EAS}$ is used in place of MSR_{DS}
- $EPSC_{EPID}$ is used in place of all Process ID registers

This instruction is privileged.

An attempt to execute ***stvepx*** while MSR_{SPV}=0 will cause a Vector Unavailable interrupt.

Corequisite Categories:
Vector

Special Registers Altered:
None

> **Programming Note**
>
> This instruction behaves identically to a ***stvx*** instruction except for using the EPSC register to provide the translation context.

Store Vector by External Process ID Indexed LRU — *X-form*

stvepxl VRS,RA,RB

31	VRS	RA	RB	775	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + (RB)
MEM(EA & 0xFFFF_FFFF_FFFF_FFF0, 16) ← (VRS)
mark_as_not_likely_to_be_needed_again_anytime_soon
(EA)
```

Let the effective address (EA) be the sum (RA|0)+(RB). The contents of VRS are stored into the quadword in storage addressed by the result of EA ANDed with 0xFFFF_FFFF_FFFF_FFF0.

The ***stvepxl*** instruction provides a hint that the quadword addressed by EA will probably not be needed again by the program in the near future.

For ***stvepxl***, the normal translation mechanism is not used. The contents of the EPSC register are used to provide the context in which translation occurs. The following substitutions are made for just the translation and access control process:

- $EPSC_{EPR}$ is used in place of MSR_{PR}
- $EPSC_{EAS}$ is used in place of MSR_{DS}
- $EPSC_{EPID}$ is used in place of all Process ID registers

This instruction is privileged.

An attempt to execute ***stvepxl*** while MSR_{SPV}=0 will cause a Vector Unavailable interrupt.

Corequisite Categories:
Vector

Special Registers Altered:
None

> **Programming Note**
>
> See the Programming Notes for the ***lvxl*** instruction in Section 5.7.2 of Book I.

> **Programming Note**
>
> This instruction behaves identically to a ***stvxl*** instruction except for using the EPSC register to provide the translation context.

Chapter 4. Storage Control

4.1 Storage Addressing

A program references storage using the effective address computed by the processor 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 4.7.2 and in Section 4.7.3. The real address that results from the respective translations is used to access main storage.

For a complete discussion of storage addressing and effective address calculation, see Section 1.10 of Book I.

4.2 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 Breakpoint).

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 2.1 of Book II and Section 5.7 on page 588 in this Book.

4.3 Instruction Fetch

The effective address for an instruction fetch is processed under control of MSR_{IS}. The Address Translation mechanism is described beginning in Section 4.7.2.

4.3.1 Implicit Branch

Explicitly altering certain MSR bits (using ***mtmsr***), or explicitly altering TLB entries, certain System Registers 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 ***mtmsr*** instruction that changes the value of MSR_{CM} may change the real address from which the current instruction stream is being fetched. The MSR bits and System Registers (excluding implementation-dependent registers) for which alteration can cause an implicit branch are indicated as such in Chapter 10. “Synchronization Requirements for Context Alterations” on page 625. Implicit branches are not supported by the Power ISA. If an implicit branch occurs, the results are boundedly undefined.

4.3.2 Address Wrapping Combined with Changing MSR Bit CM

If the current instruction is at effective address 2^{32}-4 and is an ***mtmsr*** instruction that changes the contents of MSR_{CM}, the effective address of the next sequential instruction is undefined.

Programming Note

In the case described in the preceding paragraph, if an interrupt occurs before the next sequential instruction is executed, the contents of SRR0, CSRR0, or MCSRR0, as appropriate to the interrupt, are undefined.

4.4 Data Access

The effective address for a data access is processed under control of MSR_{DS}. The Address Translation mechanism is described beginning in Section 4.7.2.

Storage control attributes may also affect instruction fetch.

4.5 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 by the processor 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, the processor performs operations out-of-order when it has resources 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, the processor abandons any results of the operation (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 4.8.1.1.

The only permitted side effects of performing an operation out-of-order are the following.

- A Machine Check that could be caused by in-order execution may occur out-of-order.
- 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.

4.6 Invalid Real Address

A storage access (including an access that is performed out-of-order; see Section 4.5) may cause a Machine Check if the accessed storage location contains an uncorrectable error or does not exist. See Section 5.6.2 on page 576.

4.7 Storage Control

This section describes the address translation facility, access control, and storage control attributes.

Demand-paged virtual memory is supported, as well as a variety of other management schemes that depend on precise control of effective-to-real address translation and flexible memory protection. Translation misses and protection faults cause precise exceptions. Sufficient information is available to correct the fault and restart the faulting instruction.

The effective address space is divided into pages. The page represents the granularity of effective address translation, access control, and storage control attributes. Up to sixteen page sizes (1KB, 4KB, 16KB, 64KB, 256KB, 1MB, 4MB, 16MB, 64MB, 256MB, 1GB, 4GB, 16GB, 64GB, 256GB, 1TB) may be simultaneously supported. In order for an effective to real translation to exist, a valid entry for the page containing the effective address must be in the Translation Lookaside Buffer (TLB). Addresses for which no TLB entry exists cause TLB Miss exceptions.

4.7.1 Storage Control Registers

In addition to the registers described below, the Machine State Register provides the IS and DS bits, that specify which of the two address spaces the respective instruction or data storage accesses are directed towards. MSR_{PR} bit is also used by the storage access control mechanism.

4.7.1.1 Process ID Register

The Process ID Register (PID) is a 32-bit register. Process ID Register bits are numbered 32 (most-significant bit) to 63 (least-significant bit). The Process ID Register provides a value that is used to construct a virtual address for accessing storage.

The Process ID Register can be read using ***mfspr*** and can be written using ***mtspr***. An implementation may opt to implement only the least-significant *n* bits of the Process ID Register, where $0 \leq n \leq 32$, and *n* must be the same as the number of implemented bits in the TID field of the TLB entry. The most-significant $32-n$ bits of the Process ID Register are treated as reserved.

Some implementations may support more than one Process ID Register. See User's Manual for the implementation.

4.7.1.2 Translation Lookaside Buffer

The Translation Lookaside Buffer (TLB) is the hardware resource that controls translation, protection, and storage control attributes. The organization of the TLB (e.g. unified versus separate instruction and data, hierarchies, associativity, number of entries, etc.) is implementation-dependent. Thus, the software for updating the TLB is also implementation-dependent. For the purposes of this discussion, a unified TLB organization is assumed. The differences for an implementation with separate instruction and data TLBs are for the most part obvious (e.g. separate instructions or separate index ranges for reading, writing, searching, and invalidating each TLB). For details on how to synchronize TLB updates with instruction execution see Chapter 10.

Maintenance of TLB entries is under software control. System software determines TLB entry replacement strategy and the format and use of any page state information. The TLB entry contains all the information required to identify the page, to specify the translation, to specify access controls, and to specify the storage control attributes. The format of the TLB entry is implementation-dependent.

While the TLB is managed by software, an implementation may include partial or full hardware assist for TLB management (e.g. support of the Server environment's virtual memory architecture). However, such implementations should be able to disable such support with implementation-dependent software or hardware configuration mechanisms.

A TLB entry is written by copying information from a GPR or other implementation-dependent source, using a series of ***tlbwe*** instructions (see page 562). A TLB entry is read by copying information to a GPR or other implementation-dependent target, using a series of ***tlbre*** instructions (see page 560). Software can also search for specific TLB entries using the ***tlbsx*** instruction (see page 561). Writing, reading and searching the TLB is implementation-dependent.

Each TLB entry describes a page that is eligible for translation and access controls. Fields in the TLB entry fall into four categories:

- Page identification fields (information required to identify the page to the hardware translation mechanism).
- Address translation fields
- Access control fields
- Storage attribute fields

While the fields in the TLB entry are required, no particular TLB entry format is formally specified. The ***tlbre*** and ***tlbwe*** instructions provide the ability to read or write portions of individual entries. Below are shown the field definitions for the TLB entry.

Page Identification Fields

Name Description

EPN **Effective Page Number** (up to 54 bits)
Bits 0:*n*–1 of the EPN field are compared to bits 0:*n*–1 of the effective address (EA) of the storage access (where *n*=64–log2(*page size* in bytes) and *page size* is specified by the SIZE field of the TLB entry). See Table 1.
Note: Bits X:Y of the EPN field may be implemented, where X=0 or X=32, and Y ≤ 53. The number of bits implemented for EPN are not required to be the same number of bits as are implemented for RPN.

TS Translation Address Space
This bit indicates the address space this TLB entry is associated with. For instruction storage accesses, MSR_{IS} must match the value of TS in the TLB entry for that TLB entry to provide the translation. Likewise, for data storage accesses, MSR_{DS} must match the value of TS in the TLB entry. For ***tlbsx*** and ***tlbivax*** instructions, an implementation-dependent source provides the address space specification that must match the value of TS.

SIZE Page Size
The SIZE field specifies the size of the page associated with the TLB entry as 4^{SIZE}KB, where 0 ≤ SIZE ≤ 15. Implementations may implement any one or more of these page sizes. See Table 1.

TID **Translation ID** (implementation-dependent size)
Field used to identify a shared page (TID=0) or the owner's process ID of a private page (TID≠0). See Section 4.7.2.

V Valid
This bit indicates that this TLB entry is valid and may be used for translation. The Valid bit for a given entry can be set or cleared with a ***tlbwe*** instruction; alternatively, the Valid bit for an entry may be cleared by a ***tlbivax*** instruction.

Translation Field

Name Description

RPN **Real Page Number** (up to 54 bits)
Bits 0:*n*–1 of the RPN field are used to replace bits 0:*n*–1 of the effective address to produce the real address for the storage access (where *n*=64–log2(*page size* in bytes) and *page size* is specified by the SIZE field of the TLB entry). Software must set unused low-order RPN bits (i.e. bits *n*:53) to 0. See Section 4.7.3.
Note: Bits X:Y of the RPN field may be implemented, where X ≥ 0 and 53 ≥ Y. The number of bits implemented for EPN are not required to be the same number of bits as are implemented for RPN.

Storage Control Bits (see Section 4.8.3 on page 552)

Name Description

W **Write-Through Required** See Section 1.6.1 of Book II.

I **Caching Inhibited** See Section 1.6.2 of Book II.

M **Memory Coherence Required** See Section 1.6.3 of Book II.

G **Guarded** See Section 1.6.4 of Book II and Section 4.8.1.

E **Endian Mode** See Section 1.10.1 of Book I and Section 1.6.5 of Book II.

U0:U3 **User-Definable Storage Control Attributes** See Section 4.8.2.
Specifies implementation-dependent and system-dependent storage control attributes for the page associated with the TLB entry.

VLE **Variable Length Encoding** [Category: VLE] See Section 4.8.3 and Chapter 1 of Book VLE.

Access Control Fields

Name Description

UX **User State Execute Enable** See Section 4.7.4.1.
0 Instruction fetch and execution is not permitted from this page while MSR_{PR}=1 and will cause an Execute Access Control exception type Instruction Storage interrupt.
1 Instruction fetch and execution is permitted from this page while MSR_{PR}=1.

SX **Supervisor State Execute Enable** See Section 4.7.4.1.

0 Instruction fetch and execution is not permitted from this page while MSR_{PR}=0 and will cause an Execute Access Control exception type Instruction Storage interrupt.

1 Instruction fetch and execution is permitted from this page while MSR_{PR}=1.

UW **User State Write Enable** See Section 4.7.4.2.

0 *Store* operations, including ***dcba dcbz***, and ***dcbzep*** are not permitted to this page when MSR_{PR}=1 and will cause a Write Access Control exception. Except as noted in Table 3 on page 550, a Write Access Control exception will cause a Data Storage interrupt.

1 *Store* operations, including ***dcba, dcbz***, and ***dcbzep*** are permitted to this page when MSR_{PR}=1.

SW **Supervisor State Write Enable** See Section 4.7.4.2.

0 *Store* operations, including ***dcba***, ***dcbi***, ***dcbz***, and ***dcbzep*** are not permitted to this page when MSR_{PR}=0. *Store* operations, including ***dcbi, dcbz,*** and ***dcbzep***, will cause a Write Access Control exception. Except as noted in Table 3 on page 550, a Write Access Control exception will cause a Data Storage interrupt.

1 *Store* operations, including ***dcba***, ***dcbi***, ***dcbz***, and ***dcbzep***, are permitted to this page when MSR_{PR}=0.

UR **User State Read Enable** See Section 4.7.4.3.

0 *Load* operations (including load-class *Cache Management* instructions) are not permitted from this page when MSR_{PR}=1 and will cause a Read Access Control exception. Except as noted in Table 3 on page 550, a Read Access Control exception will cause a Data Storage interrupt.

1 *Load* operations (including load-class *Cache Management* instructions) are permitted from this page when MSR_{PR}=1.

SR **Supervisor State Read Enable** See Section 4.7.4.3.

0 *Load* operations (including load-class *Cache Management* instructions) are not permitted from this page when MSR_{PR}=0 and will cause a Read Access Control exception. Except as noted in Table 3 on page 550, a Read Access Control exception will cause a Data Storage interrupt.

1 *Load* operations (including load-class *Cache Management* instructions) are permitted from this page when MSR_{PR}=0.

4.7.2 Page Identification

Instruction effective addresses are generated for sequential instruction fetches and for addresses that correspond to a change in program flow (branches, interrupts). Data effective addresses are generated by *Load*, *Store*, and *Cache Management* instructions. *TLB Management* instructions generate effective addresses to determine the presence of or to invalidate a specific TLB entry associated with that address.

The Valid (V) bit, Effective Page Number (EPN) field, Translation Space Identifier (TS) bit, Page Size (SIZE) field, and Translation ID (TID) field of a particular TLB entry identify the page associated with that TLB entry. Except as noted, all comparisons must succeed to validate this entry for subsequent translation and access control processing. Failure to locate a matching TLB entry based on this criteria for instruction fetches will result in an Instruction TLB Miss exception type Instruction TLB Error interrupt. Failure to locate a matching TLB entry based on this criteria for data storage accesses will result in a Data TLB Miss exception which may result in a Data TLB Error interrupt. Figure 8 on page 546 illustrates the criteria for a virtual address to match a specific TLB entry.

There are two address spaces, one typically associated with interrupt-related storage accesses and one typically associated with non-interrupt-related storage accesses. There are two bits in the Machine State Register, the Instruction Address Space bit (IS) and the Data Address Space bit (DS), that control which address space instruction and data storage accesses, respectively, are performed in, and a bit in the TLB entry (TS) that specifies which address space that TLB entry is associated with.

Load, *Store*, *Cache Management*, *Branch*, ***tlbsx***, and ***tlbivax*** instructions and next-sequential-instruction fetches produce a 64-bit effective address. The virtual address space is extended from this 64-bit effective address space by prepending a one-bit address space identifier and a process identifier. For instruction fetches, the address space identifier is provided by MSR_{IS} and the process identifier is provided by the contents of the Process ID Register. For data storage accesses, the address space identifier is provided by the MSR_{DS} and the process identifier is provided by the contents of the Process ID Register. For ***tlbsx***, and ***tlbivax*** instructions, the address space identifier and the process identifier are provided by implementation-dependent sources.

This virtual address is used to locate the associated entry in the TLB. The address space identifier, the process identifier, and the effective address of the storage access are compared to the Translation Address Space bit (TS), the Translation ID field (TID), and the value in the Effective Page Number field (EPN), respectively, of each TLB entry.

The virtual address of a storage access matches a TLB entry if, for every TLB entry i in the congruence class specified by EA:

- the value of the address specifier for the storage access (MSR_{IS} for instruction fetches, MSR_{DS} for data storage accesses, and implementation-dependent source for ***tlbsx*** and ***tlbivax***) is equal to the value of the TS bit of the TLB entry, and
- either the value of the process identifier (Process ID Register for instruction and data storage accesses, and implementation-dependent source for ***tlbsx*** and ***tlbivax***) is equal to the value in the TID field of the TLB entry, or the value of the TID field of the TLB entry is equal to 0, and
- the contents of bits 0:*n*–1 of the effective address of the storage or TLB access are equal to the value of bits 0:*n*-1 of the EPN field of the TLB entry (where n=64-$\log_2$(*page size* in bytes) and *page size* is specified by the value of the SIZE field of the TLB entry). See Table 1.

A TLB Miss exception occurs if there is no valid entry in the TLB for the page specified by the virtual address (Instruction or Data TLB Error interrupt). Although the possibility to place multiple entries into the TLB that match a specific virtual address exists, assuming a set-associative or fully-associative organization, doing so is a programming error and the results are undefined.

Table 1: Page Size and Effective Address to EPN Comparison

SIZE	Page Size (4^{SIZE}KB)	EA to EPN Comparison (bits 0:53–2¥SIZE)
=0b0000	1KB	$EPN_{0:53}$ =? $EA_{0:53}$
=0b0001	4KB	$EPN_{0:51}$ =? $EA_{0:51}$
=0b0010	16KB	$EPN_{0:49}$ =? $EA_{0:49}$
=0b0011	64KB	$EPN_{0:47}$ =? $EA_{0:47}$
=0b0100	256KB	$EPN_{0:45}$ =? $EA_{0:45}$
=0b0101	1MB	$EPN_{0:43}$ =? $EA_{0:43}$
=0b0110	4MB	$EPN_{0:41}$ =? $EA_{0:41}$
=0b0111	16MB	$EPN_{0:39}$ =? $EA_{0:39}$
=0b1000	64MB	$EPN_{0:37}$ =? $EA_{0:37}$
=0b1001	256MB	$EPN_{0:35}$ =? $EA_{0:35}$
=0b1010	1GB	$EPN_{0:33}$ =? $EA_{0:33}$
=0b1011	4GB	$EPN_{0:31}$ =? $EA_{0:31}$
=0b1100	16GB	$EPN_{0:29}$ =? $EA_{0:29}$
=0b1101	64GB	$EPN_{0:27}$ =? $EA_{0:27}$
=0b1110	256GB	$EPN_{0:25}$ =? $EA_{0:25}$
=0b1111	1TB	$EPN_{0:23}$ =? $EA_{0:23}$

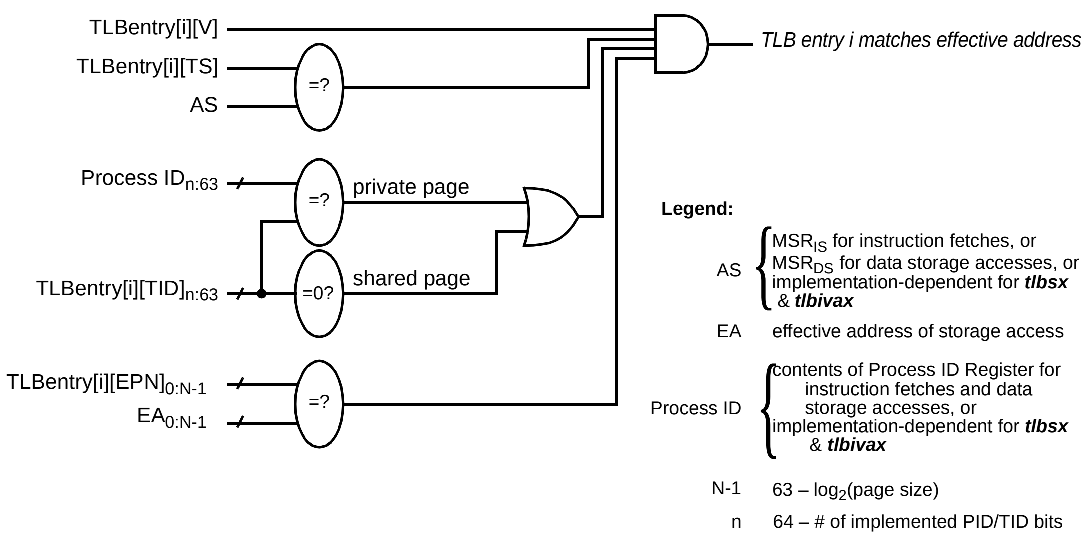


Figure 8. Virtual Address to TLB Entry Match Process

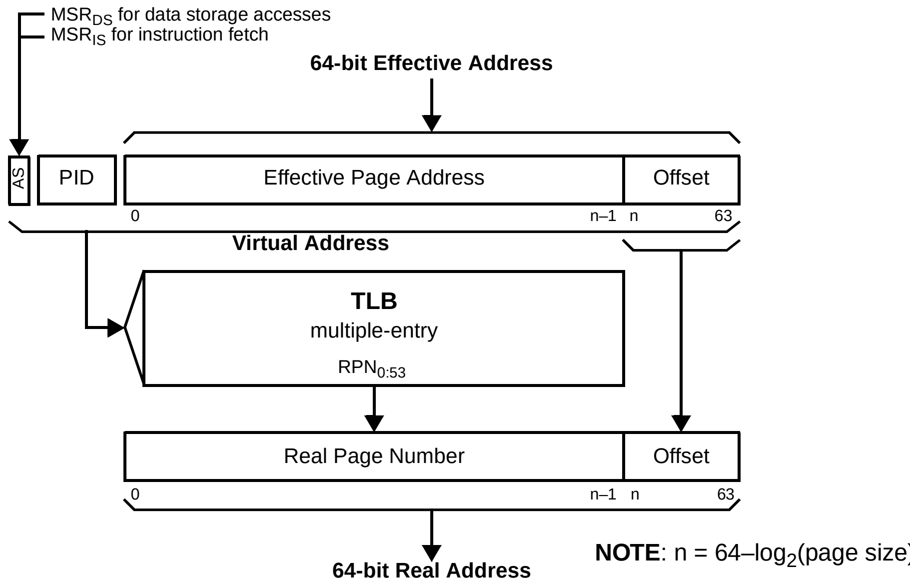


Figure 9. Effective-to-Real Address Translation Flow

4.7.3 Address Translation

A program references memory by using the effective address computed by the processor when it executes a *Load*, *Store*, *Cache Management*, or *Branch* instruction, and when it fetches the next instruction. The effective address is translated to a real address according to the procedures described in this section. The storage subsystem uses the real address for the access. All storage access effective addresses are translated to real addresses using the TLB mechanism. See Figure 9.

If the virtual address of the storage access matches a TLB entry in accordance with the selection criteria specified in Section 4.7.2, the value of the Real Page Number field (RPN) of the selected TLB entry provides the real page number portion of the real address. Let n=64–log$_2$(*page size* in bytes) where *page size* is specified by the SIZE field of the TLB entry. Bits n:63 of the effective address are appended to bits 0:n–1 of the 54-bit RPN field of the selected TLB entry to produce the 64-bit real address (i.e. RA = $RPN_{0:n-1}$ || $EA_{n:63}$). The page size is determined by the value of the SIZE field of the selected TLB entry. See Table 2.

The rest of the selected TLB entry provides the access control bits (UX, SX, UW, SW, UR, SR), and storage control attributes (U0, U1, U2, U3, W, I, M, G, E) for the storage access. The access control bits and storage attribute bits specify whether or not the access is allowed and how the access is to be performed. See Sections 4.7.4 and 4.7.5.

The Real Page Number field (RPN) of the matching TLB entry provides the translation for the effective address of the storage access. Based on the setting of the SIZE field of the matching TLB entry, the RPN field replaces the corresponding most-significant N bits of the effective address (where N = 64 – log$_2$(page size)), as shown in Table 2, to produce the 64-bit real address that is to be presented to main storage to perform the storage access.

Table 2: Effective Address to Real Address

SIZE	Page Size (4^{SIZE} KB)	RPN Bits Required to be Equal to 0	Real Address
=0b0000	1KB	none	$RPN_{0:53}$ || $EA_{54:63}$
=0b0001	4KB	$RPN_{52:53}$=0	$RPN_{0:51}$ || $EA_{52:63}$
=0b0010	16KB	$RPN_{50:53}$=0	$RPN_{0:49}$ || $EA_{50:63}$
=0b0011	64KB	$RPN_{48:53}$=0	$RPN_{0:47}$ || $EA_{48:63}$
=0b0100	256KB	$RPN_{46:53}$=0	$RPN_{0:45}$ || $EA_{46:63}$
=0b0101	1MB	$RPN_{44:53}$=0	$RPN_{0:43}$ || $EA_{44:63}$
=0b0110	4MB	$RPN_{42:53}$=0	$RPN_{0:41}$ || $EA_{42:63}$
=0b0111	16MB	$RPN_{40:53}$=0	$RPN_{0:39}$ || $EA_{40:63}$
=0b1000	64MB	$RPN_{38:53}$=0	$RPN_{0:37}$ || $EA_{38:63}$
=0b1001	256MB	$RPN_{36:53}$=0	$RPN_{0:35}$ || $EA_{36:63}$
=0b1010	1GB	$RPN_{34:53}$=0	$RPN_{0:33}$ || $EA_{34:63}$
=0b1011	4GB	$RPN_{32:53}$=0	$RPN_{0:31}$ || $EA_{32:63}$
=0b1100	16GB	$RPN_{30:53}$=0	$RPN_{0:29}$ || $EA_{30:63}$
=0b1101	64GB	$RPN_{28:53}$=0	$RPN_{0:27}$ || $EA_{28:63}$
=0b1110	256GB	$RPN_{26:53}$=0	$RPN_{0:25}$ || $EA_{26:63}$
=0b1111	1TB	$RPN_{24:53}$=0	$RPN_{0:23}$ || $EA_{24:63}$

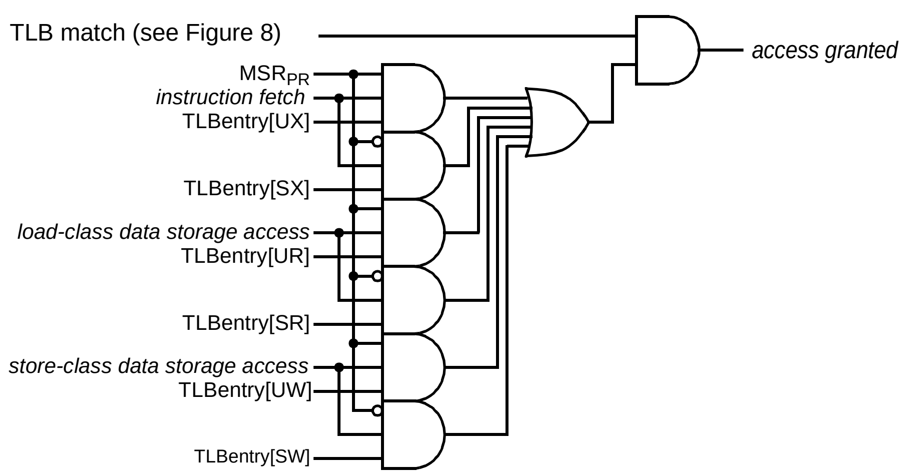


Figure 10. Access Control Process

4.7.4 Storage Access Control

After a matching TLB entry has been identified, an access control mechanism selectively grants shared access, grants execute access, grants read access, grants write access, and prohibits access to areas of storage based on a number of criteria. Figure 10 illustrates the access control process and is described in detail in Sections 4.7.4.1 through 4.7.4.5.

An Execute, Read, or Write Access Control exception occurs if the appropriate TLB entry is found but the access is not allowed by the access control mechanism (Instruction or Data Storage interrupt). See Section 5.6 for additional information about these and other interrupt types. In certain cases, Execute, Read, and Write Access Control exceptions may result in the restart of (re-execution of at least part of) a *Load* or *Store* instruction.

Some implementation may provide additional access control capabilities beyond that described here.

4.7.4.1 Execute Access

The UX and SX bits of the TLB entry control *execute* access to the page (see Table 3).

Instructions may be fetched and executed from a page in storage while in user state (MSR_{PR}=1) if the UX access control bit for that page is equal to 1. If the UX access control bit is equal to 0, then instructions from that page will not be fetched, and will not be placed into any cache as the result of a fetch request to that page while in user state.

Instructions may be fetched and executed from a page in storage while in supervisor state (MSR_{PR}=0) if the SX access control bit for that page is equal to 1. If the SX access control bit is equal to 0, then instructions from that page will not be fetched, and will not be placed into any cache as the result of a fetch request to that page while in supervisor state.

Instructions from no-execute storage may be in the instruction cache if they were fetched into that cache when their effective addresses were mapped to execute permitted storage. Software need not flush a page from the instruction cache before marking it no-execute.

Furthermore, if the sequential execution model calls for the execution of an instruction from a page that is not enabled for execution (i.e. UX=0 when MSR_{PR}=1 or SX=0 when MSR_{PR}=0), an Execute Access Control exception type Instruction Storage interrupt is taken.

4.7.4.2 Write Access

The UW and SW bits of the TLB entry control *write* access to the page (seeTable 3).

Store operations (including *Store*-class *Cache Management* instructions) are permitted to a page in storage while in user state (MSR_{PR}=1) if the UW access control bit for that page is equal to 1. If the UW access control bit is equal to 0, then execution of the *Store* instruction is suppressed and a Write Access Control exception type Data Storage interrupt is taken.

Store operations (including *Store*-class *Cache Management* instructions) are permitted to a page in storage while in supervisor state (MSR_{PR}=0) if the SW access control bit for that page is equal to 1. If the SW access control bit is equal to 0, then execution of the *Store* instruction is suppressed and a Write Access Control exception type Data Storage interrupt is taken.

4.7.4.3 Read Access

The UR and SR bits of the TLB entry control *read* access to the page (see Table 3).

Load operations (including *Load*-class *Cache Management* instructions) are permitted from a page in storage while in user state (MSR_{PR}=1) if the UR access control bit for that page is equal to 1. If the UR access control bit is equal to 0, then execution of the *Load* instruction is suppressed and a Read Access Control exception type Data Storage interrupt is taken.

Load operations (including *Load*-class *Cache Management* instructions) are permitted from a page in storage while in supervisor state (MSR_{PR}=0) if the SR access control bit for that page is equal to 1. If the SR access control bit is equal to 0, then execution of the *Load* instruction is suppressed and a Read Access Control exception type Data Storage interrupt is taken.

4.7.4.4 Storage Access Control Applied to Cache Management Instructions

dcbi, dcbz, and ***dcbzep*** instructions are treated as *Stores* since they can change data (or cause loss of data by invalidating a dirty line). As such, they both can cause Write Access Control exception type Data Storage interrupts. If an implementation first flushes a line before invalidating it during a ***dcbi***, the ***dcbi*** is treated as a a *Load* since the data is not modified.

dcba instructions are treated as *Stores* since they can change data. As such, they can cause Write Access Control exceptions. However, such exceptions will not result in a Data Storage interrupt.

icbi and ***icbiep*** instructions are treated as *Loads* with respect to protection. As such, they can cause Read Access Control exception type Data Storage interrupts.

dcbt, ***dcbtep***, ***dcbtst***, ***dcbtstep***, and ***icbt*** instructions are treated as *Loads* with respect to protection. As such, they can cause Read Access Control exceptions. However, such exceptions will not result in a Data Storage interrupt.

dcbf, ***dcbfep***, ***dcbst***, and ***dcbstep*** instructions are treated as *Loads* with respect to protection. Flushing or storing a line from the cache is not considered a *Store* since the store has already been done to update the cache and the ***dcbf***, ***dcbfep***, ***dcbst***, or ***dcbstep*** instruction is only updating the copy in main storage. As a *Load*, they can cause Read Access Control exception type Data Storage interrupts.

Table 3: Storage Access Control Applied to Cache Instructions

Instruction	Read Protection Violation	Write Protection Violation
dcba	No	Yes[2]
dcbf	Yes	No
dcbfep	Yes	No
dcbi	Yes[3]	Yes[3]
dcblc	Yes	No
dcbst	Yes	No
dcbstep	Yes	No
dcbt	Yes[1]	No
dcbtep	Yes[1]	No
dcbtls	Yes	No
dcbtst	Yes[1]	No
dcbtstep	Yes[1]	No
dcbtstls	Yes[4]	Yes[4]
dcbz	No	Yes
dcbzep	No	Yes
dci	No	No
icbi	Yes	No
icbiep	Yes	No
icblc	Yes[5]	No
icbt	Yes[1]	No
icbtls	Yes[5]	No
ici	No	No

1. ***dcbt, dcbtep***, ***dcbtst, dcbtstep***, and ***icbt*** may cause a Read Access Control exception but does not result in a Data Storage interrupt.
2. ***dcba*** may cause a Write Access Control exception but does not result in a Data Storage interrupt.
3. ***dcbi*** may cause a Read or Write Access Control Exception based on whether the data is flushed prior to invalidation.
4. It is implementation-dependent whether ***dcbtstls*** is treated as a *Load* or a *Store*.
5. ***icbtls*** and ***icblc*** require execute or read access.

4.7.4.5 Storage Access Control Applied to String Instructions

When the string length is zero, neither ***lswx*** nor ***stswx*** can cause Data Storage interrupts.

4.7.5 TLB Management

No format for the Page Tables or the Page Table Entries is implied. Software has significant flexibility in implementing a custom replacement strategy. For example, software may choose to lock TLB entries that correspond to frequently used storage, so that those entries are never cast out of the TLB and TLB Miss exceptions to those pages never occur. At a minimum, software must maintain an entry or entries for the Instruction and Data TLB Error interrupt handlers.

TLB management is performed in software with some hardware assist. This hardware assist consists of a minimum of:

- Automatic recording of the effective address causing a TLB Miss exception. For Instruction TLB Miss exceptions, the address is saved in the Save/ Restore Register 0. For Data TLB Miss exceptions, the address is saved in the Data Exception Address Register.
- Instructions for reading, writing, searching, invalidating, and synchronizing the TLB (see Section 4.9.4.1).

Programming Note

This Note suggests one example for managing reference and change recording.

When performing physical page management, it is useful to know whether a given physical page has been referenced or altered. Note that this may be more involved than whether a given TLB entry has been used to reference or alter memory, since multiple TLB entries may translate to the same physical page. If it is necessary to replace the contents of some physical page with other contents, a page which has been referenced (accessed for any purpose) is more likely to be maintained than a page which has never been referenced. If the contents of a given physical page are to be replaced, then the contents of that page must be written to the backing store before replacement, if anything in that page has been changed. Software must maintain records to control this process.

Similarly, when performing TLB management, it is useful to know whether a given TLB entry has been referenced. When making a decision about which entry to cast-out of the TLB, an entry which has been referenced is more likely to be maintained in the TLB than an entry which has never been referenced.

Execute, Read and Write Access Control exceptions may be used to allow software to maintain reference information for a TLB entry and for its associated physical page. The entry is built, with its UX, SX, UR, SR, UW, and SW bits off, and the index and effective page number of the entry retained by software. The first attempt of application code to use the page will cause an Access Control exception (because the entry is marked “No Execute”, “No Read”, and “No Write”). The Instruction or Data Storage interrupt handler records the reference to the TLB entry and to the associated physical page in a software table, and then turns on the appropriate access control bit. An initial read from the page could be handled by only turning on the appropriate UR or SR access control bits, leaving the page “read-only”. Subsequent execute, read, or write accesses to the page via this TLB entry will proceed normally.

In a demand-paged environment, when the contents of a physical page are to be replaced, if any storage in that physical page has been altered, then the backing storage must be updated. The information that a physical page is dirty is typically recorded in a “Change” bit for that page.

Write Access Control exceptions may be used to allow software to maintain change information for a physical page. For the example just given for reference recording, the first write access to the page via the TLB entry will create a Write Access Control exception type Data Storage interrupt. The Data Storage interrupt handler records the change status to the physical page in a software table, and then turns on the appropriate UW and SW bits. All subsequent accesses to the page via this TLB entry will proceed normally.

4.8 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.

4.8.1 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 the G bit is 1 in the TLB entry that translates the effective address.

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.

Instruction fetching is not affected by the G bit. Software must set guarded pages to no execute (i.e. UX=0 and SX=0) to prevent instruction fetching from guarded storage.

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, the instruction completes before the interrupt occurs if any of the following exceptions is pending.
  - External, Decrementer, Critical Input, Machine Check, Fixed-Interval Timer, Watchdog Timer, Debug, or Imprecise mode Floating-Point or Auxiliary Processor Enabled
- *Load* or *Store* instruction that causes an Alignment exception, a Data TLB Error exception, or that causes a Data Storage exception.

  The portion of the storage operand that is in Caching Inhibited and Guarded storage is not accessed.

4.8.1.1 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.

4.8.2 User-Definable

User-definable storage control attributes control user-definable and implementation-dependent behavior of the storage system. These bits are both implementation-dependent and system-dependent in their effect. They may be used in any combination and also in combination with the other storage attribute bits.

4.8.3 Storage Control Bits

Storage control attributes are specified on a per-page basis. These attributes are specified in storage control bits in the TLB entries. The interpretation of their values is given in Figure 11.

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
E[3]	0 - Big-Endian 1 - Little-Endian
U0-U3[4]	User-Definable
VLE[5]	0 - non Variable Length Encoding (VLE). 1 - VLE

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 of the 1 value is optional for implementations that do not support multiprocessing, implementations that do not support this storage attribute assume the value of the bit to be 0, and setting M=1 in a TLB entry will have no effect.
3 [Category: Embedded.Little-Endian]
4 Support for these attributes is optional.
5 [Category: VLE]

Figure 11. Storage control bits

In Section 4.8.3.1 and 4.8.3.2, “access” includes accesses that are performed out-of-order.

Programming Note

In a uniprocessor system in which only the processor has caches, correct coherent execution does not require the processor to access storage as Memory Coherence Required, and accessing storage as not Memory Coherence Required may give better performance.

4.8.3.1 Storage Control Bit Restrictions

All combinations of W, I, M, G, and E 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 I bit must be the same for all accesses to a given real page.

Accesses to the same storage location using two effective addresses for which the W bit differs meet the memory coherence requirements described in Section 1.6.3 of Book II if the accesses are performed by a single processor. If the accesses are performed by two or more processors, coherence is enforced by the hardware only if the W bit is the same for all the accesses.

At any given time, data accesses to a given real page may use both Endian modes. When changing the Endian mode of a given real page for instruction fetching, care must be taken to prevent accesses while the change is made and to flush the instruction cache(s) after the change has been completed.

4.8.3.2 Altering the Storage Control Bits

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, dcbfep,*** or ***dcbi****,* and ***icbi*** or ***icbiep*** before permitting any other accesses to the page.

When changing the value of the W bit for a given real page from 0 to 1, software must ensure that no processor 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, dcbstep***, ***dcbf***, ***dcbfep***, or ***dcbi***.

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 to are system-dependent.

Programming Note

For example, when changing the M bit in some directory-based systems, software may be required to execute ***dcbf*** or ***dcbfep*** on each processor to flush all storage locations accessed with the old M value before permitting the locations to be accessed with the new M value.

4.9 Storage Control Instructions

4.9.1 Cache Management Instructions

This section describes aspects of cache management that are relevant only to privileged software programmers.

For a ***dcbz*** or ***dcba*** instruction that causes the target block to be newly established in the data cache without being fetched from main storage, the processor 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 4.6) 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 processor enters any power conserving mode in which data cache contents are not maintained.

Data Cache Block Invalidate X-form

dcbi RA,RB

31	///	RA	RB	470	/
0	6	11	16	21	31

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

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 processors, then the block is invalidated in those data caches. On some implementations, before the block is invalidated, if any locations in the block are considered to be modified in any such data cache, those locations are written to main storage and additional locations in the block may be written to main storage.

If the block containing the byte addressed by EA is in storage that is not Memory Coherence Required and a block containing the byte addressed by EA is in the data cache of this processor, then the block is invalidated in that data cache. On some implementations, before the block is invalidated, if any locations in the block are considered to be modified in that data cache, those locations are written to main storage and additional locations in the block may be written to main storage.

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 *Store* (see Section 4.7.4.4) on implementations that invalidate a block without first writing to main storage all locations in the block that are considered to be modified in the data cache, except that the invalidation is not ordered by ***mbar***. On other implementations this instruction is treated as a *Load* (see the section cited above).

If a processor holds a reservation and some other processor executes a ***dcbi*** to the same reservation granule, whether the reservation is lost is undefined.

dcbi may cause a cache locking exception, the details of which are implementation-dependent.

This instruction is privileged.

Special Registers Altered:

None

4.9.2 Cache Locking [Category: Embedded Cache Locking]

The *Embedded Cache Locking* category defines instructions and methods for locking cache blocks for frequently used instructions and data. Cache locking allows software to instruct the cache to keep latency sensitive data readily available for fast access. This is accomplished by marking individual cache blocks as locked.

A locked block differs from a normal block in the cache in the following way:

- blocks that are locked in the cache do not participate in the normal replacement policy when a block must be replaced.

4.9.2.1 Lock Setting and Clearing

Blocks are locked into the cache by software using *Cache Locking* instructions. The following instructions are provided to lock data items into the data and instruction cache:

- ***dcbtls*** - Data cache block touch and lock set.
- ***dcbtstls*** - Data cache block touch for store and lock set.
- ***icbtls*** - Instruction cache block touch and lock set.

The RA and RB operands in these instructions are used to identify the block to be locked. The CT field indicates which cache in the cache hierarchy should be targeted. (See Section 3.2 of Book II.)

These instructions are similar in nature to the ***dcbt***, ***dcbtst***, and ***icbt*** instructions, but are not hints and thus locking instructions do not execute speculatively and may cause additional exceptions. For unified caches, both the instruction lock set and the data lock set target the same cache.

Similarly, blocks are unlocked from the cache by software using *Lock Clear* instructions. The following instructions are provided to unlock instructions and data in their respective caches:

- ***dcblc*** - Data cache block lock clear.
- ***icblc*** - Instruction cache block lock clear.

The RA and RB operands in these instructions are used to identify the block to be unlocked. The CT field indicates which cache in the cache hierarchy should be targeted.

Additionally, an implementation-dependent method can be provided for software to clear all the locks in the cache.

An implementation is not required to unlock blocks that contain data that has been invalidated unless it is explicitly unlocked with a ***dcblc*** or ***icblc*** instruction; if the implementation does not unlock the block upon invalidation, the block remains locked even though it contains invalid data. If the implementation does not clear locks when the associated block is invalidated, the method of locking is said to be *persistent*; otherwise it is *not persistent*. An implementation may choose to implement locks as persistent or not persistent; however, the preferred method is persistent.

It is implementation-dependent if cache blocks are implicitly unlocked in the following ways:

- A locked block is invalidated as the result of a ***dcbi***, ***dcbf***, ***dcbfep***, ***icbi***, or ***icbiep*** instruction.
- A locked block is evicted because of an overlocking condition.
- A snoop hit on a locked block that requires the block to be invalidated. This can occur because the data the block contains has been modified external to the processor, or another processor has explicitly invalidated the block.
- The entire cache containing the locked block is invalidated.

4.9.2.2 Error Conditions

Setting locks in the cache can fail for a variety of reasons. A *Lock Set* instruction addressing a byte in storage that is not allowed to be accessed by the storage access control mechanism (see Section 4.7.4) will cause a Data Storage interrupt (DSI). Addresses referenced by *Cache Locking* instructions are always translated as data references; therefore, ***icbtls*** instructions that fail to translate or are not allowed by the storage access control mechanism cause Data TLB Error interrupts and Data Storage interrupts, respectively. Additionally, cache locking and clearing operations can fail due to non-privileged access. The methods for determining other failure conditions such as unable-to-lock or overlocking (see below), is implementation-dependent.

When a *Cache Locking* instruction is executed in user mode and MSR_{UCLE} is 0, a Data Storage interrupt occurs and one of the following ESR bits is set to 1.

Bit **Description**

42 ***DLK_0***

0 Default setting.
1 A ***dcbtls***, ***dcbtstls***, or ***dcblc*** instruction was executed in user mode.

43 ***DLK_1***

0 Default setting.
1 An ***icbtls*** or ***icblc*** instruction was executed in user mode.

4.9.2.2.1 Overlocking

If no exceptions occur for the execution of an ***dcbtls***, ***dcbtstls***, or ***icbtls*** instruction, an attempt is made to lock the specified block into the cache. If all of the available cache blocks into which the specified block may be loaded are already locked, an overlocking condition occurs. The overlocking condition may be reported in an implementation-dependent manner.

If an overlocking condition occurs, it is implementation-dependent whether the specified block is not locked into the cache or if another locked block is evicted and the specified block is locked.

The selection of which block is replaced in an overlocking situation is implementation-dependent. The overlocking condition is still said to exist, and is reflected in any implementation-dependent overlocking status.

An attempt to lock a block that is already present and valid in the cache will not cause an overlocking condition.

If a cache block is to be loaded because of an instruction other than a *Cache Management* or *Cache Locking* instruction and all available blocks into which the block can be loaded are locked, the instruction executes and completes, but no cache blocks are unlocked and the block is not loaded into the cache.

Programming Note

Since caches may be shared among processors, an overlocking condition may occur when loading a block even though a given processor has not locked all the available cache blocks. Similarly. blocks may be unlocked as a result of invalidations by other processors.

4.9.2.2.2 Unable-to-lock and Unable-to-unlock Conditions

If no exceptions occur and no overlocking condition exists, an attempt to set or unlock a lock may fail if any of the following are true:

- The target address is marked Caching Inhibited, or the storage attributes of the address use a coherency protocol that does not support locking.
- The target cache is disabled or not present.
- The CT field of the instructions contains a value not supported by the implementation.
- Any other implementation-specific error conditions are detected.

If an unable-to-lock or unable-to-unlock condition occurs, the lock set or unlock instruction is treated as a no-op and the condition may be reported in an implementation-dependent manner.

4.9.2.3 Cache Locking Instructions

Data Cache Block Touch and Lock Set X-form

dcbtls CT,RA,RB

31	/	CT	RA	RB	166	/
0	6	7	11	16	21	31

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

The ***dcbtls*** instruction provides a hint that the program will probably soon load from the block containing the byte addressed by EA, and that the block containing the byte addressed by EA is to be loaded and locked into the cache specified by the CT field. (See Section 3.2 of Book II.) If the CT field is set to a value not supported by the implementation, no operation is performed.

If the block already exists in the cache, the block is locked without accessing storage. If the block is in a storage location that is Caching Inhibited, then no cache operation is performed. An unable-to-lock condition may occur (see Section 4.9.2.2.2), or an overlocking condition may occur (see Section 4.9.2.2.1).

The ***dcbtls*** instruction may complete before the operation it causes has been performed.

The instruction is treated as a *Load*.

This instruction is privileged unless the Embedded Cache Locking.User Mode category is supported. If the Embedded Cache Locking.User Mode category is supported, this instruction is privileged only if MSR_{UCLE}=0.

Special Registers Altered:

None

Data Cache Block Touch for Store and Lock Set X-form

dcbtstls CT,RA,RB

31	/	CT	RA	RB	134	/
0	6	7	11	16	21	31

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

The ***dcbtstls*** instruction provides a hint that the program will probably soon store to the block containing the byte addressed by EA, and that the block containing the byte addressed by EA is to be loaded and locked into the cache specified by the CT field. (See Section 3.2 of Book II.) If the CT field is set to a value not supported by the implementation, no operation is performed.

If the block already exists in the cache, the block is locked without accessing storage. If the block is in a storage location that is Caching Inhibited, then no cache operation is performed. An unable-to-lock condition may occur (see Section 4.9.2.2.2), or an overlocking condition may occur (see Section 4.9.2.2.1).

The ***dcbtstls*** instruction may complete before the operation it causes has been performed.

It is implementation-dependent whether the instruction is treated as a *Load* or a *Store*.

This instruction is privileged unless the Embedded Cache Locking.User Mode category is supported. If the Embedded Cache Locking.User Mode category is supported, this instruction is privileged only if MSR_{UCLE}=0.

Special Registers Altered:

None

Instruction Cache Block Touch and Lock Set X-form

icbtls CT,RA,RB

31	/	CT	RA	RB	486	/
0	6	7	11	16	21	31

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

The ***icbtls*** instruction causes the block containing the byte addressed by EA to be loaded and locked into the instruction cache specified by CT, and provides a hint that the program will probably soon execute code from the block. See Section 3.2 of Book II for a definition of the CT field.

If the block already exists in the cache, the block is locked without refetching from memory. If the block is in storage that is Caching Inhibited, no cache operation is performed.

This instruction treated as a *Load* (see Section 3.2), except that the system instruction storage error handler is not invoked.

An unable-to-lock condition may occur (see Section 4.9.2.2.2), or an overlocking condition may occur (see Section 4.9.2.2.1).

This instruction is privileged unless the Embedded Cache Locking.User Mode category is supported. If the Embedded Cache Locking.User Mode category is supported, this instruction is privileged only if MSR_{UCLE}=0.

Special Registers Altered:
None

Instruction Cache Block Lock Clear X-form

icblc CT,RA,RB

31	/	CT	RA	RB	230	/
0	6	7	11	16	21	31

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

The block containing the byte addressed by EA in the instruction cache specified by the CT field is unlocked.

The instruction is treated as a *Load*.

An unable-to-unlock condition may occur (see Section 4.9.2.2.2). If the block containing the byte addressed by EA is not locked in the specified cache, no cache operation is performed.

This instruction is privileged unless the Embedded Cache Locking.User Mode category is supported. If the Embedded Cache Locking.User Mode category is supported, this instruction is privileged only if MSR_{UCLE}=0.

Special Registers Altered:
None

Data Cache Block Lock Clear X-form

dcblc CT,RA,RB

31	/	CT	RA	RB	390	/
0	6	7	11	16	21	31

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

The block containing the byte addressed by EA in the data cache specified by the CT field is unlocked.

The instruction is treated as a *Load*.

An unable-to-unlock condition may occur (see Section 4.9.2.2.2). If the block containing the byte addressed by EA is not locked in the specified cache, no cache operation is performed.

This instruction is privileged unless the Embedded Cache Locking.User Mode category is supported. If the Embedded Cache Locking.User Mode category is supported, this instruction is privileged only if MSR_{UCLE}=0.

Special Registers Altered:
None

Programming Note

The ***dcblc*** and ***icblc*** instructions are used to remove locks previously set by the corresponding lock set instructions.

4.9.3 Synchronize Instruction

The *Synchronize* instruction is described in Section 3.3.3 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 programmers.

In conjunction with the ***tlbie*** and ***tlbsync*** instructions, the ***sync*** instruction provides an ordering function for TLB invalidations and related storage accesses on other processors as described in the ***tlbsync*** instruction description on page 561.

4.9.4 Lookaside Buffer Management

All implementations include a TLB as the architected repository of translation, protection, and attribute information for storage.

Each implementation that has a TLB or similar lookaside buffer provides a means by which software can invalidate the lookaside entry that translates a given effective address.

Programming Note

The invalidate all entries function is not required because each TLB entry can be addressed directly without regard to the contents of the entry.

In addition, implementations provide a means by which software can do the following.

- Read a specified TLB entry
- Identify the TLB entry (if any) associated with a specified effective address
- Write a specified TLB entry

Programming Note

Because the presence, absence, and exact semantics of the *TLB Management* instructions are implementation-dependent, it is recommended that system software “encapsulate” uses of these instructions into subroutines to minimize the impact of moving from one implementation to another.

4.9.4.1 TLB Management Instructions

The ***tlbivax*** instruction is used to invalidate TLB entries. Additional instructions are used to read and write, and search TLB entries, and to provide an ordering function for the effects of ***tlbivax***

TLB Invalidate Virtual Address Indexed X-form

tlbivax (implementation dependent)

31	???	???	???	786	/
0	6	11	16	21	31

Bits 6:20 of the instruction encoding are implementation-dependent, and may be used to specify the TLB entry or entries to be invalidated. (E.g. they may specify virtual or effective addresses.)

If a single ***tlbivax*** instruction can invalidate more entries than those corresponding to a single VA, a means must be provided to prevent specific TLB entries from being invalidated.

If the Translation Lookaside Buffer (TLB) contains an entry specified, the entry or entries are made invalid (i.e. removed from the TLB). This instruction causes the target TLB entry to be invalidated in all processors.

If the instruction specifies a TLB entry that does not exist, the results are undefined.

Execution of this instruction may cause other implementation-dependent effects.

The operation performed by this instruction is ordered by the ***mbar*** (or ***sync***) instruction with respect to a subsequent ***tlbsync*** instruction executed by the processor executing the ***tlbivax*** instruction. The operations caused by ***tlbivax*** and ***tlbsync*** are ordered by ***mbar*** as a set of operations which is independent of the other sets that ***mbar*** orders.

This instruction is privileged.

Special Registers Altered:
None

Programming Note

The effects of the invalidation may not be visible until the completion of a context synchronizing operation (see Section 1.6.1).

Programming Note

Care must be taken not to invalidate any TLB entry that contains the mapping for any interrupt vector.

TLB Read Entry X-form

tlbre (implementation dependent)

31	???	???	???	946	/
0	6	11	16	21	31

Bits 6:20 of the instruction encoding are implementation-dependent, and may be used to specify the source TLB entry, the source portion of the source TLB entry, and the target resource that the result is placed into.

The implementation-dependent-specified TLB entry is read, and the implementation-dependent-specified portion of the TLB entry is extracted and placed into an implementation-dependent target resource.

If the instruction specifies a TLB entry that does not exist, the results are undefined.

Execution of this instruction may cause other implementation-dependent effects.

This instruction is privileged.

Special Registers Altered:
Implementation-dependent

TLB Search Indexed X-form

tlbsx RA,RB, (implementation dependent)

31	???	RA	RB	914	?
0	6	11	16	21	31

```
if RA=0 then b ← 0
else         b ← (RA)
EA ← b + (RB)
AS        ← implementation-dependent value
ProcessID ← implementation-dependent value
VA        ← AS || ProcessID || EA
If there is a TLB entry for which TLBentry_VA=VA
   then result ← implementation-dependent value
   else result ← undefined
target resource(???) ← result
```

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

Let address space (AS) be defined as implementation-dependent (e.g. could be MSR_{DS} or a bit from an implementation-dependent SPR).

Let the ProcessID be defined as implementation-dependent (e.g. could be from the PID register or from an implementation-dependent SPR).

Let the virtual address (VA) be the value AS || ProcessID || EA. See Figure 9 on page 547.

Bits 6:10 of the instruction encoding are implementation-dependent, and may be used to specify the target resource that the result of the instruction is placed into.

If the Translation Lookaside Buffer (TLB) contains an entry corresponding to VA, an implementation-dependent value is placed into an implementation-dependent-specified target. Otherwise the contents of the implementation-dependent-specified target are left undefined.

Bit 31 of the instruction encoding is implementation-dependent. For example, bit 31 may be interpreted as an "Rc" bit, used to enable recording the success or failure of the search operation.

This instruction is privileged.

Special Registers Altered:
None

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 ***tlbivax*** instructions executed by the processor executing the ***tlbsync*** instruction, with respect to the memory barrier created by a subsequent ***sync*** instruction executed by the same processor. Executing a ***tlbsync*** instruction ensures that all of the following will occur.

- All storage accesses by other processors for which the address was translated using the translations being invalidated will have been performed with respect to the processor executing the ***sync*** instruction, to the extent required by the associated Memory Coherence Required attributes, before the ***sync*** instruction's memory barrier is created.

The operation performed by this instruction is ordered by the ***mbar*** or ***msync*** instruction with respect to preceding ***tlbivax*** instructions executed by the processor executing the ***tlbsync*** instruction. The operations caused by ***tlbivax*** and ***tlbsync*** are ordered by ***mbar*** as a set of operations, which is independent of the other sets that ***mbar*** orders.

The ***tlbsync*** instruction may complete before operations caused by ***tlbivax*** instructions preceding the ***tlbsync*** instruction have been performed.

This instruction is privileged.

Special Registers Altered:
None

TLB Write Entry *X-form*

tlbwe (implementation dependent)

31	???	???	???	978	/
0	6	11	16	21	31

Bits 6:20 of the instruction encoding are implementation-dependent, and may be used to specify the target TLB entry, the target portion of the target TLB entry, and the source of the value that is to be written into the TLB.

The contents of the implementation-dependent-specified source are written into the implementation-dependent-specified portion of the implementation-dependent-specified TLB entry.

If the instruction specifies a TLB entry that does not exist, the results are undefined.

Execution of this instruction may cause other implementation-dependent effects.

This instruction is privileged.

Special Registers Altered:
Implementation-dependent

Programming Note

The effects of the update may not be visible until the completion of a context synchronizing operation (see Section 1.6.1).

Programming Note

Care must be taken not to invalidate any TLB entry that contains the mapping for any interrupt vector.

Chapter 5. Interrupts and Exceptions

5.1 Overview

An *interrupt* is the action in which the processor saves its old context (MSR and next instruction address) and begins execution at a pre-determined interrupt-handler address, with a modified MSR. *Exceptions* are the events that will, if enabled, cause the processor to take an interrupt.

Exceptions are generated by signals from internal and external peripherals, instructions, the internal timer facility, debug events, or error conditions.

Interrupts are divided into 4 classes, as described in Section 5.4.3, such that only one interrupt of each class is reported, and when it is processed no program state is lost. Since Save/Restore register pairs SRR0/SRR1, CSRR0/CSRR1, DSRR0/DSRR1 [Category: E.ED], and MCSSR0/MCSSR1 are serially reusable resources used by base, critical, debug [Category: E.ED], Machine Check interrupts, respectively, program state may be lost when an unordered interrupt is taken. (See Section 5.8.

All interrupts, except Machine Check, are context synchronizing as defined in Section 1.6.1 on page 511. A Machine Check interrupt acts like a context synchronizing operation with respect to subsequent instructions; that is, a Machine Check interrupt need not satisfy items 2-3 of Section 1.6.1 but does satisfy items 1, 4, and 5.

5.2 Interrupt Registers

5.2.1 Save/Restore Register 0

Save/Restore Register 0 (SRR0) is a 64-bit register. SRR0 bits are numbered 0 (most-significant bit) to 63 (least-significant bit). The register is used to save machine state on non-critical interrupts, and to restore machine state when an ***rfi*** is executed. On a non-critical interrupt, SRR0 is set to the current or next instruction address. When ***rfi*** is executed, instruction execution continues at the address in SRR0.

In general, SRR0 contains the address of the instruction that caused the non-critical interrupt, or the address of the instruction to return to after a non-critical interrupt is serviced.

The contents of SRR0 when an interrupt is taken are mode dependent, reflecting the computation mode currently in use (specified by MSR_{CM}) and the computation mode entered for execution of the interrupt (specified by MSR_{ICM}). The contents of SRR0 upon interrupt can be described as follows (assuming Addr is the address to be put into SRR0):

$$\text{if } (MSR_{CM} = 0) \; \& \; (MSR_{ICM} = 0) \text{ then } SRR0 \leftarrow {}^{32}\text{undefined} \parallel Addr_{32:63}$$
$$\text{if } (MSR_{CM} = 0) \; \& \; (MSR_{ICM} = 1) \text{ then } SRR0 \leftarrow {}^{32}0 \parallel Addr_{32:63}$$
$$\text{if } (MSR_{CM} = 1) \; \& \; (MSR_{ICM} = 1) \text{ then } SRR0 \leftarrow Addr_{0:63}$$
$$\text{if } (MSR_{CM} = 1) \; \& \; (MSR_{ICM} = 0) \text{ then } SRR0 \leftarrow \text{undefined}$$

The contents of SRR0 can be read into register RT using ***mfspr*** *RT,SRR0*. The contents of register RS can be written into the SRR0 using ***mtspr*** *SRR0,RS*.

5.2.2 Save/Restore Register 1

Save/Restore Register 1 (SRR1) is a 32-bit register. SRR1 bits are numbered 32 (most-significant bit) to 63 (least-significant bit). The register is used to save machine state on non-critical interrupts, and to restore machine state when an ***rfi*** is executed. When a non-critical interrupt is taken, the contents of the MSR are placed into SRR1. When ***rfi*** is executed, the contents of SRR1 are placed into the MSR.

Bits of SRR1 that correspond to reserved bits in the MSR are also reserved.

Programming Note

A MSR bit that is reserved may be inadvertently modified by ***rfi*/*rfci*/*rfmci***.

The contents of SRR1 can be read into register RT using ***mfspr*** *RT,SRR1*. The contents of register RS can be written into the SRR1 using ***mtspr*** *SRR1,RS*.

5.2.3 Critical Save/Restore Register 0

Critical Save/Restore Register 0 (CSRR0) is a 64-bit register. CSRR0 bits are numbered 0 (most-significant bit) to 63 (least-significant bit). The register is used to save machine state on critical interrupts, and to restore machine state when an ***rfci*** is executed. When a critical interrupt is taken, the CSRR0 is set to the current or next instruction address. When ***rfci*** is executed, instruction execution continues at the address in CSRR0.

In general, CSRR0 contains the address of the instruction that caused the critical interrupt, or the address of the instruction to return to after a critical interrupt is serviced.

The contents of CSRR0 when a critical interrupt is taken are mode dependent, reflecting the computation mode currently in use (specified by MSR_{CM}) and the computation mode entered for execution of the critical interrupt (specified by MSR_{ICM}). The contents of CSRR0 upon critical interrupt can be described as follows (assuming Addr is the address to be put into CSRR0):

if (MSR_{CM} = 0) & (MSR_{ICM} = 0)
   then CSRR0 ← ^{32}undefined ‖ $Addr_{32:63}$
if (MSR_{CM} = 0) & (MSR_{ICM} = 1)
   then CSRR0 ← ^{32}0 ‖ $Addr_{32:63}$
if (MSR_{CM} = 1) & (MSR_{ICM} = 1) then CSRR0 ← $Addr_{0:63}$
if (MSR_{CM} = 1) & (MSR_{ICM} = 0) then CSRR0 ← undefined

The contents of CSRR0 can be read into register RT using ***mfspr*** *RT,CSRR0*. The contents of register RS can be written into CSRR0 using ***mtspr*** *CSRR0,RS*.

5.2.4 Critical Save/Restore Register 1

Critical Save/Restore Register 1 (CSRR1) is a 32-bit register. CSRR1 bits are numbered 32 (most-significant bit) to 63 (least-significant bit). The register is used to save machine state on critical interrupts, and to restore machine state when an ***rfci*** is executed. When a critical interrupt is taken, the contents of the MSR are placed into CSRR1. When ***rfci*** is executed, the contents of CSRR1 are placed into the MSR.

Bits of CSRR1 that correspond to reserved bits in the MSR are also reserved.

Programming Note

A MSR bit that is reserved may be inadvertently modified by ***rfi/rfci/rfmci***.

The contents of CSRR1 can be read into bits 32:63 of register RT using ***mfspr*** *RT,CSRR1*, setting bits 0:31 of RT to zero. The contents of bits 32:63 of register RS can be written into the CSRR1 using ***mtspr*** *CSRR1,RS*.

5.2.5 Debug Save/Restore Register 0 [Category: Embedded.Enhanced Debug]

Debug Save/Restore Register 0 (DSRR0) is a 64-bit register used to save machine state on Debug interrupts, and to restore machine state when an ***rfdi*** is executed. When a Debug interrupt is taken, the DSRR0 is set to the current or next instruction address. When ***rfdi*** is executed, instruction execution continues at the address in DSRR0.

In general, DSRR0 contains the address of an instruction that was executing or just finished execution when the Debug exception occurred.

The contents of DSRR0 when a Debug interrupt is taken are mode dependent, reflecting the computation mode currently in use (specified by MSR_{CM}) and the computation mode entered for execution of the Debug interrupt (specified by MSR_{ICM}). The contents of DSRR0 upon Debug interrupt can be described as follows (assuming Addr is the address to be put into DSRR0):

if (MSR_{CM} = 0) & (MSR_{ICM} = 0) then DSRR0 ← ^{32}undefined ‖ $Addr_{32:63}$
if (MSR_{CM} = 0) & (MSR_{ICM} = 1) then DSRR0 ← ^{32}0 ‖ $Addr_{32:63}$
if (MSR_{CM} = 1) & (MSR_{ICM} = 1) then DSRR0 ← $Addr_{0:63}$
if (MSR_{CM} = 1) & (MSR_{ICM} = 0) then DSRR0 ← undefined

The contents of DSRR0 can be read into register RT using ***mfspr*** RT,DSRR0. The contents of register RS can be written into DSRR0 using ***mtspr*** DSRR0,RS.

5.2.6 Debug Save/Restore Register 1 [Category: Embedded.Enhanced Debug]

Debug Save/Restore Register 1 (DSRR1) is a 32-bit register used to save machine state on Debug interrupts, and to restore machine state when an ***rfdi*** is executed. When a Debug interrupt is taken, the contents of the Machine State Register are placed into DSRR1. When ***rfdi*** is executed, the contents of DSRR1 are placed into the Machine State Register.

Bits of DSRR1 that correspond to reserved bits in the Machine State Register are also reserved.

The contents of DSRR1 can be read into bits 32:63 of register RT using ***mfspr*** RT,DSRR1, setting bits 0:31 of RT to zero. The contents of bits 32:63 of register RS can be written into the DSSR1 using ***mtspr*** DSRR1,RS.

5.2.7 Data Exception Address Register

The Data Exception Address Register (DEAR) is a 64-bit register. DEAR bits are numbered 0 (most-significant bit) to 63 (least-significant bit). The DEAR contains the address that was referenced by a *Load*, *Store* or *Cache Management* instruction that caused an Alignment, Data TLB Miss, or Data Storage interrupt.

The contents of the DEAR when an interrupt is taken are mode dependent, reflecting the computation mode currently in use (specified by MSR_{CM}) and the computation mode entered for execution of the critical interrupt (specified by MSR_{ICM}). The contents of the DEAR upon interrupt can be described as follows (assuming Addr is the address to be put into DEAR):

```
if (MSR_CM = 0) & (MSR_ICM = 0)
   then DEAR ← ^32undefined || Addr_32:63
if (MSR_CM = 0) & (MSR_ICM = 1)
   then DEAR ← ^32 0 || Addr_32:63
if (MSR_CM = 1) & (MSR_ICM = 1) then DEAR ← Addr_0:63
if (MSR_CM = 1) & (MSR_ICM = 0) then DEAR ← undefined
```

The contents of DEAR can be read into register RT using ***mtspr*** *RT,DEAR*. The contents of register RS can be written into the DEAR using ***mtspr*** *DEAR,RS*.

5.2.8 Interrupt Vector Prefix Register

The Interrupt Vector Prefix Register (IVPR) is a 64-bit register. Interrupt Vector Prefix Register bits are numbered 0 (most-significant bit) to 63 (least-significant bit). Bits 48:63 are reserved. Bits 0:47 of the Interrupt Vector Prefix Register provides the high-order 48 bits of the address of the exception processing routines. The 16-bit exception vector offsets (provided in Section 5.2.10) are concatenated to the right of bits 0:47 of the Interrupt Vector Prefix Register to form the 64-bit address of the exception processing routine.

The contents of Interrupt Vector Prefix Register can be read into register RT using ***mfspr*** *RT,IVPR*. The contents of register RS can be written into Interrupt Vector Prefix Register using ***mtspr*** *IVPR,RS*.

5.2.9 Exception Syndrome Register

The Exception Syndrome Register (ESR) is a 32-bit register. ESR bits are numbered 32 (most-significant bit) to 63 (least-significant bit). The ESR provides a *syndrome* to differentiate between the different kinds of exceptions that can generate the same interrupt type. Upon the generation of one of these types of interrupts, the bit or bits corresponding to the specific exception that generated the interrupt is set, and all other ESR bits are cleared. Other interrupt types do not affect the contents of the ESR. The ESR does not need to be cleared by software. Figure 12 shows the bit definitions for the ESR.

Bit(s)	Name	Meaning	Associated Interrupt Type
32:35	Implementation-dependent		(Implementation-dependent)
36	PIL	Illegal Instruction exception	Program
37	PPR	Privileged Instruction exception	Program
38	PTR	Trap exception	Program
39	FP	Floating-point operation	Alignment Data Storage Data TLB Program
40	ST	Store operation	Alignment Data Storage Data TLB Error
41	Reserved		
42 43	DLK_0 DLK_1	(Implementation-dependent) (implementation-dependent)	(Implementation-dependent) (Implementation-dependent)
44	AP	Auxiliary Processor operation	Alignment Data Storage Data TLB Program
45	PUO	Unimplemented Operation exception	Program
46	BO	Byte Ordering exception	Data Storage Instruction Storage
47	PIE	Imprecise exception	Program
48:55	Reserved		
56	SPV	Signal Processing operation [Category: Signal Processing Engine] Vector operation [Category: Vector]	Alignment Data Storage Data TLB Embedded Floating-point Data Embedded Floating-point Round SPE/Embedded Floating-point/Vector Unavailable
57	EPID	External Process ID operation [Category: Embedded.External Process ID]	Alignment Data Storage Data TLB
58	VLEMI	VLE operation [Category: VLE]	Alignment Data Storage Data TLB SPE/Embedded Floating-point/Vector Unavailable Embedded Floating-point Data Embedded Floating-point Round Instruction Storage Program System Call
59:61	Implementation-dependent		(Implementation-dependent)
62	MIF	Misaligned Instruction [Category: VLE]	Instruction TLB Instruction Storage

Figure 12. Exception Syndrome Register Definitions

Programming Note

The information provided by the ESR is not complete. System software may also need to identify the type of instruction that caused the interrupt, examine the TLB entry accessed by a data or instruction storage access, as well as examine the ESR to fully determine what exception or exceptions caused the interrupt. For example, a Data Storage interrupt may be caused by both a Protection Violation exception as well as a Byte Ordering exception. System software would have to look beyond ESR_{BO}, such as the state of MSR_{PR} in SRR1 and the page protection bits in the TLB entry accessed by the storage access, to determine whether or not a Protection Violation also occurred.

The contents of the ESR can be read into bits 32:63 of register RT using *mfspr RT,ESR*, setting bits 0:31 of RT to zero. The contents of bits 32:63 of register RS can be written into the ESR using *mtspr ESR,RS*.

5.2.10 Interrupt Vector Offset Registers

The Interrupt Vector Offset Registers (IVORs) are 32-bit registers. Interrupt Vector Offset Register bits are numbered 32 (most-significant bit) to 63 (least-significant bit). Bits 32:47 and bits 60:63 are reserved. An Interrupt Vector Offset Register provides the quadword index from the base address provided by the IVPR (see Section 5.2.8) for its respective interrupt. Interrupt Vector Offset Registers 0 through 15 and 32-37 are provided for the defined interrupts. SPR numbers corresponding to Interrupt Vector Offset Registers 16 through 31 are reserved. SPR numbers corresponding to Interrupt Vector Offset Registers 38 through 63 are allocated for implementation-dependent use. Figure 13 provides the assignments of specific Interrupt Vector Offset Registers to specific interrupts.

IVORi	Interrupt
IVOR0	Critical Input
IVOR1	Machine Check
IVOR2	Data Storage
IVOR3	Instruction Storage
IVOR4	External
IVOR5	Alignment
IVOR6	Program
IVOR7	Floating-Point Unavailable
IVOR8	System Call
IVOR9	Auxiliary Processor Unavailable
IVOR10	Decrementer
IVOR11	Fixed-Interval Timer Interrupt
IVOR12	Watchdog Timer Interrupt
IVOR13	Data TLB Error
IVOR14	Instruction TLB Error
IVOR15	Debug
IVOR16 : IVOR31	Reserved
[Category: Signal Processing Engine] [Category: Vector]	
IVOR 32	SPE/Embedded Floating-Point/Vector Unavailable Interrupt
[Category: SP.Embedded Float_*] (IVORs 33 & 34 are required if any SP.Float_ dependent category is supported.)	
IVOR 33	Embedded Floating-Point Data Interrupt
IVOR 34	Embedded Floatg.-pt. round Interrupt
[Category: Embedded Performance Monitor]	
IVOR 35	Embedded Performance Monitor Interrupt
[Category: Embedded.Processor Control]	
IVOR 36	Processor Doorbell Interrupt
IVOR 37	Processor Doorbell Critical Interrupt
IVOR38 : IVOR63	Implementation-dependent

Figure 13. Interrupt Vector Offset Register Assignments

Bits 48:59 of the contents of IVORi can be read into bits 48:59 of register RT using *mfspr RT,IVORi*, setting bits 0:47 and bits 60:63 of GPR(RT) to zero. Bits 48:59 of the contents of register RS can be written into bits 48:59 of IVORi using *mtspr IVORi,RS*.

5.2.11 Machine Check Registers

A set of Special Purpose Registers are provided to support Machine Check interrupts.

5.2.11.1 Machine Check Save/Restore Register 0

Machine Check Save/Restore Register 0 (MCSRR0) is used to save machine state on Machine Check interrupts, and to restore machine state when an ***rfmci*** is executed. When a Machine Check interrupt is taken, the MCSRR0 is set to the current or next instruction address. When ***rfmci*** is executed, instruction execution continues at the address in MCSRR0.

In general, MCSRR0 contains the address of an instruction that was executing or about to be executed when the Machine Check exception occurred.

The contents of MCSRR0 when a Machine Check interrupt is taken are mode dependent, reflecting the computation mode currently in use (specified by MSR_{CM}) and the computation mode entered for execution of the Machine Check interrupt (specified by MSR_{ICM}). The contents of MCSRR0 upon Machine Check interrupt can be described as follows (assuming Addr is the address to be put into MCSRR0):

if (MSR_{CM} = 0) & (MSR_{ICM} = 0)
   then MCSRR0 ← ^{32}undefined || $Addr_{32:63}$
if (MSR_{CM} = 0) & (MSR_{ICM} = 1)
   then MCSRR0 ← ^{32}0 || $Addr_{32:63}$
if (MSR_{CM} = 1) & (MSR_{ICM} = 1) then MCSRR0 ← $Addr_{0:63}$
if (MSR_{CM} = 1) & (MSR_{ICM} = 0) then MCSRR0 ← undefined

The contents of MCSRR0 can be read into register RT using *mfspr RT,MCSRR0*. The contents of register RS can be written into MCSRR0 using *mtspr MCSRR0,RS*.

5.2.11.2 Machine Check Save/Restore Register 1

Machine Check Save/Restore Register 1 (MCSRR1) is used to save machine state on Machine Check interrupts, and to restore machine state when an ***rfmci*** is executed. When a Machine Check interrupt is taken, the contents of the MSR are placed into MCSRR1. When ***rfmci*** is executed, the contents of MCSRR1 are placed into the MSR.

Bits of MCSRR1 that correspond to reserved bits in the MSR are also reserved.

Programming Note

A MSR bit that is reserved may be inadvertently modified by ***rfi***/***rfci***/***rfmci***.

The contents of MCSRR1 can be read into register RT using *mfspr RT,MCSRR1*. The contents of register RS can be written into the MCSRR1 using *mtspr MCSRR1,RS*.

5.2.11.3 Machine Check Syndrome Register

MCSR (MCSR) is a 64-bit register that is used to record the cause of the Machine Check interrupt. The specific definition of the contents of this register are implementation-dependent (see the User Manual of the implementation).

The contents of MCSR can be read into register RT using *mfspr RT,MCSR*. The contents of register RS can be written into the MCSR using *mtspr MCSR,RS*.

5.2.12 External Proxy Register [Category: External Proxy]

The External Proxy Register (EPR) contains implementation-dependent information related to an External Input interrupt when an External Input interrupt occurs. The EPR is only considered valid from the time that the External Input Interrupt occurs until MSR_{EE} is set to 1 as the result of a ***mtmsr*** or a return from interrupt instruction.

The format of the EPR is shown below.

EPR
32 … 63

Figure 14. External Proxy Register

When the External Input interrupt is taken, the contents of the EPR provide information related to the External Input Interrupt.

Programming Note

The EPR is provided for faster interrupt processing as well as situations where an interrupt must be taken, but software must delay the resultant processing for later.

The EPR contains the vector from the interrupt controller. The process of receiving the interrupt into the EPR acknowledges the interrupt to the interrupt controller. The method for enabling or disabling the acknowledgment of the interrupt by placing the interrupt-related information in the EPR is implementation-dependent. If this acknowledgement is disabled, then the EPR is set to 0 when the External Input interrupt occurs.

5.3 Exceptions

There are two kinds of exceptions, 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 types of interrupts to be invoked.

Examples of exceptions that can be caused directly by the execution of an instruction include but are not limited to the following:

- an attempt to execute a reserved-illegal instruction (Illegal Instruction exception type Program interrupt)
- an attempt by an application program to execute a 'privileged' instruction (Privileged Instruction exception type Program interrupt)
- an attempt by an application program to access a 'privileged' Special Purpose Register (Privileged Instruction exception type Program interrupt)
- an attempt by an application program to access a Special Purpose Register that does not exist (Unimplemented Operation Instruction exception type Program interrupt)
- an attempt by a system program to access a Special Purpose Register that does not exist (boundedly undefined results)
- the execution of a defined instruction using an invalid form (Illegal Instruction exception type Program interrupt, Unimplemented Operation exception type Program interrupt, or Privileged Instruction exception type Program interrupt)
- an attempt to access a storage location that is either unavailable (Instruction TLB Error interrupt or Data TLB Error interrupt) or not permitted (Instruction Storage interrupt or Data Storage interrupt)
- an attempt to access storage with an effective address alignment not supported by the implementation (Alignment interrupt)
- the execution of a *System Call* instruction (System Call interrupt)
- the execution of a *Trap* instruction whose trap condition is met (Trap type Program interrupt)
- the execution of a floating-point instruction when floating-point instructions are unavailable (Floating-point Unavailable interrupt)
- the execution of a floating-point instruction that causes a floating-point enabled exception to exist (Enabled exception type Program interrupt)
- the execution of a defined instruction that is not implemented by the implementation (Illegal Instruction exception or Unimplemented Operation exception type of Program interrupt)
- the execution of an instruction that is not implemented by the implementation (Illegal Instruction exception or Unimplemented Operation exception type of Program interrupt)
- the execution of an auxiliary processor instruction when the auxiliary processor instruction is unavailable (Auxiliary Processor Unavailable interrupt)
- the execution of an instruction that causes an auxiliary processor enabled exception (Enabled exception type Program interrupt)

The invocation of an interrupt is precise, except that if one of the imprecise modes for invoking the Floating-point Enabled Exception type Program interrupt is in effect then the invocation of the Floating-point Enabled Exception type Program interrupt may be imprecise. When the interrupt 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 interrupt, has not yet occurred).

5.4 Interrupt Classification

All interrupts, except for Machine Check, can be classified as either Asynchronous or Synchronous. Independent from this classification, all interrupts, including Machine Check, can be classified into one of the following classes:

- Base
- Critical
- Machine Check
- Debug[Category:Embedded.Enhanced Debug].

5.4.1 Asynchronous Interrupts

Asynchronous interrupts are caused by events that are independent of instruction execution. For asynchronous interrupts, the address reported to the exception handling routine is the address of the instruction that would have executed next, had the asynchronous interrupt not occurred.

5.4.2 Synchronous Interrupts

Synchronous interrupts are those that are caused directly by the execution (or attempted execution) of instructions, and are further divided into two classes, *precise* and *imprecise*.

Synchronous, precise interrupts are those that *precisely* indicate the address of the instruction causing the exception that generated the interrupt; or, for certain synchronous, precise interrupt types, the address of the immediately following instruction.

Synchronous, imprecise interrupts are those that may indicate the address of the instruction causing the exception that generated the interrupt, or some instruction after the instruction causing the exception.

5.4.2.1 Synchronous, Precise Interrupts

When the execution or attempted execution of an instruction causes a synchronous, precise interrupt, the following conditions exist at the interrupt point.

- SRR0, CSRR0, or DSRR0 [Category: Embedded.Enhanced Debug] addresses either the instruction causing the exception or the instruction immediately following the instruction causing the exception. Which instruction is addressed can be determined from the interrupt type and status bits.
- An interrupt is generated such that all instructions preceding the instruction causing the exception appear to have completed with respect to the executing processor. However, some storage accesses associated with these preceding instructions may not have been performed with respect to other processors and mechanisms.
- 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. See Section 5.7 on page 588.
- Architecturally, no subsequent instruction has executed beyond the instruction causing the exception.

5.4.2.2 Synchronous, Imprecise Interrupts

When the execution or attempted execution of an instruction causes an imprecise interrupt, the following conditions exist at the interrupt point.

- SRR0 or CSRR0 addresses either the instruction causing the exception or some instruction following the instruction causing the exception that generated the interrupt.
- An interrupt is generated such that all instructions preceding the instruction addressed by SRR0 or CSRR0 appear to have completed with respect to the executing processor.
- If the imprecise interrupt is forced by the context synchronizing mechanism, due to an instruction that causes another exception that generates an interrupt (e.g., Alignment, Data Storage), then SRR0 addresses the interrupt-forcing instruction, and the interrupt-forcing instruction may have been partially executed (see Section 5.7 on page 588).
- If the imprecise interrupt is forced by the execution synchronizing mechanism, due to executing an execution synchronizing instruction other than ***msync*** or ***isync***, then SRR0 or CSRR0 addresses the interrupt-forcing instruction, and the interrupt-forcing instruction appears not to have begun execution (except for its forcing the imprecise interrupt). If the imprecise interrupt is forced by an ***msync*** or ***isync*** instruction, then SRR0 or CSRR0 may address either the ***msync*** or ***isync*** instruction, or the following instruction.
- If the imprecise interrupt is not forced by either the context synchronizing mechanism or the execution synchronizing mechanism, then the instruction addressed by SRR0 or CSRR0 may have been partially executed (see Section 5.7 on page 588).
- No instruction following the instruction addressed by SRR0 or CSRR0 has executed.

5.4.3 Interrupt Classes

Interrupts can also be classified as base, critical, Machine Check, and Debug [Category: Embedded.Enhanced Debug].

Interrupt classes other than the base class may demand immediate attention even if another class of interrupt is currently being processed and software has not yet had the opportunity to save the state of the machine (i.e. return address and captured state of the MSR). For this reason, the interrupts are organized into a hierarchy (see Section 5.8). To enable taking a critical, Machine Check, or Debug [Category: Embedded.Enhanced Debug] interrupt immediately after a base class interrupt occurs (i.e. before software has saved the state of the machine), these interrupts use the Save/Restore Register pair CSRR0/CSRR1, MCSRR0/MCSRR1, or DSRR0/DSRR1 [Category: Embedded.Enhanced Debug], and base class interrupts use Save/Restore Register pair SRR0/SRR1.

5.4.4 Machine Check Interrupts

Machine Check interrupts are a special case. They are typically caused by some kind of hardware or storage subsystem failure, or by an attempt to access an invalid address. A Machine Check may be caused indirectly by the execution of an instruction, but not be recognized and/or reported until long after the processor has executed past the instruction that caused the Machine Check. As such, Machine Check interrupts cannot properly be thought of as synchronous or asynchronous, nor as precise or imprecise. The following general rules apply to Machine Check interrupts:

1. No instruction after the one whose address is reported to the Machine Check interrupt handler in MCSRR0 has begun execution.

2. The instruction whose address is reported to the Machine Check interrupt handler in MCSRR0, and all prior instructions, may or may not have completed successfully. All those instructions that are ever going to complete appear to have done so already, and have done so within the context existing prior to the Machine Check interrupt. No further interrupt (other than possible additional Machine

Check interrupts) will occur as a result of those instructions.

5.5 Interrupt Processing

Associated with each kind of interrupt is an *interrupt vector*, that is the address of the initial instruction that is executed when the corresponding interrupt occurs.

Interrupt processing consists of saving a small part of the processor's state in certain registers, identifying the cause of the interrupt in another register, 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 can be taken, the following actions are performed, in order:

1. SRR0, DSRR0 [Category: Embedded.Enhanced Debug], MCSRR0, or CSRR0 is loaded with an instruction address that depends on the interrupt; see the specific interrupt description for details.
2. The ESR is loaded with information specific to the exception. Note that many interrupts can only be caused by a single kind of exception event, and thus do not need nor use an ESR setting to indicate to the cause of the interrupt was.
3. SRR1, DSRR1 [Category: Embedded.Enhanced Debug], or MCSRR1, or CSRR1 is loaded with a copy of the contents of the MSR.
4. The MSR is updated as described below. The new values take effect beginning with the first instruction following the interrupt. MSR bits of particular interest are the following.
   - $MSR_{WE,EE,PR,FP,FE0,FE1,IS,DS}$ are set to 0 by all interrupts.
   - MSR_{ME} is set to 0 by Machine Check interrupts and left unchanged by all other interrupts.
   - MSR_{CE} is set to 0 by critical class interrupts, Debug interrupts, and Machine Check interrupts, and is left unchanged by all other interrupts.
   - MSR_{DE} is set to 0 by critical class interrupts unless Category E.ED is supported, by Debug interrupts, and by Machine Check interrupts, and is left unchanged by all other interrupts.
   - MSR_{CM} is set to MSR_{ICM}.
   - Other supported MSR bits are left unchanged by all interrupts.

   See Section 2.2.1 for more detail on the definition of the MSR.
5. Instruction fetching and execution resumes, using the new MSR value, at a location specific to the interrupt. The location is

   $IVPR_{0:47}$ || $IVORi_{48:59}$ || 0b0000

   where IVPR is the Interrupt Vector Prefix Register and IVORi is the Interrupt Vector Offset Register for that interrupt (see Figure 13 on page 568). The contents of the Interrupt Vector Prefix Register and Interrupt Vector Offset Registers are indeterminate upon power-on reset, and must be initialized by system software using the ***mtspr*** instruction.

Interrupts may not clear reservations obtained with *Load and Reserve* instructions. The operating system should do so at appropriate points, such as at process switch.

At the end of an interrupt handling routine, execution of an ***rfi, rfdi*** [Category: Embedded.Enhanced Debug], ***rfmci***, or ***rfci*** causes the MSR to be restored from the contents of SRR1, DSRR1 [Category: Embedded.Enhanced Debug], MCSRR1, or CSRR1, and instruction execution to resume at the address contained in SRR0, DSRR0 [Category: Embedded.Enhanced Debug], MCSRR0, or CSRR0, respectively.

Programming Note

In general, at process switch, due to possible process interlocks and possible data availability requirements, the operating system needs to consider executing the following.

- ***stwcx.*** or ***stdcx.***, to clear the reservation if one is outstanding, to ensure that a ***lwarx*** or ***ldarx*** in the "old" process is not paired with a ***stwcx.*** or ***stdcx.*** in the "new" process.
- ***msync***, to ensure that all storage operations of an interrupted process are complete with respect to other processors before that process begins executing on another processor.
- ***isync***, ***rfi***, ***rfdi*** [Category: Embedded.Enhanced Debug], ***rfmci***, or ***rfci*** to ensure that the instructions in the "new" process execute in the "new" context.

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, processor 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*** or ***dcbzep*** 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: Illegal Instruction type Program interrupt caused by an instruction that has been phased out of the architecture but is still used by some programs that the operating system supports, or by an instruction that is in a category that the implementation does not support but is used by some programs that the operating system supports.

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 TLB entry.)

5.6 Interrupt Definitions

Table 15 provides a summary of each interrupt type, the various exception types that may cause that interrupt type, the classification of the interrupt, which ESR bits can be set, if any, which MSR bits can mask the interrupt type and which Interrupt Vector Offset Register is used to specify that interrupt type's vector address.

IVOR	Interrupt	Exception	Asynchronous	Synchronous, Precise	Synchronous, Imprecise	Critical	ESR (See Note 5)	MSR Mask Bit(s)	DBCR0/TCR Mask Bit	Category (Section 1.3.5 of Book I)	Notes (see page 575)	Page
IVOR0	Critical Input	Critical Input	x			x		CE		E	1	576
IVOR1	Machine Check	Machine Check						ME		E	2,4	576
IVOR2	Data Storage	Access		x			[ST],[FP,AP,SPV] [VLEMI], [EPID]			E	9	577
		Load and Reserve or *Store Conditional* to 'write-thru required' storage (W=1)		x			[ST], [VLEMI]			E	9	
		Cache Locking		x			{DLK_0,DLK_1},[ST] [VLEMI]			E	8	
		Byte Ordering		x			BO, [ST], [FP,AP,SPV], [VLEMI], [EPID]			E		
IVOR3	Inst Storage	Access		x						E		578
		Byte Ordering		x			BO, [VLEMI]			E		
		Mismatched Instruction Storage (See Book VLE.))		x			BO, VLEMI	EE		E, VLE	1	
		Misaligned Instruction Storage (See Book VLE.)		x			MIF	EE		E, VLE	1	
IVOR4	External Input	External Input	x					EE		E	1	578
IVOR5	Alignment	Alignment		x			[ST],[FP,AP,SPV] [EPID],[VLEMI]			E		579
IVOR6	Program	Illegal		x			PIL, [VLEMI]			E		580
		Privileged		x			PPR,[AP], [VLEMI]			E		
		Trap		x			PTR,[VLEMI]			E		
		FP Enabled		x	x		FP, [PIE]	FE0, FE1		E	6,7	
		AP Enabled		x	x		AP			E		
		Unimplemented Op		x			PUO, [VLEMI] [FP,AP,SPV]			E	7	
IVOR7	FP Unavailable	FP Unavailable		x						E		581
IVOR8	System Call	System Call		x			[VLEMI]			E		581
IVOR9	AP Unavailable	AP Unavailable		x						E		581
IVOR10	Decrementer		x					EE	DIE	E		582
IVOR11	FIT		x					EE	FIE	E		582
IVOR12	Watchdog		x			x		CE	WIE	E	10	582
IVOR13	Data TLB Error	Data TLB Miss		x			[ST],[FP,AP,SPV] [VLEMI],[EPID]			E		583
IVOR14	Inst TLB Error	Inst TLB Miss		x			[MIF]			E		583

IVOR	Interrupt	Exception	Asynchronous	Synchronous, Precise	Synchronous, Imprecise	Critical	ESR (See Note 5)	MSR Mask Bit(s)	DBCR0/TCR Mask Bit	Category (Section 1.3.5 of Book I)	Notes (see page 575)	Page
IVOR15	Debug	Trap		x		x		DE	IDM	E	10	584
		Inst Addr Compare		x		x		DE	IDM	E	10	
		Data Addr Compare		x		x		DE	IDM	E	10	
		Instruction Complete		x		x		DE	IDM	E	3,10	
		Branch Taken		x		x		DE	IDM	E	3,10	
		Return From Interrupt		x		x		DE	IDM	E	10	
		Interrupt Taken	x			x		DE	IDM	E	10	
		Uncond Debug Event	x			x		DE	IDM	E.ED	10	
		Critical Interrupt Taken	x					DE	IDM	E.ED		
		Critical Interrupt Return		x				DE	IDM	E.ED		
IVOR32	SPE/Embedded Floating-Point/Vector Unavailable	SPE Unavailable		x			SPV, [VLEMI]			SPE		585
		Vector Unavailable						SPV		V		
IVOR33	Embedded Floating-Point Data	Embedded Floating-Point Data		x			SPV, [VLEMI]			SP.F*		586
IVOR34	Embedded Floating-Point Round	Embedded Floating-Point Round		x			SPV, [VLEMI]			SP.F*		586
IVOR35	Embedded Performance Monitor	Embedded Performance Monitor	x							E.PM		
IVOR36	Processor Doorbell	Processor Doorbell	x					EE		E.PC		
IVOR37	Processor Critical Doorbell	Processor Critical Doorbell	x			x		CE		E.PC		

Figure 15. Interrupt and Exception Types

Figure 15 Notes

1. Although it is not specified, it is common for system implementations to provide, as part of the interrupt controller, independent mask and status bits for the various sources of Critical Input and External Input interrupts.
2. Machine Check interrupts are a special case and are not classified as asynchronous nor synchronous. See Section 5.4.4 on page 571.
3. The Instruction Complete and Branch Taken debug events are only defined for MSR_{DE}=1 when in Internal Debug Mode ($DBCR0_{IDM}$=1). In other words, when in Internal Debug Mode with MSR_{DE}=0, then Instruction Complete and Branch Taken debug events cannot occur, and no DBSR status bits are set and no subsequent imprecise Debug interrupt will occur (see Section 8.4 on page 606).
4. Machine Check status information is commonly provided as part of the system implementation, but is implementation-dependent.
5. In general, when an interrupt causes a particular ESR bit or bits to be set (or cleared) as indicated in the table, it also causes all other ESR bits to be cleared. There may be special rules regarding the handling of implementation-specific ESR bits.

   Legend:

   [xxx] means ESR_{xxx} could be set

   [xxx,yyy] means either ESR_{xxx} or ESR_{yyy} may be set, but never both

   (xxx,yyy) means either ESR_{xxx} or ESR_{yyy} will be set, but never both

   {xxx,yyy} means either ESR_{xxx} or ESR_{yyy} will be set, or possibly both

   xxx means ESR_{xxx} is set

6. The precision of the Floating-point Enabled Exception type Program interrupt is controlled by the $MSR_{FE0,FE1}$ bits. When $MSR_{FE0,FE1}$=0b01 or 0b10, the interrupt may be imprecise. When such a Program interrupt is taken, if the address saved in SRR0 is not the address of the instruction that caused the exception (i.e. the instruction that caused $FPSCR_{FEX}$ to be set to 1), ESR_{PIE} is set to 1. When $MSR_{FE0,FE1}$=0b11, the interrupt is precise. When $MSR_{FE0,FE1}$=0b00, the interrupt is masked, and the interrupt will subsequently occur imprecisely if and when Floating-point Enabled Exception type Program interrupts are enabled by setting either or both of $MSR_{FE0,FE1}$, and will also cause ESR_{PIE} to be set to 1. See Section 5.6.7. Also, exception status on the exact cause is available in the Floating-Point Status and Control Register (see Section 4.2.2 and Section 4.4 of Book I).

   The precision of the Auxiliary Processor Enabled Exception type Program interrupt is implementation-dependent.
7. Auxiliary Processor exception status is commonly provided as part of the implementation.
8. Cache locking and cache locking exceptions are implementation-dependent.
9. Software must examine the instruction and the subject TLB entry to determine the exact cause of the interrupt.
10. If the Embedded.Enhanced Debug category is enabled, this interrupt is not a critical interrupt. DSRR0 and DSRR1 are used instead of CSRR0 and CSRR1.

5.6.1 Critical Input Interrupt

A Critical Input interrupt occurs when no higher priority exception exists (see Section 5.9 on page 591), a Critical Input exception is presented to the interrupt mechanism, and MSR_{CE}=1. While the specific definition of a Critical Input exception is implementation-dependent, it would typically be caused by the activation of an asynchronous signal that is part of the system. Also, implementations may provide an alternative means (in addition to MSR_{CE}) for masking the Critical Input interrupt.

CSRR0, CSRR1, and MSR are updated as follows:

CSRR0 Set to the effective address of the next instruction to be executed.

CSRR1 Set to the contents of the MSR at the time of the interrupt.

MSR
- CM MSR_{CM} is set to MSR_{ICM}.
- ME, ICM Unchanged.
- DE Unchanged if category E.ED is supported; otherwise set to 0

All other defined MSR bits set to 0.

Instruction execution resumes at address $IVPR_{0:47}$ || $IVOR0_{48:59}$||0b0000.

Programming Note

Software is responsible for taking any action(s) that are required by the implementation in order to clear any Critical Input exception status prior to re-enabling MSR_{CE} in order to avoid another, redundant Critical Input interrupt.

5.6.2 Machine Check Interrupt

A Machine Check interrupt occurs when no higher priority exception exists (see Section 5.9 on page 591), a Machine Check exception is presented to the interrupt mechanism, and MSR_{ME}=1. The specific cause or causes of Machine Check exceptions are implementation-dependent, as are the details of the actions taken on a Machine Check interrupt.

If the Machine Check Extension is implemented, MCSRR0, MCSRR1, and MCSR are set, otherwise CSRR0, CSRR1, and ESR are set. The registers are updated as follows:

CSRR0/MCSRR0
Set to an instruction address. As closely as possible, set to the effective address of an instruction that was executing or about to be executed when the Machine Check exception occurred.

CSRR1/MCSRR1
Set to the contents of the MSR at the time of the interrupt.

MSR
- CM MSR_{CM} is set to MSR_{ICM}.
- DE Unchanged if category E.ED is supported; otherwise set to 0.

All other defined MSR bits set to 0.

ESR/MCSR
Implementation-dependent.

Instruction execution resumes at address $IVPR_{0:47}$ || $IVOR1_{48:59}$||0b0000.

Programming Note

If a Machine Check interrupt is caused by an error in the storage subsystem, the storage subsystem may return incorrect data, that may be placed into registers and/or on-chip caches.

Programming Note

On implementations on which a Machine Check interrupt can be caused by referring to an invalid real address, executing a ***dcbz***, ***dcbzep***, or ***dcba*** instruction can cause a delayed Machine Check interrupt by establishing in the data cache a block that is associated with an invalid real address. See Section 3.2 of Book II. A Machine Check interrupt can eventually occur if and when a subsequent attempt is made to write that block to main storage, for example as the result of executing an instruction that causes a cache miss for which the block is the target for replacement or as the result of executing a ***dcbst, dcbstep***, ***dcbf,*** or ***dcbfep*** instruction.

5.6.3 Data Storage Interrupt

A Data Storage interrupt may occur when no higher priority exception exists (see Section 5.9 on page 591) and a Data Storage exception is presented to the interrupt mechanism. A Data Storage exception is caused when any of the following exceptions arises during execution of an instruction:

Read Access Control exception

A Read Access Control exception is caused when one of the following conditions exist.

- While in user mode (MSR_{PR}=1), a *Load* or 'load-class' *Cache Management* instruction attempts to access a location in storage that is not user mode read enabled (i.e. page access control bit UR=0).
- While in supervisor mode (MSR_{PR}=0), a *Load* or 'load-class' *Cache Management* instruction attempts to access a location in storage that is not supervisor mode read enabled (i.e. page access control bit SR=0).

Write Access Control exception

A Write Access Control exception is caused when one of the following conditions exist.

- While in user mode (MSR_{PR}=1), a *Store* or 'store-class' *Cache Management* instruction attempts to access a location in storage that is not user mode write enabled (i.e. page access control bit UW=0).
- While in supervisor mode (MSR_{PR}=0), a *Store* or 'store-class' *Cache Management* instruction attempts to access a location in storage that is not supervisor mode write enabled (i.e. page access control bit SW=0).

Byte Ordering exception

A Byte Ordering exception may occur when the implementation cannot perform the data storage access in the byte order specified by the Endian storage attribute of the page being accessed.

Cache Locking exception

A Cache Locking exception may occur when the locked state of one or more cache lines has the potential to be altered. This exception is implementation-dependent.

Storage Synchronization exception

A Storage Synchronization exception will occur when an attempt is made to execute a *Load and Reserve* or *Store Conditional* instruction from or to a location that is Write Through Required or Caching Inhibited (if the interrupt does not occur then the instruction executes correctly: see Section 3.3.2 of Book II).

If a ***stwcx.*** or ***stdcx.*** would not perform its store in the absence of a Data Storage interrupt, and either (a) the specified effective address refers to storage that is Write Through Required or Caching Inhibited, or (b) a non-conditional *Store* to the specified effective address would cause a Data Storage interrupt, it is implementation-dependent whether a Data Storage interrupt occurs.

Instructions ***lswx*** or ***stswx*** with a length of zero, ***icbt***, ***dcbt***, ***dcbtep***, ***dcbtst***, ***dcbtstep***, or ***dcba*** cannot cause a Data Storage interrupt, regardless of the effective address.

Programming Note

The ***icbi***, ***icbiep***, and ***icbt*** instructions are treated as *Loads* from the addressed byte with respect to address translation and protection. These *Instruction Cache Management* instructions use MSR_{DS}, not MSR_{IS}, to determine translation for their operands. Instruction Storage exceptions and Instruction TLB Miss exceptions are associated with the 'fetching' of instructions not with the 'execution' of instructions. Data Storage exceptions and Data TLB Miss exceptions are associated with the 'execution' of *Instruction Cache Management* instructions.

When a Data Storage interrupt occurs, the processor suppresses the execution of the instruction causing the Data Storage exception.

SRR0, SRR1, MSR, DEAR, and ESR are updated as follows:

SRR0 Set to the effective address of the instruction causing the Data Storage interrupt.

SRR1 Set to the contents of the MSR at the time of the interrupt.

MSR

CM MSR_{CM} is set to MSR_{ICM}.
CE, ME,
DE, ICM Unchanged.

All other defined MSR bits set to 0.

DEAR Set to the effective address of a byte that is both within the range of the bytes being accessed by the *Storage Access* or *Cache Management* instruction, and within the page whose access caused the Data Storage exception.

ESR

FP	Set to 1 if the instruction causing the interrupt is a floating-point load or store; otherwise set to 0.
ST	Set to 1 if the instruction causing the interrupt is a *Store* or 'store-class' *Cache Management* instruction; otherwise set to 0.
$DLK_{0:1}$	Set to an implementation-dependent value due to a Cache Locking exception causing the interrupt.
AP	Set to 1 if the instruction causing the interrupt is an Auxiliary Processor load or store; otherwise set to 0.
BO	Set to 1 if the instruction caused a Byte Ordering exception; otherwise set to 0.
SPV	Set to 1 if the instruction causing the interrupt is a SPE operation or a Vector operation; otherwise set to 0.
VLEMI	Set to 1 if the instruction causing the interrupt resides in VLE storage.
EPID	Set to 1 if the instruction causing the interrupt is an External Process ID instruction; otherwise set to 0.

All other defined ESR bits are set to 0.

Programming Note

Read and Write Access Control and Byte Ordering exceptions are not mutually exclusive. Even if ESR_{BO} is set, system software must also examine the TLB entry accessed by the data storage access to determine whether or not a Read Access Control or Write Access Control exception may have also occurred.

Instruction execution resumes at address $IVPR_{0:47}$ || $IVOR2_{48:59}$||0b0000.

5.6.4 Instruction Storage Interrupt

An Instruction Storage interrupt occurs when no higher priority exception exists (see Section 5.9 on page 591) and an Instruction Storage exception is presented to the interrupt mechanism. An Instruction Storage exception is caused when any of the following exceptions arises during execution of an instruction:

Execute Access Control exception

An Execute Access Control exception is caused when one of the following conditions exist.

- While in user mode (MSR_{PR}=1), an instruction fetch attempts to access a location in storage that is not user mode execute enabled (i.e. page access control bit UX=0).
- While in supervisor mode (MSR_{PR}=0), an instruction fetch attempts to access a location in storage that is not supervisor mode execute enabled (i.e. page access control bit SX=0).

Byte Ordering exception

A Byte Ordering exception may occur when the implementation cannot perform the instruction fetch in the byte order specified by the Endian storage attribute of the page being accessed.

When an Instruction Storage interrupt occurs, the processor suppresses the execution of the instruction causing the Instruction Storage exception.

SRR0, SRR1, MSR, and ESR are updated as follows:

SRR0 Set to the effective address of the instruction causing the Instruction Storage interrupt.

SRR1 Set to the contents of the MSR at the time of the interrupt.

MSR

CM	MSR_{CM} is set to MSR_{ICM}.
CE, ME, DE, ICM	Unchanged.

All other defined MSR bits set to 0.

ESR

BO	Set to 1 if the instruction fetch caused a Byte Ordering exception; otherwise set to 0.
VLEMI	Set to 1 if the instruction causing the interrupt resides in VLE storage.

All other defined ESR bits are set to 0.

Programming Note

Execute Access Control and Byte Ordering exceptions are not mutually exclusive. Even if ESR_{BO} is set, system software must also examine the TLB entry accessed by the instruction fetch to determine whether or not an Execute Access Control exception may have also occurred.

Instruction execution resumes at address $IVPR_{0:47}$ || $IVOR3_{48:59}$||0b0000.

5.6.5 External Input Interrupt

An External Input interrupt occurs when no higher priority exception exists (see Section 5.9 on page 591), an External Input exception is presented to the interrupt mechanism, and MSR_{EE}=1. While the specific definition of an External Input exception is implementation-dependent, it would typically be caused by the activation of an asynchronous signal that is part of the processing system. Also, implementations may provide an alternative means (in addition to MSR_{EE}) for masking the External Input interrupt.

SRR0, SRR1, and MSR are updated as follows:

SRR0 Set to the effective address of the next instruction to be executed.

SRR1 Set to the contents of the MSR at the time of the interrupt.

MSR

CM MSR_{CM} is set to MSR_{ICM}.
CE, ME,
DE, ICM Unchanged.

All other defined MSR bits set to 0.

Instruction execution resumes at address $IVPR_{0:47}$ || $IVOR4_{48:59}$||0b0000.

Programming Note

Software is responsible for taking whatever action(s) are required by the implementation in order to clear any External Input exception status prior to re-enabling MSR_{EE} in order to avoid another, redundant External Input interrupt.

5.6.6 Alignment Interrupt

An Alignment interrupt occurs when no higher priority exception exists (see Section 5.9 on page 591) and an Alignment exception is presented to the interrupt mechanism. An Alignment exception may be caused when the implementation cannot perform a data storage access for one of the following reasons:

- The operand of a *Load* or *Store* is not aligned.
- The instruction is a *Move Assist*, *Load Multiple* or *Store Multiple*.
- The operand of ***dcbz*** or ***dcbzep*** is in storage that is Write Through Required or Caching Inhibited, or one of these instructions is executed in an implementation that has either no data cache or a Write Through data cache or the line addressed by the instruction cannot be established in the cache because the cache is disabled or locked.
- The operand of a *Store*, except *Store Conditional*, is in storage that is Write-Through Required.

For ***lmw*** and ***stmw*** with an operand that is not word-aligned, and for *Load and Reserve* and *Store Conditional* instructions with an operand that is not aligned, an implementation may yield boundedly undefined results instead of causing an Alignment interrupt. A *Store Conditional* to Write Through Required storage may either cause a Data Storage interrupt, cause an Alignment interrupt, or correctly execute the instruction. For all other cases listed above, an implementation may execute the instruction correctly instead of causing an Alignment interrupt. (For ***dcbz*** or ***dcbzep***, 'correct' execution means setting each byte of the block in main storage to 0x00.)

Programming Note

The architecture does not support the use of an unaligned effective address by *Load and Reserve* and *Store Conditional* instructions. If an Alignment interrupt occurs because one of these instructions specifies an unaligned effective address, the Alignment interrupt handler must not attempt to emulate the instruction, but instead should treat the instruction as a programming error.

When an Alignment interrupt occurs, the processor suppresses the execution of the instruction causing the Alignment exception.

SRR0, SRR1, MSR, DEAR, and ESR are updated as follows:

SRR0 Set to the effective address of the instruction causing the Alignment interrupt.

SRR1 Set to the contents of the MSR at the time of the interrupt.

MSR

CM MSR_{CM} is set to MSR_{ICM}.
CE, ME,
DE, ICM Unchanged.

All other defined MSR bits set to 0.

DEAR Set to the effective address of a byte that is both within the range of the bytes being accessed by the *Storage Access* or *Cache Management* instruction, and within the page whose access caused the Alignment exception.

ESR

FP Set to 1 if the instruction causing the interrupt is a floating-point load or store; otherwise set to 0.
ST Set to 1 if the instruction causing the interrupt is a *Store*; otherwise set to 0.
AP Set to 1 if the instruction causing the interrupt is an Auxiliary Processor load or store; otherwise set to 0.
SPV Set to 1 if the instruction causing the interrupt is a SPE operation or a Vector operation; otherwise set to 0.
VLEMI Set to 1 if the instruction causing the interrupt resides in VLE storage.
EPID Set to 1 if the instruction causing the interrupt is an External Process ID instruction; otherwise set to 0.

All other defined ESR bits are set to 0.

Instruction execution resumes at address $IVPR_{0:47}$ || $IVOR5_{48:59}$||0b0000.

5.6.7 Program Interrupt

A Program interrupt occurs when no higher priority exception exists (see Section 5.9 on page 591), a Program exception is presented to the interrupt mechanism, and, for Floating-point Enabled exception, $MSR_{FE0,FE1}$ are non-zero. A Program exception is caused when any of the following exceptions arises during execution of an instruction:

Floating-point Enabled exception

A Floating-point Enabled exception is caused when $FPSCR_{FEX}$ 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. Note that in this context, the term ‘enabled exception’ refers to the enabling provided by control bits in the Floating-Point Status and Control Register. See Section 4.2.2 of Book I.

Auxiliary Processor Enabled exception

The cause of an Auxiliary Processor Enabled exception is implementation-dependent.

Illegal Instruction exception

An Illegal Instruction exception *does* occur when execution is attempted of any of the following kinds of instructions.

- a reserved-illegal instruction
- when MSR_{PR}=1 (user mode), an ***mtspr*** or ***mfspr*** that specifies an SPRN value with $SPRN_5$=0 (user-mode accessible) that represents an unimplemented Special Purpose Register

An Illegal Instruction exception *may* occur when execution is attempted of any of the following kinds of instructions. If the exception does not occur, the alternative is shown in parentheses.

- an instruction that is in invalid form (boundedly undefined results)
- an ***lswx*** instruction for which register RA or register RB is in the range of registers to be loaded (boundedly undefined results)
- a reserved-no-op instruction (no-operation performed is preferred)
- a defined instruction that is not implemented by the implementation (Unimplemented Operation exception)

Privileged Instruction exception

A Privileged Instruction exception occurs when MSR_{PR}=1 and execution is attempted of any of the following kinds of instructions.

- a privileged instruction
- an ***mtspr*** or ***mfspr*** instruction that specifies an SPRN value with $SPRN_5$=1

Trap exception

A Trap exception occurs when any of the conditions specified in a *Trap* instruction are met and the exception is not also enabled as a Debug interrupt. If enabled as a Debug interrupt (i.e. $DBCR0_{TRAP}$=1, $DBCR0_{IDM}$=1, and MSR_{DE}=1), then a Debug interrupt will be taken instead of the Program interrupt.

Unimplemented Operation exception

An Unimplemented Operation exception *may* occur when execution is attempted of a defined instruction that is not implemented by the implementation. Otherwise an Illegal Instruction exception occurs.

An Unimplemented Operation exception *may* also occur when the processor is in 32-bit mode and execution is attempted of an instruction that is part of the 64-Bit category. Otherwise the instruction executes normally.

SRR0, SRR1, MSR, and ESR are updated as follows:

SRR0 For all Program interrupts except an Enabled exception when in one of the imprecise modes (see Section 2.2.1 on page 513) or when a disabled exception is subsequently enabled, set to the effective address of the instruction that caused the Program interrupt.

For an imprecise Enabled exception, set to the effective address of the excepting instruction or to the effective address of some subsequent instruction. If it points to a subsequent instruction, that instruction has not been executed, and ESR_{PIE} is set to 1. If a subsequent instruction is an ***msync*** or ***isync***, SRR0 will point at the ***msync*** or ***isync*** instruction, or at the following instruction.

If $FPSCR_{FEX}$=1 but both MSR_{FE0}=0 and MSR_{FE1}=0, an Enabled exception type Program interrupt will occur imprecisely prior to or at the next synchronizing event if these MSR bits are altered by any instruction that can set the MSR so that the expression

$(MSR_{FE0} \mid MSR_{FE1})$ & $FPSCR_{FEX}$

is 1. When this occurs, SRR0 is loaded with the address of the instruction that would have executed next, not with the address of the instruction that modified the MSR causing the interrupt, and ESR_{PIE} is set to 1.

SRR1 Set to the contents of the MSR at the time of the interrupt.

MSR

CM MSR_{CM} is set to MSR_{ICM}.

CE, ME, DE, ICM Unchanged.

All other defined MSR bits set to 0.

ESR

PIL Set to 1 if an Illegal Instruction exception type Program interrupt; otherwise set to 0
PPR Set to 1 if a Privileged Instruction exception type Program interrupt; otherwise set to 0
PTR Set to 1 if a Trap exception type Program interrupt; otherwise set to 0
PUO Set to 1 if an Unimplemented Operation exception type Program interrupt; otherwise set to 0
FP Set to 1 if the instruction causing the interrupt is a floating-point instruction; otherwise set to 0.
PIE Set to 1 if a Floating-point Enabled exception type Program interrupt, and the address saved in SRR0 is not the address of the instruction causing the exception (i.e. the instruction that caused $FPSCR_{FEX}$ to be set); otherwise set to 0.
AP Set to 1 if the instruction causing the interrupt is an Auxiliary Processor instruction; otherwise set to 0.
SPV Set to 1 if the instruction causing the interrupt is a SPE operation or a Vector operation; otherwise set to 0.
VLEMI Set to 1 if the instruction causing the interrupt resides in VLE storage.

All other defined ESR bits are set to 0.

Instruction execution resumes at address $IVPR_{0:47}$ || $IVOR6_{48:59}$||0b0000.

5.6.8 Floating-Point Unavailable Interrupt

A Floating-Point Unavailable interrupt occurs when no higher priority exception exists (see Section 5.9 on page 591), an attempt is made to execute a floating-point instruction (i.e. any instruction listed in Section 4.6 of Book I), and MSR_{FP}=0.

When a Floating-Point Unavailable interrupt occurs, the processor suppresses the execution of the instruction causing the Floating-Point Unavailable interrupt.

SRR0, SRR1, and MSR are updated as follows:

SRR0 Set to the effective address of the instruction that caused the interrupt.

SRR1 Set to the contents of the MSR at the time of the interrupt.

MSR

CM MSR_{CM} is set to MSR_{ICM}.
CE, ME, DE, ICM Unchanged.

All other defined MSR bits set to 0.

Instruction execution resumes at address $IVPR_{0:47}$ || $IVOR7_{48:59}$||0b0000.

5.6.9 System Call Interrupt

A System Call interrupt occurs when no higher priority exception exists (see Section 5.9 on page 591) and a *System Call* (***sc***) instruction is executed.

SRR0, SRR1, and MSR are updated as follows:

SRR0 Set to the effective address of the instruction *after* the ***sc*** instruction.

SRR1 Set to the contents of the MSR at the time of the interrupt.

MSR

CM MSR_{CM} is set to MSR_{ICM}.
VLEMI Set to 1 if the instruction causing the interrupt resides in VLE storage.
CE, ME, DE, ICM Unchanged.

All other defined MSR bits set to 0.

Instruction execution resumes at address $IVPR_{0:47}$ || $IVOR8_{48:59}$||0b0000.

5.6.10 Auxiliary Processor Unavailable Interrupt

An Auxiliary Processor Unavailable interrupt occurs when no higher priority exception exists (see Section 5.9 on page 591), an attempt is made to execute an Auxiliary Processor instruction (including Auxiliary Processor loads, stores, and moves), the target Auxiliary Processor is present on the implementation, and the Auxiliary Processor is configured as unavailable. Details of the Auxiliary Processor, its instruction set, and its configuration are implementation-dependent. See User's Manual for the implementation.

When an Auxiliary Processor Unavailable interrupt occurs, the processor suppresses the execution of the instruction causing the Auxiliary Processor Unavailable interrupt.

Registers SRR0, SRR1, and MSR are updated as follows:

SRR0 Set to the effective address of the instruction that caused the interrupt.

SRR1 Set to the contents of the MSR at the time of the interrupt.

MSR

CM MSR_{CM} is set to MSR_{ICM}.
CE, ME, DE, ICM Unchanged.

All other defined MSR bits set to 0.

Instruction execution resumes at address $IVPR_{0:47}$ || $IVOR9_{48:59}$||0b0000.

5.6.11 Decrementer Interrupt

A Decrementer interrupt occurs when no higher priority exception exists (see Section 5.9 on page 591), a Decrementer exception exists (TSR_{DIS}=1), and the interrupt is enabled (TCR_{DIE}=1 and MSR_{EE}=1). See Section 7.3 on page 599.

Programming Note

MSR_{EE} also enables the External Input and Fixed-Interval Timer interrupts.

SRR0, SRR1, MSR, and TSR are updated as follows:

SRR0 Set to the effective address of the next instruction to be executed.

SRR1 Set to the contents of the MSR at the time of the interrupt.

MSR
CM MSR_{CM} is set to MSR_{ICM}.
CE, ME,
DE, ICM Unchanged.

All other defined MSR bits set to 0.

TSR (See Section 7.5.1 on page 601.)
DIS Set to 1.

Instruction execution resumes at address $IVPR_{0:47}$ || $IVOR10_{48:59}$||0b0000.

Programming Note

Software is responsible for clearing the Decrementer exception status prior to re-enabling the MSR_{EE} bit in order to avoid another redundant Decrementer interrupt. To clear the Decrementer exception, the interrupt handling routine must clear TSR_{DIS}. Clearing is done by writing a word to TSR using ***mtspr*** with a 1 in any bit position that is to be cleared and 0 in all other bit positions. The write-data to the TSR is not direct data, but a mask. A 1 causes the bit to be cleared, and a 0 has no effect.

5.6.12 Fixed-Interval Timer Interrupt

A Fixed-Interval Timer interrupt occurs when no higher priority exception exists (see Section 5.9 on page 591), a Fixed-Interval Timer exception exists (TSR_{FIS}=1), and the interrupt is enabled (TCR_{FIE}=1 and MSR_{EE}=1). See Section 7.6 on page 602.

Programming Note

MSR_{EE} also enables the External Input and Decrementer interrupts.

SRR0, SRR1, MSR, and TSR are updated as follows:

SRR0 Set to the effective address of the next instruction to be executed.

SRR1 Set to the contents of the MSR at the time of the interrupt.

MSR
CM MSR_{CM} is set to MSR_{ICM}.
CE, ME,
DE, ICM Unchanged.

All other defined MSR bits set to 0.

TSR (See Section 7.5.1 on page 601.)
FIS Set to 1

Instruction execution resumes at address $IVPR_{0:47}$ || $IVOR11_{48:59}$||0b0000.

Programming Note

Software is responsible for clearing the Fixed-Interval Timer exception status prior to re-enabling the MSR_{EE} bit in order to avoid another redundant Fixed-Interval Timer interrupt. To clear the Fixed-Interval Timer exception, the interrupt handling routine must clear TSR_{FIS}. Clearing is done by writing a word to TSR using ***mtspr*** with a 1 in any bit position that is to be cleared and 0 in all other bit positions. The write-data to the TSR is not direct data, but a mask. A 1 causes the bit to be cleared, and a 0 has no effect.

5.6.13 Watchdog Timer Interrupt

A Watchdog Timer interrupt occurs when no higher priority exception exists (see Section 5.9 on page 591), a Watchdog Timer exception exists (TSR_{WIS}=1), and the interrupt is enabled (i.e. TCR_{WIE}=1 and MSR_{CE}=1). See Section 7.7 on page 602.

Programming Note

MSR_{CE} also enables the Critical Input interrupt.

CSRR0, CSRR1, MSR, and TSR are updated as follows:

CSRR0 Set to the effective address of the next instruction to be executed.

CSRR1 Set to the contents of the MSR at the time of the interrupt.

MSR
CM MSR_{CM} is set to MSR_{ICM}.
ME, ICM,

DE Unchanged.

All other defined MSR bits set to 0.

TSR (See Section 7.5.1 on page 601.)
WIS Set to 1.

Instruction execution resumes at address $IVPR_{0:47}$ || $IVOR12_{48:59}$||0b0000.

Programming Note

Software is responsible for clearing the Watchdog Timer exception status prior to re-enabling the MSR_{CE} bit in order to avoid another redundant Watchdog Timer interrupt. To clear the Watchdog Timer exception, the interrupt handling routine must clear TSR_{WIS}. Clearing is done by writing a word to TSR using ***mtspr*** with a 1 in any bit position that is to be cleared and 0 in all other bit positions. The write-data to the TSR is not direct data, but a mask. A 1 causes the bit to be cleared, and a 0 has no effect.

5.6.14 Data TLB Error Interrupt

A Data TLB Error interrupt occurs when no higher priority exception exists (see Section 5.9 on page 591) and any of the following Data TLB Error exceptions is presented to the interrupt mechanism.

TLB Miss exception

Caused when the virtual address associated with a data storage access does not match any valid entry in the TLB as specified in Section 4.7.2 on page 545.

If a ***stwcx.*** or ***stdcx.*** would not perform its store in the absence of a Data Storage interrupt, and 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.

When a Data TLB Error interrupt occurs, the processor suppresses the execution of the instruction causing the Data TLB Error interrupt.

SRR0, SRR1, MSR, DEAR and ESR are updated as follows:

SRR0 Set to the effective address of the instruction causing the Data TLB Error interrupt

SRR1 Set to the contents of the MSR at the time of the interrupt.

MSR
CM MSR_{CM} is set to MSR_{ICM}.
CE, ME, DE, ICM Unchanged.

All other defined MSR bits set to 0.

DEAR Set to the effective address of a byte that is both within the range of the bytes being accessed by the *Storage Access* or *Cache Management* instruction, and within the page whose access caused the Data TLB Error exception.

ESR
ST Set to 1 if the instruction causing the interrupt is a *Store*, ***dcbi***, ***dcbz***, or ***dcbzep*** instruction; otherwise set to 0.
FP Set to 1 if the instruction causing the interrupt is a floating-point load or store; otherwise set to 0.
AP Set to 1 if the instruction causing the interrupt is an Auxiliary Processor load or store; otherwise set to 0.
SPV Set to 1 if the instruction causing the interrupt is a SPE operation or a Vector operation; otherwise set to 0.
VLEMI Set to 1 if the instruction causing the interrupt resides in VLE storage.
EPID Set to 1 if the instruction causing the interrupt is an External Process ID instruction; otherwise set to 0.

All other defined ESR bits are set to 0.

Instruction execution resumes at address $IVPR_{0:47}$ || $IVOR13_{48:59}$||0b0000.

5.6.15 Instruction TLB Error Interrupt

An Instruction TLB Error interrupt occurs when no higher priority exception exists (see Section 5.9 on page 591) and any of the following Instruction TLB Error exceptions is presented to the interrupt mechanism.

TLB Miss exception

Caused when the virtual address associated with an instruction fetch does not match any valid entry in the TLB as specified in Section 4.7.2 on page 545.

When an Instruction TLB Error interrupt occurs, the processor suppresses the execution of the instruction causing the Instruction TLB Miss exception.

SRR0, SRR1, and MSR are updated as follows:

SRR0 Set to the effective address of the instruction causing the Instruction TLB Error interrupt.

SRR1 Set to the contents of the MSR at the time of the interrupt.

MSR
CM MSR_{CM} is set to MSR_{ICM}.
CE, ME, DE, ICM Unchanged.

All other defined MSR bits set to 0.

Instruction execution resumes at address $IVPR_{0:47}$ || $IVOR14_{48:59}$||0b0000.

5.6.16 Debug Interrupt

A Debug interrupt occurs when no higher priority exception exists (see Section 5.9 on page 591), a Debug exception exists in the DBSR, and Debug interrupts are enabled ($DBCR0_{IDM}$=1 and MSR_{DE}=1). A Debug exception occurs when a Debug Event causes a corresponding bit in the DBSR to be set. See Section 8.5.

If the Embedded.Enhanced Debug category is not supported or is supported and is not enabled, CSRR0, CSRR1, MSR, and DBSR are updated as follows. If the Embedded.Enhanced Debug category is supported and is enabled, DSRR0 and DSRR1 are updated as specified below and CSRR0 and CSRR1 are not changed. The means by which the Embedded.Enhanced Debug category is enabled is implementation-dependent.

CSRR0 or **DSRR0** [Category: Embedded.Enhanced Debug]

For Debug exceptions that occur while Debug interrupts are enabled ($DBCR0_{IDM}$=1 and MSR_{DE}=1), CSRR0 is set as follows:

- For Instruction Address Compare (IAC1, IAC2, IAC3, IAC4), Data Address Compare (DAC1R, DAC1W, DAC2R, DAC2W), Data Value Compare (DVC1, DVC2), Trap (TRAP), or Branch Taken (BRT) debug exceptions, set to the address of the instruction causing the Debug interrupt.
- For Instruction Complete (ICMP) debug exceptions, set to the address of the instruction that would have executed after the one that caused the Debug interrupt.
- For Unconditional Debug Event (UDE) debug exceptions, set to the address of the instruction that would have executed next if the Debug interrupt had not occurred.
- For Interrupt Taken (IRPT) debug exceptions, set to the interrupt vector value of the interrupt that caused the Interrupt Taken debug event.
- For Return From Interrupt (RET) debug exceptions, set to the address of the ***rfi*** instruction that caused the Debug interrupt.
- For Critical Interrupt Taken (CRPT) debug exceptions, DSRR0 is set to the address of the first instruction of the critical interrupt handler. CSRR0 is unaffected.
- For Critical Interrupt Return (CRET) debug exceptions, DSRR0 is set to the address of the ***rfci*** instruction that

caused the Debug interrupt. See Section 8.4.10, “Critical Interrupt Return Debug Event [Category: Embedded.Enhanced Debug]”.

For Debug exceptions that occur while Debug interrupts are disabled ($DBCR0_{IDM}$=0 or MSR_{DE}=0), a Debug interrupt will occur at the next synchronizing event if $DBCR0_{IDM}$ and MSR_{DE} are modified such that they are both 1 and if the Debug exception Status is still set in the DBSR. When this occurs, CSRR0 or DSRR0 [Category:Embedded.Enhanced Debug] is set to the address of the instruction that would have executed next, not with the address of the instruction that modified the Debug Control Register 0 or MSR and thus caused the interrupt.

CSRR1 or **DSRR1** [Category: Embedded.Enhanced Debug]

Set to the contents of the MSR at the time of the interrupt.

MSR

CM MSR_{CM} is set to MSR_{ICM}.
ME, ICM Unchanged.

All other supported MSR bits set to 0.

DBSR Set to indicate type of Debug Event (see Section 8.5.2)

Instruction execution resumes at address $IVPR_{0:47}$ || $IVOR15_{48:59}$||0b0000.

5.6.17 SPE/Embedded Floating-Point/Vector Unavailable Interrupt [Categories: SPE.Embedded Float Scalar Double, SPE.Embedded Float Vector, Vector]

The SPE/Embedded Floating-Point/Vector Unavailable interrupt occurs when no higher priority exception exists, and an attempt is made to execute an SPE, SPE.Embedded Float Scalar Double, SPE.Embedded Float Vector, or Vector instruction and MSR_{SPV} = 0.

When an Embedded Floating-Point Unavailable interrupt occurs, the processor suppresses the execution of the instruction causing the exception.

SRR0, SRR1, MSR, and ESR are updated as follows:

SRR0 Set to the effective address of the instruction causing the Embedded Floating-Point Unavailable interrupt.

SRR1 Set to the contents of the MSR at the time of the interrupt.

MSR

CM MSR_{CM} is set to MSR_{ICM}.
VLEMI Set to 1 if the instruction causing the interrupt resides in VLE storage.
CE, ME,
DE, ICM Unchanged.

All other defined MSR bits set to 0.

ESR

SPV Set to 1.
VLEMI Set to 1 if the instruction causing the interrupt resides in VLE storage.

All other defined ESR bits are set to 0.

Instruction execution resumes at address $IVPR_{0:47}$ || $IVOR32_{48:59}$||0b0000.

Programming Note

This interrupt is also used by the Signal Processing Engine in the same manner. It should be used by software to determine if the application is using the upper 32 bits of the GPRs in a 32-bit implementation and thus be required to save and restore them on context switch.

5.6.18 Embedded Floating-Point Data Interrupt [Categories: SPE.Embedded Float Scalar Double, SPE.Embedded Float Scalar Single, SPE.Embedded Float Vector]

The Embedded Floating-Point Data interrupt occurs when no higher priority exception exists (see Section 5.9) and an Embedded Floating-Point Data exception is presented to the interrupt mechanism. The Embedded Floating-Point Data exception causing the interrupt is indicated in the SPEFSCR; these exceptions include Embedded Floating-Point Invalid Operation/Input Error (FINV, FINVH), Embedded Floating-Point Divide By Zero (FDBZ, FDBZH), Embedded Floating-Point Overflow (FOV, FOVH), and Embedded Floating-Point Underflow (FUNF, FUNFH)

When an Embedded Floating-Point Data interrupt occurs, the processor suppresses the execution of the instruction causing the exception.

SRR0, SRR1, MSR, and ESR are updated as follows:

SRR0 Set to the effective address of the instruction causing the Embedded Floating-Point Data interrupt.

SRR1 Set to the contents of the MSR at the time of the interrupt.

MSR

CM MSR_{CM} is set to MSR_{ICM}.

VLEMI Set to 1 if the instruction causing the interrupt resides in VLE storage.

CE, ME,
DE, ICM Unchanged.

All other defined MSR bits set to 0.

ESR

SPV Set to 1.

All other defined ESR bits are set to 0.

Instruction execution resumes at address $IVPR_{0:47}$ || $IVOR33_{48:59}$||0b0000.

5.6.19 Embedded Floating-Point Round Interrupt [Categories: SPE.Embedded Float Scalar Double, SPE.Embedded Float Scalar Single, SPE.Embedded Float Vector]

The Embedded Floating-Point Round interrupt occurs when no higher priority exception exists (see Section 5.9 on page 591), $SPEFSCR_{FINXE}$ is set to 1, and any of the following occurs:

- the unrounded result of an *Embedded Floating-Point* operation is not exact
- an overflow occurs and overflow exceptions are disabled (FOVF or FOVFH is set to 1 and FOVFE is set to 0)
- an underflow occurs and underflow exceptions are disabled (FUNF is set to 1 and FUNFE is set to 0).

The value of $SPEFSCR_{FINXS}$ is 1, indicating that one of the above exceptions has occurred, and additional information about the exception is found in $SPEFSCR_{FGH\ FG\ FXH\ FX}$.

When an Embedded Floating-Point Round interrupt occurs, the processor completes the execution of the instruction causing the exception and writes the result to the destination register prior to taking the interrupt.

SRR0, SRR1, MSR, and ESR are updated as follows:

SRR0 Set to the effective address of the instruction following the instruction causing the Embedded Floating-Point Round interrupt.

SRR1 Set to the contents of the MSR at the time of the interrupt.

MSR

CM MSR_{CM} is set to MSR_{ICM}.

CE, ME,
DE, ICM Unchanged.

All other defined MSR bits set to 0.

ESR

SPV Set to 1.

VLEMI Set to 1 if the instruction causing the interrupt resides in VLE storage.

All other defined ESR bits are set to 0.

Instruction execution resumes at address $IVPR_{0:47}$ || $IVOR34_{48:59}$||0b0000.

Programming Note

If an implementation does not support ±Infinity rounding modes and the rounding mode is set to be +Infinity or -Infinity, an Embedded Floating-Point Round interrupt occurs after every *Embedded Floating-Point* instruction for which rounding might occur regardless of the value of FINXE, provided no higher priority exception exists.

When an Embedded Floating-Point Round interrupt occurs, the unrounded (truncated) result of an inexact high or low element is placed in the target register. If only a single element is inexact, the other exact element is updated with the correctly rounded result, and the FG and FX bits corresponding to the other exact element will both be 0.

The bits FG (FGH) and FX (FXH) are provided so that an interrupt handler can round the result as it desires. FG (FGH) is the value of the bit immediately to the right of the least significant bit of the destination format mantissa from the infinitely precise intermediate calculation before rounding. FX (FXH) is the value of the 'or' of all the bits to the right of the FG (FGH) of the destination format mantissa from the infinitely precise intermediate calculation before rounding.

5.6.20 Performance Monitor Interrupt [Category: Embedded.Performance Monitor]

The Performance Monitor interrupt is part of the optional Performance Monitor facility; see Appendix E.

5.6.21 Processor Doorbell Interrupt [Category: Embedded.Processor Control]

A Processor Doorbell Interrupt occurs when no higher priority exception exists, a Processor Doorbell exception is present, and MSR_{EE}=1. Processor Doorbell exceptions are generated when DBELL messages (see Chapter 9) are received and accepted by the processor.

When a Processor Doorbell Interrupt occurs, SRR0 is set to the address of the next instruction to be executed and SRR1 is set to the contents of the MSR at the time of the interrupt.

Instruction execution resumes at address $IVPR_{0:47}$ || $IVOR36_{48:59}$ || 0b0000.

5.6.22 Processor Doorbell Critical Interrupt [Category: Embedded.Processor Control]

A Processor Doorbell Critical Interrupt occurs when no higher priority exception exists, a Processor Doorbell Critical exception is present, and MSR_{CE}=1. Processor Doorbell Critical exceptions are generated when DBELL_CRIT messages (see Chapter 9) are received and accepted by the processor.

When a Processor Doorbell Critical Interrupt occurs, CSRR0 is set to the address of the next instruction to be executed and CSRR1 is set to the contents of the MSR at the time of the interrupt.

Instruction execution resumes at address $IVPR_{0:47}$ || $IVOR37_{48:59}$ || 0b0000.

5.7 Partially Executed Instructions

In general, the architecture permits load and store instructions to be partially executed, interrupted, and then to be restarted from the beginning upon return from the interrupt. Unaligned *Load* and *Store* instructions, or *Load Multiple*, *Store Multiple*, *Load String*, and *Store String* instructions may be broken up into multiple, smaller accesses, and these accesses may be performed in any order. In order to guarantee that a particular load or store instruction will complete without being interrupted and restarted, software must mark the storage being referred to as Guarded, and must use an elementary (non-string or non-multiple) load or store that is aligned on an operand-sized boundary.

In order to guarantee that *Load* and *Store* instructions can, in general, be restarted and completed correctly without software intervention, the following rules apply when an execution is partially executed and then interrupted:

- For an elementary *Load*, no part of the target register RT or FRT, will have been altered.
- For *'with update'* forms of *Load* or *Store*, the update register, register RA, will not have been altered.

On the other hand, the following effects are permissible when certain instructions are partially executed and then restarted:

- For any *Store*, some of the bytes at the target storage location may have been altered (if write access to that page in which bytes were altered is permitted by the access control mechanism). In addition, for *Store Conditional* instructions, CR0 has been set to an undefined value, and it is undefined whether the reservation has been cleared.
- For any *Load*, some of the bytes at the addressed storage location may have been accessed (if read access to that page in which bytes were accessed is permitted by the access control mechanism).
- For *Load Multiple* or *Load String*, some of the registers in the range to be loaded may have been altered. Including the addressing registers (RA, and possibly RB) in the range to be loaded is a programming error, and thus the rules for partial execution do not protect against overwriting of these registers.

In no case will access control be violated.

As previously stated, the only load or store instructions that are guaranteed to not be interrupted after being partially executed are elementary, aligned, guarded loads and stores. All others may be interrupted after being partially executed. The following list identifies the specific instruction types for which interruption after partial execution may occur, as well as the specific interrupt types that could cause the interruption:

1. Any *Load* or *Store* (except elementary, aligned, guarded):

   Any asynchronous interrupt
   Machine Check
   Program (Imprecise Mode Floating-Point Enabled)
   Program (Imprecise Mode Auxiliary Processor Enabled)

2. Unaligned elementary *Load* or *Store*, or any multiple or string:

   All of the above listed under item 1, plus the following:
   Data Storage (if the access crosses a protection boundary)
   Debug (Data Address Compare, Data Value Compare)

3. ***mtcrf*** may also be partially executed due to the occurrence of any of the interrupts listed under item 1 at the time the ***mtcrf*** was executing.
   - All instructions prior to the ***mtcrf*** have completed execution. (Some storage accesses generated by these preceding instructions may not have completed.)
   - No subsequent instruction has begun execution.
   - The ***mtcrf*** instruction (the address of which was saved in SRR0/CSRR0/MCSRR0/DSRR0 [Category: Embedded.Enhanced Debug] at the occurrence of the interrupt), may appear not to have begun or may have partially executed.

5.8 Interrupt Ordering and Masking

It is possible for multiple exceptions to exist simultaneously, each of which could cause the generation of an interrupt. Furthermore, for interrupts classes other than the Machine Check interrupt and critical interrupts, the architecture does not provide for reporting more than one interrupt of the same class (unless the Embedded.Enhanced Debug category is supported). Therefore, the architecture defines that interrupts are ordered with respect to each other, and provides a masking mechanism for certain persistent interrupt types.

When an interrupt is masked (disabled), and an event causes an exception that would normally generate the interrupt, the exception *persists* as a *status* bit in a register (which register depends upon the exception type). However, no interrupt is generated. Later, if the interrupt is enabled (unmasked), and the exception status has not been cleared by software, the interrupt due to the original exception event will then finally be generated.

All asynchronous interrupts can be masked. In addition, certain synchronous interrupts can be masked. An example of such an interrupt is the Floating-Point Enabled exception type Program interrupt. The execution of a floating-point instruction that causes the $FPSCR_{FEX}$ bit to be set to 1 is considered an exception event, regardless of the setting of $MSR_{FE0,FE1}$. If $MSR_{FE0,FE1}$ are both 0, then the Floating-Point Enabled exception type of Program interrupt is masked, but the exception persists in the $FPSCR_{FEX}$ bit. Later, if the $MSR_{FE0,FE1}$ bits are enabled, the interrupt will finally be generated.

The architecture enables implementations to avoid situations in which an interrupt would cause the state information (saved in Save/Restore Registers) from a previous interrupt to be overwritten and lost. In order to do this, the architecture defines interrupt classes in a hierarchical manner. At each interrupt class, hardware automatically disables any further interrupts associated with the interrupt class by masking the interrupt enable in the MSR when the interrupt is taken. In addition, each interrupt class masks the interrupt enable in the MSR for each lower class in the hierarchy. The hierarchy of interrupt classes is as follows from highest to lowest:

Interrupt Class	MSR Enables Cleared	Save/Restore Registers
Machine Check	ME,DE, CE, EE	MSRR0/1
Debug[1]	DE,CE,EE	DSRR0/1
Critical	CE,EE	CSRR0/1
Base	EE	SRR0/1

[1] The Debug interrupt class is Category: E.ED. Note: MSR_{DE} may be cleared when a critical interrupt occurs if Category: E.ED is not supported.

Figure 16. Interrupt Hierarchy

If the Embedded.Enhanced Debug category is not supported (or is supported and is not enabled), then the Debug interrupt becomes a Critical class interrupt and all critical class interrupts will clear DE, CE, and EE in the MSR.

Base Class interrupts that occur as a result of precise exceptions are not masked by the EE bit in the MSR and any such exception that occurs prior to software saving the state of SRR0/1 in a base class exception handler will result in a situation that could result in the loss of state information.

This first step of the hardware clearing the MSR enable bits lower in the hierarchy shown in Figure 16 prevents any subsequent asynchronous interrupts from overwriting the Save/Restore Registers (SRR0/SRR1, CSRR0/CSRR1, MCSRR0/MCSRR1, or DSRR0/DSRR1 [Category: Embedded.Enhanced Debug]), prior to software being able to save their contents. Hardware also automatically clears, on any interrupt, $MSR_{WE,PR,FP,FE0,FE1,IS,DS}$. The clearing of these bits assists in the avoidance of subsequent interrupts of certain other types. However, *guaranteeing* that interrupt classes lower in the hierarchy do not occur and thus do not overwrite the Save/Restore Registers (SRR0/SRR1, CSRR0/CSRR1, DSRR0/DSRR1 [Category: Embedded.Enhanced Debug], or MCSRR0/MCSRR1) also requires the cooperation of system software. Specifically, system software must avoid the execution of instructions that could cause (or enable) a subsequent interrupt, if the contents of the Save/Restore Registers (SRR0/SRR1, CSRR0/CSRR1, DSRR0/DSRR1 [Category: Embedded.Enhanced Debug]), or MCSRR0/MCSRR1) have not yet been saved.

5.8.1 Guidelines for System Software

The following list identifies the actions that system software must avoid, prior to having saved the Save/Restore Registers' contents:

- Re-enabling an interrupt class that is at the same or a lower level in the interrupt hierarchy. This includes the following actions:
  - Re-enabling of MSR_{EE}
  - Re-enabling of $MSR_{CE,EE}$ in critical class interrupt handlers, and if the Embedded.Enhanced Debug category is not supported, re-enabling of MSR_{DE}.
  - Category: Embedded.Enhanced Debug: Re-enabling of $MSR_{CE,EE,DE}$ in Debug class interrupt handlers
  - Re-enabling of $MSR_{EE,CE,DE,ME}$ in Machine Check interrupt handlers.
- Branching (or sequential execution) to addresses not mapped by the TLB, or mapped without UX=1 or SX=1 permission.

  This prevents Instruction Storage and Instruction TLB Error interrupts.
- *Load*, *Store* or *Cache Management* instructions to addresses not mapped by the TLB or not having required access permissions.

  This prevents Data Storage and Data TLB Error interrupts.
- Execution of *System Call* (***sc***) or *Trap* (***tw***, ***twi***, ***td***, ***tdi***) instructions

  This prevents System Call and Trap exception type Program interrupts.
- Execution of any floating-point instruction

  This prevents Floating-Point Unavailable interrupts. Note that this interrupt would occur upon the execution of any floating-point instruction, due to the automatic clearing of MSR_{FP}. However, even if software were to re-enable MSR_{FP}, floating-point instructions must still be avoided in order to prevent Program interrupts due to various possible Program interrupt exceptions (Floating-Point Enabled, Unimplemented Operation).
- Re-enabling of MSR_{PR}

  This prevents Privileged Instruction exception type Program interrupts. Alternatively, software could re-enable MSR_{PR}, but avoid the execution of any privileged instructions.
- Execution of any Auxiliary Processor instruction

  This prevents Auxiliary Processor Unavailable interrupts, and Auxiliary Processor Enabled type and Unimplemented Operation type Program interrupts.
- Execution of any Illegal instructions

  This prevents Illegal Instruction exception type Program interrupts.
- Execution of any instruction that could cause an Alignment interrupt

  This prevents Alignment interrupts. Included in this category are any string or multiple instructions, and any unaligned elementary load or store instructions. See Section 5.6.6 on page 579 for a complete list of instructions that may cause Alignment interrupts.

It is not necessary for hardware or software to avoid interrupts higher in the interrupt hierarchy (see Figure 16) from within interrupt handlers (and hence, for example, hardware does not automatically clear $MSR_{CE,ME,DE}$ upon a base class interrupt), since interrupts at each level of the hierarchy use different pairs of Save/Restore Registers to save the instruction address and MSR (i.e. SRR0/SRR1 for base class interrupts, and MCSRR0/MCSRR1,DSRR0/DSRR1 [Category: Embedded.Enhanced Debug], or CSRR0/CSRR1 for non-base class interrupts). The converse, however, is not true. That is, hardware and software must cooperate in the avoidance of interrupts lower in the hierarchy from occurring within interrupt handlers, even though the these interrupts use different Save/Restore Register pairs. This is because the interrupt higher in the hierarchy may have occurred from within a interrupt handler for an interrupt lower in the hierarchy prior to the interrupt handler having saved the Save/Restore Registers. Therefore, within an interrupt handler, Save/Restore Registers for all interrupts lower in the hierarchy may contain data that is necessary to the system software.

5.8.2 Interrupt Order

The following is a prioritized listing of the various enabled interrupts for which exceptions might exist simultaneously:

1. Synchronous (Non-Debug) Interrupts:
   - Data Storage
   - Instruction Storage
   - Alignment
   - Program
   - Floating-Point Unit Unavailable
   - Auxiliary Processor Unavailable
   - Embedded Floating-Point Unavailable [SP.Category: SP.Embedded Float_*]
   - SPE/Embedded Floating-Point/Vector Unavailable
   - Embedded Floating-Point Data [Category: SP.Embedded Float_*]
   - Embedded Floating-Point Round [Category: SP.Embedded Float_*]
   - System Call
   - Data TLB Error
   - Instruction TLB Error

   Only one of the above types of synchronous interrupts may have an existing exception generating it at any given time. This is guaranteed by the exception priority mechanism (see Section 5.9 on page 591) and the requirements of the Sequential Execution Model.

2. Machine Check
3. Debug
4. Critical Input
5. Watchdog Timer
6. Processor Doorbell Critical
7. External Input
8. Fixed-Interval Timer
9. Decrementer
10. Processor Doorbell
11. Embedded Performance Monitor

Even though, as indicated above, the base, synchronous exception types listed under item 1 are generated with higher priority than the non-base interrupt classes listed in items 2-5, the fact is that these base class interrupts will immediately be followed by the highest priority existing interrupt in items 2-5, without executing any instructions at the base class interrupt handler. This is because the base interrupt classes do not automatically disable the MSR mask bits for the interrupts listed in 2-5. In all other cases, a particular interrupt class from the above list will automatically disable any subsequent interrupts of the same class, as well as all other interrupt classes that are listed below it in the priority order.

5.9 Exception Priorities

All synchronous (precise and imprecise) interrupts are reported in program order, as required by the Sequential Execution Model. The one exception to this rule is the case of multiple synchronous imprecise interrupts. Upon a synchronizing event, all previously executed instructions are required to report any synchronous imprecise interrupt-generating exceptions, and the interrupt will then be generated with all of those exception types reported cumulatively, in both the ESR, and any status registers associated with the particular exception type (e.g. the Floating-Point Status and Control Register).

For any single instruction attempting to cause multiple exceptions for which the corresponding synchronous interrupt types are enabled, this section defines the priority order by which the instruction will be permitted to cause a *single* enabled exception, thus generating a particular synchronous interrupt. Note that it is this exception priority mechanism, along with the requirement that synchronous interrupts be generated in program order, that guarantees that at any given time, there exists for consideration only one of the synchronous interrupt types listed in item 1 of Section 5.8.2 on page 591. The exception priority mechanism also prevents certain debug exceptions from existing in combination with certain other synchronous interrupt-generating exceptions.

Because unaligned *Load* and *Store* instructions, or *Load Multiple*, *Store Multiple*, *Load String*, and *Store Sting* instructions may be broken up into multiple, smaller accesses, and these accesses may be performed in any order. The exception priority mechanism applies to each of the multiple storage accesses in the order they are performed by the implementation.

This section does not define the permitted setting of multiple exceptions for which the corresponding interrupt types are disabled. The generation of exceptions for which the corresponding interrupt types are disabled will have no effect on the generation of other exceptions for which the corresponding interrupt types are enabled. Conversely, if a particular exception for which the corresponding interrupt type is enabled is shown in the following sections to be of a higher priority than another exception, it will prevent the setting of that other exception, independent of whether that other exception's corresponding interrupt type is enabled or disabled.

Except as specifically noted, only one of the exception types listed for a given instruction type will be permitted to be generated at any given time. The priority of the exception types are listed in the following sections ranging from highest to lowest, within each instruction type.

> **Programming Note**
>
> Some exception types may even be mutually exclusive of each other and could otherwise be considered the same priority. In these cases, the exceptions are listed in the order suggested by the sequential execution model.

5.9.1 Exception Priorities for Defined Instructions

5.9.1.1 Exception Priorities for Defined Floating-Point Load and Store Instructions

The following prioritized list of exceptions may occur as a result of the attempted execution of any defined *Floating-Point Load* and *Store* instruction.

1. Debug (Instruction Address Compare)
2. Instruction TLB Error
3. Instruction Storage Interrupt (all types)
4. Program (Illegal Instruction)
5. Floating-Point Unavailable
6. Program (Unimplemented Operation)
7. Data TLB Error
8. Data Storage (all types)
9. Alignment
10. Debug (Data Address Compare, Data Value Compare)
11. Debug (Instruction Complete)

If the instruction is causing both a Debug (Instruction Address Compare) and a Debug (Data Address Compare) or Debug (Data Value Compare), and is not causing any of the exceptions listed in items 2-9, it is permissible for both exceptions to be generated and recorded in the DBSR. A single Debug interrupt will result.

5.9.1.2 Exception Priorities for Other Defined Load and Store Instructions and Defined Cache Management Instructions

The following prioritized list of exceptions may occur as a result of the attempted execution of any other defined *Load* or *Store* instruction, or defined *Cache Management* instruction.

1. Debug (Instruction Address Compare)
2. Instruction TLB Error
3. Instruction Storage Interrupt (all types)
4. Program (Illegal Instruction)
5. Program (Privileged Instruction) (***dcbi*** only)
6. Program (Unimplemented Operation)
7. Data TLB Error
8. Data Storage (all types)
9. Alignment
10. Debug (Data Address Compare, Data Value Compare)
11. Debug (Instruction Complete)

If the instruction is causing both a Debug (Instruction Address Compare) and a Debug (Data Address Compare) or Debug (Data Value Compare), and is not causing any of the exceptions listed in items 2-9, it is permissible for both exceptions to be generated and recorded in the DBSR. A single Debug interrupt will result.

5.9.1.3 Exception Priorities for Other Defined Floating-Point Instructions

The following prioritized list of exceptions may occur as a result of the attempted execution of any defined floating-point instruction other than a load or store.

1. Debug (Instruction Address Compare)
2. Instruction TLB Error
3. Instruction Storage Interrupt (all types)
4. Program (Illegal Instruction)
5. Floating-Point Unavailable
6. Program (Unimplemented Operation)
7. Program (Floating-point Enabled)
8. Debug (Instruction Complete)

5.9.1.4 Exception Priorities for Defined Privileged Instructions

The following prioritized list of exceptions may occur as a result of the attempted execution of any defined privileged instruction, except ***dcbi***, ***rfi***, and ***rfci*** instructions.

1. Debug (Instruction Address Compare)
2. Instruction TLB Error
3. Instruction Storage Interrupt (all types)
4. Program (Illegal Instruction)
5. Program (Privileged Instruction)
6. Program (Unimplemented Operation)
7. Debug (Instruction Complete)

For ***mtmsr***, ***mtspr*** (DBCR0, DBCR1, DBCR2), ***mtspr*** (TCR), and ***mtspr*** (TSR), if they are not causing Debug (Instruction Address Compare) nor Program (Privileged Instruction) exceptions, it is possible that they are simultaneously enabling (via mask bits) multiple existing exceptions (and at the same time possibly causing a Debug (Instruction Complete) exception). When this occurs, the interrupts will be handled in the order defined by Section 5.8.2 on page 591.

5.9.1.5 Exception Priorities for Defined Trap Instructions

The following prioritized list of exceptions may occur as a result of the attempted execution of a defined *Trap* instruction.

1. Debug (Instruction Address Compare)
2. Instruction TLB Error

3. Instruction Storage Interrupt (all types)
4. Program (Illegal Instruction)
5. Program (Unimplemented Operation)
6. Debug (Trap)
7. Program (Trap)
8. Debug (Instruction Complete)

If the instruction is causing both a Debug (Instruction Address Compare) and a Debug (Trap), and is not causing any of the exceptions listed in items 2-5, it is permissible for both exceptions to be generated and recorded in the DBSR. A single Debug interrupt will result.

5.9.1.6 Exception Priorities for Defined System Call Instruction

The following prioritized list of exceptions may occur as a result of the attempted execution of a defined *System Call* instruction.

1. Debug (Instruction Address Compare)
2. Instruction TLB Error
3. Instruction Storage Interrupt (all types)
4. Program (Illegal Instruction)
5. Program (Unimplemented Operation)
6. System Call
7. Debug (Instruction Complete)

5.9.1.7 Exception Priorities for Defined Branch Instructions

The following prioritized list of exceptions may occur as a result of the attempted execution of any defined branch instruction.

1. Debug (Instruction Address Compare)
2. Instruction TLB Error
3. Instruction Storage Interrupt (all types)
4. Program (Illegal Instruction)
5. Program (Unimplemented Operation)
6. Debug (Branch Taken)
7. Debug (Instruction Complete)

If the instruction is causing both a Debug (Instruction Address Compare) and a Debug (Branch Taken), and is not causing any of the exceptions listed in items 2-5, it is permissible for both exceptions to be generated and recorded in the DBSR. A single Debug interrupt will result.

5.9.1.8 Exception Priorities for Defined Return From Interrupt Instructions

The following prioritized list of exceptions may occur as a result of the attempted execution of an ***rfi***, ***rfci***, ***rfmci***, ***rfdi*** [Category:Embedded.Enhanced Debug] instruction.

1. Debug (Instruction Address Compare)
2. Instruction TLB Error
3. Instruction Storage Interrupt (all types)
4. Program (Illegal Instruction)
5. Program (Privileged Instruction)
6. Program (Unimplemented Operation)
7. Debug (Return From Interrupt)
8. Debug (Instruction Complete)

If the ***rfi*** or ***rfci***, ***rfmci***, or ***rfdi*** [Category: Embedded.Enhanced Debug] instruction is causing both a Debug (Instruction Address Compare) and a Debug (Return From Interrupt), and is not causing any of the exceptions listed in items 2-5, it is permissible for both exceptions to be generated and recorded in the DBSR. A single Debug interrupt will result.

5.9.1.9 Exception Priorities for Other Defined Instructions

The following prioritized list of exceptions may occur as a result of the attempted execution of all other instructions not listed above.

1. Debug (Instruction Address Compare)
2. Instruction TLB Error
3. Instruction Storage Interrupt (all types)
4. Program (Illegal Instruction)
5. Program (Unimplemented Operation)
6. Debug (Instruction Complete)

5.9.2 Exception Priorities for Reserved Instructions

The following prioritized list of exceptions may occur as a result of the attempted execution of any reserved instruction.

1. Debug (Instruction Address Compare)
2. Instruction TLB Error
3. Instruction Storage Interrupt (all types)
4. Program (Illegal Instruction)

Chapter 6. Reset and Initialization

6.1 Background

This chapter describes the requirements for processor reset. This includes both the means of causing reset, and the specific initialization that is required to be performed automatically by the processor hardware. This chapter also provides an overview of the operations that should be performed by initialization software, in order to fully initialize the processor.

In general, the specific actions taken by a processor upon reset are implementation-dependent. Also, it is the responsibility of system initialization software to initialize the majority of processor and system resources after reset. Implementations are required to provide a minimum processor initialization such that this system software may be fetched and executed, thereby accomplishing the rest of system initialization.

6.2 Reset Mechanisms

This specification defines two processor mechanisms for internally invoking a reset operation using either the Watchdog Timer (see Section 7.7 on page 602) or the Debug facilities using $DBCR0_{RST}$ (see Section 8.5.1.1 on page 613). In addition, implementations will typically provide additional means for invoking a reset operation, via an external mechanism such as a signal pin which when activated will cause the processor to reset.

6.3 Processor State After Reset

The initial processor state is controlled by the register contents after reset. In general, the contents of most registers are undefined after reset.

The processor hardware is only guaranteed to initialize those registers (or specific bits in registers) which must be initialized in order for software to be able to reliably perform the rest of system initialization.

The Machine State Register and Processor Version Register and a TLB entry are updated as follows:

Machine State Register

Bit	Setting	Comments
CM	0	Computation Mode (set to 32-bit mode)
ICM	0	Interrupt Computation Mode (set to 32-bit)
UCLE	0	User Cache Locking Enable
SPV	0	SPE/Embedded Floating-Point/ Vector Unavailable
WE	0	Wait State disabled
CE	0	Critical Input interrupts disabled
DE	0	Debug interrupts disabled
EE	0	External Input interrupts disabled
PR	0	Supervisor mode
FP	0	FP unavailable
ME	0	Machine Check interrupts disabled
FE0	0	FP exception type Program interrupts disabled
FE1	0	FP exception type Program interrupts disabled
IS	0	Instruction Address Space 0
DS	0	Data Address Space 0
PMM	0	Performance Monitor Mark

Figure 17. Machine State Register Initial Values

Processor Version Register

Implementation-Dependent. (This register is read-only, and contains a value which identifies the specific implementation)

TLB entry

A TLB entry (which entry is implementation-dependent) is initialized in an implementation-dependent manner that maps the last 4KB page in the implemented effective storage address space, with the following field settings:

Field	Setting	Comments
EPN	see below	Represents the last 4K page in effective address space
RPN	see below	Represents the last 4K page in physical address space
TS	0	translation address space 0
SIZE	0b0001	4KB page size
W	?	implementation-dependent value
I	?	implementation-dependent value
M	?	implementation-dependent value
G	?	implementation-dependent value
E	?	implementation-dependent value
U0	?	implementation-dependent value
U1	?	implementation-dependent value
U2	?	implementation-dependent value
U3	?	implementation-dependent value
TID	?	implementation-dependent value, but page must be accessible
UX	?	implementation-dependent value
UR	?	implementation-dependent value
UW	?	implementation-dependent value
SX	1	page is execute accessible in supervisor mode
SR	1	page is read accessible in supervisor mode
SW	1	page is write accessible in supervisor mode
VLE	?	implementation-dependent value
ACM	?	implementation-dependent value

Figure 18. TLB Initial Values

The initial settings of EPN and RPN are dependent upon the number of bits implemented in the EPN and RPN fields and the minimum page size supported by the implementation. For example, an implementation that allows 1KB pages and 32 bits of effective address would implement a 22 bit EPN and set the initial value of the boot entry to 2^{22}-4 (0x3FFC) while an implementation that supports only 4K pages as the smallest size and 32 bits of effective address would implement a 20 bit EPN and set the initial value of the boot entry to 2^{20}-1 (0xFFFF).

Instruction execution begins at the last word address of the page mapped by the boot TLB entry. Note that this address is different from the PowerPC Architecture System Reset interrupt vector.

An implementation may provide additional methods for initializing the TLB entry used for initial boot by providing an implementation-dependent RPN, or initializing other TLB entries.

6.4 Software Initialization Requirements

When reset occurs, the processor is initialized to a minimum configuration to start executing initialization code. Initialization code is necessary to complete the processor and system configuration. The initialization code described in this section is the minimum recommended for configuring the processor to run application code.

Initialization code should configure the following processor resources:

- Invalidate the instruction cache and data cache (implementation-dependent).
- Initialize system memory as required by the operating system or application code.
- Initialize the Interrupt Vector Prefix Register and Interrupt Vector Offset Register.
- Initialize other processor registers as needed by the system.
- Initialize off-chip system facilities.
- Dispatch the operating system or application code.

Chapter 7. Timer Facilities

7.1 Overview

The Time Base, Decrementer, Fixed-interval Timer, and Watchdog Timer provide timing functions for the system. The remainder of this section describes these registers and related facilities.

7.2 Time Base (TB)

The Time Base (TB) is a 64-bit register (see Figure 19) containing a 64-bit unsigned integer that is incremented periodically. Each increment adds 1 to the low-order bit (bit 63). The frequency at which the integer is updated is implementation-dependent.

TBU (bits 0–31)	TBL (bits 32–63)

Field	Description
TBU	Upper 32 bits of Time Base
TBL	Lower 32 bits of Time Base

Figure 19. Time Base

The Time Base 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 period of the Time Base depends on the driving frequency. As an order of magnitude example, suppose that the CPU clock is 1 GHz and that the Time Base is driven by this frequency divided by 32. Then the period of the Time Base would be

$$T_{TB} = \frac{2^{64} \times 32}{1\ \text{GHz}} = 5.90 \times 10^{11}\ \text{seconds}$$

which is approximately 18,700 years.

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 user mode (MSR_{PR}=1). If the means is under software control, it must be privileged. There must be a method for getting all processors' Time Bases to start incrementing with values that are identical or almost identical in all processors.

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.

See the description of the Time Base in Book II, for ways to compute time of day in POSIX format from the Time Base.

7.2.1 Writing the Time Base

Writing the Time Base is privileged. Reading the Time Base is not privileged; it is discussed in 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; see Appendix B, “Assembler Extended Mnemonics” on page 635.

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.

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.

7.3 Decrementer

The Decrementer (DEC) is a 32-bit decrementing counter that provides a mechanism for causing a Decrementer interrupt after a programmable delay. The contents of the Decrementer are treated as a signed integer.

DEC
32 … 63

Figure 20. Decrementer

The Decrementer is driven by the same frequency as the Time Base. The period of the Decrementer will depend on the driving frequency, but if the same values are used as given above for the Time Base (see Section 7.2), and if the Time Base update frequency is constant, the period would be

$$T_{\mathrm{DEC}} = \frac{2^{32} \times 32}{1\ \mathrm{GHz}} = 137 \text{ seconds.}$$

The Decrementer counts down.

The operation of the Decrementer satisfies the following constraints.

1. The operation of the Time Base and the Decrementer is coherent, i.e., the counters are driven by the same fundamental time base.
2. Loading a GPR from the Decrementer has no effect on the accuracy of the Time Base.
3. 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.

7.3.1 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 (see Appendix B, “Assembler Extended Mnemonics” on page 635), the Decrementer can be written from GPR Rx using:

```
mtdec  Rx
```

The Decrementer can be read into GPR Rx using:

```
mfdec  Rx
```

Copying the Decrementer to a GPR has no effect on the Decrementer contents or on the interrupt mechanism.

7.3.2 Decrementer Events

A Decrementer event occurs when a decrement occurs on a Decrementer value of 0x0000_0001.

Upon the occurrence of a Decrementer event, the Decrementer may be reloaded from a 32-bit Decrementer Auto-Reload Register (DECAR). See Section 7.4. Upon the occurrence of a Decrementer event, the Decrementer has the following basic modes of operation.

Decrement to one and stop on zero

If TCR_{ARE}=0, TSR_{DIS} is set to 1, the value 0x0000_0000 is then placed into the DEC, and the Decrementer stops decrementing.

If enabled by TCR_{DIE}=1 and MSR_{EE}=1, a Decrementer interrupt is taken. See Section 5.6.11, “Decrementer Interrupt” on page 582 for details of register behavior caused by the Decrementer interrupt.

Decrement to one and auto-reload

If TCR_{ARE}=1, TSR_{DIS} is set to 1, the contents of the Decrementer Auto-Reload Register is then placed into the DEC, and the Decrementer continues decrementing from the reloaded value.

If enabled by TCR_{DIE}=1 and MSR_{EE}=1, a Decrementer interrupt is taken. See Section 5.6.11, “Decrementer Interrupt” on page 582 for details of register behavior caused by the Decrementer interrupt.

Forcing the Decrementer to 0 using the ***mtspr*** instruction will not cause a Decrementer exception; however, decrementing which was in progress at the instant of the ***mtspr*** may cause the exception. To eliminate the Decrementer as a source of exceptions, set TCR_{DIE} to 0 (clear the Decrementer Interrupt Enable bit).

If it is desired to eliminate all Decrementer activity, the procedure is as follows:

1. Write 0 to TCR_{DIE}. This will prevent Decrementer activity from causing exceptions.
2. Write 0 to TCR_{ARE} to disable the Decrementer auto-reload.
3. Write 0 to Decrementer. This will halt Decrementer decrementing. While this action will not cause a Decrementer exception to be set in TSR_{DIS}, a near simultaneous decrement may have done so.
4. Write 1 to TSR_{DIS}. This action will clear TSR_{DIS} to 0 (see Section 7.5.1 on page 601). This will clear any Decrementer exception which may be pending. Because the Decrementer is frozen at zero, no further Decrementer events are possible.

If the auto-reload feature is disabled (TCR_{ARE}=0), then once the Decrementer decrements to zero, it will stay there until software reloads it using the ***mtspr*** instruction.

On reset, TCR_{ARE} is set to 0. This disables the auto-reload feature.

7.4 Decrementer Auto-Reload Register

The Decrementer Auto-Reload Register is a 32-bit register as shown below.

DECAR	
32	63

Figure 21. Decrementer

Bits of the decrementer auto-reload register are numbered 32 (most-significant bit) to 63 (least-significant bit). The Decrementer Auto-Reload Register is provided to support the auto-reload feature of the Decrementer. See Section 7.3.2

The contents of the Decrementer Auto-Reload Register cannot be read. The contents of bits 32:63 of register RS can be written to the Decrementer Auto-Reload Register using the ***mtspr*** instruction.

7.5 Timer Control Register

The Timer Control Register (TCR) is a 32-bit register. Timer Control Register bits are numbered 32 (most-significant bit) to 63 (least-significant bit). The Timer Control Register controls Decrementer (see Section 7.3), Fixed-Interval Timer (see Section 7.6), and Watchdog Timer (see Section 7.7) options.

The relationship of the Timer facilities to the TCR and TB is shown in the figure below.

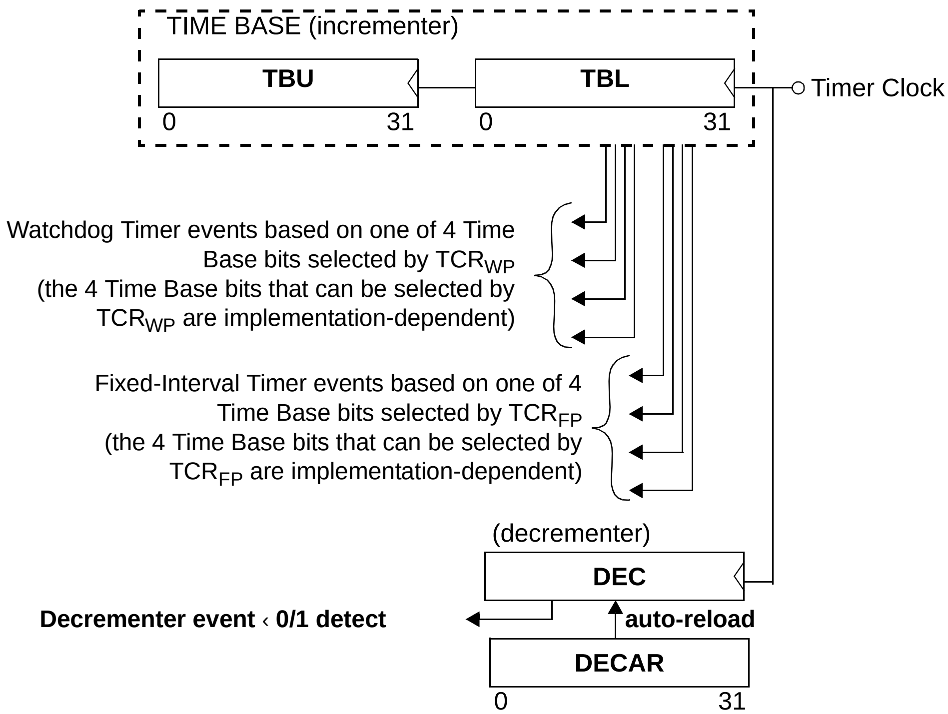


Figure 22. Relationships of the Timer Facilities

The contents of the Timer Control Register can be read using the ***mfspr*** instruction. The contents of bits 32:63 of register RS can be written to the Timer Control Register using the ***mtspr*** instruction.

The contents of the TCR are defined below:

Bit(s) Description

32:33 ***Watchdog Timer Period*** (WP) (see Section 7.7 on page 602)

Specifies one of 4 bit locations of the Time Base used to signal a Watchdog Timer exception on a transition from 0 to 1. The 4 Time Base bits that can be specified to serve as the Watchdog Timer period are implementation-dependent.

34:35 ***Watchdog Timer Reset Control*** (WRC) (see Section 7.7 on page 602)

00 No Watchdog Timer reset will occur

TCR_{WRC} resets to 0b00. This field may be set by software, but cannot be cleared by software (except by a software-induced reset)

01-11
Force processor to be reset on second time-out of Watchdog Timer. The exact function of any of these settings is implementation-dependent.

The Watchdog Timer Reset Control field is cleared to zero by processor reset. These bits are set only by software. Once a 1 has been written to one of these bits, that bit remains a 1 until a reset occurs. This is to prevent errant code from disabling the Watchdog reset function.

36 ***Watchdog Timer Interrupt Enable*** (WIE) (see Section 7.7 on page 602)

0 Disable Watchdog Timer interrupt
1 Enable Watchdog Timer interrupt

37 ***Decrementer Interrupt Enable*** (DIE) (see Section 7.3 on page 599)

0 Disable Decrementer interrupt
1 Enable Decrementer interrupt

38:39 ***Fixed-Interval Timer Period*** (FP) (see Section 7.6 on page 602)

Specifies one of 4 bit locations of the Time Base used to signal a Fixed-Interval Timer exception on a transition from 0 to 1. The 4 Time Base bits that can be specified to serve as the Fixed-Interval Timer period are implementation-dependent.

40 ***Fixed-Interval Timer Interrupt Enable*** (FIE) (see Section 7.6 on page 602

0 Disable Fixed-Interval Timer interrupt
1 Enable Fixed-Interval Timer interrupt

41 ***Auto-Reload Enable*** (ARE)

0 Disable auto-reload of the Decrementer

Decrementer exception is presented (i.e. TSR_{DIS} is set to 1) when the Decrementer is decremented from a value of 0x0000_0001. The next value placed in the Decrementer is the value 0x0000_0000. The Decrementer then stops decrementing. If MSR_{EE}=1, TCR_{DIE}=1, and TSR_{DIS}=1, a Decrementer interrupt is taken. Software must reset TSR_{DIS}.

1 Enable auto-reload of the Decrementer

Decrementer exception is presented (i.e. TSR_{DIS} is set to 1) when the Decrementer is decremented from a value of 0x0000_0001. The contents of the Decrementer Auto-Reload Register is placed in the Decrementer. The Decrementer resumes decrementing. If MSR_{EE}=1, TCR_{DIE}=1, and TSR_{DIS}=1, a Decrementer interrupt is taken. Software must reset TSR_{DIS}.

42 Implementation-dependent

43:63 Reserved

7.5.1 Timer Status Register

The Timer Status Register (TSR) is a 32-bit register. Timer Status Register bits are numbered 32 (most-significant bit) to 63 (least-significant bit). The Timer Status Register contains status on timer events and the most recent Watchdog Timer-initiated processor reset.

The Timer Status Register is set via hardware, and read and cleared via software. The contents of the Timer Status Register can be read using the ***mfspr*** instruction. Bits in the Timer Status Register can be cleared using the ***mtspr*** instruction. Clearing is done by writing bits 32:63 of a General Purpose Register to the Timer Status Register with a 1 in any bit position that is to be cleared and 0 in all other bit positions. The write-data to the Timer Status Register is not direct data, but a mask. A 1 causes the bit to be cleared, and a 0 has no effect.

The contents of the TSR are defined below:

Bit(s) Description

32 ***Enable Next Watchdog Timer*** (ENW) (see Section 7.7 on page 602)

0 Action on next Watchdog Timer time-out is to set TSRENW
1 Action on next Watchdog Timer time-out is governed by TSRWIS

33 ***Watchdog Timer Interrupt Statu***s (WIS) (see Section 7.7 on page 602)

0 A Watchdog Timer event has not occurred.
1 A Watchdog Timer event has occurred. When MSR_{CE}=1 and TCR_{WIE}=1, a Watchdog Timer interrupt is taken.

34:35 ***Watchdog Timer Reset Statu***s (WRS) (see Section 7.7 on page 602)

These two bits are set to one of three values when a reset is caused by the Watchdog Timer. These bits are undefined at power-up.

00 No Watchdog Timer reset has occurred.
01 Implementation-dependent reset information.
10 Implementation-dependent reset information.
11 Implementation-dependent reset information.

36 ***Decrementer Interrupt Status*** (DIS) (see Section 7.3.2 on page 599)

0 A Decrementer event has not occurred.
1 A Decrementer event has occurred. When MSR_{EE}=1 and TCR_{DIE}=1, a Decrementer interrupt is taken.

37 ***Fixed-Interval Timer Interrupt Status*** (FIS) (see Section 7.6 on page 602)

0 A Fixed-Interval Timer event has not occurred.
1 A Fixed-Interval Timer event has occurred. When MSR_{EE}=1 and TCR_{FIE}=1, a Fixed-Interval Timer interrupt is taken.

38:63 Reserved

7.6 Fixed-Interval Timer

The Fixed-Interval Timer (FIT) is a mechanism for providing timer interrupts with a repeatable period, to facilitate system maintenance. It is similar in function to an auto-reload Decrementer, except that there are fewer selections of interrupt period available. The Fixed-Interval Timer exception occurs on 0 to 1 transitions of a selected bit from the Time Base (see Section 7.5).

The Fixed-Interval Timer exception is logged by TSR_{FIS}. A Fixed-Interval Timer interrupt will occur if TCR_{FIE} and MSR_{EE} are enabled. See Section 5.6.12 on page 582 for details of register behavior caused by the Fixed-Interval Timer interrupt.

Note that a Fixed-Interval Timer exception will also occur if the selected Time Base bit transitions from 0 to 1 due to an ***mtspr*** instruction that writes a 1 to the bit when its previous value was 0.

7.7 Watchdog Timer

The Watchdog Timer is a facility intended to aid system recovery from faulty software or hardware. Watchdog time-outs occur on 0 to 1 transitions of selected bits from the Time Base (Section 7.5).

When a Watchdog Timer time-out occurs while Watchdog Timer Interrupt Status is clear (TSR_{WIS} = 0) and the next Watchdog Time-out is enabled (TSR_{ENW} = 1), a Watchdog Timer exception is generated and logged by setting TSR_{WIS} to 1. This is referred to as a Watchdog Timer First Time Out. A Watchdog Timer interrupt will occur if enabled by TCR_{WIE} and MSR_{CE}. See Section 5.6.13 on page 582 for details of register behavior caused by the Watchdog Timer interrupt. The purpose of the Watchdog Timer First time-out is to give an indication that there may be problem and give the system a chance to perform corrective action or capture a failure before a reset occurs from the Watchdog Timer Second time-out as explained further below.

Note that a Watchdog Timer exception will also occur if the selected Time Base bit transitions from 0 to 1 due to an ***mtspr*** instruction that writes a 1 to the bit when its previous value was 0.

When a Watchdog Timer time-out occurs while TSR_{WIS} = 1 and TSR_{ENW} = 1, a processor reset occurs if it is enabled by a non-zero value of the Watchdog Reset Control field in the Timer Control Register (TCR_{WRC}). This is referred to as a Watchdog Timer Second Time Out. The assumption is that TSR_{WIS} was not cleared because the processor was unable to execute the Watchdog Timer interrupt handler, leaving reset as the only available means to restart the system. Note that once TCR_{WRC} has been set to a non-zero value, it cannot be reset by software; this feature prevents errant software from disabling the Watchdog Timer reset capability.

A more complete view of Watchdog Timer behavior is afforded by Figure 23 and Table 24, which describe the Watchdog Timer state machine and Watchdog Timer controls. The numbers in parentheses in the figure refer to the discussion of modes of operation which follow the table.

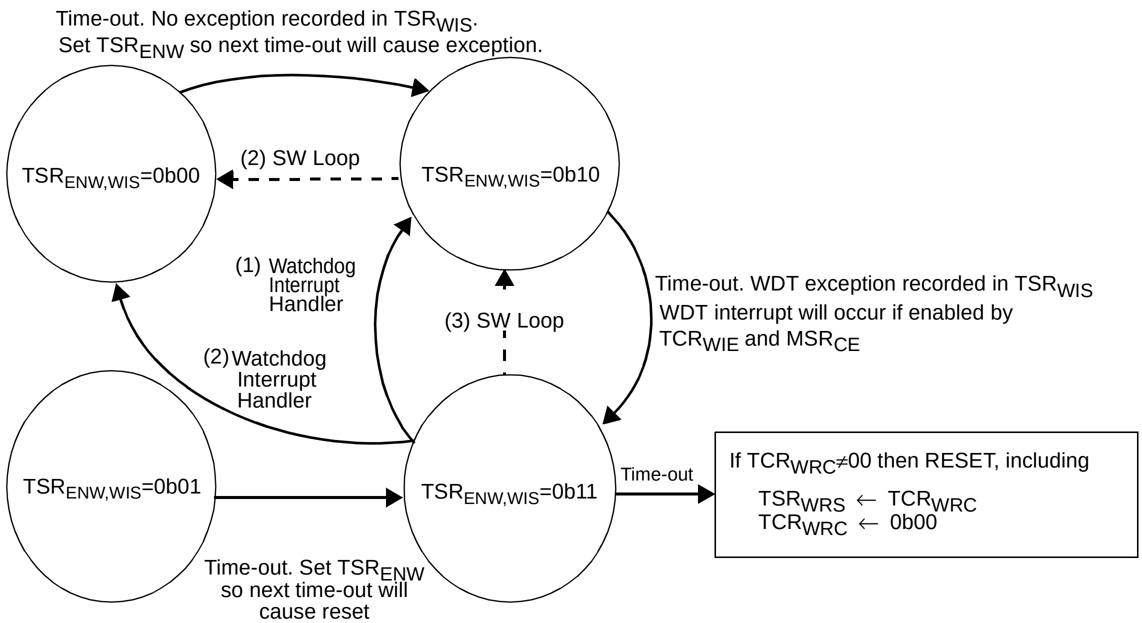


Figure 23. Watchdog State Machine

Enable Next WDT (TSR_{ENW})	WDT Status (TSR_{WIS})	Action when timer interval expires
0	0	Set Enable Next Watchdog Timer (TSR_{ENW}=1).
0	1	Set Enable Next Watchdog Timer (TSR_{ENW}=1).
1	0	Set Watchdog Timer interrupt status bit (TSR_{WIS}=1). If Watchdog Timer interrupt is enabled (TCR_{WIE}=1 and MSR_{CE}=1), then interrupt.
1	1	Cause Watchdog Timer reset action specified by TCR_{WRC}. Reset will copy pre-reset TCR_{WRC} into TSR_{WRS}, then clear TCR_{WRC}.

Figure 24. Watchdog Timer Controls

The controls described in the above table imply three different modes of operation that a programmer might select for the Watchdog Timer. Each of these modes assumes that TCR_{WRC} has been set to allow processor reset by the Watchdog facility:

1. Always take the Watchdog Timer interrupt when pending, and never attempt to prevent its occurrence. In this mode, the Watchdog Timer interrupt caused by a first time-out is used to clear TSR_{WIS} so a second time-out never occurs. TSR_{ENW} is not cleared, thereby allowing the next time-out to cause another interrupt.

2. Always take the Watchdog Timer interrupt when pending, but avoid when possible. In this mode a recurring code loop of reliable duration (or perhaps a periodic interrupt handler such as the Fixed-Interval Timer interrupt handler) is used to repeatedly clear TSR_{ENW} such that a first time-out exception is avoided, and thus no Watchdog Timer interrupt occurs. Once TSR_{ENW} has been cleared, software has between one and two full Watchdog periods before a Watchdog exception will be posted in TSR_{WIS}. If this occurs before the software is able to clear TSR_{ENW} again, a Watchdog Timer interrupt will occur. In this case, the Watchdog Timer interrupt handler will then clear both TSR_{ENW} and TSR_{WIS}, in order to (hopefully) avoid the next Watchdog Timer interrupt.

3. Never take the Watchdog Timer interrupt. In this mode, Watchdog Timer interrupts are disabled (via TCR_{WIE}=0), and the system depends upon a

recurring code loop of reliable duration (or perhaps a periodic interrupt handler such as the Fixed-Interval Timer interrupt handler) to repeatedly clear TSR_{WIS} such that a second time-out is avoided, and thus no reset occurs. TSR_{ENW} is not cleared, thereby allowing the next time-out to set TSR_{WIS} again. The recurring code loop must have a period which is less than one Watchdog Timer period in order to guarantee that a Watchdog Timer reset will not occur.

7.8 Freezing the Timer Facilities

The debug mechanism provides a means of temporarily freezing the timers upon a debug event. Specifically, the Time Base and Decrementer can be frozen and prevented from incrementing/decrementing, respectively, whenever a debug event is set in the Debug Status Register. Note that this also freezes the FIT and Watchdog timer. This allows a debugger to simulate the appearance of ‘real time’, even though the application has been temporarily ‘halted’ to service the debug event. See the description of bit 63 of the Debug Control Register 0 (Freeze Timers on Debug Event or $DBCR0_{FT}$) in Section 8.5.1.1 on page 613.

Chapter 8. Debug Facilities

8.1 Overview

Processors provide debug facilities to enable hardware and software debug functions, such as instruction and data breakpoints and program single stepping. The debug facilities consist of a set of Debug Control Registers (DBCR0, DBCR1, and DBCR2) (see Section 8.5.1 on page 613), a set of Address and Data Value Compare Registers (IAC1, IAC2, IAC3, IAC4, DAC1, DAC2, DVC1, and DVC2), (see Section 8.4.3, Section 8.4.4, and Section 8.4.5), a Debug Status Register (DBSR) (see Section 8.5.2) for enabling and recording various kinds of debug events, and a special Debug interrupt type built into the interrupt mechanism (see Section 5.6.16). The debug facilities also provide a mechanism for software-controlled processor reset, and for controlling the operation of the timers in a debug environment.

The ***mfspr*** and ***mtspr*** instructions (see Section 3.4.1) provide access to the registers of the debug facilities.

In addition to the facilities described here, implementations will typically include debug facilities, modes, and access mechanisms which are implementation-specific. For example, implementations will typically provide access to the debug facilities via a dedicated interface such as the IEEE 1149.1 Test Access Port (JTAG).

8.2 Internal Debug Mode

Debug events include such things as instruction and data breakpoints. These debug events cause status bits to be set in the Debug Status Register. The existence of a set bit in the Debug Status Register is considered a Debug exception. Debug exceptions, if enabled, will cause Debug interrupts.

There are two different mechanisms that control whether Debug interrupts are enabled. The first is the MSR_{DE} bit, and this bit must be set to 1 to enable

Debug interrupts. The second mechanism is an enable bit in the Debug Control Register 0 (DBCR0). This bit is the Internal Debug Mode bit ($DBCR0_{IDM}$), and it must also be set to 1 to enable Debug interrupts.

When $DBCR0_{IDM}$=1, the processor is in Internal Debug Mode. In this mode, debug events will (if also enabled by MSR_{DE}) cause Debug interrupts. Software at the Debug interrupt vector location will thus be given control upon the occurrence of a debug event, and can access (via the normal instructions) all architected processor resources. In this fashion, debug monitor software can control the processor and gather status, and interact with debugging hardware connected to the processor.

When the processor is not in Internal Debug Mode ($DBCR0_{IDM}$=0), debug events may still occur and be recorded in the Debug Status Register. These exceptions may be monitored via software by reading the Debug Status Register (using ***mfspr***), or may eventually cause a Debug interrupt if later enabled by setting $DBCR0_{IDM}$=1 (and MSR_{DE}=1). Processor behavior when debug events occur while $DBCR0_{IDM}$=0 is implementation-dependent.

8.3 External Debug Mode [Category: Embedded.Enhanced Debug]

The External Debug Mode is a mode in which facilities external to the processor can access processor resources and control execution. These facilities are defined as the *external debug* facilities and are not defined here, however some instructions and registers share internal and external debug roles and are briefly described as necessary.

A ***dnh*** instruction is provided to stop instruction fetching and execution and allow the processor to be managed by an external debug facility. After the ***dnh*** instruction is executed, instructions are not fetched, interrupts are not taken, and the processor does not execute instructions.

8.4 Debug Events

Debug events are used to cause Debug exceptions to be recorded in the Debug Status Register (see Section 8.5.2). In order for a debug event to be enabled to set a Debug Status Register bit and thereby cause a Debug exception, the specific event type must be enabled by a corresponding bit or bits in the Debug Control Register DBCR0 (see Section 8.5.1.1), DBCR1 (see Section 8.5.1.2), or DBCR2 (see Section 8.5.1.3), in most cases; the Unconditional Debug Event (UDE) is an exception to this rule. Once a Debug Status Register bit is set, if Debug interrupts are enabled by MSR_{DE}, a Debug interrupt will be generated.

Certain debug events are not allowed to occur when MSR_{DE}=0. In such situations, no Debug exception occurs and thus no Debug Status Register bit is set. Other debug events may cause Debug exceptions and set Debug Status Register bits regardless of the state of MSR_{DE}. The associated Debug interrupts that result from such Debug exceptions will be delayed until MSR_{DE}=1, provided the exceptions have not been cleared from the Debug Status Register in the meantime.

Any time that a Debug Status Register bit is allowed to be set while MSR_{DE}=0, a special Debug Status Register bit, Imprecise Debug Event ($DBSR_{IDE}$), will also be set. $DBSR_{IDE}$ indicates that the associated Debug exception bit in the Debug Status Register was set while Debug interrupts were disabled via the $MMSR_{DE}$ bit. Debug interrupt handler software can use this bit to determine whether the address recorded in CSRR0/DSRR0 [Category: Embedded.Enhanced Debug] should be interpreted as the address associated with the instruction causing the Debug exception, or simply the address of the instruction after the one which set the MSR_{DE} bit, thereby enabling the delayed Debug interrupt.

Debug interrupts are ordered with respect to other interrupt types (see Section 7.8 on page 179). Debug exceptions are prioritized with respect to other exceptions (see Section 7.9 on page 183).

There are eight types of debug events defined:

1. Instruction Address Compare debug events
2. Data Address Compare debug events
3. Trap debug events
4. Branch Taken debug events
5. Instruction Complete debug events
6. Interrupt Taken debug events
7. Return debug events
8. Unconditional debug events

Programming Note

There are two classes of debug exception types:

Type 1: exception before instruction

Type 2: exception after instruction

Almost all debug exceptions fall into the first type. That is, they all take the interrupt upon encountering an instruction having the exception without updating any architectural state (other than DBSR, CSRR0/DSRR0 [Category: Embedded.Enhanced Debug], CSRR1/DSRR1 [Category: Embedded.Enhanced Debug], MSR) for that instruction.

The CSRR0/DSRR0 [Category: Embedded.Enhanced Debug] for this type of exception points to the instruction that encountered the exception. This includes IAC, DAC, branch taken, etc.

The only exception which fall into the second type is the instruction complete debug exception. This exception is taken upon completing and updating one instruction and then pointing CSRR0/DSRR0 [Category: Embedded.Enhanced Debug] to the next instruction to execute.

To make forward progress for any Type 1 debug exception one does the following:

1. Software sets up Type 1 exceptions (e.g. branch taken debug exceptions) and then returns to normal program operation
2. Hardware takes Debug interrupt upon the first branch taken Debug exception, pointing to the branch with CSRR0/DSRR0 [Category: Embedded.Enhanced Debug].
3. Software, in the debug handler, sees the branch taken exception type, does whatever logging/analysis it wants to, then clears all debug event enables in the DBCR except for the instruction complete debug event enable.
4. Software does an ***rfci*** or ***rfdi*** [Category: Embedded.Enhanced Debug].
5. Hardware would execute and complete one instruction (the branch taken in this case), and then take a Debug interrupt with CSRR0/DSRR0 [Category: Embedded.Enhanced Debug] pointing to the target of the branch.
6. Software would see the instruction complete interrupt type. It clears the instruction complete event enable, then enables the branch taken interrupt event again.
7. Software does an ***rfci*** *or* ***rfdi*** [Category: Embedded.Enhanced Debug].
8. Hardware resumes on the target of the taken branch and continues until another taken branch, in which case we end up at step 2 again.

This, at first, seems like a double tax (i.e. 2 debug interrupts for every instance of a Type 1 exception), but there doesn't seem like any other clean way to make forward progress on Type 1 debug exceptions. The only other way to avoid the double tax is to have the debug handler routine actually emulate the instruction pointed to for the Type 1 exceptions, determine the next instruction that would have been executed by the interrupted program flow and load the CSRR0/DSRR0 [Category: Embedded.Enhanced Debug] with that address and do an ***rfci/rfdi*** [Category: Embedded.Enhanced Debug]; this is probably not faster.

8.4.1 Instruction Address Compare Debug Event

One or more Instruction Address Compare debug events (IAC1, IAC2, IAC3 or IAC4) occur if they are enabled and execution is attempted of an instruction at an address that meets the criteria specified in the DBCR0, DBCR1, IAC1, IAC2, IAC3, and IAC4 Registers.

Instruction Address Compare User/Supervisor Mode

$DBCR1_{IAC1US}$ specifies whether IAC1 debug events can occur in user mode or supervisor mode, or both.

$DBCR1_{IAC2US}$ specifies whether IAC2 debug events can occur in user mode or supervisor mode, or both.

$DBCR1_{IAC3US}$ specifies whether IAC3 debug events can occur in user mode or supervisor mode, or both.

$DBCR1_{IAC4US}$ specifies whether IAC4 debug events can occur in user mode or supervisor mode, or both.

Effective/Real Address Mode

$DBCR1_{IAC1ER}$ specifies whether effective addresses, real addresses, effective addresses and MSR_{IS}=0, or effective addresses and MSR_{IS}=1 are used in determining an address match on IAC1 debug events.

$DBCR1_{IAC2ER}$ specifies whether effective addresses, real addresses, effective addresses and MSR_{IS}=0, or effective addresses and MSR_{IS}=1 are used in determining an address match on IAC2 debug events.

$DBCR1_{IAC3ER}$ specifies whether effective addresses, real addresses, effective addresses and MSR_{IS}=0, or effective addresses and MSR_{IS}=1 are used in determining an address match on IAC3 debug events.

$DBCR1_{IAC4ER}$ specifies whether effective addresses, real addresses, effective addresses and MSR_{IS}=0, or effective addresses and MSR_{IS}=1 are used in determining an address match on IAC4 debug events.

Instruction Address Compare Mode

$DBCR1_{IAC12M}$ specifies whether all or some of the bits of the address of the instruction fetch must match the contents of the IAC1 or IAC2, whether the address must be inside a specific range specified by the IAC1 and IAC2 or outside a specific range specified by the IAC1 and IAC2 for an IAC1 or IAC2 debug event to occur.

$DBCR1_{IAC34M}$ specifies whether all or some of the bits of the address of the instruction fetch must match the contents of the IAC3 Register or IAC4 Register, whether the address must be inside a specific range specified by the IAC3 Register and IAC4 Register or outside a specific range specified by the IAC3 Register and IAC4 Register for an IAC3 or IAC4 debug event to occur.

There are four instruction address compare modes.

There are four instruction address compare modes.

- *Exact address compare mode*
  If the address of the instruction fetch is equal to the value in the enabled IAC Register, an instruction address match occurs. For 64-bit implementations, the addresses are masked to compare only bits 32:63 when the processor is executing in 32-bit mode.
- *Address bit match mode*
  For IAC1 and IAC2 debug events, if the address of the instruction fetch access, ANDed with the contents of the IAC2, are equal to the contents of the IAC1, also ANDed with the contents of the IAC2, an instruction address match occurs.

  For IAC3 and IAC4 debug events, if the address of the instruction fetch, ANDed with the contents of the IAC4, are equal to the contents of the IAC3, also ANDed with the contents of the IAC4, an instruction address match occurs.

  For 64-bit implementations, the addresses are masked to compare only bits 32:63 when the processor is executing in 32-bit mode.
- *Inclusive address range compare mode*
  For IAC1 and IAC2 debug events, if the 64-bit address of the instruction fetch is greater than or equal to the contents of the IAC1 and less than the contents of the IAC2, an instruction address match occurs.

  For IAC3 and IAC4 debug events, if the 64-bit address of the instruction fetch is greater than or equal to the contents of the IAC3 and less than the contents of the IAC4, an instruction address match occurs.
- For 64-bit implementations, the addresses are masked to compare only bits 32:63 when the processor is executing in 32-bit mode.
- *Exclusive address range compare mode*
  For IAC1 and IAC2 debug events, if the 64-bit address of the instruction fetch is less than the contents of the IAC1 or greater than or equal to the contents of the IAC2, an instruction address match occurs.

  For IAC3 and IAC4 debug events, if the 64-bit address of the instruction fetch is less than the contents of the IAC3 or greater than or equal to the contents of the IAC4, an instruction address match occurs.

  For 64-bit implementations, the addresses are masked to compare only bits 32:63 when the processor is executing in 32-bit mode.

See the detailed description of DBCR0 (see Section 8.5.1.1, “Debug Control Register 0 (DCBR0)” on page 613) and DBCR1 (see Section 8.5.1.2, “Debug Control Register 1 (DCBR1)” on page 614) and the modes for detecting IAC1, IAC2, IAC3 and IAC4 debug events. Instruction Address Compare debug events can occur regardless of the setting of MSR_{DE} or $DBCR0_{IDM}$.

When an Instruction Address Compare debug event occurs, the corresponding $DBSR_{IAC1}$, $DBSR_{IAC2}$, $DBSR_{IAC3}$, or $DBSR_{IAC4}$ bit or bits are set to record the debug exception. If MSR_{DE}=0, $DBSR_{IDE}$ is also set to 1 to record the imprecise debug event.

If MSR_{DE}=1 (i.e. Debug interrupts are enabled) at the time of the Instruction Address Compare debug exception, a Debug interrupt will occur immediately (provided there exists no higher priority exception which is enabled to cause an interrupt). The execution of the instruction causing the exception will be suppressed, and CSRR0/DSRR0 [Category: Embedded.Enhanced Debug] will be set to the address of the excepting instruction.

If MSR_{DE}=0 (i.e. Debug interrupts are disabled) at the time of the Instruction Address Compare debug exception, a Debug interrupt will not occur, and the instruction will complete execution (provided the instruction is not causing some other exception which will generate an enabled interrupt).

Later, if the debug exception has not been reset by clearing $DBSR_{IAC1}$, $DBSR_{IAC2}$, $DBSR_{IAC3}$, and $DBSR_{IAC4}$, and MSR_{DE} is set to 1, a delayed Debug interrupt will occur. In this case, CSRR0/DSRR0 [Category: Embedded.Enhanced Debug will contain the address of the instruction after the one which enabled the Debug interrupt by setting MSR_{DE} to 1. Software in the Debug interrupt handler can observe $DBSR_{IDE}$ to determine how to interpret the value in CSRR0/DSRR0 [Category: Embedded.Enhanced Debug.

8.4.2 Data Address Compare Debug Event

One or more Data Address Compare debug events (DAC1R, DAC1W, DAC2R, DAC2W) occur if they are enabled, execution is attempted of a data storage access instruction, and the type, address, and possibly even the data value of the data storage access meet the criteria specified in the Debug Control Register 0, Debug Control Register 2, and the DAC1, DAC2, DVC1, and DVC2 Registers.

Data Address Compare Read/Write Enable

$DBCR0_{DAC1}$ specifies whether DAC1R debug events can occur on read-type data storage accesses and whether DAC1W debug events can occur on write-type data storage accesses.

$DBCR0_{DAC2}$ specifies whether DAC2R debug events can occur on read-type data storage accesses and whether DAC2W debug events can occur on write-type data storage accesses.

Indexed-string instructions (***lswx***, ***stswx***) for which the XER field specifies zero bytes as the length of the string are treated as no-ops, and are not allowed to cause Data Address Compare debug events.

All *Load* instructions are considered reads with respect to debug events, while all *Store* instructions are considered writes with respect to debug events. In addition, the *Cache Management* instructions, and certain special cases, are handled as follows.

- ***dcbt***, ***dcbtls***, ***dcbtep***, ***dcbtst***, ***dcbtstls***, ***dcbtstep***, ***icbt***, ***icbtls***, ***icbtep***, ***icbi***, ***icblc***, ***dcblc***, and ***icbiep*** are all considered reads with respect to debug events. Note that ***dcbt***, ***dcbtep***, ***dcbtst***, ***dcbtstep***, ***icbt***, and ***icbtep*** are treated as no-operations when they report Data Storage or Data TLB Miss exceptions, instead of being allowed to cause interrupts. However, these instructions are allowed to cause Debug interrupts, even when they would otherwise have been no-op'ed due to a Data Storage or Data TLB Miss exception.
- ***dcbz***, ***dcbzep***, ***dcbi***, ***dcbf***, ***dcbfep***, ***dcba***, ***dcbst***, and ***dcbstep*** are all considered writes with respect to debug events. Note that ***dcbf***, ***dcbfep***, ***dcbst***, and ***dcbstep*** are considered reads with respect to Data Storage exceptions, since they do not actually change the data at a given address. However, since the execution of these instructions may result in write activity on the processor's data bus, they are treated as writes with respect to debug events.

Data Address Compare User/Supervisor Mode

$DBCR2_{DAC1US}$ specifies whether DAC1R and DAC1W debug events can occur in user mode or supervisor mode, or both.

$DBCR2_{DAC2US}$ specifies whether DAC2R and DAC2W debug events can occur in user mode or supervisor mode, or both.

Effective/Real Address Mode

$DBCR2_{DAC1ER}$ specifies whether effective addresses, real addresses, effective addresses and MSR_{DS}=0, or effective addresses and MSR_{DS}=1 are used to in determining an address match on DAC1R and DAC1W debug events.

$DBCR2_{DAC2ER}$ specifies whether effective addresses, real addresses, effective addresses and MSR_{DS}=0, or effective addresses and MSR_{DS}=1 are used to in determining an address match on DAC2R and DAC2W debug events.

Data Address Compare Mode

$DBCR2_{DAC12M}$ specifies whether all or some of the bits of the address of the data storage access must match the contents of the DAC1 or DAC2, whether the address must be inside a specific range specified by the DAC1 and DAC2 or outside a specific range specified by the DAC1 and DAC2 for a DAC1R, DAC1W, DAC2R or DAC2W debug event to occur.

There are four data address compare modes.

- *Exact address compare mode*
  If the 64-bit address of the data storage access is equal to the value in the enabled Data Address Compare Register, a data address match occurs.

  For 64-bit implementations, the addresses are masked to compare only bits 32:63 when the processor is executing in 32-bit mode.
- *Address bit match mode*
  If the address of the data storage access, ANDed with the contents of the DAC2, are equal to the contents of the DAC1, also ANDed with the contents of the DAC2, a data

address match occurs.

For 64-bit implementations, the addresses are masked to compare only bits 32:63 when the processor is executing in 32-bit mode.

- *Inclusive address range compare mode*
  If the 64-bit address of the data storage access is greater than or equal to the contents of the DAC1 and less than the contents of the DAC2, a data address match occurs.

  For 64-bit implementations, the addresses are masked to compare only bits 32:63 when the processor is executing in 32-bit mode.
- *Exclusive address range compare mode*
  If the 64-bit address of the data storage access is less than the contents of the DAC1 or greater than or equal to the contents of the DAC2, a data address match occurs.

  For 64-bit implementations, the addresses are masked to compare only bits 32:63 when the processor is executing in 32-bit mode.

Data Value Compare Mode

$DBCR2_{DVC1M}$ and $DBCR2_{DVC1BE}$ specify whether and how the data value being accessed by the storage access must match the contents of the DVC1 for a DAC1R or DAC1W debug event to occur.

$DBCR2_{DVC2M}$ and $DBCR2_{DVC2BE}$ specify whether and how the data value being accessed by the storage access must match the contents of the DVC2 for a DAC2R or DAC2W debug event to occur.

The description of DBCR0 (see Section 8.5.1.1) and DBCR2 (see Section 8.5.1.3) and the modes for detecting Data Address Compare debug events. Data Address Compare debug events can occur regardless of the setting of MSR_{DE} or $DBCR0_{IDM}$.

When an Data Address Compare debug event occurs, the corresponding $DBSR_{DAC1R}$, $DBSR_{DAC1W}$, $DBSR_{DAC2R}$, or $DBSR_{DAC2W}$ bit or bits are set to 1 to record the debug exception. If MSR_{DE}=0, $DBSR_{IDE}$ is also set to 1 to record the imprecise debug event.

If MSR_{DE}=1 (i.e. Debug interrupts are enabled) at the time of the Data Address Compare debug exception, a Debug interrupt will occur immediately (provided there exists no higher priority exception which is enabled to cause an interrupt), the execution of the instruction causing the exception will be suppressed, and CSRR0/DSRR0 [Category: Embedded.Enhanced Debug will be set to the address of the excepting instruction. Depending on the type of instruction and/or the alignment of the data access, the instruction causing the exception may have been partially executed (see Section 5.7).

If MSR_{DE}=0 (i.e. Debug interrupts are disabled) at the time of the Data Address Compare debug exception, a Debug interrupt will not occur, and the instruction will complete execution (provided the instruction is not causing some other exception which will generate an enabled interrupt). Also, $DBSR_{IDE}$ is set to indicate that the debug exception occurred while Debug interrupts were disabled by MSR_{DE}=0.

Later, if the debug exception has not been reset by clearing $DBSR_{DAC1R}$, $DBSR_{DAC1W}$, $DBSR_{DAC2R}$, $DBSR_{DAC2W}$, and MSR_{DE} is set to 1, a delayed Debug interrupt will occur. In this case, CSRR0/DSRR0 [Category: Embedded.Enhanced Debug will contain the address of the instruction after the one which enabled the Debug interrupt by setting MSR_{DE} to 1. Software in the Debug interrupt handler can observe $DBSR_{IDE}$ to determine how to interpret the value in CSRR0/DSRR0 [Category: Embedded.Enhanced Debug.

8.4.3 Trap Debug Event

A Trap debug event (TRAP) occurs if $DBCR0_{TRAP}$=1 (i.e. Trap debug events are enabled) and a ***Trap*** instruction (***tw***, ***twi***, ***td***, ***tdi***) is executed and the conditions specified by the instruction for the trap are met. The event can occur regardless of the setting of MSR_{DE} or $DBCR0_{IDM}$.

When a Trap debug event occurs, $DBSR_{TR}$ is set to 1 to record the debug exception. If MSR_{DE}=0, $DBSR_{IDE}$ is also set to 1 to record the imprecise debug event.

If MSR_{DE}=1 (i.e. Debug interrupts are enabled) at the time of the Trap debug exception, a Debug interrupt will occur immediately (provided there exists no higher priority exception which is enabled to cause an interrupt), and CSRR0/DSRR0 [Category: Embedded.Enhanced Debug] will be set to the address of the excepting instruction.

If MSR_{DE}=0 (i.e. Debug interrupts are disabled) at the time of the Trap debug exception, a Debug interrupt will not occur, and a Trap exception type Program interrupt will occur instead if the trap condition is met.

Later, if the debug exception has not been reset by clearing $DBSR_{TR}$, and MSR_{DE} is set to 1, a delayed Debug interrupt will occur. In this case, CSRR0/DSRR0 [Category: Embedded.Enhanced Debug will contain the address of the instruction after the one which enabled the Debug interrupt by setting MSR_{DE} to 1. Software in the debug interrupt handler can observe $DBSR_{IDE}$ to determine how to interpret the value in CSRR0/DSRR0 [Category: Embedded.Enhanced Debug].

8.4.4 Branch Taken Debug Event

A Branch Taken debug event (BRT) occurs if $DBCR0_{BRT}$=1 (i.e. Branch Taken Debug events are enabled), execution is attempted of a branch instruction whose direction will be taken (that is, either an unconditional branch, or a conditional branch whose branch condition is met), and MSR_{DE}=1.

Branch Taken debug events are not recognized if MSR_{DE}=0 at the time of the execution of the branch instruction and thus $DBSR_{IDE}$ can not be set by a Branch Taken debug event. This is because branch instructions occur very frequently. Allowing these common events to be recorded as exceptions in the DBSR while debug interrupts are disabled via MSR_{DE} would result in an inordinate number of imprecise Debug interrupts.

When a Branch Taken debug event occurs, the $DBSR_{BRT}$ bit is set to 1 to record the debug exception and a Debug interrupt will occur immediately (provided there exists no higher priority exception which is enabled to cause an interrupt). The execution of the instruction causing the exception will be suppressed, and CSRR0/DSRR0 [Category: Embedded.Enhanced Debug] will be set to the address of the excepting instruction.

8.4.5 Instruction Complete Debug Event

An Instruction Complete debug event (ICMP) occurs if $DBCR0_{ICMP}$=1 (i.e. Instruction Complete debug events are enabled), execution of any instruction is completed, and MSR_{DE}=1. Note that if execution of an instruction is suppressed due to the instruction causing some other exception which is enabled to generate an interrupt, then the attempted execution of that instruction does not cause an Instruction Complete debug event. The ***sc*** instruction does not fall into the type of an instruction whose execution is suppressed, since the instruction actually completes execution and then generates a System Call interrupt. In this case, the Instruction Complete debug exception will also be set.

Instruction Complete debug events are not recognized if MSR_{DE}=0 at the time of the execution of the instruction, $DBSR_{IDE}$ can not be set by an ICMP debug event. This is because allowing the common event of Instruction Completion to be recorded as an exception in the DBSR while Debug interrupts are disabled via MSR_{DE} would mean that the Debug interrupt handler software would receive an inordinate number of imprecise Debug interrupts every time Debug interrupts were re-enabled via MSR_{DE}.

When an Instruction Complete debug event occurs, $DBSR_{ICMP}$ is set to 1 to record the debug exception, a Debug interrupt will occur immediately (provided there exists no higher priority exception which is enabled to cause an interrupt), and CSRR0/DSRR0 [Category: Embedded.Enhanced Debug] will be set to the address of the instruction after the one causing the Instruction Complete debug exception.

8.4.6 Interrupt Taken Debug Event

8.4.6.1 Causes of Interrupt Taken Debug Events

Only base class interrupts can cause an Interrupt Taken debug event. If the Embedded.Enhanced Debug category is not supported or is supported and not enabled, all other interrupts automatically clear MSR_{DE}, and thus would always prevent the associated Debug interrupt from occurring precisely. If the Embedded.Enhanced Debug category is supported and enabled, then critical class interrupts do not automatically clear MSR_{DE}, but they cause Critical Interrupt Taken debug events instead of Interrupt Taken debug events.

Also, if the Embedded.Enhanced Debug category is not supported or is supported and not enabled, Debug interrupts themselves are critical class interrupts, and thus any Debug interrupt (for any other debug event) would always end up setting the additional exception of $DBSR_{IRPT}$ upon entry to the Debug interrupt handler. At this point, the Debug interrupt handler would be unable to determine whether or not the Interrupt Taken debug event was related to the original debug event.

8.4.6.2 Interrupt Taken Debug Event Description

An Interrupt Taken debug event (IRPT) occurs if $DBCR0_{IRPT}$=1 (i.e. Interrupt Taken debug events are enabled) and a base class interrupt occurs. Interrupt Taken debug events can occur regardless of the setting of MSR_{DE}.

When an Interrupt Taken debug event occurs, $DBSR_{IRPT}$ is set to 1 to record the debug exception. If MSR_{DE}=0, $DBSR_{IDE}$ is also set to 1 to record the imprecise debug event.

If MSR_{DE}=1 (i.e. Debug interrupts are enabled) at the time of the Interrupt Taken debug event, a Debug interrupt will occur immediately (provided there exists no higher priority exception which is enabled to cause an interrupt), and Critical Save/Restore Register 0/Debug Save/Restore Register 0 [Category: Embedded.Enhanced Debug] will be set to the address of the interrupt vector which caused the Interrupt Taken debug event. No instructions at the base interrupt handler will have been executed.

If MSR_{DE}=0 (i.e. Debug interrupts are disabled) at the time of the Interrupt Taken debug event, a Debug interrupt will not occur, and the handler for the interrupt which caused the Interrupt Taken debug event will be allowed to execute.

Later, if the debug exception has not been reset by clearing $DBSR_{IRPT}$, and MSR_{DE} is set to 1, a delayed Debug interrupt will occur. In this case, CSRR0/DSRR0

[Category: Embedded.Enhanced Debug] will contain the address of the instruction after the one which enabled the Debug interrupt by setting MSR_{DE} to 1. Software in the Debug interrupt handler can observe the $DBSR_{IDE}$ bit to determine how to interpret the value in CSRR0/DSRR0 [Category: Embedded.Enhanced Debug.

8.4.7 Return Debug Event

A Return debug event (RET) occurs if $DBCR0_{RET}$=1 and an attempt is made to execute an ***rfi***. Return debug events can occur regardless of the setting of MSR_{DE}.

When a Return debug event occurs, $DBSR_{RET}$ is set to 1 to record the debug exception. If MSR_{DE}=0, $DBSR_{IDE}$ is also set to 1 to record the imprecise debug event.

If MSR_{DE}=1 at the time of the Return Debug event, a Debug interrupt will occur immediately, and CSRR0/DSRR0 [Category: Embedded.Enhanced Debug will be set to the address of the ***rfi***.

If MSR_{DE}=0 at the time of the Return Debug event, a Debug interrupt will not occur.

Later, if the Debug exception has not been reset by clearing $DBSR_{RET}$, and MSR_{DE} is set to 1, a delayed imprecise Debug interrupt will occur. In this case, CSRR0/DSRR0 [Category: Embedded.Enhanced Debug will contain the address of the instruction after the one which enabled the Debug interrupt by setting MSR_{DE} to 1. An imprecise Debug interrupt can be caused by executing an ***rfi*** when $DBCR0_{RET}$=1 and MSR_{DE}=0, and the execution of that ***rfi*** happens to cause MSR_{DE} to be set to 1. Software in the Debug interrupt handler can observe the $DBSR_{IDE}$ bit to determine how to interpret the value in CSRR0/DSRR0 [Category: Embedded.Enhanced Debug].

8.4.8 Unconditional Debug Event

An Unconditional debug event (UDE) occurs when the Unconditional Debug Event (UDE) signal is activated by the debug mechanism. The exact definition of the UDE signal and how it is activated is implementation-dependent. The Unconditional debug event is the only debug event which does not have a corresponding enable bit for the event in DBCR0 (hence the name of the event). The Unconditional debug event can occur regardless of the setting of MSR_{DE}.

When an Unconditional debug event occurs, the $DBSR_{UDE}$ bit is set to 1 to record the Debug exception. If MSR_{DE}=0, $DBSR_{IDE}$ is also set to 1 to record the imprecise debug event.

If MSR_{DE}=1 (i.e. Debug interrupts are enabled) at the time of the Unconditional Debug exception, a Debug interrupt will occur immediately (provided there exists no higher priority exception which is enabled to cause an interrupt), and CSRR0/DSRR0 [Category: Embedded.Enhanced Debug] will be set to the address of the instruction which would have executed next had the interrupt not occurred.

If MSR_{DE}=0 (i.e. Debug interrupts are disabled) at the time of the Unconditional Debug exception, a Debug interrupt will not occur.

Later, if the Unconditional Debug exception has not been reset by clearing $DBSR_{UDE}$, and MSR_{DE} is set to 1, a delayed Debug interrupt will occur. In this case, CSRR0/DSRR0 [Category: Embedded.Enhanced Debug] will contain the address of the instruction after the one which enabled the Debug interrupt by setting MSR_{DE} to 1. Software in the Debug interrupt handler can observe $DBSR_{IDE}$ to determine how to interpret the value in CSRR0/DSRR0 [Category: Embedded.Enhanced Debug].

8.4.9 Critical Interrupt Taken Debug Event [Category: Embedded.Enhanced Debug]

A Critical Interrupt Taken debug event (CIRPT) occurs if $DBCR0_{CIRPT}$ = 1 (i.e. Critical Interrupt Taken debug events are enabled) and a critical interrupt occurs. A critical interrupt is any interrupt that saves state in CSRR0 and CSRR1 when the interrupt is taken. Critical Interrupt Taken debug events can occur regardless of the setting of MSR_{DE}.

When a Critical Interrupt Taken debug event occurs, $DBSR_{CIRPT}$ is set to 1 to record the debug event. If MSR_{DE}=0, $DBSR_{IDE}$ is also set to 1 to record the imprecise debug event.

If MSR_{DE} = 1 (i.e. Debug Interrupts are enabled) at the time of the Critical Interrupt Taken debug event, a Debug Interrupt will occur immediately (provided there is no higher priority exception which is enabled to cause an interrupt), and DSRR0 will be set to the address of the first instruction of the critical interrupt handler. No instructions at the critical interrupt handler will have been executed.

If MSR_{DE} = 0 (i.e. Debug Interrupts are disabled) at the time of the Critical Interrupt Taken debug event, a Debug Interrupt will not occur, and the handler for the critical interrupt which caused the debug event will be allowed to execute normally. Later, if the debug exception has not been reset by clearing $DBSR_{CIRPT}$ and MSR_{DE} is set to 1, a delayed Debug Interrupt will occur. In this case DSRR0 will contain the address of the instruction after the one that set MSR_{DE} = 1. Software in the Debug Interrupt handler can observe $DBSR_{IDE}$ to determine how to interpret the value in DSRR0.

8.4.10 Critical Interrupt Return Debug Event [Category: Embedded.Enhanced Debug]

A Critical Interrupt Return debug event (CRET) occurs if $DBCR0_{CRET}$ = 1 (i.e. Critical Interrupt Return debug events are enabled) and an attempt is made to execute an ***rfci*** instruction. Critical Interrupt Return debug events can occur regardless of the setting of MSR_{DE}.

When a Critical Interrupt Return debug event occurs, $DBSR_{CRET}$ is set to 1 to record the debug event. If MSR_{DE}=0, $DBSR_{IDE}$ is also set to 1 to record the imprecise debug event.

If MSR_{DE} = 1 (i.e. Debug Interrupts are enabled) at the time of the Critical Interrupt Return debug event, a Debug Interrupt will occur immediately (provided there is no higher priority exception which is enabled to cause an interrupt), and DSRR0 will be set to the address of the ***rfci*** instruction.

If MSR_{DE} = 0 (i.e. Debug Interrupts are disabled) at the time of the Critical Interrupt Return debug event, a Debug Interrupt will not occur. Later, if the debug exception has not been reset by clearing $DBSR_{CRET}$ and MSR_{DE} is set to 1, a delayed Debug Interrupt will occur. In this case DSRR0 will contain the address of the instruction after the one that set MSR_{DE} = 1. An imprecise Debug Interrupt can be caused by executing an ***rfci*** when $DBCR0_{CRET}$ = 1 and MSR_{DE} = 0, and the execution of the ***rfci*** happens to cause MSR_{DE} to be set to 1. Software in the Debug Interrupt handler can observe $DBSR_{IDE}$ to determine how to interpret the value in DSRR0.

8.5 Debug Registers

This section describes debug-related registers that are accessible to software running on the processor. These registers are intended for use by special debug tools and debug software, and not by general application or operating system code.

8.5.1 Debug Control Registers

Debug Control Register 0 (DBCR0), Debug Control Register 1 (DBCR1), and Debug Control Register 2 (DBCR2) are each 32-bit registers. Bits of DBCR0, DBCR1, and DBCR2 are numbered 32 (most-significant bit) to 63 (least-significant bit). DBCR0, DBCR1, and DBCR2 are used to enable debug events, reset the processor, control timer operation during debug events, and set the debug mode of the processor.

8.5.1.1 Debug Control Register 0 (DCBR0)

The contents of the DCBR0 can be read into bits 32:63 of register RT using *mfspr RT,DBCR0*, setting bits 0:31 of RT to 0. The contents of bits 32:63 of register RS can be written to the DCBR0 using *mtspr DBCR0,RS*. The bit definitions for DCBR0 are shown below.

Bit(s) Description

32 ***External Debug Mode*** (EDM) **[Category: Embedded.Enhanced Debug]**
The EDM bit is a read-only bit that reflects whether the processor is controlled by an external debug facility. When EDM is set, internal debug mode is suppressed and the taking of debug interrupts does not occur.

0 The processor is not in external debug mode.
1 The processor is in external debug mode.

33 ***Internal Debug Mode*** (IDM)

0 Debug interrupts are disabled.
1 If MSR_{DE}=1, then the occurrence of a debug event or the recording of an earlier debug event in the Debug Status Register when MSR_{DE}=0 or $DBCR0_{IDM}$=0 will cause a Debug interrupt.

34:35 ***Reset*** (RST)

00 No action
01 Implementation-specific
10 Implementation-specific
11 Implementation-specific

Warning: Writing 0b01, 0b10, or 0b11 to these bits may cause a processor reset to occur.

36 ***Instruction Completion Debug Event*** (ICMP)

0 ICMP debug events are disabled
1 ICMP debug events are enabled

Note: Instruction Completion will not cause an ICMP debug event if MSR_{DE}=0.

37 ***Branch Taken Debug Event Enable*** (BRT)

0 BRT debug events are disabled
1 BRT debug events are enabled

Note: Taken branches will not cause a BRT debug event if MSR_{DE}=0.

38 ***Interrupt Taken Debug Event Enable*** (IRPT)

0 IRPT debug events are disabled
1 IRPT debug events are enabled

Note: Critical interrupts will not cause an IRPT Debug event even if MSR_{DE}=0. If the

Embedded.Enhanced Debug category is supported, see Section 8.4.9.

39 ***Trap Debug Event Enable*** (TRAP)

0 TRAP debug events cannot occur
1 TRAP debug events can occur

40 ***Instruction Address Compare 1 Debug Event Enable*** (IAC1)

0 IAC1 debug events cannot occur
1 IAC1 debug events can occur

41 ***Instruction Address Compare 2 Debug Event Enable*** (IAC2)

0 IAC2 debug events cannot occur
1 IAC2 debug events can occur

42 ***Instruction Address Compare 3 Debug Event Enable*** (IAC3)

0 IAC3 debug events cannot occur
1 IAC3 debug events can occur

43 ***Instruction Address Compare 4 Debug Event Enable*** (IAC4)

0 IAC4 debug events cannot occur
1 IAC4 debug events can occur

44:45 ***Data Address Compare 1 Debug Event Enable*** (DAC1)

00 DAC1 debug events cannot occur
01 DAC1 debug events can occur only if a store-type data storage access
10 DAC1 debug events can occur only if a load-type data storage access
11 DAC1 debug events can occur on any data storage access

46:47 ***Data Address Compare 2 Debug Event Enable*** (DAC2)

00 DAC2 debug events cannot occur
01 DAC2 debug events can occur only if a store-type data storage access
10 DAC2 debug events can occur only if a load-type data storage access
11 DAC2 debug events can occur on any data storage access

48 ***Return Debug Event Enable*** (RET)

0 RET debug events cannot occur
1 RET debug events can occur

Note: Return From Critical Interrupt will not cause an RET debug event if MSR_{DE}=0. If the Embedded.Enhanced Debug category is supported, see Section 8.4.10

49:56 Reserved

57 ***Critical Interrupt Taken Debug Event*** (CIRPT) **[Category: Embedded.Enhanced Debug]**
A Critical Interrupt Taken Debug Event occurs when $DBCR0_{CIRPT}$ = 1 and a critical interrupt (any interrupt that uses the critical class, i.e. uses CSRR0 and CSRR1) occurs.

0 Critical interrupt taken debug events are disabled.
1 Critical interrupt taken debug events are enabled.

58 ***Critical Interrupt Return Debug Event*** (CRET) **[Category: Embedded.Enhanced Debug]**
A Critical Interrupt Return Debug Event occurs when $DBCR0_{CRET}$= 1 and a return from critical interrupt (an ***rfci*** instruction is executed) occurs.

0 Critical interrupt return debug events are disabled.
1 Critical interrupt return debug events are enabled.

59:62 Implementation-dependent

63 ***Freeze Timers on Debug Event*** (FT)

0 Enable clocking of timers
1 Disable clocking of timers if any DBSR bit is set (except MRR)

8.5.1.2 Debug Control Register 1 (DCBR1)

The contents of the DCBR1 can be read into bits 32:63 a register RT using ***mfspr*** *RT,DBCR1*, setting bits 0:31 of RT to 0. The contents of bits 32:63 of register RS can be written to the DBCR1 using ***mtspr*** *DBCR1,RS*. The bit definitions for DCBR1 are shown below.

Bit(s) Description

32:33 ***Instruction Address Compare 1 User/ Supervisor Mode***(IAC1US)

00 IAC1 debug events can occur
01 Reserved
10 IAC1 debug events can occur only if MSR_{PR}=0
11 IAC1 debug events can occur only if MSR_{PR}=1

34:35 ***Instruction Address Compare 1 Effective/ Real Mode*** (IAC1ER)

00 IAC1 debug events are based on effective addresses
01 IAC1 debug events are based on real addresses
10 IAC1 debug events are based on effective addresses and can occur only if MSR_{IS}=0
11 IAC1 debug events are based on effective addresses and can occur only if MSR_{IS}=1

36:37 ***Instruction Address Compare 2 User/ Supervisor Mode*** (IAC2US)

00 IAC2 debug events can occur
01 Reserved
10 IAC2 debug events can occur only if MSR_{PR}=0
11 IAC2 debug events can occur only if MSR_{PR}=1

38:39 ***Instruction Address Compare 2 Effective/ Real Mode*** (IAC2ER)

00 IAC2 debug events are based on effective addresses
01 IAC2 debug events are based on real addresses
10 IAC2 debug events are based on effective addresses and can occur only if MSR_{IS}=0
11 IAC2 debug events are based on effective addresses and can occur only if MSR_{IS}=1

40:41 ***Instruction Address Compare 1/2 Mode*** (IAC12M)

00 Exact address compare

IAC1 debug events can occur only if the address of the instruction fetch is equal to the value specified in IAC1.

IAC2 debug events can occur only if the address of the instruction fetch is equal to the value specified in IAC2.

01 Address bit match

IAC1 and IAC2 debug events can occur only if the address of the instruction fetch, ANDed with the contents of IAC2 are equal to the contents of IAC1, also ANDed with the contents of IAC2.

If IAC1US≠IAC2US or IAC1ER≠IAC2ER, results are boundedly undefined.

10 Inclusive address range compare

IAC1 and IAC2 debug events can occur only if the address of the instruction fetch is greater than or equal to the value specified in IAC1 and less than the value specified in IAC2.

If IAC1US≠IAC2US or IAC1ER≠IAC2ER, results are boundedly undefined.

11 Exclusive address range compare

IAC1 and IAC2 debug events can occur only if the address of the instruction fetch is less than the value specified in IAC1 or is greater than or equal to the value specified in IAC2.

If IAC1US≠AC2US or IAC1ER≠IAC2ER, results are boundedly undefined.

42:47 Reserved

48:49 ***Instruction Address Compare 3 User/ Supervisor Mode*** (IAC3US)

00 IAC3 debug events can occur
01 Reserved
10 IAC3 debug events can occur only if MSR_{PR}=0
11 IAC3 debug events can occur only if MSR_{PR}=1

50:51 ***Instruction Address Compare 3 Effective/ Real Mode*** (IAC3ER)

00 IAC3 debug events are based on effective addresses
01 IAC3 debug events are based on real addresses
10 IAC3 debug events are based on effective addresses and can occur only if MSR_{IS}=0
11 IAC3 debug events are based on effective addresses and can occur only if MSR_{IS}=1

52:53 ***Instruction Address Compare 4 User/ Supervisor Mode*** (IAC4US)

00 IAC4 debug events can occur
01 Reserved
10 IAC4 debug events can occur only if MSR_{PR}=0
11 IAC4 debug events can occur only if MSR_{PR}=1

54:55 ***Instruction Address Compare 4 Effective/ Real Mode*** (IAC4ER)

00 IAC4 debug events are based on effective addresses
01 IAC4 debug events are based on real addresses
10 IAC4 debug events are based on effective addresses and can occur only if MSR_{IS}=0
11 IAC4 debug events are based on effective addresses and can occur only if MSR_{IS}=1

56:57 ***Instruction Address Compare 3/4 Mode*** (IAC34M)

00 Exact address compare

IAC3 debug events can occur only if the address of the instruction fetch is equal to the value specified in IAC3.

IAC4 debug events can occur only if the address of the instruction fetch is equal to the value specified in IAC4.

01 Address bit match

IAC3 and IAC4 debug events can occur only if the address of the data storage access, ANDed with the contents of IAC4 are equal to the contents of IAC3, also ANDed with the contents of IAC4.

If IAC3US≠IAC4US or IAC3ER≠IAC4ER, results are boundedly undefined.

10 Inclusive address range compare

IAC3 and IAC4 debug events can occur only if the address of the instruction fetch is greater than or equal to the value specified in IAC3 and less than the value specified in IAC4.

If IAC3US≠IAC4US or IAC3ER≠IAC4ER, results are boundedly undefined.

11 Exclusive address range compare

IAC3 and IAC4 debug events can occur only if the address of the instruction fetch is less than the value specified in IAC3 or is greater than or equal to the value specified in IAC4.

If IAC3US≠IAC4US or IAC3ER≠IAC4ER, results are boundedly undefined.

58:63 Reserved

8.5.1.3 Debug Control Register 2 (DCBR2)

The contents of the DCBR2 can be copied into bits 32:63 register RT using ***mfspr*** *RT,DBCR2*, setting bits 0:31 of register RT to 0. The contents of bits 32:63 of a register RS can be written to the DCBR2 using ***mtspr*** *DBCR2,RS*. The bit definitions for DCBR2 are shown below.

Bit(s) Description

32:33 ***Data Address Compare 1 User/Supervisor Mode*** (DAC1US)

00 DAC1 debug events can occur
01 Reserved
10 DAC1 debug events can occur only if MSR_{PR}=0
11 DAC1 debug events can occur only if MSR_{PR}=1

34:35 ***Data Address Compare 1 Effective/Real Mode*** (DAC1ER)

00 DAC1 debug events are based on effective addresses
01 DAC1 debug events are based on real addresses
10 DAC1 debug events are based on effective addresses and can occur only if MSR_{DS}=0
11 DAC1 debug events are based on effective addresses and can occur only if MSR_{DS}=1

36:37 ***Data Address Compare 2 User/Supervisor Mode*** (DAC2US)

00 DAC2 debug events can occur
01 Reserved
10 DAC2 debug events can occur only if MSRPR=0
11 DAC2 debug events can occur only if MSRPR=1

38:39 ***Data Address Compare 2 Effective/Real Mode*** (DAC2ER)

00 DAC2 debug events are based on effective addresses
01 DAC2 debug events are based on real addresses
10 DAC2 debug events are based on effective addresses and can occur only if MSR_{DS}=0
11 DAC2 debug events are based on effective addresses and can occur only if MSR_{DS}=1

40:41 ***Data Address Compare 1/2 Mode*** (DAC12M)

00 Exact address compare

DAC1 debug events can occur only if the address of the data storage access is equal to the value specified in DAC1.

DAC2 debug events can occur only if the address of the data storage access is equal to the value specified in DAC2.

01 Address bit match

DAC1 and DAC2 debug events can occur only if the address of the data storage access, ANDed with the contents of DAC2 are equal to the contents of DAC1, also ANDed with the contents of DAC2.

If DAC1US≠DAC2US or DAC1ER≠DAC2ER, results are boundedly undefined.

10 Inclusive address range compare

DAC1 and DAC2 debug events can occur only if the address of the data storage access is greater than or equal to the value specified in DAC1 and less than the value specified in DAC2.

If DAC1US ≠ DAC2US or DAC1ER ≠ DAC2ER, results are boundedly undefined.

11 Exclusive address range compare

DAC1 and DAC2 debug events can occur only if the address of the data storage access is less than the value specified in DAC1 or is greater than or equal to the value specified in DAC2.

If DAC1US ≠ DAC2US or DAC1ER ≠ DAC2ER, results are boundedly undefined.

42:43 Reserved

44:45 ***Data Value Compare 1 Mode*** (DVC1M)

00 DAC1 debug events can occur
01 DAC1 debug events can occur only when all bytes specified in $DBCR2_{DVC1BE}$ in the data value of the data storage access match their corresponding bytes in DVC1
10 DAC1 debug events can occur only when at least one of the bytes specified in $DBCR2_{DVC1BE}$ in the data value of the data storage access matches its corresponding byte in DVC1
11 DAC1 debug events can occur only when all bytes specified in $DBCR2_{DVC1BE}$ within at least one of the halfwords of the data value of the data storage access matches their corresponding bytes in DVC1

46:47 ***Data Value Compare 2 Mode*** (DVC2M)

00 DAC2 debug events can occur
01 DAC2 debug events can occur only when all bytes specified in $DBCR2_{DVC2BE}$ in the data value of the data storage access match their corresponding bytes in DVC2
10 DAC2 debug events can occur only when at least one of the bytes specified in $DBCR2_{DVC2BE}$ in the data value of the data storage access matches its corresponding byte in DVC2
11 DAC2 debug events can occur only when all bytes specified in $DBCR2_{DVC2BE}$ within at least one of the halfwords of the data value of the data storage access matches their corresponding bytes in DVC2

48:55 ***Data Value Compare 1 Byte Enables*** (DVC1BE)

Specifies which bytes in the aligned data value being read or written by the storage access are compared to the corresponding bytes in DVC1.

56:63 ***Data Value Compare 2 Byte Enables*** (DVC2BE)

Specifies which bytes in the aligned data value being read or written by the storage access are compared to the corresponding bytes in DVC2

8.5.2 Debug Status Register

The Debug Status Register (DBSR) is a 32-bit register and contains status on debug events and the most recent processor reset.

The DBSR is set via hardware, and read and cleared via software. The contents of the DBSR can be read into bits 32:63 of a register RT using the ***mfspr*** instruction, setting bits 0:31 of RT to zero. Bits in the DBSR can be cleared using the ***mtspr*** instruction. Clearing is done by writing bits 32:63 of a register to the DBSR with a 1 in any bit position that is to be cleared and 0 in all other bit positions. The write-data to the DBSR is not direct data, but a mask. A 1 causes the bit to be cleared, and a 0 has no effect.

The bit definitions for the DBSR are shown below:

Bit(s) Description

32 ***Imprecise Debug Event*** (IDE)

Set to 1 if MSR_{DE}=0 and a debug event causes its respective Debug Status Register bit to be set to 1.

33 ***Unconditional Debug Event*** (UDE)

Set to 1 if an Unconditional debug event occurred. See Section 8.4.8.

34:35 ***Most Recent Reset*** (MRR)

Set to one of three values when a reset occurs. These two bits are undefined at power-up.

00 No reset occurred since these bits last cleared by software
01 Implementation-dependent reset information
10 Implementation-dependent reset information
11 Implementation-dependent reset information

36 ***Instruction Complete Debug Event*** (ICMP)

Set to 1 if an Instruction Completion debug event occurred and $DBCR0_{ICMP}$=1. See Section 8.4.5.

37 ***Branch Taken Debug Event*** (BRT)

Set to 1 if a Branch Taken debug event occurred and $DBCR0_{BRT}$=1. See Section 8.4.4.

38 ***Interrupt Taken Debug Event*** (IRPT)

Set to 1 if an Interrupt Taken debug event occurred and $DBCR0_{IRPT}$=1. See Section 8.4.6.

39 ***Trap Instruction Debug Event*** (TRAP)

Set to 1 if a Trap Instruction debug event occurred and $DBCR0_{TRAP}$=1. See Section 8.4.3.

40 ***Instruction Address Compare 1 Debug Event*** (IAC1)

Set to 1 if an IAC1 debug event occurred and $DBCR0_{IAC1}$=1. See Section 8.4.1.

41 ***Instruction Address Compare 2 Debug Event*** (IAC2)

Set to 1 if an IAC2 debug event occurred and $DBCR0_{IAC2}$=1. See Section 8.4.1.

42 ***Instruction Address Compare 3 Debug Event*** (IAC3)

Set to 1 if an IAC3 debug event occurred and $DBCR0_{IAC3}$=1. See Section 8.4.1.

43 ***Instruction Address Compare 4 Debug Event*** (IAC4)

Set to 1 if an IAC4 debug event occurred and $DBCR0_{IAC4}$=1. See Section 8.4.1.

44 ***Data Address Compare 1 Read Debug Event*** (DAC1R)

Set to 1 if a read-type DAC1 debug event occurred and $DBCR0_{DAC1}$=0b10 or $DBCR0_{DAC1}$=0b11. See Section 8.4.2.

45 ***Data Address Compare 1 Write Debug Event*** (DAC1W)

Set to 1 if a write-type DAC1 debug event occurred and $DBCR0_{DAC1}$=0b01 or $DBCR0_{DAC1}$=0b11. See Section 8.4.2.

46 ***Data Address Compare 2 Read Debug Event*** (DAC2R)

Set to 1 if a read-type DAC2 debug event occurred and $DBCR0_{DAC2}$=0b10 or $DBCR0_{DAC2}$=0b11. See Section 8.4.2.

47 ***Data Address Compare 2 Write Debug Event*** (DAC2W)

Set to 1 if a write-type DAC2 debug event occurred and $DBCR0_{DAC2}$=0b01 or $DBCR0_{DAC2}$=0b11. See Section 8.4.2.

48 ***Return Debug Event*** (RET)

Set to 1 if a Return debug event occurred and $DBCR0_{RET}$=1. See Section 8.4.2.

49:52 Reserved

53:56 Implementation-dependent

57 ***Critical Interrupt Taken Debug Event*** (CIRPT) **[Category: Embedded.Enhanced Debug]**

A Critical Interrupt Taken Debug Event occurs when $DBCR0_{CIRPT}$=1 and a critical interrupt (any interrupt that uses the critical class, i.e. uses CSRR0 and CSRR1) occurs.

0 Critical interrupt taken debug events are disabled.

1 Critical interrupt taken debug events are enabled.

58 ***Critical Interrupt Return Debug Event*** (CRET) **[Category: Embedded.Enhanced Debug]**

A Critical Interrupt Return Debug Event occurs when $DBCR0_{CRET}$=1 and a return from critical interrupt (an ***rfci*** instruction is executed) occurs.

0 Critical interrupt return debug events are disabled.

1 Critical interrupt return debug events are enabled.

59:63 Implementation-dependent

8.5.3 Instruction Address Compare Registers

The Instruction Address Compare Register 1, 2, 3, and 4 (IAC1, IAC2, IAC3, and IAC4 respectively) are each 64-bits, with bit 63 being reserved.

A debug event may be enabled to occur upon an attempt to execute an instruction from an address specified in either IAC1, IAC2, IAC3, or IAC4, inside or outside a range specified by IAC1 and IAC2 or, inside or outside a range specified by IAC3 and IAC4, or to blocks of addresses specified by the combination of the IAC1 and IAC2, or to blocks of addresses specified by the combination of the IAC3 and IAC4. Since all instruction addresses are required to be word-aligned, the two low-order bits of the Instruction Address Compare Registers are reserved and do not participate in the comparison to the instruction address (see Section 8.4.1 on page 607).

The contents of the Instruction Address Compare *i* Register (where *i*={1,2,3, or 4}) can be read into register RT using *mfspr RT,IACi*. The contents of register RS can be written to the Instruction Address Compare *i* Register using *mtspr IACi,RS*.

8.5.4 Data Address Compare Registers

The Data Address Compare Register 1 and 2 (DAC1 and DAC2 respectively) are each 64-bits.

A debug event may be enabled to occur upon loads, stores, or cache operations to an address specified in either the DAC1 or DAC2, inside or outside a range specified by the DAC1 and DAC2, or to blocks of addresses specified by the combination of the DAC1 and DAC1 (see Section 8.4.2).

The contents of the Data Address Compare *i* Register (where *i*={1 or 2}) can be read into register RT using *mfspr RT,DACi*. The contents of register RS can be written to the Data Address Compare *i* Register using *mtspr DACi,RS*.

The contents of the DAC1 or DAC2 are compared to the address generated by a data storage access instruction.

8.5.5 Data Value Compare Registers

The Data Value Compare Register 1 and 2 (DVC1 and DVC2 respectively) are each 64-bits.

A DAC1R, DAC1W, DAC2R, or DAC2W debug event may be enabled to occur upon loads or stores of a specific data value specified in either or both of the DVC1 and DVC2. $DBCR2_{DVC1M}$ and $DBCR2_{DVC1BE}$ control how the contents of the DVC1 is compared with the value and $DBCR2_{DVC2M}$ and $DBCR2_{DVC2BE}$ control how the contents of the DVC2 is compared with the value (see Section 8.4.2 and Section 8.5.1.3).

The contents of the Data Value Compare *i* Register (where *i*={1 or 2}) can be read into register RT using *mfspr RT,DVCi.* The contents of register RS can be written to the Data Value Compare *i* Register using *mtspr DVCi,RS*.

8.6 Debugger Notify Halt Instruction [Category: Embedded.Enhanced Debug]

The ***dnh*** instruction provides the means for the transfer of information between the processor and an implementation-dependent external debug facility. ***dnh*** also causes the processor to stop fetching and executing instructions.

Debugger Notify Halt *XFX-form*

dnh DUI,DUIS

19	DUI	DUIS	198	/
0	6	11	21	31

```
if enabled by implementation-dependent means
then
   implementation-dependent register ← dui
   halt processor
else
   illegal instruction exception
```

Execution of the ***dnh*** instruction causes the processor to stop fetching instructions and taking interrupts if execution of the instruction has been enabled. The contents of the DUI field are sent to the external debug facility to identify the reason for the halt.

If execution of the ***dnh*** instruction has not been previously enabled, executing the ***dnh*** instruction produces an Illegal Instruction exception. The means by which execution of the ***dnh*** instruction is enabled is implementation-dependent.

The current state of the processor debug facility, whether the processor is in IDM or EDM mode has no effect on the execution of the ***dnh*** instruction.

The instruction is context synchronizing.

Programming Note

The DUIS field in the instruction may be used to pass information to an external debug facility. After the ***dnh*** instruction has executed, the instruction itself can be read back by the Illegal Instruction Interrupt handler or the external debug facility if the contents of the DUIS field are of interest. If the processor entered the Illegal Instruction Interrupt handler, software can use SRR0 to obtain the address of the ***dnh*** instruction which caused the handler to be invoked. If the ***dnh*** instruction has been executed and the processor has stopped fetching instructions, the external debug facility can issue a ***mfspr*** NIA to obtain the address of the ***dnh*** instruction that was executed.

Special Registers Altered:

None

Chapter 9. Processor Control [Category: Embedded.Processor Control]

9.1 Overview

The Processor Control facility provides a mechanism for processors within a coherence domain to send messages to all devices in the coherence domain. The facility provides a mechanism for sending interrupts that are not dependent on the interrupt controller to processors and allows message filtering by the processors that receive the message.

The Processor Control facility is also useful for sending messages to a device that provides specialized services such as secure boot operations controlled by a security device.

The Processor Control facility defines how processors send messages and what actions processors take on the receipt of a message. The actions taken by devices other than processors are not defined.

9.2 Programming Model

Processors initiate a message by executing the ***msgsnd*** instruction and specifying a message type and message payload in a general purpose register. Sending a message causes the message to be sent to all the devices, including the sending processor, in the coherence domain in a reliable manner.

Each device receives all messages that are sent. The actions that a device takes are dependent on the message type and payload. There are no restrictions on what messages a processor can send.

To provide inter processor interrupt capability two message types are defined, *Processor Doorbell* and *Processor Doorbell Critical*. A *Processor Doorbell* [*Critical*] message causes an interrupt to occur on processors when the message is received and the processor determines through examination of the payload that the message should be accepted. The examination of the payload for this purpose is termed *filtering*. The acceptance of a *Processor Doorbell* [*Critical*] message causes an exception to be generated on the accepting processor.

Processors accept and filter messages defined in Section 9.2.1. Processors may also accept other implementation-dependent defined messages.

9.2.1 Processor Message Handling and Filtering

Processors filter, accept, and handle message types defined as follows. The message type is specified in the message and is determined by the contents of register $RB_{32:36}$ used as the operand in the ***msgsnd*** instruction.The message type is interpreted as follows:

Value	Description
0	***Doorbell Interrupt*** (DBELL) A Processor Doorbell exception is generated on the processor when the processor has filtered the message based on the payload and has determined that it should accept the message. A Processor Doorbell Interrupt occurs when no higher priority exception exists, a Processor Doorbell exception exists, and MSR_{EE}=1.
1	***Doorbell Critical Interrupt*** (DBELL_CRIT) A Processor Doorbell Critical exception is generated on the processor when the processor has filtered the message based on the payload and has determined that it should accept the message. A Processor Doorbell

Critical Interrupt occurs when no higher priority exception exists, a Processor Doorbell Critical exception exists, and MSR_{CE}=1.

Message types other than these and their associated actions are implementation-dependent.

9.2.1.1 Doorbell Message Filtering

A processor receiving a DBELL message type will filter the message and either ignore the message or accept the message and generate a Processor Doorbell exception based on the payload and the state of the processor at the time the message is received.

The payload is specified in the message and is determined by the contents of register $RB_{37:63}$ used as the operand in the ***msgsnd*** instruction. The payload bits are defined below.

Bit **Description**

37 ***Broadcast*** (BRDCAST)
The message is accepted by all processors regardless of the value of the PIR register and the value of PIRTAG.

- 0 If the value of PIR and PIRTAG are equal a Processor Doorbell exception is generated.
- 1 A Processor Doorbell exception is generated regardless of the value of PIRTAG and PIR.

38:41 Reserved

50:63 ***PIR Tag*** (PIRTAG)
The contents of this field are compared with bits 50:63 of the PIR register.

If a DBELL message is received by a processor and either $payload_{BRDCAST}$=1 or $PIR_{50:63}$=$payload_{PIRTAG}$ then a Processor Doorbell exception is generated. The exception condition remains until a Processor Doorbell Interrupt is taken, or a ***msgclr*** instruction is executed on the receiving processor with a message type of DBELL. A change to any of the filtering criteria (i.e. changing the PIR register) will not clear a pending Processor Doorbell exception.

DBELL messages are not cumulative. That is, if a DBELL message is accepted and the interrupt is pended because MSR_{EE}=0, further DBELL messages that would be accepted are ignored until the Processor Doorbell exception is cleared by taking the interrupt or cleared by executing a ***msgclr*** with a message type of DBELL on the receiving processor.

The temporal relationship between when a DBELL message is sent and when it is received in a given processor is not defined.

9.2.1.2 Doorbell Critical Message Filtering

A processor receiving a DBELL_CRIT message type will filter the message and either ignore the message or accept the message and generate a Processor Doorbell Critical exception based on the payload and the state of the processor at the time the message is received.

The payload is specified in the message and is determined by the contents of register $RB_{37:63}$ used as the operand in the ***msgsnd*** instruction. The payload bits are defined below.

Bit **Description**

37 ***Broadcast*** (BRDCAST)
The message is accepted by all processors regardless of the value of the PIR register and the value of PIRTAG.

- 0 If the value of PIR and PIRTAG are equal a Processor Doorbell Critical exception is generated.
- 1 A Processor Doorbell Critical exception is generated regardless of the value of PIRTAG and PIR.

38:41 Reserved

50:63 ***PIR Tag*** (PIRTAG)
The contents of this field are compared with bits 50:63 of the PIR register.

If a DBELL_CRIT message is received by a processor and either $payload_{BRDCAST}$=1 or $PIR_{50:63}$=$payload_{PIRTAG}$ then a Processor Doorbell Critical exception is generated. The exception condition remains until a Processor Doorbell Critical Interrupt is taken, or a ***msgclr*** instruction is executed on the receiving processor with a message type of DBELL_CRIT. A change to any of the filtering criteria (i.e. changing the PIR register) will not clear a pending Processor Doorbell Critical exception.

DBELL_CRIT messages are not cumulative. That is, if a DBELL_CRIT message is accepted and the interrupt is pended because MSR_{CE}=0, further DBELL_CRIT messages that would be accepted are ignored until the Processor Doorbell Critical exception is cleared by taking the interrupt or cleared by executing a ***msgclr*** with a message type of DBELL_CRIT on the receiving processor.

The temporal relationship between when a DBELL_CRIT message is sent and when it is received in a given processor is not defined.

9.3 Processor Control Instructions

msgsnd and ***msgclr*** instructions are provided for sending and clearing messages to processors and other devices in the coherence domain. These instructions are privileged.

In the instruction descriptions the statement "this instructions is treated as a *Store*" means that the instruction is treated as a *Store* with respect to the storage access ordering mechanism caused by memory barriers in Section 1.7.1 of Book II.

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
send_msg_to_choherence_domain(msgtype, payload)
```

msgsnd sends a message to all devices in the coherence domain. The message contains a type and a payload. The message type (*msgtype*) is defined by the contents of $RB_{32:36}$ and the message payload is defined by the contents of $RB_{37:63}$. Message delivery is reliable and guaranteed. Each device may perform specific actions based on the message type and payload or may ignore messages. Consult the implementation user's manual for specific actions taken based on message type and payload.

For processors, actions taken on receipt of a message are defined in Section 9.2.1.

For storage access ordering, ***msgsnd*** is treated as a *Store* with respect to memory barriers.

This instruction is privileged.

Special Registers Altered:
None

Message Clear — *X-form*

msgclr RB

31	///	///	RB	238	/
0	6	11	16	21	31

```
msgtype ← GPR(RB)32:36
clear_received_message(msgtype)
```

msgclr clears a message of *msgtype* previously accepted by the processor executing the ***msgclr***. *msgtype* is defined by the contents of $RB_{32:36}$. A message is said to be cleared when a pending exception generated by an accepted message has not yet taken its associated interrupt.

If a pending exception exists for *msgtype* that exception is cleared at the completion of the ***msgclr*** instruction.

For processors, the types of messages that can be cleared are defined in Section 9.2.1.

This instruction is privileged.

Special Registers Altered:
None

Programming Note

Execution of a ***msgclr*** instruction that clears a pending exception when the associated interrupt is masked because the interrupt enable (MSR_{EE} or MSR_{CE}) is not set to 1 will always clear the pending exception (and thus the interrupt will not occur) if a subsequent instruction causes MSR_{EE} or MSR_{CE} to be set to 1.

Chapter 10. Synchronization Requirements for Context Alterations

Changing the contents of certain System Registers, the contents of TLB 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 certain bits in the MSR has the side effect of changing how instruction addresses are calculated. 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 5 (for data access) and Table 4 (for instruction fetch and execution).

The notation "CSI" in the tables means any context synchronizing instruction (e.g., ***sc***, ***isync***, ***rfi***, ***rfci***, ***rfmci***, or ***rfdi*** [Category: Embedded. Enhanced Debug]). 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 an ***rfi***, ***rfci***, ***rfmci***, or ***rfdi*** [Category:Embeddd.Enhanced Debug] that returns from an interrupt handler, provide the required synchronization.

No software synchronization is required before or after a context-altering instruction that is also context synchronizing (e.g., ***rfi***, etc.) 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 4, because all instructions preceding the context-altering instruction are fetched and decoded before the context-altering instruction is executed (the processor must determine whether any of these preceding instructions are context synchronizing).

Unless otherwise stated, the material in this chapter assumes a uniprocessor environment.

Instruction or Event	Required Before	Required After	Notes
interrupt	none	none	
rfi	none	none	
rfci	none	none	
rfmci	none	none	
rfdi[Category:E.ED]	none	none	
sc	none	none	
mtmsr (CM)	none	none	
mtmsr (ICM)	none	CSI	
mtmsr (UCLE)	none	none	
mtmsr (SPV)	none	none	
mtmsr (WE)	--	--	4
mtmsr (CE)	none	none	5
mtmsr (EE)	none	none	5
mtmsr (PR)	none	CSI	
mtmsr (FP)	none	CSI	
mtmsr (DE)	none	CSI	
mtmsr (ME)	none	CSI	3
mtmsr (FE0)	none	CSI	
mtmsr (FE1)	none	CSI	
mtmsr (IS)	none	CSI	2
mtspr (DEC)	none	none	8
mtspr (PID)	none	CSI	2
mtspr (IVPR)	none	none	
mtspr (DBSR)	--	--	6
mtspr (DBCR0,DBCR1)	--	--	6
mtspr (IAC1,IAC2,IAC3, IAC4)	--	--	6
mtspr (IVORi)	none	none	
mtspr (TSR)	none	none	8
mtspr (TCR)	none	none	8
tlbivax	none	CSI, or CSI and ***sync***	1,7
tlbwe	none	CSI, or CSI and ***sync***	1,7
wrtee	none	none	5
wrteei	none	none	5

Table 4: Synchronization requirements for instruction fetch and/or execution

Instruction or Event	Required Before	Required After	Notes
interrupt	none	none	
rfi	none	none	
rfci	none	none	
rfmci	none	none	
rfdi[Category:E.ED]	none	none	
sc	none	none	
mtmsr (CM)	none	CSI	
mtmsr (ICM)	none	none	
mtmsr (PR)	none	CSI	
mtmsr (ME)	none	CSI	3
mtmsr (DS)	none	CSI	
mtspr (PID)	CSI	CSI	
mtspr (DBSR)	--	--	6
mtspr (DBCR0,DBCR2)	---	---	6
mtspr (DAC1,DAC2, DVC1,DVC2)	--	--	6
tlbivax	CSI	CSI, or CSI and ***sync***	1,7
tlbwe	CSI	CSI, or CSI and ***sync***	1,7

Table 5: Synchronization requirements for data access

Notes:

1. There are additional software synchronization requirements for this instruction in multiprocessor environments (e.g., it may be necessary to invalidate one or more TLB entries on all processors in the multiprocessor system and to be able to determine that the invalidations have completed and that all side effects of the invalidations have taken effect); it is also necessary to execute a ***tlbsync*** instruction.
2. 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.
3. A context synchronizing instruction is required after altering MSR_{ME} to ensure that the alteration takes effect for subsequent Machine Check interrupts, which may not be recoverable and therefore may not be context synchronizing.
4. Synchronization requirements for changing the Wait State Enable are implementation-dependent,.
5. The effect of changing MSR_{EE} or MSR_{CE} is immediate.

If an ***mtmsr***, ***wrtee***, or ***wrteei*** instruction sets MSR_{EE} to '0', an External Input, DEC or FIT interrupt does not occur after the instruction is executed.

If an ***mtmsr***, ***wrtee***, or ***wrteei*** instruction changes MSR_{EE} from '0' to '1' when an External Input, Decrementer, Fixed-Interval Timer, or higher priority enabled exception exists, the corresponding interrupt occurs immediately after the ***mtmsr***, ***wrtee***, or ***wrteei*** is executed, and before the next instruction is executed in the program that set MSR_{EE} to '1'.

If an ***mtmsr*** instruction sets MSR_{CE} to '0', a Critical Input or Watchdog Timer interrupt does not occur after the instruction is executed.

If an ***mtmsr*** instruction changes MSR_{CE} from '0' to '1' when a Critical Input, Watchdog Timer or higher priority enabled exception exists, the corresponding interrupt occurs immediately after the ***mtmsr*** is executed, and before the next instruction is executed in the program that set MSR_{CE} to '1'.

6. Synchronization requirements for changing any of the Debug Facility Registers are implementation-dependent.

7. For data accesses, the context synchronizing instruction before the ***tlbwe*** or ***tlbivax*** instruction ensures that all storage accesses due to preceding instructions have completed to a point at which they have reported all exceptions they will cause.

   The context synchronizing instruction after the ***tlbwe*** or ***tlbivax*** ensures that subsequent storage accesses (data and instruction) will use the updated value in the TLB entry(s) being affected. It does not ensure that all storage accesses previously translated by the TLB entry(s) being updated have completed with respect to storage; if these completions must be ensured, the ***tlbwe*** or ***tlbivax*** must be followed by an ***sync*** instruction as well as by a context synchronizing instruction.

Programming Note

The following sequence illustrates why it is necessary, for data accesses, to ensure that all storage accesses due to instructions before the ***tlbwe*** or ***tlbivax*** have completed to a point at which they have reported all exceptions they will cause. Assume that valid TLB entries exist for the target storage location when the sequence starts.

- A program issues a load or store to a page.
- The same program executes a ***tlbwe*** or ***tlbivax*** that invalidates the corresponding TLB entry.
- The *Load* or *Store* instruction finally executes, and gets a TLB Miss exception.
- The TLB Miss exception is semantically incorrect. In order to prevent it, a context synchronizing instruction must be executed between steps 1 and 2.

8. The elapsed time between the Decrementer reaching zero, or the transition of the selected Time Base bit for the Fixed-Interval Timer or the Watchdog Timer, and the signalling of the Decrementer, Fixed-Interval Timer or the Watchdog Timer exception is not defined.

Appendix A. Implementation-Dependent Instructions

This appendix documents architectural resources that are allocated for specific implementation-sensitive functions which have scope-limited utility. Implementations may exercise reasonable flexibility in implementing these functions, but that flexibility should be limited to that allowed in this appendix.

A.1 Embedded Cache Initialization [Category: Embedded.Cache Initialization]

Data Cache Invalidate *X-form*

dci CT

31	/	CT	///	///	454	/
0	6	7	11	16	21	31

If CT is not supported by the implementation, this instruction designates the primary data cache as the target data cache.

If CT is supported by the implementation, let CT designate either the primary data cache or another level of the data cache hierarchy, as specified in Book II Section 3.2, as the target data cache.

The contents of the target data cache of the processor executing the ***dci*** instruction are invalidated.

Software must place a ***sync*** instruction before the ***dci*** to guarantee all previous data storage accesses complete before the ***dci*** is performed.

Software must place a ***sync*** instruction after the ***dci*** to guarantee that the ***dci*** completes before any subsequent data storage accesses are performed.

This instruction is privileged.

Special Registers Altered:
None

Extended Mnemonics:

Extended mnemonic for *Data Cache Invalidate*

Extended:	**Equivalent to:**
dccci	dci 0

Instruction Cache Invalidate *X-form*

ici CT

31	/	CT	///	///	966	/
0	6	7	11	16	21	31

If CT is not supported by the implementation, this instruction designates the primary instruction cache as the target instruction cache.

If CT is supported by the implementation, let CT designate either the primary instruction cache or another level of the instruction cache hierarchy, as specified in Book II Section 3.2, as the target instruction cache.

The contents of the target instruction cache of the processor executing the ***ici*** instruction are invalidated.

Software must place a ***sync*** instruction before the ***ici*** to guarantee all previous instruction storage accesses complete before the ***ici*** is performed.

Software must place an ***isync*** instruction after the ***ici*** to invalidate any instructions that may have already been fetched from the previous contents of the instruction cache after the ***isync***.

This instruction is privileged.

Special Registers Altered:
None

Extended Mnemonics:

Extended mnemonic for *Instruction Cache Invalidate*

Extended:	**Equivalent to:**
iccci	ici 0

A.2 Embedded Cache Debug Facility [Category: Embedded.Cache Debug]

A.2.1 Embedded Cache Debug Registers

A.2.1.1 Data Cache Debug Tag Register High

The Data Cache Debug Tag Register High (DCDBTRH) is a 32-bit Special Purpose Register (SPRN=0x39D). Data Cache Debug Tag Register High is read using ***mfspr*** and is set by ***dcread***.

DCDBTRH
32 … 63

Figure 25. Data Cache Debug Tag Register High

Programming Note

An example implementation of DCDBTRH could have the following content and format.

Bit(s)	Description
32:55	**Tag Real Address** (TRA) Bits 0:23 of the lower 32 bits of the 36-bit real address associated with this cache block
56	**Valid** (V) The valid indicator for the cache block (1 indicates valid)
57:59	Reserved
60:63	**Tag Extended Real Address** (TERA) Upper 4 bits of the 36-bit real address associated with this cache block

Implementations may support different content and format based on their cache implementation.

A.2.1.2 Data Cache Debug Tag Register Low

The Data Cache Debug Tag Register Low (DCDBTRL) is a 32-bit Special Purpose Register (SPRN=0x39C). Data Cache Debug Tag Register Low is read using ***mfspr*** and is set by ***dcread***.

DCDBTRL
32 … 63

Figure 26. Data Cache Debug Tag Register Low

Programming Note

An example implementation of DCDBTRL could have the following content and format.

Bit(s)	Description
32:44	Reserved (TRA)
45	**U bit parity** (UPAR**)**
46:47	**Tag parity** (TPAR)
48:51	**Data parity** (DPAR)
52:55	**Modified** (dirty) **parity** (MPAR)
56:59	**Dirty Indicators** (D) The “dirty” (modified) indicators for each of the four doublewords in the cache block
60	**U0 Storage Attribute** (U0) The U0 storage attribute for the page associated with this cache block
61	**U1 Storage Attribute** (U1) The U1 storage attribute for the page associated with this cache block
62	**U2 Storage Attribute** (U2) The U2 storage attribute for the page associated with this cache block
63	**U3 Storage Attribute** (U3) The U3 storage attribute for the page associated with this cache block

Implementations may support different content and format based on their cache implementation.

A.2.1.3 Instruction Cache Debug Data Register

The Instruction Cache Debug Data Register (ICDBDR) is a read-only 32-bit Special Purpose Register (SPRN=`0x3D3`). Instruction Cache Debug Data Register can be read using ***mfspr*** and is set by ***icread***.

ICDBDR	
32	63

Figure 27. Instruction Cache Debug Data Register

A.2.1.4 Instruction Cache Debug Tag Register High

The Instruction Cache Debug Tag Register High (ICDBTRH) is a 32-bit Special Purpose Register (SPRN=`0x39F`). Instruction Cache Debug Tag Register High is read using ***mfspr*** and is set by ***icread***.

ICDBTRH	
32	63

Figure 28. Instruction Cache Debug Tag Register High

Programming Note

An example implementation of ICDBTRH could have the following content and format.

Bit(s)	Description
32:55	**Tag Effective Address** (TEA) Bits 0:23 of the 32-bit effective address associated with this cache block
56	**Valid** (V) The valid indicator for the cache block (1 indicates valid)
57:58	**Tag parity** (TPAR)
59	**Instruction Data parity** (DPAR)
60:63	Reserved

Implementations may support different content and format based on their cache implementation.

A.2.1.5 Instruction Cache Debug Tag Register Low

The Instruction Cache Debug Tag Register Low (ICDBTRL) is a 32-bit Special Purpose Register (SPRN=`0x39E`). Instruction Cache Debug Tag Register Low is read using ***mfspr*** and is set by ***icread***.

ICDBTRL	
32	63

Figure 29. Instruction Cache Debug Tag Register Low

Programming Note

An example implementation of ICDBTRL could have the following content and format.

Bit(s)	Description
32:53	Reserved
54	**Translation Space** (TS) The address space portion of the virtual address associated with this cache block.
55	**Translation ID Disable** (TD) TID Disable field for the memory page associated with this cache block
56:63	**Translation ID** (TID) TID field portion of the virtual address associated with this cache block

Other implementations may support different content and format based on their cache implementation.

A.2.2 Embedded Cache Debug Instructions

Data Cache Read — *X-form*

dcread RT,RA,RB

31	RT	RA	RB	486	/
0	6	11	16	21	31

[Alternative Encoding]

31	RT	RA	RB	326	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + (RB)
C  ← log2(cache size)
B  ← log2(cache block size)
IDX← EA64-C:63-B
WD ← EA64-B:61
RT0:31← undefined
RT32:63← (data cache data)[IDX]WD×32:WD×32+31
DCDBTRH← (data cache tag high)[IDX]
DCDBTRL← (data cache tag low)[IDX]
```

Let the effective address (EA) be the sum of the contents of register RA, or 0 if RA is equal to 0, and the contents of register RB.

Let C = $\log_2$(cache size in bytes).
Let B = $\log_2$(cache block size in bytes).

$EA_{64-C:63-B}$ selects one of the 2^{C-B} data cache blocks.

$EA_{64-B:61}$ selects one of the data words in the selected data cache block.

The selected word in the selected data cache block is placed into register RT.

The contents of the data cache directory entry associated with the selected data cache block are placed into DCDBTRH and DCDBTRL (see Figure 25 and Figure 26).

dcread requires software to guarantee execution synchronization before subsequent ***mfspr*** instructions can read the results of the ***dcread*** instruction into GPRs. In order to guarantee that the ***mfspr*** instructions obtain the results of the ***dcread*** instruction, a sequence such as the following must be used:

```
msync                    # ensure that all previous
                         # cache operations have
                         # completed

dcread   regT,regA,regB# read cache information;

isync                    # ensure dcread completes
                         # before attempting to
                         # read results

mfspr    regD,dcdbtrh    # move high portion of tag
                         # into GPR D

mfspr    regE,dcdbtrl    # move low portion of tag
                         # into GPR E
```

This instruction is privileged.

Special Registers Altered:
DCDBTRH DCDBTRL

Programming Note

dcread can be used by a debug tool to determine the contents of the data cache, without knowing the specific addresses of the blocks which are currently contained within the cache.

Programming Note

Execution of ***dcread*** before the data cache has completed all cache operations associated with previously executed instructions (such as block fills and block flushes) is undefined.

Instruction Cache Read — *X-form*

icread RA,RB

31	///	RA	RB	998	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + (RB)
C  ← log2(cache size)
B  ← log2(cache block size)
IDX← EA64-C:63-B
WD ← EA64-B:61
ICDBDR← (instruction cache data)[IDX]WD×32:WD×32+31
ICDBTRH← (instruction cache tag high)[IDX]
ICDBTRL← (instruction cache tag low)[IDX]
```

Let the effective address (EA) be the sum of the contents of register RA, or 0 if RA is equal to 0, and the contents of register RB.

Let C = $\log_2$(cache size in bytes).
Let B = $\log_2$(cache block size in bytes).

$EA_{64-C:63-B}$ selects one of the 2^{C-B} instruction cache blocks.

$EA_{64-B:61}$ selects one of the data words in the selected instruction cache block.

The selected word in the selected instruction cache block is placed into ICDBDR.

The contents of the instruction cache directory entry associated with the selected cache block are placed into ICDBTRH and ICDBTRL (see Figure 28 and Figure 29).

icread requires software to guarantee execution synchronization before subsequent ***mfspr*** instructions can read the results of the ***icread*** instruction into GPRs. In order to guarantee that the ***mfspr*** instructions obtain the results of the ***icread*** instruction, a sequence such as the following must be used:

```
icread    regA,regB    # read cache information

isync                  # ensure icread completes
                       # before attempting to
                       # read results

mficdbdr  regC         # move instruction
                       # information into GPR C

mficdbtrh regD         # move high portion of
                       # tag into GPR D

mficdbtrl regE         # move low portion of tag
                       # into GPR E
```

This instruction is privileged.

Special Registers Altered:
ICDBDR ICDBTRH ICDBTRL

Programming Note

icread can be used by a debug tool to determine the contents of the instruction cache, without knowing the specific addresses of the blocks which are currently contained within the cache.

Appendix B. 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 III.

Assemblers should provide the extended mnemonics and symbols listed here, and may provide others.

B.1 Move To/From Special Purpose Register Mnemonics

This section defines extended mnemonics for the ***mtspr*** and ***mfspr*** instructions, including the Special Purpose Registers (SPRs) defined in Book I and certain privileged SPRs, 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).

Table 6: Extended mnemonics for moving to/from an SPR

Special Purpose Register	**Move To SPR**		**Move From SPR**	
	Extended	**Equivalent to**	**Extended**	**Equivalent to**
Fixed-Point Exception Register	mtxer Rx	mtspr 1,Rx	mfxer Rx	mfspr Rx,1
Link Register	mtlr Rx	mtspr 8,Rx	mflr Rx	mfspr Rx,8
Count Register	mtctr Rx	mtspr 9,Rx	mfctr Rx	mfspr Rx,9
Decrementer	mtdec Rx	mtspr 22,Rx	mfdec Rx	mfspr Rx,22
Save/Restore Register 0	mtsrr0 Rx	mtspr 26,Rx	mfsrr0 Rx	mfspr Rx,26
Save/Restore Register 1	mtsrr1 Rx	mtspr 27,Rx	mfsrr1 Rx	mfspr Rx,27
Special Purpose Registers G0 through G3	mtsprg n,Rx	mtspr 272+n,Rx	mfsprg Rx,n	mfspr Rx,272+n
Time Base [Lower]	mttbl Rx	mtspr 284,Rx	mftb Rx	mfspr Rx,268
Time Base Upper	mttbu Rx	mtspr 285,Rx	mftbu Rx	mfspr Rx,269
Processor Version Register	-	-	mfpvr Rx	mfspr Rx,287

Appendix C. Guidelines for 64-bit Implementations in 32-bit Mode and 32-bit Implementations

C.1 Hardware Guidelines

C.1.1 64-bit Specific Instructions

The instructions in the Category: 64-Bit are considered restricted only to 64-bit processing. A 32-bit implementation need not implement the group; likewise, the 32-bit applications will not utilize any of these instructions. All other instructions shall either be supported directly by the implementation, or sufficient infrastructure will be provided to enable software emulation of the instructions. A 64-bit implementation that is executing in 32-bit mode may choose to take an Unimplemented Instruction Exception when these 64-bit specific instructions are executed.

C.1.2 Registers on 32-bit Implementations

The Power ISA provides 32-bit and 64-bit registers. All 32-bit registers shall be supported as defined in the specification except the MSR. The MSR shall be supported as defined in the specification except that bits 32:33 (CM and ICM) are treated as reserved bits. Only bits 32:63 of the 64-bit registers are required to be implemented in hardware in a 32-bit implementation except for the 64-bit FPRs. Such 64-bit registers include the LR, the CTR, the XER, the 32 GPRs, SRR0 and CSRR0.

Likewise, other than floating-point instructions, all instructions which are defined to return a 64-bit result shall return only bits 32:63 of the result on a 32-bit implementation.

C.1.3 Addressing on 32-bit Implementations

Only bits 32:63 of the 64-bit instruction and data storage effective addresses need to be calculated and presented to main storage. Given that the only branch and data storage access instructions that are not included in Section C.1.1 are defined to prepend 32 0s to bits 32:63 of the effective address computation, a 32-bit implementation can simply bypass the prepending of the 32 0s when implementing these instructions. For Branch to Link Register and Branch to Count Register instructions, given the LR and CTR are implemented only as 32-bit registers, only concatenating 2 0s to the right of bits 32:61 of these registers is necessary to form the 32-bit branch target address.

For next sequential instruction address computation, the behavior is the same as for 64-bit implementations in 32-bit mode.

C.1.4 TLB Fields on 32-bit Implementations

32-bit implementations should support bits 32:53 of the Effective Page Number (EPN) field in the TLB. This size provides support for a 32-bit effective address, which Power ISA ABIs may have come to expect to be available. 32-bit implementations may support greater than 32-bit real addresses by supporting more than bits 32:53 of the Real Page Number (RPN) field in the TLB.

C.2 32-bit Software Guidelines

C.2.1 32-bit Instruction Selection

Any software that uses any of the instructions listed in Category: 64-Bit shall be considered 64-bit software, and correct execution cannot be guaranteed on 32-bit implementations. Generally speaking, 32-bit software should avoid using any instruction or instructions that depend on any particular setting of bits 0:31 of any 64-bit application-accessible system register, including General Purpose Registers, for producing the correct 32-bit results. Context switching may or may not preserve the upper 32 bits of application-accessible 64-bit system registers and insertion of arbitrary settings of those upper 32 bits at arbitrary times during the execution of the 32-bit application must not affect the final result.

Appendix D. Type FSL Storage Control [Category: Embedded.MMU Type FSL]

D.1 Type FSL Storage Control Overview

The Embedded category provides two different memory management and TLB programming models from which an implementation may choose. Both models use the same definition of the general contents of a Translation Lookaside Buffer (TLB) entry, but differ on what methods and resources are used to manipulate the TLB itself. The programming model presented here is called Type FSL and it defines functions and structures that are visible to software. These are divided into the following areas:

- The TLB itself. The TLB consists of one or more structures called TLB arrays each of which may have differing characteristics.
- The address translation mechanism.
- Methods and effects of changing and manipulating TLB arrays.
- Configuration information available to the operating system that describes the structure and form of the TLB arrays and translation mechanism.

The TLB structure and the methods of performing translations are called the Memory Management Unit (MMU).

The programming model for reading and writing TLBs is software managed. Hardware page table formats are not defined and software is free to choose any form in which to hold information about address translation. Address translation is accomplished through a set of TLB arrays, PID registers, and address space identifiers from the MSR, all of which are software managed.

TLB entries are used to translate both instruction and data memory references providing a unified memory management model.

D.2 Type FSL Storage Control Registers

D.2.1 Process ID Registers (PID*n*)

Process ID Registers are used by system software to specify which TLB entries are used by the processor to accomplish address translation for loads, stores, and instruction fetches. Section 4.7.1.1 defines the PID register. The PID register is synonymous with PID0. In addition to PID0, 2 additional PID registers, PID1 and PID2 are defined. An implementation may choose to provide any number of PIDs up to a maximum of 3. The number of PIDs implemented is indicated by the value of $MMUCFG_{NPIDS}$ and the number of bits implemented in each PID register is indicated by the value of $MMUCFG_{PIDSIZE}$. PID values are used to construct virtual addresses for accessing memory.

PID*n*	
32	63

Figure 30. Process ID Register (PID0–PID2)

Bit	Description
32:49	Reserved
50:63	***Process ID*** Identifies the process

Programming Note

The suggested software convention for PID usage is to use PID0 to denote private mappings for a process and to use other PIDs to handle mappings that may be common to multiple processes. This method allows for processes sharing address space to also share TLB entries if the shared address space is mapped at the same virtual address in each process.

D.2.2 Translation Lookaside Buffer

The MMU contains up to four TLB arrays. TLB arrays are on-chip storage areas for holding TLB entries. A

TLB entry contains effective to real address mappings for loads, stores, and instruction fetches. A TLB array contains zero or more TLB entries. Each of the TLB entries has specific fields that can be accessed using the corresponding fields in the MMU Assist Registers (see Section D.2.4). Each TLB array that is implemented has a configuration register (TLB*n*CFG) associated with it describing the size and attributes of the TLB entries in that array (see Section D.2.5.2).

A TLB entry contains the fields described in Section 4.7.1.2 as well as these additional fields:

Field	Description
IPROT	Invalidation protection. This entry is protected from all TLB invalidation mechanisms except the explicit writing of a 0 to the V bit.
ACM	The Alternate Coherency Mode (ACM) attribute allows an implementation to employ more than a single coherency method. This allows for a processor to participate in multiple coherency protocols. If the M attribute (Memory Coherence Required) is not set for a page (M=0), the page has no coherency associated with it and the ACM attribute is ignored. If the M attribute is set to 1 for a page (M=1), the ACM attribute is used to determine the coherence domain (or protocol) used. The values for ACM are implementation-dependent.

D.2.3 Address Space Identifiers

The address space identifier is called the AS bit. Thus there are two possible address spaces, 0 and 1. The value of the AS bit (see Section 4.7.2, Figure 8) is determined by the type of translation performed and from the contents of the MSR when an address is translated. If the type of translation performed is an instruction fetch, the value of the AS bit is taken from the contents of MSR_{IS}. If the type of translation performed is a load, store, or other data translation including target addresses of software initiated instruction fetch hints and locks the value of the AS bit is taken from the contents of MSR_{DS}.

Programming Note

While system software is free to use address space bits as it sees fit, it should be noted that on interrupt, the MSR_{IS} and MSR_{DS} bits are set to 0. This encourages software to use address space 0 for system software and address space 1 for user software.

D.2.4 MMU Assist Registers

The MMU Assist Registers (MAS) are used to transfer data to and from the TLB arrays. MAS registers can be read and written by software using ***mfspr*** and ***mtspr*** instructions. Execution of a ***tlbre*** instruction causes the TLB entry specified by $MAS0_{TLBSEL}$, $MAS0_{ESEL}$, and $MAS2_{EPN}$ to be copied to the MAS registers. Conversely, execution of a ***tlbwe*** instruction causes the TLB entry specified by $MAS0_{TLBSEL}$, $MAS0_{ESEL}$, and $MAS2_{EPN}$ to be written with contents of the MAS registers. MAS registers may also be updated by hardware on the occurrence of an Instruction or Data TLB Error interrupt or as the result of a ***tlbsx*** instruction.

All MAS registers are privileged. All MAS registers with the exception of MAS7 must be implemented. MAS7 is not required to be implemented if the processor supports 32 bits or less of real address.

Processors are only required to implement the necessary bits of any multi-bit field in a MAS register such that only the resources supplied by the processor are represented. Any non-implemented bits in a field should have no effect when writing and should always read as zero. For example, a processor that implements only 2 TLB arrays will likely only implement the lower-order bit of the $MAS0_{TLBSEL}$ field.

D.2.4.1 MAS0 Register

The MAS0 register contains fields for identifying and selecting a TLB entry.

MAS0	
32	63

Figure 31. MAS0 register

These bits are interpreted as follows:

Bit	Description
32:33	Reserved
34:35	***TLB Select*** (TLBSEL) Selects TLB for access. 00 TLB0 01 TLB1 10 TLB2 11 TLB3
36:47	***Entry Select*** (ESEL) Identifies an entry in the selected array to be used for ***tlbwe*** and ***tlbre***. Valid values for ESEL are from 0 to $TLBnCFG_{ASSOC}$ - 1. That is, ESEL selects the entry in the TLB array from the set of entries which can be used for translating addresses with the EPN specified by $MAS2_{EPN}$. For fully-associative TLB arrays, ESEL ranges from 0 to $TLBnCFG_{NENTRY}$ - 1. ESEL is also updated on TLB error exceptions (misses), and ***tlbsx*** hit and miss cases.
48:51	Reserved
52:63	***Next Victim*** (NV) NV is a hint to software to identify the next victim to be targeted for a TLB miss replacement

operation for those TLBs that support the NV field. If the TLB selected by $MAS0_{TLBSEL}$ does not support the NV field, then this field is undefined. The computation of this field is implementation-dependent. NV is updated on TLB error exceptions (misses), ***tlbsx*** hit and miss cases as shown in Table 7, and on execution of ***tlbre*** if the TLB array being accessed supports the NV field. When NV is updated by a supported TLB array, the NV field will always present a value that can be used in the $MAS0_{ESEL}$ field.

D.2.4.2 MAS1 Register

The MAS1 register contains fields for selecting a TLB entry during translation.

MAS1
32 … 63

Figure 32. MAS1 register

These bits are interpreted as follows:

Bit Definition

32 ***TLB Valid Bit*** (V)

0 This TLB entry is invalid.
1 This TLB entry is valid.

33 ***Invalidate Protect*** (IPROT)
Indicates this TLB entry is protected from invalidate operations due to execution of ***tlbivax***, ***tlbivax*** invalidations from another processor, or invalidate all operations. IPROT is only implemented for TLB entries in TLB arrays where $TLBnCFG_{IPROT}$ is indicated.

0 Entry is not protected from invalidation
1 Entry is protected from invalidation.

34:47 ***Translation Identity*** (TID)
During translation, TID is compared with the current process IDs (PIDs) to select a TLB entry. A TID value of 0 defines an entry as global and matches with all process IDs.

48:50 Reserved

51 ***Translation Space*** (TS)
During translation, TS is compared with AS (the IS or DS fields of the MSR depending on the type of access) to select a TLB entry.

52:55 ***Translation Size*** (TSIZE)
TSIZE defines the page size of the TLB entry. For TLB arrays that contain fixed-size TLB entries, this field is ignored. For variable page size TLB arrays, the page size is 4^{TSIZE} Kbytes. TSIZE must be between $TLBnCFG_{MINSIZE}$ and $TLBnCFG_{MAXSIZE}$. Encodings for page size are defined in Section 4.7.1.2.

56:63 Reserved

D.2.4.3 MAS2 Register

The MAS2 register is a 64-bit register in 64-bit mode and a 32-bit register in 32-bit mode. The register contains fields for specifying the effective page address and the storage attributes for a TLB entry.

MAS2
0 … 63

Figure 33. MAS2 register

These bits are interpreted as follows:

Bit Description

0:51 ***Effective Page Number*** (EPN)
Depending on page size, only the bits associated with a page boundary are valid. Bits that represent offsets within a page are ignored and should be zero. $EPN_{0:31}$ are accessible only in 64-bit implementations as the upper 32 bits of the effective address of the page.

52:55 Reserved

56:57 ***Alternate Coherency Mode*** (ACM)
The ACM attribute allows an implementation to employ more than a single coherency method. This allows for a processor to participate in multiple coherency protocols. If the M attribute (Memory Coherence Required) is not set for a page (M=0), the page has no coherency associated with it and the ACM attribute is ignored. If the M attribute is set to 1 for a page (M=1), the ACM attribute is used to determine the coherence domain (or protocol) used. The values for ACM are implementation-dependent.

Programming Note

Some previous implementations may have a storage bit in the bit 57 position labeled as X0.

58 ***VLE Mode*** (VLE)
[Category: VLE]
Identifies pages which contain instructions to be decoded as VLE instructions (see Chapter 1 of Book VLE). Setting the VLE attribute to 1 and setting the E attribute to 1 is considered a programming error and an attempt to fetch instructions from a page so marked produces an Instruction Storage Interrupt Byte Ordering Exception and sets ESR_{BO}.

0 Instructions fetched from the page are decoded and executed as non-VLE instructions.

1 Instructions fetched from the page are decoded and executed as VLE instructions.

Programming Note

Some previous implementations may have a storage bit in this position labeled as X1. Software should not use the presence of this bit (the ability to set to 1 and read a 1) to determine if the implementation supports the VLE.

59 ***Write Through*** (W)

- 0 This page is not Write-Through Required storage.
- 1 This page is Write-Through Required storage.

60 ***Caching Inhibited*** (I)

- 0 This page is not Caching Inhibited storage.
- 1 This page is Caching Inhibited storage

61 ***Memory Coherence Required*** (M)

- 0 This page is not Memory Coherence Required storage.
- 1 This page is Memory Coherence Required storage.

62 ***Guarded*** (G)

- 0 This page is not Guarded storage.
- 1 This page is Guarded storage.

63 ***Endianness*** (E)

- 0 The page is accessed in Big-Endian byte order.
- 1 The page is accessed in Little-Endian byte order.

D.2.4.4 MAS3 Register

The MAS3 register contains fields for specifying the real page address, user defined attributes, and the permission attributes for a TLB entry.

MAS3	
32	63

Figure 34. MAS3 register

These bits are interpreted as follows:

Bit **Description**

32:51 ***Real Page Number (bits 32:51)*** (RPNL or $RPN_{32:51}$)
Depending on page size, only the bits associated with a page boundary are valid. Bits that represent offsets within a page are ignored and should be zero. $RPN_{0:31}$ are accessed through MAS7.

52:53 Reserved

54:57 ***User Bits*** (U0:U3)
These bits are associated with a TLB entry and can be used by system software. For example, these bits may be used to hold information useful to a page scanning algorithm or be used to mark more abstract page attributes.

58:63 ***Permission Bits*** (UX, SX, UW, SW, UR, SR). User and supervisor execute, write, and read permission bits. The effect of the Permission Bits are defined in Section 4.7.1.2.

D.2.4.5 MAS4 Register

The MAS4 register contains fields for specifying default information to be pre-loaded on certain MMU related exceptions. See Section D.4.5 for more information.

MAS4	
32	63

Figure 35. MAS4 register

The MAS4 fields are described below.

Bit **Description**

32:33 Reserved

34:35 ***TLBSEL Default Value*** (TLBSELD)
Specifies the default value loaded in $MAS0_{TLBSEL}$ on a TLB miss exception.

36:43 Reserved

44:47 ***TID Default Selection Value*** (TIDSELD)
Specifies which of the current PID registers should be used to load the $MAS1_{TID}$ field on a TLB miss exception.

The PID registers are addressed as follows:

0000 = PID0 (PID)
0001 = PID1
0010 = PID2

A value that references a non-implemented PID register causes a value of 0 to be placed in $MAS1_{TID}$.

48:51 Reserved

52:55 ***Default TSIZE Value*** (TSIZED)
Specifies the default value loaded into $MAS1_{TSIZE}$ on a TLB miss exception.

56:57 ***Default ACM Value*** (ACMD)
Specifies the default value loaded into $MAS2_{ACM}$ on a TLB miss exception.

58 ***Default VLE Value*** (VLED)
Specifies the default value loaded into $MAS2_{VLE}$ on a TLB miss exception.

59 ***Default W Value*** (WD)
Specifies the default value loaded into $MAS2_W$ on a TLB miss exception.

60 ***Default I Value*** (ID)
Specifies the default value loaded into $MAS2_I$ on a TLB miss exception.

61 ***Default M Value*** (MD)
Specifies the default value loaded into $MAS2_M$ on a TLB miss exception.

62 ***Default G Value*** (GD)
Specifies the default value loaded into $MAS2_G$ on a TLB miss exception.

63 ***Default E Value*** (ED)
Specifies the default value loaded into $MAS2_E$ on a TLB miss exception.

D.2.4.6 MAS6 Register

The MAS6 register contains fields for specifying PID and AS values to be used when searching TLB entries with the ***tlbsx*** instruction.

MAS6	
32	63

Figure 36. MAS6 register

These bits are interpreted as follows:

Bit **Description**

32:33 Reserved

34:47 ***Search PID0*** (SPID0)
Specifies the value of PID0 used when searching the TLB during execution of ***tlbsx***. This field is valid for only the number of bits implemented for PID registers.

48:62 Reserved

63 ***Address Space Value for Searches*** (SAS)
Specifies the value of AS used when searching the TLB during execution of ***tlbsx***.

D.2.4.7 MAS7 Register

The MAS7 register contains the high order address bits of the RPN for implementations that support more than 32 bits of physical address. Implementations that do not support more than 32 bits of physical addressing are not required to implement MAS7.

MAS7	
32	63

Figure 37. MAS7 register

These bits are interpreted as follows:

Bit **Description**

32:63 ***Real Page Number*** (bits 0:31) (RPNU or $RPN_{0:31}$)
$RPN_{32:51}$ are accessed through MAS3.

Table 7: MAS Register Update Summary

MAS Field Updated	Value Loaded on Event			
	Data or Instruction TLB Error Interrupt	***tlbsx* hit**	***tlbsx* miss**	***tlbre***
$MAS0_{TLBSEL}$	$MAS4_{TLBSELD}$	TLB array that hit	$MAS4_{TLBSELD}$	—
$MAS0_{ESEL}$	***if*** TLB array [$MAS4_{TLBSELD}$] supports next victim ***then*** hardware hint, ***else*** undefined	Number of entry that hit	***if*** TLB array [$MAS4_{TLBSELD}$] supports next victim ***then*** hardware hint, ***else*** undefined	—
$MAS0_{NV}$	***if*** TLB array [$MAS4_{TLBSELD}$] supports next victim ***then*** next hardware hint, ***else*** undefined	***if*** TLB array [$MAS4_{TLBSELD}$] supports next victim ***then*** hardware hint, ***else*** undefined	***if*** TLB array [$MAS4_{TLBSELD}$] supports next victim ***then*** next hardware hint, ***else*** undefined	***if*** TLB array [$MAS4_{TLBSELD}$] supports next victim ***then*** hardware hint, ***else*** undefined
$MAS1_{V}$	1	1	0	TLB_{V}
$MAS1_{IPROT}$	0	TLB_{IPROT}	0	TLB_{IPROT}
$MAS1_{TID}$	***if*** PID[$MAS4_{TIDSELD}$] implemented ***then*** PID[$MAS4_{TIDSELD}$] ***else*** 0	TLB_{TID}	$MAS6_{SPID0}$	TLB_{TID}
$MAS1_{TS}$	MSR_{IS} or MSR_{DS}	TLB_{TS}	$MAS6_{SAS}$	TLB_{TS}
$MAS1_{TSIZE}$	$MAS4_{TSIZED}$	TLB_{SIZE}	$MAS4_{TSIZED}$	TLB_{SIZE}
$MAS2_{EPN}$	$EA_{0:51}$[1]	TLB_{EPN}	undefined	TLB_{EPN}
$MAS2_{ACM}$	$MAS4_{ACMD}$	TLB_{ACM}	$MAS4_{ACMD}$	TLB_{ACM}
$MAS2_{VLE}$	$MAS4_{VLED}$	TLB_{VLE}	$MAS4_{VLED}$	TLB_{VLE}
$MAS2_{W}$	$MAS4_{WD}$	TLB_{W}	$MAS4_{WD}$	TLB_{W}
$MAS2_{I}$	$MAS4_{ID}$	TLB_{I}	$MAS4_{ID}$	TLB_{I}
$MAS2_{M}$	$MAS4_{MD}$	TLB_{M}	$MAS4_{MD}$	TLB_{M}
$MAS2_{G}$	$MAS4_{GD}$	TLB_{G}	$MAS4_{GD}$	TLB_{G}
$MAS2_{E}$	$MAS4_{ED}$	TLB_{E}	$MAS4_{ED}$	TLB_{E}
$MAS3_{RPN}$	0	TLB_{RPN} (bits 32:51)	0	TLB_{RPN} (bits 32:51)
$MAS3_{U0\ U1\ U2\ U3}$	0	$TLB_{U0\ U1\ U2\ U3}$	0	$TLB_{U0\ U1\ U2\ U3}$
$MAS3_{UX\ SX\ UW\ SW\ UR\ SR}$	0	$TLB_{UX\ SX\ UW\ SW\ UR\ SR}$	0	$TLB_{UX\ SX\ UW\ SW\ UR\ SR}$
MAS4	—	—	—	—
$MAS6_{SPID0}$	PID0	—	—	—
$MAS6_{SAS}$	MSR_{IS} or MSR_{DS}	—	—	—
$MAS7_{RPN}$	0	TLB_{RPN} (bits 0:31)	0	TLB_{RPN} (bits 0:31)

1. If MSR_{CM}=0 (32-bit mode) at the time of the exception, $EPN_{0:31}$ are set to 0.

D.2.5 MMU Configuration and Control Registers

D.2.5.1 MMU Configuration Register (MMUCFG)

The read-only MMUCFG register is described as follows.

MMUCFG	
32	63

Figure 38. MMU Configuration Register

These bits are interpreted as follows:

Bit Description

32:39 Reserved

40:46 ***Real Address Size*** (RASIZE)
Number of bits in a real address supported by the implementation.

47:48 Reserved

49:52 ***Number of PID Registers*** (NPIDS)
Indicates the number of PID registers provided by the processor.

53:57 ***PID Register Size*** (PIDSIZE)
The value of PIDSIZE is one less than the number of bits implemented for each of the PID registers implemented by the processor. The processor implements only the least significant PIDSIZE+1 bits in the PID registers. The maximum number of PID register bits that may be implemented is 14.

58:59 Reserved

60:61 ***Number of TLBs*** (NTLBS)
The value of NTLBS is one less than the number of software-accessible TLB structures that are implemented by the processor. NTLBS is set to one less than the number of TLB structures so that its value matches the maximum value of $MAS0_{TLBSEL}$.

00 1 TLB
01 2 TLBs
10 3 TLBs
11 4 TLBs

62:63 ***MMU Architecture Version Number*** (MAVN)
Indicates the version number of the architecture of the MMU implemented by the processor.

00 Version 1.0
01 Reserved
10 Reserved
11 Reserved

D.2.5.2 TLB Configuration Registers (TLB*n*CFG)

The TLB*n*CFG read-only registers provide information about each specific TLB that is implemented. There is one TLB*n*CFG register implemented for each TLB array that is implemented. TLB0CFG corresponds to TLB0, TLB1CFG corresponds to TLB1, etc.

TLB*n*CFG provides configuration information for the corresponding TLB array.

TLB*n*CFG	
32	63

Figure 39. TLB Configuration Register

These bits are interpreted as follows:

Bit Description

32:39 ***Associativity*** (ASSOC)
Total number of entries in a TLB array which can be used for translating addresses with a given EPN. This number is referred to as the associativity level of the TLB array. A value equal to NENTRY or 0 indicates the array is fully-associative.

40:43 ***Minimum Page Size*** (MINSIZE)
Minimum page size of TLB array. Page size encoding is defined in Section 4.7.1.2.

44:47 ***Maximum Page Size*** (MAXSIZE)
Maximum page size of TLB array. Page size encoding is defined in Section 4.7.1.2.

48 ***Invalidate Protection*** (IPROT)
Invalidate protect capability of TLB array.

0 Indicates invalidate protection capability not supported.
1 Indicates invalidate protection capability supported.

49 ***Page Size Availability*** (AVAIL)
Page size availability of TLB array.

0 Fixed selectable page size from MINSIZE to MAXSIZE (all TLB entries are the same size).
1 Variable page size from MINSIZE to MAXSIZE (each TLB entry can be sized separately).

50:51 Reserved

52:63 ***Number of Entries*** (NENTRY)
Number of entries in TLB array.

D.2.5.3 MMU Control and Status Register (MMUCSR0)

The MMUCSR0 register is used for general control of the MMU including invalidation of the TLB arrays and page sizes for programmable fixed size arrays. For TLB arrays that have programmable fixed sizes, the TLB*n*_PS fields allow software to specify the page size.

MMUCSR0
32 … 63

Figure 40. MMU Control and Status Register 0

These bits are interpreted as follows:

Bit **Description**

32:40 Reserved

41:56 ***TLBn Array Page Size***
A 4-bit field specifies the page size for TLB*n* array. Page size encoding is defined in Section 4.7.1.2. For each TLB array *n*, the field is implemented only if TLB*n*CFG_{AVAIL}=0 and TLB*n*$CFG_{MINSIZE}$≠TLB*n*$CFG_{MAXSIZE}$. If the value of TLB*n*_PS is not between TLB*n*$CFG_{MINSIZE}$ and TLB*n*$CFG_{MAXSIZE}$ the page size is set to TLB*n*$CFG_{MINSIZE}$.

41:44 ***TLB3 Array Page Size*** (TLB3_PS)
Page size of the TLB3 array.

45:48 ***TLB2 Array Page Size*** (TLB2_PS)
Page size of the TLB2 array.

49:52 ***TLB1 Array Page Size*** (TLB1_PS)
Page size of the TLB1 array.

53:56 ***TLB0 Array Page Size*** (TLB0_PS)
Page size of the TLB0 array.

57:62 ***TLBn Invalidate All***
TLB invalidate all bit for the TLB*n* array.

0 If this bit reads as a 1, an invalidate all operation for the TLB*n* array is in progress. Hardware will set this bit to 0 when the invalidate all operation is completed. Writing a 0 to this bit during an invalidate all operation is ignored.

1 TLB*n* invalidation operation. Hardware initiates a TLB*n* invalidate all operation. When this operation is complete, this bit is cleared. Writing a 1 during an invalidate all operation produces an undefined result. If the TLB array supports IPROT, entries that have IPROT set will not be invalidated.

57 ***TLB2 Invalidate All*** (TLB2_FI)
TLB invalidate all bit for the TLB2 array.

58 ***TLB3 Invalidate All*** (TLB3_FI)
TLB invalidate all bit for the TLB3 array.

59:60 Reserved

61 ***TLB0 Invalidate All*** (TLB0_FI)
TLB invalidate all bit for the TLB0 array.

62 ***TLB1 Invalidate All*** (TLB1_FI)
TLB invalidate all bit for the TLB1 array.

63 Reserved

Programming Note

Changing the fixed page size of an entire array must be done with great care. If any entries in the array are valid, changing the page size may cause those entries to overlap, creating a serious programming error. It is suggested that the entire TLB array be invalidated and any entries with IPROT have their V bits set to zero before changing page size.

D.3 Page Identification and Address Translation

Page Identification occurs as described in Section 4.7.2 except the matching TLB entry may be identified using more than one PID register. Accesses that would result in multiple matching entries are not allowed and are considered a serious programming error by system software and the results of such a translation are undefined. A PID register containing a 0 value (or the same value as another PID register) will form a non unique match and is permissible.

Once a match occurs the matching TLB entry is used for access control, storage attributes, and effective to real address translation.

D.4 TLB Management

D.4.1 Reading TLB Entries

TLB entries can be read by executing ***tlbre*** instructions. At the time of ***tlbre*** execution, the MAS registers are used to index a specific TLB entry and upon completion of the ***tlbre*** instruction, the MAS registers will contain the contents of the indexed TLB entry.

Specifying invalid values for $MAS0_{TLBSEL}$ and $MAS0_{ESEL}$ produce undefined results.

D.4.2 Writing TLB Entries

TLB entries can be written by executing ***tlbwe*** instructions. At the time of ***tlbwe*** execution, the MAS registers are used to index a specific TLB entry and contain the contents to be written to the indexed TLB entry. Upon completion of the ***tlbwe*** instruction, the contents of the MAS registers corresponding to TLB entry fields will be written to the indexed TLB entry.

Specifying invalid values for $MAS0_{TLBSEL\ ESEL}$ produces undefined results.

D.4.3 Invalidating TLB Entries

TLB entries may be invalidated by three different methods. The TLB entry can be invalidated as the result of a ***tlbwe*** instruction that sets the $MAS1_V$ bit in the entry to 0. TLB entries may also be invalidated as a result of a ***tlbivax*** instruction or from an invalidation resulting from a ***tlbivax*** on another processor. Lastly, TLB entries may be invalidated as a result of an invalidate all operation specified through appropriate settings in the MMUCSR0.

In both multiprocessor and uniprocessor systems, invalidations can occur on a wider set of TLB entries than intended. That is, a virtual address presented for invalidation may cause not only the intended TLB targeted for invalidation to be invalidated, but may also invalidate other TLB entries depending on the implementation. This is because parts of the translation mechanism may not be fully specified to the hardware at invalidate time. This is especially true in SMP systems, where the invalidation address must be supplied to all processors in the system, and there may be other limitations imposed by the hardware implementation. This phenomenon is known as generous invalidates. The architecture assures that the intended TLB will be invalidated, but does not guarantee that it will be the only one. A TLB entry invalidated by writing the V bit of the TLB entry to 0 by use of a ***tlbwe*** instruction is guaranteed to invalidate only the addressed TLB entry. Invalidates occurring from ***tlbivax*** instructions or from ***tlbivax*** instructions on another processor may cause generous invalidates.

The architecture provides a method to protect against generous invalidations. This is important since there are certain virtual memory regions that must be properly mapped to make forward progress. To prevent this, the architecture specifies an IPROT bit for TLB entries. If the IPROT bit is set to 1 in a given TLB entry, that entry is protected from invalidations resulting from ***tlbivax*** instructions, or from invalidate all operations. TLB entries with the IPROT field set may only be invalidated by explicitly writing the TLB entry and specifying a 0 for the V ($MAS1_V$) field.

Programming Note

The most obvious issue with generous invalidations is the code memory region that serves as the exception handler for MMU faults. If this region does not have a valid mapping, an MMU exception cannot be handled because the first address of the exception handler will result in another MMU exception.

Programming Note

Not all TLB arrays in a given implementation will implement the IPROT attribute. It is likely that implementations that are suitable for demand page environments will implement it for only a single array, while not implementing it for other TLB arrays.

Programming Note

Operating systems need to use great care when using protected (IPROT) TLB entries, particularly in SMP systems. An SMP system that contains TLB entries on other processors will require a cross processor interrupt or some other synchronization mechanism to assure that each processor performs the required invalidation by writing its own TLB entries.

Programming Note

To ensure a TLB entry that is not protected by IPROT is invalidated if software does not know which TLB array the entry is in, software should issue a ***tlbivax*** instruction targeting each TLB in the implementation with the EA to be invalidated.

Programming Note

The preferred method of invalidating entire TLB arrays is invalidation using MMUCSR0.

Programming Note

Invalidations using MMUCSR0 only affect the TLB array on the processor that performs the invalidation. To perform invalidations in a multiprocessor system on all processors in a coherence domain, software should use ***tlbivax***.

D.4.4 Searching TLB Entries

Software may search the MMU by using the ***tlbsx*** instruction. The ***tlbsx*** instruction uses PID values and an AS value from the MAS registers instead of the PID registers and the MSR. This allows software to search address spaces that differ from the current address space defined by the PID registers. This is useful for TLB fault handling.

D.4.5 TLB Replacement Hardware Assist

The architecture provides mechanisms to assist software in creating and updating TLB entries when MMU related exceptions occur. This is called TLB Replacement Hardware Assist. Hardware will update the MAS registers on the occurrence of a Data TLB Error Interrupt or Instruction TLB Error interrupt.

When a Data or Instruction TLB Error interrupt (miss) occurs, MAS0, MAS1, and MAS2 are automatically updated using the defaults specified in MAS4 as well as the AS and EPN values corresponding to the access that caused the exception. MAS6 is updated to set $MAS6_{SPID0}$ to the value of PID0 and $MAS6_{SAS}$ to the value of MSR_{DS} or MSR_{IS} depending on the type of access that caused the error. In addition, if $MAS4_{TLBSELD}$ identifies a TLB array that supports NV (Next Victim), $MAS0_{ESEL}$ is loaded with a value that hardware believes represents the best TLB entry to victimize to create a new TLB entry and $MAS0_{NV}$ is updated with the TLB entry index of what hardware believes to be the next victim. Thus $MAS0_{ESEL}$ identifies the current TLB entry to be replaced, and $MAS0_{NV}$ points to the next victim. When software writes the TLB entry, the $MAS0_{NV}$ field is written to the TLB array. The algorithm used by the hardware to determine which TLB entry should be targeted for replacement is implementation-dependent.

The automatic update of the MAS registers sets up all the necessary fields for creating a new TLB entry with the exception of RPN, the U0-U3 attribute bits, and the permission bits. With the exception of the upper 32 bits of RPN and the page attributes (should software desire to specify changes from the default attributes), all the remaining fields are located in MAS3, requiring only the single MAS register manipulation by software before writing the TLB entry.

For Instruction Storage interrupt (ISI) and Data Storage interrupt (DSI) related exceptions, the MAS registers are not updated. Software must explicitly search the TLB to find the appropriate entry.

The update of MAS registers through TLB Replacement Hardware Assist is summarized in Table 7.

D.5 32-bit and 64-bit Specific MMU Behavior

MMU behavior is largely unaffected by whether the processor is in 32-bit computation mode (MSR_{CM}=0) or 64-bit computation mode (MSR_{CM}=1). The only differences occur in the EPN field of the TLB entry and the EPN field of MAS2. The differences are summarized here.

- Executing a ***tlbwe*** instruction in 32-bit mode will set bits 0:31 of the TLB EPN field to 0, regardless of the value of bits 0:31 of the EPN field in MAS2.
- Updates to MAS registers via TLB Replacement Hardware Assist (see Section D.4.5), update bits 0:51 of the EPN field regardless of the computation mode of the processor at the time of the exception or the interrupt computation mode in which the interrupt is taken. If the instruction causing the exception was executing in 32-bit mode, then bits 0:31 of the EPN field in MAS2 will be set to 0.
- Executing a ***tlbre*** instruction in 32-bit mode will set bits 0:31 of the MAS2 EPN field to an undefined value.

Programming Note

This allows a 32-bit OS to operate seamlessly on a 64-bit implementation and a 64-bit OS to easily support 32-bit applications.

D.6 Type FSL MMU Instructions

The instructions described in this section, replace the instructions described in Section 4.9.4.1, "TLB Management Instructions".

TLB Invalidate Virtual Address Indexed X-form

tlbivax RA,RB

31	///	RA	RB	786	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + (RB)
for each processor
   for TLB array = EA59:60
      for each TLB entry
         m ← ¬((1 << (2×(entrySIZE-1))) - 1)
         if ((EA0:51 & m) = (entryEPN & m)) | EA61
         then if entryIPROT = 0
              then entryV ← 0
```

Let the effective address (EA) be the sum(RA|0)+ (RB). The EA is interpreted as show below.

$EA_{0:51}$ $EA_{0:51}$

$EA_{52:58}$ Reserved

$EA_{59:60}$ TLB array selector

00 TLB0
01 TLB1
10 TLB2
11 TLB3

EA_{61} TLB Invalidate All

$EA_{62:63}$ Reserved

If EA_{61}=0, then if the TLB array targeted by $EA_{59:60}$ contains an entry identified by $EA_{0:51}$, that entry is made invalid unless the TLB entry is protected by the IPROT attribute. A TLB entry is identified if, for $m = \neg((1 << (2\times(TLB\_entry_{size}-1))) - 1)$, $EA_{0:51}$&m is equal to TLB_entry_{EPN}&m. The AS bit does not participate in the comparison.

If EA_{61}=1, then all entries not protected by the IPROT attribute in the TLB array targeted by $EA_{59:60}$ are made invalid.

This instruction causes the target TLB entry to be invalidated in all processors.

The operation performed by this instruction is ordered by the ***mbar*** (or ***sync***) instruction with respect to a subsequent ***tlbsync*** instruction executed by the processor executing the ***tlbivax*** instruction. The operations caused by ***tlbivax*** and ***tlbsync*** are ordered by ***mbar*** as a set of operations which is independent of the other sets that ***mbar*** orders.

The effects of the invalidation are not guaranteed to be visible to the programming model until the completion of a context synchronizing operation.

Invalidations may occur for other TLB entries in the designated array, but in no case will any TLB entries with the IPROT attribute set be made invalid.

In some implementations, if RA does not equal 0, it may produce an Illegal Instruction exception.

This instruction is privileged.

Special Registers Altered:
None

Programming Note

The use of EA_{61} to invalidate TLB arrays may be phased out in future versions of the architecture. The preferred method of invalidating TLB arrays is invalidation using MMUCSR0.

TLB Search Indexed *X-form*

tlbsx RA,RB

31	///	RA	RB	914	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + (RB)
pid ← MAS6_SPID0
as ← MAS6_SAS
va ← as || pid || EA
if Valid_matching_entry_exists(va) then
   entry ← matching entry found
   array ← TLB array number where TLB entry found
   index ← index into TLB array of TLB entry found
   if TLB array supports Next Victim then
      hint ← hardware hint for Next Victim
   else
      hint ← undefined
   rpn ← entry_RPN
   MAS0_TLBSEL ← array
   MAS0_ESEL ← index
   MAS0_NV ← hint
   MAS1_V ← 1
   MAS1_IPROT TID TS TSIZE ← entry_IPROT TID TS SIZE
   MAS2_EPN VLE W I M G E ACM ← entry_EPN VLE W I M G E ACM
   MAS3_RPNL ← rpn_32:51
   MAS3_U0:U3 UX SX UW SW UR SR ← entry_U0:U3 UX SX UW SW UR SR
   MAS7_RPNU ← rpn_0:31
else
   MAS0_TLBSEL ← MAS4_TLBSELD
   MAS0_ESEL ← hint
   MAS0_NV ← hint
   MAS1_V IPROT ← 0
   MAS1_TID TS ← MAS6_SPID0 SAS
   MAS1_TSIZE ← MAS4_TSIZED
   MAS2_VLE W I M G E ACM ← MAS4_VLED WD ID MD GD ED ACMD
   MAS2_EPN ← undefined
   MAS3_RPNL ← 0
   MAS3_U0:U3 UX SX UW SW UR SR ← 0
   MAS7_RPNU ← 0
```

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

If any valid TLB array contains an entry corresponding to the virtual address formed by MAS6$_{SAS\ SPID0}$ and EA, that entry as well as the index and array are read into the MAS registers. If no valid matching translation exists, MAS1$_{V}$ is set to 0 and the MAS registers are loaded with defaults to facilitate a TLB replacement.

If the TLB array supports MAS0$_{NV}$, an implementation defined value, hint, specifying the index for the next entry to be replaced is loaded into MAS0$_{NV}$ regardless of whether a match occurs; otherwise MAS0$_{NV}$ is set to an undefined value. It is also loaded into MAS0$_{ESEL}$ if no match occurs.

In some implementations, if RA does not equal 0, it may produce an Illegal Instruction exception.

This instruction is privileged.

Special Registers Altered:
MAS0 MAS1 MAS2 MAS3 MAS7

TLB Read Entry *X-form*

tlbre

31	///	///	///	946	/
0	6	11	16	21	31

```
entry ← SelectTLB(MAS0_TLBSEL, MAS0_ESEL, MAS2_EPN)
rpn ← entry_RPN
if TLB array supports Next Victim then
    MAS0_NV ← hint
else
    MAS0_NV ← undefined
MAS1_V IPROT TID TS TSIZE ← entry_V IPROT TID TS SIZE
MAS2_EPN VLE W I M G E ACM ← entry_EPN VLE W I M G E ACM
MAS3_RPNL ← rpn_32:51
MAS3_U0:U3 UX SX UW SW UR SR ← entry_U0:U3 UX SX UW SW UR SR
MAS7_RPNU ← rpn_0:31
```

The contents of the TLB entry specified by MAS0$_{TLBSEL}$, MAS0$_{ESEL}$, and MAS2$_{EPN}$ are read and placed into the MAS registers.

If the TLB array supports MAS0$_{NV}$, then an implementation defined value, hint, specifying the index for the next entry to be replaced is loaded into MAS0$_{NV}$; otherwise MAS0$_{NV}$ is set to an undefined value.

If the specified entry does not exist, the results are undefined.

This instruction is privileged.

Special Registers Altered:
MAS0 MAS1 MAS2 MAS3 MAS7

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 ***tlbivax*** instructions executed by the processor executing the ***tlbsync*** instruction, with respect to the memory barrier created by a subsequent ***sync*** (***msync***) instruction executed by the same processor. Executing a ***tlbsync*** instruction ensures that all of the following will occur.

- All TLB invalidations caused by ***tlbivax*** instructions preceding the ***tlbsync*** instruction will have completed on any other processor before any storage accesses associated with data accesses caused by instructions following the ***sync*** (***msync***) instruction are performed with respect to that processor.
- All storage accesses by other processors for which the address was translated using the translations being invalidated will have been performed with respect to the processor executing the ***sync*** (***msync***) instruction, to the extent required by the associated Memory Coherence Required attributes, before the ***sync*** (***msync***) instruction's memory barrier is created.

The operation performed by this instruction is ordered by the ***mbar*** or ***sync*** (***msync***) instruction with respect to preceding ***tlbivax*** instructions executed by the processor executing the ***tlbsync*** instruction. The operations caused by ***tlbivax*** and ***tlbsync*** are ordered by ***mbar*** as a set of operations, which is independent of the other sets that ***mbar*** orders.

The ***tlbsync*** instruction may complete before operations caused by ***tlbivax*** instructions preceding the ***tlbsync*** instruction have been performed.

This instruction is privileged.

Special Registers Altered:

None

TLB Write Entry *X-form*

tlbwe

31	///	///	///	978	/
0	6	11	16	21	31

$entry \leftarrow SelectTLB(MAS0_{TLBSEL},\ MAS0_{ESEL},\ MAS2_{EPN})$
$rpn \leftarrow MAS7_{RPNU}\ ||\ MAS3_{RPNL}$
$hint \leftarrow MAS0_{NV}$
$entry_{V\ IPROT\ TID\ TS\ SIZE} \leftarrow MAS1_{V\ IPROT\ TID\ TS\ TSIZE}$
$entry_{EPN\ VLE\ W\ I\ M\ G\ E\ ACM} \leftarrow MAS2_{EPN\ VLE\ W\ I\ M\ G\ E\ ACM}$
$entry_{U0:U3\ UX\ SX\ UW\ SW\ UR\ SR} \leftarrow MAS3_{U0:U3\ UX\ SX\ UW\ SW\ UR\ SR}$
$entry_{RPN} \leftarrow rpn$

The contents of the MAS registers are written to the TLB entry specified by $MAS0_{TLBSEL}$, $MAS0_{ESEL}$, and $MAS2_{EPN}$.

$MAS0_{NV}$ provides a suggestion to hardware of where the next hardware hint for replacement should be given when the next Data or Instruction TLB Error Interrupt, ***tlbsx***, or ***tlbre*** instruction occurs.

If the specified entry does not exist, the results are undefined.

A context synchronizing instruction is required after a ***tlbwe*** instruction to ensure any subsequent instructions that will use the updated TLB values execute in the new context.

This instruction is privileged.

Special Registers Altered:

None

Appendix E. Example Performance Monitor [Category: Embedded.Performance Monitor]

E.1 Overview

This appendix describes an example of a Performance Monitor facility. It defines an architecture suitable for performance monitoring facilities in the Embedded environment. The architecture itself presents only programming model visible features in conjunction with architecturally defined behavioral features. Much of the selection of events is by necessity implementation-dependent and is not described as part of the architecture; however, this document provides guidelines for some features of a performance monitor implementation that should be followed by all implementations.

The example Performance Monitor facility provides the ability to monitor and count predefined events such as processor clocks, misses in the instruction cache or data cache, types of instructions decoded, or mispredicted branches. The count of such events can be used to trigger the Performance Monitor exception. While most of the specific events are not architected, the mechanism of controlling data collection is.

The example Performance Monitor facility can be used to do the following:

- Improve system performance by monitoring software execution and then recoding algorithms for more efficiency. For example, memory hierarchy behavior can be monitored and analyzed to optimize task scheduling or data distribution algorithms.
- Characterize processors in environments not easily characterized by benchmarking.
- Help system developers bring up and debug their systems.

E.2 Programming Model

The example Performance Monitor facility defines a set of Performance Monitor Registers (PMRs) that are used to collect and control performance data collection and an interrupt to allow intervention by software. The PMRs provide various controls and access to collected data. They are categorized as follows:

- Counter registers. These registers are used for data collection. The occurrence of selected events are counted here. These registers are named PMC0..15. User and supervisor level access to these registers is through different PMR numbers allowing different access rights.
- Global controls. This register control global settings of the Performance Monitor facility and affect all counters. This register is named PMGC0. User and supervisor level access to these registers is through different PMR numbers allowing different access rights. In addition, a bit in the MSR (MSR_{PMM}) is defined to enable/disable counting.
- Local controls. These registers control settings that apply only to a particular counter. These registers are named PMLCa0..15 and PMLCb0..15. User and supervisor level access to these registers is through different PMR numbers allowing different access rights. Each set of local control registers (PMLCa*n* and PMLCb*n*) contains controls that apply to the associated same numbered counter register (e.g. PMLCa0 and PMLCb0 contain controls for PMC0 while PMLCa1 and PMLCb1 contain controls for PMC1).

Assembler Note

The counter registers, global controls, and local controls have alias names which cause the assembler to use different PMR numbers. The names PMC0...15, PMGC0, PMLCa0...15, and PMLCb0...15 cause the assembler to use the supervisor level PMR number, and the names UPMC0...15, UPMGC0, UPMLCa0...15, and UPMLCb0...15 cause the assembler to use the user-level PMR number.

A given implementation may implement fewer counter registers (and their associated control registers) than are architected. Architected counter and counter control registers that are not implemented behave the same as unarchitected Performance Monitor Registers.

PMRs are described in Section E.3.

Software uses the global and local controls to select which events are counted in the counter registers, when such events should be counted, and what action should be taken when a counter overflows. Software can use the collected information to determine performance attributes of a given segment of code, a process, or the entire software system. PMRs can be read by software using the ***mfpmr*** instruction and PMRs can be written by using the ***mtpmr*** instruction. Both instructions are described in Section E.4.

Since counters are defined as 32-bit registers, it is possible for the counting of some events to overflow. A Performance Monitor interrupt is provided that can be programmed to occur in the event of a counter overflow. The Performance Monitor interrupt is described in detail in Section E.2.5 and Section E.2.6.

E.2.1 Event Counting

Event counting can be configured in several different ways. This section describes configurability and specific unconditional counting modes.

E.2.2 Processor Context Configurability

Counting can be enabled if conditions in the processor state match a software-specified condition. Because a software task scheduler may switch a processor's execution among multiple processes and because statistics on only a particular process may be of interest, a facility is provided to mark a process. The Performance Monitor mark bit, MSR_{PMM}, is used for this purpose. System software may set this bit to 1 when a marked process is running. This enables statistics to be gathered only during the execution of the marked process. The states of MSR_{PR} and MSR_{PMM} together define a state that the processor (supervisor or user) and the process (marked or unmarked) may be in at any time. If this state matches an individual state specified by the $PMLCan_{FCS}$, $PMLCan_{FCU}$, $PMLCan_{FCM1}$ and $PMLCan_{FCM0}$ fields in PMLCa*n* (the state for which monitoring is enabled), counting is enabled for PMC*n*.

Each event, on an implementation basis, may count regardless of the value of MSR_{PMM}. The counting behavior of each event should be documented in the User's Manual.

The processor states and the settings of the $PMLCan_{FCS}$, $PMLCan_{FCU}$, $PMLCan_{FCM1}$ and $PMLCan_{FCM0}$ fields in PMLCa*n* necessary to enable monitoring of each processor state are shown in Figure 41.

Processor State	FCS	FCU	FCM1	FCM0
Marked	0	0	0	1
Not marked	0	0	1	0
Supervisor	0	1	0	0
User	1	0	0	0
Marked and supervisor	0	1	0	1
Marked and user	1	0	0	1
Not marked and supervisor	0	1	1	0
Not mark and user	1	0	1	0
All	0	0	0	0
None	X	X	1	1
None	1	1	X	X

Figure 41. Processor States and PMLCa*n* Bit Settings

Two unconditional counting modes may be specified:

- Counting is unconditionally enabled regardless of the states of MSR_{PMM} and MSR_{PR}. This can be accomplished by setting $PMLCan_{FCS}$, $PMLCan_{FCU}$, $PMLCan_{FCM1}$, and $PMLCan_{FCM0}$ to 0 for each counter control.
- Counting is unconditionally disabled regardless of the states of MSR_{PMM} and MSR_{PR}. This can be accomplished by setting $PMGC0_{FAC}$ to 1 or by setting $PMLCan_{FC}$ to 1 for each counter control. Alternatively, this can be accomplished by setting $PMLCan_{FCM1}$ to 1 and $PMLCan_{FCM0}$ to 1 for each counter control or by setting $PMLCan_{FCS}$ to 1 and $PMLCan_{FCU}$ to 1 for each counter control.

Programming Note

Events may be counted in a fuzzy manner. That is, events may not be counted precisely due to the nature of an implementation. Users of the Performance Monitor facility should be aware that an event may be counted even if it was precisely filtered, though it should not have been. In general such discrepancies are statistically unimportant and users should not assume that counts are explicitly accurate.

E.2.3 Event Selection

Events to count are determined by placing an implementation defined event value into the $PMLCa0..15_{EVENT}$ field. Which events may be programmed into which counter are implementation specific and should be defined in the User's Manual. In general, most events may be programmed into any of the implementation available counters. Programming a counter with an event that is not supported for that counter gives boundedly undefined results.

Programming Note

Event name and event numbers will differ greatly across implementations and software should not expect that events and event names will be consistent.

E.2.4 Thresholds

Thresholds are values that must be exceeded for an event to be counted. Threshold values are programmed in the PMLCb0..$15_{THRESHOLD}$ field. The events which may be thresholded and the units of each event that may be thresholded are implementation-dependent. Programming a threshold value for an event that is not defined to use a threshold gives boundedly undefined results.

E.2.5 Performance Monitor Exception

A Performance Monitor exception occurs when counter overflow detection is enabled and a counter overflows. More specifically, for each counter register *n*, if $PMGC0_{PMIE}$=1 and PMLCan_{CE}=1 and PMCn_{OV}=1 and MSR_{EE} = 1, a Performance Monitor exception is said to exist. The Performance Monitor exception condition will cause a Performance Monitor interrupt if the exception is the highest priority exception.

The Performance Monitor exception is level sensitive and the exception condition may cease to exist if any of the required conditions fail to be met. Thus it is possible for a counter to overflow and continue counting events until PMCn_{OV} becomes 0 without taking a Performance Monitor interrupt if MSR_{EE} = 0 during the overflow condition. To avoid this, software should program the counters to freeze if an overflow condition is detected (see Section E.3.4).

E.2.6 Performance Monitor Interrupt

A Performance Monitor interrupt occurs when a Performance Monitor exception exists and no higher priority exception exists. When a Performance Monitor interrupt occurs, SRR0 and SRR1 record the current state of the NIA and the MSR, the MSR is set to handle the interrupt, and instruction execution resumes at $IVPR_{0:47}$ || $IVOR35_{48:59}$ || 0b0000.

The Performance Monitor interrupt is precise and asynchronous.

Programming Note

When taking a Performance Monitor interrupt software should clear the overflow condition by reading the counter register and setting the counter register to a non-overflow value since the normal return from the interrupt will set MSR_{EE} back to 1.

E.3 Performance Monitor Registers

E.3.1 Performance Monitor Global Control Register 0

The Performance Monitor Global Control Register 0 (PMGC0) controls all Performance Monitor counters.

PMGC0
32 … 63

Figure 42. [User] Performance Monitor Global Control Register 0

These bits are interpreted as follows:

Bit Description

32 ***Freeze All Counters*** (FAC)
The FAC bit is sticky; that is, once set to 1 it remains set to 1 until it is set to 0 by an ***mtpmr*** instruction.

- 0 The PMCs can be incremented (if enabled by other Performance Monitor control fields).
- 1 The PMCs can not be incremented.

33 ***Performance Monitor Interrupt Enable*** (PMIE)

- 0 Performance Monitor interrupts are disabled.
- 1 Performance Monitor interrupts are enabled and occur when an enabled condition or event occurs. Enabled conditions and events are described in Section E.2.5.

34 ***Freeze Counters on Enabled Condition or Event*** (FCECE)
Enabled conditions and events are described in Section E.2.5.

- 0 The PMCs can be incremented (if enabled by other Performance Monitor control fields).
- 1 The PMCs can be incremented (if enabled by other Performance Monitor control fields) only until an enabled condition or event occurs. When an enabled condition or event occurs, $PMGC0_{FAC}$ is set to 1. It is the user's responsibility to set $PMGC0_{FAC}$ to 0.

35:63 Reserved

The UPMGC0 register is an alias to the PMGC0 register for user mode read only access.

E.3.2 Performance Monitor Local Control A Registers

The Performance Monitor Local Control A Registers 0 through 15 (PMLCa0..15) function as event selectors and give local control for the corresponding numbered Performance Monitor counters. PMLCa works with the corresponding numbered PMLCb register.

PMLCa0..15
32 63

Figure 43. [User] Performance Monitor Local Control A Registers

PMLCa is set to 0 at reset. These bits are interpreted as follows:

Bit Description

32 ***Freeze Counter*** (FC)

- 0 The PMC can be incremented (if enabled by other Performance Monitor control fields).
- 1 The PMC can not be incremented.

33 ***Freeze Counter in Supervisor State*** (FCS)

- 0 The PMC is incremented (if enabled by other Performance Monitor control fields).
- 1 The PMC can not be incremented if MSR_{PR} is 0.

34 ***Freeze Counter in User State*** (FCU)

- 0 The PMC can be incremented (if enabled by other Performance Monitor control fields).
- 1 The PMC can not be incremented if MSR_{PR} is 1.

35 ***Freeze Counter while Mark is Set*** (FCM1)

- 0 The PMC can be incremented (if enabled by other Performance Monitor control fields).
- 1 The PMC can not be incremented if MSR_{PMM} is 1.

36 ***Freeze Counter while Mark is Cleared*** (FCM0)

- 0 The PMC can be incremented (if enabled by other Performance Monitor control fields).
- 1 The PMC can not be incremented if MSR_{PMM} is 0.

37 ***Condition Enable*** (CE)

- 0 Overflow conditions for PMC*n* cannot occur (PMC*n* cannot cause interrupts, cannot freeze counters)
- 1 Overflow conditions occur when the most-significant-bit of PMC*n* is equal to 1.

It is recommended that CE be set to 0 when counter PMC*n* is selected for chaining; see Section E.5.1.

38:40 Reserved

41:47 ***Event Selector*** (EVENT)
Up to 128 events selectable; see Section E.2.3.

48:53 Setting is implementation-dependent.

54:63 Reserved

The UPMLCa0..15 registers are aliases to the PMLCa0..15 registers for user mode read only access.

E.3.3 Performance Monitor Local Control B Registers

The Performance Monitor Local Control B Registers 0 through 15 (PMLCb0..15) specify a threshold value and a multiple to apply to a threshold event selected for the corresponding Performance Monitor counter. Threshold capability is implementation counter dependent. Not all events or all counters of an implementation are guaranteed to support thresholds. PMLCb works with the corresponding numbered PMLCa register.

PMLCb0..15
32 63

Figure 44. [User] Performance Monitor Local Control B Register

PMLCb is set to 0 at reset. These bits are interpreted as follows:

Bit Description

32:52 Reserved

53:55 ***Threshold Multiple*** (THRESHMUL)

- 000 Threshold field is multiplied by 1 (THRESHOLD $\times$ 1)
- 001 Threshold field is multiplied by 2 (THRESHOLD $\times$ 2)
- 010 Threshold field is multiplied by 4 (THRESHOLD $\times$ 4)
- 011 Threshold field is multiplied by 8 (THRESHOLD $\times$ 8)
- 100 Threshold field is multiplied by 16 (THRESHOLD $\times$ 16)
- 101 Threshold field is multiplied by 32 (THRESHOLD $\times$ 32)
- 110 Threshold field is multiplied by 64 (THRESHOLD $\times$ 64)

111 Threshold field is multiplied by 128 (THRESHOLD $\times$ 128)

56:57 Reserved

58:63 ***Threshold*** (THRESHOLD)
Only events that exceed the value THRESHOLD multiplied as described by THRESHMUL are counted. Events to which a threshold value applies are implementation-dependent as are the unit (for example duration in cycles) and the granularity with which the threshold value is interpreted.

Programming Note

By varying the threshold value, software can obtain a profile of the event characteristics subject to thresholding. For example, if PMC1 is configured to count cache misses that last longer than the threshold value, software can measure the distribution of cache miss durations for a given program by monitoring the program repeatedly using a different threshold value each time.

The UPMLCb0..15 registers are aliases to the PMLCb0..15 registers for user mode read only access.

E.3.4 Performance Monitor Counter Registers

The Performance Monitor Counter Registers (PMC0..15) are 32-bit counters that can be programmed to generate interrupt signals when they overflow. Each counter is enabled to count up to 128 events.

PMC0..15
32 ... 63

Figure 45. [User] Performance Monitor Counter Registers

PMCs are set to 0 at reset. These bits are interpreted as follows:

Bit Description

32 ***Overflow*** (OV)

0 Counter has not reached an overflow state.
1 Counter has reached an overflow state.

33:63 ***Counter Value*** (CV)
Indicates the number of occurrences of the specified event.

The minimum value for a counter is 0 (0x0000_0000) and the maximum value is 4,294,967,295 (0xFFFF_FFFF). A counter can increment up to the maximum value and then wraps to the minimum value. A counter enters the overflow state when the high-order bit is set to 1, which normally occurs only when the counter increments from a value below 2,147,483,648 (0x8000_0000) to a value greater than or equal to 2,147,483,648 (0x8000_0000).

Several different actions may occur when an overflow state is reached, depending on the configuration:

- If PMLCan_{CE} is 0, no special actions occur on overflow: the counter continues incrementing, and no exception is signaled.
- If PMLCan_{CE} and $PMGC0_{FCECE}$ are 1, all counters are frozen when PMC*n* overflows.
- If PMLCan_{CE}, $PMGC0_{PMIE}$, and MSR_{EE} are 1, an exception is signalled when PMC*n* reaches overflow. Note that the interrupts are masked by setting MSR_{EE} to 0. An overflow condition may be present while MSR_{EE} is zero, but the interrupt is not taken until MSR_{EE} is set to 1.

If an overflow condition occurs while MSR_{EE} is 0 (the exception is masked), the exception is still signalled once MSR_{EE} is set to 1 if the overflow condition is still present and the configuration has not been changed in the meantime to disable the exception; however, if MSR_{EE} remains 0 until after the counter leaves the overflow state (MSB becomes 0), or if MSR_{EE} remains 0 until after PMLCan_{CE} or $PMGC0_{PMIE}$ are set to 0, the exception does not occur.

Programming Note

Loading a PMC with an overflowed value can cause an immediate exception. For example, if $PMLCan_{CE}$, $PMGC0_{PMIE}$, and MSR_{EE} are all 1, and an ***mtpmr*** loads an overflowed value into a PMCn that previously held a non-overflowed value, then an interrupt will be generated before any event counting has occurred.

The following sequence is generally recommended for setting the counter values and configurations.

1. Set $PMGC0_{FAC}$ to 1 to freeze the counters.
2. Perform a series of ***mtpmr*** operations to initialize counter values and configure the control registers
3. Release the counters by setting $PMGC0_{FAC}$ to 0 with a final ***mtpmr***.

E.4 Performance Monitor Instructions

Move From Performance Monitor Register XFX-form

mfpmr RT,PMRN

31	RT	pmrn	334	/
0	6	11	21	31

```
n ← pmrn5:9 || pmrn0:4
if length(PMR(n)) = 64 then
   RT ← PMR(n)
else
   RT ← 320 || PMR(n)32:63
```

Let PMRN denote a Performance Monitor Register number and PMR the set of Performance Monitor Registers.

The contents of the designated Performance Monitor Register are placed into register RT.

The list of defined Performance Monitor Registers and their privilege class is provided in Figure 46.

Execution of this instruction specifying a defined and privileged Performance Monitor Register when MSR_{PR}=1 will result in a Privileged Instruction exception.

Execution of this instruction specifying an undefined Performance Monitor Register will either result in an Illegal Instruction exception or will produce an undefined value for register RT.

Special Registers Altered:
None

Move To Performance Monitor Register XFX-form

mtpmr PMRN,RS

31	RS	pmrn	462	/
0	6	11	21	31

```
n ← pmrn5:9 || pmrn0:4
if length(PMR(n)) = 64 then
   PMR(n) ← (RS)
else
   PMR(n) ← (RS)32:63
```

Let PMRN denote a Performance Monitor Register number and PMR the set of Performance Monitor Registers.

The contents of the register RS are placed into the designated Performance Monitor Register.

The list of defined Performance Monitor Registers and their privilege class is provided in Figure 46.

Execution of this instruction specifying a defined and privileged Performance Monitor Register when MSR_{PR}=1 will result in a Privileged Instruction exception.

Execution of this instruction specifying an undefined Performance Monitor Register will either result in an Illegal Instruction exception or will perform no operation.

Special Registers Altered:
None

decimal	PMR[1] pmrn$_{5:9}$ pmrn$_{0:4}$	Register Name	Privileged mtpmr	Privileged mfpmr	Cat
0-15	00000 0xxxx	PMC0..15	-	no	E.PM
16-31	00000 1xxxx	PMC0..15	yes	yes	E.PM
128-143	00100 0xxxx	PMLCA0..15	-	no	E.PM
144-159	00100 1xxxx	PMLCA0..15	yes	yes	E.PM
256-271	01000 0xxxx	PMLCB0..15	-	no	E.PM
272-287	01000 1xxxx	PMLCB0..15	yes	yes	E.PM
384	01100 00000	PMGC0	-	no	E.PM
400	01100 10000	PMGC0	yes	yes	E.PM

\- This register is not defined for this instruction.
[1] Note that the order of the two 5-bit halves of the PMR number is reversed.

Figure 46. Embedded.Peformance Monitor PMRs

E.5 Performance Monitor Software Usage Notes

E.5.1 Chaining Counters

An implementation may contain events that are used to "chain" counters together to provide a larger range of event counts. This is accomplished by programming the desired event into one counter and programming another counter with an event that occurs when the first counter transitions from 1 to 0 in the most significant bit.

The counter chaining feature can be used to decrease the processing pollution caused by Performance Monitor interrupts, (things like cache contamination, and pipeline effects), by allowing a higher event count than is possible with a single counter. Chaining two counters together effectively adds 32 bits to a counter register where the first counter's carry-out event acts like a carry-out feeding the second counter. By defining the event of interest to be another PMC's overflow generation, the chained counter increments each time the first counter rolls over to zero. Multiple counters may be chained together.

Because the entire chained value cannot be read in a single instruction, an overflow may occur between counter reads, producing an inaccurate value. A sequence like the following is necessary to read the complete chained value when it spans multiple counters and the counters are not frozen. The example shown is for a two-counter case.

```
loop:
  mfpmr    Rx,pmctr1    #load from upper counter
  mfpmr    Ry,pmctr0    #load from lower counter
  mfpmr    Rz,pmctr1    #load from upper counter
  cmp      cr0,0,Rz,Rx  #see if ‘old’ = ‘new’
  bc       4,2,loop
            #loop if carry occurred between reads
```

The comparison and loop are necessary to ensure that a consistent set of values has been obtained. The above sequence is not necessary if the counters are frozen.

E.5.2 Thresholding

Threshold event measurement enables the counting of duration and usage events. Assume an example event, dLFB load miss cycles, requires a threshold value. A dLFB load miss cycles event is counted only when the number of cycles spent recovering from the miss is greater than the threshold. If the event is counted on two counters and each counter has an individual threshold, one execution of a performance monitor program can sample two different threshold values. Measuring code performance with multiple concurrent thresholds expedites code profiling significantly.

Book VLE:

Power ISA Operating Environment Architecture - Variable Length Encoding (VLE) Environment

Chapter 1. Variable Length Encoding Introduction

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

1.1 Overview

Variable Length Encoding (VLE) is a code density optimized re-encoding of much of the instruction set defined by Books I, II, and III-E using both 16-bit and 32-bit instruction formats.

VLE offers more efficient binary representations of applications for the embedded processor spaces where code density plays a major role in affecting overall system cost, and to a somewhat lesser extent, performance.

VLE is a supplement to the instruction set defined by Book I-III and code pages using VLE encoding or non-VLE encoding can be intermingled in a system providing focus on both high performance and code density where most needed.

VLE provides alternative encodings to instructions defined in Books I-III to enable reduced code footprint. This set of alternative encodings is selected on a page basis. A single storage attribute bit selects between standard instruction encodings and VLE instructions for that page of memory.

Instruction encodings in pages marked as VLE are either 16 or 32 bits long, and are aligned on 16-bit boundaries. Because of this, all instruction pages marked as VLE are required to use Big-Endian byte ordering.

The programming model uses the same register set with both instruction set encodings, although some registers are not accessible by VLE instructions using the 16-bit formats and not all condition register (CR) fields are used by *Conditional Branch* instructions or instructions that access the condition register executing from a VLE instruction page. In addition, immediate fields and displacements differ in size and use, due to the more restrictive encodings imposed by VLE instruction formats.

VLE additional instruction fields are described in Section 1.4.17, “Instruction Fields”.

Other than the requirement of Big-Endian byte ordering for instruction pages and the additional storage attribute to identify whether the instruction page corresponds to a VLE section of code, VLE complies with the memory model, register model, timer facilities, debug facilities, and interrupt/exception model defined in Book I-III and therefore execute in the same environment as non-VLE instructions.

1.2 Documentation Conventions

Book VLE adheres to the documentation conventions defined inSection 1.3 of Book I. Note however that this book defines instructions that apply to the User Instruction Set Architecture, the Virtual Environment Architecture, and the Operating Environment Architecture.

1.2.1 Description of Instruction Operation

The RTL (register transfer language) descriptions in Book VLE conform to the conventions described in Section 1.3.4 of Book I.

1.3 Instruction Mnemonics and Operands

The description of each instruction includes the mnemonic and a formatted list of operands. VLE instruction semantics are either identical or similar to those of other instructions in the architecture. Where the semantics, side-effects, and binary encodings are identical, the standard mnemonics and formats are used. Such unchanged instructions are listed and appropriately referenced, but the instruction definitions are not replicated in this book. Where the semantics are similar but the binary encodings differ, the standard mnemonic is typically preceded with an ***e_*** to denote a VLE instruction. To distinguish between similar instructions available in both 16- and 32-bit forms under VLE and standard instructions, VLE instructions encoded with 16 bits have an ***se_*** prefix. The following are examples:

```
stwx RS,RA,RB     // standard Book I instruction
e_stw RS,D(RA)    // 32-bit VLE instruction
se_stw RZ,SD4(RX) // 16-bit VLE instruction
```

1.4 VLE Instruction Formats

All VLE instructions to be executed are either two or four bytes long and are halfword-aligned in storage. Thus, whenever instruction addresses are presented to the processor (as in *Branch* instructions), the low-order bit is treated as 0. Similarly, whenever the processor generates an instruction address, the low-order bit is zero.

The format diagrams given below show horizontally all valid combinations of instruction fields. Only those formats that are unique to VLE-defined instructions are included here. Instruction forms that are available in VLE or non-VLE mode are described in Section 1.6 of Book I and are not repeated here.

In some cases an instruction field must contain a particular value. If a field that must contain a particular value does not contain that value, the instruction form is invalid and the results are as described for invalid instruction forms in Book I.

VLE instructions use split field notation as defined in Section 1.6 of Book I.

1.4.1 BD8-form (16-bit Branch Instructions)

0	5	6	7	8 15
OPCD	BO16	BI16		BD8
OPCD		XO	LK	BD8

Figure 1. BD8 instruction format

1.4.2 C-form (16-bit Control Instructions)

0	15
OPCD	
OPCD	LK

Figure 2. C instruction format

1.4.3 IM5-form (16-bit register + immediate Instructions)

0	6	7	12 15
OPCD	XO	UI5	RX

Figure 3. IM5 instruction format

1.4.4 OIM5-form (16-bit register + offset immediate Instructions)

0	6	7	12 15
OPCD	XO	OIM5	RX
OPCD	RC	OIM5	RX

Figure 4. OIM5 instruction format

1.4.5 IM7-form (16-bit Load immediate Instructions)

0	5	12 15
OPCD	UI7	RX

Figure 5. IM7 instruction format

1.4.6 R-form (16-bit Monadic Instructions)

0	6	12 15
OPCD	XO	RX

Figure 6. R instruction format

1.4.7 RR-form (16-bit Dyadic Instructions)

0	6	7	8	12 15
OPCD	XO		RY	RX
OPCD	XO	RC	RY	RX
OPCD	XO		ARY	RX
OPCD	XO		RY	ARX

Figure 7. RR instruction format

1.4.8 SD4-form (16-bit Load/Store Instructions)

0	4	8	12 15
OPCD	SD4	RZ	RX

Figure 8. SD4 instruction format

1.4.9 BD15-form

0	10	12	16	31
OPCD	BO32	BI32	BD15	LK

Figure 9. BD15 instruction format

1.4.10 BD24-form

0	6	7	31
OPCD	0	BD24	LK

Figure 10. BD24 instruction format

1.4.11 D8-form

0	6	11	16	24 31
OPCD	RT	RA	XO	D8
OPCD	RS	RA	XO	D8

Figure 11. D8 instruction format

1.4.12 I16A-form

0	6	11	16	21 31
OPCD	si	RA	XO	si
OPCD	ui	RA	XO	ui

Figure 12. I16A instruction format

1.4.13 I16L-form

0	6	11	16	21 31
OPCD	RT	ui	XO	ui

Figure 13. I16L instruction format

1.4.14 M-form

0	6	11	16	21	26	31
OPCD	RS	RA	SH	MB	ME	XO
OPCD	RS	RA	SH	MB	ME	XO

Figure 14. M instruction format

1.4.15 SCI8-form

0	6	9	11	16	20	21	22	24 31
OPCD	RT		RA	XO	Rc	F	SCL	UI8
OPCD	RT		RA	XO		F	SCL	UI8
OPCD	RS		RA	XO	Rc	F	SCL	UI8
OPCD	RS		RA	XO		F	SCL	UI8
OPCD	000	BF32	RA	XO		F	SCL	UI8
OPCD	001	BF32	RA	XO		F	SCL	UI8
OPCD	XO		RA	XO		F	SCL	UI8

Figure 15. SC18 instruction format

1.4.16 LI20-form

0	6	11	16	17	21 31
OPCD	RT	li20	XO	li20	li20

Figure 16. LI20 instruction format

1.4.17 Instruction Fields

VLE uses instruction fields defined in Section 1.6.22 of Book I as well as VLE-defined instruction fields defined below.

ARX (12:15)
: Field used to specify an “alternate” General Purpose Register in the range R8:R23 to be used as a destination.

ARY (8:11)
Field used to specify an “alternate” General Purpose Register in the range R8:R23 to be used as a source.

BD8 (8:15), BD15 (16:30), BD24 (7:30)
Immediate field specifying a signed two's complement branch displacement which is concatenated on the right with 0b0 and sign-extended to 64 bits.

BD15. (Used by 32-bit branch conditional class instructions) A 15-bit signed displacement that is sign-extended and shifted left one bit (concatenated with 0b0) and then added to the current instruction address to form the branch target address.

BD24. (Used by 32-bit branch class instructions) A 24-bit signed displacement that is sign-extended and shifted left one bit (concatenated with 0b0) and then added to the current instruction address to form the branch target address.

BD8. (Used by 16-bit branch and branch conditional class instructions) An 8-bit signed displacement that is sign-extended and shifted left one bit (concatenated with 0b0) and then added to the current instruction address to form the branch target address.

BI16 (6:7), BI32 (12:15)
Field used to specify one of the Condition Register fields to be used as a condition of a Branch Conditional instruction.

BO16 (5), BO32 (10:11)

Field used to specify whether to branch if the condition is true, false, or to decrement the Count Register and branch if the Count Register is not zero in a Branch Conditional instruction.

BF32 (9:10)
Field used to specify one of the Condition Register fields to be used as a target of a compare instruction.

D8 (24:31)
The D8 field is a 8-bit signed displacement which is sign-extended to 64 bits.

F (21) Fill value used to fill the remaining 56 bits of a scaled-immediate 8 value.

LI20 (17:20 || 11:15 || 21:31)
A 20-bit signed immediate value which is sign-extended to 64 bits for the ***e_li*** instruction.

LK (7, 16, 31)
LINK bit.

0 Do not set the Link Register.
1 Set the Link Register. The sum of the value 2 or 4 and the address of the *Branch* instruction is placed into the Link Register.

OIM5 (7:11)
Offset Immediate field used to specify a 5-bit unsigned fixed-point value in the range [1:32] encoded as [0:31]. Thus the binary encoding of 0b00000 represents an immediate value of 1, 0b00001 represents an immediate value of 2, and so on.

OPCD (0:3, 0:4, 0:5, 0:9, 0:14, 0:15)
Primary opcode field.

Rc (6, 7, 20, 31)
RECORD bit.

0 Do not alter the Condition Register.
1 Set Condition Register Field 0.

RX (12:15)
Field used to specify a General Purpose Register in the ranges R0:R7 or R24:R31 to be used as a source or as a destination. R0 is encoded as 0b0000, R1 as 0b0001, etc. R24 is encoded as 0b1000, R25 as 0b1001, etc.

RY (8:11)
Field used to specify a General Purpose Register in the ranges R0:R7 or R24:R31 to be used as a source. R0 is encoded as 0b0000, R1 as 0b0001, etc. R24 is encoded as 0b1000, R25 as 0b1001, etc.

RZ (8:11)
Field used to specify a General Purpose Register in the ranges R0:R7 or R24:R31 to be used as a source or as a destination for load/store data. R0 is encoded as 0b0000, R1 as 0b0001, etc. R24 is encoded as 0b1000, R25 as 0b1001, etc.

SCL (22:23)
Field used to specify a scale amount in *Immediate* instructions using the SCI8-form. Scaling involves left shifting by 0, 8, 16, or 24 bits.

SD4 (4:7)
Used by 16-bit load and store class instructions. The SD4 field is a 4-bit unsigned immediate value zero-extended to 64 bits, shifted left according to the size of the operation, and then added to the base register to form a 64-bit EA. For byte operations, no shift is performed. For half-word operations, the immediate is shifted left one bit (concatenated with 0b0). For word operations, the immediate is shifted left two bits (concatenated with 0b00).SI (6:10 || 21:31, 11:15 || 21:31)
A 16-bit signed immediate value sign-extended to 64 bits and used as one operand of the instruction.

UI (6:10 || 21:31, 11:15 || 21:31)
A 16-bit unsigned immediate value zero-extended to 64 bits or padded with 16 zeros and used as one operand of the instruction. The instruction encoding differs between the I16A and I16L instruction formats as shown in Section 1.4.12 and Section 1.4.13.

UI5 (7:11)
Immediate field used to specify a 5-bit unsigned fixed-point value.

UI7 (5:11)
Immediate field used to specify a 7-bit unsigned fixed-point value.

UI8 (24:31)
Immediate field used to specify an 8-bit unsigned fixed-point value.

XO (6, 6:7, 6:10, 6:11, 16, 16:19,16:23)
Extended opcode field.

Assembler Note

For scaled immediate instructions using the SCI8-form, the instruction assembly syntax requires a single immediate value, *sci8*, that the assembler will synthesize into the appropriate F, SCL, and UI8 fields. The F, SCL, and UI8 fields must be able to be formed correctly from the given *sci8* value or the assembler will flag the assembly instruction as an error.

Chapter 2. VLE Storage Addressing

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

2.1 Data Storage Addressing Modes

Table 1 lists data storage addressing modes supported by the VLE category.

Table 1: Data Storage Addressing Modes

Mode	Form	Description
Base+16-bit displacement (32-bit instruction format)	D-form	The 16-bit D field is sign-extended and added to the contents of the GPR designated by RA or to zero if RA = 0 to produce the EA.
Base+8-bit displacement (32-bit instruction format)	D8-form	The 8-bit D8 field is sign-extended and added to the contents of the GPR designated by RA or to zero if RA = 0 to produce the EA.
Base+scaled 4-bit displacement (16-bit instruction format)	SD4-form	The 4-bit SD4 field zero-extended, scaled (shifted left) according to the size of the operand, and added to the contents of the GPR designated by RX to produce the EA. (Note that RX = 0 is not a special case.)
Base+Index (32-bit instruction format)	X-form	The GPR contents designated by RB are added to the GPR contents designated by RA or to zero if RA = 0 to produce the EA.

2.2 Instruction Storage Addressing Modes

Table 2 lists instruction storage addressing modes supported by the VLE category.

Table 2: Instruction Storage Addressing Modes

Mode	Description
Taken BD24-form Branch instructions (32-bit instruction format)	The 24-bit BD24 field is concatenated on the right with 0b0, sign-extended, and then added to the address of the branch instruction.
Taken B15-form Branch instructions (32-bit instruction format)	The 15-bit BD15 field is concatenated on the right with 0b0, sign-extended, and then added to the address of the branch instruction to form the EA of the next instruction.
Take BD8-form Branch instructions (16-bit instruction format)	The 8-bit BD8 field is concatenated on the right with 0b0, sign-extended, and then added to the address of the branch instruction to form the EA of the next instruction.
Sequential instruction fetching (or non-taken branch instructions)	The value 4 [2] is added to the address of the current 32-bit [16-bit] instruction to form the EA of the next instruction. If the address of the current instruction is 0xFFFF_FFFF_FFFF_FFFC [0xFFFF_FFFF_FFFF_FFFE] in 64-bit mode or 0xFFFF_FFFC [0xFFFF_FFFE] in 32-bit mode, the address of the next sequential instruction is undefined.
Any Branch instruction with LK = 1 (32-bit instruction format)	The value 4 is added to the address of the current branch instruction and the result is placed into the LR. If the address of the current instruction is 0xFFFF_FFFF_FFFF_FFFC in 64-bit mode o r0xFFFF_FFFC in 32-bit mode, the result placed into the LR is undefined.
Branch ***se_bl. se_blrl. se_bctrl*** instructions (16-bit instruction format)	The value 2 is added to the address of the current branch instruction and the result is placed into the LR. If the address of the current instruction is 0xFFFF_FFFF_FFFF_FFFE in 64-bit mode or 0xFFFF_FFFE in 32-bit mode, the result placed into the LR is undefined.

2.2.1 Misaligned, Mismatched, and Byte Ordering Instruction Storage Exceptions

A *Misaligned Instruction Storage Exception* occurs when an implementation which supports VLE attempts to execute an instruction that is not 32-bit aligned and the VLE storage attribute is not set for the page that corresponds to the effective address of the instruction. The attempted execution can be the result of a *Branch* instruction which has bit 62 of the target address set to 1 or the result of an ***rfi***, ***se_rfi***, ***rfci***, ***se_rfci***, ***rfdi***, ***se_rfdi***, ***rfmci***, or ***se_rfmci*** instruction which has bit 62 set in SRR0, SRR0, CSRR0, CSRR0, DSRR0, DSRR0, MCSRR0, or MCSRR0 respectively. If a Misaligned Instruction Storage Exception is detected and no higher priority exception exists, an Instruction Storage Interrupt will occur setting SRR0 to the misaligned address for which execution was attempted.

A *Mismatched Instruction Storage Exception* occurs when an implementation which supports VLE attempts to execute an instruction that crosses a page boundary for which the first page has the VLE storage attribute set to 1 and the second page has the VLE storage attribute bit set to 0. If a Mismatched Instruction Storage Exception is detected and no higher priority exception exists, an Instruction Storage Interrupt will occur setting SRR0 to the misaligned address for which execution was attempted.

A *Byte Ordering Instruction Storage Exception* occurs when an implementation which supports VLE attempts to execute an instruction that has the VLE storage attribute set to 1 and the E (Endian) storage attribute set to 1 for the page that corresponds to the effective address of the instruction. If a Byte Ordering Instruction Storage Exception is detected and no higher priority exception exists, an Instruction Storage Interrupt will occur setting SRR0 to the address for which execution was attempted.

2.2.2 VLE Exception Syndrome Bits

Two bits in the Exception Syndrome Register (ESR) (see Section 5.2.9 of Book III-E) are provided to facilitate VLE exception handling, VLEMI and MIF.

ESR_{VLEMI} is set when an exception and subsequent interrupt is caused by the execution or attempted execution of an instruction that resides in memory with the VLE storage attribute set.

ESR_{MIF} is set when an Instruction Storage Interrupt is caused by a Misaligned Instruction Storage Exception or when an Instruction TLB Error Interrupt was caused by a TLB miss on the second half of a misaligned 32-bit instruction.

ESR_{BO} is set when an Instruction Storage Interrupt is caused by a Mismatched Instruction Storage Exception or a Byte Ordering Instruction Storage Exception.

Programming Note

When an Instruction TLB Error Interrupt occurs as the result of a Instruction TLB miss on the second half of a 32-bit VLE instruction that is aligned to only 16-bits, SRR0 will point to the first half of the instruction and ESR_{MIF} will be set to 1. Any other status posted as a result of the TLB miss (such as MAS register updates described in TYPE-FSL Memory Management) will reflect the page corresponding to the second half of the instruction which caused the Instruction TLB miss.

Chapter 3. VLE Compatibility with Books I–III

This chapter addresses the relationship between VLE and Books I–III.

3.1 Overview

Category VLE uses the same semantics as Books I–III. Due to the limited instruction encoding formats, VLE instructions typically support reduced immediate fields and displacements, and not all operations defined by Books I–III are encoded in category VLE. The basic philosophy is to capture all useful operations, with most frequent operations given priority. Immediate fields and displacements are provided to cover the majority of ranges encountered in embedded control code. Instructions are encoded in either a 16- or 32-bit format, and these may be freely intermixed.

VLE instructions cannot access floating-point registers (FPRs). VLE instructions use GPRs and SPRs with the following limitations:

- VLE instructions using the 16-bit formats are limited to addressing GPR0–GPR7, and GPR24–GPR31 in most instructions. Move instructions are provided to transfer register contents between these registers and GPR8–GPR23.
- VLE compare and bit test instructions using the 16-bit formats implicitly set their results in CR0.

VLE instruction encodings are generally different than instructions defined by Books I–III, except that most instructions falling within primary opcode 31 are encoded identically and have identical semantics unless they affect or access a resource not supported by category VLE.

3.2 VLE Processor and Storage Control Extensions

This section describes additional functionality to support category VLE.

3.2.1 Instruction Extensions

This section describes extensions to support VLE operations. Because instructions may reside on a half-word boundary, bit 62 is not masked by instructions that read an instruction address from a register, such as the LR, CTR, or a save/restore register 0, that holds an instruction address:

The instruction set defined by Books I-III is modified to support halfword instruction addressing, as follows:

- For *Return From Interrupt* instructions, such as ***rfi***, ***rfci***, ***rfdi***, and ***rfmci*** no longer mask bit 62 of the respective save/restore register 0. The destination address is $SRR0_{0:62}$ || 0b0, $CSRR0_{0:62}$ || 0b0, $DSRR0_{0:62}$ || 0b0, $MCSRR0_{0:62}$ || 0b0 respectively.
- For ***bclr***, ***bclrl***, ***bcctr***, and ***bcctrl*** no longer mask bit 62 of the LR or CTR. The destination address is $LR_{0:62}$ || 0b0 or $CTR_{0:62}$ || 0b0.

3.2.2 MMU Extensions

VLE operation is indicated by the VLE storage attribute. When the VLE storage attribute for a page is set to 1, instruction fetches from that page are decoded and processed as VLE instructions. See Section 4.8.3 of Book III-E.

When instructions are executing from a page that has the VLE storage attribute set to 1, the processor is said to be in *VLE mode*.

3.3 VLE Limitations

VLE instruction fetches are valid only when performed in a Big-Endian mode. Attempting to fetch an instruction in a Little-Endian mode from a page with the VLE storage attribute set causes an Instruction Storage Byte-ordering exception.

Support for concurrent modification and execution of VLE instructions is implementation-dependent.

Chapter 4. Branch Operation Instructions

This section defines *Branch* instructions that can be executed when a processor is in VLE mode and the registers that support them.

4.1 Branch Processor Registers

The registers that support branch operations are:

- Section 4.1.1, "Condition Register (CR)"
- Section 4.1.2, "Link Register (LR)"
- Section 4.1.3, "Count Register (CTR)"

4.1.1 Condition Register (CR)

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

Category VLE uses the entire CR, but some comparison operations and all *Branch* instructions are limited to using CR0–CR3. The full Book I condition register field and logical operations are provided however.

CR	
32	63

Figure 17. Condition Register

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

- Specified fields of the condition register can be set by a move to the CR from a GPR (***mtcrf***, ***mtocrf***).
- A specified CR field can be set by a move to the CR from another CR field (***e_mcrf***) or from $XER_{32:35}$ (***mcrxr***).
- CR field 0 can be set as the implicit result of a fixed-point instruction.
- A specified CR field can be set as the result of a fixed-point compare instruction.
- CR field 0 can be set as the result of a fixed-point bit test instruction.

Other instructions from implemented categories may also set bits in the CR in the same manner that they would when not in VLE mode.

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

For all fixed-point instructions in which the Rc bit is defined and set, and for ***e_add2i.***, ***e_and2i.***,and ***e_and2is.,*** the first three bits of CR field 0 ($CR_{32:34}$) are set by signed comparison of the result to zero, and the fourth bit of CR field 0 (CR_{35}) is copied from the final state of XER_{SO}. "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 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, the value placed into the first three bits of CR field 0 is undefined.

The bits of CR field 0 are interpreted as shown below.

CR Bit Description

32 ***Negative*** (LT)
The result is negative.

33 ***Positive*** (GT)
The result is positive.

34 ***Zero*** (EQ)
The result is 0.

35 ***Summary overflow*** (SO)
This is a copy of the contents of XER_{SO} at the completion of the instruction.

4.1.1.1 Condition Register Setting for Compare Instructions

For compare instructions, a CR field specified by the BF operand for the ***e_cmph***, ***e_cmphl***, ***e_cmpi***, and ***e_cmpli*** instructions, or CR0 for the ***se_cmpl***, ***e_cmp16i***, ***e_cmph16i***, ***e_cmphl16i***, ***e_cmpl16i***, ***se_cmp***, ***se_cmph***, ***se_cmphl***, ***se_cmpi***, and ***se_cmpli*** instructions, is set to reflect the result of the comparison. The CR field bits are interpreted as shown below. A complete description of how the bits are set is given in the instruction descriptions and Section 5.6, “Fixed-Point Compare and Bit Test Instructions”.

Condition register bits settings for compare instructions are interpreted as follows. (Note: ***e_cmpi***, and ***e_cmpli*** instructions have a BF32 field instead of BF field; for these instructions, BF32 should be substituted for BF in the list below.)

CR Bit Description

4×BF + 32
Less Than (LT)
For signed fixed-point compare, (RA) or (RX) < sci8, SI, (RB), or (RY).
For unsigned fixed-point compare, (RA) or (RX) $<^{u}$ sci8, UI, UI5, (RB), or (RY).

4×BF + 33
Greater Than (GT)
For signed fixed-point compare, (RA) or (RX) > sci8, SI, (RB), or (RY).
For unsigned fixed-point compare, (RA) or (RX) $>^{u}$ sci8, UI, UI5, (RB), or (RY).

4×BF + 34
Equal (EQ)
For fixed-point compare, (RA) or (RX) = sci8, UI, UI5, SI, (RB), or (RY).

4×BF + 35
Summary Overflow (SO)
For fixed-point compare, this is a copy of the contents of XER_{SO} at the completion of the instruction.

4.1.1.2 Condition Register Setting for the Bit Test Instruction

The Bit Test Immediate instruction, ***se_btsti***, also sets CR field 0. See the instruction description and also Section 5.6, “Fixed-Point Compare and Bit Test Instructions”.

4.1.2 Link Register (LR)

VLE instructions use the Link Register (LR) as defined in Book I, although category VLE defines a subset of all variants of Book I conditional branches involving the LR.

4.1.3 Count Register (CTR)

VLE instructions use the Count Register (CTR) as defined in Book I, although category VLE defines a subset of the variants of Book I conditional branches involving the CTR.

4.2 Branch Instructions

The sequence of instruction execution can be changed by the branch instructions. Because VLE instructions must be aligned on half-word boundaries, the low-order bit of the generated branch target address is forced to 0 by the processor in performing the branch.

The branch instructions compute the EA of the target in one of the following ways, as described in Section 2.2, “Instruction Storage Addressing Modes”

1. Adding a displacement to the address of the branch instruction.
2. Using the address contained in the LR (Branch to Link Register [and Link]).
3. Using the address contained in the CTR (Branch to Count Register [and Link]).

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 EA of the instruction following the branch instruction is placed into the LR after the branch target address has been computed; this is done regardless of whether the branch is taken.

In branch conditional instructions, the BI32 or BI16 instruction field specifies the CR bit to be tested. For 32-bit instructions using BI32, $CR_{32:47}$ (corresponding to bits in CR0:CR3) may be specified. For 16-bit instructions using BI16, only $CR_{32:35}$ (bits within CR0) may be specified.

In branch conditional instructions, the BO32 or BO16 field specifies the conditions under which the branch is taken and how the branch is affected by or affects the CR and CTR. Note that VLE instructions also have different encodings for the BO32 and BO16 fields than in Book I’s BO field.

If the BO32 field specifies that the CTR is to be decremented, in 64-bit mode $CTR_{0:63}$ are decremented, and in 32-bit mode $CTR_{32:63}$ are decremented. If BO16 or BO32 specifies a condition that must be TRUE or FALSE, that condition is obtained from the contents of $CR_{BI32+32}$ or $CR_{BI16+32}$. (Note that CR bits are numbered 32:63. BI32 or BI16 refers to the condition register bit field in the branch instruction encoding. For example, specifying BI32 = 2 refers to CR_{34}.)

For Figure 18 let M = 0 in 64-bit mode and M = 32 in 32-bit mode.

Encodings for the BO32 field for VLE are shown in Figure 18.

BO32	Description
00	Branch if the condition is false.
01	Branch if the condition is true.
10	Decrement $CTR_{M:63}$, then branch if the decremented $CTR_{M:63} \neq 0$
11	Decrement $CTR_{M:63}$, then branch if the decremented $CTR_{M:63}=0$.

Figure 18. BO32 field encodings

Encodings for the BO16 field for VLE are shown in Figure 19.

BO16	Description
0	Branch if the condition is false.
1	Branch if the condition is true.

Figure 19. BO16 field encodings

Branch [and Link] BD24-form

e_b	target_addr	(LK=0)
e_bl	target_addr	(LK=1)

30	0	BD24	LK
0	6	7	31

```
NIA ←iea CIA + EXTS(BD24 || 0b0)
if LK then LR ←iea CIA + 4
```

target_addr specifies the branch target address.

The branch target address is the sum of BD24 || 0b0 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 LK=1 then the effective address of the instruction following the Branch instruction is placed into the Link Register.

Special Registers Altered:

LR (if LK=1)

Branch Conditional [and Link] BD15-form

e_bc	BO32,BI32,target_addr	(LK=0)
e_bcl	BO32,BI32,target_addr	(LK=1)

30	8	BO32	BI32	BD15	LK
0	6	10	12	16	31

```
if (64-bit mode)
  then M ← 0
  else M ← 32
if BO32_0 then CTR_M:63 ← CTR_M:63 - 1
ctr_ok  ← ¬BO32_0 | ((CTR_M:63 ≠ 0) ⊕ BO32_1)
cond_ok ← BO32_0 | (CR_BI32+32 ≡ BO32_1)
if ctr_ok & cond_ok then
  NIA ←iea (CIA + EXTS(BD15 || 0b0))
else
  NIA ←iea CIA + 4
if LK then LR ←iea CIA + 4
```

The BI32 field specifies the Condition Register bit to be tested. The BO32 field is used to resolve the branch as described in Figure 18. *target_addr* specifies the branch target address.

The branch target address is the sum of BD15 || 0b0 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 LK=1 then the effective address of the instruction following the *Branch* instruction is placed into the Link Register.

Special Registers Altered:

CTR (if $BO32_0$=1)

LR (if LK=1)

Branch [and Link] BD8-form

se_b	target_addr	(LK=0)
se_bl	target_addr	(LK=1)

58	0	LK	BD8
0	6	7	8 15

```
NIA ←iea CIA + EXTS(BD8 || 0b0)
if LK then LR ←iea CIA + 2
```

target_addr specifies the branch target address.

The branch target address is the sum of BD8 || 0b0 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 LK=1 then the effective address of the instruction following the Branch instruction is placed into the Link Register.

Special Registers Altered:

LR (if LK=1)

Branch Conditional Short Form BD8-form

se_bc BO16,BI16,target_addr

28	BO16	BI16	BD8
0	5	6	8 15

```
cond_ok ← (CR_BI16+32 ≡ BO16)
if cond_ok then
                NIA ←iea CIA + EXTS(BD8 || 0b0)
else            NIA ←iea CIA + 2
```

The BI16 field specifies the Condition Register bit to be tested. The BO16 field is used to resolve the branch as described in Figure 19. *target_addr* specifies the branch target address.

The branch target address is the sum of BD8 || 0b0 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.

Special Registers Altered:

None

Branch to Count Register [and Link] C-form

se_bctr (LK=0)
se_bctrl (LK=1)

03	LK
0	15

```
NIA ←iea CTR0:62 || 0b0
if LK then LR ←iea CIA + 2
```

The branch target address is $CTR_{0:62}$ || 0b0 with the high-order 32 bits of the branch target address set to 0 in 32-bit mode.

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

Special Registers Altered:

LR (if LK=1)

Branch to Link Register [and Link] C-form

se_blr (LK=0)
se_blrl (LK=1)

02	LK
0	15

```
NIA ←iea LR0:62 || 0b0
if LK then LR ←iea CIA + 2
```

The branch target address is $LR_{0:62}$ || 0b0 with the high-order 32 bits of the branch target address set to 0 in 32-bit mode.

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

Special Registers Altered:

LR (if LK=1)

4.3 System Linkage Instructions

The *System Linkage* instructions enable the program to call upon the system to perform a service and provide a means by which the system can return from performing a service or from processing an interrupt. System Linkage instructions defined by the VLE category are identical in semantics to *System Linkage* instructions defined in Book I and Book III-E with the exception of the LEV field, but are encoded differently.

se_sc provides the same functionality as the Book I (and Book III-E) instruction ***sc*** without the LEV field. ***se_rfi***, ***se_rfci***, ***se_rfdi***, and ***se_rfmci*** provide the same functionality as the Book III-E instructions ***rfi***, ***rfci***, ***rfdi***, and ***rfmci*** respectively.

System Call — *C-form*

se_sc

02
0 … 15

```
SRR1 ←iea MSR
SRR0 ← CIA+2
NIA ←iea IVPR0:47 || IVOR848:59 || 0b0000
MSR ← new_value (see below)
```

The effective address of the instruction following the *System Call* instruction is placed into SRR0. The contents of the MSR are copied into SRR1.

Then a System Call interrupt is generated. The interrupt causes the MSR to be set as described in Section 5.6 of Book III-E.

The interrupt causes the next instruction to be fetched from effective address

$$IVPR_{0:47} \,||\, IVOR8_{48:59} \,||\, 0b0000.$$

This instruction is context synchronizing.

Special Registers Altered:
SRR0 SRR MSR

Illegal — *C-form*

se_illegal

0
0 … 15

se_illegal is used to request an Illegal Instruction exception.

The behavior is the same as if an illegal instruction was executed.

This instruction is context synchronizing.

Special Registers Altered:
SRR0 SRR1 MSR ESR

Return From Machine Check Interrupt C-form

se_rfmci

11
0 … 15

```
MSR ← MCSRR1
NIA ←iea MCSRR0[0:62] || 0b0
```

The ***se_rfmci*** instruction is used to return from a machine check class interrupt, or as a means of establishing a new context and synchronizing on that new context simultaneously.

The contents of MCSRR1 are placed into 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 $MCSRR0_{0:62}$||0b0. 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 values placed into the save/restore registers by the interrupt processing mechanism (see Chapter 5 of Book III-E) is the address and MSR value of the instruction that would have been executed next had the interrupt not occurred (that is, the address in MCSRR0 at the time of the execution of the ***se_rfmci***).

This instruction is privileged and context synchronizing.

Special Registers Altered:
MSR

Return From Critical Interrupt C-form

se_rfci

09
0 … 15

```
MSR ← CSRR1
NIA ←iea CSRR0[0:62] || 0b0
```

The ***se_rfci*** instruction is used to return from a critical class interrupt, or as a means of establishing a new context and synchronizing on that new context simultaneously.

The contents of CSRR1 are placed into 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 $CSRR0_{0:62}$||0b0. 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 values placed into the save/restore registers by the interrupt processing mechanism (see Chapter 5 of Book III-E) is the address and MSR value of the instruction that would have been executed next had the interrupt not occurred (that is, the address in CSRR0 at the time of the execution of the ***se_rfci***).

This instruction is privileged and context synchronizing.

Special Registers Altered:
MSR

Return From Interrupt *C-form*

se_rfi

08
0 15

```
MSR ← SRR1
NIA ←iea SRR0_{0:62} || 0b0
```

The ***se_rfi*** instruction is used to return from a non-critical class interrupt, or as a means of establishing a new context and synchronizing on that new context simultaneously.

The contents of SRR1 are placed into 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 $SRR0_{0:62}$||0b0. 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 values placed into the save/restore registers by the interrupt processing mechanism (see Chapter 5 of Book III-E) is the address and MSR value of the instruction that would have been executed next had the interrupt not occurred (that is, the address in SRR0 at the time of the execution of the ***se_rfi***).

This instruction is privileged and context synchronizing.

Special Registers Altered:
MSR

Return From Debug Interrupt *C-form*

se_rfdi

10
0 15

```
MSR ← DSRR1
NIA ←iea DSRR0_{32:62} || 0b0
```

The ***se_rfdi*** instruction is used to return from a debug class interrupt, or as a means of establishing a new context and synchronizing on that new context simultaneously.

The contents of DSRR1 are placed into 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 $DSRR0_{0:62}$||0b0. 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 the save/restore registers by the interrupt processing mechanism (see Chapter 5 of Book III-E) is the address of the instruction that would have been executed next had the interrupt not occurred (that is, the address in DSRR0 at the time of the execution of ***se_rfdi***).

This instruction is privileged and context synchronizing.

Special Registers Altered:
MSR

Corequisite Categories:
Embedded.Enhanced Debug

4.4 Condition Register Instructions

Condition Register instructions are provided to transfer values to and from various portions of the CR. Category VLE does not introduce any additional functionality beyond that defined in Book I for CR operations, but does remap the CR-logical and ***mcrf*** instruction functionality into primary opcode 31. These instructions operate identically to the Book I instructions, but are encoded differently.

Condition Register AND XL-form

e_crand BT,BA,BB

31	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:

CR_{BT+32}

Condition Register AND with Complement XL-form

e_crandc BT,BA,BB

31	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 one's 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:

CR_{BT+32}

Condition Register Equivalent XL-form

e_creqv BT,BA,BB

31	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:

CR_{BT+32}

Condition Register NAND XL-form

e_crnand BT,BA,BB

31	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:

CR_{BT+32}

Condition Register NOR XL-form

e_crnor BT,BA,BB

31	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:

CR_{BT+32}

Condition Register OR with Complement XL-form

e_crorc BT,BA,BB

31	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:

CR_{BT+32}

Move CR Field XL-form

e_mcrf BF,BFA

31	BF	//	BFA	/////	16	/
0	6	9	11	16	21	31

$CR_{4xBF+32:4xBF+35} \leftarrow CR_{4xBFA+32:4xBFA+35}$

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

Special Registers Altered:

CR field BF

Condition Register OR XL-form

e_cror BT,BA,BB

31	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:

CR_{BT+32}

Condition Register XOR XL-form

e_crxor BT,BA,BB

31	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:

CR_{BT+32}

Chapter 5. Fixed-Point Instructions

This section lists the fixed-point instructions supported by category VLE.

5.1 Fixed-Point Load Instructions

The fixed-point *Load* instructions compute the effective address (EA) of the memory to be accessed as described in Section 2.1, “Data Storage Addressing Modes”

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

Category VLE supports both Big- and Little-Endian byte ordering for data accesses.

Some fixed-point load instructions have an update form in which RA is updated with the EA. For these forms, if RA≠0 and RA≠RT, the EA is placed into RA and the memory element (byte, halfword, word, or doubleword) addressed by EA is loaded into RT. If RA=0 or RA =RT, the instruction form is invalid. This is the same behavior as specified for load with update instructions in Book I.

The fixed-point *Load* instructions from Book I, ***lbzx***, ***lbzux***, ***lhzx***, ***lhzux***, ***lwzx***, and ***lwzux*** are available while executing in VLE mode. The mnemonics, decoding, and semantics for these instructions are identical to those in Book I. See Section 3.3.2 of Book I for the instruction definitions.

The fixed-point *Load* instructions from Book I, ***lwax***, ***lwaux***, ***ldx***, and ***ldux*** are available while executing in VLE mode on 64-bit implementations. The mnemonics, decoding, and semantics for these instructions are identical to those in Book I. See Section 3.3.2 of Book Ifor the instruction definitions.

Load Byte and Zero D-form

e_lbz RT,D(RA)

12	RT	RA	D
0	6	11	16 31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + EXTS(D)
RT ← 560 || MEM(EA, 1)
```

Let the effective address (EA) be the sum (RA|0) + D. 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 Short Form SD4-form

se_lbz RZ,SD4(RX)

08	SD4	RZ	RX
0	4	8	12 15

```
EA ← (RX)+ 600 || SD4
RZ ← 560 || MEM(EA, 1)
```

Let the effective address (EA) be the sum RX + SD4. 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 D8-form

e_lbzu RT,D8(RA)

06	RT	RA	0	D8
0	6	11	16	24 31

```
EA ← (RA) + EXTS(D8)
RT ← 560 || MEM(EA, 1)
RA ← EA
```

Let the effective address (EA) be the sum (RA) + D8. 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 Algebraic D-form

e_lha RT,D(RA)

14	RT	RA	D
0	6	11	16 31

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

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

Special Registers Altered:
None

Load Halfword and Zero D-form

e_lhz RT,D(RA)

22	RT	RA	D
0	6	11	16 31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + EXTS(D)
RT ← 480 || MEM(EA, 2)
```

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

Special Registers Altered:
None

Load Halfword and Zero Short Form SD4-form

se_lhz RZ,SD4(RX)

10	SD4	RZ	RX
0	4	8	12 15

```
EA ← (RX)+ (590 || SD4 || 0)
RZ ← 480 || MEM(EA, 2)
```

Let the effective address (EA) be the sum (RX) + (SD4 || 0). The halfword in storage addressed by EA is loaded into $RZ_{48:63}$. $RZ_{0:47}$ are set to 0.

Special Registers Altered:
None

Load Halfword Algebraic with Update D8-form

e_lhau RT,D8(RA)

06	RT	RA	03	D8
0	6	11	16	24 31

```
EA ← (RA) + EXTS(D8)
RT ← EXTS(MEM(EA, 2))
RA ← EA
```

Let the effective address (EA) be the sum (RA) + D8. 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 RA.

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

Special Registers Altered:
None

Load Halfword and Zero with Update D8-form

e_lhzu RT,D8(RA)

06	RT	RA	01	D8
0	6	11	16	24 31

```
EA ← (RA) + EXTS(D8)
RT ← 480 || MEM(EA, 2))
RA ← EA
```

Let the effective address (EA) be the sum (RA) + D8. 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 Word and Zero D-form

e_lwz RT,D(RA)

20	RT	RA	D
0	6	11	16 31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + EXTS(D)
RT ← 320 || MEM(EA, 4)
```

Let the effective address (EA) be the sum (RA|0) + D. 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 Short FormSD4-form

se_lwz RZ,SD4(RX)

12	SD4	RZ	RX
0	4	8	12 15

```
EA ← (RX)+ (580 || SD4 || 20)
RZ ← 320 || MEM(EA, 2)
```

Let the effective address (EA) be the sum (RX) + (SD4 || 00). The word in storage addressed by EA is loaded into $RZ_{32:63}$. $RZ_{0:31}$ are set to 0.

Special Registers Altered:
None

Load Word and Zero with Update D8-form

e_lwzu RT,D8(RA)

06	RT	RA	02	D8
0	6	11	16	24 31

```
EA ← (RA) + EXTS(D8)
RT ← 320 || MEM(EA, 4))
RA ← EA
```

Let the effective address (EA) be the sum (RA) + D8. 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

5.2 Fixed-Point Store Instructions

The fixed-point *Store* instructions compute the EA of the memory to be accessed as described in Section 2.1, “Data Storage Addressing Modes”.

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

Category VLE supports both Big- and Little-Endian byte ordering for data accesses.

Some fixed-point store instructions have an update form, in which register RA is updated with the effective address. For these forms, the following rules (from Book I) 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 memory element and then EA is placed into register RA (RS).

The fixed-point *Store* instructions from Book I, ***stbx***, ***stbux***, ***sthx***, ***sthux***, ***stwx***, and ***stwux*** are available while executing in VLE mode. The mnemonics, decoding, and semantics for those instructions are identical to those in Book I; see Section 3.3.3 of Book I for the instruction definitions.

The fixed-point *Store* instructions from Book I, ***stdx*** and ***stdux*** are available while executing in VLE mode on 64-bit implementations. The mnemonics, decoding, and semantics for these instructions are identical to those in Book I; see Section 3.3.3 of Book I for the instruction definitions.

Store Byte — *D-form*

e_stb RS,D(RA)

13	RS	RA	D
0	6	11	16 … 31

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

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

Special Registers Altered:
None

Store Byte Short Form — *SD4-form*

se_stb RZ,SD4(RX)

09	SD4	RZ	RX
0	4	8	12 … 15

```
EA ← (RX) + EXTS(SD4)
MEM(EA, 1) ← (RZ)56:63
```

Let the effective address (EA) be the sum (RX) + SD4. $(RZ)_{56:63}$ are stored in the byte in storage addressed by EA.

Special Registers Altered:
None

Store Byte with Update D8-form

e_stbu RS,D8(RA)

06	RS	RA	04	D8
0	6	11	16	24 31

```
EA ← (RA) + EXTS(D8)
MEM(EA, 1) ← (RS)56:63
RA ← EA
```

Let the effective address (EA) be the sum (RA) + D8. $(RS)_{56:63}$ are stored in 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

e_sth RS,D(RA)

23	RS	RA	D
0	6	11	16 31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + EXTS(D)
MEM(EA, 2) ← (RS)48:63
```

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

Special Registers Altered:
None

Store Halfword with Update D8-form

e_sthu RS,D8(RA)

06	RS	RA	05	D8
0	6	11	16	24 31

```
EA ← (RA) + EXTS(D8)
MEM(EA, 2) ← (RS)48:63
RA ← EA
```

Let the effective address (EA) be the sum (RA) + D8. $(RS)_{48:63}$ are stored in 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 Short Form SD4-form

se_sth RZ,SD4(RX)

11	SD4	RZ	RX
0	4	8	12 15

```
EA ← (RX) + (⁵⁹0 || SD4 || 0)
MEM(EA, 2) ← (RZ)48:63
```

Let the effective address (EA) be the sum (RX) + (SD4 || 0). $(RZ)_{48:63}$ are stored in the halfword in storage addressed by EA.

Special Registers Altered:
None

Store Word — *D-form*

e_stw RS,D(RA)

21	RS	RA	D
0	6	11	16 31

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

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

Special Registers Altered:

None

Store Word with Update — *D8-form*

e_stwu RS,D8(RA)

06	RS	RA	06	D8
0	6	11	16	24 31

```
EA ← (RA) + EXTS(D8)
MEM(EA, 4) ← (RS)32:63
RA ← EA
```

Let the effective address (EA) be the sum (RA) + D8. $(RS)_{32:63}$ are stored in 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 Short Form — *SD4-form*

se_stw RZ,SD4(RX)

13	SD4	RZ	RX
0	4	8	12 15

```
EA ← (RX) + (^58 0 || SD4 || ^2 0)
MEM(EA, 4) ← (RZ)32:63
```

Let the effective address (EA) be the sum (RX)+ (SD4 || 00). $(RZ)_{32:63}$ are stored in the word in storage addressed by EA.

Special Registers Altered:

None

5.3 Fixed-Point Load and Store with Byte Reversal Instructions

The fixed-point *Load with Byte Reversal* and *Store with Byte Reversal* instructions from Book I, ***lhbrx***, ***lwbrx***, ***sthbrx***, and ***stwbrx*** are available while executing in VLE mode. The mnemonics, decoding, and semantics for these instructions are identical to those in Book I. See Section 3.3.4 of Book I for the instruction definitions.

5.4 Fixed-Point Load and Store Multiple Instructions

The *Load/Store Multiple* instructions have preferred forms; see Section 1.8.1 of Book I. In the preferred forms storage alignment satisfies the following rule.

- The combination of the EA and RT (RS) is such that the low-order byte of GPR 31 is loaded (stored) from (into) the last byte of an aligned quadword in storage.

Load Multiple Word — *D8-form*

e_lmw RT,D8(RA)

06	RT	RA	08	D8
0	6	11	16	24 31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + EXTS(D8)
r ← RT
do while r ≤31
   GPR(r) ← 320 || MEM(EA,4)
   r  ← r + 1
   EA ← EA + 4
```

Let n = (32-RT). Let the effective address (EA) be the sum (RA|0) + D8.

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.

Special Registers Altered:

None

Store Multiple Word — *D8-form*

e_stmw RS,D8(RA)

06	RS	RA	9	D8
0	6	11	16	24 31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + EXTS(D8)
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) + D8.

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

Special Registers Altered:

None

5.5 Fixed-Point Arithmetic Instructions

The fixed-point *Arithmetic* instructions use the contents of the GPRs as source operands, and place results into GPRs, into status bits in the XER and into CR0.

The fixed-point *Arithmetic* instructions treat source operands as signed integers unless the instruction is explicitly identified as performing an unsigned operation.

The ***e_add2i.*** instruction and other *Arithmetic* instructions with Rc=1 set the first three bits of CR0 to characterize the result placed into the target register. In 64-bit mode, these bits are set by signed comparison of the result to 0. In 32-bit mode, these bits are set by signed comparison of the low-order 32 bits of the result to zero.

e_addic[**.**] and ***e_subfic***[**.**] always set CA to reflect the carry out of bit 0 in 64-bit mode and out of bit 32 in 32-bit mode.

The fixed-point *Arithmetic* instructions from Book I, ***add[.], addo[.]***, ***addc[.]***, ***addco[.]***, ***adde[.]***, ***addeo[.]***, ***addme[.]***, ***addmeo[.]***, ***addze[.]***, ***addzeo[.]***, ***divw[.]***, ***divwo[.]***, ***divwu[.]***, ***divwuo[.]***, ***mulhw[.]***, ***mulhwu[.]***, ***mullw[.]***, ***mullwo[.] neg[.]***, ***nego[.]***, ***subf[.]***, ***subfo[.] subfe[.]***, ***subfeo[.]***, ***subfme[.]***, ***subfmeo[.]***, ***subfze[.]***, ***subfzeo[.]***, ***subfc[.]***, and ***subfco[.]*** are available while executing in VLE mode. The mnemonics, decoding, and semantics for these instructions are identical to those in Book I; see Section 3.3.8 of Book I for the instruction definitions.

The fixed-point *Arithmetic* instructions from Book I, ***mulld[.]***, ***mulldo[.]***, ***mulhd[.]***, ***muldu[.]***, ***divd[.]***, ***divdo[.]***, ***divdu[.]***, and ***divduo[.]*** are available while executing in VLE mode on 64-bit implementations. The mnemonics, decoding, and semantics for those instructions are identical to these in Book I; see Section 3.3.8 of Book I for the instruction definitions.

Add Short Form RR-form

se_add RX,RY

01	0	RY	RX
0	6	8	12 15

```
RX ← (RX) + (RY)
```

The sum (RX) + (RY) is placed into register RX.

Special Registers Altered:
None

Add Immediate D-form

e_add16i RT,RA,SI

07	RT	RA	SI
0	6	11	16 31

```
RT ← (RA) + EXTS(SI)
```

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

Special Registers Altered:
None

Add (2 operand) Immediate and Record I16A-form

e_add2i. RA,si

28	si	RA	17	si
0	6	11	16	21 31

```
RA ← (RA) + EXTS(si)
```

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

Special Registers Altered:
CR0

Add (2 operand) Immediate Shifted I16A-form

e_add2is RA,si

28	si	RA	18	si
0	6	11	16	21 31

RA← (RA) + EXTS(si || ^{16}0)

The sum (RA) + (si || 0x0000) is placed into register RA.

Special Registers Altered:
None

Add Scaled Immediate SCI8-form

e_addi RT,RA,sci8 (Rc=0)
e_addi. RT,RA,sci8 (Rc=1)

06	RT	RA	8	Rc	F	SCL	UI8
0	6	11	16	20	21	22	24 31

sci8 ← $^{56\text{-SCL}\times 8}$F || UI8 ||$^{\text{SCL}\times 8}$F
RT ← (RA) + sci8

The sum (RA) + sci8 is placed into register RT.

Special Registers Altered:
CR0 (if Rc=1)

Add Immediate Short Form OIM5-form

se_addi RX,oimm

08	0	OIM5	RX
0	6	7	12 15

oimm ← (^{59}0 || OIM5) + 1
RX ← (RX) + oimm

The sum (RX) + oimm is placed into RX. The value of oimm must be in the range of 1 to 32.

Special Registers Altered:
None

Add Scaled Immediate Carrying SCI8-form

e_addic RT,RA,sci8 (Rc=0)
e_addic. RT,RA,sci8 (Rc=1)

06	RT	RA	9	Rc	F	SCL	UI8
0	6	11	16	20	21	22	24 31

```
sci8 ← 56-SCL×8F || UI8 ||SCL×8F
RT ← (RA) + sci8
```

The sum (RA) + sci8 is placed into register RT.

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

Subtract RR-form

se_sub RX,RY

1	2	RY	RX
0	6	8	12 15

```
RX ← (RX) + ¬(RY) + 1
```

The sum (RX) + ¬(RY) + 1 is placed into register RX.

Special Registers Altered:
None

Subtract From Scaled Immediate Carrying SCI8-form

e_subfic RT,RA,sci8 (Rc=0)
e_subfic. RT,RA,sci8 (Rc=1)

06	RT	RA	11	Rc	F	SCL	UI8
0	6	11	16	20	21	22	24 31

```
sci8 ← 56-SCL×8F || UI8 ||SCL×8F
RT ← ¬(RA) + sci8 + 1
```

The sum ¬(RA) + sci8 + 1 is placed into register RT.

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

Subtract From Short Form RR-form

se_subf RX,RY

01	3	RY	RX
0	6	8	12 15

```
RX ← ¬(RX) + (RY) + 1
```

The sum ¬(RX) + (RY) + 1 is placed into register RX.

Special Registers Altered:
None

Subtract Immediate OIM5-form

se_subi RX,oimm (Rc=0)
se_subi. RX,oimm (Rc=1)

09	Rc	OIM5	RX
0	6	7	12 15

```
oimm ← (590 || OIM5) + 1
RX ← (RX) + ¬oimm + 1
```

The sum (RA) + ¬oimm + 1 is placed into register RX. The value of oimm must be in the range 1 to 32.

Special Registers Altered:
CR0 (if Rc=1)

Multiply Low Scaled Immediate SCI8-form

e_mulli RT,RA,sci8

06	RT	RA	20	F	SCL	UI8
0	6	11	16	21	22	24 31

```
sci8 ← 56-SCL×8F || UI8 || SCL×8F
prod0:127 ← (RA) × sci8
RT ← prod64:127
```

The 64-bit first operand is (RA). The 64-bit second operand is the sci8 operand. 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 (2 operand) Low Immediate I16A-form

e_mull2i RA,si

28	si	RA	20	si
0	6	11	16	21 31

```
prod0:127 ← (RA) × EXTS(si)
RA ← prod64:127
```

The 64-bit first operand is (RA). The 64-bit second operand is the sign-extended value of the si operand. The low-order 64-bits of the 128-bit product of the operands are placed into register RA.

Both operands and the product are interpreted as signed integers.

Special Registers Altered:
None

Multiply Low Word Short Form RR-form

se_mullw RX,RY

01	1	RY	RX
0	6	8	12 15

```
RX ← (RX)32:63 × (RY)32:63
```

The 32-bit operands are the low-order 32-bits of RX and of RY. The 64-bit product of the operands is placed into register RX.

Both operands and the product are interpreted as signed integers.

Special Registers Altered:
None

Negate Short Form R-form

se_neg RX

0	03	RX
0	6	12 15

```
RX ← ¬(RX)+ 1
```

The sum ¬(RX) + 1 is placed into register RX

If the processor is in 64-bit mode and register RX contains the most negative 64-bit number (0x8000_0000_0000_0000), the result is the most negative 64-bit number. Similarly, if the processor is in 32-bit mode and register RX contains the most negative 32-bit number (0x8000_0000), the result is the most negative 32-bit number.

Special Registers Altered:
None

5.6 Fixed-Point Compare and Bit Test Instructions

The fixed-point *Compare* instructions compare the contents of register RA or register RX with one of the following:

- The value of the scaled immediate field *sci8* formed from the F, UI8, and SCL fields as: $sci8 \leftarrow {}^{56-SCL\times8}F \,||\, UI8 \,||\, {}^{SCL\times8}F$
- The zero-extended value of the UI field
- The zero-extended value of the UI5 field
- The sign-extended value of the SI field
- The contents of register RB or register RY.

The following comparisons are signed: ***e_cmph***, ***e_cmpi***, ***e_cmp16i***, ***e_cmph16i***, ***se_cmp***, ***se_cmph***, and ***se_cmpi***.

The following comparisons are unsigned: ***e_cmphl***, ***e_cmpli***, ***e_cmphl16i***, ***e_cmpl16i***, ***se_cmpli***, ***se_cmpl***, and ***se_cmphl***.

The fixed-point *Bit Test* instruction tests the bit specified by the UI5 instruction field and sets the CR0 field as follows.

.

Bit	Name	Description
0	LT	Always set to 0
1	GT	RX_{ui5} = 1
2	EQ	RX_{ui5} = 0
3	SO	Summary overflow from the XER

The fixed-point *Compare* instructions from Book I, ***cmp*** and ***cmpl*** are available while executing in VLE mode. The mnemonics, decoding, and semantics for these instructions are identical to those in Book I; see Section 3.3.9 of Book I for the instruction definitions.

Bit Test Immediate — *IM5-form*

se_btsti RX,UI5

25	1	UI5	RX
0	6	7	12 15

```
a ← UI5
b ← a+32 0 || 1 || 31-a 0
c ← (RX) & b
if c = 0 then d ← 0b001 else d ← 0b010
CR0 ← d || XER_SO
```

Bit UI5+32 of register RX is tested for equality to '0' and the result is recorded in CR0. EQ is set if the tested bit is 0, LT is cleared, and GT is set to the inverse value of EQ.

Special Registers Altered:

CR0

Compare Immediate Word — *I16A-form*

e_cmp16i RA,si

28	si	RA	19	si
0	6	11	16	21 31

```
b ← EXTS(si)
if (RA)32:63 < b32:63 then c ← 0b100
if (RA)32:63 > b32:63 then c ← 0b010
if (RA)32:63 = b32:63 then c ← 0b001
CR0 ← c || XER_SO
```

The low-order 32 bits of register RA are compared with si, treating operands as signed integers. The result of the comparison is placed into CR0.

Special Registers Altered:

CR0

Compare Scaled Immediate Word SCI8-form

e_cmpi BF32,RA,sci8

06	000	BF32	RA	21	F	SCL	UI8
0	6	9	11	16	21	22	24 31

$$sci8 \leftarrow {}^{56-SCL\times8}F \;||\; UI8 \;||\; {}^{SCL\times8}F$$
$$\text{if } (RA)_{32:63} < sci8_{32:63} \text{ then } c \leftarrow 0b100$$
$$\text{if } (RA)_{32:63} > sci8_{32:63} \text{ then } c \leftarrow 0b010$$
$$\text{if } (RA)_{32:63} = sci8_{32:63} \text{ then } c \leftarrow 0b001$$
$$CR_{4\times BF32+32:4\times BF32+35} \leftarrow c \;||\; XER_{SO}$$

The low-order 32 bits of register RA are compared with sci8, treating operands as signed integers. The result of the comparison is placed into CR field BF32.

Special Registers Altered:

CR field BF32

Compare Immediate Word Short Form IM5-form

se_cmpi RX,UI5

10	1	UI5	RX
0	6	7	12 15

$$b \leftarrow {}^{59}0 \;||\; UI5$$
$$\text{if } (RX)_{32:63} < b_{32:63} \text{ then } c \leftarrow 0b100$$
$$\text{if } (RX)_{32:63} > b_{32:63} \text{ then } c \leftarrow 0b010$$
$$\text{if } (RX)_{32:63} = b_{32:63} \text{ then } c \leftarrow 0b001$$
$$CR0 \leftarrow c \;||\; XER_{SO}$$

The low-order 32 bits of register RX are compared with UI5, treating operands as signed integers. The result of the comparison is placed into CR0.

Special Registers Altered:

CR0

Compare Word RR-form

se_cmp RX,RY

3	0	RY	RX
0	6	8	12 15

$$\text{if } (RX)_{32:63} < (RY)_{32:63} \text{ then } c \leftarrow 0b100$$
$$\text{if } (RX)_{32:63} > (RY)_{32:63} \text{ then } c \leftarrow 0b010$$
$$\text{if } (RX)_{32:63} = (RY)_{32:63} \text{ then } c \leftarrow 0b001$$
$$CR0 \leftarrow c \;||\; XER_{SO}$$

The low-order 32 bits of register RX are compared with the low-order 32 bits of register RY, treating operands as signed integers. The result of the comparison is placed into CR0.

Special Registers Altered:

CR0

Compare Logical Immediate Word I16A-form

e_cmpl16i RA,ui

28	ui	RA	21	ui
0	6	11	16	21 31

$$b \leftarrow {}^{48}0 \;||\; ui$$
$$\text{if } (RA)_{32:63} <^{u} b_{32:63} \text{ then } c \leftarrow 0b100$$
$$\text{if } (RA)_{32:63} >^{u} b_{32:63} \text{ then } c \leftarrow 0b010$$
$$\text{if } (RA)_{32:63} = b_{32:63} \text{ then } c \leftarrow 0b001$$
$$CR0 \leftarrow c \;||\; XER_{SO}$$

The low-order 32 bits of register RA are compared with ui, treating operands as unsigned integers. The result of the comparison is placed into CR0.

Special Registers Altered:

CR0

Compare Logical Scaled Immediate Word SCI8-form

e_cmpli BF32,RA,sci8

06	01	BF32	RA	21	F	SCL	UI8
0	6	9	11	16	21	22	24 31

```
sci8 ← 56-SCL×8F || UI8 ||SCL×8F
if (RA)32:63 <u sci8 32:63 then c ← 0b100
if (RA)32:63 >u sci8 32:63 then c ← 0b010
if (RA)32:63 = sci8 32:63 then c ← 0b001
CR4×BF32+32:4×BF32+35 ← c || XER SO
```

The low-order 32 bits of register RA are compared with sci8, treating operands as unsigned integers. The result of the comparison is placed into CR field BF32.

Special Registers Altered:

CR field BF32

Compare Logical Immediate Word OIM5-form

se_cmpli RX,oimm

08	1	OIM5	RX
0	6	7	12 15

```
oimm ← 590 || (OIM5 + 1)
if (RX)32:63 <u oimm32:63 then c ← 0b100
if (RX)32:63 >u oimm32:63 then c ← 0b010
if (RX)32:63 = oimm32:63 then c ← 0b001
CR0 ← c || XER SO
```

The low-order 32 bits of register RX are compared with oimm, treating operands as unsigned integers. The result of the comparison is placed into CR0. The value of oimm must be in the range of 1 to 32.

Special Registers Altered:

CR0

Compare Logical Word RR-form

se_cmpl RX,RY

3	1	RY	RX
0	6	8	12 15

```
if (RX)32:63 <u (RY)32:63 then c ← 0b100
if (RX)32:63 >u (RY)32:63 then c ← 0b010
if (RX)32:63 = (RY)32:63 then c ← 0b001
CR0 ← c || XER SO
```

The low-order 32 bits of register RX are compared with the low-order 32 bits of register RY, treating operands as unsigned integers. The result of the comparison is placed into CR0.

Special Registers Altered:

CR0

Compare Halfword X-form

e_cmph BF,RA,RB

31	BF	//	RA	RB	14	/
0	6	9	11	16	21	31

```
a ← EXTS((RA)48:63)
b ← EXTS((RB)48:63)
if a < b then c ← 0b100
if a > b then c ← 0b010
if a = b then c ← 0b001
CR4×BF+32:4×BF+35 ← c || XER SO
```

The low-order 16 bits of register RA are compared with the low-order 16 bits of register RB, treating operands as signed integers. The result of the comparison is placed into CR field BF.

Special Registers Altered:

CR field BF

Compare Halfword Short Form RR-form

se_cmph RX,RY

3	2	RY	RX
0	6	8	12 15

a ← EXTS($(RX)_{48:63}$)
b ← EXTS($(RY)_{48:63}$)
if a < b then c ← 0b100
if a > b then c ← 0b010
if a = b then c ← 0b001
CR0 ← c || XER_{SO}

The low-order 16 bits of register RX are compared with the low-order 16 bits of register RY, treating operands as signed integers. The result of the comparison is placed into CR0.

Special Registers Altered:
CR0

Compare Halfword Immediate I16A-form

e_cmph16i RA,si

28	si	RA	22	si
0	6	11	16	21 31

a ← EXTS($(RA)_{48:63}$)
b ← EXTS(si)
if a < b then c ← 0b100
if a > b then c ← 0b010
if a = b then c ← 0b001
CR0 ← c || XER_{SO}

The low-order 16 bits of register RA are compared with si, treating operands as signed integers. The result of the comparison is placed into CR0.

Special Registers Altered:
CR0

Compare Halfword Logical X-form

e_cmphl BF,RA,RB

31	BF	//	RA	RB	46	/
0	6	9	11	16	21	31

a ← EXTZ($(RA)_{48:63}$)
b ← EXTZ($(RB)_{48:63}$)
if a $<^u$ b then c ← 0b100
if a $>^u$ b then c ← 0b010
if a = b then c ← 0b001
$CR_{4\times BF+32:4\times BF+35}$ ← c || XER_{SO}

The low-order 16 bits of register RA are compared with the low-order 16 bits of register RB, treating operands as unsigned integers. The result of the comparison is placed into CR field BF.

Special Registers Altered:
CR field BF

Compare Halfword Logical Short Form RR-form

se_cmphl RX,RY

3	3	RY	RX
0	6	8	12 15

a ← $(RX)_{48:63}$
b ← $(RY)_{48:63}$
if a $<^u$ b then c ← 0b100
if a $>^u$ b then c ← 0b010
if a = b then c ← 0b001
CR0 ← c || XER_{SO}

The low-order 16 bits of register RX are compared with the low-order 16 bits of register RY, treating operands as unsigned integers. The result of the comparison is placed into CR0.

Special Registers Altered:
CR0

Compare Halfword Logical Immediate I16A-form

e_cmphl16i RA,ui

28	ui	RA	23	ui
0	6	11	16	21 31

```
a ← 48 0 || (RA)48:63
b ← 48 0 || ui
if a <u b then c ← 0b100
if a >u b then c ← 0b010
if a = b then c ← 0b001
CR0 ← c || XERSO
```

The low-order 16 bits of register RA are compared with the ui field, treating operands as signed integers. The result of the comparison is placed into CR0.

Special Registers Altered:

CR0

5.7 Fixed-Point Trap Instructions

The fixed-point *Trap* instruction from Book I, ***tw*** is available while executing in VLE mode. The mnemonics, decoding, and semantics for this instruction is identical to that in Book I; see Section 3.3.10 of Book I for the instruction definition.

The fixed-point *Trap* instruction from Book I, ***td*** is available while executing in VLE mode on 64-bit implementations. The mnemonic, decoding, and semantics for the ***td*** instruction are identical to those in Book I; see Section 3.3.10 of Book I for the instruction definitions.

5.8 Fixed-Point Select Instruction

The fixed-point *Select* instruction provides a means to select one of two registers and place the result in a destination register under the control of a predicate value supplied by a CR bit.

The fixed-point *Select* instruction from Book I, ***isel*** is available while executing in VLE mode. The mnemonics, decoding, and semantics for this instruction is identical to that in Book I; see Section of Book I for the instruction definition.

5.9 Fixed-Point Logical, Bit, and Move Instructions

The *Logical* instructions perform bit-parallel operations on 64-bit operands. The *Bit* instructions manipulate a bit, or create a bit mask, in a register. The *Move* instructions move a register or an immediate value into a register.

The X-form Logical instructions with Rc=1, the SCI8-form Logical instructions with Rc=1, the RR-form Logical instructions with Rc=1, the ***e_and2i.*** instruction, and the ***e_and2is.*** instruction set the first three bits of CR field 0 as the arithmetic instructions described in Section 5.5, “Fixed-Point Arithmetic Instructions”. (Also see Section 4.1.1.) The Logical instructions do not change the SO, OV, and CA bits in the XER.

The fixed-point *Logical* instructions from Book I, ***and[.]***, ***or[.]***, ***xor[.]***, ***nand[.]***, ***nor[.]***, ***eqv[.]***, ***andc[.]***, ***orc[.]***, ***extsb[.]***, ***extsh[.]***, ***cntlzw[.]***, and ***popcntb*** are available while executing in VLE mode. The mnemonics, decoding, and semantics for these instructions are identical to those in Book I; see Section 3.3.12 of Book I for the instruction definitions.

The fixed-point *Logical* instructions from Book I, ***extsw[.]*** and ***cntlzd[.]*** are available while executing in VLE mode on 64-bit implementations. The mnemonics, decoding, and semantics for these instructions are identical to those in Book I; see Section 3.3.12 of Book I for the instruction definitions.

AND (two operand) Immediate I16L-form

e_and2i. RT,ui

28	RT	ui	25	ui
0	6	11	16	21 … 31

$RT \leftarrow (RT)\ \&\ (^{48}0\ \|\ ui)$

The contents of register RT are ANDed with $^{48}0\ \|\ ui$ and the result is placed into register RT.

Special Registers Altered:

CR0

AND (2 operand) Immediate Shifted I16L-form

e_and2is. RT,ui

28	RT	ui	29	ui
0	6	11	16	21 … 31

$RT \leftarrow (RT)\ \&\ (^{32}0\ \|\ ui\ \|\ ^{16}0)$

The contents of register RT are ANDed with $^{32}0\ \|\ ui\ \|\ ^{16}0$ and the result is placed into register RT.

Special Registers Altered:

CR0

AND Scaled Immediate Carrying SCI8-form

e_andi	RA,RS,sci8	(Rc=0)
e_andi.	RA,RS,sci8	(Rc=1)

06	RS	RA	12	Rc	F	SCL	UI8
0	6	11	16	20	21	22	24 … 31

$sci8 \leftarrow {}^{56-SCL\times 8}F\ \|\ UI8\ \|\ {}^{SCL\times 8}F$
$RA \leftarrow (RS)\ \&\ sci8$

The contents of register RS are ANDed with sci8 and the result is placed into register RA.

Special Registers Altered:

CR0 (if Rc=1)

AND Immediate Short Form IM5-form

se_andi RX,UI5

11	1	UI5	RX
0	6	7	12 … 15

$RX \leftarrow (RX)\ \&\ {}^{59}0\ \|\ UI5$

The contents of register RX are ANDed with $^{59}0\ \|\ UI5$ and the result is placed into register RX.

Special Registers Altered:

None

OR (two operand) Immediate I16L-form

e_or2i RT,ui

28	RT	ui	24	ui
0	6	11	16	21 31

```
RT ← (RT) | (⁴⁸0 || ui)
```

The contents of register RT are ORed with $^{48}0$ || ui and the result is placed into register RT.

Special Registers Altered:
None

OR (2 operand) Immediate Shifted I16L-form

e_or2is RT,ui

28	RT	ui	26	ui
0	6	11	16	21 31

```
RT ← (RT) | (³²0 || ui || ¹⁶0)
```

The contents of register RT are ORed with $^{32}0$ || ui || $^{16}0$ and the result is placed into register RT.

Special Registers Altered:
None

OR Scaled Immediate SCI8-form

e_ori RA,RS,sci8 (Rc=0)
e_ori. RA,RS,sci8 (Rc=1)

06	RS	RA	13	Rc	F	SCL	UI8
0	6	11	16	20	21	22	24 31

```
sci8 ← 56-SCL×8F || UI8 || SCL×8F
RA ← (RS) | sci8
```

The contents of register RS are ORed with sci8 and the result is placed into register RA.

Special Registers Altered:
CR0 (if Rc=1)

XOR Scaled Immediate SCI8-form

e_xori RA,RS,sci8 (Rc=0)
e_xori. RA,RS,sci8 (Rc=1)

06	RS	RA	14	Rc	F	SCL	UI8
0	6	11	16	20	21	22	24 31

```
sci8 ← 56-SCL×8F || UI8 || SCL×8F
RA ← (RS) ⊕ sci8
```

The contents of register RS are XORed with sci8 and the result is placed into register RA.

Special Registers Altered:
CR0 (if Rc=1)

AND Short Form RR-form

se_and RX,RY (Rc=0)
se_and. RX,RY (Rc=1)

17	1	Rc	RY	RX
0	6	7	8	12 15

```
RX ← (RX) & (RY)
```

The contents of register RX are ANDed with the contents of register RY and the result is placed into register RX.

Special Registers Altered:
CR0 (if Rc=1)

AND with Complement Short Form RR-form

se_andc RX,RY

17	1	RY	RX
0	6	8	12 15

```
RX ← (RX) & ¬(RY)
```

The contents of register RX are ANDed with the complement of the contents of register RY and the result is placed into register RX.

Special Registers Altered:
None

OR Short Form — *RR-form*

se_or RX,RY

17	0	RY	RX
0	6	8	12 15

```
RX ← (RX) | (RY)
```

The contents of register RX are ORed with the contents of register RY and the result is placed into register RX.

Special Registers Altered:
None

NOT Short Form — *R-form*

se_not RX

0	02	RX
0	6	12 15

```
RX ← ¬(RX)
```

The contents of RX are complemented and placed into register RX.

Special Registers Altered:
None

Bit Clear Immediate — *IM5-form*

se_bclri RX,UI5

24	0	UI5	RX
0	6	7	12 15

a ← UI5
RX ← (RX) & ($^{a+32}$1 || 0 || $^{31-a}$1)

Bit UI5+32 of register RX is set to 0.

Special Registers Altered:
None

Bit Generate Immediate — *IM5-form*

se_bgeni RX,UI5

24	1	UI5	RX
0	6	7	12 15

a ← UI5
RX ← ($^{a+32}$0 || 1 || $^{31-a}$0)

Bit UI5+32 of register RX is set to 1. All other bits in register RX are set to 0.

Special Registers Altered:
None

Bit Mask Generate Immediate — *IM5-form*

se_bmaski RX,UI5

11	0	UI5	RX
0	6	7	12 15

a ← UI5
if a = 0 then RX ← ^{64}1
else RX ← $^{64-a}$0 || ^{a}1

If UI5 is not zero, the low-order UI5 bits are set to 1 in register RX and all other bits in register RX are set to 0. If UI5 is 0, all bits in register RX are set to 1.

Special Registers Altered:
None

Bit Set Immediate — *IM5-form*

se_bseti RX,UI5

25	0	UI5	RX
0	6	7	12 15

a ← UI5
RX ← (RX) | ($^{a+32}$0 || 1 || $^{31-a}$0)

Bit UI5+32 of register RX is set to 1.

Special Registers Altered:
None

Extend Sign Byte Short Form — *R-form*

se_extsb RX

0	13	RX
0	6	12 15

```
s ← (RX)56
RX ← 56s || (RX)56:63
```

$(RX)_{56:63}$ are placed into $RX_{56:63}$. Bit 56 of register RX is placed into $RX_{0:55}$.

Special Registers Altered:
None

Extend Sign Halfword Short Form — *R-form*

se_extsh RX

0	15	RX
0	6	12 15

```
s ← (RX)48
RX ← 48s || (RX)48:63
```

$(RX)_{48:63}$ are placed into $RX_{48:63}$. Bit 48 of register RX is placed into $RX_{0:47}$.

Special Registers Altered:
None

Extend Zero Byte — *R-form*

se_extzb RX

0	12	RX
0	6	12 15

```
RX ← 560 || (RX)56:63
```

$(RX)_{56:63}$ are placed into $RX_{56:63}$. $RX_{0:55}$ are set to 0.

Special Registers Altered:
None

Extend Zero Halfword — *R-form*

se_extzh RX

0	14	RX
0	6	12 15

```
RX ← 480 || (RX)48:63
```

$(RX)_{48:63}$ are placed into $RX_{48:63}$. $RX_{0:47}$ are set to 0.

Special Registers Altered:
None

Load Immediate — *LI20-form*

e_li RT,LI20

28	RT	$li20_{4:8}$	0	$li20_{0:3}$	$li20_{9:19}$
0	6	11	16	17	21 31

```
RT ← EXTS(li205:8 || li200:4 || li209:19)
```

The sign-extended LI20 field is placed into RT.

Special Registers Altered:
None

Load Immediate Short Form — *IM7-form*

se_li RX,UI7

09	UI7	RX
0	5	12 15

```
RX ← 570 || UI7
```

The zero-extended UI7 field is placed into RX.

Special Registers Altered:
None

Load Immediate Shifted — *I16L-form*

e_lis RT,ui

28	RT	ui	28	ui
0	6	11	16	21 31

```
RT ← 320 || ui || 160
```

The zero-extended value of ui shifted left 16 bits is placed into RT.

Special Registers Altered:
None

Move from Alternate Register RR-form

se_mfar RX,ARY

0	3	ARY	RX
0	6	8	12 15

```
r ← ARY+8
RX ← GPR(r)
```

The contents of register ARY+8 are placed into RX. ARY specifies a register in the range R8:R23.

Special Registers Altered:
None

Move to Alternate Register RR-form

se_mtar ARX,RY

0	2	RY	ARX
0	6	8	12 15

```
r ← ARX+8
GPR(r) ← (RY)
```

The contents of register RY are placed into register ARX+8. ARX specifies a register in the range R8:R23.

Special Registers Altered:
None

Move Register RR-form

se_mr RX,RY

0	1	RY	RX
0	6	8	12 15

```
RX ← (RY)
```

The contents of register RY are placed into RX.

Special Registers Altered:
None

5.10 Fixed-Point Rotate and Shift Instructions

The fixed-point *Shift* instructions from Book I, ***slw[.]***, ***srw[.]***, ***srawi[.]***, and ***sraw[.]*** are available while executing in VLE mode. The mnemonics, decoding, and semantics for those instructions are identical to those in Book I; see Section 3.3.13.2 of Book I for the instruction definitions.

The fixed-point *Shift* instructions from Book I, ***sld[.]***, ***srd[.]***, ***sradi[.]***, and ***srad[.]*** are available while executing in VLE mode on 64-bit implementations. The mnemonics, decoding, and semantics for those instructions are identical to those in Book I; see Section 3.3.13.2 of Book I for the instruction definitions.

Rotate Left Word *X-form*

e_rlw	RA,RS,RB	(Rc=0)
e_rlw.	RA,RS,RB	(Rc=1)

31	RS	RA	RB	280	Rc
0	6	11	16	21	31

```
n ← (RB)59:63
RA ← ROTL32((RS)32:63,n)
```

The contents of register RS are rotated_{32} left the number of bits specified by $(RB)_{59:63}$ and the result is placed into register RA.

Special Registers Altered:

CR0 (if Rc=1)

Rotate Left Word Immediate *X-form*

e_rlwi	RA,RS,SH	(Rc=0)
e_rlwi.	RA,RS,SH	(Rc=1)

31	RS	RA	SH	312	Rc
0	6	11	16	21	31

```
n ← SH
RA ← ROTL32((RS)32:63,n)
```

The contents of register RS are rotated_{32} left SH bits and the result is placed into register RA.

Special Registers Altered:

CR0 (if Rc=1)

Rotate Left Word Immediate then Mask Insert *M-form*

e_rlwimi RA,RS,SH,MB,ME

29	RS	RA	SH	MB	ME	0
0	6	11	16	21	26	31

```
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 is inserted into register RA under control of the generated mask.

Special Registers Altered:

None

Rotate Left Word Immediate then AND with Mask *M-form*

e_rlwinm RA,RS,SH,MB,ME

29	RS	RA	SH	MB	ME	1
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 is ANDed with the generated mask and the result is placed into register RA.

Special Registers Altered:

None

Shift Left Word Immediate *X-form*

e_slwi RA,RS,SH (Rc=0)
e_slwi. RA,RS,SH (Rc=1)

31	RS	RA	SH	56	Rc
0	6	11	16	21	31

```
n ← SH
r ← ROTL32((RS)32:63, n)
m ← MASK(32, 63-n)
RA ← r & m
```

The contents of the low-order 32 bits of register RS are shifted left SH bits. 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 0.

Special Registers Altered:
CR0 (if Rc=1)

Shift Left Word *RR-form*

se_slw RX,RY

16	2	RY	RX
0	6	8	12 15

```
n ← (RY)58:63
r ← ROTL32((RX)32:63, n)
if (RY)58 = 0 then m ← MASK(32, 63-n)
else               m ← 640
RX ← r & m
```

The contents of the low-order 32 bits of register RX are shifted left the number of bits specified by $(RY)_{58:63}$. 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 $RX_{32:63}$. $RX_{0:31}$ are set to 0. Shift amounts from 32-63 give a zero result.

Special Registers Altered:
None

Shift Left Word Immediate Short Form *IM5-form*

se_slwi RX,UI5

27	0	UI5	RX
0	6	7	12 15

```
n ← UI5
r ← ROTL32((RX)32:63, n)
m ← MASK(32, 63-n)
RX ← r & m
```

The contents of the low-order 32 bits of register RX are shifted left UI5 bits. 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 $RX_{32:63}$. $RX_{0:31}$ are set to 0.

Special Registers Altered:
None

Shift Right Algebraic Word Immediate *IM5-form*

se_srawi RX,UI5

26	1	UI5	RX
0	6	7	12 15

```
n ← UI5
r ← ROTL32((RX)32:63, 64-n)
m ← MASK(n+32, 63)
s ← (RX)32
RX ← r&m | (64s)&¬m
CA ← s & ((r&¬m)32:63≠0)
```

The contents of the low-order 32 bits of register RX are shifted right UI5 bits. Bits shifted out of position 63 are lost, and bit 32 of RX is replicated to fill the vacated positions on the left. Bit 32 of RX is replicated to fill $RX_{0:31}$ and the 32-bit result is placed into $RX_{32:63}$. CA is set to 1 if the low-order 32 bits of register RX contain a negative value and any 1-bits are shifted out of bit position 63; otherwise CA is set to 0. A shift amount of zero causes RX to receive $EXTS((RX)_{32:63})$, and CA to be set to 0.

Special Registers Altered:
CA

Shift Right Algebraic Word *RR-form*

se_sraw RX,RY

16	1	RY	RX
0	6	8	12 15

$n \leftarrow (RY)_{59:63}$
$r \leftarrow ROTL_{32}((RX)_{32:63}, 64-n)$
$\text{if } (RY)_{58} = 0 \text{ then } m \leftarrow MASK(n+32, 63)$
$\text{else } m \leftarrow {}^{64}0$
$s \leftarrow (RX)_{32}$
$RX \leftarrow r\&m \mid ({}^{64}s)\&\neg m$
$CA \leftarrow s \; \& \; ((r\&\neg m)_{32:63}\neq 0)$

The contents of the low-order 32 bits of register RX are shifted right the number of bits specified by $(RY)_{58:63}$. Bits shifted out of position 63 are lost, and bit 32 of RX is replicated to fill the vacated positions on the left. Bit 32 of RX is replicated to fill $RX_{0:31}$ and the 32-bit result is placed into $RX_{32:63}$. CA is set to 1 if the low-order 32 bits of register RX contain a negative value and any 1-bits are shifted out of bit position 63; otherwise CA is set to 0. A shift amount of zero causes RX to receive $EXTS((RX)_{32:63})$, and CA to be set to 0. Shift amounts from 32-63 give a result of 64 sign bits, and cause CA to receive the sign bit of $(RX)_{32:63}$.

Special Registers Altered:
CA

Shift Right Word Immediate Short Form *IM5-form*

se_srwi RX,UI5

26	0	UI5	RX
0	6	7	12 15

$n \leftarrow UI5$
$r \leftarrow ROTL_{32}((RX)_{32:63}, 64-n)$
$m \leftarrow MASK(n+32, 63)$
$RX \leftarrow r \; \& \; m$

The contents of the low-order 32 bits of register RX are shifted right UI5 bits. 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 $RX_{32:63}$. $RX_{0:31}$ are set to 0.

Special Registers Altered:
None

Shift Right Word Immediate *X-form*

e_srwi RA,RS,SH (Rc=0)
e_srwi. RA,RS,SH (Rc=1)

31	RS	RA	SH	568	Rc
0	6	11	16	21	31

$n \leftarrow SH$
$r \leftarrow ROTL_{32}((RS)_{32:63}, 64-n)$
$m \leftarrow MASK(n+32, 63)$
$RA \leftarrow r \; \& \; m$

The contents of the low-order 32 bits of register RS are shifted right SH bits. 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 0.

Special Registers Altered:
CR0 (if Rc=1)

Shift Right Word *RR-form*

se_srw RX,RY

16	0	RY	RX
0	6	8	12 15

$n \leftarrow (RY)_{59:63}$
$r \leftarrow ROTL_{32}((RX)_{32:63}, 64-n)$
$\text{if } (RY)_{58} = 0 \text{ then } m \leftarrow MASK(n+32, 63)$
$\text{else } m \leftarrow {}^{64}0$
$RX \leftarrow r \; \& \; m$

The contents of the low-order 32 bits of register RX are shifted right the number of bits specified by $(RY)_{58:63}$. 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 $RX_{32:63}$. $RX_{0:31}$ are set to 0. Shift amounts from 32 to 63 give a zero result.

Special Registers Altered:
None

5.11 Move To/From System Register Instructions

The VLE category provides 16-bit forms of instructions to move to/from the LR and CTR.

The fixed-point *Move To/From System Register* instructions from Book I, ***mfspr***, ***mtcrf***, ***mfcr***, ***mtocrf***, ***mfocrf***, ***mcrxr***, ***mtdcrux***, ***mfdcrux***, ***mfapidi***, and ***mtspr*** are available while executing in VLE mode. The mnemonics, decoding, and semantics for these instructions are identical to those in Book I; see Section 3.3.14 of Book I for the instruction definitions.

The fixed-point *Move To/From System Register* instructions from Book III-E, ***mfspr***, ***mtspr***, ***mfdcr***, ***mtdcr***, ***mtmsr***, ***mfmsr***, ***wrtee***, and ***wrteei*** are available while executing in VLE mode. The mnemonics, decoding, and semantics for these instructions are identical to those in Book III-E; see Section 3.4.1 of Book III-E for the instruction definitions.

Move From Count Register R-form

se_mfctr RX

0	10	RX
0	6	12 15

RX ← CTR

The CTR contents are placed into register RX.

Special Registers Altered:
None

Move To Count Register R-form

se_mtctr RX

0	11	RX
0	6	12 15

CTR ← (RX)

The contents of register RX are placed into the CTR.

Special Registers Altered:
CTR

Move From Link Register R-form

se_mflr RX

0	8	RX
0	6	12 15

RX ← LR

The LR contents are placed into register RX.

Special Registers Altered:
None

Move To Link Register R-form

se_mtlr RX

0	9	RX
0	6	12 15

LR ← (RX)

The contents of register RX are placed into the LR.

Special Registers Altered:
LR

Chapter 6. Storage Control Instructions

6.1 Storage Synchronization Instructions

The memory synchronization instructions implemented by category VLE are identical in semantics to those defined in Book II and Book III-E. The ***se_isync*** instruction is defined by category VLE, but has the same semantics as ***isync***.

The *Load and Reserve* and *Store Conditional* instructions from Book II, ***lwarx*** and ***stwcx.*** are available while executing in VLE mode. The mnemonics, decoding, and semantics for those instructions are identical to those in Book II; see Section 3.3.2 of Book II for the instruction definitions.

The *Load and Reserve* and *Store Conditional* instructions from Book II, ***ldarx*** and ***stdcx.*** are available while executing in VLE mode on 64-bit implementations. The mnemonics, decoding, and semantics for those instructions are identical to those in Book II; see Section 3.3.2 of Book II for the instruction definitions.

The *Memory Barrier* instructions from Book II, ***sync*** (***msync***) and ***mbar*** are available while executing in VLE mode. The mnemonics, decoding, and semantics for those instructions are identical to those in Book II; see Section 3.3.3 of Book II for the instruction definitions.

The ***wait*** instruction from Book II is available while executing in VLE mode if the category Wait is implemented. The mnemonics, decoding, and semantics for ***wait*** are identical to those in Book II; see Section 3.3 of Book II for the instruction definition.

Instruction Synchronize — *C-form*

se_isync

01
0 … 15

Executing an ***se_isync*** instruction ensures that all instructions preceding the ***se_isync*** instruction have completed before the ***se_isync*** instruction completes, and that no subsequent instructions are initiated until after the ***se_isync*** instruction completes. It also ensures that all instruction cache block invalidations caused by ***icbi*** instructions preceding the ***se_isync*** instruction have been performed with respect to the processor executing the ***se_isync*** instruction, and then causes any prefetched instructions to be discarded.

Except as described in the preceding sentence, the ***se_isync*** instruction may complete before storage accesses associated with instructions preceding the ***se_isync*** instruction have been performed. This instruction is context synchronizing.

The ***se_isync*** instruction has identical semantics to the Book II ***isync*** instruction, but has a different encoding.

Special Registers Altered:
None

6.2 Cache Management Instructions

Cache management instructions implemented by category VLE are identical to those defined in Book II and Book III-E.

The *Cache Management* instructions from Book II, ***dcba***, ***dcbf***, ***dcbst***, ***dcbt***, ***dcbtst***, ***dcbz***, ***icbi***, and ***icbt*** are available while executing in VLE mode. The mnemonics, decoding, and semantics for these instructions are identical to those in Book II; see Section 3.2 of Book II for the instruction definitions.

The *Cache Management* instruction from Book III-E, ***dcbi*** is available while executing in VLE mode. The mnemonics, decoding, and semantics for this instruction are identical to those in Book III-E; see Section 4.9.1 of Book III-E for the instruction definition.

6.3 Cache Locking Instructions

Cache locking instructions implemented by category VLE are identical to those defined in Book III-E. If the *Cache Locking* instructions are implemented in category VLE, the category Embedded Cache Locking must also be implemented.

The *Cache Locking* instructions from Book III-E, ***dcbtls***, ***dcbtstls, dcblc, icbtls***, and ***icblc*** are available while executing in VLE mode. The mnemonics, decoding, and semantics for these instructions are identical to those in Book III-E; see Section 4.9.2 of Book III-E for the instruction definitions.

6.4 TLB Management Instructions

The TLB management instructions implemented by category VLE are identical to those defined in Book III-E.

The *TLB Management* instructions from Book III-E, ***tlbre***, ***tlbwe, tlbivax, tlbsync***, and ***tlbsx*** are available while executing in VLE mode. The mnemonics, decoding, and semantics for these instructions are identical to those in Book III-E. See Section 4.9.4.1 of Book III-E for the instruction definitions.

Instructions and resources from category Embedded.MMU Type FSL are available if the appropriate category is implemented.

6.5 Instruction Alignment and Byte Ordering

Only Big-Endian instruction memory is supported when executing from a page of VLE instructions. Attempting to fetch VLE instructions from a page marked as Little-Endian generates an instruction storage interrupt byte-ordering exception.

Chapter 7. Additional Categories Available in VLE

Instructions and resources from categories other than Base and Embedded are available in VLE. These include categories for which all the instructions in the category use primary opcode 4 or primary opcode 31.

7.1 Move Assist

Move Assist instructions implemented by category VLE are identical to those defined in Book I. If the *Move Assist* instructions are implemented in category VLE, category Move Assist must also be implemented. The mnemonics, decoding, and semantics for those instructions are identical to those in Book I; see Section 3.3.6 of Book I for the instruction definitions.

7.2 Vector

Vector instructions implemented by category VLE are identical to those defined in Book I. If the *Vector* instructions are implemented in category VLE, category Vector must also be implemented. The mnemonics, decoding, and semantics for those instructions are identical to those in Book I; see Chapter 5 of Book I for the instruction definitions.

7.3 Signal Processing Engine

Signal Processing Engine instructions implemented by category VLE are identical to those defined in Book I. If the *Signal Processing Engine* instructions are implemented in category VLE, category Signal Processing Engine must also be implemented. The mnemonics, decoding, and semantics for those instructions are identical to those in Book I; see Chapter 6 of Book Ifor the instruction definitions.

7.4 Embedded Floating Point

Embedded Floating Point instructions implemented by category VLE are identical to those defined in Book I. If the *Embedded Floating Point* instructions are implemented in category VLE, the appropriate category SPE.Embedded Float Scalar Double, SPE.Embedded Float Scalar Single, or SPE.Embedded Float Vector must also be implemented. The mnemonics, decoding, and semantics for those instructions are identical to those in Book I; see Chapter 7 of Book I for the instruction definitions.

7.5 Legacy Move Assist

Legacy Move Assist instructions implemented by category VLE are identical to those defined in Book I. If the *Legacy Move Assist* instructions are implemented in category VLE, category Legacy Move Assist must also be implemented. The mnemonics, decoding, and semantics for those instructions are identical to those in Book I; see Chapter 8 of Book I for the instruction definitions.

7.6 External PID

External Process ID instructions implemented by category VLE are identical to those defined in Book III-E. If the *External Process ID* instructions are implemented in category VLE, category Embedded.External PID must also be implemented. The mnemonics, decoding, and semantics for those instructions are identical to those in Book III-E; see Chapter 3.3.4 of Book III-E for the instruction definitions.

7.7 Embedded Performance Monitor

Embedded Performance Monitor instructions implemented by category VLE are identical to those defined in Book III-E. If the *Embedded Performance Monitor* instructions are implemented in category VLE, category Embedded.Performance Monitor must also be implemented. The mnemonics, decoding, and semantics for those instructions are identical to those in Book III-E; see Appendix E of Book III-E for the instruction definitions.

7.8 Processor Control

Processor Control instructions implemented by category VLE are identical to those defined in Book III-E. If the *Processor Control* instructions are implemented in category VLE, category Embedded.Processor Control must also be implemented. The mnemonics, decoding, and semantics for those instructions are identical to those in Book III-E; see Chapter 9 of Book III-E for the instruction definitions.

Appendix A. VLE Instruction Set Sorted by Mnemonic

This appendix lists all the instructions available in VLE mode in the Power ISA, in order by mnemonic. Opcodes that are not defined below are treated as illegal by category VLE.

Form	Opcode (hexadecimal)[2]	Mode Dep.[1]	Priv[1]	Cat[1]	Mnemonic	Instruction
XO	7C000214			B	add[o][.]	Add
XO	7C000014			B	addc[o][.]	Add Carrying
XO	7C000114	SR		B	adde[o][.]	Add Extended
XO	7C0001D4	SR		B	addme[o][.]	Add to Minus One Extended
XO	7C000194	SR		B	addze[o][.]	Add to Zero Extended
X	7C000038	SR		B	and[.]	AND
X	7C000078	SR		B	andc[.]	AND with Complement
EVX	1000020F			SP	brinc	Bit Reverse Increment
X	7C000000			B	cmp	Compare
X	7C000040			B	cmpl	Compare Logical
X	7C000074	SR		64	cntlzd[.]	Count Leading Zeros Doubleword
X	7C000034	SR		B	cntlzw[.]	Count Leading Zeros Word
X	7C0005EC			E	dcba	Data Cache Block Allocate
X	7C0000AC			B	dcbf	Data Cache Block Flush
X	7C0000FE		P	E.PD	dcbfep	Data Cache Block Flush by External Process ID
X	7C0003AC		P	E	dcbi	Data Cache Block Invalidate
X	7C00030C		M	ECL	dcblc	Data Cache Block Lock Clear
X	7C00006C			B	dcbst	Data Cache Block Store
X	7C00022C			B	dcbt	Data Cache Block Touch
X	7C00027E		P	E.PD	dcbtep	Data Cache Block Touch by External Process ID
X	7C00014C		M	ECL	dcbtls	Data Cache Block Touch and Lock Set
X	7C0001EC			B	dcbtst	Data Cache Block Touch for Store
X	7C0001FE		P	E.PD	dcbtstep	Data Cache Block Touch for Store by External Process ID
X	7C00010C		M	ECL	dcbtstls	Data Cache Block Touch for Store and Lock Set
X	7C0007EC			B	dcbz	Data Cache Block set to Zero
X	7C0007FE		P	E.PD	dcbzep	Data Cache Block set to Zero by External Process ID
X	7C00038C		P	E.CI	dci	Data Cache Invalidate
X	7C00028C		P	E.CD	dcread	Data Cache Read
X	7C0003CC		P	E.CD	dcread	Data Cache Read
XO	7C0003D2	SR		64	divd[o][.]	Divide Doubleword
XO	7C000392	SR		64	divdu[o][.]	Divide Doubleword Unsigned
XO	7C0003D6	SR		B	divw[o][.]	Divide Word
XO	7C000396	SR		B	divwu[o][.]	Divide Word Unsigned
D	1C000000			VLE	e_add16i	Add Immediate
I16A	70008800	SR		VLE	e_add2i.	Add (2 operand) Immediate and Record
I16A	70009000			VLE	e_add2is	Add (2 operand) Immediate Shifted
SCI8	18008000	SR		VLE	e_addi[.]	Add Scaled Immediate
SCI8	18009000	SR		VLE	e_addic[.]	Add Scaled Immediate Carrying
I16L	7000C800	SR		VLE	e_and2i.	AND (2 operand) Immediate
I16L	7000E800	SR		VLE	e_and2is.	AND (2 operand) Immediate Shifted
SCI8	1800C000	SR		VLE	e_andi[.]	AND Scaled Immediate
BD24	78000000			VLE	e_b[l]	Branch [and Link]
BD15	7A000000	CT		VLE	e_bc[l]	Branch Conditional [and Link]
IA16	70009800			VLE	e_cmp16i	Compare Immediate Word
IA16	7000B000			VLE	e_cmph16i	Compare Halfword Immediate

Form	Opcode (hexadecimal)[2]	Mode Dep.[1]	Priv[1]	Cat[1]	Mnemonic	Instruction
X	7C00001C			VLE	e_cmph	Compare Halfword
IA16	7000B800			VLE	e_cmphl16i	Compare Halfword Logical Immediate
X	7C00005C			VLE	e_cmphl	Compare Halfword Logical
SCI8	1800A800			VLE	e_cmpi	Compare Scaled Immediate Word
I16A	7000A800			VLE	e_cmpl16i	Compare Logical Immediate Word
SCI8	1880A800			VLE	e_cmpli	Compare Logical Scaled Immediate Word
XL	7C000202			VLE	e_crand	Condition Register AND
XL	7C000102			VLE	e_crandc	Condition Register AND with Complement
XL	7C000242			VLE	e_creqv	Condition Register Equivalent
XL	7C0001C2			VLE	e_crnand	Condition Register NAND
XL	7C000042			VLE	e_crnor	Condition Register NOR
XL	7C000382			VLE	e_cror	Condition Register OR
XL	7C000342			VLE	e_crorc	Condition Register OR with Complement
XL	7C000182			VLE	e_crxor	Condition Register XOR
D	30000000			VLE	e_lbz	Load Byte and Zero
D8	18000000			VLE	e_lbzu	Load Byte and Zero with Update
D	38000000			VLE	e_lha	Load Halfword Algebraic
D8	18000300			VLE	e_lhau	Load Halfword Algebraic with Update
D	58000000			VLE	e_lhz	Load Halfword and Zero
D8	18000100			VLE	e_lhzu	Load Halfword and Zero with Update
LI20	70000000			VLE	e_li	Load Immediate
I16L	7000E000			VLE	e_lis	Load Immediate Shifted
D8	18000800			VLE	e_lmw	Load Multiple Word
D	50000000			VLE	e_lwz	Load Word and Zero
D8	18000200			VLE	e_lwzu	Load Word and Zero with Update
XL	7C000020			VLE	e_mcrf	Move CR Field
I16A	7000A000			VLE	e_mull2i	Multiply (2 operand) Low Immediate
SCI8	1800A000			VLE	e_mulli	Multiply Low Scaled Immediate
I16L	7000C000			VLE	e_or2i	OR (2operand) Immediate
I16L	7000D000			VLE	e_or2is	OR (2 operand) Immediate Shifted
SCI8	1800D000	SR		VLE	e_ori[.]	OR Scaled Immediate
X	7C000230	SR		VLE	e_rlw[.]	Rotate Left Word
X	7C000270	SR		VLE	e_rlwi[.]	Rotate Left Word Immediate
M	74000000			VLE	e_rlwimi	Rotate Left Word Immediate then Mask Insert
M	74000001			VLE	e_rlwinm	Rotate Left Word Immediate then AND with Mask
X	7C000070	SR		VLE	e_slwi[.]	Shift Left Word Immediate
X	7C000470	SR		VLE	e_srwi[.]	Shift Right Word Immediate
D	34000000			VLE	e_stb	Store Byte
D8	18000400			VLE	e_stbu	Store Byte with Update
D	5C000000			VLE	e_sth	Store Halfword
D8	18000500			VLE	e_sthu	Store Halfword with Update
D8	18000900			VLE	e_stmw	Store Multiple Word
D	54000000			VLE	e_stw	Store Word
D8	18000600			VLE	e_stwu	Store word with Update
SCI8	1800B000	SR		VLE	e_subfic[.]	Subtract From Scaled Immediate Carrying
SCI8	1800E000	SR		VLE	e_xori[.]	XOR Scaled Immediate
EVX	100002E4			SP.FD	efdabs	Floating-Point Double-Precision Absolute Value
EVX	100002E0			SP.FD	efdadd	Floating-Point Double-Precision Add
EVX	100002EF			SP.FD	efdcfs	Floating-Point Double-Precision Convert from Single-Precision
EVX	100002F3			SP.FD	efdcfsf	Convert Floating-Point Double-Precision from Signed Fraction
EVX	100002F1			SP.FD	efdcfsi	Convert Floating-Point Double-Precision from Signed Integer
EVX	100002E3			SP.FD	efdcfsid	Convert Floating-Point Double-Precision from Signed Integer Doubleword
EVX	100002F2			SP.FD	efdcfuf	Convert Floating-Point Double-Precision from Unsigned Fraction
EVX	100002F0			SP.FD	efdcfui	Convert Floating-Point Double-Precision from Unsigned Integer

Form	Opcode (hexadecimal)[2]	Mode Dep.[1]	Priv[1]	Cat[1]	Mnemonic	Instruction
EVX	100002E2			SP.FD	efdcfuid	Convert Floating-Point Double-Precision from Unsigned Integer Doubleword
EVX	100002EE			SP.FD	efdcmpeq	Floating-Point Double-Precision Compare Equal
EVX	100002EC			SP.FD	efdcmpgt	Floating-Point Double-Precision Compare Greater Than
EVX	100002ED			SP.FD	efdcmplt	Floating-Point Double-Precision Compare Less Than
EVX	100002F7			SP.FD	efdctsf	Convert Floating-Point Double-Precision to Signed Fraction
EVX	100002F5			SP.FD	efdctsi	Convert Floating-Point Double-Precision to Signed Integer
EVX	100002EB			SP.FD	efdctsidz	Convert Floating-Point Double-Precision to Signed Integer Doubleword with Round Towards Zero
EVX	100002FA			SP.FD	efdctsiz	Convert Floating-Point Double-Precision to Signed Integer with Round Towards Zero
EVX	100002F6			SP.FD	efdctuf	Convert Floating-Point Double-Precision to Unsigned Fraction
EVX	100002F4			SP.FD	efdctui	Convert Floating-Point Double-Precision to Unsigned Integer
EVX	100002EA			SP.FD	efdctuidz	Convert Floating-Point Double-Precision to Unsigned Integer Doubleword with Round Towards Zero
EVX	100002F8			SP.FD	efdctuiz	Convert Floating-Point Double-Precision to Unsigned Integer with Round Towards Zero
EVX	100002E9			SP.FD	efddiv	Floating-Point Double-Precision Divide
EVX	100002E8			SP.FD	efdmul	Floating-Point Double-Precision Multiply
EVX	100002E5			SP.FD	efdnabs	Floating-Point Double-Precision Negative Absolute Value
EVX	100002E6			SP.FD	efdneg	Floating-Point Double-Precision Negate
EVX	100002E1			SP.FD	efdsub	Floating-Point Double-Precision Subtract
EVX	100002FE			SP.FD	efdtsteq	Floating-Point Double-Precision Test Equal
EVX	100002FC			SP.FD	efdtstgt	Floating-Point Double-Precision Test Greater Than
EVX	100002FD			SP.FD	efdtstlt	Floating-Point Double-Precision Test Less Than
EVX	100002E4			SP.FS	efsabs	Floating-Point Single-Precision Absolute Value
EVX	100002E0			SP.FS	efsadd	Floating-Point Single-Precision Add
EVX	100002CF			SP.FD	efscfd	Floating-Point Single-Precision Convert from Double-Precision
EVX	100002F3			SP.FS	efscfsf	Convert Floating-Point Single-Precision from Signed Fraction
EVX	100002F1			SP.FS	efscfsi	Convert Floating-Point Single-Precision from Signed Integer
EVX	100002E3			SP.FS	efscfsid	Convert Floating-Point Single-Precision from Signed Integer Doubleword
EVX	100002F2			SP.FS	efscfuf	Convert Floating-Point Single-Precision from Unsigned Fraction
EVX	100002F0			SP.FS	efscfui	Convert Floating-Point Single-Precision from Unsigned Integer
EVX	100002E2			SP.FS	efscfuid	Convert Floating-Point Single-Precision from Unsigned Integer Doubleword
EVX	100002EE			SP.FS	efscmpeq	Floating-Point Single-Precision Compare Equal
EVX	100002EC			SP.FS	efscmpgt	Floating-Point Single-Precision Compare Greater Than
EVX	100002ED			SP.FS	efscmplt	Floating-Point Single-Precision Compare Less Than
EVX	100002F7			SP.FS	efsctsf	Convert Floating-Point Single-Precision to Signed Fraction
EVX	100002F5			SP.FS	efsctsi	Convert Floating-Point Single-Precision to Signed Integer
EVX	100002EB			SP.FS	efsctsidz	Convert Floating-Point Single-Precision to Signed Integer Doubleword with Round Towards Zero
EVX	100002FA			SP.FS	efsctsiz	Convert Floating-Point Single-Precision to Signed Integer with Round Towards Zero
EVX	100002F6			SP.FS	efsctuf	Convert Floating-Point Single-Precision to Unsigned Fraction
EVX	100002F4			SP.FS	efsctui	Convert Floating-Point Single-Precision to Unsigned Integer
EVX	100002EA			SP.FS	efsctuidz	Convert Floating-Point Single-Precision to Unsigned Integer Doubleword with Round Towards Zero
EVX	100002F8			SP.FS	efsctuiz	Convert Floating-Point Single-Precision to Unsigned Integer with Round Towards Zero
EVX	100002E9			SP.FS	efsdiv	Floating-Point Single-Precision Divide
EVX	100002E8			SP.FS	efsmul	Floating-Point Single-Precision Multiply

Form	Opcode (hexadeci-mal)[2]	Mode Dep.[1]	Priv[1]	Cat[1]	Mnemonic	Instruction
EVX	100002E5			SP.FS	efsnabs	Floating-Point Single-Precision Negative Absolute Value
EVX	100002E6			SP.FS	efsneg	Floating-Point Single-Precision Negate
EVX	100002E1			SP.FS	efssub	Floating-Point Single-Precision Subtract
EVX	100002FE			SP.FS	efststeq	Floating-Point Single-Precision Test Equal
EVX	100002FC			SP.FS	efststgt	Floating-Point Single-Precision Test Greater Than
EVX	100002FD			SP.FS	efststlt	Floating-Point Single-Precision Test Less Than
X	7C000238	SR		B	eqv[.]	Equivalent
EVX	10000208			SP	evabs	Vector Absolute Value
EVX	10000202			SP	evaddiw	Vector Add Immediate Word
EVX	100004C9			SP	evaddsmiaaw	Vector Add Signed, Modulo, Integer to Accumulator Word
EVX	100004C1			SP	evaddssiaaw	Vector Add Signed, Saturate, Integer to Accumulator Word
EVX	100004C8			SP	evaddumiaaw	Vector Add Unsigned, Modulo, Integer to Accumulator Word
EVX	100004C0			SP	evaddusiaaw	Vector Add Unsigned, Saturate, Integer to Accumulator Word
EVX	10000200			SP	evaddw	Vector Add Word
EVX	10000211			SP	evand	Vector AND
EVX	10000212			SP	evandc	Vector AND with Complement
EVX	10000234			SP	evcmpeq	Vector Compare Equal
EVX	10000231			SP	evcmpgts	Vector Compare Greater Than Signed
EVX	10000230			SP	evcmpgtu	Vector Compare Greater Than Unsigned
EVX	10000233			SP	evcmplts	Vector Compare Less Than Signed
EVX	10000232			SP	evcmpltu	Vector Compare Less Than Unsigned
EVX	1000020E			SP	evcntlsw	Vector Count Leading Sign Bits Word
EVX	1000020D			SP	evcntlzw	Vector Count Leading Zeros Bits Word
EVX	100004C6			SP	evdivws	Vector Divide Word Signed
EVX	100004C7			SP	evdivwu	Vector Divide Word Unsigned
EVX	10000219			SP	eveqv	Vector Equivalent
EVX	1000020A			SP	evextsb	Vector Extend Sign Byte
EVX	1000020B			SP	evextsh	Vector Extend Sign Halfword
EVX	10000284			SP.FV	evfsabs	Vector Floating-Point Single-Precision Absolute Value
EVX	10000280			SP.FV	evfsadd	Vector Floating-Point Single-Precision Add
EVX	10000293			SP.FV	evfscfsf	Vector Convert Floating-Point Single-Precision from Signed Fraction
EVX	10000291			SP.FV	evfscfsi	Vector Convert Floating-Point Single-Precision from Signed Integer
EVX	10000292			SP.FV	evfscfuf	Vector Convert Floating-Point Single-Precision from Unsigned Fraction
EVX	10000290			SP.FV	evfscfui	Vector Convert Floating-Point Single-Precision from Unsigned Integer
EVX	1000028E			SP.FV	evfscmpeq	Vector Floating-Point Single-Precision Compare Equal
EVX	1000028C			SP.FV	evfscmpgt	Vector Floating-Point Single-Precision Compare Greater Than
EVX	1000028D			SP.FV	evfscmplt	Vector Floating-Point Single-Precision Compare Less Than
EVX	10000297			SP.FV	evfsctsf	Vector Convert Floating-Point Single-Precision to Signed Fraction
EVX	10000295			SP.FV	evfsctsi	Vector Convert Floating-Point Single-Precision to Signed Integer
EVX	1000029A			SP.FV	evfsctsiz	Vector Convert Floating-Point Single-Precision to Signed Integer with Round Towards Zero
EVX	10000296			SP.FV	evfsctuf	Vector Convert Floating-Point Single-Precision to Unsigned Fraction
EVX	10000294			SP.FV	evfsctui	Vector Convert Floating-Point Single-Precision to Unsigned Integer
EVX	10000298			SP.FV	evfsctuiz	Vector Convert Floating-Point Single-Precision to Unsigned Integer with Round Towards Zero
EVX	10000289			SP.FV	evfsdiv	Vector Floating-Point Single-Precision Divide
EVX	10000288			SP.FV	evfsmul	Vector Floating-Point Single-Precision Multiply
EVX	10000285			SP.FV	evfsnabs	Vector Floating-Point Single-Precision Negative Absolute Value
EVX	10000286			SP.FV	evfsneg	Vector Floating-Point Single-Precision Negate

Form	Opcode (hexadecimal)[2]	Mode Dep.[1]	Priv[1]	Cat[1]	Mnemonic	Instruction
EVX	10000281			SP.FV	evfssub	Vector Floating-Point Single-Precision Subtract
EVX	1000029E			SP.FV	evfststeq	Vector Floating-Point Single-Precision Test Equal
EVX	1000029C			SP.FV	evfststgt	Vector Floating-Point Single-Precision Test Greater Than
EVX	1000029D			SP.FV	evfststlt	Vector Floating-Point Single-Precision Test Less Than
EVX	10000301			SP	evldd	Vector Load Doubleword into Doubleword
EVX	7C00011D		P	E.PD	evlddepx	Vector Load Doubleword into Doubleword by External Process ID Indexed
EVX	10000300			SP	evlddx	Vector Load Doubleword into Doubleword Indexed
EVX	10000305			SP	evldh	Vector Load Doubleword into 4 Halfwords
EVX	10000304			SP	evldhx	Vector Load Doubleword into 4 Halfwords Indexed
EVX	10000303			SP	evldw	Vector Load Doubleword into 2 Words
EVX	10000302			SP	evldwx	Vector Load Doubleword into 2 Words Indexed
EVX	10000309			SP	evlhhesplat	Vector Load Halfword into Halfwords Even and Splat
EVX	10000308			SP	evlhhesplatx	Vector Load Halfword into Halfwords Even and Splat Indexed
EVX	1000030F			SP	evlhhossplat	Vector Load Halfword into Halfwords Odd and Splat
EVX	1000030E			SP	evlhhossplatx	Vector Load Halfword into Halfwords Odd Signed and Splat Indexed
EVX	1000030D			SP	evlhhousplat	Vector Load Halfword into Halfwords Odd Unsigned and Splat
EVX	1000030C			SP	evlhhousplatx	Vector Load Halfword into Halfwords Odd Unsigned and Splat Indexed
EVX	10000311			SP	evlwhe	Vector Load Word into Two Halfwords Even
EVX	10000310			SP	evlwhex	Vector Load Word into Two Halfwords Even Indexed
EVX	10000317			SP	evlwhos	Vector Load Word into Two Halfwords Odd Signed (with sign extension)
EVX	10000316			SP	evlwhosx	Vector Load Word into Two Halfwords Odd Signed Indexed (with sign extension)
EVX	10000315			SP	evlwhou	Vector Load Word into Two Halfwords Odd Unsigned (zero-extended)
EVX	10000314			SP	evlwhoux	Vector Load Word into Two Halfwords Odd Unsigned Indexed (zero-extended)
EVX	1000031D			SP	evlwhsplat	Vector Load Word into Two Halfwords and Splat
EVX	1000031C			SP	evlwhsplatx	Vector Load Word into Two Halfwords and Splat Indexed
EVX	10000319			SP	evlwwsplat	Vector Load Word into Word and Splat
EVX	10000318			SP	evlwwsplatx	Vector Load Word into Word and Splat Indexed
EVX	1000022C			SP	evmergehi	Vector Merge High
EVX	1000022E			SP	evmergehilo	Vector Merge High/Low
EVX	1000022D			SP	evmergelo	Vector Merge Low
EVX	1000022F			SP	evmergelohi	Vector Merge Low/High
EVX	1000052B			SP	evmhegsmfaa	Vector Multiply Halfwords, Even, Guarded, Signed, Modulo, Fractional and Accumulate
EVX	100005AB			SP	evmhegsmfan	Vector Multiply Halfwords, Even, Guarded, Signed, Modulo, Fractional and Accumulate Negative
EVX	10000529			SP	evmhegsmiaa	Vector Multiply Halfwords, Even, Guarded, Signed, Modulo, Integer and Accumulate
EVX	100005A9			SP	evmhegsmian	Vector Multiply Halfwords, Even, Guarded, Signed, Modulo, Integer and Accumulate Negative
EVX	10000528			SP	evmhegumiaa	Vector Multiply Halfwords, Even, Guarded, Unsigned, Modulo, Integer and Accumulate
EVX	100005A8			SP	evmhegumian	Vector Multiply Halfwords, Even, Guarded, Unsigned, Modulo, Integer and Accumulate Negative
EVX	1000040B			SP	evmhesmf	Vector Multiply Halfwords, Even, Signed, Modulo, Fractional
EVX	1000042B			SP	evmhesmfa	Vector Multiply Halfwords, Even, Signed, Modulo, Fractional to Accumulate
EVX	1000050B			SP	evmhesmfaaw	Vector Multiply Halfwords, Even, Signed, Modulo, Fractional and Accumulate into Words
EVX	1000058B			SP	evmhesmfanw	Vector Multiply Halfwords, Even, Signed, Modulo, Fractional and Accumulate Negative into Words
EVX	10000409			SP	evmhesmi	Vector Multiply Halfwords, Even, Signed, Modulo, Integer

Form	Opcode (hexadecimal)[2]	Mode Dep.[1]	Priv[1]	Cat[1]	Mnemonic	Instruction
EVX	10000429			SP	evmhesmia	Vector Multiply Halfwords, Even, Signed, Modulo, Integer to Accumulator
EVX	10000509			SP	evmhesmiaaw	Vector Multiply Halfwords, Even, Signed, Modulo, Integer and Accumulate into Words
EVX	10000589			SP	evmhesmianw	Vector Multiply Halfwords, Even, Signed, Modulo, Integer and Accumulate Negative into Words
EVX	10000403			SP	evmhessf	Vector Multiply Halfwords, Even, Signed, Saturate, Fractional
EVX	10000423			SP	evmhessfa	Vector Multiply Halfwords, Even, Signed, Saturate, Fractional to Accumulator
EVX	10000503			SP	evmhessfaaw	Vector Multiply Halfwords, Even, Signed, Saturate, Fractional and Accumulate into Words
EVX	10000583			SP	evmhessfanw	Vector Multiply Halfwords, Even, Signed, Saturate, Fractional and Accumulate Negative into Words
EVX	10000501			SP	evmhessiaaw	Vector Multiply Halfwords, Even, Signed, Saturate, Integer and Accumulate into Words
EVX	10000581			SP	evmhessianw	Vector Multiply Halfwords, Even, Signed, Saturate, Integer and Accumulate Negative into Words
EVX	10000408			SP	evmheumi	Vector Multiply Halfwords, Even, Unsigned, Modulo, Integer
EVX	10000428			SP	evmheumia	Vector Multiply Halfwords, Even, Unsigned, Modulo, Integer to Accumulator
EVX	10000508			SP	evmheumiaaw	Vector Multiply Halfwords, Even, Unsigned, Modulo, Integer and Accumulate into Words
EVX	10000588			SP	evmheumianw	Vector Multiply Halfwords, Even, Unsigned, Modulo, Integer and Accumulate Negative into Words
EVX	10000500			SP	evmheusiaaw	Vector Multiply Halfwords, Even, Unsigned, Saturate Integer and Accumulate into Words
EVX	10000580			SP	evmheusianw	Vector Multiply Halfwords, Even, Unsigned, Saturate Integer and Accumulate Negative into Words
EVX	1000052F			SP	evmhogsmfaa	Vector Multiply Halfwords, Odd, Guarded, Signed, Modulo, Fractional and Accumulate
EVX	100005AF			SP	evmhogsmfan	Vector Multiply Halfwords, Odd, Guarded, Signed, Modulo, Fractional and Accumulate Negative
EVX	1000052D			SP	evmhogsmiaa	Vector Multiply Halfwords, Odd, Guarded, Signed, Modulo, Integer and Accumulate
EVX	100005AD			SP	evmhogsmian	Vector Multiply Halfwords, Odd, Guarded, Signed, Modulo, Integer and Accumulate Negative
EVX	1000052C			SP	evmhogumiaa	Vector Multiply Halfwords, Odd, Guarded, Unsigned, Modulo, Integer and Accumulate
EVX	100005AC			SP	evmhogumian	Vector Multiply Halfwords, Odd, Guarded, Unsigned, Modulo, Integer and Accumulate Negative
EVX	1000040F			SP	evmhosmf	Vector Multiply Halfwords, Odd, Signed, Modulo, Fractional
EVX	1000042F			SP	evmhosmfa	Vector Multiply Halfwords, Odd, Signed, Modulo, Fractional to Accumulator
EVX	1000050F			SP	evmhosmfaaw	Vector Multiply Halfwords, Odd, Signed, Modulo, Fractional and Accumulate into Words
EVX	1000058F			SP	evmhosmfanw	Vector Multiply Halfwords, Odd, Signed, Modulo, Fractional and Accumulate Negative into Words
EVX	1000040D			SP	evmhosmi	Vector Multiply Halfwords, Odd, Signed, Modulo, Integer
EVX	1000042D			SP	evmhosmia	Vector Multiply Halfwords, Odd, Signed, Modulo, Integer to Accumulator
EVX	1000050D			SP	evmhosmiaaw	Vector Multiply Halfwords, Odd, Signed, Modulo, Integer and Accumulate into Words
EVX	1000058D			SP	evmhosmianw	Vector Multiply Halfwords, Odd, Signed, Modulo, Integer and Accumulate Negative into Words
EVX	10000407			SP	evmhossf	Vector Multiply Halfwords, Odd, Signed, Fractional
EVX	10000427			SP	evmhossfa	Vector Multiply Halfwords, Odd, Signed, Fractional to Accumulator

Form	Opcode (hexadecimal)[2]	Mode Dep.[1]	Priv[1]	Cat[1]	Mnemonic	Instruction
EVX	10000507			SP	evmhossfaaw	Vector Multiply Halfwords, Odd, Signed, Saturate, Fractional and Accumulate into Words
EVX	10000587			SP	evmhossfanw	Vector Multiply Halfwords, Odd, Signed, Saturate, Fractional and Accumulate Negative into Words
EVX	10000505			SP	evmhossiaaw	Vector Multiply Halfwords, Odd, Signed, Saturate, Integer and Accumulate into Words
EVX	10000585			SP	evmhossianw	Vector Multiply Halfwords, Odd, Signed, Saturate, Integer and Accumulate Negative into Words
EVX	1000040C			SP	evmhoumi	Vector Multiply Halfwords, Odd, Unsigned, Modulo, Integer
EVX	1000042C			SP	evmhoumia	Vector Multiply Halfwords, Odd, Unsigned, Modulo, Integer to Accumulator
EVX	1000050C			SP	evmhoumiaaw	Vector Multiply Halfwords, Odd, Unsigned, Modulo, Integer and Accumulate into Words
EVX	1000058C			SP	evmhoumianw	Vector Multiply Halfwords, Odd, Unsigned, Modulo, Integer and Accumulate Negative into Words
EVX	10000504			SP	evmhousiaaw	Vector Multiply Halfwords, Odd, Unsigned, Saturate, Integer and Accumulate into Words
EVX	10000584			SP	evmhousianw	Vector Multiply Halfwords, Odd, Unsigned, Saturate, Integer and Accumulate Negative into Words
EVX	100004C4			SP	evmra	Initialize Accumulator
EVX	1000044F			SP	evmwhsmf	Vector Multiply Word High Signed, Modulo, Fractional
EVX	1000046F			SP	evmwhsmfa	Vector Multiply Word High Signed, Modulo, Fractional to Accumulator
EVX	1000054F			SP	evmwhsmfaaw	Vector Multiply Word High Signed, Modulo, Fractional and Accumulate into Words
EVX	100005CF			SP	evmwhsmfanw	Vector Multiply Word High Signed, Modulo, Fractional and Accumulate Negative into Words
EVX	1000044D			SP	evmwhsmi	Vector Multiply Word High Signed, Modulo, Integer
EVX	1000046D			SP	evmwhsmia	Vector Multiply Word High Signed, Modulo, Integer to Accumulator
EVX	1000054D			SP	evmwhsmiaaw	Vector Multiply Word High Signed, Modulo, Integer and Accumulate into Words
EVX	100005CD			SP	evmwhsmianw	Vector Multiply Word High Signed, Modulo, Integer and Accumulate Negative into Words
EVX	10000447			SP	evmwhssf	Vector Multiply Word High Signed, Fractional
EVX	10000467			SP	evmwhssfa	Vector Multiply Word High Signed, Fractional to Accumulator
EVX	10000547			SP	evmwhssfaaw	Vector Multiply Word High Signed, Fractional and Accumulate into Words
EVX	100005C7			SP	evmwhssfanw	Vector Multiply Word High Signed, Fractional and Accumulate Negative into Words
EVX	100005C5			SP	evmwhssianw	Vector Multiply Word High Signed, Integer and Accumulate Negative into Words
EVX	1000044C			SP	evmwhumi	Vector Multiply Word High Unsigned, Modulo, Integer
EVX	1000046C			SP	evmwhumia	Vector Multiply Word High Unsigned, Modulo, Integer to Accumulator
EVX	1000054C			SP	evmwhumiaaw	Vector Multiply Word High Unsigned, Modulo, Integer and Accumulate into Words
EVX	100005CC			SP	evmwhumianw	Vector Multiply Word High Unsigned, Modulo, Integer and Accumulate Negative into Words
EVX	10000544			SP	evmwhusiaaw	Vector Multiply Word High Unsigned, Integer and Accumulate into Words
EVX	100005C4			SP	evmwhusianw	Vector Multiply Word High Unsigned, Integer and Accumulate Negative into Words
EVX	10000549			SP	evmwlsmiaaw	Vector Multiply Word Low Signed, Modulo, Integer and Accumulate into Words
EVX	100005C9			SP	evmwlsmianw	Vector Multiply Word Low Signed, Modulo, Integer and Accumulate Negative into Words

Form	Opcode (hexadecimal)[2]	Mode Dep.[1]	Priv[1]	Cat[1]	Mnemonic	Instruction
EVX	10000541			SP	evmwlssiaaw	Vector Multiply Word Low Signed, Saturate, Integer and Accumulate into Words
EVX	100005C1			SP	evmwlssianw	Vector Multiply Word Low Signed, Saturate, Integer and Accumulate Negative into Words
EVX	10000448			SP	evmwlumi	Vector Multiply Word Low Unsigned, Modulo, Integer
EVX	10000468			SP	evmwlumia	Vector Multiply Word Low Unsigned, Modulo, Integer to Accumulator
EVX	10000548			SP	evmwlumiaaw	Vector Multiply Word Low Unsigned, Modulo, Integer and Accumulate into Words
EVX	100005C8			SP	evmwlumianw	Vector Multiply Word Low Unsigned, Modulo, Integer and Accumulate Negative into Words
EVX	10000540			SP	evmwlusiaaw	Vector Multiply Word Low Unsigned Saturate, Integer and Accumulate into Words
EVX	100005C0			SP	evmwlusianw	Vector Multiply Word Low Unsigned Saturate, Integer and Accumulate Negative into Words
EVX	1000045B			SP	evmwsmf	Vector Multiply Word Signed, Modulo, Fractional
EVX	1000047B			SP	evmwsmfa	Vector Multiply Word Signed, Modulo, Fractional to Accumulator
EVX	1000055B			SP	evmwsmfaa	Vector Multiply Word Signed, Modulo, Fractional and Accumulate
EVX	100005DB			SP	evmwsmfan	Vector Multiply Word Signed, Modulo, Fractional and Accumulate Negative
EVX	10000459			SP	evmwsmi	Vector Multiply Word Signed, Modulo, Integer
EVX	10000479			SP	evmwsmia	Vector Multiply Word Signed, Modulo, Integer to Accumulator
EVX	10000559			SP	evmwsmiaa	Vector Multiply Word Signed, Modulo, Integer and Accumulate
EVX	100005D9			SP	evmwsmian	Vector Multiply Word Signed, Modulo, Integer and Accumulate Negative
EVX	10000453			SP	evmwssf	Vector Multiply Word Signed, Saturate, Fractional
EVX	10000473			SP	evmwssfa	Vector Multiply Word Signed, Saturate, Fractional to Accumulator
EVX	10000553			SP	evmwssfaa	Vector Multiply Word Signed, Saturate, Fractional and Accumulate
EVX	100005D3			SP	evmwssfan	Vector Multiply Word Signed, Saturate, Fractional and Accumulate Negative
EVX	10000458			SP	evmwumi	Vector Multiply Word Unsigned, Modulo, Integer
EVX	10000478			SP	evmwumia	Vector Multiply Word Unsigned, Modulo, Integer to Accumulator
EVX	10000558			SP	evmwumiaa	Vector Multiply Word Unsigned, Modulo, Integer and Accumulate
EVX	100005D8			SP	evmwumian	Vector Multiply Word Unsigned, Modulo, Integer and Accumulate Negative
EVX	1000021E			SP	evnand	Vector NAND
EVX	10000209			SP	evneg	Vector Negate
EVX	10000218			SP	evnor	Vector NOR
EVX	10000217			SP	evor	Vector OR
EVX	1000021B			SP	evorc	Vector OR with Complement
EVX	10000228			SP	evrlw	Vector Rotate Left Word
EVX	1000022A			SP	evrlwi	Vector Rotate Left Word Immediate
EVX	1000020C			SP	evrndw	Vector Round Word
EVSEL	10000278			SP	evsel	Vector Select
EVX	10000224			SP	evslw	Vector Shift Left Word
EVX	10000226			SP	evslwi	Vector Shift Left Word Immediate
EVX	1000022B			SP	evsplatfi	Vector Splat Fractional Immediate
EVX	10000229			SP	evsplati	Vector Splat Immediate
EVX	10000223			SP	evsrwis	Vector Shift Right Word Immediate Signed
EVX	10000222			SP	evsrwiu	Vector Shift Right Word Immediate Unsigned
EVX	10000221			SP	evsrws	Vector Shift Right Word Signed

Form	Opcode (hexadeci-mal)[2]	Mode Dep.[1]	Priv[1]	Cat[1]	Mnemonic	Instruction
EVX	10000220			SP	evsrwu	Vector Shift Right Word Unsigned
EVX	10000321			SP	evstdd	Vector Store Doubleword of Doubleword
EVX	7C00019D		P	E.PD	evstddepx	Vector Store Doubleword into Doubleword by External Process ID Indexed
EVX	10000320			SP	evstddx	Vector Store Doubleword of Doubleword Indexed
EVX	10000325			SP	evstdh	Vector Store Doubleword of Four Halfwords
EVX	10000324			SP	evstdhx	Vector Store Doubleword of Four Halfwords Indexed
EVX	10000323			SP	evstdw	Vector Store Doubleword of Two Words
EVX	10000322			SP	evstdwx	Vector Store Doubleword of Two Words Indexed
EVX	10000331			SP	evstwhe	Vector Store Word of Two Halfwords from Even
EVX	10000330			SP	evstwhex	Vector Store Word of Two Halfwords from Even Indexed
EVX	10000335			SP	evstwho	Vector Store Word of Two Halfwords from Odd
EVX	10000334			SP	evstwhox	Vector Store Word of Two Halfwords from Odd Indexed
EVX	10000339			SP	evstwwe	Vector Store Word of Word from Even
EVX	10000338			SP	evstwwex	Vector Store Word of Word from Even Indexed
EVX	1000033D			SP	evstwwo	Vector Store Word of Word from Odd
EVX	1000033C			SP	evstwwox	Vector Store Word of Word from Odd Indexed
EVX	100004CB			SP	evsubfsmiaaw	Vector Subtract Signed, Modulo, Integer to Accumulator Word
EVX	100004C3			SP	evsubfssiaaw	Vector Subtract Signed, Saturate, Integer to Accumulator Word
EVX	100004CA			SP	evsubfumiaaw	Vector Subtract Unsigned, Modulo, Integer to Accumulator Word
EVX	100004C2			SP	evsubfusiaaw	Vector Subtract Unsigned, Saturate, Integer to Accumulator Word
EVX	10000204			SP	evsubfw	Vector Subtract from Word
EVX	10000206			SP	evsubifw	Vector Subtract Immediate from Word
EVX	10000216			SP	evxor	Vector XOR
X	7C000774	SR		B	extsb[.]	Extend Shign Byte
X	7C000734	SR		B	extsh[.]	Extend Sign Halfword
X	7C0007B4	SR		64	extsw[.]	Extend Sign Word
X	7C0007AC			B	icbi	Instruction Cache Block Invalidate
X	7C0007BE		P	E.PD	icbiep	Instruction Cache Block Invalidate by External Process ID
X	7C0001CC		M	ECL	icblc	Instruction Cache Block Lock Clear
X	7C00002C			E	icbt	Instruction Cache Block Touch
X	7C0003CC		M	ECL	icbtls	Instruction Cache Block Touch and Lock Set
X	7C00078C		P	E.CI	ici	Instruction Cache Invalidate
X	7C0007CC		P	E.CD	icread	Instruction Cache Read
A	7C00001E			B.in	isel	Integer Select
X	7C0000BE		P	E.PD	lbepx	Load Byte by External Process ID Indexed
X	7C0000EE			B	lbzux	Load Byte and Zero with Update Indexed
X	7C0000AE			B	lbzx	Load Byte and Zero Indexed
X	7C0000A8			64	ldarx	Load Doubleword and Reserve Indexed
X	7C00003A		P	E.PD	ldepx	Load Doubleword by External Process ID Indexed
X	7C00006A			64	ldux	Load Doubleword with Update Indexed
X	7C00002A			64	ldx	Load Doubleword Indexed
X	7C0004BE		P	E.PD	lfdepx	Load Floating-Point Double by External Process ID Indexed
X	7C0002EE			B	lhaux	Load Halfword Algebraic with Update Indexed
X	7C0002AE			B	lhax	Load Halfword Algebraic Indexed
X	7C00062C			B	lhbrx	Load Halfword Byte-Reversed Indexed
X	7C00023E		P	E.PD	lhepx	Load Halfword by External Process ID Indexed
X	7C00026E			B	lhzux	Load Halfword and Zero with Update Indexed
X	7C00022E			B	lhzx	Load Halfword and Zero Indexed
X	7C0004AA			MA	lswi	Load String Word Immediate
X	7C00042A			MA	lswx	Load String Word Indexed
X	7C00000E			V	lvebx	Load Vector Element Byte Indexed
X	7C00004E			V	lvehx	Load Vector Element Halfword Indexed
X	7C00024E		P	E.PD	lvepx	Load Vector by External Process ID Indexed
X	7C00020E		P	E.PD	lvepxl	Load Vector by External Process ID Indexed LRU
X	7C00008E			V	lvewx	Load Vector Element Word Indexed
X	7C00000C			V	lvsl	Load Vector for Shift Left Indexed

Form	Opcode (hexadecimal)[2]	Mode Dep.[1]	Priv[1]	Cat[1]	Mnemonic	Instruction
X	7C00004C			V	lvsr	Load Vector for Shift Right Indexed
X	7C0000CE			V	lvx[l]	Load Vector Indexed [Last]
X	7C000028			B	lwarx	Load Word and Reserve Indexed
X	7C0002EA			64	lwaux	Load Word Algebraic with Update Indexed
X	7C0002AA			64	lwax	Load Word Algebraic Indexed
X	7C00042C			B	lwbrx	Load Word Byte-Reversed Indexed
X	7C00003E		P	E.PD	lwepx	Load Word by External Process ID Indexed
X	7C00006E			B	lwzux	Load Word and Zero with Update Indexed
X	7C00002E			B	lwzx	Load Word and Zero Indexed
X	10000158	SR		LIM	macchw[o][.]	Multiply Accumulate Cross Halfword to Word Modulo Signed
X	100001D8	SR		LIM	macchws[o][.]	Multiply Accumulate Cross Halfword to Word Saturate Signed
X	10000198	SR		LIM	macchwsu[o][.]	Multiply Accumulate Cross Halfword to Word Saturate Unsigned
X	10000118	SR		LIM	macchwu[o][.]	Multiply Accumulate Cross Halfword to Word Modulo Unsigned
X	10000058	SR		LIM	machhw[o][.]	Multiply Accumulate High Halfword to Word Modulo Signed
X	100000D8	SR		LIM	machhws[o][.]	Multiply Accumulate High Halfword to Word Saturate Signed
X	10000098	SR		LIM	machhwsu[o][.]	Multiply Accumulate High Halfword to Word Saturate Unsigned
X	10000018	SR		LIM	machhwu[o][.]	Multiply Accumulate High Halfword to Word Modulo Unsigned
X	10000358	SR		LIM	maclhw[o][.]	Multiply Accumulate Low Halfword to Word Modulo Signed
X	100003D8	SR		LIM	maclhws[o][.]	Multiply Accumulate Low Halfword to Word Saturate Signed
X	10000398	SR		LIM	maclhwsu[o][.]	Multiply Accumulate Low Halfword to Word Saturate Unsigned
X	10000318	SR		LIM	maclhwu[o][.]	Multiply Accumulate Low Halfword to Word Modulo Unsigned
XFX	7C0006AC			E	mbar	Memory Barrier
X	7C000400			B	mcrxr	Move To Condition Register From XER
XFX	7C000026			B	mfcr	Move From Condition Register
XFX	7C000286		P	E	mfdcr	Move From Device Control Register
XFX	7C000246		P	E	mfdcrux	Move From Device Control Register User-mode Indexed
XFX	7C000206		P	E	mfdcrx	Move From Device Control Register Indexed
X	7C0000A6		P	B	mfmsr	Move From Machine State Register
XFX	7C100026			B	mfocrf	Move From One Condition Register Field
XFX	7C00029C		O	E.PM	mfpmr	Move From Performance Monitor Register
XFX	7C0002A6		O	B	mfspr	Move From Special Purpose Register
VX	10000604			V	mfvscr	Move from Vector Status and Control Register
X	7C0001DC		P	E.PC	msgclr	Message Clear
X	7C00019C		P	E.PC	msgsnd	Message Send
XFX	7C000120			B	mtcrf	Move To Condition Register Fields
XFX	7C000386		P	E	mtdcr	Move To Device Control Register
X	7C000346			E	mtdcrux	Move To Device Control Register User-mode Indexed
X	7C000306		P	E	mtdcrx	Move To Device Control Register Indexed
X	7C000124		P	E	mtmsr	Move To Machine State Register
XFX	7C100120			B	mtocrf	Move To One Condition Register Field
XFX	7C00039C		O	E.PM	mtpmr	Move To Performance Monitor Register
XFX	7C0003A6		O	B	mtspr	Move To Special Purpose Register
VX	10000644			V	mtvscr	Move to Vector Status and Control Register
X	10000150	SR		LIM	mulchw[o][.]	Multiply Cross Halfword to Word Signed
X	10000110	SR		LIM	mulchwu[o][.]	Multiply Cross Halfword to Word Unsigned
XO	7C000092	SR		64	mulhd[.]	Multiply High Doubleword
XO	7C000012	SR		64	mulhdu[.]	Multiply High Doubleword Unsigned
X	10000050	SR		LIM	mulhhw[o][.]	Multiply High Halfword to Word Signed
X	10000010	SR		LIM	mulhhwu[o][.]	Multiply High Halfword to Word Unsigned
XO	7C000096	SR		B	mulhw[.]	Multiply High Word
XO	7C000016	SR		B	mulhwu[.]	Multiply High Word Unsigned
XO	7C0001D2	SR		64	mulld[o][.]	Multiply Low Doubleword

Form	Opcode (hexadecimal)[2]	Mode Dep.[1]	Priv[1]	Cat[1]	Mnemonic	Instruction
XO	7C0001D6	SR		B	mullw[o][.]	Multiply Low Word
X	7C0003B8	SR		B	nand[.]	NAND
X	7C0000D0	SR		B	neg[o][.]	Negate
X	1000015C	SR		LIM	nmacchw[o][.]	Negative Multiply Accumulate Cross Halfword to Word Modulo Signed
X	100001DC	SR		LIM	nmacchws[o][.]	Negative Multiply Accumulate Cross Halfword to Word Saturate Signed
X	1000005C	SR		LIM	nmachhw[o][.]	Negative Multiply Accumulate High Halfword to Word Modulo Signed
X	100000DC	SR		LIM	nmachhws[o][.]	Negative Multiply Accumulate High Halfword to Word Saturate Signed
X	1000035C	SR		LIM	nmaclhw[o][.]	Negative Multiply Accumulate Low Halfword to Word Modulo Signed
X	100003DC	SR		LIM	nmaclhws[o][.]	Negative Multiply Accumulate Low Halfword to Word Saturate Signed
X	7C0000F8	SR		B	nor[.]	NOR
X	7C000378	SR		B	or[.]	OR
X	7C000338	SR		B	orc[.]	OR with Complement
X	7C0000F4			B	popcntb	Population Count Bytes
RR	0400----			VLE	se_add	Add Short Form
OIM5	2000----			VLE	se_addi	Add Immediate Short Form
RR	4600----	SR		VLE	se_and[.]	AND Short Form
RR	4500----			VLE	se_andc	AND with Complement Short Form
IM5	2E00----			VLE	se_andi	AND Immediate Short Form
BD8	E800----			VLE	se_b[l]	Branch [and Link]
BD8	E000----			VLE	se_bc	Branch Conditional Short Form
IM5	6000----			VLE	se_bclri	Bit Clear Immediate
C	0006----			VLE	se_bctr	Branch To Count Register [and Link]
IM5	6200----			VLE	se_bgeni	Bit Generate Immediate
C	0004----			VLE	se_blr	Branch To Link Register [and Link]
IM5	2C00----			VLE	se_bmaski	Bit Mask Generate Immediate
IM5	6400----			VLE	se_bseti	Bit Set Immediate
IM5	6600----			VLE	se_btsti	Bit Test Immediate
RR	0C00----			VLE	se_cmp	Compare Word
RR	0E00----			VLE	se_cmph	Compare Halfword Short Form
RR	0F00----			VLE	se_cmphl	Compare Halfword Logical Short Form
IM5	2A00----			VLE	se_cmpi	Compare Immediate Word Short Form
RR	0D00----			VLE	se_cmpl	Compare Logical Word
OIM5	2200----			VLE	se_cmpli	Compare Logical Immendiate Word
R	00D0----			VLE	se_extsb	Extend Sign Byte Short Form
R	00F0----			VLE	se_extsh	Extend Sign Halfword Short Form
R	00C0----			VLE	se_extzb	Extend Zero Byte
R	00E0----			VLE	se_extzh	Extend Zero Halfword
C	0000----			VLE	se_illegal	Illegal
C	0001----			VLE	se_isync	Instruction Synchronize
SD4	8000----			VLE	se_lbz	Load Byte and Zero Short Form
SD4	A000----			VLE	se_lhz	Load Halfword and Zero Short Form
IM7	4800----			VLE	se_li	Load Immediate Short Form
SD4	C000----			VLE	se_lwz	Load Word and Zero Short Form
RR	0300----			VLE	se_mfar	Move from Alternate Register
R	00A0----			VLE	se_mfctr	Move From Count Register
R	0080----			VLE	se_mflr	Move From Link Register
RR	0100----			VLE	se_mr	Move Register
RR	0200----			VLE	se_mtar	Move To Alternate Register
R	00B0----			VLE	se_mtctr	Move To Count Register
R	0090----			VLE	se_mtlr	Move To Link Register
RR	0500----			VLE	se_mullw	Multiply Low Word Short Form
R	0030----			VLE	se_neg	Negate Short Form
R	0020----			VLE	se_not	NOT Short Form
RR	4400----			VLE	se_or	OR SHort Form
C	0009----		P	VLE	se_rfci	Return From Critical Interrupt

Form	Opcode (hexadecimal)[2]	Mode Dep.[1]	Priv[1]	Cat[1]	Mnemonic	Instruction
C	000A----		P	VLE	se_rfdi	Return From Debug Interrupt
C	0008----		P	VLE	se_rfi	Return from Interrupt
C	000B----		P	VLE	se_rfmci	Return From Machine Check Interrupt
C	0002----			VLE	se_sc	System Call
RR	4200----			VLE	se_slw	Shift Left Word
IM5	6C00----			VLE	se_slwi	Shift Left Word Immediate Short Form
RR	4100----	SR		VLE	se_sraw	Shift Right Algebraic Word
IM5	6A00----	SR		VLE	se_srawi	Shift Right Algebraic Immediate
RR	4000----			VLE	se_srw	Shift Right Word
IM5	6800----			VLE	se_srwi	Shift Right Word Immediate Short Form
SD4	9000----			VLE	se_stb	Store Byte Short Form
SD4	B000----			VLE	se_sth	Store Halfword SHort Form
SD4	D000----			VLE	se_stw	Store Word Short Form
RR	0600----			VLE	se_sub	Subtract
RR	0700----			VLE	se_subf	Subtract From Short Form
OIM5	2400----	SR		VLE	se_subi[.]	Subtract Immediate
X	7C000036	SR		64	sld[.]	Shift Left Doubleword
X	7C000030	SR		B	slw[.]	Shift Left Word
X	7C000634	SR		64	srad[.]	Shift Right Algebraic Doubleword
X	7C000674	SR		64	sradi[.]	Shift Right Algebraic Doubleword Immediate
X	7C000630	SR		B	sraw[.]	Shift Right Algebraic Word
X	7C000670	SR		B	srawi[.]	Shift Right Algebraic Word Immediate
X	7C000436	SR		64	srd[.]	Shift Right Doubleword
X	7C000430	SR		B	srw[.]	Shift Right Word
X	7C0001BE		P	E.PD	stbepx	Store Byte by External Process ID Indexed
X	7C0001EE			B	stbux	Store Byte with Update Indexed
X	7C0001AE			B	stbx	Store Bye Indexed
X	7C0001AD			64	stdcx.	Store Doubleword Conditional Indexed
X	7C00013A		P	E.PD	stdepx	Store Doubleword by External Process ID Indexed
X	7C00016A			64	stdux	Store Doubleword with Update Indexed
X	7C00012A			64	stdx	Store Doubleword Indexed
X	7C0005BE		P	E.PD	stfdepx	Store Floating-Point Double by External Process ID Indexed
X	7C00072C			B	sthbrx	Store Halfword Byte-Reversed Indexed
X	7C00033E		P	E.PD	sthepx	Store Halfword by External Process ID Indexed
X	7C00036E			B	sthux	Store Halfword with Update Indexed
X	7C00032E			B	sthx	Store Halfword Indexed
X	7C0005AA			MA	stswi	Store String Word Immediate
X	7C00052A			MA	stswx	Store String Word Indexed
VX	7C00010E			V	stvebx	Store Vector Element Byte Indexed
VX	7C00014E			V	stvehx	Store Vector Element Halfword Indexed
X	7C00064E		P	E.PD	stvepx	Store Vector by External Process ID Indexed
X	7C00060E		P	E.PD	stvepxl	Store Vector by External Process ID Indexed LRU
VX	7C00018E			V	stvewx	Store Vector Element Word Indexed
VX	7C0001CE			V	stvx[l]	Store Vector Indexed [Last]
X	7C00052C			B	stwbrx	Store Word Byte-Reversed Indexed
X	7C00012D			B	stwcx.	Store Word Conditional Indexed
X	7C00013E		P	E.PD	stwepx	Store Word by External Process ID Indexed
X	7C00016E			B	stwux	Store Word with Update Indexed
X	7C00012E			B	stwx	Store Word Indexed
XO	7C000050	SR		B	subf[o][.]	Subtract From
XO	7C000010	SR		B	subfc[o][.]	Subtract From Carrying
XO	7C000110	SR		B	subfe[o][.]	Subtract From Extended
XO	7C0001D0	SR		B	subfme[o][.]	Subtract From Minus One Extended
XO	7C000190	SR		B	subfze[o][.]	Subtract From Zero Extended
X	7C0004AC			B	sync	Synchronize
X	7C000088			64	td	Trap Doubleword
X	7C000624		P	E	tlbivax	TLB Invalidate Virtual Address Indexed
X	7C000764		P	E	tlbre	TLB Read Entry
X	7C000724		P	E	tlbsx	TLB Search Indexed
X	7C00046C		P	E	tlbsync	TLB Synchronize
X	7C0007A4		P	E	tlbwe	TLB Write Entry

Form	Opcode (hexadeci-mal)[2]	Mode Dep.[1]	Priv[1]	Cat[1]	Mnemonic	Instruction
X	7C000008			B	tw	Trap Word
VX	10000180			V	vaddcuw	Vector Add Carryout Unsigned Word
VX	1000000A			V	vaddfp	Vector Add Floating-Point
VX	10000300			V	vaddsbs	Vector Add Signed Byte Saturate
VX	10000340			V	vaddshs	Vector Add Signed Halfword Saturate
VX	10000380			V	vaddsws	Vector Add Signed Word Saturate
VX	10000000			V	vaddubm	Vector Add Unsigned Byte Modulo
VX	10000200			V	vaddubs	Vector Add Unsigned Byte Saturate
VX	10000040			V	vadduhm	Vector Add Unsigned Halfword Modulo
VX	10000240			V	vadduhs	Vector Add Unsigned Halfword Saturate
VX	10000080			V	vadduwm	Vector Add Unsigned Word Modulo
VX	10000280			V	vadduws	Vector Add Unsigned Word Saturate
VX	10000404			V	vand	Vector AND
VX	10000444			V	vandc	Vector AND with Complement
VX	10000502			V	vavgsb	Vector Average Signed Byte
VX	10000542			V	vavgsh	Vector Average Signed Halfword
VX	10000582			V	vavgsw	Vector Average Signed Word
VX	10000402			V	vavgub	Vector Average Unsigned Byte
VX	10000442			V	vavguh	Vector Average Unsigned Halfword
VX	10000482			V	vavguw	Vector Average Unsigned Word
VX	100003CA			V	vcfpsxws	Vector Convert from Single-Precision to Signed Fixed-Point Word Saturate
VX	1000038A			V	vcfpuxws	Vector Convert from Single-Precision to Unsigned Fixed-Point Word Saturate
VX	100003C6			V	vcmpbfp[.]	Vector Compare Bounds Single-Precision
VC	100000C6			V	vcmpeqfp[.]	Vector Compare Equal To Single-Precision
VC	10000006			V	vcmpequb[.]	Vector Compare Equal To Unsigned Byte
VC	10000046			V	vcmpequh[.]	Vector Compare Equal To Unsigned Halfword
VC	10000086			V	vcmpequw[.]	Vector Compare Equal To Unsigned Word
VC	100001C6			V	vcmpgefp[.]	Vector Compare Greater Than or Equal To Single-Precision
VC	100002C6			V	vcmpgtfp[.]	Vector Compare Greater Than Single-Precision
VC	10000306			V	vcmpgtsb[.]	Vector Compare Greater Than Signed Byte
VC	10000346			V	vcmpgtsh[.]	Vector Compare Greater Than Signed Halfword
VC	10000386			V	vcmpgtsw[.]	Vector Compare Greater Than Signed Word
VC	10000206			V	vcmpgtub[.]	Vector Compare Greater Than Unsigned Byte
VC	10000246			V	vcmpgtuh[.]	Vector Compare Greater Than Unsigned Halfword
VC	10000286			V	vcmpgtuw[.]	Vector Compare Greater Than Unsigned Word
VX	1000034A			V	vcsxwfp	Vector Convert from Signed Fixed-Point Word to Single-Precision
VX	1000030A			V	vcuxwfp	Vector Convert from Unsigned Fixed-Point Word to Single-Precision
VX	1000018A			V	vexptefp	Vector 2 Raised to the Exponent Estimate Floating-Point
VX	100001CA			V	vlogefp	Vector Log Base 2 Estimate Floating-Point
VA	1000002E			V	vmaddfp	Vector Multiply-Add Single-Precision
VX	1000040A			V	vmaxfp	Vector Maximum Single-Precision
VX	10000102			V	vmaxsb	Vector Maximum Signed Byte
VX	10000142			V	vmaxsh	Vector Maximum Signed Halfword
VX	10000182			V	vmaxsw	Vector Maximum Signed Word
VX	10000002			V	vmaxub	Vector Maximum Unsigned Byte
VX	10000042			V	vmaxuh	Vector Maximum Unsigned Halfword
VX	10000082			V	vmaxuw	Vector Maximum Unsigned Word
VA	10000020			V	vmhaddshs	Vector Multiply-High-Add Signed Halfword Saturate
VA	10000021			V	vmhraddshs	Vector Multiply-High-Round-Add Signed Halfword Saturate
VX	1000044A			V	vminfp	Vector Minimum Single-Precision
VX	10000302			V	vminsb	Vector Minimum Signed Byte
VX	10000342			V	vminsh	Vector Minimum Signed Halfword
VX	10000382			V	vminsw	Vector Minimum Signed Word
VX	10000202			V	vminub	Vector Minimum Unsigned Byte
VX	10000242			V	vminuh	Vector Minimum Unsigned Halfword
VX	10000282			V	vminuw	Vector Minimum Unsigned Word
VA	10000022			V	vmladduhm	Vector Multiply-Low-Add Unsigned Halfword Modulo

Form	Opcode (hexadecimal)[2]	Mode Dep.[1]	Priv[1]	Cat[1]	Mnemonic	Instruction
VX	1000000C			V	vmrghb	Vector Merge High Byte
VX	1000004C			V	vmrghh	Vector Merge High Halfword
VX	1000008C			V	vmrghw	Vector Merge High Word
VX	1000010C			V	vmrglb	Vector Merge Low Byte
VX	1000014C			V	vmrglh	Vector Merge Low Halfword
VX	1000018C			V	vmrglw	Vector Merge Low Word
VA	10000025			V	vmsummbm	Vector Multiply-Sum Mixed Byte Modulo
VA	10000028			V	vmsumshm	Vector Multiply-Sum Signed Halfword Modulo
VA	10000029			V	vmsumshs	Vector Multiply-Sum Signed Halfword Saturate
VA	10000024			V	vmsumubm	Vector Multiply-Sum Unsigned Byte Modulo
VA	10000026			V	vmsumuhm	Vector Multiply-Sum Unsigned Halfword Modulo
VA	10000027			V	vmsumuhs	Vector Multiply-Sum Unsigned Halfword Saturate
VX	10000308			V	vmulesb	Vector Multiply Even Signed Byte
VX	10000348			V	vmulesh	Vector Multiply Even Signed Halfword
VX	10000208			V	vmuleub	Vector Multiply Even Unsigned Byte
VX	10000248			V	vmuleuh	Vector Multiply Even Unsigned Halfword
VX	10000108			V	vmulosb	Vector Multiply Odd Signed Byte
VX	10000148			V	vmulosh	Vector Multiply Odd Signed Halfword
VX	10000008			V	vmuloub	Vector Multiply Odd Unsigned Byte
VX	10000048			V	vmulouh	Vector Multiply Odd Unsigned Halfword
VA	1000002F			V	vnmsubfp	Vector Negative Multiply-Subtract Single-Precision
VX	10000504			V	vnor	Vector NOR
VX	10000484			V	vor	Vector OR
VA	1000002B			V	vperm	Vector Permute
VX	1000030E			V	vpkpx	Vector Pack Pixel
VX	1000018E			V	vpkshss	Vector Pack Signed Halfword Signed Saturate
VX	1000010E			V	vpkshus	Vector Pack Signed Halfword Unsigned Saturate
VX	100001CE			V	vpkswss	Vector Pack Signed Word Signed Saturate
VX	1000014E			V	vpkswus	Vector Pack Signed Word Unsigned Saturate
VX	1000000E			V	vpkuhum	Vector Pack Unsigned Halfword Unsigned Modulo
VX	1000008E			V	vpkuhus	Vector Pack Unsigned Halfword Unsigned Saturate
VX	1000004E			V	vpkuwum	Vector Pack Unsigned Word Unsigned Modulo
VX	100000CE			V	vpkuwus	Vector Pack Unsigned Word Unsigned Saturate
VX	1000010A			V	vrefp	Vector Reciprocal Estimate Single-Precision
VX	100002CA			V	vrfim	Vector Round to Single-Precision Integer toward -Infinity
VX	1000020A			V	vrfin	Vector Round to Single-Precision Integer Nearest
VX	1000028A			V	vrfip	Vector Round to Single-Precision Integer toward +Infinity
VX	1000024A			V	vrfiz	Vector Round to Single-Precision Integer toward Zero
VX	10000004			V	vrlb	Vector Rotate Left Byte
VX	10000044			V	vrlh	Vector Rotate Left Halfword
VX	10000084			V	vrlw	Vector Rotate Left Word
VX	1000014A			V	vrsqrtefp	Vector Reciprocal Square Root Estimate Single-Precision
VA	1000002A			V	vsel	Vector Select
VX	100001C4			V	vsl	Vector Shift Left
VX	10000104			V	vslb	Vector Shift Left Byte
VA	1000002C			V	vsldoi	Vector Shift Left Double by Octet Immediate
VX	10000144			V	vslh	Vector Shift Left Halfword
VX	1000040C			V	vslo	Vector Shift Left by Octet
VX	10000184			V	vslw	Vector Shift Left Word
VX	1000020C			V	vspltb	Vector Splat Byte
VX	1000024C			V	vsplth	Vector Splat Halfword
VX	1000030C			V	vspltisb	Vector Splat Immediate Signed Byte
VX	1000034C			V	vspltish	Vector Splat Immediate Signed Halfword
VX	1000038C			V	vspltisw	Vector Splat Immediate Signed Word
VX	1000028C			V	vspltw	Vector Splat Word
VX	100002C4			V	vsr	Vector Shift Right
VX	10000304			V	vsrab	Vector Shift Right Algebraic Word
VX	10000344			V	vsrah	Vector Shift Right Algebraic Word
VX	10000384			V	vsraw	Vector Shift Right Algebraic Word
VX	10000204			V	vsrb	Vector Shift Right Byte
VX	10000244			V	vsrh	Vector Shift Right Halfword

Form	Opcode (hexadeci-mal)[2]	Mode Dep.[1]	Priv[1]	Cat[1]	Mnemonic	Instruction
VX	1000044C			V	vsro	Vector Shift Right by Octet
VX	10000284			V	vsrw	Vector Shift Right Word
VX	10000580			V	vsubcuw	Vector Subtract and Write Carry-Out Unsigned Word
VX	1000004A			V	vsubfp	Vector Subtract Single-Precision
VX	10000700			V	vsubsbs	Vector Subtract Signed Byte Saturate
VX	10000740			V	vsubshs	Vector Subtract Signed Halfword Saturate
VX	10000780			V	vsubsws	Vector Subtract Signed Word Saturate
VX	10000400			V	vsububm	Vector Subtract Unsigned Byte Modulo
VX	10000600			V	vsububs	Vector Subtract Unsigned Byte Saturate
VX	10000440			V	vsubuhm	Vector Subtract Unsigned Byte Modulo
VX	10000640			V	vsubuhs	Vector Subtract Unsigned Halfword Saturate
VX	10000480			V	vsubuwm	Vector Subtract Unsigned Word Modulo
VX	10000680			V	vsubuws	Vector Subtract Unsigned Word Saturate
VX	10000688			V	vsum2sws	Vector Sum across Half Signed Word Saturate
VX	10000708			V	vsum4sbs	Vector Sum across Quarter Signed Byte Saturate
VX	10000648			V	vsum4shs	Vector Sum across Quarter Signed Halfword Saturate
VX	10000608			V	vsum4ubs	Vector Sum across Quarter Unsigned Byte Saturate
VX	10000788			V	vsumsws	Vector Sum across Signed Word Saturate
VX	1000034E			V	vupkhpx	Vector Unpack High Pixel
VX	1000020E			V	vupkhsb	Vector Unpack High Signed Byte
VX	1000024E			V	vupkhsh	Vector Unpack High Signed Halfword
VX	100003CE			V	vupklpx	Vector Unpack Low Pixel
VX	1000028E			V	vupklsb	Vector Unpack Low Signed Byte
VX	100002CE			V	vupklsh	Vector Unpack Low Signed Halfword
VX	100004C4			V	vxor	Vector XOR
X	7C00007C			WT	wait	Wait
X	7C000106		P	E	wrtee	Write MSR External Enable
X	7C000146		P	E	wrteei	Write MSR External Enable Immediate
D	7C000278	SR		B	xor[.]	XOR

1 See the key to the mode dependency and privilege columns on page 839 and the key to the category column in Section 1.3.5 of Book I.

2 For 16-bit instructions, the “Opcode” column represents the 16-bit hexadecimal instruction encoding with the opcode and extended opcode in the corresponding fields in the instruction, and with 0’s in bit positions which are not opcode bits; dashes are used following the opcode to indicate the form is a 16-bit instruction. For 32-bit instructions, the “Opcode” column represents the 32-bit hexadecimal instruction encoding with the opcode and extended opcode in the corresponding fields in the instruction, and with 0’s in bit positions which are not opcode bits.

Appendix B. VLE Instruction Set Sorted by Opcode

This appendix lists all the instructions available in VLE mode in the Power ISA , in order by opcode. Opcodes that are not defined below are treated as illegal by category VLE.

Form	Opcode (hexadecimal)[2]	Mode Dep.[1]	Priv[1]	Cat[1]	Mnemonic	Instruction
C	0000----			VLE	se_illegal	Illegal
C	0001----			VLE	se_isync	Instruction Synchronize
C	0002----			VLE	se_sc	System Call
C	0004----			VLE	se_blr	Branch To Link Register [and Link]
C	0006----			VLE	se_bctr	Branch To Count Register [and Link]
C	0008----		P	VLE	se_rfi	Return from Interrupt
C	0009----		P	VLE	se_rfci	Return From Critical Interrupt
C	000A----		P	VLE	se_rfdi	Return From Debug Interrupt
C	000B----		P	VLE	se_rfmci	Return From Machine Check Interrupt
R	0020----			VLE	se_not	NOT Short Form
R	0030----			VLE	se_neg	Negate Short Form
R	0080----			VLE	se_mflr	Move From Link Register
R	0090----			VLE	se_mtlr	Move To Link Register
R	00A0----			VLE	se_mfctr	Move From Count Register
R	00B0----			VLE	se_mtctr	Move To Count Register
R	00C0----			VLE	se_extzb	Extend Zero Byte
R	00D0----			VLE	se_extsb	Extend Sign Byte Short Form
R	00E0----			VLE	se_extzh	Extend Zero Halfword
R	00F0----			VLE	se_extsh	Extend Sign Halfword Short Form
RR	0100----			VLE	se_mr	Move Register
RR	0200----			VLE	se_mtar	Move To Alternate Register
RR	0300----			VLE	se_mfar	Move from Alternate Register
RR	0400----			VLE	se_add	Add Short Form
RR	0500----			VLE	se_mullw	Multiply Low Word Short Form
RR	0600----			VLE	se_sub	Subtract
RR	0700----			VLE	se_subf	Subtract From Short Form
RR	0C00----			VLE	se_cmp	Compare Word
RR	0D00----			VLE	se_cmpl	Compare Logical Word
RR	0E00----			VLE	se_cmph	Compare Halfword Short Form
RR	0F00----			VLE	se_cmphl	Compare Halfword Logical Short Form
VX	10000000			V	vaddubm	Vector Add Unsigned Byte Modulo
VX	10000002			V	vmaxub	Vector Maximum Unsigned Byte
VX	10000004			V	vrlb	Vector Rotate Left Byte
VC	10000006			V	vcmpequb[.]	Vector Compare Equal To Unsigned Byte
VX	10000008			V	vmuloub	Vector Multiply Odd Unsigned Byte
VX	1000000A			V	vaddfp	Vector Add Floating-Point
VX	1000000C			V	vmrghb	Vector Merge High Byte
VX	1000000E			V	vpkuhum	Vector Pack Unsigned Halfword Unsigned Modulo
X	10000010	SR		LIM	mulhhwu[o][.]	Multiply High Halfword to Word Unsigned
X	10000018	SR		LIM	machhwu[o][.]	Multiply Accumulate High Halfword to Word Modulo Unsigned
VA	10000020			V	vmhaddshs	Vector Multiply-High-Add Signed Halfword Saturate
VA	10000021			V	vmhraddshs	Vector Multiply-High-Round-Add Signed Halfword Saturate
VA	10000022			V	vmladduhm	Vector Multiply-Low-Add Unsigned Halfword Modulo
VA	10000024			V	vmsumubm	Vector Multiply-Sum Unsigned Byte Modulo

Form	Opcode (hexadecimal)[2]	Mode Dep.[1]	Priv[1]	Cat[1]	Mnemonic	Instruction
VA	10000025			V	vmsummbm	Vector Multiply-Sum Mixed Byte Modulo
VA	10000026			V	vmsumuhm	Vector Multiply-Sum Unsigned Halfword Modulo
VA	10000027			V	vmsumuhs	Vector Multiply-Sum Unsigned Halfword Saturate
VA	10000028			V	vmsumshm	Vector Multiply-Sum Signed Halfword Modulo
VA	10000029			V	vmsumshs	Vector Multiply-Sum Signed Halfword Saturate
VA	1000002A			V	vsel	Vector Select
VA	1000002B			V	vperm	Vector Permute
VA	1000002C			V	vsldoi	Vector Shift Left Double by Octet Immediate
VA	1000002E			V	vmaddfp	Vector Multiply-Add Single-Precision
VA	1000002F			V	vnmsubfp	Vector Negative Multiply-Subtract Single-Precision
VX	10000040			V	vadduhm	Vector Add Unsigned Halfword Modulo
VX	10000042			V	vmaxuh	Vector Maximum Unsigned Halfword
VX	10000044			V	vrlh	Vector Rotate Left Halfword
VC	10000046			V	vcmpequh[.]	Vector Compare Equal To Unsigned Halfword
VX	10000048			V	vmulouh	Vector Multiply Odd Unsigned Halfword
VX	1000004A			V	vsubfp	Vector Subtract Single-Precision
VX	1000004C			V	vmrghh	Vector Merge High Halfword
VX	1000004E			V	vpkuwum	Vector Pack Unsigned Word Unsigned Modulo
X	10000050	SR		LIM	mulhhw[o][.]	Multiply High Halfword to Word Signed
X	10000058	SR		LIM	machhw[o][.]	Multiply Accumulate High Halfword to Word Modulo Signed
X	1000005C	SR		LIM	nmachhw[o][.]	Negative Multiply Accumulate High Halfword to Word Modulo Signed
VX	10000080			V	vadduwm	Vector Add Unsigned Word Modulo
VX	10000082			V	vmaxuw	Vector Maximum Unsigned Word
VX	10000084			V	vrlw	Vector Rotate Left Word
VC	10000086			V	vcmpequw[.]	Vector Compare Equal To Unsigned Word
VX	1000008C			V	vmrghw	Vector Merge High Word
VX	1000008E			V	vpkuhus	Vector Pack Unsigned Halfword Unsigned Saturate
X	10000098	SR		LIM	machhwsu[o][.]	Multiply Accumulate High Halfword to Word Saturate Unsigned
VC	100000C6			V	vcmpeqfp[.]	Vector Compare Equal To Single-Precision
VX	100000CE			V	vpkuwus	Vector Pack Unsigned Word Unsigned Saturate
X	100000D8	SR		LIM	machhws[o][.]	Multiply Accumulate High Halfword to Word Saturate Signed
X	100000DC	SR		LIM	nmachhws[o][.]	Negative Multiply Accumulate High Halfword to Word Saturate Signed
VX	10000102			V	vmaxsb	Vector Maximum Signed Byte
VX	10000104			V	vslb	Vector Shift Left Byte
VX	10000108			V	vmulosb	Vector Multiply Odd Signed Byte
VX	1000010A			V	vrefp	Vector Reciprocal Estimate Single-Precision
VX	1000010C			V	vmrglb	Vector Merge Low Byte
VX	1000010E			V	vpkshus	Vector Pack Signed Halfword Unsigned Saturate
X	10000110	SR		LIM	mulchwu[o][.]	Multiply Cross Halfword to Word Unsigned
X	10000118	SR		LIM	macchwu[o][.]	Multiply Accumulate Cross Halfword to Word Modulo Unsigned
VX	10000142			V	vmaxsh	Vector Maximum Signed Halfword
VX	10000144			V	vslh	Vector Shift Left Halfword
VX	10000148			V	vmulosh	Vector Multiply Odd Signed Halfword
VX	1000014A			V	vrsqrtefp	Vector Reciprocal Square Root Estimate Single-Precision
VX	1000014C			V	vmrglh	Vector Merge Low Halfword
VX	1000014E			V	vpkswus	Vector Pack Signed Word Unsigned Saturate
X	10000150	SR		LIM	mulchw[o][.]	Multiply Cross Halfword to Word Signed
X	10000158	SR		LIM	macchw[o][.]	Multiply Accumulate Cross Halfword to Word Modulo Signed
X	1000015C	SR		LIM	nmacchw[o][.]	Negative Multiply Accumulate Cross Halfword to Word Modulo Signed
VX	10000180			V	vaddcuw	Vector Add Carryout Unsigned Word
VX	10000182			V	vmaxsw	Vector Maximum Signed Word
VX	10000184			V	vslw	Vector Shift Left Word
VX	1000018A			V	vexptefp	Vector 2 Raised to the Exponent Estimate Floating-Point
VX	1000018C			V	vmrglw	Vector Merge Low Word

Form	Opcode (hexadecimal)[2]	Mode Dep.[1]	Priv[1]	Cat[1]	Mnemonic	Instruction
VX	1000018E			V	vpkshss	Vector Pack Signed Halfword Signed Saturate
X	10000198	SR		LIM	macchwsu[o][.]	Multiply Accumulate Cross Halfword to Word Saturate Unsigned
VX	100001C4			V	vsl	Vector Shift Left
VC	100001C6			V	vcmpgefp[.]	Vector Compare Greater Than or Equal To Single-Precision
VX	100001CA			V	vlogefp	Vector Log Base 2 Estimate Floating-Point
VX	100001CE			V	vpkswss	Vector Pack Signed Word Signed Saturate
X	100001D8	SR		LIM	macchws[o][.]	Multiply Accumulate Cross Halfword to Word Saturate Signed
X	100001DC	SR		LIM	nmacchws[o][.]	Negative Multiply Accumulate Cross Halfword to Word Saturate Signed
EVX	10000200			SP	evaddw	Vector Add Word
VX	10000200			V	vaddubs	Vector Add Unsigned Byte Saturate
EVX	10000202			SP	evaddiw	Vector Add Immediate Word
VX	10000202			V	vminub	Vector Minimum Unsigned Byte
EVX	10000204			SP	evsubfw	Vector Subtract from Word
VX	10000204			V	vsrb	Vector Shift Right Byte
EVX	10000206			SP	evsubifw	Vector Subtract Immediate from Word
VC	10000206			V	vcmpgtub[.]	Vector Compare Greater Than Unsigned Byte
EVX	10000208			SP	evabs	Vector Absolute Value
VX	10000208			V	vmuleub	Vector Multiply Even Unsigned Byte
EVX	10000209			SP	evneg	Vector Negate
EVX	1000020A			SP	evextsb	Vector Extend Sign Byte
VX	1000020A			V	vrfin	Vector Round to Single-Precision Integer Nearest
EVX	1000020B			SP	evextsh	Vector Extend Sign Halfword
EVX	1000020C			SP	evrndw	Vector Round Word
VX	1000020C			V	vspltb	Vector Splat Byte
EVX	1000020D			SP	evcntlzw	Vector Count Leading Zeros Bits Word
EVX	1000020E			SP	evcntlsw	Vector Count Leading Sign Bits Word
VX	1000020E			V	vupkhsb	Vector Unpack High Signed Byte
EVX	1000020F			SP	brinc	Bit Reverse Increment
EVX	10000211			SP	evand	Vector AND
EVX	10000212			SP	evandc	Vector AND with Complement
EVX	10000216			SP	evxor	Vector XOR
EVX	10000217			SP	evor	Vector OR
EVX	10000218			SP	evnor	Vector NOR
EVX	10000219			SP	eveqv	Vector Equivalent
EVX	1000021B			SP	evorc	Vector OR with Complement
EVX	1000021E			SP	evnand	Vector NAND
EVX	10000220			SP	evsrwu	Vector Shift Right Word Unsigned
EVX	10000221			SP	evsrws	Vector Shift Right Word Signed
EVX	10000222			SP	evsrwiu	Vector Shift Right Word Immediate Unsigned
EVX	10000223			SP	evsrwis	Vector Shift Right Word Immediate Signed
EVX	10000224			SP	evslw	Vector Shift Left Word
EVX	10000226			SP	evslwi	Vector Shift Left Word Immediate
EVX	10000228			SP	evrlw	Vector Rotate Left Word
EVX	10000229			SP	evsplati	Vector Splat Immediate
EVX	1000022A			SP	evrlwi	Vector Rotate Left Word Immediate
EVX	1000022B			SP	evsplatfi	Vector Splat Fractional Immediate
EVX	1000022C			SP	evmergehi	Vector Merge High
EVX	1000022D			SP	evmergelo	Vector Merge Low
EVX	1000022E			SP	evmergehilo	Vector Merge High/Low
EVX	1000022F			SP	evmergelohi	Vector Merge Low/High
EVX	10000230			SP	evcmpgtu	Vector Compare Greater Than Unsigned
EVX	10000231			SP	evcmpgts	Vector Compare Greater Than Signed
EVX	10000232			SP	evcmpltu	Vector Compare Less Than Unsigned
EVX	10000233			SP	evcmplts	Vector Compare Less Than Signed
EVX	10000234			SP	evcmpeq	Vector Compare Equal
VX	10000240			V	vadduhs	Vector Add Unsigned Halfword Saturate
VX	10000242			V	vminuh	Vector Minimum Unsigned Halfword
VX	10000244			V	vsrh	Vector Shift Right Halfword

Form	Opcode (hexadeci-mal)[2]	Mode Dep.[1]	Priv[1]	Cat[1]	Mnemonic	Instruction
VC	10000246			V	vcmpgtuh[.]	Vector Compare Greater Than Unsigned Halfword
VX	10000248			V	vmuleuh	Vector Multiply Even Unsigned Halfword
VX	1000024A			V	vrfiz	Vector Round to Single-Precision Integer toward Zero
VX	1000024C			V	vsplth	Vector Splat Halfword
VX	1000024E			V	vupkhsh	Vector Unpack High Signed Halfword
EVSEL	10000278			SP	evsel	Vector Select
EVX	10000280			SP.FV	evfsadd	Vector Floating-Point Single-Precision Add
VX	10000280			V	vadduws	Vector Add Unsigned Word Saturate
EVX	10000281			SP.FV	evfssub	Vector Floating-Point Single-Precision Subtract
VX	10000282			V	vminuw	Vector Minimum Unsigned Word
EVX	10000284			SP.FV	evfsabs	Vector Floating-Point Single-Precision Absolute Value
VX	10000284			V	vsrw	Vector Shift Right Word
EVX	10000285			SP.FV	evfsnabs	Vector Floating-Point Single-Precision Negative Absolute Value
EVX	10000286			SP.FV	evfsneg	Vector Floating-Point Single-Precision Negate
VC	10000286			V	vcmpgtuw[.]	Vector Compare Greater Than Unsigned Word
EVX	10000288			SP.FV	evfsmul	Vector Floating-Point Single-Precision Multiply
EVX	10000289			SP.FV	evfsdiv	Vector Floating-Point Single-Precision Divide
VX	1000028A			V	vrfip	Vector Round to Single-Precision Integer toward +Infinity
EVX	1000028C			SP.FV	evfscmpgt	Vector Floating-Point Single-Precision Compare Greater Than
VX	1000028C			V	vspltw	Vector Splat Word
EVX	1000028D			SP.FV	evfscmplt	Vector Floating-Point Single-Precision Compare Less Than
EVX	1000028E			SP.FV	evfscmpeq	Vector Floating-Point Single-Precision Compare Equal
VX	1000028E			V	vupklsb	Vector Unpack Low Signed Byte
EVX	10000290			SP.FV	evfscfui	Vector Convert Floating-Point Single-Precision from Unsigned Integer
EVX	10000291			SP.FV	evfscfsi	Vector Convert Floating-Point Single-Precision from Signed Integer
EVX	10000292			SP.FV	evfscfuf	Vector Convert Floating-Point Single-Precision from Unsigned Fraction
EVX	10000293			SP.FV	evfscfsf	Vector Convert Floating-Point Single-Precision from Signed Fraction
EVX	10000294			SP.FV	evfsctui	Vector Convert Floating-Point Single-Precision to Unsigned Integer
EVX	10000295			SP.FV	evfsctsi	Vector Convert Floating-Point Single-Precision to Signed Integer
EVX	10000296			SP.FV	evfsctuf	Vector Convert Floating-Point Single-Precision to Unsigned Fraction
EVX	10000297			SP.FV	evfsctsf	Vector Convert Floating-Point Single-Precision to Signed Fraction
EVX	10000298			SP.FV	evfsctuiz	Vector Convert Floating-Point Single-Precision to Unsigned Integer with Round Towards Zero
EVX	1000029A			SP.FV	evfsctsiz	Vector Convert Floating-Point Single-Precision to Signed Integer with Round Towards Zero
EVX	1000029C			SP.FV	evfststgt	Vector Floating-Point Single-Precision Test Greater Than
EVX	1000029D			SP.FV	evfststlt	Vector Floating-Point Single-Precision Test Less Than
EVX	1000029E			SP.FV	evfststeq	Vector Floating-Point Single-Precision Test Equal
VX	100002C4			V	vsr	Vector Shift Right
VC	100002C6			V	vcmpgtfp[.]	Vector Compare Greater Than Single-Precision
VX	100002CA			V	vrfim	Vector Round to Single-Precision Integer toward -Infinity
VX	100002CE			V	vupklsh	Vector Unpack Low Signed Halfword
EVX	100002CF			SP.FD	efscfd	Floating-Point Single-Precision Convert from Double-Precision
EVX	100002E0			SP.FD	efdadd	Floating-Point Double-Precision Add
EVX	100002E0			SP.FS	efsadd	Floating-Point Single-Precision Add
EVX	100002E1			SP.FD	efdsub	Floating-Point Double-Precision Subtract
EVX	100002E1			SP.FS	efssub	Floating-Point Single-Precision Subtract

Form	Opcode (hexadecimal)[2]	Mode Dep.[1]	Priv[1]	Cat[1]	Mnemonic	Instruction
EVX	100002E2			SP.FD	efdcfuid	Convert Floating-Point Double-Precision from Unsigned Integer Doubleword
EVX	100002E2			SP.FS	efscfuid	Convert Floating-Point Single-Precision from Unsigned Integer Doubleword
EVX	100002E3			SP.FD	efdcfsid	Convert Floating-Point Double-Precision from Signed Integer Doubleword
EVX	100002E3			SP.FS	efscfsid	Convert Floating-Point Single-Precision from Signed Integer Doubleword
EVX	100002E4			SP.FD	efdabs	Floating-Point Double-Precision Absolute Value
EVX	100002E4			SP.FS	efsabs	Floating-Point Single-Precision Absolute Value
EVX	100002E5			SP.FD	efdnabs	Floating-Point Double-Precision Negative Absolute Value
EVX	100002E5			SP.FS	efsnabs	Floating-Point Single-Precision Negative Absolute Value
EVX	100002E6			SP.FD	efdneg	Floating-Point Double-Precision Negate
EVX	100002E6			SP.FS	efsneg	Floating-Point Single-Precision Negate
EVX	100002E8			SP.FD	efdmul	Floating-Point Double-Precision Multiply
EVX	100002E8			SP.FS	efsmul	Floating-Point Single-Precision Multiply
EVX	100002E9			SP.FD	efddiv	Floating-Point Double-Precision Divide
EVX	100002E9			SP.FS	efsdiv	Floating-Point Single-Precision Divide
EVX	100002EA			SP.FD	efdctuidz	Convert Floating-Point Double-Precision to Unsigned Integer Doubleword with Round Towards Zero
EVX	100002EA			SP.FS	efsctuidz	Convert Floating-Point Single-Precision to Unsigned Integer Doubleword with Round Towards Zero
EVX	100002EB			SP.FD	efdctsidz	Convert Floating-Point Double-Precision to Signed Integer Doubleword with Round Towards Zero
EVX	100002EB			SP.FS	efsctsidz	Convert Floating-Point Single-Precision to Signed Integer Doubleword with Round Towards Zero
EVX	100002EC			SP.FD	efdcmpgt	Floating-Point Double-Precision Compare Greater Than
EVX	100002EC			SP.FS	efscmpgt	Floating-Point Single-Precision Compare Greater Than
EVX	100002ED			SP.FD	efdcmplt	Floating-Point Double-Precision Compare Less Than
EVX	100002ED			SP.FS	efscmplt	Floating-Point Single-Precision Compare Less Than
EVX	100002EE			SP.FD	efdcmpeq	Floating-Point Double-Precision Compare Equal
EVX	100002EE			SP.FS	efscmpeq	Floating-Point Single-Precision Compare Equal
EVX	100002EF			SP.FD	efdcfs	Floating-Point Double-Precision Convert from Single-Precision
EVX	100002F0			SP.FD	efdcfui	Convert Floating-Point Double-Precision from Unsigned Integer
EVX	100002F0			SP.FS	efscfui	Convert Floating-Point Single-Precision from Unsigned Integer
EVX	100002F1			SP.FD	efdcfsi	Convert Floating-Point Double-Precision from Signed Integer
EVX	100002F1			SP.FS	efscfsi	Convert Floating-Point Single-Precision from Signed Integer
EVX	100002F2			SP.FD	efdcfuf	Convert Floating-Point Double-Precision from Unsigned Fraction
EVX	100002F2			SP.FS	efscfuf	Convert Floating-Point Single-Precision from Unsigned Fraction
EVX	100002F3			SP.FD	efdcfsf	Convert Floating-Point Double-Precision from Signed Fraction
EVX	100002F3			SP.FS	efscfsf	Convert Floating-Point Single-Precision from Signed Fraction
EVX	100002F4			SP.FD	efdctui	Convert Floating-Point Double-Precision to Unsigned Integer
EVX	100002F4			SP.FS	efsctui	Convert Floating-Point Single-Precision to Unsigned Integer
EVX	100002F5			SP.FD	efdctsi	Convert Floating-Point Double-Precision to Signed Integer
EVX	100002F5			SP.FS	efsctsi	Convert Floating-Point Single-Precision to Signed Integer
EVX	100002F6			SP.FD	efdctuf	Convert Floating-Point Double-Precision to Unsigned Fraction
EVX	100002F6			SP.FS	efsctuf	Convert Floating-Point Single-Precision to Unsigned Fraction
EVX	100002F7			SP.FD	efdctsf	Convert Floating-Point Double-Precision to Signed Fraction
EVX	100002F7			SP.FS	efsctsf	Convert Floating-Point Single-Precision to Signed Fraction

Form	Opcode (hexadecimal)[2]	Mode Dep.[1]	Priv[1]	Cat[1]	Mnemonic	Instruction
EVX	100002F8			SP.FD	efdctuiz	Convert Floating-Point Double-Precision to Unsigned Integer with Round Towards Zero
EVX	100002F8			SP.FS	efsctuiz	Convert Floating-Point Single-Precision to Unsigned Integer with Round Towards Zero
EVX	100002FA			SP.FD	efdctsiz	Convert Floating-Point Double-Precision to Signed Integer with Round Towards Zero
EVX	100002FA			SP.FS	efsctsiz	Convert Floating-Point Single-Precision to Signed Integer with Round Towards Zero
EVX	100002FC			SP.FD	efdtstgt	Floating-Point Double-Precision Test Greater Than
EVX	100002FC			SP.FS	efststgt	Floating-Point Single-Precision Test Greater Than
EVX	100002FD			SP.FD	efdtstlt	Floating-Point Double-Precision Test Less Than
EVX	100002FD			SP.FS	efststlt	Floating-Point Single-Precision Test Less Than
EVX	100002FE			SP.FD	efdtsteq	Floating-Point Double-Precision Test Equal
EVX	100002FE			SP.FS	efststeq	Floating-Point Single-Precision Test Equal
EVX	10000300			SP	evlddx	Vector Load Doubleword into Doubleword Indexed
VX	10000300			V	vaddsbs	Vector Add Signed Byte Saturate
EVX	10000301			SP	evldd	Vector Load Doubleword into Doubleword
EVX	10000302			SP	evldwx	Vector Load Doubleword into 2 Words Indexed
VX	10000302			V	vminsb	Vector Minimum Signed Byte
EVX	10000303			SP	evldw	Vector Load Doubleword into 2 Words
EVX	10000304			SP	evldhx	Vector Load Doubleword into 4 Halfwords Indexed
VX	10000304			V	vsrab	Vector Shift Right Algebraic Word
EVX	10000305			SP	evldh	Vector Load Doubleword into 4 Halfwords
VC	10000306			V	vcmpgtsb[.]	Vector Compare Greater Than Signed Byte
EVX	10000308			SP	evlhhesplatx	Vector Load Halfword into Halfwords Even and Splat Indexed
VX	10000308			V	vmulesb	Vector Multiply Even Signed Byte
EVX	10000309			SP	evlhhesplat	Vector Load Halfword into Halfwords Even and Splat
VX	1000030A			V	vcuxwfp	Vector Convert from Unsigned Fixed-Point Word to Single-Precision
EVX	1000030C			SP	evlhhousplatx	Vector Load Halfword into Halfwords Odd Unsigned and Splat Indexed
VX	1000030C			V	vspltisb	Vector Splat Immediate Signed Byte
EVX	1000030D			SP	evlhhousplat	Vector Load Halfword into Halfwords Odd Unsigned and Splat
EVX	1000030E			SP	evlhhossplatx	Vector Load Halfword into Halfwords Odd Signed and Splat Indexed
VX	1000030E			V	vpkpx	Vector Pack Pixel
EVX	1000030F			SP	evlhhossplat	Vector Load Halfword into Halfwords Odd and Splat
EVX	10000310			SP	evlwhex	Vector Load Word into Two Halfwords Even Indexed
EVX	10000311			SP	evlwhe	Vector Load Word into Two Halfwords Even
EVX	10000314			SP	evlwhoux	Vector Load Word into Two Halfwords Odd Unsigned Indexed (zero-extended)
EVX	10000315			SP	evlwhou	Vector Load Word into Two Halfwords Odd Unsigned (zero-extended)
EVX	10000316			SP	evlwhosx	Vector Load Word into Two Halfwords Odd Signed Indexed (with sign extension)
EVX	10000317			SP	evlwhos	Vector Load Word into Two Halfwords Odd Signed (with sign extension)
EVX	10000318			SP	evlwwsplatx	Vector Load Word into Word and Splat Indexed
X	10000318	SR		LIM	maclhwu[o][.]	Multiply Accumulate Low Halfword to Word Modulo Unsigned
EVX	10000319			SP	evlwwsplat	Vector Load Word into Word and Splat
EVX	1000031C			SP	evlwhsplatx	Vector Load Word into Two Halfwords and Splat Indexed
EVX	1000031D			SP	evlwhsplat	Vector Load Word into Two Halfwords and Splat
EVX	10000320			SP	evstddx	Vector Store Doubleword of Doubleword Indexed
EVX	10000321			SP	evstdd	Vector Store Doubleword of Doubleword
EVX	10000322			SP	evstdwx	Vector Store Doubleword of Two Words Indexed
EVX	10000323			SP	evstdw	Vector Store Doubleword of Two Words
EVX	10000324			SP	evstdhx	Vector Store Doubleword of Four Halfwords Indexed

Form	Opcode (hexadecimal)[2]	Mode Dep.[1]	Priv[1]	Cat[1]	Mnemonic	Instruction
EVX	10000325			SP	evstdh	Vector Store Doubleword of Four Halfwords
EVX	10000330			SP	evstwhex	Vector Store Word of Two Halfwords from Even Indexed
EVX	10000331			SP	evstwhe	Vector Store Word of Two Halfwords from Even
EVX	10000334			SP	evstwhox	Vector Store Word of Two Halfwords from Odd Indexed
EVX	10000335			SP	evstwho	Vector Store Word of Two Halfwords from Odd
EVX	10000338			SP	evstwwex	Vector Store Word of Word from Even Indexed
EVX	10000339			SP	evstwwe	Vector Store Word of Word from Even
EVX	1000033C			SP	evstwwox	Vector Store Word of Word from Odd Indexed
EVX	1000033D			SP	evstwwo	Vector Store Word of Word from Odd
VX	10000340			V	vaddshs	Vector Add Signed Halfword Saturate
VX	10000342			V	vminsh	Vector Minimum Signed Halfword
VX	10000344			V	vsrah	Vector Shift Right Algebraic Word
VC	10000346			V	vcmpgtsh[.]	Vector Compare Greater Than Signed Halfword
VX	10000348			V	vmulesh	Vector Multiply Even Signed Halfword
VX	1000034A			V	vcsxwfp	Vector Convert from Signed Fixed-Point Word to Single-Precision
VX	1000034C			V	vspltish	Vector Splat Immediate Signed Halfword
VX	1000034E			V	vupkhpx	Vector Unpack High Pixel
X	10000358	SR		LIM	maclhw[o][.]	Multiply Accumulate Low Halfword to Word Modulo Signed
X	1000035C	SR		LIM	nmaclhw[o][.]	Negative Multiply Accumulate Low Halfword to Word Modulo Signed
VX	10000380			V	vaddsws	Vector Add Signed Word Saturate
VX	10000382			V	vminsw	Vector Minimum Signed Word
VX	10000384			V	vsraw	Vector Shift Right Algebraic Word
VC	10000386			V	vcmpgtsw[.]	Vector Compare Greater Than Signed Word
VX	1000038A			V	vcfpuxws	Vector Convert from Single-Precision to Unsigned Fixed-Point Word Saturate
VX	1000038C			V	vspltisw	Vector Splat Immediate Signed Word
X	10000398	SR		LIM	maclhwsu[o][.]	Multiply Accumulate Low Halfword to Word Saturate Unsigned
VC	100003C6			V	vcmpbfp[.]	Vector Compare Bounds Single-Precision
VX	100003CA			V	vcfpsxws	Vector Convert from Single-Precision to Signed Fixed-Point Word Saturate
VX	100003CE			V	vupklpx	Vector Unpack Low Pixel
X	100003D8	SR		LIM	maclhws[o][.]	Multiply Accumulate Low Halfword to Word Saturate Signed
X	100003DC	SR		LIM	nmaclhws[o][.]	Negative Multiply Accumulate Low Halfword to Word Saturate Signed
VX	10000400			V	vsububm	Vector Subtract Unsigned Byte Modulo
VX	10000402			V	vavgub	Vector Average Unsigned Byte
EVX	10000403			SP	evmhessf	Vector Multiply Halfwords, Even, Signed, Saturate, Fractional
VX	10000404			V	vand	Vector AND
EVX	10000407			SP	evmhossf	Vector Multiply Halfwords, Odd, Signed, Fractional
EVX	10000408			SP	evmheumi	Vector Multiply Halfwords, Even, Unsigned, Modulo, Integer
EVX	10000409			SP	evmhesmi	Vector Multiply Halfwords, Even, Signed, Modulo, Integer
VX	1000040A			V	vmaxfp	Vector Maximum Single-Precision
EVX	1000040B			SP	evmhesmf	Vector Multiply Halfwords, Even, Signed, Modulo, Fractional
EVX	1000040C			SP	evmhoumi	Vector Multiply Halfwords, Odd, Unsigned, Modulo, Integer
VX	1000040C			V	vslo	Vector Shift Left by Octet
EVX	1000040D			SP	evmhosmi	Vector Multiply Halfwords, Odd, Signed, Modulo, Integer
EVX	1000040F			SP	evmhosmf	Vector Multiply Halfwords, Odd, Signed, Modulo, Fractional
EVX	10000423			SP	evmhessfa	Vector Multiply Halfwords, Even, Signed, Saturate, Fractional to Accumulator
EVX	10000427			SP	evmhossfa	Vector Multiply Halfwords, Odd, Signed, Fractional to Accumulator
EVX	10000428			SP	evmheumia	Vector Multiply Halfwords, Even, Unsigned, Modulo, Integer to Accumulator
EVX	10000429			SP	evmhesmia	Vector Multiply Halfwords, Even, Signed, Modulo, Integer to Accumulator
EVX	1000042B			SP	evmhesmfa	Vector Multiply Halfwords, Even, Signed, Modulo, Fractional to Accumulate

Form	Opcode (hexadeci-mal)[2]	Mode Dep.[1]	Priv[1]	Cat[1]	Mnemonic	Instruction
EVX	1000042C			SP	evmhoumia	Vector Multiply Halfwords, Odd, Unsigned, Modulo, Integer to Accumulator
EVX	1000042D			SP	evmhosmia	Vector Multiply Halfwords, Odd, Signed, Modulo, Integer to Accumulator
EVX	1000042F			SP	evmhosmfa	Vector Multiply Halfwords, Odd, Signed, Modulo, Fractional to Accumulator
VX	10000440			V	vsubuhm	Vector Subtract Unsigned Byte Modulo
VX	10000442			V	vavguh	Vector Average Unsigned Halfword
VX	10000444			V	vandc	Vector AND with Complement
EVX	10000447			SP	evmwhssf	Vector Multiply Word High Signed, Fractional
EVX	10000448			SP	evmwlumi	Vector Multiply Word Low Unsigned, Modulo, Integer
VX	1000044A			V	vminfp	Vector Minimum Single-Precision
EVX	1000044C			SP	evmwhumi	Vector Multiply Word High Unsigned, Modulo, Integer
VX	1000044C			V	vsro	Vector Shift Right by Octet
EVX	1000044D			SP	evmwhsmi	Vector Multiply Word High Signed, Modulo, Integer
EVX	1000044F			SP	evmwhsmf	Vector Multiply Word High Signed, Modulo, Fractional
EVX	10000453			SP	evmwssf	Vector Multiply Word Signed, Saturate, Fractional
EVX	10000458			SP	evmwumi	Vector Multiply Word Unsigned, Modulo, Integer
EVX	10000459			SP	evmwsmi	Vector Multiply Word Signed, Modulo, Integer
EVX	1000045B			SP	evmwsmf	Vector Multiply Word Signed, Modulo, Fractional
EVX	10000467			SP	evmwhssfa	Vector Multiply Word High Signed, Fractional to Accumulator
EVX	10000468			SP	evmwlumia	Vector Multiply Word Low Unsigned, Modulo, Integer to Accumulator
EVX	1000046C			SP	evmwhumia	Vector Multiply Word High Unsigned, Modulo, Integer to Accumulator
EVX	1000046D			SP	evmwhsmia	Vector Multiply Word High Signed, Modulo, Integer to Accumulator
EVX	1000046F			SP	evmwhsmfa	Vector Multiply Word High Signed, Modulo, Fractional to Accumulator
EVX	10000473			SP	evmwssfa	Vector Multiply Word Signed, Saturate, Fractional to Accumulator
EVX	10000478			SP	evmwumia	Vector Multiply Word Unsigned, Modulo, Integer to Accumulator
EVX	10000479			SP	evmwsmia	Vector Multiply Word Signed, Modulo, Integer to Accumulator
EVX	1000047B			SP	evmwsmfa	Vector Multiply Word Signed, Modulo, Fractional to Accumulator
VX	10000480			V	vsubuwm	Vector Subtract Unsigned Word Modulo
VX	10000482			V	vavguw	Vector Average Unsigned Word
VX	10000484			V	vor	Vector OR
EVX	100004C0			SP	evaddusiaaw	Vector Add Unsigned, Saturate, Integer to Accumulator Word
EVX	100004C1			SP	evaddssiaaw	Vector Add Signed, Saturate, Integer to Accumulator Word
EVX	100004C2			SP	evsubfusiaaw	Vector Subtract Unsigned, Saturate, Integer to Accumulator Word
EVX	100004C3			SP	evsubfssiaaw	Vector Subtract Signed, Saturate, Integer to Accumulator Word
EVX	100004C4			SP	evmra	Initialize Accumulator
VX	100004C4			V	vxor	Vector XOR
EVX	100004C6			SP	evdivws	Vector Divide Word Signed
EVX	100004C7			SP	evdivwu	Vector Divide Word Unsigned
EVX	100004C8			SP	evaddumiaaw	Vector Add Unsigned, Modulo, Integer to Accumulator Word
EVX	100004C9			SP	evaddsmiaaw	Vector Add Signed, Modulo, Integer to Accumulator Word
EVX	100004CA			SP	evsubfumiaaw	Vector Subtract Unsigned, Modulo, Integer to Accumulator Word
EVX	100004CB			SP	evsubfsmiaaw	Vector Subtract Signed, Modulo, Integer to Accumulator Word
EVX	10000500			SP	evmheusiaaw	Vector Multiply Halfwords, Even, Unsigned, Saturate Integer and Accumulate into Words

Form	Opcode (hexadecimal)[2]	Mode Dep.[1]	Priv[1]	Cat[1]	Mnemonic	Instruction
EVX	10000501			SP	evmhessiaaw	Vector Multiply Halfwords, Even, Signed, Saturate, Integer and Accumulate into Words
VX	10000502			V	vavgsb	Vector Average Signed Byte
EVX	10000503			SP	evmhessfaaw	Vector Multiply Halfwords, Even, Signed, Saturate, Fractional and Accumulate into Words
EVX	10000504			SP	evmhousiaaw	Vector Multiply Halfwords, Odd, Unsigned, Saturate, Integer and Accumulate into Words
VX	10000504			V	vnor	Vector NOR
EVX	10000505			SP	evmhossiaaw	Vector Multiply Halfwords, Odd, Signed, Saturate, Integer and Accumulate into Words
EVX	10000507			SP	evmhossfaaw	Vector Multiply Halfwords, Odd, Signed, Saturate, Fractional and Accumulate into Words
EVX	10000508			SP	evmheumiaaw	Vector Multiply Halfwords, Even, Unsigned, Modulo, Integer and Accumulate into Words
EVX	10000509			SP	evmhesmiaaw	Vector Multiply Halfwords, Even, Signed, Modulo, Integer and Accumulate into Words
EVX	1000050B			SP	evmhesmfaaw	Vector Multiply Halfwords, Even, Signed, Modulo, Fractional and Accumulate into Words
EVX	1000050C			SP	evmhoumiaaw	Vector Multiply Halfwords, Odd, Unsigned, Modulo, Integer and Accumulate into Words
EVX	1000050D			SP	evmhosmiaaw	Vector Multiply Halfwords, Odd, Signed, Modulo, Integer and Accumulate into Words
EVX	1000050F			SP	evmhosmfaaw	Vector Multiply Halfwords, Odd, Signed, Modulo, Fractional and Accumulate into Words
EVX	10000528			SP	evmhegumiaa	Vector Multiply Halfwords, Even, Guarded, Unsigned, Modulo, Integer and Accumulate
EVX	10000529			SP	evmhegsmiaa	Vector Multiply Halfwords, Even, Guarded, Signed, Modulo, Integer and Accumulate
EVX	1000052B			SP	evmhegsmfaa	Vector Multiply Halfwords, Even, Guarded, Signed, Modulo, Fractional and Accumulate
EVX	1000052C			SP	evmhogumiaa	Vector Multiply Halfwords, Odd, Guarded, Unsigned, Modulo, Integer and Accumulate
EVX	1000052D			SP	evmhogsmiaa	Vector Multiply Halfwords, Odd, Guarded, Signed, Modulo, Integer and Accumulate
EVX	1000052F			SP	evmhogsmfaa	Vector Multiply Halfwords, Odd, Guarded, Signed, Modulo, Fractional and Accumulate
EVX	10000540			SP	evmwlusiaaw	Vector Multiply Word Low Unsigned Saturate, Integer and Accumulate into Words
EVX	10000541			SP	evmwlssiaaw	Vector Multiply Word Low Signed, Saturate, Integer and Accumulate into Words
VX	10000542			V	vavgsh	Vector Average Signed Halfword
EVX	10000544			SP	evmwhusiaaw	Vector Multiply Word High Unsigned, Integer and Accumulate into Words
EVX	10000547			SP	evmwhssfaaw	Vector Multiply Word High Signed, Fractional and Accumulate into Words
EVX	10000548			SP	evmwlumiaaw	Vector Multiply Word Low Unsigned, Modulo, Integer and Accumulate into Words
EVX	10000549			SP	evmwlsmiaaw	Vector Multiply Word Low Signed, Modulo, Integer and Accumulate into Words
EVX	1000054C			SP	evmwhumiaaw	Vector Multiply Word High Unsigned, Modulo, Integer and Accumulate into Words
EVX	1000054D			SP	evmwhsmiaaw	Vector Multiply Word High Signed, Modulo, Integer and Accumulate into Words
EVX	1000054F			SP	evmwhsmfaaw	Vector Multiply Word High Signed, Modulo, Fractional and Accumulate into Words
EVX	10000553			SP	evmwssfaa	Vector Multiply Word Signed, Saturate, Fractional and Accumulate
EVX	10000558			SP	evmwumiaa	Vector Multiply Word Unsigned, Modulo, Integer and Accumulate

Form	Opcode (hexadecimal)[2]	Mode Dep.[1]	Priv[1]	Cat[1]	Mnemonic	Instruction
EVX	10000559			SP	evmwsmiaa	Vector Multiply Word Signed, Modulo, Integer and Accumulate
EVX	1000055B			SP	evmwsmfaa	Vector Multiply Word Signed, Modulo, Fractional and Accumulate
EVX	10000580			SP	evmheusianw	Vector Multiply Halfwords, Even, Unsigned, Saturate Integer and Accumulate Negative into Words
VX	10000580			V	vsubcuw	Vector Subtract and Write Carry-Out Unsigned Word
EVX	10000581			SP	evmhessianw	Vector Multiply Halfwords, Even, Signed, Saturate, Integer and Accumulate Negative into Words
VX	10000582			V	vavgsw	Vector Average Signed Word
EVX	10000583			SP	evmhessfanw	Vector Multiply Halfwords, Even, Signed, Saturate, Fractional and Accumulate Negative into Words
EVX	10000584			SP	evmhousianw	Vector Multiply Halfwords, Odd, Unsigned, Saturate, Integer and Accumulate Negative into Words
EVX	10000585			SP	evmhossianw	Vector Multiply Halfwords, Odd, Signed, Saturate, Integer and Accumulate Negative into Words
EVX	10000587			SP	evmhossfanw	Vector Multiply Halfwords, Odd, Signed, Saturate, Fractional and Accumulate Negative into Words
EVX	10000588			SP	evmheumianw	Vector Multiply Halfwords, Even, Unsigned, Modulo, Integer and Accumulate Negative into Words
EVX	10000589			SP	evmhesmianw	Vector Multiply Halfwords, Even, Signed, Modulo, Integer and Accumulate Negative into Words
EVX	1000058B			SP	evmhesmfanw	Vector Multiply Halfwords, Even, Signed, Modulo, Fractional and Accumulate Negative into Words
EVX	1000058C			SP	evmhoumianw	Vector Multiply Halfwords, Odd, Unsigned, Modulo, Integer and Accumulate Negative into Words
EVX	1000058D			SP	evmhosmianw	Vector Multiply Halfwords, Odd, Signed, Modulo, Integer and Accumulate Negative into Words
EVX	1000058F			SP	evmhosmfanw	Vector Multiply Halfwords, Odd, Signed, Modulo, Fractional and Accumulate Negative into Words
EVX	100005A8			SP	evmhegumian	Vector Multiply Halfwords, Even, Guarded, Unsigned, Modulo, Integer and Accumulate Negative
EVX	100005A9			SP	evmhegsmian	Vector Multiply Halfwords, Even, Guarded, Signed, Modulo, Integer and Accumulate Negative
EVX	100005AB			SP	evmhegsmfan	Vector Multiply Halfwords, Even, Guarded, Signed, Modulo, Fractional and Accumulate Negative
EVX	100005AC			SP	evmhogumian	Vector Multiply Halfwords, Odd, Guarded, Unsigned, Modulo, Integer and Accumulate Negative
EVX	100005AD			SP	evmhogsmian	Vector Multiply Halfwords, Odd, Guarded, Signed, Modulo, Integer and Accumulate Negative
EVX	100005AF			SP	evmhogsmfan	Vector Multiply Halfwords, Odd, Guarded, Signed, Modulo, Fractional and Accumulate Negative
EVX	100005C0			SP	evmwlusianw	Vector Multiply Word Low Unsigned Saturate, Integer and Accumulate Negative into Words
EVX	100005C1			SP	evmwlssianw	Vector Multiply Word Low Signed, Saturate, Integer and Accumulate Negative into Words
EVX	100005C4			SP	evmwhusianw	Vector Multiply Word High Unsigned, Integer and Accumulate Negative into Words
EVX	100005C5			SP	evmwhssianw	Vector Multiply Word High Signed, Integer and Accumulate Negative into Words
EVX	100005C7			SP	evmwhssfanw	Vector Multiply Word High Signed, Fractional and Accumulate Negative into Words
EVX	100005C8			SP	evmwlumianw	Vector Multiply Word Low Unsigned, Modulo, Integer and Accumulate Negative into Words
EVX	100005C9			SP	evmwlsmianw	Vector Multiply Word Low Signed, Modulo, Integer and Accumulate Negative into Words
EVX	100005CC			SP	evmwhumianw	Vector Multiply Word High Unsigned, Modulo, Integer and Accumulate Negative into Words

Form	Opcode (hexadeci-mal)[2]	Mode Dep.[1]	Priv[1]	Cat[1]	Mnemonic	Instruction
EVX	100005CD			SP	evmwhsmianw	Vector Multiply Word High Signed, Modulo, Integer and Accumulate Negative into Words
EVX	100005CF			SP	evmwhsmfanw	Vector Multiply Word High Signed, Modulo, Fractional and Accumulate Negative into Words
EVX	100005D3			SP	evmwssfan	Vector Multiply Word Signed, Saturate, Fractional and Accumulate Negative
EVX	100005D8			SP	evmwumian	Vector Multiply Word Unsigned, Modulo, Integer and Accumulate Negative
EVX	100005D9			SP	evmwsmian	Vector Multiply Word Signed, Modulo, Integer and Accumulate Negative
EVX	100005DB			SP	evmwsmfan	Vector Multiply Word Signed, Modulo, Fractional and Accumulate Negative
VX	10000600			V	vsububs	Vector Subtract Unsigned Byte Saturate
VX	10000604			V	mfvscr	Move from Vector Status and Control Register
VX	10000608			V	vsum4ubs	Vector Sum across Quarter Unsigned Byte Saturate
VX	10000640			V	vsubuhs	Vector Subtract Unsigned Halfword Saturate
VX	10000644			V	mtvscr	Move to Vector Status and Control Register
VX	10000648			V	vsum4shs	Vector Sum across Quarter Signed Halfword Saturate
VX	10000680			V	vsubuws	Vector Subtract Unsigned Word Saturate
VX	10000688			V	vsum2sws	Vector Sum across Half Signed Word Saturate
VX	10000700			V	vsubsbs	Vector Subtract Signed Byte Saturate
VX	10000708			V	vsum4sbs	Vector Sum across Quarter Signed Byte Saturate
VX	10000740			V	vsubshs	Vector Subtract Signed Halfword Saturate
VX	10000780			V	vsubsws	Vector Subtract Signed Word Saturate
VX	10000788			V	vsumsws	Vector Sum across Signed Word Saturate
D8	18000000			VLE	e_lbzu	Load Byte and Zero with Update
D8	18000100			VLE	e_lhzu	Load Halfword and Zero with Update
D8	18000200			VLE	e_lwzu	Load Word and Zero with Update
D8	18000300			VLE	e_lhau	Load Halfword Algebraic with Update
D8	18000400			VLE	e_stbu	Store Byte with Update
D8	18000500			VLE	e_sthu	Store Halfword with Update
D8	18000600			VLE	e_stwu	Store word with Update
D8	18000800			VLE	e_lmw	Load Multiple Word
D8	18000900			VLE	e_stmw	Store Multiple Word
SCI8	18008000	SR		VLE	e_addi[.]	Add Scaled Immediate
SCI8	18009000	SR		VLE	e_addic[.]	Add Scaled Immediate Carrying
SCI8	1800A000			VLE	e_mulli	Multiply Low Scaled Immediate
SCI8	1800A800			VLE	e_cmpi	Compare Scaled Immediate Word
SCI8	1800B000	SR		VLE	e_subfic[.]	Subtract From Scaled Immediate Carrying
SCI8	1800C000	SR		VLE	e_andi[.]	AND Scaled Immediate
SCI8	1800D000	SR		VLE	e_ori[.]	OR Scaled Immediate
SCI8	1800E000	SR		VLE	e_xori[.]	XOR Scaled Immediate
SCI8	1880A800			VLE	e_cmpli	Compare Logical Scaled Immediate Word
D	1C000000			VLE	e_add16i	Add Immediate
OIM5	2000----			VLE	se_addi	Add Immediate Short Form
OIM5	2200----			VLE	se_cmpli	Compare Logical Immediate Word
OIM5	2400----	SR		VLE	se_subi[.]	Subtract Immediate
IM5	2A00----			VLE	se_cmpi	Compare Immediate Word Short Form
IM5	2C00----			VLE	se_bmaski	Bit Mask Generate Immediate
IM5	2E00----			VLE	se_andi	AND Immediate Short Form
D	30000000			VLE	e_lbz	Load Byte and Zero
D	34000000			VLE	e_stb	Store Byte
D	38000000			VLE	e_lha	Load Halfword Algebraic
RR	4000----			VLE	se_srw	Shift Right Word
RR	4100----	SR		VLE	se_sraw	Shift Right Algebraic Word
RR	4200----			VLE	se_slw	Shift Left Word
RR	4400----			VLE	se_or	OR SHort Form
RR	4500----			VLE	se_andc	AND with Complement Short Form
RR	4600----	SR		VLE	se_and[.]	AND Short Form
IM7	4800----			VLE	se_li	Load Immediate Short Form
D	50000000			VLE	e_lwz	Load Word and Zero

Form	Opcode (hexadecimal)[2]	Mode Dep.[1]	Priv[1]	Cat[1]	Mnemonic	Instruction
D	54000000			VLE	e_stw	Store Word
D	58000000			VLE	e_lhz	Load Halfword and Zero
D	5C000000			VLE	e_sth	Store Halfword
IM5	6000----			VLE	se_bclri	Bit Clear Immediate
IM5	6200----			VLE	se_bgeni	Bit Generate Immediate
IM5	6400----			VLE	se_bseti	Bit Set Immediate
IM5	6600----			VLE	se_btsti	Bit Test Immediate
IM5	6800----			VLE	se_srwi	Shift Right Word Immediate Short Form
IM5	6A00----	SR		VLE	se_srawi	Shift Right Algebraic Immediate
IM5	6C00----			VLE	se_slwi	Shift Left Word Immediate Short Form
LI20	70000000			VLE	e_li	Load Immediate
I16A	70008800	SR		VLE	e_add2i.	Add (2 operand) Immediate and Record
I16A	70009000			VLE	e_add2is	Add (2 operand) Immediate Shifted
IA16	70009800			VLE	e_cmp16i	Compare Immediate Word
I16A	7000A000			VLE	e_mull2i	Multiply (2 operand) Low Immediate
I16A	7000A800			VLE	e_cmpl16i	Compare Logical Immediate Word
IA16	7000B000			VLE	e_cmph16i	Compare Halfword Immediate
IA16	7000B800			VLE	e_cmphl16i	Compare Halfword Logical Immediate
I16L	7000C000			VLE	e_or2i	OR (2operand) Immediate
I16L	7000C800	SR		VLE	e_and2i.	AND (2 operand) Immediate
I16L	7000D000			VLE	e_or2is	OR (2 operand) Immediate Shifted
I16L	7000E000			VLE	e_lis	Load Immediate Shifted
I16L	7000E800	SR		VLE	e_and2is.	AND (2 operand) Immediate Shifted
M	74000000			VLE	e_rlwimi	Rotate Left Word Immediate then Mask Insert
M	74000001			VLE	e_rlwinm	Rotate Left Word Immediate then AND with Mask
BD24	78000000			VLE	e_b[l]	Branch [and Link]
BD15	7A000000	CT		VLE	e_bc[l]	Branch Conditional [and Link]
X	7C000000			B	cmp	Compare
X	7C000008			B	tw	Trap Word
X	7C00000C			V	lvsl	Load Vector for Shift Left Indexed
X	7C00000E			V	lvebx	Load Vector Element Byte Indexed
XO	7C000010	SR		B	subfc[o][.]	Subtract From Carrying
XO	7C000012	SR		64	mulhdu[.]	Multiply High Doubleword Unsigned
XO	7C000014			B	addc[o][.]	Add Carrying
XO	7C000016	SR		B	mulhwu[.]	Multiply High Word Unsigned
X	7C00001C			VLE	e_cmph	Compare Halfword
A	7C00001E			B.in	isel	Integer Select
XL	7C000020			VLE	e_mcrf	Move CR Field
XFX	7C000026			B	mfcr	Move From Condition Register
X	7C000028			B	lwarx	Load Word and Reserve Indexed
X	7C00002A			64	ldx	Load Doubleword Indexed
X	7C00002C			E	icbt	Instruction Cache Block Touch
X	7C00002E			B	lwzx	Load Word and Zero Indexed
X	7C000030	SR		B	slw[.]	Shift Left Word
X	7C000034	SR		B	cntlzw[.]	Count Leading Zeros Word
X	7C000036	SR		64	sld[.]	Shift Left Doubleword
X	7C000038	SR		B	and[.]	AND
X	7C00003A		P	E.PD	ldepx	Load Doubleword by External Process ID Indexed
X	7C00003E		P	E.PD	lwepx	Load Word by External Process ID Indexed
X	7C000040			B	cmpl	Compare Logical
XL	7C000042			VLE	e_crnor	Condition Register NOR
X	7C00004C			V	lvsr	Load Vector for Shift Right Indexed
X	7C00004E			V	lvehx	Load Vector Element Halfword Indexed
XO	7C000050	SR		B	subf[o][.]	Subtract From
X	7C00005C			VLE	e_cmphl	Compare Halfword Logical
X	7C00006A			64	ldux	Load Doubleword with Update Indexed
X	7C00006C			B	dcbst	Data Cache Block Store
X	7C00006E			B	lwzux	Load Word and Zero with Update Indexed
X	7C000070	SR		VLE	e_slwi[.]	Shift Left Word Immediate
X	7C000074	SR		64	cntlzd[.]	Count Leading Zeros Doubleword
X	7C000078	SR		B	andc[.]	AND with Complement

Form	Opcode (hexadecimal)[2]	Mode Dep.[1]	Priv[1]	Cat[1]	Mnemonic	Instruction
X	7C00007C			WT	wait	Wait
X	7C000088			64	td	Trap Doubleword
X	7C00008E			V	lvewx	Load Vector Element Word Indexed
XO	7C000092	SR		64	mulhd[.]	Multiply High Doubleword
XO	7C000096	SR		B	mulhw[.]	Multiply High Word
X	7C0000A6		P	B	mfmsr	Move From Machine State Register
X	7C0000A8			64	ldarx	Load Doubleword and Reserve Indexed
X	7C0000AC			B	dcbf	Data Cache Block Flush
X	7C0000AE			B	lbzx	Load Byte and Zero Indexed
X	7C0000BE		P	E.PD	lbepx	Load Byte by External Process ID Indexed
X	7C0000CE			V	lvx[l]	Load Vector Indexed [Last]
X	7C0000D0	SR		B	neg[o][.]	Negate
X	7C0000EE			B	lbzux	Load Byte and Zero with Update Indexed
X	7C0000F4			B	popcntb	Population Count Bytes
X	7C0000F8	SR		B	nor[.]	NOR
X	7C0000FE		P	E.PD	dcbfep	Data Cache Block Flush by External Process ID
XL	7C000102			VLE	e_crandc	Condition Register AND with Completement
X	7C000106		P	E	wrtee	Write MSR External Enable
X	7C00010C		M	ECL	dcbtstls	Data Cache Block Touch for Store and Lock Set
VX	7C00010E			V	stvebx	Store Vector Element Byte Indexed
XO	7C000110	SR		B	subfe[o][.]	Subtract From Extended
XO	7C000114	SR		B	adde[o][.]	Add Extended
EVX	7C00011D		P	E.PD	evlddepx	Vector Load Doubleword into Doubleword by External Process ID Indexed
XFX	7C000120			B	mtcrf	Move To Condition Register Fields
X	7C000124		P	E	mtmsr	Move To Machine State Register
X	7C00012A			64	stdx	Store Doubleword Indexed
X	7C00012D			B	stwcx.	Store Word Conditional Indexed
X	7C00012E			B	stwx	Store Word Indexed
X	7C00013A		P	E.PD	stdepx	Store Doubleword by External Process ID Indexed
X	7C00013E		P	E.PD	stwepx	Store Word by External Process ID Indexed
X	7C000146		P	E	wrteei	Write MSR External Enable Immediate
X	7C00014C		M	ECL	dcbtls	Data Cache Block Touch and Lock Set
VX	7C00014E			V	stvehx	Store Vector Element Halfword Indexed
X	7C00016A			64	stdux	Store Doubleword with Update Indexed
X	7C00016E			B	stwux	Store Word with Update Indexed
XL	7C000182			VLE	e_crxor	Condition Register XOR
VX	7C00018E			V	stvewx	Store Vector Element Word Indexed
XO	7C000190	SR		B	subfze[o][.]	Subtract From Zero Extended
XO	7C000194	SR		B	addze[o][.]	Add to Zero Extended
X	7C00019C		P	E.PC	msgsnd	Message Send
EVX	7C00019D		P	E.PD	evstddepx	Vector Store Doubleword into Doubleword by External Process ID Indexed
X	7C0001AD			64	stdcx.	Store Doubleword Conditional Indexed
X	7C0001AE			B	stbx	Store Bye Indexed
X	7C0001BE		P	E.PD	stbepx	Store Byte by External Process ID Indexed
XL	7C0001C2			VLE	e_crnand	Condition Register NAND
X	7C0001CC		M	ECL	icblc	Instruction Cache Block Lock Clear
VX	7C0001CE			V	stvx[l]	Store Vector Indexed [Last]
XO	7C0001D0	SR		B	subfme[o][.]	Subtract From Minus One Extended
XO	7C0001D2	SR		64	mulld[o][.]	Multiply Low Doubleword
XO	7C0001D4	SR		B	addme[o][.]	Add to Minus One Extended
XO	7C0001D6	SR		B	mullw[o][.]	Multiply Low Word
X	7C0001DC		P	E.PC	msgclr	Message Clear
X	7C0001EC			B	dcbtst	Data Cache Block Touch for Store
X	7C0001EE			B	stbux	Store Byte with Update Indexed
X	7C0001FE		P	E.PD	dcbtstep	Data Cache Block Touch for Store by External Process ID
XL	7C000202			VLE	e_crand	Condition Register AND
XFX	7C000206		P	E	mfdcrx	Move From Device Control Register Indexed
X	7C00020E		P	E.PD	lvepxl	Load Vector by External Process ID Indexed LRU
XO	7C000214			B	add[o][.]	Add

Form	Opcode (hexadecimal)[2]	Mode Dep.[1]	Priv[1]	Cat[1]	Mnemonic	Instruction
X	7C00022C			B	dcbt	Data Cache Block Touch
X	7C00022E			B	lhzx	Load Halfword and Zero Indexed
X	7C000230	SR		VLE	e_rlw[.]	Rotate Left Word
X	7C000238	SR		B	eqv[.]	Equivalent
X	7C00023E		P	E.PD	lhepx	Load Halfword by External Process ID Indexed
XL	7C000242			VLE	e_creqv	Condition Register Equivalent
XFX	7C000246		P	E	mfdcrux	Move From Device Control Register User-mode Indexed
X	7C00024E		P	E.PD	lvepx	Load Vector by External Process ID Indexed
X	7C00026E			B	lhzux	Load Halfword and Zero with Update Indexed
X	7C000270	SR		VLE	e_rlwi[.]	Rotate Left Word Immediate
D	7C000278	SR		B	xor[.]	XOR
X	7C00027E		P	E.PD	dcbtep	Data Cache Block Touch by External Process ID
XFX	7C000286		P	E	mfdcr	Move From Device Control Register
X	7C00028C		P	E.CD	dcread	Data Cache Read
XFX	7C00029C		O	E.PM	mfpmr	Move From Performance Monitor Register
XFX	7C0002A6		O	B	mfspr	Move From Special Purpose Register
X	7C0002AA			64	lwax	Load Word Algebraic Indexed
X	7C0002AE			B	lhax	Load Halfword Algebraic Indexed
X	7C0002EA			64	lwaux	Load Word Algebraic with Update Indexed
X	7C0002EE			B	lhaux	Load Halfword Algebraic with Update Indexed
X	7C000306		P	E	mtdcrx	Move To Device Control Register Indexed
X	7C00030C		M	ECL	dcblc	Data Cache Block Lock Clear
X	7C00032E			B	sthx	Store Halfword Indexed
X	7C000338	SR		B	orc[.]	OR with Complement
X	7C00033E		P	E.PD	sthepx	Store Halfword by External Process ID Indexed
XL	7C000342			VLE	e_crorc	Condition Register OR with Complement
X	7C000346			E	mtdcrux	Move To Device Control Register User-mode Indexed
X	7C00036E			B	sthux	Store Halfword with Update Indexed
X	7C000378	SR		B	or[.]	OR
XL	7C000382			VLE	e_cror	Condition Register OR
XFX	7C000386		P	E	mtdcr	Move To Device Control Register
X	7C00038C		P	E.CI	dci	Data Cache Invalidate
XO	7C000392	SR		64	divdu[o][.]	Divide Doubleword Unsigned
XO	7C000396	SR		B	divwu[o][.]	Divide Word Unsigned
XFX	7C00039C		O	E.PM	mtpmr	Move To Performance Monitor Register
XFX	7C0003A6		O	B	mtspr	Move To Special Purpose Register
X	7C0003AC		P	E	dcbi	Data Cache Block Invalidate
X	7C0003B8	SR		B	nand[.]	NAND
X	7C0003CC		M	ECL	icbtls	Instruction Cache Block Touch and Lock Set
X	7C0003CC		P	E.CD	dcread	Data Cache Read
XO	7C0003D2	SR		64	divd[o][.]	Divide Doubleword
XO	7C0003D6	SR		B	divw[o][.]	Divide Word
X	7C000400			B	mcrxr	Move To Condition Register From XER
X	7C00042A			MA	lswx	Load String Word Indexed
X	7C00042C			B	lwbrx	Load Word Byte-Reversed Indexed
X	7C000430	SR		B	srw[.]	Shift Right Word
X	7C000436	SR		64	srd[.]	Shift Right Doubleword
X	7C00046C		P	E	tlbsync	TLB Synchronize
X	7C000470	SR		VLE	e_srwi[.]	Shift Right Word Immediate
X	7C0004AA			MA	lswi	Load String Word Immediate
X	7C0004AC			B	sync	Synchronize
X	7C0004BE		P	E.PD	lfdepx	Load Floating-Point Double by External Process ID Indexed
X	7C00052A			MA	stswx	Store String Word Indexed
X	7C00052C			B	stwbrx	Store Word Byte-Reversed Indexed
X	7C0005AA			MA	stswi	Store String Word Immediate
X	7C0005BE		P	E.PD	stfdepx	Store Floating-Point Double by External Process ID Indexed
X	7C0005EC			E	dcba	Data Cache Block Allocate
X	7C00060E		P	E.PD	stvepxl	Store Vector by External Process ID Indexed LRU
X	7C000624		P	E	tlbivax	TLB Invalidate Virtual Address Indexed
X	7C00062C			B	lhbrx	Load Halfword Byte-Reversed Indexed
X	7C000630	SR		B	sraw[.]	Shift Right Algebraic Word

Form	Opcode (hexadecimal)[2]	Mode Dep.[1]	Priv[1]	Cat[1]	Mnemonic	Instruction
X	7C000634	SR		64	srad[.]	Shift Right Algebraic Doubleword
X	7C00064E		P	E.PD	stvepx	Store Vector by External Process ID Indexed
X	7C000670	SR		B	srawi[.]	Shift Right Algebraic Word Immediate
X	7C000674	SR		64	sradi[.]	Shift Right Algebraic Doubleword Immediate
XFX	7C0006AC			E	mbar	Memory Barrier
X	7C000724		P	E	tlbsx	TLB Search Indexed
X	7C00072C			B	sthbrx	Store Halfword Byte-Reversed Indexed
X	7C000734	SR		B	extsh[.]	Extend Sign Halfword
X	7C000764		P	E	tlbre	TLB Read Entry
X	7C000774	SR		B	extsb[.]	Extend Shign Byte
X	7C00078C		P	E.CI	ici	Instruction Cache Invalidate
X	7C0007A4		P	E	tlbwe	TLB Write Entry
X	7C0007AC			B	icbi	Instruction Cache Block Invalidate
X	7C0007B4	SR		64	extsw[.]	Extend Sign Word
X	7C0007BE		P	E.PD	icbiep	Instruction Cache Block Invalidate by External Process ID
X	7C0007CC		P	E.CD	icread	Instruction Cache Read
X	7C0007EC			B	dcbz	Data Cache Block set to Zero
X	7C0007FE		P	E.PD	dcbzep	Data Cache Block set to Zero by External Process ID
XFX	7C100026			B	mfocrf	Move From One Condition Register Field
XFX	7C100120			B	mtocrf	Move To One Condition Register Field
SD4	8000----			VLE	se_lbz	Load Byte and Zero Short Form
SD4	9000----			VLE	se_stb	Store Byte Short Form
SD4	A000----			VLE	se_lhz	Load Halfword and Zero Short Form
SD4	B000----			VLE	se_sth	Store Halfword SHort Form
SD4	C000----			VLE	se_lwz	Load Word and Zero Short Form
SD4	D000----			VLE	se_stw	Store Word Short Form
BD8	E000----			VLE	se_bc	Branch Conditional Short Form
BD8	E800----			VLE	se_b[l]	Branch [and Link]

[1] See the key to the mode dependency and privilege column below and the key to the category column in Section 1.3.5 of Book I.

[2]For 16-bit instructions, the “Opcode” column represents the 16-bit hexadecimal instruction encoding with the opcode and extended opcode in the corresponding fields in the instruction, and with 0’s in bit positions which are not opcode bits; dashes are used following the opcode to indicate the form is a 16-bit instruction. For 32-bit instructions, the “Opcode” column represents the 32-bit hexadecimal instruction encoding with the opcode and extended opcode in the corresponding fields in the instruction, and with 0’s in bit positions which are not opcode bits.

Mode Dependency and Privilege Abbreviations

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.

Mode Dep.	Description
CT	If the instruction tests the Count Register, it tests the low-order 32 bits in 32-bit mode and all 64 bits in 64-bit mode.
SR	The setting of status registers (such as XER and CR0) is mode-dependent.
32	The instruction must be executed only in 32-bit mode.
64	The instruction must be executed only in 64-bit mode.

Key to Privilege Column

Priv.	Description
P	Denotes a privileged instruction.

Priv.	Description
O	Denotes an instruction that is treated as privileged or nonprivileged (or hypervisor, for ***mtspr***), depending on the SPR number.
M	Denotes an instruction that is treated as privileged or nonprivileged, depending on the value of the UCLE bit of the MSR.
H	Denotes an instruction that can be executed only in hypervisor state.

Appendices:

Power ISA Books I-III Appendices

Appendix A. Incompatibilities with the POWER Architecture

This appendix identifies the known incompatibilities that must be managed in the migration from the POWER Architecture to the Power ISA. Some of the incompatibilities can, at least in principle, be detected by the processor, which could trap and let software simulate the POWER operation. Others cannot be detected by the processor even in principle.

In general, the incompatibilities identified here are those that affect a POWER application program; incompatibilities for instructions that can be used only by POWER system programs are not necessarily discussed.

A.1 New Instructions, Formerly Privileged Instructions

Instructions new to Power ISA typically use opcode values (including extended opcode) that are illegal in POWER. A few instructions that are privileged in POWER (e.g., ***dclz***, called ***dcbz*** in Power ISA) have been made nonprivileged in Power ISA. Any POWER program that executes one of these now-valid or now-nonprivileged instructions, expecting to cause the system illegal instruction error handler or the system privileged instruction error handler to be invoked, will not execute correctly on Power ISA.

A.2 Newly Privileged Instructions

The following instructions are nonprivileged in POWER but privileged in Power ISA.

mfmsr
mfsr

A.3 Reserved Fields in Instructions

These fields are shown with “/”s in the instruction layouts. In both POWER and Power ISA these fields are ignored by the processor. The Power ISA states that these fields must contain zero. The POWER Architecture lacks such a statement, but it is expected that essentially all POWER programs contain zero in these fields.

In several cases the Power ISA assumes that reserved fields in POWER instructions indeed contain zero. The cases include the following.

- ***bclr***[***l***] and ***bcctr***[***l***] assume that bits 19:20 in the POWER instructions contain zero.
- ***cmpi***, ***cmp***, ***cmpli***, and ***cmpl*** assume that bit 10 in the POWER instructions contains zero.
- ***mtspr*** and ***mfspr*** assume that bits 16:20 in the POWER instructions contain zero.
- ***mtcrf*** and ***mfcr*** assume that bit 11 in the POWER instructions is contains zero.
- *Synchronize* assumes that bits 9:10 in the POWER instruction (***dcs***) contain zero. (This assumption provides compatibility for application programs, but not necessarily for operating system programs; see Section A.22.)
- ***mtmsr*** assumes that bit 15 in the POWER instruction contains zero.

A.4 Reserved Bits in Registers

Both POWER and Power ISA permit software to write any value to these bits. However in POWER reading such a bit always returns 0, while in Power ISA reading it may return either 0 or the value that was last written to it.

A.5 Alignment Check

The POWER MSR AL bit (bit 24) is no longer supported; the corresponding Power ISA MSR bit, bit 56, is reserved. The low-order bits of the EA are always used. (Notice that the value 0 — the normal value for a reserved bit —- means “ignore the low-order EA bits” in POWER, and the value 1 means “use the low-order EA bits".) POWER-compatible operating system code will probably write the value 1 to this bit.

A.6 Condition Register

The following instructions specify a field in the CR explicitly (via the BF field) and also, in POWER, use bit 31 as the Record bit. In Power ISA, bit 31 is a reserved field for these instructions and is ignored by the processor. In POWER, if bit 31 contains 1 the instructions execute normally (i.e., as if the bit contained 0) except as follows:

cmp	CR0 is undefined if Rc=1 and BF≠0
cmpl	CR0 is undefined if Rc=1 and BF≠0
mcrxr	CR0 is undefined if Rc=1 and BF≠0
fcmpu	CR1 is undefined if Rc=1
fcmpo	CR1 is undefined if Rc=1
mcrfs	CR1 is undefined if Rc=1 and BF≠1

A.7 LK and Rc Bits

For the instructions listed below, if bit 31 (LK or Rc bit in POWER) contains 1, in POWER the instruction executes as if the bit contained 0 except as follows: if LK=1, the Link Register is set (to an undefined value, except for ***svc***); if Rc=1, Condition Register Field 0 or 1 is set to an undefined value. In Power ISA, bit 31 is a reserved field for these instructions and is ignored by the processor.

Power ISA instructions for which bit 31 is the LK bit in POWER:

sc (***svc*** in POWER)
the *Condition Register Logical* instructions
mcrf
isync (***ics*** in POWER)

Power ISA instructions for which bit 31 is the Rc bit in POWER:

fixed-point X-form *Load* and *Store* instructions
fixed-point X-form *Compare* instructions
the X-form *Trap* instruction
mtspr, ***mfspr***, ***mtcrf***, ***mcrxr***, ***mfcr***, ***mtocrf***, ***mfocrf***
floating-point X-form *Load* and *Store* instructions
floating-point *Compare* instructions
mcrfs
dcbz (***dclz*** in POWER)

A.8 BO Field

POWER shows certain bits in the BO field — used by *Branch Conditional* instructions — as "x". Although the POWER Architecture does not say how these bits are to be interpreted, they are in fact ignored by the processor.

Power ISA shows these bits as "z", "a", or "t". The "z" bits are ignored, as in POWER. However, the "a" and "t" bits can be used by software to provide a hint about how the branch is likely to behave. If a POWER program has the "wrong" value for these bits, the program will produce the same results as on POWER but performance may be affected.

A.9 BH Field

Bits 19:20 of the *Branch Conditional to Link Register* and *Branch Conditional to Count Register* instructions are reserved in POWER but are defined as a branch hint (BH) field in Power ISA. Because these bits are hints, they may affect performance but do not affect the results of executing the instruction.

A.10 Branch Conditional to Count Register

For the case in which the Count Register is decremented and tested (i.e., the case in which BO_2=0), POWER specifies only that the branch target address is undefined, with the implication that the Count Register, and the Link Register if LK=1, are updated in the normal way. Power ISA specifies that this instruction form is invalid.

A.11 System Call

There are several respects in which Power ISA is incompatible with POWER for *System Call* instructions — which in POWER are called *Supervisor Call* instructions.

- POWER provides a version of the *Supervisor Call* instruction (bit 30 = 0) that allows instruction fetching to continue at any one of 128 locations. It is used for "fast SVCs". Power ISA provides no such version: if bit 30 of the instruction is 0 the instruction form is invalid.
- POWER provides a version of the *Supervisor Call* instruction (bits 30:31 = 0b11) that resumes instruction fetching at one location and sets the Link Register to the address of the next instruction. Power ISA provides no such version: bit 31 is a reserved field.
- For POWER, information from the MSR is saved in the Count Register. For Power ISA this information is saved in SRR1.
- In POWER bits 16:19 and 27:29 of the instruction comprise defined instruction fields or a portion thereof, while in Power ISA these bits comprise reserved fields.

- In POWER bits 20:26 of the instruction comprise a portion of the SV field, while in Power ISA these bits comprise the LEV field.
- POWER saves the low-order 16 bits of the instruction, in the Count Register. Power ISA does not save them.
- The settings of MSR bits by the associated interrupt differ between POWER and Power ISA; see *POWER Processor Architecture* and Book III.

A.12 Fixed-Point Exception Register (XER)

Bits 48:55 of the XER are reserved in Power ISA, while in POWER the corresponding bits (16:23) are defined and contain the comparison byte for the ***lscbx*** instruction (which Power ISA lacks).

A.13 Update Forms of Storage Access Instructions

Power ISA requires that RA not be equal to either RT (fixed-point *Load* only) or 0. If the restriction is violated the instruction form is invalid. POWER permits these cases, and simply avoids saving the EA.

A.14 Multiple Register Loads

Power ISArequires that RA, and RB if present in the instruction format, not be in the range of registers to be loaded, while POWER permits this and does not alter RA or RB in this case. (The Power ISA restriction applies even if RA=0, although there is no obvious benefit to the restriction in this case since RA is not used to compute the effective address if RA=0.) If the Power ISA restriction is violated, either the system illegal instruction error handler is invoked or the results are boundedly undefined. The instructions affected are:

lmw (***lm*** in POWER)
lswi (***lsi*** in POWER)
lswx (***lsx*** in POWER)

For example, an ***lmw*** instruction that loads all 32 registers is valid in POWER but is an invalid form in Power ISA.

A.15 Load/Store Multiple Instructions

There are two respects in which Power ISA is incompatible with POWER for *Load Multiple* and *Store Multiple* instructions.

- If the EA is not word-aligned, in Power ISA either an Alignment exception occurs or the addressed bytes are loaded, while in POWER an Alignment interrupt occurs if MSR_{AL}=1 (the low-order two bits of the EA are ignored if MSR_{AL}=0).
- In Power ISA the instruction may be interrupted by a system-caused interrupt, while in POWER the instruction cannot be thus interrupted.

A.16 Move Assist Instructions

There are several respects in which Power ISA is incompatible with POWER for *Move Assist* instructions.

- In Power ISA an ***lswx*** instruction with zero length leaves the contents of RT undefined (if RT≠RA and RT≠RB) or is an invalid instruction form (if RT=RA or RT=RB), while in POWER the corresponding instruction (***lsx***) is a no-op in these cases.
- In Power ISA an ***lswx*** instruction with zero length may alter the Reference bit, and a ***stswx*** instruction with zero length may alter the Reference and Change bits, while in POWER the corresponding instructions (***lsx*** and ***stsx***) do not alter the Reference and Change bits in this case.
- In Power ISA a *Move Assist* instruction may be interrupted by a system-caused interrupt, while in POWER the instruction cannot be thus interrupted.

A.17 Move To/From SPR

There are several respects in which Power ISA is incompatible with POWER for *Move To/From Special Purpose Register* instructions.

- The SPR field is ten bits long in Power ISA, but only five in POWER (see also Section A.3, “Reserved Fields in Instructions”).
- ***mfspr*** can be used to read the Decrementer in problem state in POWER, but only in privileged state in Power ISA.
- If the SPR value specified in the instruction is not one of the defined values, POWER behaves as follows.
  - If the instruction is executed in problem state and SPR_0=1, a Privileged Instruction type Program interrupt occurs. No architected registers are altered except those set by the interrupt.
  - Otherwise no architected registers are altered.

  In this same case, Power ISA behaves as follows.
  - If the instruction is executed in problem state and spr_0=1, either an Illegal Instruction type Program interrupt or a Privileged Instruction type Program interrupt occurs. No architected

registers are altered except those set by the interrupt.

- Otherwise either an Illegal Instruction type Program interrupt occurs (in which case no architected registers are altered except those set by the interrupt) or the results are boundedly undefined (or possibly undefined, for ***mtspr***; see Book III).

A.18 Effects of Exceptions on FPSCR Bits FR and FI

For the following cases, POWER does not specify how FR and FI are set, while Power ISA preserves them for Invalid Operation Exception caused by a *Compare* instruction, sets FI to 1 and FR to an undefined value for disabled Overflow Exception, and clears them otherwise.

- Invalid Operation Exception (enabled or disabled)
- Zero Divide Exception (enabled or disabled)
- Disabled Overflow Exception

A.19 Store Floating-Point Single Instructions

There are several respects in which Power ISA is incompatible with POWER for *Store Floating-Point Single* instructions.

- POWER uses $FPSCR_{UE}$ to help determine whether denormalization should be done, while Power ISA does not. Using $FPSCR_{UE}$ is in fact incorrect: if $FPSCR_{UE}$=1 and a denormalized single-precision number is copied from one storage location to another by means of ***lfs*** followed by ***stfs***, the two “copies” may not be the same.
- For an operand having an exponent that is less than 874 (unbiased exponent less than -149), POWER stores a zero (if $FPSCR_{UE}$=0) while Power ISA stores an undefined value.

A.20 Move From FPSCR

POWER defines the high-order 32 bits of the result of ***mffs*** to be 0xFFFF_FFFF, while Power ISA specifies that they are undefined.

A.21 Zeroing Bytes in the Data Cache

The ***dclz*** instruction of POWER and the ***dcbz*** instruction of Power ISA have the same opcode. However, the functions differ in the following respects.

- ***dclz*** clears a line while ***dcbz*** clears a block.
- ***dclz*** saves the EA in RA (if RA≠0) while ***dcbz*** does not.
- ***dclz*** is privileged while ***dcbz*** is not.

A.22 Synchronization

The *Synchronize* instruction (called ***dcs*** in POWER) and the ***isync*** instruction (called ***ics*** in POWER) cause more pervasive synchronization in Power ISA than in POWER. However, unlike ***dcs***, *Synchronize* does not wait until data cache block writes caused by preceding instructions have been performed in main storage. Also, *Synchronize* has an L field while ***dcs*** does not, and some uses of the instruction by the operating system require L=2<S>. (The L field corresponds to reserved bits in ***dcs*** and hence is expected to be zero in POWER programs; see Section A.3.)

A.23 Move To Machine State Register Instruction

The ***mtmsr*** instruction has an L field in Power ISA but not in POWER. The function of the variant of ***mtmsr*** with L=1 differs from the function of the instruction in the POWER architecture in the following ways.

- In Power ISA, this variant of ***mtmsr*** modifies only the EE and RI bits of the MSR, while in the POWER ***mtmsr*** modifies all bits of the MSR.
- This variant of ***mtmsr*** is execution synchronizing in Power ISA but is context synchronizing in POWER. (The POWER architecture lacks Power ISA’s distinction between execution synchronization and context synchronization. The statement in the POWER architecture specification that ***mtmsr*** is “synchronizing” is equivalent to stating that the instruction is context synchronizing.)

Also, ***mtmsr*** is optional in Power ISA but required in POWER.

A.24 Direct-Store Segments

POWER’s direct-store segments are not supported in Power ISA.

A.25 Segment Register Manipulation Instructions

The definitions of the four *Segment Register Manipulation* instructions ***mtsr***, ***mtsrin***, ***mfsr***, and ***mfsrin*** differ in two respects between POWER and Power ISA. Instructions similar to ***mtsrin*** and ***mfsrin*** are called ***mtsri*** and ***mfsri*** in POWER.

privilege: ***mfsr*** and ***mfsri*** are problem state instructions in POWER, while ***mfsr*** and ***mfsrin*** are privileged in Power ISA.

function: the “indirect” instructions (***mtsri*** and ***mfsri***) in POWER use an RA register in computing the Segment Register number, and the computed EA is stored into RA (if RA≠0 and RA≠RT), while in Power ISA ***mtsrin*** and ***mfsrin*** have no RA field and the EA is not stored.

mtsr, ***mtsrin*** (***mtsri***), and ***mfsr*** have the same opcodes in Power ISA as in POWER. ***mfsri*** (POWER) and ***mfsrin*** (Power ISA) have different opcodes.

Also, the *Segment Register Manipulation* instructions are required in POWER whereas they are optional in Power ISA.

A.26 TLB Entry Invalidation

The ***tlbi*** instruction of POWER and the ***tlbie*** instruction of Power ISA have the same opcode. However, the functions differ in the following respects.

- ***tlbi*** computes the EA as (RA|0) + (RB), while ***tlbie*** lacks an RA field and computes the EA and related information as (RB).
- ***tlbi*** saves the EA in RA (if RA≠0), while ***tlbie*** lacks an RA field and does not save the EA.
- For ***tlbi*** the high-order 36 bits of RB are used in computing the EA, while for ***tlbie*** these bits contain additional information that is not directly related to the EA.
- ***tlbie*** has an L field, while ***tlbi*** does not.

Also, ***tlbi*** is required in POWER whereas ***tlbie*** is optional in Power ISA.

A.27 Alignment Interrupts

Placing information about the interrupting instruction into the DSISR and the DAR when an Alignment interrupt occurs is optional in Power ISA but required in POWER.

A.28 Floating-Point Interrupts

POWER uses MSR bit 20 to control the generation of interrupts for floating-point enabled exceptions, and Power ISA uses the corresponding MSR bit, bit 52, for the same purpose. However, in Power ISA this bit is part of a two-bit value that controls the occurrence, precision, and recoverability of the interrupt, while in POWER this bit is used independently to control the occurrence of the interrupt (in POWER all floating-point interrupts are precise).

A.29 Timing Facilities

A.29.1 Real-Time Clock

The POWER Real-Time Clock is not supported in Power ISA. Instead, Power ISA provides a Time Base. Both the RTC and the TB are 64-bit Special Purpose Registers, but they differ in the following respects.

- The RTC counts seconds and nanoseconds, while the TB counts “ticks”. The ticking rate of the TB is implementation-dependent.
- The RTC increments discontinuously: 1 is added to RTCU when the value in RTCL passes 999_999_999. The TB increments continuously: 1 is added to TBU when the value in TBL passes 0xFFFF_FFFF.
- The RTC is written and read by the ***mtspr*** and ***mfspr*** instructions, using SPR numbers that denote the RTCU and RTCL. The TB is written and read by the same instructions using different SPR numbers.
- The SPR numbers that denote POWER’s RTCL and RTCU are invalid in Power ISA.
- The RTC is guaranteed to increment at least once in the time required to execute ten *Add Immediate* instructions. No analogous guarantee is made for the TB.
- Not all bits of RTCL need be implemented, while all bits of the TB must be implemented.

A.29.2 Decrementer

The Power ISA Decrementer differs from the POWER Decrementer in the following respects.

- The Power ISA DEC decrements at the same rate that the TB increments, while the POWER DEC decrements every nanosecond (which is the same rate that the RTC increments).
- Not all bits of the POWER DEC need be implemented, while all bits of the Power ISA DEC must be implemented.
- The interrupt caused by the DEC has its own interrupt vector location in Power ISA, but is considered an External interrupt in POWER.

A.30 Deleted Instructions

The following instructions are part of the POWER Architecture but have been dropped from the Power ISA.

abs	Absolute
clcs	Cache Line Compute Size
clf	Cache Line Flush
cli (*)	Cache Line Invalidate
dclst	Data Cache Line Store
div	Divide
divs	Divide Short
doz	Difference Or Zero
dozi	Difference Or Zero Immediate
lscbx	Load String And Compare Byte Indexed
maskg	Mask Generate
maskir	Mask Insert From Register
mfsri	Move From Segment Register Indirect
mul	Multiply
nabs	Negative Absolute
rac (*)	Real Address Compute
rfi (*)	Return From Interrupt
rfsvc	Return From SVC
rlmi	Rotate Left Then Mask Insert
rrib	Rotate Right And Insert Bit
sle	Shift Left Extended
sleq	Shift Left Extended With MQ
sliq	Shift Left Immediate With MQ
slliq	Shift Left Long Immediate With MQ
sllq	Shift Left Long With MQ
slq	Shift Left With MQ
sraiq	Shift Right Algebraic Immediate With MQ
sraq	Shift Right Algebraic With MQ
sre	Shift Right Extended
srea	Shift Right Extended Algebraic
sreq	Shift Right Extended With MQ
sriq	Shift Right Immediate With MQ
srliq	Shift Right Long Immediate With MQ
srlq	Shift Right Long With MQ
srq	Shift Right With MQ

(*) This instruction is privileged.

Note: Many of these instructions use the MQ register. The MQ is not defined in the Power ISA.

A.31 Discontinued Opcodes

The opcodes listed below are defined in the POWER Architecture but have been dropped from the Power ISA. The list contains the POWER mnemonic (MNEM), the primary opcode (PRI), and the extended opcode (XOP) if appropriate. The corresponding instructions are reserved in Power ISA.

MNEM	PRI	XOP
abs	31	360
clcs	31	531
clf	31	118
cli (*)	31	502
dclst	31	630
div	31	331
divs	31	363
doz	31	264
dozi	09	-
lscbx	31	277
maskg	31	29
maskir	31	541
mfsri	31	627
mul	31	107
nabs	31	488
rac (*)	31	818
rfi (*)	19	50
rfsvc	19	82
rlmi	22	-
rrib	31	537
sle	31	153
sleq	31	217
sliq	31	184
slliq	31	248
sllq	31	216
slq	31	152
sraiq	31	952
sraq	31	920
sre	31	665
srea	31	921
sreq	31	729
sriq	31	696
srliq	31	760
srlq	31	728
srq	31	664

(*) This instruction is privileged.

Assembler Note

It might be helpful to current software writers for the Assembler to flag the discontinued POWER instructions.

A.32 POWER2 Compatibility

The POWER2 instruction set is a superset of the POWER instruction set. Some of the instructions added for POWER2 are included in the Power ISA. Those that have been renamed in the Power ISA are listed in this section, as are the new POWER2 instructions that are not included in the Power ISA.

Other incompatibilities are also listed.

A.32.1 Cross-Reference for Changed POWER2 Mnemonics

The following table lists the new POWER2 instruction mnemonics that have been changed in the Power ISA User Instruction Set Architecture, sorted by POWER2 mnemonic.

To determine the Power ISA mnemonic for one of these POWER2 mnemonics, find the POWER2 mnemonic in the second column of the table: the remainder of the line gives the Power ISA mnemonic and the page on which the instruction is described, as well as the instruction names.

POWER2 mnemonics that have not changed are not listed.

Page	POWER2		Power ISA	
	Mnemonic	**Instruction**	**Mnemonic**	**Instruction**
126	fcir[.]	Floating Convert Double to Integer with Round	fctiw[.]	Floating Convert To Integer Word
127	fcirz[.]	Floating Convert Double to Integer with Round to Zero	fctiwz[.]	Floating Convert To Integer Word with round toward Zero

A.32.2 Floating-Point Conversion to Integer

The ***fcir*** and ***fcirz*** instructions of POWER2 have the same opcodes as do the ***fctiw*** and ***fctiwz*** instructions, respectively, of Power ISA. However, the functions differ in the following respects.

- ***fcir*** and ***fcirz*** set the high-order 32 bits of the target FPR to 0xFFFF_FFFF, while ***fctiw*** and ***fctiwz*** set them to an undefined value.
- Except for enabled Invalid Operation Exceptions, ***fcir*** and ***fcirz*** set the FPRF field of the FPSCR based on the result, while ***fctiw*** and ***fctiwz*** set it to an undefined value.
- ***fcir*** and ***fcirz*** do not affect the VXSNAN bit of the FPSCR, while ***fctiw*** and ***fctiwz*** do.
- ***fcir*** and ***fcirz*** set $FPSCR_{XX}$ to 1 for certain cases of “Large Operands” (i.e., operands that are too large to be represented as a 32-bit signed fixed-point integer), while ***fctiw*** and ***fctiwz*** do not alter it for any case of “Large Operand”. (The IEEE standard requires not altering it for “Large Operands”.)

A.32.3 Floating-Point Interrupts

POWER2 uses MSR bits 20 and 23 to control the generation of interrupts for floating-point enabled exceptions, and Power ISA uses the corresponding MSR bits, bits 52 and 55, for the same purpose. However, in Power ISA these bits comprise a two-bit value that controls the occurrence, precision, and recoverability of the interrupt, while in POWER2 these bits are used independently to control the occurrence (bit 20) and the precision (bit 23) of the interrupt. Moreover, in Power ISA all floating-point interrupts are considered Program interrupts, while in POWER2 imprecise floating-point interrupts have their own interrupt vector location.

A.32.4 Trace

The Trace interrupt vector location differs between the two architectures, and there are many other differences.

A.33 Deleted Instructions

The following instructions are new in POWER2 implementations of the POWER Architecture but have been dropped from the Power ISA.

lfq	Load Floating-Point Quad
lfqu	Load Floating-Point Quad with Update

lfqux Load Floating-Point Quad with Update Indexed
lfqx Load Floating-Point Quad Indexed
stfq Store Floating-Point Quad
stfqu Store Floating-Point Quad with Update
stfqux Store Floating-Point Quad with Update Indexed
stfqx Store Floating-Point Quad Indexed

A.33.1 Discontinued Opcodes

The opcodes listed below are new in POWER2 implementations of the POWER Architecture but have been dropped from the Power ISA. The list contains the POWER2 mnemonic (MNEM), the primary opcode (PRI), and the extended opcode (XOP) if appropriate. The corresponding instructions are either illegal or reserved in Power ISA; see Appendix D.

MNEM	PRI	XOP
lfq	56	-
lfqu	57	-
lfqux	31	823
lfqx	31	791
stfq	60	-
stfqu	61	-
stfqux	31	951
stfqx	31	919

Appendix B. Platform Support Requirements

As described in Chapter 1 of Book I, the architecture is structured as a collection of categories. Each category is comprised of facilities and/or instructions that together provide a unit of functionality. The Server and Embedded categories are referred to as “special” because all implementations must support at least one of these categories. Each special category, when taken together with the Base category, is referred to as an “environment”, and provides the minimum functionality required to develop operating systems and applications.

Every processor implementation supports at least one of the environments, and may also support a set of categories chosen based on the target market for the implementation. To facilitate the development of operating systems and applications for a well-defined purpose or customer set, usually embodied in a unique hardware platform, this appendix documents the association between a platform and the set of categories it requires.

Adding a new platform may permit cost-performance optimization by clearly identifying a unique set of categories. However, this has the potential to fragment the application base. As a result, new platforms will be added only when the optimization benefit clearly outweighs the loss due to fragmentation.

The platform support requirements are documented in Figure 20. An “x” in a column indicates that the category is required. A “+” in a column indicates that the requirement is being phased in.

Category	Server Platform	Embedded Platform
Base	x	x
Server	x	
Embedded		x
Alternate Time Base		
Cache Specification		
Embedded.Cache Debug		
Embedded.Cache Initialization		
Embedded.Enhanced Debug		
Embedded.External PID		
Embedded.Little-Endian		
Embedded.MMU Type FSL		*
Embedded.Performance Monitor		
Embedded.Processor Control		
Embedded Cache Locking		
External Control		
External Proxy		
Floating-Point Floating-Point.Record	x x	
Legacy Move Assist		
Legacy Integer Multiply-Accumulate		
Load/Store Quadword		
Memory Coherence	x	
Move Assist	x	
Server.Performance Monitor	x	
Signal Processing Engine SPE.Embedded Float Scalar Double SPE.Embedded Float Scalar Single SPE.Embedded Float Vector		
Stream	x	
Trace	x	
Variable Length Encoding		
Vector Vector.Little-Endian	+ +[1]	
Wait		
64-Bit	x	

[1] If the Vector category is supported, Vector.Little-Endian is required on the Server platform.

-

Figure 20. Platform Support Requirements

Appendix C. Complete SPR List

This appendix lists all the Special Purpose Registers in the Power ISA , ordered by SPR number.

decimal	SPR[1] $spr_{5:9}$ $spr_{0:4}$	Register Name	Privileged mtspr	Privileged mfspr	Length (bits)	Cat[2]
1	00000 00001	XER	no	no	64	B
8	00000 01000	LR	no	no	64	B
9	00000 01001	CTR	no	no	64	B
18	00000 10010	DSISR	yes	yes	32	S
19	00000 10011	DAR	yes	yes	64	S
22	00000 10110	DEC	yes	yes	32	B
25	00000 11001	SDR1	hypv[3]	yes	64	S
26	00000 11010	SRR0	yes	yes	64	B
27	00000 11011	SRR1	yes	yes	64	B
29	00000 11101	AMR	yes	yes	64	S
48	00001 10000	PID	yes	yes	32	E
54	00001 10110	DECAR	yes	yes	32	E
58	00001 11010	CSRR0	yes	yes	64	E
59	00001 11011	CSRR1	yes	yes	32	E
61	00001 11101	DEAR	yes	yes	64	E
62	00001 11110	ESR	yes	yes	32	E
63	00001 11111	IVPR	yes	yes	64	E
136	00100 01000	CTRL	-	no	32	S
152	00100 11000	CTRL	yes	-	32	S
256	01000 00000	VRSAVE	no	no	32	V
259	01000 00011	SPRG3	-	no	64	B
260-263	01000 001xx	SPRG[4-7]	-	no	64	E
268	01000 01100	TB	-	no	64	B
269	01000 01101	TBU	-	no	32	B
272-275	01000 100xx	SPRG[0-3]	yes	yes	64	B
276-279	01000 101xx	SPRG[4-7]	yes	yes	64	E
282	01000 11010	EAR	hypv[3]	yes	32	EC
284	01000 11100	TBL	hypv[4]	-	32	B
285	01000 11101	TBU	hypv[4]	-	32	B
286	01000 11110	TBU40	hypv	-	64	S
286	01000 11110	PIR	-	yes	32	E
287	01000 11111	PVR	-	yes	32	B
304	01001 10000	HSPRG0	hypv[3]	hypv[3]	64	S
304	01001 10000	DBSR	yes[5]	yes	32	E
305	01001 10001	HSPRG1	hypv[3]	hypv[3]	64	S
306	01001 10010	HDSISR	hypv[3]	hypv[3]	32	B
307	01001 10011	HDAR	hypv[3]	hypv[3]	64	B
308	01001 10100	DBCR0	yes	yes	32	E
309	01001 10101	PURR	hypv[3]	yes	64	S
309	01001 10101	DBCR1	yes	yes	32	E
310	01001 10110	HDEC	hypv[3]	yes	32	S

decimal	SPR[1]	Register Name	Privileged		Length (bits)	Cat[2]
	$spr_{5:9}$ $spr_{0:4}$		mtspr	mfspr		
310	01001 10110	DBCR2	yes	yes	32	E
312	01001 11000	RMOR	hypv[3]	hypv[3]	64	S
312	01001 11000	IAC1	yes	yes	64	E
313	01001 11001	HRMOR	hypv[3]	hypv[3]	64	S
313	01001 11001	IAC2	yes	yes	64	E
314	01001 11010	HSRR0	hypv[3]	hypv[3]	64	S
314	01001 11010	IAC3	yes	yes	64	E
315	01001 11011	HSRR1	hypv[3]	hypv[3]	64	S
315	01001 11011	IAC4	yes	yes	64	E
316	01001 11100	DAC1	yes	yes	64	E
317	01001 11101	DAC2	yes	yes	64	E
318	01001 11110	LPCR	hypv[3]	hypv[3]	64	S
318	01001 11110	DVC1	yes	yes	64	E
319	01001 11111	LPIDR	hypv[3]	hypv[3]	32	S
319	01001 11111	DVC2	yes	yes	64	E
336	01010 10000	TSR	yes[5]	yes	32	E
340	01010 10100	TCR	yes	yes	32	E
400-415	01100 1xxxx	IVOR[0-15]	yes	yes	32	E
512	10000 00000	SPEFSCR	no	no	32	SP
526	10000 01110	ATB/ATBL	-	no	64	ATB
527	10000 01111	ATBU	-	no	32	ATB
528	10000 10000	IVOR32	yes	yes	32	SP
529	10000 10001	IVOR33	yes	yes	32	SP
530	10000 10010	IVOR34	yes	yes	32	SP
531	10000 10011	IVOR35	yes	yes	32	E.PM
532	10000 10100	IVOR36	yes	yes	32	E.PC
533	10000 10101	IVOR37	yes	yes	32	E.PC
570	10001 11010	MCSRR0	yes	yes	64	E
571	10001 11011	MCSRR1	yes	yes	32	E
572	10001 11100	MCSR	yes	yes	64	E
574	10001 11110	DSRR0	yes	yes	64	E.ED
575	10001 11111	DSRR1	yes	yes	32	E.ED
604	10010 11100	SPRG8	yes	yes	64	XSR
605	10010 11101	SPRG9	yes	yes	64	XSR
624	10011 10000	MAS0	yes	yes	32	E.MF
625	10011 10001	MAS1	yes	yes	32	E.MF
626	10011 10010	MAS2	yes	yes	64	E.MF
627	10011 10011	MAS3	yes	yes	32	E.MF
628	10011 10100	MAS4	yes	yes	32	E.MF
630	10011 10110	MAS6	yes	yes	32	E.MF
633	10011 11001	PID1	yes	yes	32	E.MF
634	10011 11010	PID2	yes	yes	32	E.MF
688-691	10101 100xx	TLB[0-3]CFG	yes	yes	32	E.MF
702	10101 11110	EPR	-	yes	32	EXP
768-783	11000 0xxxx	perf_mon	-	no	64	S.PM
784-799	11000 1xxxx	perf_mon	yes	yes	64	S.PM
896	11100 00000	PPR	no	no	64	S
924	11100 11100	DCBTRL	-[6]	yes	32	E.CD
925	11100 11101	DCBTRH	-[6]	yes	32	E.CD
926	11100 11110	ICBTRL	-[7]	yes	32	E.CD
927	11100 11111	ICDBTRH	-[7]	yes	32	E.CD
944	11101 10000	MAS7	yes	yes	32	E.MF
947	11101 10011	EPLC	yes	yes	32	E.PD
948	11101 10100	EPSC	yes	yes	32	E.PD
979	11110 10011	ICBDR	-[7]	yes	32	E.CD
1012	11111 10100	MMUCSR0	yes	yes	32	E.MF
1013	11111 10101	DABR	hypv[3]	yes	64	S

decimal	SPR[1]		Register Name	Privileged		Length (bits)	Cat[2]
	$spr_{5:9}$	$spr_{0:4}$		mtspr	mfspr		
1015	11111	10111	DABRX	hypv[3]	yes	64	S
1015	11111	10111	MMUCFG	yes	yes	32	E.MF
1023	11111	11111	PIR	-	yes	32	S

\- 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] See Section 1.3.5 of Book I.
[3] This register is a hypervisor resource, and can be modified by this instruction only in hypervisor state (see Chapter 2 of Book III-S).
[4] <S>This register is a hypervisor resource, and can be modified by this instruction only in hypervisor state (see Chapter 2 of Book III-S).
<E>This register is privileged.
[5] This register cannot be directly written to. Instead, bits in the register corresponding to 1 bits in (RS) can be cleared using *mtspr SPR,RS*.
[6] The register can be written by the ***dcread*** instruction.
[7] The register can be written by the ***icread*** instruction.

All SPR numbers that are not shown above and are not implementation-specific are reserved.

Appendix D. Illegal Instructions

With the exception of the instruction consisting entirely of binary 0s, the instructions in this class are available for future extensions of the Power ISA; that is, some future version of the Power ISA may define any of these instructions to perform new functions.

The following primary opcodes are illegal.

1, 5, 6, 57, 60, 61

The following primary opcodes have unused extended opcodes. Their unused extended opcodes can be determined from the opcode maps in Appendix F of Book Appendices. All unused extended opcodes are illegal.

4, 19, 30, 31, 56, 58, 59, 62, 63

An instruction consisting entirely of binary 0s is illegal, and is guaranteed to be illegal in all future versions of this architecture.

Appendix E. 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 18) and the extended opcode shown below.

   256 Service Processor "Attention"

2. Instructions for the POWER Architecture that have not been included in the Power ISA. These are listed in Section A.31, "Discontinued Opcodes" and Section A.33.1, "Discontinued Opcodes".
3. Implementation-specific instructions used to conform to the Power ISA specification.
4. Any other implementation-dependent instructions that are not defined in the Power ISA.

Appendix F. Opcode Maps

This appendix contains tables showing the opcodes and extended opcodes.

For the primary opcode table (Table 3 on page 768), each cell is in the following format.

Opcode in Decimal		Opcode in Hexadecimal
	Instruction Mnemonic	
Category		Instruction Format

The category abbreviations are shown on Section 1.3.5 of Book I.

The extended opcode tables show the extended opcode in decimal, the instruction mnemonic, the category, and the instruction format. These tables appear in order of primary opcode within three groups. The first group consists of the primary opcodes that have small extended opcode fields (2-4 bits), namely 30, 58, and 62. The second group consists of primary opcodes that have 11-bit extended opcode fields. The third group consists of primary opcodes that have 10-bit extended opcode fields. The tables for the second and third groups are rotated.

In the extended opcode tables several special markings are used.

- A prime (‘) following an instruction mnemonic denotes an additional cell, after the lowest-numbered one, used by the instruction. For example, ***subfc*** occupies cells 8 and 520 of primary opcode 31, with the former corresponding to OE=0 and the latter to OE=1. Similarly, ***sradi*** occupies cells 826 and 827, with the former corresponding to sh_5=0 and the latter to sh_5=1 (the 9-bit extended opcode 413, shown on page 85, excludes the sh_5 bit).
- Two vertical bars (||) are used instead of primed mnemonics when an instruction occupies an entire column of a table. The instruction mnemonic is repeated in the last cell of the column.
- For primary opcode 31, an asterisk (*) in a cell that would otherwise be empty means that the cell is reserved because it is “overlaid”, by a fixed-point or *Storage Access* instruction having only a primary opcode, by an instruction having an extended opcode in primary opcode 30, 58, or 62, or by a potential instruction in any of the categories just mentioned. The overlaying instruction, if any, is also shown. A cell thus reserved should not be assigned to an instruction having primary opcode 31. (The overlaying is a consequence of opcode decoding for fixed-point instructions: the primary opcode, and the extended opcode if any, are mapped internally to a 10-bit “compressed opcode” for ease of subsequent decoding.)
- Parentheses around the opcode or extended opcode mean that the instruction was defined in earlier versions of the Power ISA but is no longer defined in the Power ISA.
- Curly brackets around the opcode or extended opcode mean that the instruction will be defined in future versions of the Power ISA.
- ***long*** is used as filler for mnemonics that are longer than a table cell.

An empty cell, a cell containing only an asterisk, or a cell in which the opcode or extended opcode is parenthesized, corresponds to an illegal instruction.

The instruction consisting entirely of binary 0s causes the system illegal instruction error handler to be invoked for all members of the POWER family, and this is likely to remain true in future models (it is guaranteed in the Power ISA). An instruction having primary opcode 0 but not consisting entirely of binary 0s is reserved except for the following extended opcode (instruction bits 21:30).

256 Service Processor “Attention” (Power ISA only)

Table 3: Primary opcodes

0 00 Illegal, Reserved	1 01	2 02 tdi 64 D	3 03 twi B D	See primary opcode 0 extensions on page 767 Trap Doubleword Immediate Trap Word Immediate
4 04 Vector, LMA, SP V, LMA, SP	5 05	6 06	7 07 mulli BD	See Table 7 and Table 8 Multiply Low Immediate
8 08 subfic B D	9 09	10 0A cmpli B D	11 0B cmpi B D	Subtract From Immediate Carrying Compare Logical Immediate Compare Immediate
12 0C addic B D	13 0D addic. B D	14 0E addi B D	15 0F addis B D	Add Immediate Carrying Add Immediate Carrying and Record Add Immediate Add Immediate Shifted
16 10 bc B B	17 11 sc B SC	18 12 b B I	19 13 CR ops, etc. XL	Branch Conditional System Call Branch See Table 10 on page 781
20 14 rlwimi B M	21 15 rlwinm B M	22 16	23 17 rlwnm B M	Rotate Left Word Imm. then Mask Insert Rotate Left Word Imm. then AND with Mask Rotate Left Word then AND with Mask
24 18 ori B D	25 19 oris B D	26 1A xori B D	27 1B xoris B D	OR Immediate OR Immediate Shifted XOR Immediate XOR Immediate Shifted
28 1C andi. B D	29 1D andis. B D	30 1E FX Dwd Rot MD[S]	31 1F FX Extended Ops	AND Immediate AND Immediate Shifted See Table 4 on page 769 See Table 10 on page 781
32 20 lwz B D	33 21 lwzu B D	34 22 lbz B D	35 23 lbzu B D	Load Word and Zero Load Word and Zero with Update Load Byte and Zero Load Byte and Zero with Update
36 24 stw B D	37 25 stwu B D	38 26 stb B D	39 27 stbu B D	Store Word Store Word with Update Store Byte Store Byte with Update
40 28 lhz B D	41 29 lhzu B D	42 2A lha B D	43 2B lhau B D	Load Half and Zero Load Half and Zero with Update Load Half Algebraic Load Half Algebraic with Update
44 2C sth B D	45 2D sthu B D	46 2E lmw B D	47 2F stmw B D	Store Half Store Half with Update Load Multiple Word Store Multiple Word
48 30 lfs FP D	49 31 lfsu FP D	50 32 lfd FP D	51 33 lfdu FP D	Load Floating-Point Single Load Floating-Point Single with Update Load Floating-Point Double Load Floating-Point Double with Update
52 34 stfs FP D	53 35 stfsu FP D	54 36 stfd FP D	55 37 stfdu FP D	Store Floating-Point Single Store Floating-Point Single with Update Store Floating-Point Double Store Floating-Point Double with Update
56 38 lq LSQ DQ	57 39	58 3A FX DS-form Loads DS	59 3B FP Single Extended Ops A	Load Quadword See Table 5 on page 769 See Table 15 on page 785
60 3C	61 3D	62 3E FX DS-form Stores DS	63 3F FP Double Extended Ops	See Table 6 on page 769 See Table 16 on page 787

Table 4: Extended opcodes for primary opcode 30 (instruction bits 27:30)				
	00	**01**	**10**	**11**
00	0 ***rldicl*** 64 MD	1 ***rldicl'*** MD	2 ***rldicr*** 64 MD	3 ***rldicr'*** MD
01	4 ***rldic*** 64 MD	5 ***rldic'*** MD	6 ***rldimi*** 64 MD	7 ***rldimi'*** MD
10	8 ***rldcl*** 64 MDS	9 ***rldcr*** 64 MDS		
11				

Table 5: Extended opcodes for primary opcode 58 (instruction bits 30:31)		
	0	**1**
0	0 ***ld*** 64 DS	1 ***ldu*** 64 DS
1	2 ***lwa*** 64 DS	

Table 6: Extended opcodes for primary opcode 62 (instruction bits 30:31)		
	0	**1**
0	0 ***std*** 64 DS	1 ***stdu*** 64 DS
1	2 ***stq*** LSQ DS	

Table 7: (Left) Extended opcodes for primary opcode 4 [Category: V & LMA] (instruction bits 21:31)

	000000	000001	000010	000011	000100	000101	000110	000111	001000	001001	001010	001011	001100	001101	001110	001111
00000	0 ***vaddubm*** V VX		2 ***vmaxub*** V VX		4 ***vrlb*** V VX		6 ***vcmpequb*** V VC		8 ***vmuloub*** V VX		10 ***vaddfp*** V VX		12 ***vmrghb*** V VX		14 ***vpkuhum*** V VX	
00001	64 ***vadduhm*** V VX		66 ***vmaxuh*** V VX		68 ***vrlh*** V VX		70 ***vcmpequh*** V VC		72 ***vmulouh*** V VX		74 ***vsubfp*** V VX		76 ***vmrghh*** V VX		78 ***vpkuwum*** V VX	
00010	128 ***vadduwm*** V VX		130 ***vmaxuw*** V VX		132 ***vrlw*** V VX		134 ***vcmpequw*** V VC						140 ***vmrghw*** V VX		142 ***vpkuhus*** V VX	
00011							198 ***vcmpeqfp*** V VC								206 ***vpkuwus*** V VX	
00100			258 ***vmaxsb*** V VX		260 ***vslb*** V VX				264 ***vmulosb*** V VX		266 ***vrefp*** V VX		268 ***vmrglb*** V VX		270 ***vpkshus*** V VX	
00101			322 ***vmaxsh*** V VX		324 ***vslh*** V VX				328 ***vmulosh*** V VX		330 ***vrsqrtefp*** V VX		332 ***vmrglh*** V VX		334 ***vpkswus*** V VX	
00110	384 ***vaddcuw*** V VX		386 ***vmaxsw*** V VX		388 ***vslw*** V VX						394 ***vexptefp*** V VX		396 ***vmrglw*** V VX		398 ***vpkshss*** V VX	
00111					452 ***vsl*** V VX		454 ***vcmpgefp*** V VC				458 ***vlogefp*** V VX				462 ***vpkswss*** V VX	
01000	512 ***vaddubs*** V VX		514 ***vminub*** V VX		516 ***vsrb*** V VX		518 ***vcmpgtub*** V VC		520 ***vmuleub*** V VX		522 ***vrfin*** V VX		524 ***vspltb*** V VX		526 ***vupkhsb*** V VX	
01001	576 ***vadduhs*** V VX		578 ***vminuh*** V VX		580 ***vsrh*** V VX		582 ***vcmpgtuh*** V VC		584 ***vmuleuh*** V VX		586 ***vrfiz*** V VX		588 ***vsplth*** V VX		590 ***vupkhsh*** V VX	
01010	640 ***vadduws*** V VX		642 ***vminuw*** V VX		644 ***vsrw*** V VX		646 ***vcmpgtuw*** V VC				650 ***vrfip*** V VX		652 ***vspltw*** V VX		654 ***vupklsb*** V VX	
01011					708 ***vsr*** V VX		710 ***vcmpgtfp*** V VC				714 ***vrfim*** V VX				718 ***vupklsh*** V VX	
01100	768 ***vaddsbs*** V VX		770 ***vminsb*** V VX		772 ***vsrab*** V VX		774 ***vcmpgtsb*** V VC		776 ***vmulesb*** V VX		778 ***vcuxwfp*** V VX		780 ***vspltisb*** V VX		782 ***vpkpx*** V VX	
01101	832 ***vaddshs*** V VX		834 ***vminsh*** V VX		836 ***vsrah*** V VX		838 ***vcmpgtsh*** V VC		840 ***vmulesh*** V VX		842 ***vcsxwfp*** V VX		844 ***vspltish*** V VX		846 ***vupkhpx*** V VX	
01110	896 ***vaddsws*** V VX		898 ***vminsw*** V VX		900 ***vsraw*** V VX		902 ***vcmpgtsw*** V VC				906 ***vcfpuxws*** V VX		908 ***vspltisw*** V VX			
01111							966 ***vcmpbfp*** V VC				970 ***vcfpsxws*** V VX				974 ***vupklpx*** V VX	
10000	1024 ***vsububm*** V VX		1026 ***vavgub*** V VX		1028 ***vand*** V VX		1030 ***vcmpequb.*** V VC				1034 ***vmaxfp*** V VX		1036 ***vslo*** V VX			
10001	1088 ***vsubuhm*** V VX		1090 ***vavgub*** V VX		1092 ***vandc*** V VX		1094 ***vcmpequh.*** V VC				1098 ***vminfp*** V VX		1100 ***vsro*** V VX			
10010	1152 ***vsubuwm*** V VX		1154 ***vavgub*** V VX		1156 ***vor*** V VX		1158 ***vcmpequw.*** V VC									
10011					1220 ***vxor*** V VX		1222 ***vcmpeqfp.*** V VC									
10100			1282 ***vavgsb*** V VX		1284 ***vnor*** V VX											
10101			1346 ***vavgsb*** V VX													
10110	1408 ***vsubcuw*** V VX		1410 ***vavgsb*** V VX													
10111							1478 ***vcmpgefp*** V VC									
11000	1536 ***vsububs*** V VX				1540 ***mfvscr*** V VX		1542 ***vcmpgtub.*** V VC		1544 ***vsum4ubs*** V VX							
11001	1600 ***vsubuhs*** V VX				1604 ***mtvscr*** V VX		1606 ***vcmpgtuh.*** V VC		1608 ***vsum4shs*** V VX							
11010	1664 ***vsubuws*** V VX						1670 ***vcmpgtuw.*** V VC		1672 ***vsum2sws*** V VX							
11011							1734 ***vcmpgtfp.*** V VC									
11100	1792 ***vsubsbs*** V VX						1798 ***vcmpgtsb.*** V VC		1800 ***vsum4sbs*** V VX							
11101	1856 ***vsubshs*** V VX						1862 ***vcmpgtsh.*** V VC									
11110	1920 ***vsubsws*** V VX						1926 ***vcmpgtsw.*** V VC		1928 ***vsumsws*** V VX							
11111							1990 ***vcmpbfp.*** V VC									

Table 7 (Left-Center) Extended opcodes for primary opcode 4 [Category: V & LMA] (instruction bits 21:31)

	010000	010001	010010	010011	010100	010101	010110	010111	011000	011001	011010	011011	011100	011101	011110	011111
00000	16 *mulhhwu* LMA XO	17 *mulhhwu.* LMA XO							24 *machhwu* LMA XO	24 *long* LMA XO						
00001	80 *mullhw* LMA XO	81 *mullhw.* LMA XO							88 *machhw* LMA XO	89 *machhw.* LMA XO			92 *nmachhw* LMA XO	93 *long* LMA XO		
00010									152 *long* LMA XO	153 *long* LMA XO						
00011									216 *machhws* LMA XO	217 *long* LMA XO			220 *long* LMA XO	220 *long* LMA XO		
00100	272 *mulchwu* LMA X	273 *mulchwu.* LMA X							280 *macchwu* LMA XO	281 *long* LMA XO						
00101	336 *mulchw* LMA X	337 *mulchw.* LMA X							344 *macchw* LMA XO	345 *macchw.* LMA XO			348 *nmacchw* LMA XO	349 *long* LMA XO		
00110									408 *long* LMA XO	409 *long* LMA XO						
00111									472 *macchws* LMA XO	473 *long* LMA XO			476 *long* LMA XO	477 *long* LMA XO		
01000																
01001																
01010																
01011																
01100	784 *mullhwu* LMA X	784 *mullhwu.* LMA X							792 *maclhwu* LMA XO	793 *maclhwu.* LMA XO						
01101	848 *mullhw* LMA X	849 *mullhw.* LMA X							856 *maclhw* LMA XO	857 *maclhw.* LMA XO			860 *nmaclhw* LMA XO	861 *nmaclhw.* LMA XO		
01110									920 *long* LMA XO	921 *long* LMA XO						
01111									984 *maclhws* LMA XO	985 *maclhws.* LMA XO			988 *long* LMA XO	989 *long* LMA XO		
10000	1040 *long* LMA XO	1041 *long* LMA XO							1048 *long* LMA XO	1049 *long* LMA XO						
10001	1104 *mullhwo.* LMA XO	1105 *mullhwo.* LMA XO							1112 *machhwo* LMA XO	1113 *long* LMA XO			1116 *long* LMA XO	1117 *long* LMA XO		
10010									1176 *long* LMA XO	1177 *long* LMA XO						
10011									1240 *long* LMA XO	1241 *long* LMA XO			1244 *long* LMA XO	1245 *long* LMA XO		
10100									1304 *long* LMA XO	1305 *long* LMA XO						
10101									1368 *macchwo* LMA XO	1369 *long* LMA XO			1372 *long* LMA XO	1373 *long* LMA XO		
10110									1432 *long* LMA XO	1433 *long* LMA XO						
10111									1496 *long* LMA XO	1497 *long* LMA XO			1500 *long* LMA XO	1501 *long* LMA XO		
11000																
11001																
11010																
11011																
11100									1816 *long* LMA XO	1817 *long* LMA XO						
11101									1880 *maclhwo* LMA XO	1881 *maclhwo.* LMA XO			1884 *long* LMA XO	1885 *long* LMA XO		
11110									1944 *long* LMA XO	1946 *long* LMA XO						
11111									2008 *long* LMA XO	2009 *long* LMA XO			2012 *long* LMA XO	2013 *long* LMA XO		

Table 7 (Right-Center) Extended opcodes for primary opcode 4 [Category: V & LMA] (instruction bits 21:31)

	100000	100001	100010	100011	100100	100101	100110	100111	101000	101001	101010	101011	101100	101101	101110	101111
00000	32 *vmhaddshs* V VA	32 *vmhraddshs* V VA	34 *vmladduhm* V VA		36 *vmsumubm* V VA	37 *vmsummbm* V VA	38 *vmsumuhm* V VA	39 *vmsumuhs* V VA	40 *vmsumshm* V VA	41 *vmsumshs* V VA	42 ***vsel*** V VA	43 ***vperm*** V VA	44 ***vsdoi*** V VA		46 ***vmaddfp*** V VA	47 ***vnmsubfp*** V VA
00001																
00010																
00011																
00100																
00101																
00110																
00111																
01000																
01001																
01010																
01011																
01100																
01101																
01110																
01111																
10000																
10001																
10010																
10011																
10100																
10101																
10110																
10111																
11000																
11001																
11010																
11011																
11100																
11101																
11110																
11111	*vmhaddshs*	*vmhraddshs*	*vmladduhm*		*vmsumubm*	*vmsummbm*	*vmsumuhm*	*vmsumuhs*	*vmsumshm*	*vmsumshs*	***vsel***	***vperm***	***vsdoi***		***vmaddfp***	***vnmsubfp***

Table 7 (Right) Extended opcodes for primary opcode 4 [Category: V & LMA] (instruction bits 21:31)																
	110000	110001	110010	110011	110100	110101	110110	110111	111000	111001	111010	111011	111100	111101	111110	111111
00000																
00001																
00010																
00011																
00100																
00101																
00110																
00111																
01000																
01001																
01010																
01011																
01100																
01101																
01110																
01111																
10000																
10001																
10010																
10011																
10100																
10101																
10110																
10111																
11000																
11001																
11010																
11011																
11100																
11101																
11110																
11111																

Table 8: (Left) Extended opcodes for primary opcode 4 [Category: SP.*] (instruction bits 21:31)

	000000	000001	000010	000011	000100	000101	000110	000111	001000	001001	001010	001011	001100	001101	001110	001111
00000																
00001																
00010																
00011																
00100																
00101																
00110																
00111																
01000	512 ***evaddw*** SP EVX		514 ***evaddiw*** SP EVX		516 ***evsubfw*** SP EVX		518 ***evsubifw*** SP EVX		520 ***evabs*** SP EVX	521 ***evneg*** SP EVX	522 ***evextsb*** SP EVX	523 ***evextsh*** SP EVX	524 ***evrndw*** SP EVX	525 ***evcntlzw*** SP EVX	526 ***evcntlsw*** SP EVX	527 ***brinc*** SP EVX
01001																
01010	640 ***evfsadd*** sp.fv EVX	641 ***evfssub*** sp.fv EVX			644 ***evfsabs*** sp.fv EVX	645 ***evfsnabs*** sp.fv EVX	646 ***evfsneg*** sp.fv EVX		648 ***evfsmul*** sp.fv EVX	649 ***evfsdiv*** sp.fv EVX			652 ***long*** sp.fv EVX	653 ***evfscmplt*** sp.fv EVX	654 ***long*** sp.fv EVX	
01011	704 ***efsadd*** sp.fs EVX	705 ***efssub*** sp.fs EVX			708 ***efsabs*** sp.fs EVX	709 ***efsnabs*** sp.fs EVX	710 ***efsneg*** sp.fs EVX		712 ***efsmul*** sp.fs EVX	713 ***efsdiv*** sp.fs EVX			716 ***efscmpgt*** sp.fs EVX	717 ***efscmplt*** sp.fs EVX	718 ***efscmpeq*** sp.fs EVX	719 ***efscfd*** sp.fd EVX
01100	768 ***evlddx*** SP EVX	769 ***evldd*** SP EVX	770 ***evldwx*** SP EVX	771 ***evldw*** SP EVX	772 ***evldhx*** SP EVX	773 ***evldh*** SP EVX			776 ***long*** SP EVX	777 ***long*** SP EVX			780 ***long*** SP EVX	781 ***long*** SP EVX	782 ***long*** SP EVX	783 ***long*** SP EVX
01101																
01110																
01111																
10000				1027 ***evmhessf*** SP EVX				1031 ***evmhossf*** SP EVX	1032 ***long*** SP EVX	1033 ***long*** SP EVX		1035 ***long*** SP EVX	1036 ***long*** SP EVX	1037 ***long*** SP EVX		1039 ***long*** SP EVX
10001								1095 ***long*** SP EVX	1096 ***long*** SP EVX				1100 ***long*** SP EVX	1101 ***long*** SP EVX		1103 ***long*** SP EVX
10010																
10011	1216 ***long*** SP EVX	1217 ***long*** SP EVX	1218 ***long*** SP EVX	1219 ***long*** SP EVX	1220 ***evmra*** SP EVX		1222 ***evdivws*** SP EVX	1223 ***evdivwu*** SP EVX	1224 ***long*** SP EVX	1225 ***long*** SP EVX	1226 ***long*** SP EVX	1227 ***long*** SP EVX				
10100	1280 ***long*** SP EVX	1281 ***long*** SP EVX		1283 ***long*** SP EVX		1285 ***long*** SP EVX		1287 ***long*** SP EVX	1288 ***long*** SP EVX	1289 ***long*** SP EVX		1291 ***long*** SP EVX	1292 ***long*** SP EVX	1293 ***long*** SP EVX		1295 ***long*** SP EVX
10101	1344 ***long*** SP EVX	1345 ***long*** SP EVX							1352 ***long*** SP EVX	1353 ***long*** SP EVX						
10110	1408 ***long*** SP EVX	1409 ***long*** SP EVX		1411 ***long*** SP EVX	1412 ***long*** SP EVX	1413 ***long*** SP EVX		1415 ***long*** SP EVX	1416 ***long*** SP EVX	1417 ***long*** SP EVX		1419 ***long*** SP EVX	1420 ***long*** SP EVX	1421 ***long*** SP EVX		1423 ***long*** SP EVX
10111	1472 ***long*** SP EVX	1473 ***long*** SP EVX							1480 ***long*** SP EVX	1481 ***long*** SP EVX						
11000																
11001																
11010																
11011																
11100																
11101																
11110																
11111																

Table 8 (Left-Center) Extended opcodes for primary opcode 4 [Category: SP.*] (instruction bits 21:31)

	010000	010001	010010	010011	010100	010101	010110	010111	011000	011001	011010	011011	011100	011101	011110	011111
00000																
00001																
00010																
00011																
00100																
00101																
00110																
00111																
01000		529 ***evand*** SP EVX	530 ***evandc*** SP EVX				534 ***evxor*** SP EVX	535 ***evor*** SP EVX	536 ***evnor*** SP EVX	537 ***eveqv*** SP EVX		539 ***evorc*** SP EVX			542 ***evnand*** SP EVX	
01001																
01010	656 ***evfsfui*** sp.fv EVX	657 ***evfscfsi*** sp.fv EVX	658 ***evfscfuf*** sp.fv EVX	659 ***evfscfsf*** sp.fv EVX	660 ***evfsctui*** sp.fv EVX	661 ***evfsctsi*** sp.fv EVX	662 ***evfsctuf*** sp.fv EVX	663 ***evfsctsf*** sp.fv EVX	664 ***evfsctuiz*** sp.fv EVX		666 ***evfsctsiz*** sp.fv EVX		668 ***evfststgt*** sp.fv EVX	669 ***evfststlt*** sp.fv EVX	670 ***evfststeq*** sp.fv EVX	
01011	720 ***efscfui*** sp.fs EVX	721 ***efscfsi*** sp.fs EVX	722 ***efscfuf*** sp.fs EVX	723 ***efscfsf*** sp.fs EVX	724 ***efsctui*** sp.fs EVX	725 ***efsctsi*** sp.fs EVX	726 ***efsctuf*** sp.fs EVX	727 ***efsctsf*** sp.fs EVX	728 ***efsctuiz*** sp.fs EVX		730 ***efsctsiz*** sp.fs EVX		732 ***efststgt*** sp.fs EVX	733 ***efststlt*** sp.fs EVX	734 ***efststeq*** sp.fs EVX	
01100	784 ***evlwhex*** SP EVX	785 ***evlwhe*** SP EVX			788 ***evlwhoux*** SP EVX	789 ***evlwhou*** SP EVX	790 ***evlwhosx*** SP EVX	791 ***evlwhos*** SP EVX	792 ***long*** SP EVX	793 ***long*** SP EVX			796 ***long*** SP EVX	797 ***long*** SP EVX		
01101																
01110																
01111																
10000																
10001									1112 ***long*** SP EVX	1113 ***long*** SP EVX		1115 ***long*** SP EVX				
10010																
10011																
10100																
10101				1363 ***long*** SP EVX					1368 ***long*** SP EVX	1369 ***long*** SP EVX		1371 ***long*** SP EVX				
10110																
10111				1491 ***long*** SP EVX					1496 ***long*** SP EVX	1497 ***long*** SP EVX		1499 ***long*** SP EVX				
11000																
11001																
11010																
11011																
11100																
11101																
11110																
11111																

Table 8 (Right-Center) Extended opcodes for primary opcode 4 [Category: SP.*] (instruction bits 21:31)

	100000	100001	100010	100011	100100	100101	100110	100111	101000	101001	101010	101011	101100	101101	101110	101111
00000																
00001																
00010																
00011																
00100																
00101																
00110																
00111																
01000	544 ***evsrwu*** SP EVX	545 ***evsrws*** SP EVX	546 ***evsrwiu*** SP EVX	547 ***evsrwis*** SP EVX	548 ***evslw*** SP EVX		550 ***evslwi*** SP EVX		552 ***evrlw*** SP EVX	553 ***evsplati*** SP EVX	554 ***evrlwi*** SP EVX	555 ***evsplatfi*** SP EVX	556 ***long*** SP EVX	557 ***long*** SP EVX	558 ***long*** SP EVX	559 ***long*** SP EVX
01001																
01010																
01011	736 ***efdadd*** sp.fdEVX	737 ***efdsub*** sp.fdEVX	738 ***efdcfuid*** sp.fdEVX	739 ***efdcfsid*** sp.fdEVX	740 ***efdabs*** sp.fdEVX	741 ***efdnabs*** sp.fdEVX	742 ***efdneg*** sp.fdEVX		744 ***efdmul*** sp.fdEVX	745 ***efddiv*** sp.fdEVX	746 ***efdctuidz*** sp.fdEVX	747 ***efdctsidz*** sp.fdEVX	748 ***efdcmpgt*** sp.fdEVX	749 ***efdcmplt*** sp.fdEVX	750 ***efdcmpeq*** sp.fdEVX	751 ***efdcfs*** sp.fdEVX
01100	800 ***evstddx*** SP EVX	801 ***evstdd*** SP EVX	802 ***evstdwx*** SP EVX	803 ***evstdw*** SP EVX	804 ***evstdhx*** SP EVX	805 ***evstdh*** SP EVX										
01101																
01110																
01111																
10000				1059 ***long*** SP EVX				1063 ***long*** SP EVX	1064 ***long*** SP EVX	1065 ***long*** SP EVX		1067 ***long*** SP EVX	1068 ***long*** SP EVX	1069 ***long*** SP EVX		1071 ***long*** SP EVX
10001								1127 ***long*** SP EVX	1128 ***long*** SP EVX				1132 ***long*** SP EVX	1133 ***long*** SP EVX		1135 ***long*** SP EVX
10010																
10011																
10100									1320 ***long*** SP EVX	1321 ***long*** SP EVX		1323 ***long*** SP EVX	1324 ***long*** SP EVX	1325 ***long*** SP EVX		1327 ***long*** SP EVX
10101																
10110									1448 ***long*** SP EVX	1449 ***long*** SP EVX		1451 ***long*** SP EVX	1452 ***long*** SP EVX	1453 ***long*** SP EVX		1455 ***long*** SP EVX
10111																
11000																
11001																
11010																
11011																
11100																
11101																
11110																
11111																

Table 8 (Right) Extended opcodes for primary opcode 4 [Category: SP.*] (instruction bits 21:31)																
	110000	**110001**	**110010**	**110011**	**110100**	**110101**	**110110**	**110111**	**111000**	**111001**	**111010**	**111011**	**111100**	**111101**	**111110**	**111111**
00000																
00001																
00010																
00011																
00100																
00101																
00110																
00111																
01000	560 ***evcmpgtu*** SP EVX	561 ***evcmpgts*** SP EVX	562 ***evcmpltu*** SP EVX	563 ***evcmplts*** SP EVX	564 ***evcmpeq*** SP EVX											
01001									632 ***evsel*** SP EVS	633 ***evsel'*** SP EVS	634 ***evsel'*** SP EVS	635 ***evsel'*** SP EVS	636 ***evsel'*** SP EVS	637 ***evsel'*** SP EVS	638 ***evsel'*** SP EVS	639 ***evsel'*** SP EVS
01010																
01011	752 ***efdcfui*** sp.fdEVX	753 ***efdcfsi*** sp.fdEVX	754 ***efdcfuf*** sp.fdEVX	755 ***efdcfsf*** sp.fdEVX	756 ***efdctui*** sp.fdEVX	757 ***efdctsi*** sp.fdEVX	758 ***efdctuf*** sp.fdEVX	759 ***efdctsf*** sp.fdEVX	760 ***efdctuiz*** sp.fdEVX		762 ***efdctsiz*** sp.fdEVX		764 ***efdtstgt*** sp.fdEVX	765 ***efdtstlt*** sp.fdEVX	766 ***efdtsteq*** sp.fdEVX	
01100	816 ***evstwhex*** SP EVX	817 ***evstwhe*** SP EVX			820 ***evstwhox*** SP EVX	821 ***evstwho*** SP EVX			824 ***evstwwex*** SP EVX	825 ***evstwwe*** SP EVX			828 ***evstwwox*** SP EVX	829 ***evstwwo*** SP EVX		
01101																
01110																
01111																
10000																
10001				1139 ***long*** SP EVX						1145 ***long*** SP EVX		1147 ***long*** SP EVX				
10010																
10011																
10100																
10101																
10110																
10111																
11000																
11001																
11010																
11011																
11100																
11101																
11110																
11111																

Table 9: (Left) Extended opcodes for primary opcode 19 (instruction bits 21:30)

	00000	00001	00010	00011	00100	00101	00110	00111	01000	01001	01010	01011	01100	01101	01110	01111
00000	0 *mcrf* B XL															
00001		33 *crnor* B XL					38 *rfmci* E XL	39 *rfdi* E.ED X								
00010																
00011																
00100		129 *crandc* B XL														
00101																
00110		193 *crxor* B XL					198 *dnh* E.EDXFX									
00111		225 *crnand* B XL														
01000		257 *crand* B XL														
01001		289 *creqv* B XL														
01010																
01011																
01100																
01101		417 *crorc* B XL														
01110		449 *cror* B XL														
01111																
10000																
10001																
10010																
10011																
10100																
10101																
10110																
10111																
11000																
11001																
11010																
11011																
11100																
11101																
11110																
11111																

Table 9. (Right) Extended opcodes for primary opcode 19 (instruction bits 21:30)

	10000	10001	10010	10011	10100	10101	10110	10111	11000	11001	11010	11011	11100	11101	11110	11111
00000	16 ***bclr*** B XL		18 ***rfid*** S XL													
00001			50 ***rfi*** E XL	51 ***rfci*** E XL												
00010			(82) ***rfsvc*** XL													
00011																
00100							150 ***isync*** B XL									
00101																
00110																
00111																
01000			274 ***hrfid*** S XL													
01001																
01010																
01011																
01100																
01101																
01110																
01111																
10000	528 ***bcctr*** B XL															
10001																
10010																
10011																
10100																
10101																
10110																
10111																
11000																
11001																
11010																
11011																
11100																
11101																
11110																
11111																

Table 10: (Left) Extended opcodes for primary opcode 31 (instruction bits 21:30)

	00000	00001	00010	00011	00100	00101	00110	00111	01000	01001	01010	01011	01100	01101	01110	01111
00000	0 ***cmp*** B X				4 ***tw*** B X		6 ***lvsl*** V X	7 ***lvebx*** V X	8 ***subfc*** B XO	9 ***mulhdu*** 64 XO	10 ***addc*** B XO	11 ***mulhwu*** B XO			14 Res’d VLE	15 See Table 14
00001	32 ***cmpl*** B X	33 Res’d VLE					38 ***lvsr*** V X	39 ***lvehx*** V X	40 ***subf*** B XO						46 Res’d VLE	
00010					68 ***td*** 64 X			71 ***lvewx*** V X		73 ***mulhd*** 64 XO		75 ***mulhw*** B XO			78 ***dlmzb*** LMA X	
00011								103 ***lvx*** V X	104 ***neg*** B XO							
00100		129 Res’d VLE		131 ***wrtee*** E X			134 ***dcbtstls*** ECL X	135 ***stvebx*** V X	136 ***subfe*** B XO		138 ***adde*** B XO					
00101				163 ***wrteei*** E X			166 ***dcbtls*** ECL X	167 ***stvehx*** V X								
00110		193 Res’d VLE						199 ***stvewx*** V X	200 ***subfze*** B XO		202 ***addze*** B XO				206 ***msgsnd*** E.PC X	
00111		225 Res’d VLE					230 ***icblc*** ECL X	231 ***stvx*** V X	232 ***subfme*** B XO	233 ***mulld*** 64 XO	234 ***addme*** B XO	235 ***mullw*** B XO			238 ***msgclr*** E.PC X	
01000		257 Res’d VLE		259 ***mfdrx*** E X			262 Res’d AP	263 ***lvepxl*** E.PD X			266 ***add*** B XO					
01001		289 Res’d VLE		291 ***mfdrux*** E X				295 ***lvepx*** E.PD X								
01010				323 ***mfdcr*** E X			326 ***dcread*** E.CD X								334 ***mfpmr*** E.PM X	
01011								{359} ***lvxl*** V X								
01100				387 ***mtdcrx*** E X			390 ***dcblc*** ECL X									
01101		417 Res’d VLE		419 ***mtdcrux*** E X												
01110		449 Res’d VLE		451 ***mtdcr*** E X			454 ***dci*** E.CI X			457 ***divdu*** 64 XO		459 ***divwu*** B XO			462 ***mtpmr*** E.PM X	
01111							486 Res’d AP	{487} ***stvxl*** V X		489 ***divd*** 64 XO		491 ***divw*** B XO				
10000	512 ***mcrxr*** B X							{519} ***lvlx*** V X	520 ***subfc’*** B XO	521 ***mulhdu’*** 64XO	522 ***addc’*** B XO	523 ***mulhwu’*** B XO				
10001								{551} ***lvrx*** V X	552 ***subf’*** B XO							
10010										585 ***mulhd’*** 64 XO		587 ***mulhw’*** B XO				
10011									616 ***neg’*** B XO							
10100								{647} ***stvlx*** V X	648 ***subfe’*** B XO		650 ***adde’*** B XO					
10101								{679} ***stvrx*** V X								
10110									712 ***subfze’*** B XO		714 ***addze’*** B XO					
10111									744 ***subfme’*** B XO	745 ***mulld’*** 64 XO	746 ***addme’*** B XO	747 ***mullw’*** B XO				
11000								775 ***stvepxl*** E.PD X			778 ***add’*** B XO					
11001								807 ***stvepx*** E.PD X								
11010																
11011																
11100								903 ***stvlxl*** V X								
11101								935 ***stvrxl*** V X								
11110							966 ***ici*** E.CI X			969 ***divdu’*** 64 XO		971 ***divwu’*** B XO				
11111							998 ***icread*** E.CD X			1001 ***divd’*** 64 XO		1003 ***divw’*** B XO				See Table 14

Table 10. (Right) Extended opcodes for primary opcode 31 (instruction bits 21:30)

	10000	10001	10010	10011	10100	10101	10110	10111	11000	11001	11010	11011	11100	11101	11110	11111
00000	16 Res'd VLE			19 mfcr B XFX	20 lwarx B X	21 ldx 64 X	22 icbt E X	23 lwzx B X	24 slw B X		26 cntlzw B X	27 sld 64 X	28 and B X	29 ldepx E.PD X	30 rldicl* 64 MD	31 lwepx E.PD X
00001						53 ldux 64 X	54 dcbst B X	55 lwzux B X	56 Res'd VLE		58 cntlzd 64 X		60 andc B X		62 See Table 11	
00010			(82) mtsrd X	83 mfmsr B X	84 ldarx 64 X		86 dcbf B X	87 lbzx B X							94 rldicr* 64 MD	95 lbepx E.PD X
00011			(114) mtsrdin X				118 clf X	119 lbzux B X			122 popcntb B X		124 nor B X		126 rldicr* 64 MD	127 dcbfep E.PD X
00100	144 mtcrf B XFX		146 mtmsr B X			149 stdx 64 X	150 stwcx. B X	151 stwx B X						157 stdepx E.PD X	158 rldic* 64 MD	159 See Table 13
00101			178 mtmsrd S X			181 stdux 64 X		183 stwux B X							190 rldic* 64 MD	191 rlwinm* B M
00110			210 mtsr S X				214 stdcx. 64 X	215 stbx B X							222 rldimi* 64 MD	223 stbepx E.PD X
00111			242 mtsrin S X				246 dcbtst B X	247 stbux B X							254 rldimi* 64 MD	255 See Table 13
01000			274 tlbiel S X	275 mfapidi E X			278 dcbt B X	279 lhzx B X	280 Res'd VLE				284 eqv B X	285 evlddepx E.PD evx	286 rldcl* 64 MDS	286 See Table 13
01001			306 tlbie S X		308 Res'd		310 eciwx EC X	311 lhzux B X	312 Res'd VLE				316 xor B X		318 rldcr* 64 MDS	319 See Table 13
01010				339 mfspr B XFX		341 lwax 64 X	342 Res'd AP	343 lhax B X							350 *	351 xori* B D
01011			370 tlbia S X	371 mftb S XFX		373 lwaux 64 X	374 Res'd AP	375 lhaux B X							382 *	383 xoris* B D
01100			402 slbmte S X					407 sthx B X					412 orc B X	413 evstddepx E.PD evx	414 *	415 See Table 13
01101			434 slbie S X				438 ecowx EC X	439 sthux B X					444 or B X		446 *	447 andis.* B D
01110				467 mtspr B XFX		469 *	470 dcbi E X	471 lmw* All D					476 nand B X		478 *	
01111			498 slbia S X			501 *		503 stmw* All D							510 *	
10000					532 Res'd	533 lswx B MA	534 lwbrx B X	535 lfsx FP X	536 srw B X			539 srd 64 X				
10001							566 tlbsync S X	567 lfsux FP X	568 Res'd VLE							
10010				595 mfsr S X		597 lswi B MA	598 sync B X	599 lfdx FP X								607 lfdepx E.PD X
10011								631 lfdux FP X								
10100				659 mfsrin S X	660 Res'd	661 stswx B MA	662 stwbrx B X	663 stfsx FP X								
10101								695 stfsux FP X								
10110						725 stswi B MA		727 stfdx FP X								735 stfdepx E.PD X
10111							758 dcba E X	759 stfdux FP X								
11000			786 tlbivax E X				790 lhbrx B X		792 sraw B X		794 srad 64 X					
11001			818 rac X				822 Res'd	823 Res'd	824 srawi B X		826 sradi 64 XS	827 sradi' 64 XS				
11010				851 slbmfev S X			854 See Table 12									
11011																
11100			914 tlbsx E X	915 slbmfee S X			918 sthbrx B X				922 extsh B X					
11101			946 tlbre E X					951 Res'd AP			954 extsb B X					
11110			978 tlbwe E X				982 icbi B X	983 stfiwx FP X			986 extsw 64 X					991 icbiep E.PD X
11111			1010 Res'd				1014 dcbz B X									1023 dcbzep E.PD X

Table 11: Opcode: 31, Extended Opcode: 62

0	00001	
00001	62 *rldicl** 64 MD	62 *wait* WT X

Table 12: Opcode: 31, Extended Opcode: 854

	10110	
11010	854 *eieio* S X	854 *mbar* E X

Table 13: Opcode: 31, Extended Opcode: 159

	11111	
00100	159 *rlwimi** B M	159 *stwepx* E.PD X
00101	191 *rlwinm** B M	
00110		223 *stbepx* E.PD X
00111	255 *rlwnm** B M	255 *dcbstep* E.PD X
01000	287 *ori** B D	287 *lhepx* E.PD X
01001	319 *oris** B D	319 *dcbtep* E.PD X
01010	351 *xori** B D	
01011	383 *xoris** B D	
01100	415 *andi.** B D	415 *sthepx* E.PD X

Table 14: Opcode: 31, Extended Opcode: 15

	01111	
00000		15 *isel* B.in A
00001	47 *	
00010	79 *tdi** 64 D	
00011	111 *twi** B D	
00100	143 *	
00101	175 *	
00110	207 *	
00111	239 *mulli** B D	
01000	271 *subfic** B D	
01001		
01010	335 *cmpli** B D	
01011	367 *cmpi** B D	
01100	399 *addic** B D	
01101	431 *addic.** B D	
01110	463 *addi** B D	
01111	495 *addis** B D	
10000		
10001		
10010		
10011		
10100		
10101		
10110		
10111		
11000		
11001		
11010		
11011		
11100		
11101		
11110		
11111		*isel*

Table 15:(Left) Extended opcodes for primary opcode 59 (instruction bits 21:30)

	00000	00001	00010	00011	00100	00101	00110	00111	01000	01001	01010	01011	01100	01101	01110	01111
00000																
00001																
00010																
00011																
00100																
00101																
00110																
00111																
01000																
01001																
01010																
01011																
01100																
01101																
01110																
01111																
10000																
10001																
10010																
10011																
10100																
10101																
10110																
10111																
11000																
11001																
11010																
11011																
11100																
11101																
11110																
11111																

Table 15. (Right) Extended opcodes for primary opcode 59 (instruction bits 21:30)

	10000	10001	10010	10011	10100	10101	10110	10111	11000	11001	11010	11011	11100	11101	11110	11111
00000			18 ***fdivs*** FP A		20 ***fsubs*** FP A	21 ***fadds*** FP A	22 ***fsqrts*** FP A		24 ***fres*** FP A	25 ***fmuls*** FP A	26 ***frsqrtes*** FP A		28 ***fmsub*** FP A	29 ***fmadds*** FP A	30 ***fnmsubs*** FP A	31 ***fnmadds*** FP A
00001																
00010																
00011																
00100																
00101																
00110																
00111																
01000																
01001																
01010																
01011																
01100																
01101																
01110																
01111																
10000																
10001																
10010																
10011																
10100																
10101																
10110																
10111																
11000																
11001																
11010																
11011																
11100																
11101																
11110																
11111			***fdivs***		***fsubs***	***fadds***	***fsqrts***		***fres***	***fmuls***	***frsqrtes***		***fmsub***	***fmadds***	***fnmsubs***	***fnmadds***

Table 16:(Left) Extended opcodes for primary opcode 63 (instruction bits 21:30)

	00000	00001	00010	00011	00100	00101	00110	00111	01000	01001	01010	01011	01100	01101	01110	01111
00000	0 ***fcmpu*** FP X												12 ***frsp*** FP X		14 ***fctiw*** FP X	15 ***fctiwz*** FP X
00001	32 ***fcmpo*** FP X						38 ***mtfsb1*** FP X		40 ***fneg*** FP X							
00010	64 ***mcrfs*** FP X						70 ***mtfsb0*** FP X		72 ***fmr*** FP X							
00011																
00100							134 ***mtfsfi*** FP X		136 ***fnabs*** FP X							
00101																
00110																
00111																
01000									264 ***fabs*** FP X							
01001																
01010																
01011																
01100									392 ***frin*** FP X							
01101									424 ***friz*** FP X							
01110									456 ***frip*** FP X							
01111									488 ***frim*** FP X							
10000																
10001																
10010								583 ***mffs*** FP X								
10011																
10100																
10101																
10110								711 ***mtfsf*** FP XFL								
10111																
11000																
11001															814 ***fctid*** FP X	815 ***fctidz*** FP X
11010															846 ***fcfid*** FP X	
11011																
11100																
11101																
11110																
11111																

Table 16. (Right) Extended opcodes for primary opcode 63 (instruction bits 21:30)

	10000	10001	10010	10011	10100	10101	10110	10111	11000	11001	11010	11011	11100	11101	11110	11111
00000			18 ***fdiv*** FP A		20 ***fsub*** FP A	21 ***fadd*** FP A	22 ***fsqrt*** FP A	23 ***fsel*** FP A	24 ***fre*** FP A	25 ***fmul*** FP A	26 ***frsqrte*** FP A		28 ***fmsub*** FP A	29 ***fmadd*** FP A	30 ***fnmsub*** FP A	31 ***fnmadd*** FP A
00001																
00010																
00011																
00100																
00101																
00110																
00111																
01000																
01001																
01010																
01011																
01100																
01101																
01110																
01111																
10000																
10001																
10010																
10011																
10100																
10101																
10110																
10111																
11000																
11001																
11010																
11011																
11100																
11101																
11110																
11111			***fdiv***		***fsub***	***fadd***	***fsqrt***	***fsel***	***fre***	***fmul***	***frsqrte***		***fmsub***	***fmadd***	***fnmsub***	***fnmadd***

Appendix G. Power ISA Instruction Set Sorted by Category

This appendix lists all the instructions in the Power ISA, grouped by category, and in order by mnemonic within category.

Form	Opcode		Mode Dep.[1]	Priv[1]	Page	Cat[1]	Mnemonic	Instruction
	Pri	Ext						
X	31	58	SR		76	64	cntlzd[.]	Count Leading Zeros Doubleword
XO	31	489	SR		66	64	divd[o][.]	Divide Doubleword
XO	31	457	SR		66	64	divdu[o][.]	Divide Doubleword Unsigned
X	31	986	SR		76	64	extsw[.]	Extend Sign Word
DS	58	0			46	64	ld	Load Doubleword
X	31	84			371	64	ldarx	Load Doubleword And Reserve Indexed
DS	58	1			46	64	ldu	Load Doubleword with Update
X	31	53			46	64	ldux	Load Doubleword with Update Indexed
X	31	21			46	64	ldx	Load Doubleword Indexed
DS	58	2			45	64	lwa	Load Word Algebraic
X	31	373			45	64	lwaux	Load Word Algebraic with Update Indexed
X	31	341			45	64	lwax	Load Word Algebraic Indexed
XO	31	73	SR		65	64	mulhd[.]	Multiply High Doubleword
XO	31	9	SR		65	64	mulhdu[.]	Multiply High Doubleword Unsigned
XO	31	233	SR		65	64	mulld[o][.]	Multiply Low Doubleword
MDS	30	8	SR		81	64	rldcl[.]	Rotate Left Doubleword then Clear Left
MDS	30	9	SR		82	64	rldcr[.]	Rotate Left Doubleword then Clear Right
MD	30	2	SR		81	64	rldic[.]	Rotate Left Doubleword Immediate then Clear
MD	30	0	SR		79	64	rldicl[.]	Rotate Left Doubleword Immediate then Clear Left
MD	30	1	SR		80	64	rldicr[.]	Rotate Left Doubleword Immediate then Clear Right
MD	30	3	SR		82	64	rldimi[.]	Rotate Left Doubleword Immediate then Mask Insert
X	31	27	SR		85	64	sld[.]	Shift Left Doubleword
X	31	794	SR		85	64	srad[.]	Shift Right Algebraic Doubleword
XS	31	413	SR		85	64	sradi[.]	Shift Right Algebraic Doubleword Immediate
X	31	539	SR		85	64	srd[.]	Shift Right Doubleword
DS	62	0			50	64	std	Store Doubleword
X	31	214			371	64	stdcx.	Store Doubleword Conditional Indexed
DS	62	1			50	64	stdu	Store Doubleword with Update
X	31	181			50	64	stdux	Store Doubleword with Update Indexed
X	31	149			50	64	stdx	Store Doubleword Indexed
X	31	68			70	64	td	Trap Doubleword
D	2				70	64	tdi	Trap Doubleword Immediate
XO	31	266	SR		59	B	add[o][.]	Add
XO	31	10	SR		60	B	addc[o][.]	Add Carrying
XO	31	138	SR		61	B	adde[o][.]	Add Extended
D	14				58	B	addi	Add Immediate
D	12		SR		59	B	addic	Add Immediate Carrying
D	13		SR		59	B	addic.	Add Immediate Carrying and Record
D	15				58	B	addis	Add Immediate Shifted
XO	31	234	SR		61	B	addme[o][.]	Add to Minus One Extended
XO	31	202	SR		62	B	addze[o][.]	Add to Zero Extended
X	31	28	SR		73	B	and[.]	AND
X	31	60	SR		74	B	andc[.]	AND with Complement

Form	Opcode		Mode Dep.[1]	Priv[1]	Page	Cat[1]	Mnemonic	Instruction
	Pri	Ext						
D	28		SR		71	B	andi.	AND Immediate
D	29		SR		71	B	andis.	AND Immediate Shifted
I	18				31	B	b[l][a]	Branch
B	16		CT		31	B	bc[l][a]	Branch Conditional
XL	19	528	CT		32	B	bcctr[l]	Branch Conditional to Count Register
XL	19	16	CT		32	B	bclr[l]	Branch Conditional to Link Register
X	31	0			67	B	cmp	Compare
D	11				67	B	cmpi	Compare Immediate
X	31	32			68	B	cmpl	Compare Logical
D	10				68	B	cmpli	Compare Logical Immediate
X	31	26	SR		74	B	cntlzw[.]	Count Leading Zeros Word
XL	19	257			33	B	crand	Condition Register AND
XL	19	129			34	B	crandc	Condition Register AND with Complement
XL	19	289			34	B	creqv	Condition Register Equivalent
XL	19	225			33	B	crnand	Condition Register NAND
XL	19	33			34	B	crnor	Condition Register NOR
XL	19	449			33	B	cror	Condition Register OR
XL	19	417			34	B	crorc	Condition Register OR with Complement
XL	19	193			33	B	crxor	Condition Register XOR
X	31	86			367	B	dcbf	Data Cache Block Flush
X	31	54			366	B	dcbst	Data Cache Block Store
X	31	278			360	B	dcbt	Data Cache Block Touch
X	31	246			365	B	dcbtst	Data Cache Block Touch for Store
X	31	1014			366	B	dcbz	Data Cache Block set to Zero
XO	31	491	SR		64	B	divw[o][.]	Divide Word
XO	31	459	SR		64	B	divwu[o][.]	Divide Word Unsigned
X	31	284	SR		74	B	eqv[.]	Equivalent
X	31	954	SR		74	B	extsb[.]	Extend Sign Byte
X	31	922	SR		74	B	extsh[.]	Extend Sign Halfword
X	31	982			359	B	icbi	Instruction Cache Block Invalidate
XL	19	150			369	B	isync	Instruction Synchronize
D	34				41	B	lbz	Load Byte and Zero
D	35				41	B	lbzu	Load Byte and Zero with Update
X	31	119			41	B	lbzux	Load Byte and Zero with Update Indexed
X	31	87			42	B	lbzx	Load Byte and Zero Indexed
D	42				43	B	lha	Load Halfword Algebraic
D	43				43	B	lhau	Load Halfword Algebraic with Update
X	31	375			43	B	lhaux	Load Halfword Algebraic with Update Indexed
X	31	343			43	B	lhax	Load Halfword Algebraic Indexed
X	31	790			51	B	lhbrx	Load Halfword Byte-Reverse Indexed
D	40				42	B	lhz	Load Halfword and Zero
D	41				42	B	lhzu	Load Halfword and Zero with Update
X	31	311			42	B	lhzux	Load Halfword and Zero with Update Indexed
X	31	279			42	B	lhzx	Load Halfword and Zero Indexed
D	46				52	B	lmw	Load Multiple Word
X	31	20			370	B	lwarx	Load Word And Reserve Indexed
X	31	534			51	B	lwbrx	Load Word Byte-Reverse Indexed
D	32				44	B	lwz	Load Word and Zero
D	33				44	B	lwzu	Load Word and Zero with Update
X	31	55			44	B	lwzux	Load Word and Zero with Update Indexed
X	31	23			44	B	lwzx	Load Word and Zero Indexed
XL	19	0			34	B	mcrf	Move Condition Register Field
X	31	512			91	B	mcrxr	Move to Condition Register from XER
XFX	31	19			89	B	mfcr	Move From Condition Register
X	31	83		P	417, 527	B	mfmsr	Move From Machine State Register
XFX	31	339		O	88,378	B	mfspr	Move From Special Purpose Register
XFX	31	144			89	B	mtcrf	Move To Condition Register Fields
XFX	31	467		O	87	B	mtspr	Move To Special Purpose Register
XO	31	75	SR		63	B	mulhw[.]	Multiply High Word

Form	Opcode Pri	Opcode Ext	Mode Dep.[1]	Priv[1]	Page	Cat[1]	Mnemonic	Instruction
XO	31	11	SR		63	B	mulhwu[.]	Multiply High Word Unsigned
D	7				63	B	mulli	Multiply Low Immediate
XO	31	235	SR		63	B	mullw[o][.]	Multiply Low Word
X	31	476	SR		73	B	nand[.]	NAND
XO	31	104	SR		62	B	neg[o][.]	Negate
X	31	124	SR		74	B	nor[.]	NOR
X	31	444	SR		73	B	or[.]	OR
X	31	412	SR		74	B	orc[.]	OR with Complement
D	24				71	B	ori	OR Immediate
D	25				72	B	oris	OR Immediate Shifted
M	20		SR		79	B	rlwimi[.]	Rotate Left Word Immediate then Mask Insert
M	21		SR		77	B	rlwinm[.]	Rotate Left Word Immediate then AND with Mask
M	23		SR		78	B	rlwnm[.]	Rotate Left Word then AND with Mask
SC	17				35, 404, 515	B	sc	System Call
X	31	24	SR		83	B	slw[.]	Shift Left Word
X	31	792	SR		84	B	sraw[.]	Shift Right Algebraic Word
X	31	824	SR		84	B	srawi[.]	Shift Right Algebraic Word Immediate
X	31	536	SR		83	B	srw[.]	Shift Right Word
D	38				47	B	stb	Store Byte
D	39				47	B	stbu	Store Byte with Update
X	31	247			47	B	stbux	Store Byte with Update Indexed
X	31	215			47	B	stbx	Store Byte Indexed
D	44				48	B	sth	Store Halfword
X	31	918			51	B	sthbrx	Store Halfword Byte-Reverse Indexed
D	45				48	B	sthu	Store Halfword with Update
X	31	439			48	B	sthux	Store Halfword with Update Indexed
X	31	407			48	B	sthx	Store Halfword Indexed
D	47				53	B	stmw	Store Multiple Word
D	36				49	B	stw	Store Word
X	31	662			51	B	stwbrx	Store Word Byte-Reverse Indexed
X	31	150			370	B	stwcx.	Store Word Conditional Indexed
D	37				49	B	stwu	Store Word with Update
X	31	183			49	B	stwux	Store Word with Update Indexed
X	31	151			49	B	stwx	Store Word Indexed
XO	31	40	SR		59	B	subf[o][.]	Subtract From
XO	31	8	SR		60	B	subfc[o][.]	Subtract From Carrying
XO	31	136	SR		61	B	subfe[o][.]	Subtract From Extended
D	8		SR		60	B	subfic	Subtract From Immediate Carrying
XO	31	232	SR		61	B	subfme[o][.]	Subtract From Minus One Extended
XO	31	200	SR		62	B	subfze[o][.]	Subtract From Zero Extended
X	31	598			372	B	sync	Synchronize
X	31	566		H	453, 561, 651	B	tlbsync	TLB Synchronize
X	31	4			69	B	tw	Trap Word
D	3				69	B	twi	Trap Word Immediate
X	31	316	SR		73	B	xor[.]	XOR
D	26				72	B	xori	XOR Immediate
D	27				72	B	xoris	XOR Immediate Shifted
A	31	15			70	B.in	isel	Integer Select
XFX	31	19			90	B.in	mfocrf	Move From One Condition Register Field
XFX	31	144			90	B.in	mtocrf	Move To One Condition Register Field
X	31	122			76	B.in	popcntb	Population Count Bytes
X	31	758			360	E	dcba	Data Cache Block Allocate
X	31	470		P	554	E	dcbi	Data Cache Block Invalidate
X	31	22			359	E	icbt	Instruction Cache Block Touch
X	31	854			374	E	mbar	Memory Barrier
X	31	275			91	E	mfapidi	Move From APID Indirect
XFX	31	323		S	527	E	mfdcr	Move From Device Control Register

Form	Opcode Pri	Opcode Ext	Mode Dep.[1]	Priv[1]	Page	Cat[1]	Mnemonic	Instruction
X	31	291			91	E	mfdcrux	Move From Device Control Register User-mode Indexed
X	31	259		P	527	E	mfdcrx	Move From Device Control Register Indexed
XFX	31	451		P	526	E	mtdcr	Move To Device Control Register
X	31	419			91	E	mtdcrux	Move To Device Control Register User-mode Indexed
X	31	387		P	526	E	mtdcrx	Move To Device Control Register Indexed
X	31	146		P	527	E	mtmsr	Move To Machine State Register
XL	19	51		P	516	E	rfci	Return From Critical Interrupt
XL	19	50		P	515	E	rfi	Return From Interrupt
XL	19	38		P	516	E	rfmci	Return From Machine Check Interrupt
X	31	786		P	560, 649	E	tlbivax	TLB Invalidate Virtual Address Indexed
X	31	946		P	560, 650	E	tlbre	TLB Read Entry
X	31	914		P	561, 650	E	tlbsx	TLB Search Indexed
X	31	978		P	562, 651	E	tlbwe	TLB Write Entry
X	31	131		S	528	E	wrtee	Write MSR External Enable
X	31	163		S	528	E	wrteei	Write MSR External Enable Immediate
X	31	326			632	E.CD	dcread	Data Cache Read [Alternative Encoding]
X	31	486			632	E.CD	dcread	Data Cache Read
X	31	998			633	E.CD	icread	Instruction Cache Read
X	31	454			629	E.CI	dci	Data Cache Invalidate
X	31	966			629	E.CI	ici	Instruction Cache Invalidate
XFX	19	198			620	E.ED	dnh	Debugger Notify Halt
X	19	39			516	E.ED	rfdi	Return From Debug Interrupt
X	31	238			623	E.PC	msgclr	Message Clear
X	31	206			623	E.PC	msgsnd	Message Send
X	31	127			534	E.PD	dcbfep	Data Cache Block Flush by External PID
X	31	63			533	E.PD	dcbstep	Data Cache Block Store by External PID
X	31	319			533	E.PD	dcbtep	Data Cache Block Touch by External PID
X	31	255			535	E.PD	dcbtstep	Data Cache Block Touch for Store by External PID
X	31	1023			536	E.PD	dcbzep	Data Cache Block set to Zero by External PID
EVX	31	285			538	E.PD	evlddepx	Vector Load Doubleword into Doubleword by External Process ID Indexed
EVX	31	413			538	E.PD	evstddepx	Vector Store Doubleword into Doubleword by External Process ID Indexed
X	31	991			536	E.PD	icbiep	Instruction Cache Block Invalidate by External PID
X	31	95			529	E.PD	lbepx	Load Byte by External Process ID Indexed
X	31	29			530	E.PD	ldepx	Load Doubleword by External Process ID Indexed
X	31	607			537	E.PD	lfdepx	Load Floating-Point Double by External Process ID Indexed
X	31	287			529	E.PD	lhepx	Load Halfword by External Process ID Indexed
X	31	295			539	E.PD	lvepx	Load Vector by External Process ID Indexed
X	31	263			539	E.PD	lvepxl	Load Vector by External Process ID Indexed LRU
X	31	31			530	E.PD	lwepx	Load Word by External Process ID Indexed
X	31	223			531	E.PD	stbepx	Store Byte by External Process ID Indexed
X	31	157			532	E.PD	stdepx	Store Doubleword by External Process ID Indexed
X	31	735			537	E.PD	stfdepx	Store Floating-Point Double by External Process ID Indexed
X	31	415			531	E.PD	sthepx	Store Halfword by External Process ID Indexed
X	31	807			540	E.PD	stvepx	Store Vector by External Process ID Indexed
X	31	775			540	E.PD	stvepxl	Store Vector by External Process ID Indexed LRU
X	31	159			532	E.PD	stwepx	Store Word by External Process ID Indexed
XFX	31	334			658	E.PM	mfpmr	Move From Performance Monitor Register
XFX	31	462			658	E.PM	mtpmr	Move To Performance Monitor Register
X	31	310			382	EC	eciwx	External Control In Word Indexed
X	31	438			382	EC	ecowx	External Control Out Word Indexed
X	31	390			558	ECL	dcblc	Data Cache Block Lock Clear
X	31	166			557	ECL	dcbtls	Data Cache Block Touch and Lock Set

Form	Opcode Pri	Opcode Ext	Mode Dep.[1]	Priv[1]	Page	Cat[1]	Mnemonic	Instruction
X	31	134			557	ECL	dcbtstls	Data Cache Block Touch for Store and Lock Set
X	31	230			559	ECL	icblc	Instruction Cache Block Lock Clear
X	31	486			558	ECL	icbtls	Instruction Cache Block Touch and Lock Set
X	63	32			129	FP	fcmpo	Floating Compare Ordered
X	63	0			129	FP	fcmpu	Floating Compare Unordered
D	50				113	FP	lfd	Load Floating-Point Double
D	51				113	FP	lfdu	Load Floating-Point Double with Update
X	31	631			113	FP	lfdux	Load Floating-Point Double with Update Indexed
X	31	599			113	FP	lfdx	Load Floating-Point Double Indexed
D	48				115	FP	lfs	Load Floating-Point Single
D	49				115	FP	lfsu	Load Floating-Point Single with Update
X	31	567			115	FP	lfsux	Load Floating-Point Single with Update Indexed
X	31	535			115	FP	lfsx	Load Floating-Point Single Indexed
X	63	64			131	FP	mcrfs	Move to Condition Register from FPSCR
D	54				116	FP	stfd	Store Floating-Point Double
D	55				116	FP	stfdu	Store Floating-Point Double with Update
X	31	759			116	FP	stfdux	Store Floating-Point Double with Update Indexed
X	31	727			116	FP	stfdx	Store Floating-Point Double Indexed
X	31	983			117	FP	stfiwx	Store Floating-Point as Integer Word Indexed
D	52				115	FP	stfs	Store Floating-Point Single
D	53				115	FP	stfsu	Store Floating-Point Single with Update
X	31	695			115	FP	stfsux	Store Floating-Point Single with Update Indexed
X	31	663			115	FP	stfsx	Store Floating-Point Single Indexed
X	63	264			118	FP[R]	fabs[.]	Floating Absolute Value
A	63	21			119	FP[R]	fadd[.]	Floating Add
A	59	21			119	FP[R]	fadds[.]	Floating Add Single
X	63	846			127	FP[R]	fcfid[.]	Floating Convert From Integer Doubleword
X	63	814			125	FP[R]	fctid[.]	Floating Convert To Integer Doubleword
X	63	815			126	FP[R]	fctidz[.]	Floating Convert To Integer Doubleword with round toward Zero
X	63	14			126	FP[R]	fctiw[.]	Floating Convert To Integer Word
X	63	15			127	FP[R]	fctiwz[.]	Floating Convert To Integer Word with round toward Zero
A	63	18			120	FP[R]	fdiv[.]	Floating Divide
A	59	18			120	FP[R]	fdivs[.]	Floating Divide Single
A	63	29			123	FP[R]	fmadd[.]	Floating Multiply-Add
A	59	29			123	FP[R]	fmadds[.]	Floating Multiply-Add Single
X	63	72			118	FP[R]	fmr[.]	Floating Move Register
A	63	28			123	FP[R]	fmsub[.]	Floating Multiply-Subtract
A	59	28			123	FP[R]	fmsubs[.]	Floating Multiply-Subtract Single
A	63	25			120	FP[R]	fmul[.]	Floating Multiply
A	59	25			120	FP[R]	fmuls[.]	Floating Multiply Single
X	63	136			118	FP[R]	fnabs[.]	Floating Negative Absolute Value
X	63	40			118	FP[R]	fneg[.]	Floating Negate
A	63	31			124	FP[R]	fnmadd[.]	Floating Negative Multiply-Add
A	59	31			124	FP[R]	fnmadds[.]	Floating Negative Multiply-Add Single
A	63	30			124	FP[R]	fnmsub[.]	Floating Negative Multiply-Subtract
A	59	30			124	FP[R]	fnmsubs[.]	Floating Negative Multiply-Subtract Single
A	63	24			121	FP[R]	fre[.]	Floating Reciprocal Estimate
A	59	24			121	FP[R]	fres[.]	Floating Reciprocal Estimate Single
X	63	488			128	FP[R]	frim[.]	Floating Round to Integer Minus
A	63	23			130	FP[R]	fsel[.]	Floating Select
A	63	22			121	FP[R]	fsqrt[.]	Floating Square Root
A	59	22			121	FP[R]	fsqrts[.]	Floating Square Root Single
A	63	20			119	FP[R]	fsub[.]	Floating Subtract
A	59	20			119	FP[R]	fsubs[.]	Floating Subtract Single
X	63	583			131	FP[R]	mffs[.]	Move From FPSCR
X	63	70			132	FP[R]	mtfsb0[.]	Move To FPSCR Bit 0
X	63	38			132	FP[R]	mtfsb1[.]	Move To FPSCR Bit 1
XFL	63	711			131	FP[R]	mtfsf[.]	Move To FPSCR Fields
X	63	134			131	FP[R]	mtfsfi[.]	Move To FPSCR Field Immediate

Form	Opcode Pri	Opcode Ext	Mode Dep.[1]	Priv[1]	Page	Cat[1]	Mnemonic	Instruction
X	63	392			128	FP[R].in	frin[.]	Floating Round to Integer Nearest
X	63	456			128	FP[R].in	frip[.]	Floating Round to Integer Plus
X	63	424			128	FP[R].in	friz[.]	Floating Round to Integer Toward Zero
X	63	12			125	FP[R].in	frsp[.]	Floating Round to Single-Precision
A	63	26			122	FP[R].in	frsqrte[.]	Floating Reciprocal Square Root Estimate
A	59	26			122	FP[R].in	frsqrtes[.]	Floating Reciprocal Square Root Estimate Single
XO	4	172			289	LMA	macchw[o][.]	Multiply Accumulate Cross Halfword to Word Modulo Signed
XO	4	236			289	LMA	macchws[o][.]	Multiply Accumulate Cross Halfword to Word Saturate Signed
XO	4	204			290	LMA	macchwsu[o][.]	Multiply Accumulate Cross Halfword to Word Saturate Unsigned
XO	4	140			290	LMA	macchwu[o][.]	Multiply Accumulate Cross Halfword to Word Modulo Unsigned
XO	4	44			291	LMA	machhw[o][.]	Multiply Accumulate High Halfword to Word Modulo Signed
XO	4	108			291	LMA	machhws[o][.]	Multiply Accumulate High Halfword to Word Saturate Signed
XO	4	76			292	LMA	machhwsu[o][.]	Multiply Accumulate High Halfword to Word Saturate Unsigned
XO	4	12			292	LMA	machhwu[o][.]	Multiply Accumulate High Halfword to Word Modulo Unsigned
XO	4	428			293	LMA	maclhw[o][.]	Multiply Accumulate Low Halfword to Word Modulo Signed
XO	4	492			293	LMA	maclhws[o][.]	Multiply Accumulate Low Halfword to Word Saturate Signed
XO	4	460			294	LMA	maclhwsu[o][.]	Multiply Accumulate Low Halfword to Word Saturate Unsigned
XO	4	396			294	LMA	maclhwu[o][.]	Multiply Accumulate Low Halfword to Word Modulo Unsigned
X	4	168			294	LMA	mulchw[.]	Multiply Cross Halfword to Word Signed
X	4	136			294	LMA	mulchwu[.]	Multiply Cross Halfword to Word Unsigned
X	4	40			295	LMA	mulhhw[.]	Multiply High Halfword to Word Signed
X	4	8			295	LMA	mulhhwu[.]	Multiply High Halfword to Word Unsigned
X	4	424			295	LMA	mullhw[.]	Multiply Low Halfword to Word Signed
X	4	392			295	LMA	mullhwu[.]	Multiply Low Halfword to Word Unsigned
XO	4	174			296	LMA	nmacchw[o][.]	Negative Multiply Accumulate Cross Halfword to Word Modulo Signed
XO	4	238			296	LMA	nmacchws[o][.]	Negative Multiply Accumulate Cross Halfword to Word Saturate Signed
XO	4	46			297	LMA	nmachhw[o][.]	Negative Multiply Accumulate High Halfword to Word Modulo Signed
XO	4	110			297	LMA	nmachhws[o][.]	Negative Multiply Accumulate High Halfword to Word Saturate Signed
XO	4	430			298	LMA	nmaclhw[o][.]	Negative Multiply Accumulate Low Halfword to Word Modulo Signed
XO	4	494			298	LMA	nmaclhws[o][.]	Negative Multiply Accumulate Low Halfword to Word Saturate Signed
X	31	78			287	LMV	dlmzb[.]	Determine Leftmost Zero Byte
DQ	56			P	410	LSQ	lq	Load Quadword
DS	62	2		P	410	LSQ	stq	Store Quadword
X	31	597			55	MA	lswi	Load String Word Immediate
X	31	533			55	MA	lswx	Load String Word Indexed
X	31	725			56	MA	stswi	Store String Word Immediate
X	31	661			56	MA	stswx	Store String Word Indexed
X	31	854			374	S	eieio	Enforce In-order Execution of I/O
XL	19	274		H	405	S	hrfid	Hypervisor Return From Interrupt Doubleword
X	31	595	32	P	449	S	mfsr	Move From Segment Register
X	31	659	32	P	449	S	mfsrin	Move From Segment Register Indirect
XFX	31	371			378	S	mftb	Move From Time Base

Form	Opcode Pri	Opcode Ext	Mode Dep.[1]	Priv[1]	Page	Cat[1]	Mnemonic	Instruction
X	31	146		P	415	S	mtmsr	Move To Machine State Register
X	31	178		P	416	S	mtmsrd	Move To Machine State Register Doubleword
X	31	210	32	P	448	S	mtsr	Move To Segment Register
X	31	242	32	P	448	S	mtsrin	Move To Segment Register Indirect
XL	19	18		P	405	S	rfid	Return From Interrupt Doubleword
X	31	498		P	444	S	slbia	SLB Invalidate All
X	31	434		P	443	S	slbie	SLB Invalidate Entry
X	31	915		P	446	S	slbmfee	SLB Move From Entry ESID
X	31	851		P	446	S	slbmfev	SLB Move From Entry VSID
X	31	402		P	445	S	slbmte	SLB Move To Entry
X	31	370		P	453	S	tlbia	TLB Invalidate All
X	31	306	64	H	450	S	tlbie	TLB Invalidate Entry
X	31	274	64	H	452	S	tlbiel	TLB Invalidate Entry Local
EVX	4	527			208	SP	brinc	Bit Reversed Increment
EVX	4	520			208	SP	evabs	Vector Absolute Value
EVX	4	514			208	SP	evaddiw	Vector Add Immediate Word
EVX	4	1225			208	SP	evaddsmiaaw	Vector Add Signed, Modulo, Integer to Accumulator Word
EVX	4	1217			209	SP	evaddssiaaw	Vector Add Signed, Saturate, Integer to Accumulator Word
EVX	4	1224			209	SP	evaddumiaaw	Vector Add Unsigned, Modulo, Integer to Accumulator Word
EVX	4	1216			209	SP	evaddusiaaw	Vector Add Unsigned, Saturate, Integer to Accumulator Word
EVX	4	512			209	SP	evaddw	Vector Add Word
EVX	4	529			210	SP	evand	Vector AND
EVX	4	530			210	SP	evandc	Vector AND with Complement
EVX	4	564			210	SP	evcmpeq	Vector Compare Equal
EVX	4	561			210	SP	evcmpgts	Vector Compare Greater Than Signed
EVX	4	560			211	SP	evcmpgtu	Vector Compare Greater Than Unsigned
EVX	4	563			211	SP	evcmplts	Vector Compare Less Than Signed
EVX	4	562			211	SP	evcmpltu	Vector Compare Less Than Unsigned
EVX	4	526			212	SP	evcntlsw	Vector Count Leading Signed Bits Word
EVX	4	525			212	SP	evcntlzw	Vector Count Leading Zeros Word
EVX	4	1222			212	SP	evdivws	Vector Divide Word Signed
EVX	4	1223			213	SP	evdivwu	Vector Divide Word Unsigned
EVX	4	537			213	SP	eveqv	Vector Equivalent
EVX	4	522			213	SP	evextsb	Vector Extend Sign Byte
EVX	4	523			213	SP	evextsh	Vector Extend Sign Halfword
EVX	4	769			214	SP	evldd	Vector Load Double Word into Double Word
EVX	4	768			214	SP	evlddx	Vector Load Double Word into Double Word Indexed
EVX	4	773			214	SP	evldh	Vector Load Double into Four Halfwords
EVX	4	772			214	SP	evldhx	Vector Load Double into Four Halfwords Indexed
EVX	4	771			215	SP	evldw	Vector Load Double into Two Words
EVX	4	770			215	SP	evldwx	Vector Load Double into Two Words Indexed
EVX	4	777			215	SP	evlhhesplat	Vector Load Halfword into Halfwords Even and Splat
EVX	4	776			215	SP	evlhhesplatx	Vector Load Halfword into Halfwords Even and Splat Indexed
EVX	4	783			216	SP	evlhhossplat	Vector Load Halfword into Halfword Odd Signed and Splat
EVX	4	782			216	SP	evlhhossplatx	Vector Load Halfword into Halfword Odd Signed and Splat Indexed
EVX	4	781			216	SP	evlhhousplat	Vector Load Halfword into Halfword Odd Unsigned and Splat
EVX	4	780			216	SP	evlhhousplatx	Vector Load Halfword into Halfword Odd Unsigned and Splat Indexed
EVX	4	785			217	SP	evlwhe	Vector Load Word into Two Halfwords Even
EVX	4	784			217	SP	evlwhex	Vector Load Word into Two Halfwords Even Indexed
EVX	4	791			217	SP	evlwhos	Vector Load Word into Two Halfwords Odd Signed (with sign extension)

Form	Opcode Pri	Opcode Ext	Mode Dep.[1]	Priv[1]	Page	Cat[1]	Mnemonic	Instruction
EVX	4	790			217	SP	evlwhosx	Vector Load Word into Two Halfwords Odd Signed Indexed (with sign extension)
EVX	4	789			218	SP	evlwhou	Vector Load Word into Two Halfwords Odd Unsigned (zero-extended)
EVX	4	788			218	SP	evlwhoux	Vector Load Word into Two Halfwords Odd Unsigned Indexed (zero-extended)
EVX	4	797			218	SP	evlwhsplat	Vector Load Word into Two Halfwords and Splat
EVX	4	796			218	SP	evlwhsplatx	Vector Load Word into Two Halfwords and Splat Indexed
EVX	4	793			219	SP	evlwwsplat	Vector Load Word into Word and Splat
EVX	4	792			219	SP	evlwwsplatx	Vector Load Word into Word and Splat Indexed
EVX	4	556			219	SP	evmergehi	Vector Merge High
EVX	4	558			220	SP	evmergehilo	Vector Merge High/Low
EVX	4	557			219	SP	evmergelo	Vector Merge Low
EVX	4	559			220	SP	evmergelohi	Vector Merge Low/High
EVX	4	1323			220	SP	evmhegsmfaa	Vector Multiply Halfwords, Even, Guarded, Signed, Modulo, Fractional and Accumulate
EVX	4	1451			220	SP	evmhegsmfan	Vector Multiply Halfwords, Even, Guarded, Signed, Modulo, Fractional and Accumulate Negative
EVX	4	1321			221	SP	evmhegsmiaa	Vector Multiply Halfwords, Even, Guarded, Signed, Modulo, Integer and Accumulate
EVX	4	1449			221	SP	evmhegsmian	Vector Multiply Halfwords, Even, Guarded, Signed, Modulo, Integer and Accumulate Negative
EVX	4	1320			221	SP	evmhegumiaa	Vector Multiply Halfwords, Even, Guarded, Unsigned, Modulo, Integer and Accumulate
EVX	4	1448			221	SP	evmhegumian	Vector Multiply Halfwords, Even, Guarded, Unsigned, Modulo, Integer and Accumulate Negative
EVX	4	1035			222	SP	evmhesmf	Vector Multiply Halfwords, Even, Signed, Modulo, Fractional
EVX	4	1067			222	SP	evmhesmfa	Vector Multiply Halfwords, Even, Signed, Modulo, Fractional to Accumulator
EVX	4	1291			222	SP	evmhesmfaaw	Vector Multiply Halfwords, Even, Signed, Modulo, Fractional and Accumulate into Words
EVX	4	1419			222	SP	evmhesmfanw	Vector Multiply Halfwords, Even, Signed, Modulo, Fractional and Accumulate Negative into Words
EVX	4	1033			223	SP	evmhesmi	Vector Multiply Halfwords, Even, Signed, Modulo, Integer
EVX	4	1065			223	SP	evmhesmia	Vector Multiply Halfwords, Even, Signed, Modulo, Integer to Accumulator
EVX	4	1289			223	SP	evmhesmiaaw	Vector Multiply Halfwords, Even, Signed, Modulo, Integer and Accumulate into Words
EVX	4	1417			223	SP	evmhesmianw	Vector Multiply Halfwords, Even, Signed, Modulo, Integer and Accumulate Negative into Words
EVX	4	1027			224	SP	evmhessf	Vector Multiply Halfwords, Even, Signed, Saturate, Fractional
EVX	4	1059			224	SP	evmhessfa	Vector Multiply Halfwords, Even, Signed, Saturate, Fractional to Accumulator
EVX	4	1283			225	SP	evmhessfaaw	Vector Multiply Halfwords, Even, Signed, Saturate, Fractional and Accumulate into Words
EVX	4	1411			225	SP	evmhessfanw	Vector Multiply Halfwords, Even, Signed, Saturate, Fractional and Accumulate Negative into Words
EVX	4	1281			226	SP	evmhessiaaw	Vector Multiply Halfwords, Even, Signed, Saturate, Integer and Accumulate into Words
EVX	4	1409			226	SP	evmhessianw	Vector Multiply Halfwords, Even, Signed, Saturate, Integer and Accumulate Negative into Words
EVX	4	1032			227	SP	evmheumi	Vector Multiply Halfwords, Even, Unsigned, Modulo, Integer
EVX	4	1064			227	SP	evmheumia	Vector Multiply Halfwords, Even, Unsigned, Modulo, Integer to Accumulator

Form	Opcode Pri	Opcode Ext	Mode Dep.[1]	Priv[1]	Page	Cat[1]	Mnemonic	Instruction
EVX	4	1288			227	SP	evmheumiaaw	Vector Multiply Halfwords, Even, Unsigned, Modulo, Integer and Accumulate into Words
EVX	4	1416			227	SP	evmheumianw	Vector Multiply Halfwords, Even, Unsigned, Modulo, Integer and Accumulate Negative into Words
EVX	4	1280			228	SP	evmheusiaaw	Vector Multiply Halfwords, Even, Unsigned, Saturate, Integer and Accumulate into Words
EVX	4	1408			228	SP	evmheusianw	Vector Multiply Halfwords, Even, Unsigned, Saturate, Integer and Accumulate Negative into Words
EVX	4	1327			229	SP	evmhogsmfaa	Vector Multiply Halfwords, Odd, Guarded, Signed, Modulo, Fractional and Accumulate
EVX	4	1455			229	SP	evmhogsmfan	Vector Multiply Halfwords, Odd, Guarded, Signed, Modulo, Fractional and Accumulate Negative
EVX	4	1325			229	SP	evmhogsmiaa	Vector Multiply Halfwords, Odd, Guarded, Signed, Modulo, Integer and Accumulate
EVX	4	1453			229	SP	evmhogsmian	Vector Multiply Halfwords, Odd, Guarded, Signed, Modulo, Integer and Accumulate Negative
EVX	4	1324			230	SP	evmhogumiaa	Vector Multiply Halfwords, Odd, Guarded, Unsigned, Modulo, Integer and Accumulate
EVX	4	1452			230	SP	evmhogumian	Vector Multiply Halfwords, Odd, Guarded, Unsigned, Modulo, Integer and Accumulate Negative
EVX	4	1039			230	SP	evmhosmf	Vector Multiply Halfwords, Odd, Signed, Modulo, Fractional
EVX	4	1071			230	SP	evmhosmfa	Vector Multiply Halfwords, Odd, Signed, Modulo, Fractional to Accumulator
EVX	4	1295			231	SP	evmhosmfaaw	Vector Multiply Halfwords, Odd, Signed, Modulo, Fractional and Accumulate into Words
EVX	4	1423			231	SP	evmhosmfanw	Vector Multiply Halfwords, Odd, Signed, Modulo, Fractional and Accumulate Negative into Words
EVX	4	1037			231	SP	evmhosmi	Vector Multiply Halfwords, Odd, Signed, Modulo, Integer
EVX	4	1069			231	SP	evmhosmia	Vector Multiply Halfwords, Odd, Signed, Modulo, Integer to Accumulator
EVX	4	1293			232	SP	evmhosmiaaw	Vector Multiply Halfwords, Odd, Signed, Modulo, Integer and Accumulate into Words
EVX	4	1421			231	SP	evmhosmianw	Vector Multiply Halfwords, Odd, Signed, Modulo, Integer and Accumulate Negative into Words
EVX	4	1031			233	SP	evmhossf	Vector Multiply Halfwords, Odd, Signed, Saturate, Fractional
EVX	4	1063			233	SP	evmhossfa	Vector Multiply Halfwords, Odd, Signed, Saturate, Fractional to Accumulator
EVX	4	1287			234	SP	evmhossfaaw	Vector Multiply Halfwords, Odd, Signed, Saturate, Fractional and Accumulate into Words
EVX	4	1415			234	SP	evmhossfanw	Vector Multiply Halfwords, Odd, Signed, Saturate, Fractional and Accumulate Negative into Words
EVX	4	1285			235	SP	evmhossiaaw	Vector Multiply Halfwords, Odd, Signed, Saturate, Integer and Accumulate into Words
EVX	4	1413			235	SP	evmhossianw	Vector Multiply Halfwords, Odd, Signed, Saturate, Integer and Accumulate Negative into Words
EVX	4	1036			235	SP	evmhoumi	Vector Multiply Halfwords, Odd, Unsigned, Modulo, Integer
EVX	4	1068			235	SP	evmhoumia	Vector Multiply Halfwords, Odd, Unsigned, Modulo, Integer to Accumulator
EVX	4	1292			236	SP	evmhoumiaaw	Vector Multiply Halfwords, Odd, Unsigned, Modulo, Integer and Accumulate into Words
EVX	4	1420			232	SP	evmhoumianw	Vector Multiply Halfwords, Odd, Unsigned, Modulo, Integer and Accumulate Negative into Words
EVX	4	1284			236	SP	evmhousiaaw	Vector Multiply Halfwords, Odd, Unsigned, Saturate, Integer and Accumulate into Words

Form	Opcode Pri	Opcode Ext	Mode Dep.[1]	Priv[1]	Page	Cat[1]	Mnemonic	Instruction
EVX	4	1412			236	SP	evmhousianw	Vector Multiply Halfwords, Odd, Unsigned, Saturate, Integer and Accumulate Negative into Words
EVX	4	1220			237	SP	evmra	Initialize Accumulator
EVX	4	1103			237	SP	evmwhsmf	Vector Multiply Word High Signed, Modulo, Fractional
EVX	4	1135			237	SP	evmwhsmfa	Vector Multiply Word High Signed, Modulo, Fractional to Accumulator
EVX	4	1101			237	SP	evmwhsmi	Vector Multiply Word High Signed, Modulo, Integer
EVX	4	1133			237	SP	evmwhsmia	Vector Multiply Word High Signed, Modulo, Integer to Accumulator
EVX	4	1095			238	SP	evmwhssf	Vector Multiply Word High Signed, Saturate, Fractional
EVX	4	1127			238	SP	evmwhssfa	Vector Multiply Word High Signed, Saturate, Fractional to Accumulator
EVX	4	1100			238	SP	evmwhumi	Vector Multiply Word High Unsigned, Modulo, Integer
EVX	4	1132			238	SP	evmwhumia	Vector Multiply Word High Unsigned, Modulo, Integer to Accumulator
EVX	4	1353			239	SP	evmwlsmiaaw	Vector Multiply Word Low Signed, Modulo, Integer and Accumulate into Words
EVX	4	1481			239	SP	evmwlsmianw	Vector Multiply Word Low Signed, Modulo, Integer and Accumulate Negative in Words
EVX	4	1345			239	SP	evmwlssiaaw	Vector Multiply Word Low Signed, Saturate, Integer and Accumulate into Words
EVX	4	1473			239	SP	evmwlssianw	Vector Multiply Word Low Signed, Saturate, Integer and Accumulate Negative in Words
EVX	4	1096			240	SP	evmwlumi	Vector Multiply Word Low Unsigned, Modulo, Integer
EVX	4	1128			240	SP	evmwlumia	Vector Multiply Word Low Unsigned, Modulo, Integer to Accumulator
EVX	4	1352			240	SP	evmwlumiaaw	Vector Multiply Word Low Unsigned, Modulo, Integer and Accumulate into Words
EVX	4	1480			240	SP	evmwlumianw	Vector Multiply Word Low Unsigned, Modulo, Integer and Accumulate Negative in Words
EVX	4	1344			241	SP	evmwlusiaaw	Vector Multiply Word Low Unsigned, Saturate, Integer and Accumulate into Words
EVX	4	1472			241	SP	evmwlusianw	Vector Multiply Word Low Unsigned, Saturate, Integer and Accumulate Negative in Words
EVX	4	1115			241	SP	evmwsmf	Vector Multiply Word Signed, Modulo, Fractional
EVX	4	1147			241	SP	evmwsmfa	Vector Multiply Word Signed, Modulo, Fractional to Accumulator
EVX	4	1371			242	SP	evmwsmfaa	Vector Multiply Word Signed, Modulo, Fractional and Accumulate
EVX	4	1499			242	SP	evmwsmfan	Vector Multiply Word Signed, Modulo, Fractional and Accumulate Negative
EVX	4	1113			242	SP	evmwsmi	Vector Multiply Word Signed, Modulo, Integer
EVX	4	1145			242	SP	evmwsmia	Vector Multiply Word Signed, Modulo, Integer to Accumulator
EVX	4	1369			242	SP	evmwsmiaa	Vector Multiply Word Signed, Modulo, Integer and Accumulate
EVX	4	1497			242	SP	evmwsmian	Vector Multiply Word Signed, Modulo, Integer and Accumulate Negative
EVX	4	1107			243	SP	evmwssf	Vector Multiply Word Signed, Saturate, Fractional
EVX	4	1139			243	SP	evmwssfa	Vector Multiply Word Signed, Saturate, Fractional to Accumulator
EVX	4	1363			243	SP	evmwssfaa	Vector Multiply Word Signed, Saturate, Fractional and Accumulate
EVX	4	1491			244	SP	evmwssfan	Vector Multiply Word Signed, Saturate, Fractional and Accumulate Negative
EVX	4	1112			244	SP	evmwumi	Vector Multiply Word Unsigned, Modulo, Integer
EVX	4	1144			244	SP	evmwumia	Vector Multiply Word Unsigned, Modulo, Integer to Accumulator

Form	Opcode Pri	Opcode Ext	Mode Dep.[1]	Priv[1]	Page	Cat[1]	Mnemonic	Instruction
EVX	4	1368			245	SP	evmwumiaa	Vector Multiply Word Unsigned, Modulo, Integer and Accumulate
EVX	4	1496			245	SP	evmwumian	Vector Multiply Word Unsigned, Modulo, Integer and Accumulate Negative
EVX	4	542			245	SP	evnand	Vector NAND
EVX	4	521			245	SP	evneg	Vector Negate
EVX	4	536			245	SP	evnor	Vector NOR
EVX	4	535			246	SP	evor	Vector OR
EVX	4	539			246	SP	evorc	Vector OR with Complement
EVX	4	552			246	SP	evrlw	Vector Rotate Left Word
EVX	4	554			247	SP	evrlwi	Vector Rotate Left Word Immediate
EVX	4	524			247	SP	evrndw	Vector Round Word
EVS	4	79			247	SP	evsel	Vector Select
EVX	4	548			248	SP	evslw	Vector Shift Left Word
EVX	4	550			248	SP	evslwi	Vector Shift Left Word Immediate
EVX	4	555			248	SP	evsplatfi	Vector Splat Fractional Immediate
EVX	4	553			248	SP	evsplati	Vector Splat Immediate
EVX	4	547			248	SP	evsrwis	Vector Shift Right Word Immediate Signed
EVX	4	546			248	SP	evsrwiu	Vector Shift Right Word Immediate Unsigned
EVX	4	545			249	SP	evsrws	Vector Shift Right Word Signed
EVX	4	544			249	SP	evsrwu	Vector Shift Right Word Unsigned
EVX	4	801			249	SP	evstdd	Vector Store Double of Double
EVX	4	800			249	SP	evstddx	Vector Store Double of Double Indexed
EVX	4	805			250	SP	evstdh	Vector Store Double of Four Halfwords
EVX	4	804			250	SP	evstdhx	Vector Store Double of Four Halfwords Indexed
EVX	4	803			250	SP	evstdw	Vector Store Double of Two Words
EVX	4	802			250	SP	evstdwx	Vector Store Double of Two Words Indexed
EVX	4	817			251	SP	evstwhe	Vector Store Word of Two Halfwords from Even
EVX	4	816			251	SP	evstwhex	Vector Store Word of Two Halfwords from Even Indexed
EVX	4	821			251	SP	evstwho	Vector Store Word of Two Halfwords from Odd
EVX	4	820			251	SP	evstwhox	Vector Store Word of Two Halfwords from Odd Indexed
EVX	4	825			251	SP	evstwwe	Vector Store Word of Word from Even
EVX	4	824			251	SP	evstwwex	Vector Store Word of Word from Even Indexed
EVX	4	829			252	SP	evstwwo	Vector Store Word of Word from Odd
EVX	4	828			252	SP	evstwwox	Vector Store Word of Word from Odd Indexed
EVX	4	1227			252	SP	evsubfsmiaaw	Vector Subtract Signed, Modulo, Integer to Accumulator Word
EVX	4	1219			252	SP	evsubfssiaaw	Vector Subtract Signed, Saturate, Integer to Accumulator Word
EVX	4	1226			253	SP	evsubfumiaaw	Vector Subtract Unsigned, Modulo, Integer to Accumulator Word
EVX	4	1218			253	SP	evsubfusiaaw	Vector Subtract Unsigned, Saturate, Integer to Accumulator Word
EVX	4	516			253	SP	evsubfw	Vector Subtract from Word
EVX	4	518			253	SP	evsubifw	Vector Subtract Immediate from Word
EVX	4	534			253	SP	evxor	Vector XOR
EVX	4	740			274	SP.FD	efdabs	Floating-Point Double-Precision Absolute Value
EVX	4	736			275	SP.FD	efdadd	Floating-Point Double-Precision Add
EVX	4	751			280	SP.FD	efdcfs	Floating-Point Double-Precision Convert from Single-Precision
EVX	4	755			278	SP.FD	efdcfsf	Convert Floating-Point Double-Precision from Signed Fraction
EVX	4	753			277	SP.FD	efdcfsi	Convert Floating-Point Double-Precision from Signed Integer
EVX	4	739			278	SP.FD	efdcfsid	Convert Floating-Point Double-Precision from Signed Integer Doubleword
EVX	4	754			278	SP.FD	efdcfuf	Convert Floating-Point Double-Precision from Unsigned Fraction
EVX	4	752			277	SP.FD	efdcfui	Convert Floating-Point Double-Precision from Unsigned Integer

Form	Opcode		Mode Dep.[1]	Priv[1]	Page	Cat[1]	Mnemonic	Instruction
	Pri	Ext						
EVX	4	738			278	SP.FD	efdcfuid	Convert Floating-Point Double-Precision from Unsigned Integer Doubleword
EVX	4	750			276	SP.FD	efdcmpeq	Floating-Point Double-Precision Compare Equal
EVX	4	748			276	SP.FD	efdcmpgt	Floating-Point Double-Precision Compare Greater Than
EVX	4	749			276	SP.FD	efdcmplt	Floating-Point Double-Precision Compare Less Than
EVX	4	759			280	SP.FD	efdctsf	Convert Floating-Point Double-Precision to Signed Fraction
EVX	4	757			278	SP.FD	efdctsi	Convert Floating-Point Double-Precision to Signed Integer
EVX	4	747			279	SP.FD	efdctsidz	Convert Floating-Point Double-Precision to Signed Integer Doubleword with Round toward Zero
EVX	4	762			280	SP.FD	efdctsiz	Convert Floating-Point Double-Precision to Signed Integer with Round toward Zero
EVX	4	758			280	SP.FD	efdctuf	Convert Floating-Point Double-Precision to Unsigned Fraction
EVX	4	756			278	SP.FD	efdctui	Convert Floating-Point Double-Precision to Unsigned Integer
EVX	4	746			279	SP.FD	efdctuidz	Convert Floating-Point Double-Precision to Unsigned Integer Doubleword with Round toward Zero
EVX	4	760			280	SP.FD	efdctuiz	Convert Floating-Point Double-Precision to Unsigned Integer with Round toward Zero
EVX	4	745			275	SP.FD	efddiv	Floating-Point Double-Precision Divide
EVX	4	744			275	SP.FD	efdmul	Floating-Point Double-Precision Multiply
EVX	4	741			274	SP.FD	efdnabs	Floating-Point Double-Precision Negative Absolute Value
EVX	4	742			274	SP.FD	efdneg	Floating-Point Double-Precision Negate
EVX	4	737			275	SP.FD	efdsub	Floating-Point Double-Precision Subtract
EVX	4	766			277	SP.FD	efdtsteq	Floating-Point Double-Precision Test Equal
EVX	4	764			276	SP.FD	efdtstgt	Floating-Point Double-Precision Test Greater Than
EVX	4	765			277	SP.FD	efdtstlt	Floating-Point Double-Precision Test Less Than
EVX	4	719			281	SP.FD	efscfd	Floating-Point Single-Precision Convert from Double-Precision
EVX	4	708			267	SP.FS	efsabs	Floating-Point Single-Precision Absolute Value
EVX	4	704			268	SP.FS	efsadd	Floating-Point Single-Precision Add
EVX	4	723			272	SP.FS	efscfsf	Convert Floating-Point Single-Precision from Signed Fraction
EVX	4	721			272	SP.FS	efscfsi	Convert Floating-Point Single-Precision from Signed Integer
EVX	4	722			272	SP.FS	efscfuf	Convert Floating-Point Single-Precision from Unsigned Fraction
EVX	4	720			272	SP.FS	efscfui	Convert Floating-Point Single-Precision from Unsigned Integer
EVX	4	718			270	SP.FS	efscmpeq	Floating-Point Single-Precision Compare Equal
EVX	4	716			269	SP.FS	efscmpgt	Floating-Point Single-Precision Compare Greater Than
EVX	4	717			269	SP.FS	efscmplt	Floating-Point Single-Precision Compare Less Than
EVX	4	727			273	SP.FS	efsctsf	Convert Floating-Point Single-Precision to Signed Fraction
EVX	4	725			272	SP.FS	efsctsi	Convert Floating-Point Single-Precision to Signed Integer
EVX	4	730			273	SP.FS	efsctsiz	Convert Floating-Point Single-Precision to Signed Integer with Round toward Zero
EVX	4	726			273	SP.FS	efsctuf	Convert Floating-Point Single-Precision to Unsigned Fraction
EVX	4	724			272	SP.FS	efsctui	Convert Floating-Point Single-Precision to Unsigned Integer
EVX	4	728			273	SP.FS	efsctuiz	Convert Floating-Point Single-Precision to Unsigned Integer with Round toward Zero
EVX	4	713			268	SP.FS	efsdiv	Floating-Point Single-Precision Divide
EVX	4	712			268	SP.FS	efsmul	Floating-Point Single-Precision Multiply

Form	Opcode Pri	Opcode Ext	Mode Dep.[1]	Priv[1]	Page	Cat[1]	Mnemonic	Instruction
EVX	4	709			267	SP.FS	efsnabs	Floating-Point Single-Precision Negative Absolute Value
EVX	4	710			267	SP.FS	efsneg	Floating-Point Single-Precision Negate
EVX	4	705			268	SP.FS	efssub	Floating-Point Single-Precision Subtract
EVX	4	734			271	SP.FS	efststeq	Floating-Point Single-Precision Test Equal
EVX	4	732			270	SP.FS	efststgt	Floating-Point Single-Precision Test Greater Than
EVX	4	733			271	SP.FS	efststlt	Floating-Point Single-Precision Test Less Than
EVX	4	644			259	SP.FV	evfsabs	Vector Floating-Point Single-Precision Absolute Value
EVX	4	640			260	SP.FV	evfsadd	Vector Floating-Point Single-Precision Add
EVX	4	659			264	SP.FV	evfscfsf	Vector Convert Floating-Point Single-Precision from Signed Fraction
EVX	4	657			264	SP.FV	evfscfsi	Vector Convert Floating-Point Single-Precision from Signed Integer
EVX	4	658			264	SP.FV	evfscfuf	Vector Convert Floating-Point Single-Precision from Unsigned Fraction
EVX	4	656			264	SP.FV	evfscfui	Vector Convert Floating-Point Single-Precision from Unsigned Integer
EVX	4	654			262	SP.FV	evfscmpeq	Vector Floating-Point Single-Precision Compare Equal
EVX	4	652			261	SP.FV	evfscmpgt	Vector Floating-Point Single-Precision Compare Greater Than
EVX	4	653			261	SP.FV	evfscmplt	Vector Floating-Point Single-Precision Compare Less Than
EVX	4	663			266	SP.FV	evfsctsf	Vector Convert Floating-Point Single-Precision to Signed Fraction
EVX	4	661			265	SP.FV	evfsctsi	Vector Convert Floating-Point Single-Precision to Signed Integer
EVX	4	666			265	SP.FV	evfsctsiz	Vector Convert Floating-Point Single-Precision to Signed Integer with Round toward Zero
EVX	4	662			266	SP.FV	evfsctuf	Vector Convert Floating-Point Single-Precision to Unsigned Fraction
EVX	4	660			265	SP.FV	evfsctui	Vector Convert Floating-Point Single-Precision to Unsigned Integer
EVX	4	664			265	SP.FV	evfsctuiz	Vector Convert Floating-Point Single-Precision to Unsigned Integer with Round toward Zero
EVX	4	649			260	SP.FV	evfsdiv	Vector Floating-Point Single-Precision Divide
EVX	4	648			260	SP.FV	evfsmul	Vector Floating-Point Single-Precision Multiply
EVX	4	645			259	SP.FV	evfsnabs	Vector Floating-Point Single-Precision Negative Absolute Value
EVX	4	646			259	SP.FV	evfsneg	Vector Floating-Point Single-Precision Negate
EVX	4	641			260	SP.FV	evfssub	Vector Floating-Point Single-Precision Subtract
EVX	4	670			263	SP.FV	evfststeq	Vector Floating-Point Single-Precision Test Equal
EVX	4	668			262	SP.FV	evfststgt	Vector Floating-Point Single-Precision Test Greater Than
EVX	4	669			263	SP.FV	evfststlt	Vector Floating-Point Single-Precision Test Less Than
X	31	7			146	V	lvebx	Load Vector Element Byte Indexed
X	31	39			143	V	lvehx	Load Vector Element Halfword Indexed
X	31	71			143	V	lvewx	Load Vector Element Word Indexed
X	31	6			148	V	lvsl	Load Vector for Shift Left Indexed
X	31	38			148	V	lvsr	Load Vector for Shift Right Indexed
X	31	103			144	V	lvx	Load Vector Indexed
X	31	359			144	V	lvxl	Load Vector Indexed Last
VX	4	1540			199	V	mfvscr	Move From Vector Status and Control Register
VX	4	1604			199	V	mtvscr	Move To Vector Status and Control Register
X	31	135			146	V	stvebx	Store Vector Element Byte Indexed
X	31	167			146	V	stvehx	Store Vector Element Halfword Indexed
X	31	199			147	V	stvewx	Store Vector Element Word Indexed
X	31	231			144	V	stvx	Store Vector Indexed
X	31	487			147	V	stvxl	Store Vector Indexed Last
VX	4	384			160	V	vaddcuw	Vector Add and Write Carry-Out Unsigned Word
VX	4	10			189	V	vaddfp	Vector Add Single-Precision

Form	Opcode Pri	Opcode Ext	Mode Dep.[1]	Priv[1]	Page	Cat[1]	Mnemonic	Instruction
VX	4	768			160	V	vaddsbs	Vector Add Signed Byte Saturate
VX	4	832			160	V	vaddshs	Vector Add Signed Halfword Saturate
VX	4	896			160	V	vaddsws	Vector Add Signed Word Saturate
VX	4	0			161	V	vaddubm	Vector Add Unsigned Byte Modulo
VX	4	512			162	V	vaddubs	Vector Add Unsigned Byte Saturate
VX	4	64			161	V	vadduhm	Vector Add Unsigned Halfword Modulo
VX	4	576			162	V	vadduhs	Vector Add Unsigned Halfword Saturate
VX	4	128			161	V	vadduwm	Vector Add Unsigned Word Modulo
VX	4	640			162	V	vadduws	Vector Add Unsigned Word Saturate
VX	4	1028			184	V	vand	Vector Logical AND
VX	4	1092			184	V	vandc	Vector Logical AND with Complement
VX	4	1282			175	V	vavgsb	Vector Average Signed Byte
VX	4	1346			175	V	vavgsh	Vector Average Signed Halfword
VX	4	1410			175	V	vavgsw	Vector Average Signed Word
VX	4	1026			176	V	vavgub	Vector Average Unsigned Byte
VX	4	1090			176	V	vavguh	Vector Average Unsigned Halfword
VX	4	1154			176	V	vavguw	Vector Average Unsigned Word
VX	4	842			193	V	vcfsx	Vector Convert From Signed Fixed-Point Word
VX	4	778			193	V	vcfux	Vector Convert From Unsigned Fixed-Point Word
VC	4	966			195	V	vcmpbfp[.]	Vector Compare Bounds Single-Precision
VC	4	198			195	V	vcmpeqfp[.]	Vector Compare Equal To Single-Precision
VC	4	6			181	V	vcmpequb[.]	Vector Compare Equal To Unsigned Byte
VC	4	70			181	V	vcmpequh[.]	Vector Compare Equal To Unsigned Halfword
VC	4	134			182	V	vcmpequw[.]	Vector Compare Equal To Unsigned Word
VC	4	454			196	V	vcmpgefp[.]	Vector Compare Greater Than or Equal To Single-Precision
VC	4	710			196	V	vcmpgtfp[.]	Vector Compare Greater Than Single-Precision
VC	4	774			182	V	vcmpgtsb[.]	Vector Compare Greater Than Signed Byte
VC	4	838			182	V	vcmpgtsh[.]	Vector Compare Greater Than Signed Halfword
VC	4	902			182	V	vcmpgtsw[.]	Vector Compare Greater Than Signed Word
VC	4	518			183	V	vcmpgtub[.]	Vector Compare Greater Than Unsigned Byte
VC	4	582			183	V	vcmpgtuh[.]	Vector Compare Greater Than Unsigned Halfword
VC	4	646			183	V	vcmpgtuw[.]	Vector Compare Greater Than Unsigned Word
VX	4	970			192	V	vctsxs	Vector Convert To Signed Fixed-Point Word Saturate
VX	4	906			192	V	vctuxs	Vector Convert To Unsigned Fixed-Point Word Saturate
VX	4	394			197	V	vexptefp	Vector 2 Raised to the Exponent Estimate Floating-Point
VX	4	458			197	V	vlogefp	Vector Log Base 2 Estimate Floating-Point
VA	4	46			190	V	vmaddfp	Vector Multiply-Add Single-Precision
VX	4	1034			191	V	vmaxfp	Vector Maximum Single-Precision
VX	4	258			177	V	vmaxsb	Vector Maximum Signed Byte
VX	4	322			177	V	vmaxsh	Vector Maximum Signed Halfword
VX	4	386			177	V	vmaxsw	Vector Maximum Signed Word
VX	4	2			178	V	vmaxub	Vector Maximum Unsigned Byte
VX	4	66			178	V	vmaxuh	Vector Maximum Unsigned Halfword
VX	4	130			178	V	vmaxuw	Vector Maximum Unsigned Word
VA	4	32			168	V	vmhaddshs	Vector Multiply-High-Add Signed Halfword Saturate
VA	4	33			168	V	vmhraddshs	Vector Multiply-High-Round-Add Signed Halfword Saturate
VX	4	1098			191	V	vminfp	Vector Minimum Single-Precision
VX	4	770			179	V	vminsb	Vector Minimum Signed Byte
VX	4	834			179	V	vminsh	Vector Minimum Signed Halfword
VX	4	898			179	V	vminsw	Vector Minimum Signed Word
VX	4	514			180	V	vminub	Vector Minimum Unsigned Byte
VX	4	578			180	V	vminuh	Vector Minimum Unsigned Halfword
VX	4	642			180	V	vminuw	Vector Minimum Unsigned Word
VA	4	34			169	V	vmladduhm	Vector Multiply-Low-Add Unsigned Halfword Modulo
VX	4	12			154	V	vmrghb	Vector Merge High Byte
VX	4	76			154	V	vmrghh	Vector Merge High Halfword
VX	4	140			154	V	vmrghw	Vector Merge High Word
VX	4	268			155	V	vmrglb	Vector Merge Low Byte

Form	Opcode Pri	Opcode Ext	Mode Dep.[1]	Priv[1]	Page	Cat[1]	Mnemonic	Instruction
VX	4	332			155	V	vmrglh	Vector Merge Low Halfword
VX	4	396			155	V	vmrglw	Vector Merge Low Word
VA	4	37			170	V	vmsummbm	Vector Multiply-Sum Mixed Byte Modulo
VA	4	40			170	V	vmsumshm	Vector Multiply-Sum Signed Halfword Modulo
VA	4	41			171	V	vmsumshs	Vector Multiply-Sum Signed Halfword Saturate
VA	4	36			169	V	vmsumubm	Vector Multiply-Sum Unsigned Byte Modulo
VA	4	38			171	V	vmsumuhm	Vector Multiply-Sum Unsigned Halfword Modulo
VA	4	39			172	V	vmsumuhs	Vector Multiply-Sum Unsigned Halfword Saturate
VX	4	776			166	V	vmulesb	Vector Multiply Even Signed Byte
VX	4	840			166	V	vmulesh	Vector Multiply Even Signed Halfword
VX	4	520			166	V	vmuleub	Vector Multiply Even Unsigned Byte
VX	4	584			166	V	vmuleuh	Vector Multiply Even Unsigned Halfword
VX	4	264			167	V	vmulosb	Vector Multiply Odd Signed Byte
VX	4	328			167	V	vmulosh	Vector Multiply Odd Signed Halfword
VX	4	8			167	V	vmuloub	Vector Multiply Odd Unsigned Byte
VX	4	72			167	V	vmulouh	Vector Multiply Odd Unsigned Halfword
VA	4	47			190	V	vnmsubfp	Vector Negative Multiply-Subtract Single-Precision
VX	4	1284			184	V	vnor	Vector Logical NOR
VX	4	1156			184	V	vor	Vector Logical OR
VA	4	43			157	V	vperm	Vector Permute
VX	4	782			149	V	vpkpx	Vector Pack Pixel
VX	4	398			150	V	vpkshss	Vector Pack Signed Halfword Signed Saturate
VX	4	270			150	V	vpkshus	Vector Pack Signed Halfword Unsigned Saturate
VX	4	462			150	V	vpkswss	Vector Pack Signed Word Signed Saturate
VX	4	334			150	V	vpkswus	Vector Pack Signed Word Unsigned Saturate
VX	4	14			151	V	vpkuhum	Vector Pack Unsigned Halfword Unsigned Modulo
VX	4	142			151	V	vpkuhus	Vector Pack Unsigned Halfword Unsigned Saturate
VX	4	78			151	V	vpkuwum	Vector Pack Unsigned Word Unsigned Modulo
VX	4	206			151	V	vpkuwus	Vector Pack Unsigned Word Unsigned Saturate
VX	4	266			198	V	vrefp	Vector Reciprocal Estimate Single-Precision
VX	4	714			194	V	vrfim	Vector Round to Single-Precision Integer toward -Infinity
VX	4	522			194	V	vrfin	Vector Round to Single-Precision Integer Nearest
VX	4	650			194	V	vrfip	Vector Round to Single-Precision Integer toward +Infinity
VX	4	586			194	V	vrfiz	Vector Round to Single-Precision Integer toward Zero
VX	4	4			185	V	vrlb	Vector Rotate Left Byte
VX	4	68			185	V	vrlh	Vector Rotate Left Halfword
VX	4	132			185	V	vrlw	Vector Rotate Left Word
VX	4	330			198	V	vrsqrtefp	Vector Reciprocal Square Root Estimate Single-Precision
VA	4	42			157	V	vsel	Vector Select
VX	4	452			158	V	vsl	Vector Shift Left
VX	4	260			186	V	vslb	Vector Shift Left Byte
VA	4	44			158	V	vsldoi	Vector Shift Left Double by Octet Immediate
VX	4	324			186	V	vslh	Vector Shift Left Halfword
VX	4	1036			158	V	vslo	Vector Shift Left by Octet
VX	4	388			186	V	vslw	Vector Shift Left Word
VX	4	524			156	V	vspltb	Vector Splat Byte
VX	4	588			156	V	vsplth	Vector Splat Halfword
VX	4	780			156	V	vspltisb	Vector Splat Immediate Signed Byte
VX	4	844			156	V	vspltish	Vector Splat Immediate Signed Halfword
VX	4	908			156	V	vspltisw	Vector Splat Immediate Signed Word
VX	4	652			156	V	vspltw	Vector Splat Word
VX	4	708			159	V	vsr	Vector Shift Right
VX	4	772			188	V	vsrab	Vector Shift Right Algebraic Byte
VX	4	836			188	V	vsrah	Vector Shift Right Algebraic Halfword
VX	4	900			188	V	vsraw	Vector Shift Right Algebraic Word
VX	4	516			187	V	vsrb	Vector Shift Right Byte
VX	4	580			187	V	vsrh	Vector Shift Right Halfword
VX	4	1100			159	V	vsro	Vector Shift Right by Octet

Form	Opcode		Mode Dep.[1]	Priv[1]	Page	Cat[1]	Mnemonic	Instruction
	Pri	Ext						
VX	4	644			187	V	vsrw	Vector Shift Right Word
VX	4	1408			163	V	vsubcuw	Vector Subtract and Write Carry-Out Unsigned Word
VX	4	74			189	V	vsubfp	Vector Subtract Single-Precision
VX	4	1792			163	V	vsubsbs	Vector Subtract Signed Byte Saturate
VX	4	1856			163	V	vsubshs	Vector Subtract Signed Halfword Saturate
VX	4	1920			163	V	vsubsws	Vector Subtract Signed Word Saturate
VX	4	1024			164	V	vsububm	Vector Subtract Unsigned Byte Modulo
VX	4	1536			165	V	vsububs	Vector Subtract Unsigned Byte Saturate
VX	4	1088			164	V	vsubuhm	Vector Subtract Unsigned Halfword Modulo
VX	4	1600			164	V	vsubuhs	Vector Subtract Unsigned Halfword Saturate
VX	4	1152			164	V	vsubuwm	Vector Subtract Unsigned Word Modulo
VX	4	1664			165	V	vsubuws	Vector Subtract Unsigned Word Saturate
VX	4	1672			173	V	vsum2sws	Vector Sum across Half Signed Word Saturate
VX	4	1800			174	V	vsum4sbs	Vector Sum across Quarter Signed Byte Saturate
VX	4	1608			174	V	vsum4shs	Vector Sum across Quarter Signed Halfword Saturate
VX	4	1544			174	V	vsum4ubs	Vector Sum across Quarter Unsigned Byte Saturate
VX	4	1928			173	V	vsumsws	Vector Sum across Signed Word Saturate
VX	4	846			152	V	vupkhpx	Vector Unpack High Pixel
VX	4	526			152	V	vupkhsb	Vector Unpack High Signed Byte
VX	4	590			152	V	vupkhsh	Vector Unpack High Signed Halfword
VX	4	974			153	V	vupklpx	Vector Unpack Low Pixel
VX	4	654			153	V	vupklsb	Vector Unpack Low Signed Byte
VX	4	718			153	V	vupklsh	Vector Unpack Low Signed Halfword
VX	4	1220			184	V	vxor	Vector Logical XOR
X	31	62			375	WT	wait	Wait

[1] See the key to the mode dependency and privilege columns on page 839 and the key to the category column in Section 1.3.5 of Book I.

Appendix H. Power ISA Instruction Set Sorted by Opcode

This appendix lists all the instructions in the Power ISA, in order by opcode.

Form	Opcode		Mode Dep.[1]	Priv[1]	Page	Cat[1]	Mnemonic	Instruction
	Pri	Ext						
D	2				70	64	tdi	Trap Doubleword Immediate
D	3				69	B	twi	Trap Word Immediate
VX	4	0			161	V	vaddubm	Vector Add Unsigned Byte Modulo
VX	4	2			178	V	vmaxub	Vector Maximum Unsigned Byte
VX	4	4			185	V	vrlb	Vector Rotate Left Byte
VC	4	6			181	V	vcmpequb[.]	Vector Compare Equal To Unsigned Byte
X	4	8			295	LMA	mulhhwu[.]	Multiply High Halfword to Word Unsigned
VX	4	8			167	V	vmuloub	Vector Multiply Odd Unsigned Byte
VX	4	10			189	V	vaddfp	Vector Add Single-Precision
XO	4	12			292	LMA	machhwu[o][.]	Multiply Accumulate High Halfword to Word Modulo Unsigned
VX	4	12			154	V	vmrghb	Vector Merge High Byte
VX	4	14			151	V	vpkuhum	Vector Pack Unsigned Halfword Unsigned Modulo
VA	4	32			168	V	vmhaddshs	Vector Multiply-High-Add Signed Halfword Saturate
VA	4	33			168	V	vmhraddshs	Vector Multiply-High-Round-Add Signed Halfword Saturate
VA	4	34			169	V	vmladduhm	Vector Multiply-Low-Add Unsigned Halfword Modulo
VA	4	36			169	V	vmsumubm	Vector Multiply-Sum Unsigned Byte Modulo
VA	4	37			170	V	vmsummbm	Vector Multiply-Sum Mixed Byte Modulo
VA	4	38			171	V	vmsumuhm	Vector Multiply-Sum Unsigned Halfword Modulo
VA	4	39			172	V	vmsumuhs	Vector Multiply-Sum Unsigned Halfword Saturate
X	4	40			295	LMA	mulhhw[.]	Multiply High Halfword to Word Signed
VA	4	40			170	V	vmsumshm	Vector Multiply-Sum Signed Halfword Modulo
VA	4	41			171	V	vmsumshs	Vector Multiply-Sum Signed Halfword Saturate
VA	4	42			157	V	vsel	Vector Select
VA	4	43			157	V	vperm	Vector Permute
XO	4	44			291	LMA	machhw[o][.]	Multiply Accumulate High Halfword to Word Modulo Signed
VA	4	44			158	V	vsldoi	Vector Shift Left Double by Octet Immediate
XO	4	46			297	LMA	nmachhw[o][.]	Negative Multiply Accumulate High Halfword to Word Modulo Signed
VA	4	46			190	V	vmaddfp	Vector Multiply-Add Single-Precision
VA	4	47			190	V	vnmsubfp	Vector Negative Multiply-Subtract Single-Precision
VX	4	64			161	V	vadduhm	Vector Add Unsigned Halfword Modulo
VX	4	66			178	V	vmaxuh	Vector Maximum Unsigned Halfword
VX	4	68			185	V	vrlh	Vector Rotate Left Halfword
VC	4	70			181	V	vcmpequh[.]	Vector Compare Equal To Unsigned Halfword
VX	4	72			167	V	vmulouh	Vector Multiply Odd Unsigned Halfword
VX	4	74			189	V	vsubfp	Vector Subtract Single-Precision
XO	4	76			292	LMA	machhwsu[o][.]	Multiply Accumulate High Halfword to Word Saturate Unsigned
VX	4	76			154	V	vmrghh	Vector Merge High Halfword
VX	4	78			151	V	vpkuwum	Vector Pack Unsigned Word Unsigned Modulo
EVS	4	79			247	SP	evsel	Vector Select

Form	Opcode Pri	Opcode Ext	Mode Dep.[1]	Priv[1]	Page	Cat[1]	Mnemonic	Instruction
XO	4	108			291	LMA	machhws[o][.]	Multiply Accumulate High Halfword to Word Saturate Signed
XO	4	110			297	LMA	nmachhws[o][.]	Negative Multiply Accumulate High Halfword to Word Saturate Signed
VX	4	128			161	V	vadduwm	Vector Add Unsigned Word Modulo
VX	4	130			178	V	vmaxuw	Vector Maximum Unsigned Word
VX	4	132			185	V	vrlw	Vector Rotate Left Word
VC	4	134			182	V	vcmpequw[.]	Vector Compare Equal To Unsigned Word
X	4	136			294	LMA	mulchwu[.]	Multiply Cross Halfword to Word Unsigned
XO	4	140			290	LMA	macchwu[o][.]	Multiply Accumulate Cross Halfword to Word Modulo Unsigned
VX	4	140			154	V	vmrghw	Vector Merge High Word
VX	4	142			151	V	vpkuhus	Vector Pack Unsigned Halfword Unsigned Saturate
X	4	168			294	LMA	mulchw[.]	Multiply Cross Halfword to Word Signed
XO	4	172			289	LMA	macchw[o][.]	Multiply Accumulate Cross Halfword to Word Modulo Signed
XO	4	174			296	LMA	nmacchw[o][.]	Negative Multiply Accumulate Cross Halfword to Word Modulo Signed
VC	4	198			195	V	vcmpeqfp[.]	Vector Compare Equal To Single-Precision
XO	4	204			290	LMA	macchwsu[o][.]	Multiply Accumulate Cross Halfword to Word Saturate Unsigned
VX	4	206			151	V	vpkuwus	Vector Pack Unsigned Word Unsigned Saturate
XO	4	236			289	LMA	macchws[o][.]	Multiply Accumulate Cross Halfword to Word Saturate Signed
XO	4	238			296	LMA	nmacchws[o][.]	Negative Multiply Accumulate Cross Halfword to Word Saturate Signed
VX	4	258			177	V	vmaxsb	Vector Maximum Signed Byte
VX	4	260			186	V	vslb	Vector Shift Left Byte
VX	4	264			167	V	vmulosb	Vector Multiply Odd Signed Byte
VX	4	266			198	V	vrefp	Vector Reciprocal Estimate Single-Precision
VX	4	268			155	V	vmrglb	Vector Merge Low Byte
VX	4	270			150	V	vpkshus	Vector Pack Signed Halfword Unsigned Saturate
VX	4	322			177	V	vmaxsh	Vector Maximum Signed Halfword
VX	4	324			186	V	vslh	Vector Shift Left Halfword
VX	4	328			167	V	vmulosh	Vector Multiply Odd Signed Halfword
VX	4	330			198	V	vrsqrtefp	Vector Reciprocal Square Root Estimate Single-Precision
VX	4	332			155	V	vmrglh	Vector Merge Low Halfword
VX	4	334			150	V	vpkswus	Vector Pack Signed Word Unsigned Saturate
VX	4	384			160	V	vaddcuw	Vector Add and Write Carry-Out Unsigned Word
VX	4	386			177	V	vmaxsw	Vector Maximum Signed Word
VX	4	388			186	V	vslw	Vector Shift Left Word
X	4	392			295	LMA	mullhwu[.]	Multiply Low Halfword to Word Unsigned
VX	4	394			197	V	vexptefp	Vector 2 Raised to the Exponent Estimate Floating-Point
XO	4	396			294	LMA	maclhwu[o][.]	Multiply Accumulate Low Halfword to Word Modulo Unsigned
VX	4	396			155	V	vmrglw	Vector Merge Low Word
VX	4	398			150	V	vpkshss	Vector Pack Signed Halfword Signed Saturate
X	4	424			295	LMA	mullhw[.]	Multiply Low Halfword to Word Signed
XO	4	428			293	LMA	maclhw[o][.]	Multiply Accumulate Low Halfword to Word Modulo Signed
XO	4	430			298	LMA	nmaclhw[o][.]	Negative Multiply Accumulate Low Halfword to Word Modulo Signed
VX	4	452			158	V	vsl	Vector Shift Left
VC	4	454			196	V	vcmpgefp[.]	Vector Compare Greater Than or Equal To Single-Precision
VX	4	458			197	V	vlogefp	Vector Log Base 2 Estimate Floating-Point
XO	4	460			294	LMA	maclhwsu[o][.]	Multiply Accumulate Low Halfword to Word Saturate Unsigned

Form	Opcode Pri	Opcode Ext	Mode Dep.[1]	Priv[1]	Page	Cat[1]	Mnemonic	Instruction
VX	4	462			150	V	vpkswss	Vector Pack Signed Word Signed Saturate
XO	4	492			293	LMA	maclhws[o][.]	Multiply Accumulate Low Halfword to Word Saturate Signed
XO	4	494			298	LMA	nmaclhws[o][.]	Negative Multiply Accumulate Low Halfword to Word Saturate Signed
EVX	4	512			209	SP	evaddw	Vector Add Word
VX	4	512			162	V	vaddubs	Vector Add Unsigned Byte Saturate
EVX	4	514			208	SP	evaddiw	Vector Add Immediate Word
VX	4	514			180	V	vminub	Vector Minimum Unsigned Byte
EVX	4	516			253	SP	evsubfw	Vector Subtract from Word
VX	4	516			187	V	vsrb	Vector Shift Right Byte
EVX	4	518			253	SP	evsubifw	Vector Subtract Immediate from Word
VC	4	518			183	V	vcmpgtub[.]	Vector Compare Greater Than Unsigned Byte
EVX	4	520			208	SP	evabs	Vector Absolute Value
VX	4	520			166	V	vmuleub	Vector Multiply Even Unsigned Byte
EVX	4	521			245	SP	evneg	Vector Negate
EVX	4	522			213	SP	evextsb	Vector Extend Sign Byte
VX	4	522			194	V	vrfin	Vector Round to Single-Precision Integer Nearest
EVX	4	523			213	SP	evextsh	Vector Extend Sign Halfword
EVX	4	524			247	SP	evrndw	Vector Round Word
VX	4	524			156	V	vspltb	Vector Splat Byte
EVX	4	525			212	SP	evcntlzw	Vector Count Leading Zeros Word
EVX	4	526			212	SP	evcntlsw	Vector Count Leading Signed Bits Word
VX	4	526			152	V	vupkhsb	Vector Unpack High Signed Byte
EVX	4	527			208	SP	brinc	Bit Reversed Increment
EVX	4	529			210	SP	evand	Vector AND
EVX	4	530			210	SP	evandc	Vector AND with Complement
EVX	4	534			253	SP	evxor	Vector XOR
EVX	4	535			246	SP	evor	Vector OR
EVX	4	536			245	SP	evnor	Vector NOR
EVX	4	537			213	SP	eveqv	Vector Equivalent
EVX	4	539			246	SP	evorc	Vector OR with Complement
EVX	4	542			245	SP	evnand	Vector NAND
EVX	4	544			249	SP	evsrwu	Vector Shift Right Word Unsigned
EVX	4	545			249	SP	evsrws	Vector Shift Right Word Signed
EVX	4	546			248	SP	evsrwiu	Vector Shift Right Word Immediate Unsigned
EVX	4	547			248	SP	evsrwis	Vector Shift Right Word Immediate Signed
EVX	4	548			248	SP	evslw	Vector Shift Left Word
EVX	4	550			248	SP	evslwi	Vector Shift Left Word Immediate
EVX	4	552			246	SP	evrlw	Vector Rotate Left Word
EVX	4	553			248	SP	evsplati	Vector Splat Immediate
EVX	4	554			247	SP	evrlwi	Vector Rotate Left Word Immediate
EVX	4	555			248	SP	evsplatfi	Vector Splat Fractional Immediate
EVX	4	556			219	SP	evmergehi	Vector Merge High
EVX	4	557			219	SP	evmergelo	Vector Merge Low
EVX	4	558			220	SP	evmergehilo	Vector Merge High/Low
EVX	4	559			220	SP	evmergelohi	Vector Merge Low/High
EVX	4	560			211	SP	evcmpgtu	Vector Compare Greater Than Unsigned
EVX	4	561			210	SP	evcmpgts	Vector Compare Greater Than Signed
EVX	4	562			211	SP	evcmpltu	Vector Compare Less Than Unsigned
EVX	4	563			211	SP	evcmplts	Vector Compare Less Than Signed
EVX	4	564			210	SP	evcmpeq	Vector Compare Equal
VX	4	576			162	V	vadduhs	Vector Add Unsigned Halfword Saturate
VX	4	578			180	V	vminuh	Vector Minimum Unsigned Halfword
VX	4	580			187	V	vsrh	Vector Shift Right Halfword
VC	4	582			183	V	vcmpgtuh[.]	Vector Compare Greater Than Unsigned Halfword
VX	4	584			166	V	vmuleuh	Vector Multiply Even Unsigned Halfword
VX	4	586			194	V	vrfiz	Vector Round to Single-Precision Integer toward Zero
VX	4	588			156	V	vsplth	Vector Splat Halfword
VX	4	590			152	V	vupkhsh	Vector Unpack High Signed Halfword
EVX	4	640			260	SP.FV	evfsadd	Vector Floating-Point Single-Precision Add

Form	Opcode Pri	Opcode Ext	Mode Dep.[1]	Priv[1]	Page	Cat[1]	Mnemonic	Instruction
VX	4	640			162	V	vadduws	Vector Add Unsigned Word Saturate
EVX	4	641			260	SP.FV	evfssub	Vector Floating-Point Single-Precision Subtract
VX	4	642			180	V	vminuw	Vector Minimum Unsigned Word
EVX	4	644			259	SP.FV	evfsabs	Vector Floating-Point Single-Precision Absolute Value
VX	4	644			187	V	vsrw	Vector Shift Right Word
EVX	4	645			259	SP.FV	evfsnabs	Vector Floating-Point Single-Precision Negative Absolute Value
EVX	4	646			259	SP.FV	evfsneg	Vector Floating-Point Single-Precision Negate
VC	4	646			183	V	vcmpgtuw[.]	Vector Compare Greater Than Unsigned Word
EVX	4	648			260	SP.FV	evfsmul	Vector Floating-Point Single-Precision Multiply
EVX	4	649			260	SP.FV	evfsdiv	Vector Floating-Point Single-Precision Divide
VX	4	650			194	V	vrfip	Vector Round to Single-Precision Integer toward +Infinity
EVX	4	652			261	SP.FV	evfscmpgt	Vector Floating-Point Single-Precision Compare Greater Than
VX	4	652			156	V	vspltw	Vector Splat Word
EVX	4	653			261	SP.FV	evfscmplt	Vector Floating-Point Single-Precision Compare Less Than
EVX	4	654			262	SP.FV	evfscmpeq	Vector Floating-Point Single-Precision Compare Equal
VX	4	654			153	V	vupklsb	Vector Unpack Low Signed Byte
EVX	4	656			264	SP.FV	evfscfui	Vector Convert Floating-Point Single-Precision from Unsigned Integer
EVX	4	657			264	SP.FV	evfscfsi	Vector Convert Floating-Point Single-Precision from Signed Integer
EVX	4	658			264	SP.FV	evfscfuf	Vector Convert Floating-Point Single-Precision from Unsigned Fraction
EVX	4	659			264	SP.FV	evfscfsf	Vector Convert Floating-Point Single-Precision from Signed Fraction
EVX	4	660			265	SP.FV	evfsctui	Vector Convert Floating-Point Single-Precision to Unsigned Integer
EVX	4	661			265	SP.FV	evfsctsi	Vector Convert Floating-Point Single-Precision to Signed Integer
EVX	4	662			266	SP.FV	evfsctuf	Vector Convert Floating-Point Single-Precision to Unsigned Fraction
EVX	4	663			266	SP.FV	evfsctsf	Vector Convert Floating-Point Single-Precision to Signed Fraction
EVX	4	664			265	SP.FV	evfsctuiz	Vector Convert Floating-Point Single-Precision to Unsigned Integer with Round toward Zero
EVX	4	666			265	SP.FV	evfsctsiz	Vector Convert Floating-Point Single-Precision to Signed Integer with Round toward Zero
EVX	4	668			262	SP.FV	evfststgt	Vector Floating-Point Single-Precision Test Greater Than
EVX	4	669			263	SP.FV	evfststlt	Vector Floating-Point Single-Precision Test Less Than
EVX	4	670			263	SP.FV	evfststeq	Vector Floating-Point Single-Precision Test Equal
EVX	4	704			268	SP.FS	efsadd	Floating-Point Single-Precision Add
EVX	4	705			268	SP.FS	efssub	Floating-Point Single-Precision Subtract
EVX	4	708			267	SP.FS	efsabs	Floating-Point Single-Precision Absolute Value
VX	4	708			159	V	vsr	Vector Shift Right
EVX	4	709			267	SP.FS	efsnabs	Floating-Point Single-Precision Negative Absolute Value
EVX	4	710			267	SP.FS	efsneg	Floating-Point Single-Precision Negate
VC	4	710			196	V	vcmpgtfp[.]	Vector Compare Greater Than Single-Precision
EVX	4	712			268	SP.FS	efsmul	Floating-Point Single-Precision Multiply
EVX	4	713			268	SP.FS	efsdiv	Floating-Point Single-Precision Divide
VX	4	714			194	V	vrfim	Vector Round to Single-Precision Integer toward -Infinity
EVX	4	716			269	SP.FS	efscmpgt	Floating-Point Single-Precision Compare Greater Than
EVX	4	717			269	SP.FS	efscmplt	Floating-Point Single-Precision Compare Less Than
EVX	4	718			270	SP.FS	efscmpeq	Floating-Point Single-Precision Compare Equal
VX	4	718			153	V	vupklsh	Vector Unpack Low Signed Halfword

Form	Opcode Pri	Opcode Ext	Mode Dep.[1]	Priv[1]	Page	Cat[1]	Mnemonic	Instruction
EVX	4	719			281	SP.FD	efscfd	Floating-Point Single-Precision Convert from Double-Precision
EVX	4	720			272	SP.FS	efscfui	Convert Floating-Point Single-Precision from Unsigned Integer
EVX	4	721			272	SP.FS	efscfsi	Convert Floating-Point Single-Precision from Signed Integer
EVX	4	722			272	SP.FS	efscfuf	Convert Floating-Point Single-Precision from Unsigned Fraction
EVX	4	723			272	SP.FS	efscfsf	Convert Floating-Point Single-Precision from Signed Fraction
EVX	4	724			272	SP.FS	efsctui	Convert Floating-Point Single-Precision to Unsigned Integer
EVX	4	725			272	SP.FS	efsctsi	Convert Floating-Point Single-Precision to Signed Integer
EVX	4	726			273	SP.FS	efsctuf	Convert Floating-Point Single-Precision to Unsigned Fraction
EVX	4	727			273	SP.FS	efsctsf	Convert Floating-Point Single-Precision to Signed Fraction
EVX	4	728			273	SP.FS	efsctuiz	Convert Floating-Point Single-Precision to Unsigned Integer with Round toward Zero
EVX	4	730			273	SP.FS	efsctsiz	Convert Floating-Point Single-Precision to Signed Integer with Round toward Zero
EVX	4	732			270	SP.FS	efststgt	Floating-Point Single-Precision Test Greater Than
EVX	4	733			271	SP.FS	efststlt	Floating-Point Single-Precision Test Less Than
EVX	4	734			271	SP.FS	efststeq	Floating-Point Single-Precision Test Equal
EVX	4	736			275	SP.FD	efdadd	Floating-Point Double-Precision Add
EVX	4	737			275	SP.FD	efdsub	Floating-Point Double-Precision Subtract
EVX	4	738			278	SP.FD	efdcfuid	Convert Floating-Point Double-Precision from Unsigned Integer Doubleword
EVX	4	739			278	SP.FD	efdcfsid	Convert Floating-Point Double-Precision from Signed Integer Doubleword
EVX	4	740			274	SP.FD	efdabs	Floating-Point Double-Precision Absolute Value
EVX	4	741			274	SP.FD	efdnabs	Floating-Point Double-Precision Negative Absolute Value
EVX	4	742			274	SP.FD	efdneg	Floating-Point Double-Precision Negate
EVX	4	744			275	SP.FD	efdmul	Floating-Point Double-Precision Multiply
EVX	4	745			275	SP.FD	efddiv	Floating-Point Double-Precision Divide
EVX	4	746			279	SP.FD	efdctuidz	Convert Floating-Point Double-Precision to Unsigned Integer Doubleword with Round toward Zero
EVX	4	747			279	SP.FD	efdctsidz	Convert Floating-Point Double-Precision to Signed Integer Doubleword with Round toward Zero
EVX	4	748			276	SP.FD	efdcmpgt	Floating-Point Double-Precision Compare Greater Than
EVX	4	749			276	SP.FD	efdcmplt	Floating-Point Double-Precision Compare Less Than
EVX	4	750			276	SP.FD	efdcmpeq	Floating-Point Double-Precision Compare Equal
EVX	4	751			280	SP.FD	efdcfs	Floating-Point Double-Precision Convert from Single-Precision
EVX	4	752			277	SP.FD	efdcfui	Convert Floating-Point Double-Precision from Unsigned Integer
EVX	4	753			277	SP.FD	efdcfsi	Convert Floating-Point Double-Precision from Signed Integer
EVX	4	754			278	SP.FD	efdcfuf	Convert Floating-Point Double-Precision from Unsigned Fraction
EVX	4	755			278	SP.FD	efdcfsf	Convert Floating-Point Double-Precision from Signed Fraction
EVX	4	756			278	SP.FD	efdctui	Convert Floating-Point Double-Precision to Unsigned Integer
EVX	4	757			278	SP.FD	efdctsi	Convert Floating-Point Double-Precision to Signed Integer

Form	Opcode		Mode Dep.[1]	Priv[1]	Page	Cat[1]	Mnemonic	Instruction
	Pri	Ext						
EVX	4	758			280	SP.FD	efdctuf	Convert Floating-Point Double-Precision to Unsigned Fraction
EVX	4	759			280	SP.FD	efdctsf	Convert Floating-Point Double-Precision to Signed Fraction
EVX	4	760			280	SP.FD	efdctuiz	Convert Floating-Point Double-Precision to Unsigned Integer with Round toward Zero
EVX	4	762			280	SP.FD	efdctsiz	Convert Floating-Point Double-Precision to Signed Integer with Round toward Zero
EVX	4	764			276	SP.FD	efdtstgt	Floating-Point Double-Precision Test Greater Than
EVX	4	765			277	SP.FD	efdtstlt	Floating-Point Double-Precision Test Less Than
EVX	4	766			277	SP.FD	efdtsteq	Floating-Point Double-Precision Test Equal
EVX	4	768			214	SP	evlddx	Vector Load Double Word into Double Word Indexed
VX	4	768			160	V	vaddsbs	Vector Add Signed Byte Saturate
EVX	4	769			214	SP	evldd	Vector Load Double Word into Double Word
EVX	4	770			215	SP	evldwx	Vector Load Double into Two Words Indexed
VX	4	770			179	V	vminsb	Vector Minimum Signed Byte
EVX	4	771			215	SP	evldw	Vector Load Double into Two Words
EVX	4	772			214	SP	evldhx	Vector Load Double into Four Halfwords Indexed
VX	4	772			188	V	vsrab	Vector Shift Right Algebraic Byte
EVX	4	773			214	SP	evldh	Vector Load Double into Four Halfwords
VC	4	774			182	V	vcmpgtsb[.]	Vector Compare Greater Than Signed Byte
EVX	4	776			215	SP	evlhhesplatx	Vector Load Halfword into Halfwords Even and Splat Indexed
VX	4	776			166	V	vmulesb	Vector Multiply Even Signed Byte
EVX	4	777			215	SP	evlhhesplat	Vector Load Halfword into Halfwords Even and Splat
VX	4	778			193	V	vcfux	Vector Convert From Unsigned Fixed-Point Word
EVX	4	780			216	SP	evlhhousplatx	Vector Load Halfword into Halfword Odd Unsigned and Splat Indexed
VX	4	780			156	V	vspltisb	Vector Splat Immediate Signed Byte
EVX	4	781			216	SP	evlhhousplat	Vector Load Halfword into Halfword Odd Unsigned and Splat
EVX	4	782			216	SP	evlhhossplatx	Vector Load Halfword into Halfword Odd Signed and Splat Indexed
VX	4	782			149	V	vpkpx	Vector Pack Pixel
EVX	4	783			216	SP	evlhhossplat	Vector Load Halfword into Halfword Odd Signed and Splat
EVX	4	784			217	SP	evlwhex	Vector Load Word into Two Halfwords Even Indexed
EVX	4	785			217	SP	evlwhe	Vector Load Word into Two Halfwords Even
EVX	4	788			218	SP	evlwhoux	Vector Load Word into Two Halfwords Odd Unsigned Indexed (zero-extended)
EVX	4	789			218	SP	evlwhou	Vector Load Word into Two Halfwords Odd Unsigned (zero-extended)
EVX	4	790			217	SP	evlwhosx	Vector Load Word into Two Halfwords Odd Signed Indexed (with sign extension)
EVX	4	791			217	SP	evlwhos	Vector Load Word into Two Halfwords Odd Signed (with sign extension)
EVX	4	792			219	SP	evlwwsplatx	Vector Load Word into Word and Splat Indexed
EVX	4	793			219	SP	evlwwsplat	Vector Load Word into Word and Splat
EVX	4	796			218	SP	evlwhsplatx	Vector Load Word into Two Halfwords and Splat Indexed
EVX	4	797			218	SP	evlwhsplat	Vector Load Word into Two Halfwords and Splat
EVX	4	800			249	SP	evstddx	Vector Store Double of Double Indexed
EVX	4	801			249	SP	evstdd	Vector Store Double of Double
EVX	4	802			250	SP	evstdwx	Vector Store Double of Two Words Indexed
EVX	4	803			250	SP	evstdw	Vector Store Double of Two Words
EVX	4	804			250	SP	evstdhx	Vector Store Double of Four Halfwords Indexed
EVX	4	805			250	SP	evstdh	Vector Store Double of Four Halfwords
EVX	4	816			251	SP	evstwhex	Vector Store Word of Two Halfwords from Even Indexed
EVX	4	817			251	SP	evstwhe	Vector Store Word of Two Halfwords from Even
EVX	4	820			251	SP	evstwhox	Vector Store Word of Two Halfwords from Odd Indexed

Form	Opcode Pri	Opcode Ext	Mode Dep.[1]	Priv[1]	Page	Cat[1]	Mnemonic	Instruction
EVX	4	821			251	SP	evstwho	Vector Store Word of Two Halfwords from Odd
EVX	4	824			251	SP	evstwwex	Vector Store Word of Word from Even Indexed
EVX	4	825			251	SP	evstwwe	Vector Store Word of Word from Even
EVX	4	828			252	SP	evstwwox	Vector Store Word of Word from Odd Indexed
EVX	4	829			252	SP	evstwwo	Vector Store Word of Word from Odd
VX	4	832			160	V	vaddshs	Vector Add Signed Halfword Saturate
VX	4	834			179	V	vminsh	Vector Minimum Signed Halfword
VX	4	836			188	V	vsrah	Vector Shift Right Algebraic Halfword
VC	4	838			182	V	vcmpgtsh[.]	Vector Compare Greater Than Signed Halfword
VX	4	840			166	V	vmulesh	Vector Multiply Even Signed Halfword
VX	4	842			193	V	vcfsx	Vector Convert From Signed Fixed-Point Word
VX	4	844			156	V	vspltish	Vector Splat Immediate Signed Halfword
VX	4	846			152	V	vupkhpx	Vector Unpack High Pixel
VX	4	896			160	V	vaddsws	Vector Add Signed Word Saturate
VX	4	898			179	V	vminsw	Vector Minimum Signed Word
VX	4	900			188	V	vsraw	Vector Shift Right Algebraic Word
VC	4	902			182	V	vcmpgtsw[.]	Vector Compare Greater Than Signed Word
VX	4	906			192	V	vctuxs	Vector Convert To Unsigned Fixed-Point Word Saturate
VX	4	908			156	V	vspltisw	Vector Splat Immediate Signed Word
VC	4	966			195	V	vcmpbfp[.]	Vector Compare Bounds Single-Precision
VX	4	970			192	V	vctsxs	Vector Convert To Signed Fixed-Point Word Saturate
VX	4	974			153	V	vupklpx	Vector Unpack Low Pixel
VX	4	1024			164	V	vsububm	Vector Subtract Unsigned Byte Modulo
VX	4	1026			176	V	vavgub	Vector Average Unsigned Byte
EVX	4	1027			224	SP	evmhessf	Vector Multiply Halfwords, Even, Signed, Saturate, Fractional
VX	4	1028			184	V	vand	Vector Logical AND
EVX	4	1031			233	SP	evmhossf	Vector Multiply Halfwords, Odd, Signed, Saturate, Fractional
EVX	4	1032			227	SP	evmheumi	Vector Multiply Halfwords, Even, Unsigned, Modulo, Integer
EVX	4	1033			223	SP	evmhesmi	Vector Multiply Halfwords, Even, Signed, Modulo, Integer
VX	4	1034			191	V	vmaxfp	Vector Maximum Single-Precision
EVX	4	1035			222	SP	evmhesmf	Vector Multiply Halfwords, Even, Signed, Modulo, Fractional
EVX	4	1036			235	SP	evmhoumi	Vector Multiply Halfwords, Odd, Unsigned, Modulo, Integer
VX	4	1036			158	V	vslo	Vector Shift Left by Octet
EVX	4	1037			231	SP	evmhosmi	Vector Multiply Halfwords, Odd, Signed, Modulo, Integer
EVX	4	1039			230	SP	evmhosmf	Vector Multiply Halfwords, Odd, Signed, Modulo, Fractional
EVX	4	1059			224	SP	evmhessfa	Vector Multiply Halfwords, Even, Signed, Saturate, Fractional to Accumulator
EVX	4	1063			233	SP	evmhossfa	Vector Multiply Halfwords, Odd, Signed, Saturate, Fractional to Accumulator
EVX	4	1064			227	SP	evmheumia	Vector Multiply Halfwords, Even, Unsigned, Modulo, Integer to Accumulator
EVX	4	1065			223	SP	evmhesmia	Vector Multiply Halfwords, Even, Signed, Modulo, Integer to Accumulator
EVX	4	1067			222	SP	evmhesmfa	Vector Multiply Halfwords, Even, Signed, Modulo, Fractional to Accumulator
EVX	4	1068			235	SP	evmhoumia	Vector Multiply Halfwords, Odd, Unsigned, Modulo, Integer to Accumulator
EVX	4	1069			231	SP	evmhosmia	Vector Multiply Halfwords, Odd, Signed, Modulo, Integer to Accumulator
EVX	4	1071			230	SP	evmhosmfa	Vector Multiply Halfwords, Odd, Signed, Modulo, Fractional to Accumulator
VX	4	1088			164	V	vsubuhm	Vector Subtract Unsigned Halfword Modulo

Form	Opcode Pri	Opcode Ext	Mode Dep.[1]	Priv[1]	Page	Cat[1]	Mnemonic	Instruction
VX	4	1090			176	V	vavguh	Vector Average Unsigned Halfword
VX	4	1092			184	V	vandc	Vector Logical AND with Complement
EVX	4	1095			238	SP	evmwhssf	Vector Multiply Word High Signed, Saturate, Fractional
EVX	4	1096			240	SP	evmwlumi	Vector Multiply Word Low Unsigned, Modulo, Integer
VX	4	1098			191	V	vminfp	Vector Minimum Single-Precision
EVX	4	1100			238	SP	evmwhumi	Vector Multiply Word High Unsigned, Modulo, Integer
VX	4	1100			159	V	vsro	Vector Shift Right by Octet
EVX	4	1101			237	SP	evmwhsmi	Vector Multiply Word High Signed, Modulo, Integer
EVX	4	1103			237	SP	evmwhsmf	Vector Multiply Word High Signed, Modulo, Fractional
EVX	4	1107			243	SP	evmwssf	Vector Multiply Word Signed, Saturate, Fractional
EVX	4	1112			244	SP	evmwumi	Vector Multiply Word Unsigned, Modulo, Integer
EVX	4	1113			242	SP	evmwsmi	Vector Multiply Word Signed, Modulo, Integer
EVX	4	1115			241	SP	evmwsmf	Vector Multiply Word Signed, Modulo, Fractional
EVX	4	1127			238	SP	evmwhssfa	Vector Multiply Word High Signed, Saturate, Fractional to Accumulator
EVX	4	1128			240	SP	evmwlumia	Vector Multiply Word Low Unsigned, Modulo, Integer to Accumulator
EVX	4	1132			238	SP	evmwhumia	Vector Multiply Word High Unsigned, Modulo, Integer to Accumulator
EVX	4	1133			237	SP	evmwhsmia	Vector Multiply Word High Signed, Modulo, Integer to Accumulator
EVX	4	1135			237	SP	evmwhsmfa	Vector Multiply Word High Signed, Modulo, Fractional to Accumulator
EVX	4	1139			243	SP	evmwssfa	Vector Multiply Word Signed, Saturate, Fractional to Accumulator
EVX	4	1144			244	SP	evmwumia	Vector Multiply Word Unsigned, Modulo, Integer to Accumulator
EVX	4	1145			242	SP	evmwsmia	Vector Multiply Word Signed, Modulo, Integer to Accumulator
EVX	4	1147			241	SP	evmwsmfa	Vector Multiply Word Signed, Modulo, Fractional to Accumulator
VX	4	1152			164	V	vsubuwm	Vector Subtract Unsigned Word Modulo
VX	4	1154			176	V	vavguw	Vector Average Unsigned Word
VX	4	1156			184	V	vor	Vector Logical OR
EVX	4	1216			209	SP	evaddusiaaw	Vector Add Unsigned, Saturate, Integer to Accumulator Word
EVX	4	1217			209	SP	evaddssiaaw	Vector Add Signed, Saturate, Integer to Accumulator Word
EVX	4	1218			253	SP	evsubfusiaaw	Vector Subtract Unsigned, Saturate, Integer to Accumulator Word
EVX	4	1219			252	SP	evsubfssiaaw	Vector Subtract Signed, Saturate, Integer to Accumulator Word
EVX	4	1220			237	SP	evmra	Initialize Accumulator
VX	4	1220			184	V	vxor	Vector Logical XOR
EVX	4	1222			212	SP	evdivws	Vector Divide Word Signed
EVX	4	1223			213	SP	evdivwu	Vector Divide Word Unsigned
EVX	4	1224			209	SP	evaddumiaaw	Vector Add Unsigned, Modulo, Integer to Accumulator Word
EVX	4	1225			208	SP	evaddsmiaaw	Vector Add Signed, Modulo, Integer to Accumulator Word
EVX	4	1226			253	SP	evsubfumiaaw	Vector Subtract Unsigned, Modulo, Integer to Accumulator Word
EVX	4	1227			252	SP	evsubfsmiaaw	Vector Subtract Signed, Modulo, Integer to Accumulator Word
EVX	4	1280			228	SP	evmheusiaaw	Vector Multiply Halfwords, Even, Unsigned, Saturate, Integer and Accumulate into Words
EVX	4	1281			226	SP	evmhessiaaw	Vector Multiply Halfwords, Even, Signed, Saturate, Integer and Accumulate into Words
VX	4	1282			175	V	vavgsb	Vector Average Signed Byte

Form	Opcode		Mode Dep.[1]	Priv[1]	Page	Cat[1]	Mnemonic	Instruction
	Pri	Ext						
EVX	4	1283			225	SP	evmhessfaaw	Vector Multiply Halfwords, Even, Signed, Saturate, Fractional and Accumulate into Words
EVX	4	1284			236	SP	evmhousiaaw	Vector Multiply Halfwords, Odd, Unsigned, Saturate, Integer and Accumulate into Words
VX	4	1284			184	V	vnor	Vector Logical NOR
EVX	4	1285			235	SP	evmhossiaaw	Vector Multiply Halfwords, Odd, Signed, Saturate, Integer and Accumulate into Words
EVX	4	1287			234	SP	evmhossfaaw	Vector Multiply Halfwords, Odd, Signed, Saturate, Fractional and Accumulate into Words
EVX	4	1288			227	SP	evmheumiaaw	Vector Multiply Halfwords, Even, Unsigned, Modulo, Integer and Accumulate into Words
EVX	4	1289			223	SP	evmhesmiaaw	Vector Multiply Halfwords, Even, Signed, Modulo, Integer and Accumulate into Words
EVX	4	1291			222	SP	evmhesmfaaw	Vector Multiply Halfwords, Even, Signed, Modulo, Fractional and Accumulate into Words
EVX	4	1292			236	SP	evmhoumiaaw	Vector Multiply Halfwords, Odd, Unsigned, Modulo, Integer and Accumulate into Words
EVX	4	1293			232	SP	evmhosmiaaw	Vector Multiply Halfwords, Odd, Signed, Modulo, Integer and Accumulate into Words
EVX	4	1295			231	SP	evmhosmfaaw	Vector Multiply Halfwords, Odd, Signed, Modulo, Fractional and Accumulate into Words
EVX	4	1320			221	SP	evmhegumiaa	Vector Multiply Halfwords, Even, Guarded, Unsigned, Modulo, Integer and Accumulate
EVX	4	1321			221	SP	evmhegsmiaa	Vector Multiply Halfwords, Even, Guarded, Signed, Modulo, Integer and Accumulate
EVX	4	1323			220	SP	evmhegsmfaa	Vector Multiply Halfwords, Even, Guarded, Signed, Modulo, Fractional and Accumulate
EVX	4	1324			230	SP	evmhogumiaa	Vector Multiply Halfwords, Odd, Guarded, Unsigned, Modulo, Integer and Accumulate
EVX	4	1325			229	SP	evmhogsmiaa	Vector Multiply Halfwords, Odd, Guarded, Signed, Modulo, Integer and Accumulate
EVX	4	1327			229	SP	evmhogsmfaa	Vector Multiply Halfwords, Odd, Guarded, Signed, Modulo, Fractional and Accumulate
EVX	4	1344			241	SP	evmwlusiaaw	Vector Multiply Word Low Unsigned, Saturate, Integer and Accumulate into Words
EVX	4	1345			239	SP	evmwlssiaaw	Vector Multiply Word Low Signed, Saturate, Integer and Accumulate into Words
VX	4	1346			175	V	vavgsh	Vector Average Signed Halfword
EVX	4	1352			240	SP	evmwlumiaaw	Vector Multiply Word Low Unsigned, Modulo, Integer and Accumulate into Words
EVX	4	1353			239	SP	evmwlsmiaaw	Vector Multiply Word Low Signed, Modulo, Integer and Accumulate into Words
EVX	4	1363			243	SP	evmwssfaa	Vector Multiply Word Signed, Saturate, Fractional and Accumulate
EVX	4	1368			245	SP	evmwumiaa	Vector Multiply Word Unsigned, Modulo, Integer and Accumulate
EVX	4	1369			242	SP	evmwsmiaa	Vector Multiply Word Signed, Modulo, Integer and Accumulate
EVX	4	1371			242	SP	evmwsmfaa	Vector Multiply Word Signed, Modulo, Fractional and Accumulate
EVX	4	1408			228	SP	evmheusianw	Vector Multiply Halfwords, Even, Unsigned, Saturate, Integer and Accumulate Negative into Words
VX	4	1408			163	V	vsubcuw	Vector Subtract and Write Carry-Out Unsigned Word
EVX	4	1409			226	SP	evmhessianw	Vector Multiply Halfwords, Even, Signed, Saturate, Integer and Accumulate Negative into Words
VX	4	1410			175	V	vavgsw	Vector Average Signed Word
EVX	4	1411			225	SP	evmhessfanw	Vector Multiply Halfwords, Even, Signed, Saturate, Fractional and Accumulate Negative into Words

Form	Opcode Pri	Opcode Ext	Mode Dep.[1]	Priv[1]	Page	Cat[1]	Mnemonic	Instruction
EVX	4	1412			236	SP	evmhousianw	Vector Multiply Halfwords, Odd, Unsigned, Saturate, Integer and Accumulate Negative into Words
EVX	4	1413			235	SP	evmhossianw	Vector Multiply Halfwords, Odd, Signed, Saturate, Integer and Accumulate Negative into Words
EVX	4	1415			234	SP	evmhossfanw	Vector Multiply Halfwords, Odd, Signed, Saturate, Fractional and Accumulate Negative into Words
EVX	4	1416			227	SP	evmheumianw	Vector Multiply Halfwords, Even, Unsigned, Modulo, Integer and Accumulate Negative into Words
EVX	4	1417			223	SP	evmhesmianw	Vector Multiply Halfwords, Even, Signed, Modulo, Integer and Accumulate Negative into Words
EVX	4	1419			222	SP	evmhesmfanw	Vector Multiply Halfwords, Even, Signed, Modulo, Fractional and Accumulate Negative into Words
EVX	4	1420			232	SP	evmhoumianw	Vector Multiply Halfwords, Odd, Unsigned, Modulo, Integer and Accumulate Negative into Words
EVX	4	1421			231	SP	evmhosmianw	Vector Multiply Halfwords, Odd, Signed, Modulo, Integer and Accumulate Negative into Words
EVX	4	1423			231	SP	evmhosmfanw	Vector Multiply Halfwords, Odd, Signed, Modulo, Fractional and Accumulate Negative into Words
EVX	4	1448			221	SP	evmhegumian	Vector Multiply Halfwords, Even, Guarded, Unsigned, Modulo, Integer and Accumulate Negative
EVX	4	1449			221	SP	evmhegsmian	Vector Multiply Halfwords, Even, Guarded, Signed, Modulo, Integer and Accumulate Negative
EVX	4	1451			220	SP	evmhegsmfan	Vector Multiply Halfwords, Even, Guarded, Signed, Modulo, Fractional and Accumulate Negative
EVX	4	1452			230	SP	evmhogumian	Vector Multiply Halfwords, Odd, Guarded, Unsigned, Modulo, Integer and Accumulate Negative
EVX	4	1453			229	SP	evmhogsmian	Vector Multiply Halfwords, Odd, Guarded, Signed, Modulo, Integer and Accumulate Negative
EVX	4	1455			229	SP	evmhogsmfan	Vector Multiply Halfwords, Odd, Guarded, Signed, Modulo, Fractional and Accumulate Negative
EVX	4	1472			241	SP	evmwlusianw	Vector Multiply Word Low Unsigned, Saturate, Integer and Accumulate Negative in Words
EVX	4	1473			239	SP	evmwlssianw	Vector Multiply Word Low Signed, Saturate, Integer and Accumulate Negative in Words
EVX	4	1480			240	SP	evmwlumianw	Vector Multiply Word Low Unsigned, Modulo, Integer and Accumulate Negative in Words
EVX	4	1481			239	SP	evmwlsmianw	Vector Multiply Word Low Signed, Modulo, Integer and Accumulate Negative in Words
EVX	4	1491			244	SP	evmwssfan	Vector Multiply Word Signed, Saturate, Fractional and Accumulate Negative
EVX	4	1496			245	SP	evmwumian	Vector Multiply Word Unsigned, Modulo, Integer and Accumulate Negative
EVX	4	1497			242	SP	evmwsmian	Vector Multiply Word Signed, Modulo, Integer and Accumulate Negative
EVX	4	1499			242	SP	evmwsmfan	Vector Multiply Word Signed, Modulo, Fractional and Accumulate Negative
VX	4	1536			165	V	vsububs	Vector Subtract Unsigned Byte Saturate
VX	4	1540			199	V	mfvscr	Move From Vector Status and Control Register
VX	4	1544			174	V	vsum4ubs	Vector Sum across Quarter Unsigned Byte Saturate
VX	4	1600			164	V	vsubuhs	Vector Subtract Unsigned Halfword Saturate
VX	4	1604			199	V	mtvscr	Move To Vector Status and Control Register
VX	4	1608			174	V	vsum4shs	Vector Sum across Quarter Signed Halfword Saturate
VX	4	1664			165	V	vsubuws	Vector Subtract Unsigned Word Saturate
VX	4	1672			173	V	vsum2sws	Vector Sum across Half Signed Word Saturate
VX	4	1792			163	V	vsubsbs	Vector Subtract Signed Byte Saturate
VX	4	1800			174	V	vsum4sbs	Vector Sum across Quarter Signed Byte Saturate
VX	4	1856			163	V	vsubshs	Vector Subtract Signed Halfword Saturate
VX	4	1920			163	V	vsubsws	Vector Subtract Signed Word Saturate
VX	4	1928			173	V	vsumsws	Vector Sum across Signed Word Saturate
D	7				63	B	mulli	Multiply Low Immediate

Form	Opcode		Mode Dep.[1]	Priv[1]	Page	Cat[1]	Mnemonic	Instruction
	Pri	Ext						
D	8		SR		60	B	subfic	Subtract From Immediate Carrying
D	10				68	B	cmpli	Compare Logical Immediate
D	11				67	B	cmpi	Compare Immediate
D	12		SR		59	B	addic	Add Immediate Carrying
D	13		SR		59	B	addic.	Add Immediate Carrying and Record
D	14				58	B	addi	Add Immediate
D	15				58	B	addis	Add Immediate Shifted
B	16		CT		31	B	bc[l][a]	Branch Conditional
SC	17				35, 404, 515	B	sc	System Call
I	18				31	B	b[l][a]	Branch
XL	19	0			34	B	mcrf	Move Condition Register Field
XL	19	16	CT		32	B	bclr[l]	Branch Conditional to Link Register
XL	19	18		P	405	S	rfid	Return From Interrupt Doubleword
XL	19	33			34	B	crnor	Condition Register NOR
XL	19	38		P	516	E	rfmci	Return From Machine Check Interrupt
X	19	39			516	E.ED	rfdi	Return From Debug Interrupt
XL	19	50		P	515	E	rfi	Return From Interrupt
XL	19	51		P	516	E	rfci	Return From Critical Interrupt
XL	19	129			34	B	crandc	Condition Register AND with Complement
XL	19	150			369	B	isync	Instruction Synchronize
XL	19	193			33	B	crxor	Condition Register XOR
XFX	19	198			620	E.ED	dnh	Debugger Notify Halt
XL	19	225			33	B	crnand	Condition Register NAND
XL	19	257			33	B	crand	Condition Register AND
XL	19	274		H	405	S	hrfid	Hypervisor Return From Interrupt Doubleword
XL	19	289			34	B	creqv	Condition Register Equivalent
XL	19	417			34	B	crorc	Condition Register OR with Complement
XL	19	449			33	B	cror	Condition Register OR
XL	19	528	CT		32	B	bcctr[l]	Branch Conditional to Count Register
M	20		SR		79	B	rlwimi[.]	Rotate Left Word Immediate then Mask Insert
M	21		SR		77	B	rlwinm[.]	Rotate Left Word Immediate then AND with Mask
M	23		SR		78	B	rlwnm[.]	Rotate Left Word then AND with Mask
D	24				71	B	ori	OR Immediate
D	25				72	B	oris	OR Immediate Shifted
D	26				72	B	xori	XOR Immediate
D	27				72	B	xoris	XOR Immediate Shifted
D	28		SR		71	B	andi.	AND Immediate
D	29		SR		71	B	andis.	AND Immediate Shifted
MD	30	0	SR		79	64	rldicl[.]	Rotate Left Doubleword Immediate then Clear Left
MD	30	1	SR		80	64	rldicr[.]	Rotate Left Doubleword Immediate then Clear Right
MD	30	2	SR		81	64	rldic[.]	Rotate Left Doubleword Immediate then Clear
MD	30	3	SR		82	64	rldimi[.]	Rotate Left Doubleword Immediate then Mask Insert
MDS	30	8	SR		81	64	rldcl[.]	Rotate Left Doubleword then Clear Left
MDS	30	9	SR		82	64	rldcr[.]	Rotate Left Doubleword then Clear Right
X	31	0			67	B	cmp	Compare
X	31	4			69	B	tw	Trap Word
X	31	6			148	V	lvsl	Load Vector for Shift Left Indexed
X	31	7			146	V	lvebx	Load Vector Element Byte Indexed
XO	31	8	SR		60	B	subfc[o][.]	Subtract From Carrying
XO	31	9	SR		65	64	mulhdu[.]	Multiply High Doubleword Unsigned
XO	31	10	SR		60	B	addc[o][.]	Add Carrying
XO	31	11	SR		63	B	mulhwu[.]	Multiply High Word Unsigned
A	31	15			70	B.in	isel	Integer Select
XFX	31	19			89	B	mfcr	Move From Condition Register
XFX	31	19			90	B.in	mfocrf	Move From One Condition Register Field
X	31	20			370	B	lwarx	Load Word And Reserve Indexed
X	31	21			46	64	ldx	Load Doubleword Indexed
X	31	22			359	E	icbt	Instruction Cache Block Touch
X	31	23			44	B	lwzx	Load Word and Zero Indexed

Form	Opcode Pri	Opcode Ext	Mode Dep.[1]	Priv[1]	Page	Cat[1]	Mnemonic	Instruction
X	31	24	SR		83	B	slw[.]	Shift Left Word
X	31	26	SR		74	B	cntlzw[.]	Count Leading Zeros Word
X	31	27	SR		85	64	sld[.]	Shift Left Doubleword
X	31	28	SR		73	B	and[.]	AND
X	31	29			530	E.PD	ldepx	Load Doubleword by External Process ID Indexed
X	31	31			530	E.PD	lwepx	Load Word by External Process ID Indexed
X	31	32			68	B	cmpl	Compare Logical
X	31	38			148	V	lvsr	Load Vector for Shift Right Indexed
X	31	39			143	V	lvehx	Load Vector Element Halfword Indexed
XO	31	40	SR		59	B	subf[o][.]	Subtract From
X	31	53			46	64	ldux	Load Doubleword with Update Indexed
X	31	54			366	B	dcbst	Data Cache Block Store
X	31	55			44	B	lwzux	Load Word and Zero with Update Indexed
X	31	58	SR		76	64	cntlzd[.]	Count Leading Zeros Doubleword
X	31	60	SR		74	B	andc[.]	AND with Complement
X	31	62			375	WT	wait	Wait
X	31	63			533	E.PD	dcbstep	Data Cache Block Store by External PID
X	31	68			70	64	td	Trap Doubleword
X	31	71			143	V	lvewx	Load Vector Element Word Indexed
XO	31	73	SR		65	64	mulhd[.]	Multiply High Doubleword
XO	31	75	SR		63	B	mulhw[.]	Multiply High Word
X	31	78			287	LMV	dlmzb[.]	Determine Leftmost Zero Byte
X	31	83		P	417, 527	B	mfmsr	Move From Machine State Register
X	31	84			371	64	ldarx	Load Doubleword And Reserve Indexed
X	31	86			367	B	dcbf	Data Cache Block Flush
X	31	87			42	B	lbzx	Load Byte and Zero Indexed
X	31	95			529	E.PD	lbepx	Load Byte by External Process ID Indexed
X	31	103			144	V	lvx	Load Vector Indexed
XO	31	104	SR		62	B	neg[o][.]	Negate
X	31	119			41	B	lbzux	Load Byte and Zero with Update Indexed
X	31	122			76	B.in	popcntb	Population Count Bytes
X	31	124	SR		74	B	nor[.]	NOR
X	31	127			534	E.PD	dcbfep	Data Cache Block Flush by External PID
X	31	131		S	528	E	wrtee	Write MSR External Enable
X	31	134			557	ECL	dcbtstls	Data Cache Block Touch for Store and Lock Set
X	31	135			146	V	stvebx	Store Vector Element Byte Indexed
XO	31	136	SR		61	B	subfe[o][.]	Subtract From Extended
XO	31	138	SR		61	B	adde[o][.]	Add Extended
XFX	31	144			89	B	mtcrf	Move To Condition Register Fields
XFX	31	144			90	B.in	mtocrf	Move To One Condition Register Field
X	31	146		P	527	E	mtmsr	Move To Machine State Register
X	31	146		P	415	S	mtmsr	Move To Machine State Register
X	31	149			50	64	stdx	Store Doubleword Indexed
X	31	150			370	B	stwcx.	Store Word Conditional Indexed
X	31	151			49	B	stwx	Store Word Indexed
X	31	157			532	E.PD	stdepx	Store Doubleword by External Process ID Indexed
X	31	159			532	E.PD	stwepx	Store Word by External Process ID Indexed
X	31	163		S	528	E	wrteei	Write MSR External Enable Immediate
X	31	166			557	ECL	dcbtls	Data Cache Block Touch and Lock Set
X	31	167			146	V	stvehx	Store Vector Element Halfword Indexed
X	31	178		P	416	S	mtmsrd	Move To Machine State Register Doubleword
X	31	181			50	64	stdux	Store Doubleword with Update Indexed
X	31	183			49	B	stwux	Store Word with Update Indexed
X	31	199			147	V	stvewx	Store Vector Element Word Indexed
XO	31	200	SR		62	B	subfze[o][.]	Subtract From Zero Extended
XO	31	202	SR		62	B	addze[o][.]	Add to Zero Extended
X	31	206			623	E.PC	msgsnd	Message Send
X	31	210	32	P	448	S	mtsr	Move To Segment Register
X	31	214			371	64	stdcx.	Store Doubleword Conditional Indexed
X	31	215			47	B	stbx	Store Byte Indexed

Form	Opcode		Mode Dep.[1]	Priv[1]	Page	Cat[1]	Mnemonic	Instruction
	Pri	Ext						
X	31	223			531	E.PD	stbepx	Store Byte by External Process ID Indexed
X	31	230			559	ECL	icblc	Instruction Cache Block Lock Clear
X	31	231			144	V	stvx	Store Vector Indexed
XO	31	232	SR		61	B	subfme[o][.]	Subtract From Minus One Extended
XO	31	233	SR		65	64	mulld[o][.]	Multiply Low Doubleword
XO	31	234	SR		61	B	addme[o][.]	Add to Minus One Extended
XO	31	235	SR		63	B	mullw[o][.]	Multiply Low Word
X	31	238			623	E.PC	msgclr	Message Clear
X	31	242	32	P	448	S	mtsrin	Move To Segment Register Indirect
X	31	246			365	B	dcbtst	Data Cache Block Touch for Store
X	31	247			47	B	stbux	Store Byte with Update Indexed
X	31	255			535	E.PD	dcbtstep	Data Cache Block Touch for Store by External PID
X	31	259		P	527	E	mfdcrx	Move From Device Control Register Indexed
X	31	263			539	E.PD	lvepxl	Load Vector by External Process ID Indexed LRU
XO	31	266	SR		59	B	add[o][.]	Add
X	31	274	64	H	452	S	tlbiel	TLB Invalidate Entry Local
X	31	275			91	E	mfapidi	Move From APID Indirect
X	31	278			360	B	dcbt	Data Cache Block Touch
X	31	279			42	B	lhzx	Load Halfword and Zero Indexed
X	31	284	SR		74	B	eqv[.]	Equivalent
EVX	31	285			538	E.PD	evlddepx	Vector Load Doubleword into Doubleword by External Process ID Indexed
X	31	287			529	E.PD	lhepx	Load Halfword by External Process ID Indexed
X	31	291			91	E	mfdcrux	Move From Device Control Register User-mode Indexed
X	31	295			539	E.PD	lvepx	Load Vector by External Process ID Indexed
X	31	306	64	H	450	S	tlbie	TLB Invalidate Entry
X	31	310			382	EC	eciwx	External Control In Word Indexed
X	31	311			42	B	lhzux	Load Halfword and Zero with Update Indexed
X	31	316	SR		73	B	xor[.]	XOR
X	31	319			533	E.PD	dcbtep	Data Cache Block Touch by External PID
XFX	31	323		S	527	E	mfdcr	Move From Device Control Register
X	31	326			632	E.CD	dcread	Data Cache Read [Alternative Encoding]
XFX	31	334			658	E.PM	mfpmr	Move From Performance Monitor Register
XFX	31	339		O	88,378	B	mfspr	Move From Special Purpose Register
X	31	341			45	64	lwax	Load Word Algebraic Indexed
X	31	343			43	B	lhax	Load Halfword Algebraic Indexed
X	31	359			144	V	lvxl	Load Vector Indexed Last
X	31	370		P	453	S	tlbia	TLB Invalidate All
XFX	31	371			378	S	mftb	Move From Time Base
X	31	373			45	64	lwaux	Load Word Algebraic with Update Indexed
X	31	375			43	B	lhaux	Load Halfword Algebraic with Update Indexed
X	31	387		P	526	E	mtdcrx	Move To Device Control Register Indexed
X	31	390			558	ECL	dcblc	Data Cache Block Lock Clear
X	31	402		P	445	S	slbmte	SLB Move To Entry
X	31	407			48	B	sthx	Store Halfword Indexed
X	31	412	SR		74	B	orc[.]	OR with Complement
XS	31	413	SR		85	64	sradi[.]	Shift Right Algebraic Doubleword Immediate
EVX	31	413			538	E.PD	evstddepx	Vector Store Doubleword into Doubleword by External Process ID Indexed
X	31	415			531	E.PD	sthepx	Store Halfword by External Process ID Indexed
X	31	419			91	E	mtdcrux	Move To Device Control Register User-mode Indexed
X	31	434		P	443	S	slbie	SLB Invalidate Entry
X	31	438			382	EC	ecowx	External Control Out Word Indexed
X	31	439			48	B	sthux	Store Halfword with Update Indexed
X	31	444	SR		73	B	or[.]	OR
XFX	31	451		P	526	E	mtdcr	Move To Device Control Register
X	31	454			629	E.CI	dci	Data Cache Invalidate
XO	31	457	SR		66	64	divdu[o][.]	Divide Doubleword Unsigned
XO	31	459	SR		64	B	divwu[o][.]	Divide Word Unsigned

Form	Opcode Pri	Opcode Ext	Mode Dep.[1]	Priv[1]	Page	Cat[1]	Mnemonic	Instruction
XFX	31	462			658	E.PM	mtpmr	Move To Performance Monitor Register
XFX	31	467		O	87	B	mtspr	Move To Special Purpose Register
X	31	470		P	554	E	dcbi	Data Cache Block Invalidate
X	31	476	SR		73	B	nand[.]	NAND
X	31	486			632	E.CD	dcread	Data Cache Read
X	31	486			558	ECL	icbtls	Instruction Cache Block Touch and Lock Set
X	31	487			147	V	stvxl	Store Vector Indexed Last
XO	31	489	SR		66	64	divd[o][.]	Divide Doubleword
XO	31	491	SR		64	B	divw[o][.]	Divide Word
X	31	498		P	444	S	slbia	SLB Invalidate All
X	31	512			91	B	mcrxr	Move to Condition Register from XER
X	31	533			55	MA	lswx	Load String Word Indexed
X	31	534			51	B	lwbrx	Load Word Byte-Reverse Indexed
X	31	535			115	FP	lfsx	Load Floating-Point Single Indexed
X	31	536	SR		83	B	srw[.]	Shift Right Word
X	31	539	SR		85	64	srd[.]	Shift Right Doubleword
X	31	566		H	453, 561, 651	B	tlbsync	TLB Synchronize
X	31	567			115	FP	lfsux	Load Floating-Point Single with Update Indexed
X	31	595	32	P	449	S	mfsr	Move From Segment Register
X	31	597			55	MA	lswi	Load String Word Immediate
X	31	598			372	B	sync	Synchronize
X	31	599			113	FP	lfdx	Load Floating-Point Double Indexed
X	31	607			537	E.PD	lfdepx	Load Floating-Point Double by External Process ID Indexed
X	31	631			113	FP	lfdux	Load Floating-Point Double with Update Indexed
X	31	659	32	P	449	S	mfsrin	Move From Segment Register Indirect
X	31	661			56	MA	stswx	Store String Word Indexed
X	31	662			51	B	stwbrx	Store Word Byte-Reverse Indexed
X	31	663			115	FP	stfsx	Store Floating-Point Single Indexed
X	31	695			115	FP	stfsux	Store Floating-Point Single with Update Indexed
X	31	725			56	MA	stswi	Store String Word Immediate
X	31	727			116	FP	stfdx	Store Floating-Point Double Indexed
X	31	735			537	E.PD	stfdepx	Store Floating-Point Double by External Process ID Indexed
X	31	758			360	E	dcba	Data Cache Block Allocate
X	31	759			116	FP	stfdux	Store Floating-Point Double with Update Indexed
X	31	775			540	E.PD	stvepxl	Store Vector by External Process ID Indexed LRU
X	31	786		P	560, 649	E	tlbivax	TLB Invalidate Virtual Address Indexed
X	31	790			51	B	lhbrx	Load Halfword Byte-Reverse Indexed
X	31	792	SR		84	B	sraw[.]	Shift Right Algebraic Word
X	31	794	SR		85	64	srad[.]	Shift Right Algebraic Doubleword
X	31	807			540	E.PD	stvepx	Store Vector by External Process ID Indexed
X	31	824	SR		84	B	srawi[.]	Shift Right Algebraic Word Immediate
X	31	851		P	446	S	slbmfev	SLB Move From Entry VSID
X	31	854			374	E	mbar	Memory Barrier
X	31	854			374	S	eieio	Enforce In-order Execution of I/O
X	31	914		P	561, 650	E	tlbsx	TLB Search Indexed
X	31	915		P	446	S	slbmfee	SLB Move From Entry ESID
X	31	918			51	B	sthbrx	Store Halfword Byte-Reverse Indexed
X	31	922	SR		74	B	extsh[.]	Extend Sign Halfword
X	31	946		P	560, 650	E	tlbre	TLB Read Entry
X	31	954	SR		74	B	extsb[.]	Extend Sign Byte
X	31	966			629	E.CI	ici	Instruction Cache Invalidate
X	31	978		P	562, 651	E	tlbwe	TLB Write Entry
X	31	982			359	B	icbi	Instruction Cache Block Invalidate

Form	Opcode Pri	Opcode Ext	Mode Dep.[1]	Priv[1]	Page	Cat[1]	Mnemonic	Instruction
X	31	983			117	FP	stfiwx	Store Floating-Point as Integer Word Indexed
X	31	986	SR		76	64	extsw[.]	Extend Sign Word
X	31	991			536	E.PD	icbiep	Instruction Cache Block Invalidate by External PID
X	31	998			633	E.CD	icread	Instruction Cache Read
X	31	1014			366	B	dcbz	Data Cache Block set to Zero
X	31	1023			536	E.PD	dcbzep	Data Cache Block set to Zero by External PID
D	32				44	B	lwz	Load Word and Zero
D	33				44	B	lwzu	Load Word and Zero with Update
D	34				41	B	lbz	Load Byte and Zero
D	35				41	B	lbzu	Load Byte and Zero with Update
D	36				49	B	stw	Store Word
D	37				49	B	stwu	Store Word with Update
D	38				47	B	stb	Store Byte
D	39				47	B	stbu	Store Byte with Update
D	40				42	B	lhz	Load Halfword and Zero
D	41				42	B	lhzu	Load Halfword and Zero with Update
D	42				43	B	lha	Load Halfword Algebraic
D	43				43	B	lhau	Load Halfword Algebraic with Update
D	44				48	B	sth	Store Halfword
D	45				48	B	sthu	Store Halfword with Update
D	46				52	B	lmw	Load Multiple Word
D	47				53	B	stmw	Store Multiple Word
D	48				115	FP	lfs	Load Floating-Point Single
D	49				115	FP	lfsu	Load Floating-Point Single with Update
D	50				113	FP	lfd	Load Floating-Point Double
D	51				113	FP	lfdu	Load Floating-Point Double with Update
D	52				115	FP	stfs	Store Floating-Point Single
D	53				115	FP	stfsu	Store Floating-Point Single with Update
D	54				116	FP	stfd	Store Floating-Point Double
D	55				116	FP	stfdu	Store Floating-Point Double with Update
DQ	56			P	410	LSQ	lq	Load Quadword
DS	58	0			46	64	ld	Load Doubleword
DS	58	1			46	64	ldu	Load Doubleword with Update
DS	58	2			45	64	lwa	Load Word Algebraic
A	59	18			120	FP[R]	fdivs[.]	Floating Divide Single
A	59	20			119	FP[R]	fsubs[.]	Floating Subtract Single
A	59	21			119	FP[R]	fadds[.]	Floating Add Single
A	59	22			121	FP[R]	fsqrts[.]	Floating Square Root Single
A	59	24			121	FP[R]	fres[.]	Floating Reciprocal Estimate Single
A	59	25			120	FP[R]	fmuls[.]	Floating Multiply Single
A	59	26			122	FP[R].in	frsqrtes[.]	Floating Reciprocal Square Root Estimate Single
A	59	28			123	FP[R]	fmsubs[.]	Floating Multiply-Subtract Single
A	59	29			123	FP[R]	fmadds[.]	Floating Multiply-Add Single
A	59	30			124	FP[R]	fnmsubs[.]	Floating Negative Multiply-Subtract Single
A	59	31			124	FP[R]	fnmadds[.]	Floating Negative Multiply-Add Single
DS	62	0			50	64	std	Store Doubleword
DS	62	1			50	64	stdu	Store Doubleword with Update
DS	62	2		P	410	LSQ	stq	Store Quadword
X	63	0			129	FP	fcmpu	Floating Compare Unordered
X	63	12			125	FP[R].in	frsp[.]	Floating Round to Single-Precision
X	63	14			126	FP[R]	fctiw[.]	Floating Convert To Integer Word
X	63	15			127	FP[R]	fctiwz[.]	Floating Convert To Integer Word with round toward Zero
A	63	18			120	FP[R]	fdiv[.]	Floating Divide
A	63	20			119	FP[R]	fsub[.]	Floating Subtract
A	63	21			119	FP[R]	fadd[.]	Floating Add
A	63	22			121	FP[R]	fsqrt[.]	Floating Square Root
A	63	23			130	FP[R]	fsel[.]	Floating Select
A	63	24			121	FP[R]	fre[.]	Floating Reciprocal Estimate
A	63	25			120	FP[R]	fmul[.]	Floating Multiply
A	63	26			122	FP[R].in	frsqrte[.]	Floating Reciprocal Square Root Estimate

Form	Opcode		Mode Dep.[1]	Priv[1]	Page	Cat[1]	Mnemonic	Instruction
	Pri	Ext						
A	63	28			123	FP[R]	fmsub[.]	Floating Multiply-Subtract
A	63	29			123	FP[R]	fmadd[.]	Floating Multiply-Add
A	63	30			124	FP[R]	fnmsub[.]	Floating Negative Multiply-Subtract
A	63	31			124	FP[R]	fnmadd[.]	Floating Negative Multiply-Add
X	63	32			129	FP	fcmpo	Floating Compare Ordered
X	63	38			132	FP[R]	mtfsb1[.]	Move To FPSCR Bit 1
X	63	40			118	FP[R]	fneg[.]	Floating Negate
X	63	64			131	FP	mcrfs	Move to Condition Register from FPSCR
X	63	70			132	FP[R]	mtfsb0[.]	Move To FPSCR Bit 0
X	63	72			118	FP[R]	fmr[.]	Floating Move Register
X	63	134			131	FP[R]	mtfsfi[.]	Move To FPSCR Field Immediate
X	63	136			118	FP[R]	fnabs[.]	Floating Negative Absolute Value
X	63	264			118	FP[R]	fabs[.]	Floating Absolute Value
X	63	392			128	FP[R].in	frin[.]	Floating Round to Integer Nearest
X	63	424			128	FP[R].in	friz[.]	Floating Round to Integer Toward Zero
X	63	456			128	FP[R].in	frip[.]	Floating Round to Integer Plus
X	63	488			128	FP[R]	frim[.]	Floating Round to Integer Minus
X	63	583			131	FP[R]	mffs[.]	Move From FPSCR
XFL	63	711			131	FP[R]	mtfsf[.]	Move To FPSCR Fields
X	63	814			125	FP[R]	fctid[.]	Floating Convert To Integer Doubleword
X	63	815			126	FP[R]	fctidz[.]	Floating Convert To Integer Doubleword with round toward Zero
X	63	846			127	FP[R]	fcfid[.]	Floating Convert From Integer Doubleword

1 See the key to the mode dependency and privilege columns on page 839 and the key to the category column in Section 1.3.5 of Book I.

Appendix I. Power ISA Instruction Set Sorted by Mnemonic

This appendix lists all the instructions in the Power ISA, in order by mnemonic.

Form	Opcode		Mode Dep.[1]	Priv[1]	Page	Cat[1]	Mnemonic	Instruction
	Pri	Ext						
XO	31	266	SR		59	B	add[o][.]	Add
XO	31	10	SR		60	B	addc[o][.]	Add Carrying
XO	31	138	SR		61	B	adde[o][.]	Add Extended
D	14				58	B	addi	Add Immediate
D	12		SR		59	B	addic	Add Immediate Carrying
D	13		SR		59	B	addic.	Add Immediate Carrying and Record
D	15				58	B	addis	Add Immediate Shifted
XO	31	234	SR		61	B	addme[o][.]	Add to Minus One Extended
XO	31	202	SR		62	B	addze[o][.]	Add to Zero Extended
X	31	28	SR		73	B	and[.]	AND
X	31	60	SR		74	B	andc[.]	AND with Complement
D	28		SR		71	B	andi.	AND Immediate
D	29		SR		71	B	andis.	AND Immediate Shifted
I	18				31	B	b[l][a]	Branch
B	16		CT		31	B	bc[l][a]	Branch Conditional
XL	19	528	CT		32	B	bcctr[l]	Branch Conditional to Count Register
XL	19	16	CT		32	B	bclr[l]	Branch Conditional to Link Register
EVX	4	527			208	SP	brinc	Bit Reversed Increment
X	31	0			67	B	cmp	Compare
D	11				67	B	cmpi	Compare Immediate
X	31	32			68	B	cmpl	Compare Logical
D	10				68	B	cmpli	Compare Logical Immediate
X	31	58	SR		76	64	cntlzd[.]	Count Leading Zeros Doubleword
X	31	26	SR		74	B	cntlzw[.]	Count Leading Zeros Word
XL	19	257			33	B	crand	Condition Register AND
XL	19	129			34	B	crandc	Condition Register AND with Complement
XL	19	289			34	B	creqv	Condition Register Equivalent
XL	19	225			33	B	crnand	Condition Register NAND
XL	19	33			34	B	crnor	Condition Register NOR
XL	19	449			33	B	cror	Condition Register OR
XL	19	417			34	B	crorc	Condition Register OR with Complement
XL	19	193			33	B	crxor	Condition Register XOR
X	31	758			360	E	dcba	Data Cache Block Allocate
X	31	86			367	B	dcbf	Data Cache Block Flush
X	31	127			534	E.PD	dcbfep	Data Cache Block Flush by External PID
X	31	470		P	554	E	dcbi	Data Cache Block Invalidate
X	31	390			558	ECL	dcblc	Data Cache Block Lock Clear
X	31	54			366	B	dcbst	Data Cache Block Store
X	31	63			533	E.PD	dcbstep	Data Cache Block Store by External PID
X	31	278			360	B	dcbt	Data Cache Block Touch
X	31	319			533	E.PD	dcbtep	Data Cache Block Touch by External PID
X	31	166			557	ECL	dcbtls	Data Cache Block Touch and Lock Set
X	31	246			365	B	dcbtst	Data Cache Block Touch for Store
X	31	255			535	E.PD	dcbtstep	Data Cache Block Touch for Store by External PID

Form	Opcode Pri	Opcode Ext	Mode Dep.[1]	Priv[1]	Page	Cat[1]	Mnemonic	Instruction
X	31	134			557	ECL	dcbtstls	Data Cache Block Touch for Store and Lock Set
X	31	1014			366	B	dcbz	Data Cache Block set to Zero
X	31	1023			536	E.PD	dcbzep	Data Cache Block set to Zero by External PID
X	31	454			629	E.CI	dci	Data Cache Invalidate
X	31	326			632	E.CD	dcread	Data Cache Read [Alternative Encoding]
X	31	486			632	E.CD	dcread	Data Cache Read
XO	31	489	SR		66	64	divd[o][.]	Divide Doubleword
XO	31	457	SR		66	64	divdu[o][.]	Divide Doubleword Unsigned
XO	31	491	SR		64	B	divw[o][.]	Divide Word
XO	31	459	SR		64	B	divwu[o][.]	Divide Word Unsigned
X	31	78			287	LMV	dlmzb[.]	Determine Leftmost Zero Byte
XFX	19	198			620	E.ED	dnh	Debugger Notify Halt
X	31	310			382	EC	eciwx	External Control In Word Indexed
X	31	438			382	EC	ecowx	External Control Out Word Indexed
EVX	4	740			274	SP.FD	efdabs	Floating-Point Double-Precision Absolute Value
EVX	4	736			275	SP.FD	efdadd	Floating-Point Double-Precision Add
EVX	4	751			280	SP.FD	efdcfs	Floating-Point Double-Precision Convert from Single-Precision
EVX	4	755			278	SP.FD	efdcfsf	Convert Floating-Point Double-Precision from Signed Fraction
EVX	4	753			277	SP.FD	efdcfsi	Convert Floating-Point Double-Precision from Signed Integer
EVX	4	739			278	SP.FD	efdcfsid	Convert Floating-Point Double-Precision from Signed Integer Doubleword
EVX	4	754			278	SP.FD	efdcfuf	Convert Floating-Point Double-Precision from Unsigned Fraction
EVX	4	752			277	SP.FD	efdcfui	Convert Floating-Point Double-Precision from Unsigned Integer
EVX	4	738			278	SP.FD	efdcfuid	Convert Floating-Point Double-Precision from Unsigned Integer Doubleword
EVX	4	750			276	SP.FD	efdcmpeq	Floating-Point Double-Precision Compare Equal
EVX	4	748			276	SP.FD	efdcmpgt	Floating-Point Double-Precision Compare Greater Than
EVX	4	749			276	SP.FD	efdcmplt	Floating-Point Double-Precision Compare Less Than
EVX	4	759			280	SP.FD	efdctsf	Convert Floating-Point Double-Precision to Signed Fraction
EVX	4	757			278	SP.FD	efdctsi	Convert Floating-Point Double-Precision to Signed Integer
EVX	4	747			279	SP.FD	efdctsidz	Convert Floating-Point Double-Precision to Signed Integer Doubleword with Round toward Zero
EVX	4	762			280	SP.FD	efdctsiz	Convert Floating-Point Double-Precision to Signed Integer with Round toward Zero
EVX	4	758			280	SP.FD	efdctuf	Convert Floating-Point Double-Precision to Unsigned Fraction
EVX	4	756			278	SP.FD	efdctui	Convert Floating-Point Double-Precision to Unsigned Integer
EVX	4	746			279	SP.FD	efdctuidz	Convert Floating-Point Double-Precision to Unsigned Integer Doubleword with Round toward Zero
EVX	4	760			280	SP.FD	efdctuiz	Convert Floating-Point Double-Precision to Unsigned Integer with Round toward Zero
EVX	4	745			275	SP.FD	efddiv	Floating-Point Double-Precision Divide
EVX	4	744			275	SP.FD	efdmul	Floating-Point Double-Precision Multiply
EVX	4	741			274	SP.FD	efdnabs	Floating-Point Double-Precision Negative Absolute Value
EVX	4	742			274	SP.FD	efdneg	Floating-Point Double-Precision Negate
EVX	4	737			275	SP.FD	efdsub	Floating-Point Double-Precision Subtract
EVX	4	766			277	SP.FD	efdtsteq	Floating-Point Double-Precision Test Equal
EVX	4	764			276	SP.FD	efdtstgt	Floating-Point Double-Precision Test Greater Than
EVX	4	765			277	SP.FD	efdtstlt	Floating-Point Double-Precision Test Less Than
EVX	4	708			267	SP.FS	efsabs	Floating-Point Single-Precision Absolute Value

Form	Opcode		Mode Dep.[1]	Priv[1]	Page	Cat[1]	Mnemonic	Instruction
	Pri	Ext						
EVX	4	704			268	SP.FS	efsadd	Floating-Point Single-Precision Add
EVX	4	719			281	SP.FD	efscfd	Floating-Point Single-Precision Convert from Double-Precision
EVX	4	723			272	SP.FS	efscfsf	Convert Floating-Point Single-Precision from Signed Fraction
EVX	4	721			272	SP.FS	efscfsi	Convert Floating-Point Single-Precision from Signed Integer
EVX	4	722			272	SP.FS	efscfuf	Convert Floating-Point Single-Precision from Unsigned Fraction
EVX	4	720			272	SP.FS	efscfui	Convert Floating-Point Single-Precision from Unsigned Integer
EVX	4	718			270	SP.FS	efscmpeq	Floating-Point Single-Precision Compare Equal
EVX	4	716			269	SP.FS	efscmpgt	Floating-Point Single-Precision Compare Greater Than
EVX	4	717			269	SP.FS	efscmplt	Floating-Point Single-Precision Compare Less Than
EVX	4	727			273	SP.FS	efsctsf	Convert Floating-Point Single-Precision to Signed Fraction
EVX	4	725			272	SP.FS	efsctsi	Convert Floating-Point Single-Precision to Signed Integer
EVX	4	730			273	SP.FS	efsctsiz	Convert Floating-Point Single-Precision to Signed Integer with Round toward Zero
EVX	4	726			273	SP.FS	efsctuf	Convert Floating-Point Single-Precision to Unsigned Fraction
EVX	4	724			272	SP.FS	efsctui	Convert Floating-Point Single-Precision to Unsigned Integer
EVX	4	728			273	SP.FS	efsctuiz	Convert Floating-Point Single-Precision to Unsigned Integer with Round toward Zero
EVX	4	713			268	SP.FS	efsdiv	Floating-Point Single-Precision Divide
EVX	4	712			268	SP.FS	efsmul	Floating-Point Single-Precision Multiply
EVX	4	709			267	SP.FS	efsnabs	Floating-Point Single-Precision Negative Absolute Value
EVX	4	710			267	SP.FS	efsneg	Floating-Point Single-Precision Negate
EVX	4	705			268	SP.FS	efssub	Floating-Point Single-Precision Subtract
EVX	4	734			271	SP.FS	efststeq	Floating-Point Single-Precision Test Equal
EVX	4	732			270	SP.FS	efststgt	Floating-Point Single-Precision Test Greater Than
EVX	4	733			271	SP.FS	efststlt	Floating-Point Single-Precision Test Less Than
X	31	854			374	S	eieio	Enforce In-order Execution of I/O
X	31	284	SR		74	B	eqv[.]	Equivalent
EVX	4	520			208	SP	evabs	Vector Absolute Value
EVX	4	514			208	SP	evaddiw	Vector Add Immediate Word
EVX	4	1225			208	SP	evaddsmiaaw	Vector Add Signed, Modulo, Integer to Accumulator Word
EVX	4	1217			209	SP	evaddssiaaw	Vector Add Signed, Saturate, Integer to Accumulator Word
EVX	4	1224			209	SP	evaddumiaaw	Vector Add Unsigned, Modulo, Integer to Accumulator Word
EVX	4	1216			209	SP	evaddusiaaw	Vector Add Unsigned, Saturate, Integer to Accumulator Word
EVX	4	512			209	SP	evaddw	Vector Add Word
EVX	4	529			210	SP	evand	Vector AND
EVX	4	530			210	SP	evandc	Vector AND with Complement
EVX	4	564			210	SP	evcmpeq	Vector Compare Equal
EVX	4	561			210	SP	evcmpgts	Vector Compare Greater Than Signed
EVX	4	560			211	SP	evcmpgtu	Vector Compare Greater Than Unsigned
EVX	4	563			211	SP	evcmplts	Vector Compare Less Than Signed
EVX	4	562			211	SP	evcmpltu	Vector Compare Less Than Unsigned
EVX	4	526			212	SP	evcntlsw	Vector Count Leading Signed Bits Word
EVX	4	525			212	SP	evcntlzw	Vector Count Leading Zeros Word
EVX	4	1222			212	SP	evdivws	Vector Divide Word Signed
EVX	4	1223			213	SP	evdivwu	Vector Divide Word Unsigned
EVX	4	537			213	SP	eveqv	Vector Equivalent

Form	Opcode Pri	Opcode Ext	Mode Dep.[1]	Priv[1]	Page	Cat[1]	Mnemonic	Instruction
EVX	4	522			213	SP	evextsb	Vector Extend Sign Byte
EVX	4	523			213	SP	evextsh	Vector Extend Sign Halfword
EVX	4	644			259	SP.FV	evfsabs	Vector Floating-Point Single-Precision Absolute Value
EVX	4	640			260	SP.FV	evfsadd	Vector Floating-Point Single-Precision Add
EVX	4	659			264	SP.FV	evfscfsf	Vector Convert Floating-Point Single-Precision from Signed Fraction
EVX	4	657			264	SP.FV	evfscfsi	Vector Convert Floating-Point Single-Precision from Signed Integer
EVX	4	658			264	SP.FV	evfscfuf	Vector Convert Floating-Point Single-Precision from Unsigned Fraction
EVX	4	656			264	SP.FV	evfscfui	Vector Convert Floating-Point Single-Precision from Unsigned Integer
EVX	4	654			262	SP.FV	evfscmpeq	Vector Floating-Point Single-Precision Compare Equal
EVX	4	652			261	SP.FV	evfscmpgt	Vector Floating-Point Single-Precision Compare Greater Than
EVX	4	653			261	SP.FV	evfscmplt	Vector Floating-Point Single-Precision Compare Less Than
EVX	4	663			266	SP.FV	evfsctsf	Vector Convert Floating-Point Single-Precision to Signed Fraction
EVX	4	661			265	SP.FV	evfsctsi	Vector Convert Floating-Point Single-Precision to Signed Integer
EVX	4	666			265	SP.FV	evfsctsiz	Vector Convert Floating-Point Single-Precision to Signed Integer with Round toward Zero
EVX	4	662			266	SP.FV	evfsctuf	Vector Convert Floating-Point Single-Precision to Unsigned Fraction
EVX	4	660			265	SP.FV	evfsctui	Vector Convert Floating-Point Single-Precision to Unsigned Integer
EVX	4	664			265	SP.FV	evfsctuiz	Vector Convert Floating-Point Single-Precision to Unsigned Integer with Round toward Zero
EVX	4	649			260	SP.FV	evfsdiv	Vector Floating-Point Single-Precision Divide
EVX	4	648			260	SP.FV	evfsmul	Vector Floating-Point Single-Precision Multiply
EVX	4	645			259	SP.FV	evfsnabs	Vector Floating-Point Single-Precision Negative Absolute Value
EVX	4	646			259	SP.FV	evfsneg	Vector Floating-Point Single-Precision Negate
EVX	4	641			260	SP.FV	evfssub	Vector Floating-Point Single-Precision Subtract
EVX	4	670			263	SP.FV	evfststeq	Vector Floating-Point Single-Precision Test Equal
EVX	4	668			262	SP.FV	evfststgt	Vector Floating-Point Single-Precision Test Greater Than
EVX	4	669			263	SP.FV	evfststlt	Vector Floating-Point Single-Precision Test Less Than
EVX	4	769			214	SP	evldd	Vector Load Double Word into Double Word
EVX	31	285			538	E.PD	evlddepx	Vector Load Doubleword into Doubleword by External Process ID Indexed
EVX	4	768			214	SP	evlddx	Vector Load Double Word into Double Word Indexed
EVX	4	773			214	SP	evldh	Vector Load Double into Four Halfwords
EVX	4	772			214	SP	evldhx	Vector Load Double into Four Halfwords Indexed
EVX	4	771			215	SP	evldw	Vector Load Double into Two Words
EVX	4	770			215	SP	evldwx	Vector Load Double into Two Words Indexed
EVX	4	777			215	SP	evlhhesplat	Vector Load Halfword into Halfwords Even and Splat
EVX	4	776			215	SP	evlhhesplatx	Vector Load Halfword into Halfwords Even and Splat Indexed
EVX	4	783			216	SP	evlhhossplat	Vector Load Halfword into Halfword Odd Signed and Splat
EVX	4	782			216	SP	evlhhossplatx	Vector Load Halfword into Halfword Odd Signed and Splat Indexed
EVX	4	781			216	SP	evlhhousplat	Vector Load Halfword into Halfword Odd Unsigned and Splat
EVX	4	780			216	SP	evlhhousplatx	Vector Load Halfword into Halfword Odd Unsigned and Splat Indexed
EVX	4	785			217	SP	evlwhe	Vector Load Word into Two Halfwords Even
EVX	4	784			217	SP	evlwhex	Vector Load Word into Two Halfwords Even Indexed

Form	Opcode		Mode Dep.[1]	Priv[1]	Page	Cat[1]	Mnemonic	Instruction
	Pri	Ext						
EVX	4	791			217	SP	evlwhos	Vector Load Word into Two Halfwords Odd Signed (with sign extension)
EVX	4	790			217	SP	evlwhosx	Vector Load Word into Two Halfwords Odd Signed Indexed (with sign extension)
EVX	4	789			218	SP	evlwhou	Vector Load Word into Two Halfwords Odd Unsigned (zero-extended)
EVX	4	788			218	SP	evlwhoux	Vector Load Word into Two Halfwords Odd Unsigned Indexed (zero-extended)
EVX	4	797			218	SP	evlwhsplat	Vector Load Word into Two Halfwords and Splat
EVX	4	796			218	SP	evlwhsplatx	Vector Load Word into Two Halfwords and Splat Indexed
EVX	4	793			219	SP	evlwwsplat	Vector Load Word into Word and Splat
EVX	4	792			219	SP	evlwwsplatx	Vector Load Word into Word and Splat Indexed
EVX	4	556			219	SP	evmergehi	Vector Merge High
EVX	4	558			220	SP	evmergehilo	Vector Merge High/Low
EVX	4	557			219	SP	evmergelo	Vector Merge Low
EVX	4	559			220	SP	evmergelohi	Vector Merge Low/High
EVX	4	1323			220	SP	evmhegsmfaa	Vector Multiply Halfwords, Even, Guarded, Signed, Modulo, Fractional and Accumulate
EVX	4	1451			220	SP	evmhegsmfan	Vector Multiply Halfwords, Even, Guarded, Signed, Modulo, Fractional and Accumulate Negative
EVX	4	1321			221	SP	evmhegsmiaa	Vector Multiply Halfwords, Even, Guarded, Signed, Modulo, Integer and Accumulate
EVX	4	1449			221	SP	evmhegsmian	Vector Multiply Halfwords, Even, Guarded, Signed, Modulo, Integer and Accumulate Negative
EVX	4	1320			221	SP	evmhegumiaa	Vector Multiply Halfwords, Even, Guarded, Unsigned, Modulo, Integer and Accumulate
EVX	4	1448			221	SP	evmhegumian	Vector Multiply Halfwords, Even, Guarded, Unsigned, Modulo, Integer and Accumulate Negative
EVX	4	1035			222	SP	evmhesmf	Vector Multiply Halfwords, Even, Signed, Modulo, Fractional
EVX	4	1067			222	SP	evmhesmfa	Vector Multiply Halfwords, Even, Signed, Modulo, Fractional to Accumulator
EVX	4	1291			222	SP	evmhesmfaaw	Vector Multiply Halfwords, Even, Signed, Modulo, Fractional and Accumulate into Words
EVX	4	1419			222	SP	evmhesmfanw	Vector Multiply Halfwords, Even, Signed, Modulo, Fractional and Accumulate Negative into Words
EVX	4	1033			223	SP	evmhesmi	Vector Multiply Halfwords, Even, Signed, Modulo, Integer
EVX	4	1065			223	SP	evmhesmia	Vector Multiply Halfwords, Even, Signed, Modulo, Integer to Accumulator
EVX	4	1289			223	SP	evmhesmiaaw	Vector Multiply Halfwords, Even, Signed, Modulo, Integer and Accumulate into Words
EVX	4	1417			223	SP	evmhesmianw	Vector Multiply Halfwords, Even, Signed, Modulo, Integer and Accumulate Negative into Words
EVX	4	1027			224	SP	evmhessf	Vector Multiply Halfwords, Even, Signed, Saturate, Fractional
EVX	4	1059			224	SP	evmhessfa	Vector Multiply Halfwords, Even, Signed, Saturate, Fractional to Accumulator
EVX	4	1283			225	SP	evmhessfaaw	Vector Multiply Halfwords, Even, Signed, Saturate, Fractional and Accumulate into Words
EVX	4	1411			225	SP	evmhessfanw	Vector Multiply Halfwords, Even, Signed, Saturate, Fractional and Accumulate Negative into Words
EVX	4	1281			226	SP	evmhessiaaw	Vector Multiply Halfwords, Even, Signed, Saturate, Integer and Accumulate into Words
EVX	4	1409			226	SP	evmhessianw	Vector Multiply Halfwords, Even, Signed, Saturate, Integer and Accumulate Negative into Words
EVX	4	1032			227	SP	evmheumi	Vector Multiply Halfwords, Even, Unsigned, Modulo, Integer

Form	Opcode Pri	Opcode Ext	Mode Dep.[1]	Priv[1]	Page	Cat[1]	Mnemonic	Instruction
EVX	4	1064			227	SP	evmheumia	Vector Multiply Halfwords, Even, Unsigned, Modulo, Integer to Accumulator
EVX	4	1288			227	SP	evmheumiaaw	Vector Multiply Halfwords, Even, Unsigned, Modulo, Integer and Accumulate into Words
EVX	4	1416			227	SP	evmheumianw	Vector Multiply Halfwords, Even, Unsigned, Modulo, Integer and Accumulate Negative into Words
EVX	4	1280			228	SP	evmheusiaaw	Vector Multiply Halfwords, Even, Unsigned, Saturate, Integer and Accumulate into Words
EVX	4	1408			228	SP	evmheusianw	Vector Multiply Halfwords, Even, Unsigned, Saturate, Integer and Accumulate Negative into Words
EVX	4	1327			229	SP	evmhogsmfaa	Vector Multiply Halfwords, Odd, Guarded, Signed, Modulo, Fractional and Accumulate
EVX	4	1455			229	SP	evmhogsmfan	Vector Multiply Halfwords, Odd, Guarded, Signed, Modulo, Fractional and Accumulate Negative
EVX	4	1325			229	SP	evmhogsmiaa	Vector Multiply Halfwords, Odd, Guarded, Signed, Modulo, Integer and Accumulate
EVX	4	1453			229	SP	evmhogsmian	Vector Multiply Halfwords, Odd, Guarded, Signed, Modulo, Integer and Accumulate Negative
EVX	4	1324			230	SP	evmhogumiaa	Vector Multiply Halfwords, Odd, Guarded, Unsigned, Modulo, Integer and Accumulate
EVX	4	1452			230	SP	evmhogumian	Vector Multiply Halfwords, Odd, Guarded, Unsigned, Modulo, Integer and Accumulate Negative
EVX	4	1039			230	SP	evmhosmf	Vector Multiply Halfwords, Odd, Signed, Modulo, Fractional
EVX	4	1071			230	SP	evmhosmfa	Vector Multiply Halfwords, Odd, Signed, Modulo, Fractional to Accumulator
EVX	4	1295			231	SP	evmhosmfaaw	Vector Multiply Halfwords, Odd, Signed, Modulo, Fractional and Accumulate into Words
EVX	4	1423			231	SP	evmhosmfanw	Vector Multiply Halfwords, Odd, Signed, Modulo, Fractional and Accumulate Negative into Words
EVX	4	1037			231	SP	evmhosmi	Vector Multiply Halfwords, Odd, Signed, Modulo, Integer
EVX	4	1069			231	SP	evmhosmia	Vector Multiply Halfwords, Odd, Signed, Modulo, Integer to Accumulator
EVX	4	1293			232	SP	evmhosmiaaw	Vector Multiply Halfwords, Odd, Signed, Modulo, Integer and Accumulate into Words
EVX	4	1421			231	SP	evmhosmianw	Vector Multiply Halfwords, Odd, Signed, Modulo, Integer and Accumulate Negative into Words
EVX	4	1031			233	SP	evmhossf	Vector Multiply Halfwords, Odd, Signed, Saturate, Fractional
EVX	4	1063			233	SP	evmhossfa	Vector Multiply Halfwords, Odd, Signed, Saturate, Fractional to Accumulator
EVX	4	1287			234	SP	evmhossfaaw	Vector Multiply Halfwords, Odd, Signed, Saturate, Fractional and Accumulate into Words
EVX	4	1415			234	SP	evmhossfanw	Vector Multiply Halfwords, Odd, Signed, Saturate, Fractional and Accumulate Negative into Words
EVX	4	1285			235	SP	evmhossiaaw	Vector Multiply Halfwords, Odd, Signed, Saturate, Integer and Accumulate into Words
EVX	4	1413			235	SP	evmhossianw	Vector Multiply Halfwords, Odd, Signed, Saturate, Integer and Accumulate Negative into Words
EVX	4	1036			235	SP	evmhoumi	Vector Multiply Halfwords, Odd, Unsigned, Modulo, Integer
EVX	4	1068			235	SP	evmhoumia	Vector Multiply Halfwords, Odd, Unsigned, Modulo, Integer to Accumulator
EVX	4	1292			236	SP	evmhoumiaaw	Vector Multiply Halfwords, Odd, Unsigned, Modulo, Integer and Accumulate into Words
EVX	4	1420			232	SP	evmhoumianw	Vector Multiply Halfwords, Odd, Unsigned, Modulo, Integer and Accumulate Negative into Words

Form	Opcode Pri	Opcode Ext	Mode Dep.[1]	Priv[1]	Page	Cat[1]	Mnemonic	Instruction
EVX	4	1284			236	SP	evmhousiaaw	Vector Multiply Halfwords, Odd, Unsigned, Saturate, Integer and Accumulate into Words
EVX	4	1412			236	SP	evmhousianw	Vector Multiply Halfwords, Odd, Unsigned, Saturate, Integer and Accumulate Negative into Words
EVX	4	1220			237	SP	evmra	Initialize Accumulator
EVX	4	1103			237	SP	evmwhsmf	Vector Multiply Word High Signed, Modulo, Fractional
EVX	4	1135			237	SP	evmwhsmfa	Vector Multiply Word High Signed, Modulo, Fractional to Accumulator
EVX	4	1101			237	SP	evmwhsmi	Vector Multiply Word High Signed, Modulo, Integer
EVX	4	1133			237	SP	evmwhsmia	Vector Multiply Word High Signed, Modulo, Integer to Accumulator
EVX	4	1095			238	SP	evmwhssf	Vector Multiply Word High Signed, Saturate, Fractional
EVX	4	1127			238	SP	evmwhssfa	Vector Multiply Word High Signed, Saturate, Fractional to Accumulator
EVX	4	1100			238	SP	evmwhumi	Vector Multiply Word High Unsigned, Modulo, Integer
EVX	4	1132			238	SP	evmwhumia	Vector Multiply Word High Unsigned, Modulo, Integer to Accumulator
EVX	4	1353			239	SP	evmwlsmiaaw	Vector Multiply Word Low Signed, Modulo, Integer and Accumulate into Words
EVX	4	1481			239	SP	evmwlsmianw	Vector Multiply Word Low Signed, Modulo, Integer and Accumulate Negative in Words
EVX	4	1345			239	SP	evmwlssiaaw	Vector Multiply Word Low Signed, Saturate, Integer and Accumulate into Words
EVX	4	1473			239	SP	evmwlssianw	Vector Multiply Word Low Signed, Saturate, Integer and Accumulate Negative in Words
EVX	4	1096			240	SP	evmwlumi	Vector Multiply Word Low Unsigned, Modulo, Integer
EVX	4	1128			240	SP	evmwlumia	Vector Multiply Word Low Unsigned, Modulo, Integer to Accumulator
EVX	4	1352			240	SP	evmwlumiaaw	Vector Multiply Word Low Unsigned, Modulo, Integer and Accumulate into Words
EVX	4	1480			240	SP	evmwlumianw	Vector Multiply Word Low Unsigned, Modulo, Integer and Accumulate Negative in Words
EVX	4	1344			241	SP	evmwlusiaaw	Vector Multiply Word Low Unsigned, Saturate, Integer and Accumulate into Words
EVX	4	1472			241	SP	evmwlusianw	Vector Multiply Word Low Unsigned, Saturate, Integer and Accumulate Negative in Words
EVX	4	1115			241	SP	evmwsmf	Vector Multiply Word Signed, Modulo, Fractional
EVX	4	1147			241	SP	evmwsmfa	Vector Multiply Word Signed, Modulo, Fractional to Accumulator
EVX	4	1371			242	SP	evmwsmfaa	Vector Multiply Word Signed, Modulo, Fractional and Accumulate
EVX	4	1499			242	SP	evmwsmfan	Vector Multiply Word Signed, Modulo, Fractional and Accumulate Negative
EVX	4	1113			242	SP	evmwsmi	Vector Multiply Word Signed, Modulo, Integer
EVX	4	1145			242	SP	evmwsmia	Vector Multiply Word Signed, Modulo, Integer to Accu-mulator
EVX	4	1369			242	SP	evmwsmiaa	Vector Multiply Word Signed, Modulo, Integer and Accumulate
EVX	4	1497			242	SP	evmwsmian	Vector Multiply Word Signed, Modulo, Integer and Accumulate Negative
EVX	4	1107			243	SP	evmwssf	Vector Multiply Word Signed, Saturate, Fractional
EVX	4	1139			243	SP	evmwssfa	Vector Multiply Word Signed, Saturate, Fractional to Accumulator
EVX	4	1363			243	SP	evmwssfaa	Vector Multiply Word Signed, Saturate, Fractional and Accumulate
EVX	4	1491			244	SP	evmwssfan	Vector Multiply Word Signed, Saturate, Fractional and Accumulate Negative
EVX	4	1112			244	SP	evmwumi	Vector Multiply Word Unsigned, Modulo, Integer

Form	Opcode		Mode Dep.[1]	Priv[1]	Page	Cat[1]	Mnemonic	Instruction
	Pri	Ext						
EVX	4	1144			244	SP	evmwumia	Vector Multiply Word Unsigned, Modulo, Integer to Accumulator
EVX	4	1368			245	SP	evmwumiaa	Vector Multiply Word Unsigned, Modulo, Integer and Accumulate
EVX	4	1496			245	SP	evmwumian	Vector Multiply Word Unsigned, Modulo, Integer and Accumulate Negative
EVX	4	542			245	SP	evnand	Vector NAND
EVX	4	521			245	SP	evneg	Vector Negate
EVX	4	536			245	SP	evnor	Vector NOR
EVX	4	535			246	SP	evor	Vector OR
EVX	4	539			246	SP	evorc	Vector OR with Complement
EVX	4	552			246	SP	evrlw	Vector Rotate Left Word
EVX	4	554			247	SP	evrlwi	Vector Rotate Left Word Immediate
EVX	4	524			247	SP	evrndw	Vector Round Word
EVS	4	79			247	SP	evsel	Vector Select
EVX	4	548			248	SP	evslw	Vector Shift Left Word
EVX	4	550			248	SP	evslwi	Vector Shift Left Word Immediate
EVX	4	555			248	SP	evsplatfi	Vector Splat Fractional Immediate
EVX	4	553			248	SP	evsplati	Vector Splat Immediate
EVX	4	547			248	SP	evsrwis	Vector Shift Right Word Immediate Signed
EVX	4	546			248	SP	evsrwiu	Vector Shift Right Word Immediate Unsigned
EVX	4	545			249	SP	evsrws	Vector Shift Right Word Signed
EVX	4	544			249	SP	evsrwu	Vector Shift Right Word Unsigned
EVX	4	801			249	SP	evstdd	Vector Store Double of Double
EVX	31	413			538	E.PD	evstddepx	Vector Store Doubleword into Doubleword by External Process ID Indexed
EVX	4	800			249	SP	evstddx	Vector Store Double of Double Indexed
EVX	4	805			250	SP	evstdh	Vector Store Double of Four Halfwords
EVX	4	804			250	SP	evstdhx	Vector Store Double of Four Halfwords Indexed
EVX	4	803			250	SP	evstdw	Vector Store Double of Two Words
EVX	4	802			250	SP	evstdwx	Vector Store Double of Two Words Indexed
EVX	4	817			251	SP	evstwhe	Vector Store Word of Two Halfwords from Even
EVX	4	816			251	SP	evstwhex	Vector Store Word of Two Halfwords from Even Indexed
EVX	4	821			251	SP	evstwho	Vector Store Word of Two Halfwords from Odd
EVX	4	820			251	SP	evstwhox	Vector Store Word of Two Halfwords from Odd Indexed
EVX	4	825			251	SP	evstwwe	Vector Store Word of Word from Even
EVX	4	824			251	SP	evstwwex	Vector Store Word of Word from Even Indexed
EVX	4	829			252	SP	evstwwo	Vector Store Word of Word from Odd
EVX	4	828			252	SP	evstwwox	Vector Store Word of Word from Odd Indexed
EVX	4	1227			252	SP	evsubfsmiaaw	Vector Subtract Signed, Modulo, Integer to Accumulator Word
EVX	4	1219			252	SP	evsubfssiaaw	Vector Subtract Signed, Saturate, Integer to Accumulator Word
EVX	4	1226			253	SP	evsubfumiaaw	Vector Subtract Unsigned, Modulo, Integer to Accumulator Word
EVX	4	1218			253	SP	evsubfusiaaw	Vector Subtract Unsigned, Saturate, Integer to Accumulator Word
EVX	4	516			253	SP	evsubfw	Vector Subtract from Word
EVX	4	518			253	SP	evsubifw	Vector Subtract Immediate from Word
EVX	4	534			253	SP	evxor	Vector XOR
X	31	954	SR		74	B	extsb[.]	Extend Sign Byte
X	31	922	SR		74	B	extsh[.]	Extend Sign Halfword
X	31	986	SR		76	64	extsw[.]	Extend Sign Word
X	63	264			118	FP[R]	fabs[.]	Floating Absolute Value
A	63	21			119	FP[R]	fadd[.]	Floating Add
A	59	21			119	FP[R]	fadds[.]	Floating Add Single
X	63	846			127	FP[R]	fcfid[.]	Floating Convert From Integer Doubleword
X	63	32			129	FP	fcmpo	Floating Compare Ordered
X	63	0			129	FP	fcmpu	Floating Compare Unordered
X	63	814			125	FP[R]	fctid[.]	Floating Convert To Integer Doubleword

Form	Opcode Pri	Opcode Ext	Mode Dep.[1]	Priv[1]	Page	Cat[1]	Mnemonic	Instruction
X	63	815			126	FP[R]	fctidz[.]	Floating Convert To Integer Doubleword with round toward Zero
X	63	14			126	FP[R]	fctiw[.]	Floating Convert To Integer Word
X	63	15			127	FP[R]	fctiwz[.]	Floating Convert To Integer Word with round toward Zero
A	63	18			120	FP[R]	fdiv[.]	Floating Divide
A	59	18			120	FP[R]	fdivs[.]	Floating Divide Single
A	63	29			123	FP[R]	fmadd[.]	Floating Multiply-Add
A	59	29			123	FP[R]	fmadds[.]	Floating Multiply-Add Single
X	63	72			118	FP[R]	fmr[.]	Floating Move Register
A	63	28			123	FP[R]	fmsub[.]	Floating Multiply-Subtract
A	59	28			123	FP[R]	fmsubs[.]	Floating Multiply-Subtract Single
A	63	25			120	FP[R]	fmul[.]	Floating Multiply
A	59	25			120	FP[R]	fmuls[.]	Floating Multiply Single
X	63	136			118	FP[R]	fnabs[.]	Floating Negative Absolute Value
X	63	40			118	FP[R]	fneg[.]	Floating Negate
A	63	31			124	FP[R]	fnmadd[.]	Floating Negative Multiply-Add
A	59	31			124	FP[R]	fnmadds[.]	Floating Negative Multiply-Add Single
A	63	30			124	FP[R]	fnmsub[.]	Floating Negative Multiply-Subtract
A	59	30			124	FP[R]	fnmsubs[.]	Floating Negative Multiply-Subtract Single
A	63	24			121	FP[R]	fre[.]	Floating Reciprocal Estimate
A	59	24			121	FP[R]	fres[.]	Floating Reciprocal Estimate Single
X	63	488			128	FP[R]	frim[.]	Floating Round to Integer Minus
X	63	392			128	FP[R].in	frin[.]	Floating Round to Integer Nearest
X	63	456			128	FP[R].in	frip[.]	Floating Round to Integer Plus
X	63	424			128	FP[R].in	friz[.]	Floating Round to Integer Toward Zero
X	63	12			125	FP[R].in	frsp[.]	Floating Round to Single-Precision
A	63	26			122	FP[R].in	frsqrte[.]	Floating Reciprocal Square Root Estimate
A	59	26			122	FP[R].in	frsqrtes[.]	Floating Reciprocal Square Root Estimate Single
A	63	23			130	FP[R]	fsel[.]	Floating Select
A	63	22			121	FP[R]	fsqrt[.]	Floating Square Root
A	59	22			121	FP[R]	fsqrts[.]	Floating Square Root Single
A	63	20			119	FP[R]	fsub[.]	Floating Subtract
A	59	20			119	FP[R]	fsubs[.]	Floating Subtract Single
XL	19	274		H	405	S	hrfid	Hypervisor Return From Interrupt Doubleword
X	31	982			359	B	icbi	Instruction Cache Block Invalidate
X	31	991			536	E.PD	icbiep	Instruction Cache Block Invalidate by External PID
X	31	230			559	ECL	icblc	Instruction Cache Block Lock Clear
X	31	22			359	E	icbt	Instruction Cache Block Touch
X	31	486			558	ECL	icbtls	Instruction Cache Block Touch and Lock Set
X	31	966			629	E.CI	ici	Instruction Cache Invalidate
X	31	998			633	E.CD	icread	Instruction Cache Read
A	31	15			70	B.in	isel	Integer Select
XL	19	150			369	B	isync	Instruction Synchronize
X	31	95			529	E.PD	lbepx	Load Byte by External Process ID Indexed
D	34				41	B	lbz	Load Byte and Zero
D	35				41	B	lbzu	Load Byte and Zero with Update
X	31	119			41	B	lbzux	Load Byte and Zero with Update Indexed
X	31	87			42	B	lbzx	Load Byte and Zero Indexed
DS	58	0			46	64	ld	Load Doubleword
X	31	84			371	64	ldarx	Load Doubleword And Reserve Indexed
X	31	29			530	E.PD	ldepx	Load Doubleword by External Process ID Indexed
DS	58	1			46	64	ldu	Load Doubleword with Update
X	31	53			46	64	ldux	Load Doubleword with Update Indexed
X	31	21			46	64	ldx	Load Doubleword Indexed
D	50				113	FP	lfd	Load Floating-Point Double
X	31	607			537	E.PD	lfdepx	Load Floating-Point Double by External Process ID Indexed
D	51				113	FP	lfdu	Load Floating-Point Double with Update
X	31	631			113	FP	lfdux	Load Floating-Point Double with Update Indexed
X	31	599			113	FP	lfdx	Load Floating-Point Double Indexed

Form	Opcode Pri	Opcode Ext	Mode Dep.[1]	Priv[1]	Page	Cat[1]	Mnemonic	Instruction
D	48				115	FP	lfs	Load Floating-Point Single
D	49				115	FP	lfsu	Load Floating-Point Single with Update
X	31	567			115	FP	lfsux	Load Floating-Point Single with Update Indexed
X	31	535			115	FP	lfsx	Load Floating-Point Single Indexed
D	42				43	B	lha	Load Halfword Algebraic
D	43				43	B	lhau	Load Halfword Algebraic with Update
X	31	375			43	B	lhaux	Load Halfword Algebraic with Update Indexed
X	31	343			43	B	lhax	Load Halfword Algebraic Indexed
X	31	790			51	B	lhbrx	Load Halfword Byte-Reverse Indexed
X	31	287			529	E.PD	lhepx	Load Halfword by External Process ID Indexed
D	40				42	B	lhz	Load Halfword and Zero
D	41				42	B	lhzu	Load Halfword and Zero with Update
X	31	311			42	B	lhzux	Load Halfword and Zero with Update Indexed
X	31	279			42	B	lhzx	Load Halfword and Zero Indexed
D	46				52	B	lmw	Load Multiple Word
DQ	56			P	410	LSQ	lq	Load Quadword
X	31	597			55	MA	lswi	Load String Word Immediate
X	31	533			55	MA	lswx	Load String Word Indexed
X	31	7			146	V	lvebx	Load Vector Element Byte Indexed
X	31	39			143	V	lvehx	Load Vector Element Halfword Indexed
X	31	295			539	E.PD	lvepx	Load Vector by External Process ID Indexed
X	31	263			539	E.PD	lvepxl	Load Vector by External Process ID Indexed LRU
X	31	71			143	V	lvewx	Load Vector Element Word Indexed
X	31	6			148	V	lvsl	Load Vector for Shift Left Indexed
X	31	38			148	V	lvsr	Load Vector for Shift Right Indexed
X	31	103			144	V	lvx	Load Vector Indexed
X	31	359			144	V	lvxl	Load Vector Indexed Last
DS	58	2			45	64	lwa	Load Word Algebraic
X	31	20			370	B	lwarx	Load Word And Reserve Indexed
X	31	373			45	64	lwaux	Load Word Algebraic with Update Indexed
X	31	341			45	64	lwax	Load Word Algebraic Indexed
X	31	534			51	B	lwbrx	Load Word Byte-Reverse Indexed
X	31	31			530	E.PD	lwepx	Load Word by External Process ID Indexed
D	32				44	B	lwz	Load Word and Zero
D	33				44	B	lwzu	Load Word and Zero with Update
X	31	55			44	B	lwzux	Load Word and Zero with Update Indexed
X	31	23			44	B	lwzx	Load Word and Zero Indexed
XO	4	172			289	LMA	macchw[o][.]	Multiply Accumulate Cross Halfword to Word Modulo Signed
XO	4	236			289	LMA	macchws[o][.]	Multiply Accumulate Cross Halfword to Word Saturate Signed
XO	4	204			290	LMA	macchwsu[o][.]	Multiply Accumulate Cross Halfword to Word Saturate Unsigned
XO	4	140			290	LMA	macchwu[o][.]	Multiply Accumulate Cross Halfword to Word Modulo Unsigned
XO	4	44			291	LMA	machhw[o][.]	Multiply Accumulate High Halfword to Word Modulo Signed
XO	4	108			291	LMA	machhws[o][.]	Multiply Accumulate High Halfword to Word Saturate Signed
XO	4	76			292	LMA	machhwsu[o][.]	Multiply Accumulate High Halfword to Word Saturate Unsigned
XO	4	12			292	LMA	machhwu[o][.]	Multiply Accumulate High Halfword to Word Modulo Unsigned
XO	4	428			293	LMA	maclhw[o][.]	Multiply Accumulate Low Halfword to Word Modulo Signed
XO	4	492			293	LMA	maclhws[o][.]	Multiply Accumulate Low Halfword to Word Saturate Signed
XO	4	460			294	LMA	maclhwsu[o][.]	Multiply Accumulate Low Halfword to Word Saturate Unsigned
XO	4	396			294	LMA	maclhwu[o][.]	Multiply Accumulate Low Halfword to Word Modulo Unsigned

Form	Opcode Pri	Opcode Ext	Mode Dep.[1]	Priv[1]	Page	Cat[1]	Mnemonic	Instruction
X	31	854			374	E	mbar	Memory Barrier
XL	19	0			34	B	mcrf	Move Condition Register Field
X	63	64			131	FP	mcrfs	Move to Condition Register from FPSCR
X	31	512			91	B	mcrxr	Move to Condition Register from XER
X	31	275			91	E	mfapidi	Move From APID Indirect
XFX	31	19			89	B	mfcr	Move From Condition Register
XFX	31	323		S	527	E	mfdcr	Move From Device Control Register
X	31	291			91	E	mfdcrux	Move From Device Control Register User-mode Indexed
X	31	259		P	527	E	mfdcrx	Move From Device Control Register Indexed
X	63	583			131	FP[R]	mffs[.]	Move From FPSCR
X	31	83		P	417, 527	B	mfmsr	Move From Machine State Register
XFX	31	19			90	B.in	mfocrf	Move From One Condition Register Field
XFX	31	334			658	E.PM	mfpmr	Move From Performance Monitor Register
XFX	31	339		O	88,378	B	mfspr	Move From Special Purpose Register
X	31	595	32	P	449	S	mfsr	Move From Segment Register
X	31	659	32	P	449	S	mfsrin	Move From Segment Register Indirect
XFX	31	371			378	S	mftb	Move From Time Base
VX	4	1540			199	V	mfvscr	Move From Vector Status and Control Register
X	31	238			623	E.PC	msgclr	Message Clear
X	31	206			623	E.PC	msgsnd	Message Send
XFX	31	144			89	B	mtcrf	Move To Condition Register Fields
XFX	31	451		P	526	E	mtdcr	Move To Device Control Register
X	31	419			91	E	mtdcrux	Move To Device Control Register User-mode Indexed
X	31	387		P	526	E	mtdcrx	Move To Device Control Register Indexed
X	63	70			132	FP[R]	mtfsb0[.]	Move To FPSCR Bit 0
X	63	38			132	FP[R]	mtfsb1[.]	Move To FPSCR Bit 1
XFL	63	711			131	FP[R]	mtfsf[.]	Move To FPSCR Fields
X	63	134			131	FP[R]	mtfsfi[.]	Move To FPSCR Field Immediate
X	31	146		P	527	E	mtmsr	Move To Machine State Register
X	31	146		P	415	S	mtmsr	Move To Machine State Register
X	31	178		P	416	S	mtmsrd	Move To Machine State Register Doubleword
XFX	31	144			90	B.in	mtocrf	Move To One Condition Register Field
XFX	31	462			658	E.PM	mtpmr	Move To Performance Monitor Register
XFX	31	467		O	87	B	mtspr	Move To Special Purpose Register
X	31	210	32	P	448	S	mtsr	Move To Segment Register
X	31	242	32	P	448	S	mtsrin	Move To Segment Register Indirect
VX	4	1604			199	V	mtvscr	Move To Vector Status and Control Register
X	4	168			294	LMA	mulchw[.]	Multiply Cross Halfword to Word Signed
X	4	136			294	LMA	mulchwu[.]	Multiply Cross Halfword to Word Unsigned
XO	31	73	SR		65	64	mulhd[.]	Multiply High Doubleword
XO	31	9	SR		65	64	mulhdu[.]	Multiply High Doubleword Unsigned
X	4	40			295	LMA	mulhhw[.]	Multiply High Halfword to Word Signed
X	4	8			295	LMA	mulhhwu[.]	Multiply High Halfword to Word Unsigned
XO	31	75	SR		63	B	mulhw[.]	Multiply High Word
XO	31	11	SR		63	B	mulhwu[.]	Multiply High Word Unsigned
XO	31	233	SR		65	64	mulld[o][.]	Multiply Low Doubleword
X	4	424			295	LMA	mullhw[.]	Multiply Low Halfword to Word Signed
X	4	392			295	LMA	mullhwu[.]	Multiply Low Halfword to Word Unsigned
D	7				63	B	mulli	Multiply Low Immediate
XO	31	235	SR		63	B	mullw[o][.]	Multiply Low Word
X	31	476	SR		73	B	nand[.]	NAND
XO	31	104	SR		62	B	neg[o][.]	Negate
XO	4	174			296	LMA	nmacchw[o][.]	Negative Multiply Accumulate Cross Halfword to Word Modulo Signed
XO	4	238			296	LMA	nmacchws[o][.]	Negative Multiply Accumulate Cross Halfword to Word Saturate Signed
XO	4	46			297	LMA	nmachhw[o][.]	Negative Multiply Accumulate High Halfword to Word Modulo Signed

Form	Opcode Pri	Opcode Ext	Mode Dep.[1]	Priv[1]	Page	Cat[1]	Mnemonic	Instruction
XO	4	110			297	LMA	nmachhws[o][.]	Negative Multiply Accumulate High Halfword to Word Saturate Signed
XO	4	430			298	LMA	nmaclhw[o][.]	Negative Multiply Accumulate Low Halfword to Word Modulo Signed
XO	4	494			298	LMA	nmaclhws[o][.]	Negative Multiply Accumulate Low Halfword to Word Saturate Signed
X	31	124	SR		74	B	nor[.]	NOR
X	31	444	SR		73	B	or[.]	OR
X	31	412	SR		74	B	orc[.]	OR with Complement
D	24				71	B	ori	OR Immediate
D	25				72	B	oris	OR Immediate Shifted
X	31	122			76	B.in	popcntb	Population Count Bytes
XL	19	51		P	516	E	rfci	Return From Critical Interrupt
X	19	39			516	E.ED	rfdi	Return From Debug Interrupt
XL	19	50		P	515	E	rfi	Return From Interrupt
XL	19	18		P	405	S	rfid	Return From Interrupt Doubleword
XL	19	38		P	516	E	rfmci	Return From Machine Check Interrupt
MDS	30	8	SR		81	64	rldcl[.]	Rotate Left Doubleword then Clear Left
MDS	30	9	SR		82	64	rldcr[.]	Rotate Left Doubleword then Clear Right
MD	30	2	SR		81	64	rldic[.]	Rotate Left Doubleword Immediate then Clear
MD	30	0	SR		79	64	rldicl[.]	Rotate Left Doubleword Immediate then Clear Left
MD	30	1	SR		80	64	rldicr[.]	Rotate Left Doubleword Immediate then Clear Right
MD	30	3	SR		82	64	rldimi[.]	Rotate Left Doubleword Immediate then Mask Insert
M	20		SR		79	B	rlwimi[.]	Rotate Left Word Immediate then Mask Insert
M	21		SR		77	B	rlwinm[.]	Rotate Left Word Immediate then AND with Mask
M	23		SR		78	B	rlwnm[.]	Rotate Left Word then AND with Mask
SC	17				35, 404, 515	B	sc	System Call
X	31	498		P	444	S	slbia	SLB Invalidate All
X	31	434		P	443	S	slbie	SLB Invalidate Entry
X	31	915		P	446	S	slbmfee	SLB Move From Entry ESID
X	31	851		P	446	S	slbmfev	SLB Move From Entry VSID
X	31	402		P	445	S	slbmte	SLB Move To Entry
X	31	27	SR		85	64	sld[.]	Shift Left Doubleword
X	31	24	SR		83	B	slw[.]	Shift Left Word
X	31	794	SR		85	64	srad[.]	Shift Right Algebraic Doubleword
XS	31	413	SR		85	64	sradi[.]	Shift Right Algebraic Doubleword Immediate
X	31	792	SR		84	B	sraw[.]	Shift Right Algebraic Word
X	31	824	SR		84	B	srawi[.]	Shift Right Algebraic Word Immediate
X	31	539	SR		85	64	srd[.]	Shift Right Doubleword
X	31	536	SR		83	B	srw[.]	Shift Right Word
D	38				47	B	stb	Store Byte
X	31	223			531	E.PD	stbepx	Store Byte by External Process ID Indexed
D	39				47	B	stbu	Store Byte with Update
X	31	247			47	B	stbux	Store Byte with Update Indexed
X	31	215			47	B	stbx	Store Byte Indexed
DS	62	0			50	64	std	Store Doubleword
X	31	214			371	64	stdcx.	Store Doubleword Conditional Indexed
X	31	157			532	E.PD	stdepx	Store Doubleword by External Process ID Indexed
DS	62	1			50	64	stdu	Store Doubleword with Update
X	31	181			50	64	stdux	Store Doubleword with Update Indexed
X	31	149			50	64	stdx	Store Doubleword Indexed
D	54				116	FP	stfd	Store Floating-Point Double
X	31	735			537	E.PD	stfdepx	Store Floating-Point Double by External Process ID Indexed
D	55				116	FP	stfdu	Store Floating-Point Double with Update
X	31	759			116	FP	stfdux	Store Floating-Point Double with Update Indexed
X	31	727			116	FP	stfdx	Store Floating-Point Double Indexed
X	31	983			117	FP	stfiwx	Store Floating-Point as Integer Word Indexed
D	52				115	FP	stfs	Store Floating-Point Single

Form	Opcode Pri	Opcode Ext	Mode Dep.[1]	Priv[1]	Page	Cat[1]	Mnemonic	Instruction
D	53				115	FP	stfsu	Store Floating-Point Single with Update
X	31	695			115	FP	stfsux	Store Floating-Point Single with Update Indexed
X	31	663			115	FP	stfsx	Store Floating-Point Single Indexed
D	44				48	B	sth	Store Halfword
X	31	918			51	B	sthbrx	Store Halfword Byte-Reverse Indexed
X	31	415			531	E.PD	sthepx	Store Halfword by External Process ID Indexed
D	45				48	B	sthu	Store Halfword with Update
X	31	439			48	B	sthux	Store Halfword with Update Indexed
X	31	407			48	B	sthx	Store Halfword Indexed
D	47				53	B	stmw	Store Multiple Word
DS	62	2		P	410	LSQ	stq	Store Quadword
X	31	725			56	MA	stswi	Store String Word Immediate
X	31	661			56	MA	stswx	Store String Word Indexed
X	31	135			146	V	stvebx	Store Vector Element Byte Indexed
X	31	167			146	V	stvehx	Store Vector Element Halfword Indexed
X	31	807			540	E.PD	stvepx	Store Vector by External Process ID Indexed
X	31	775			540	E.PD	stvepxl	Store Vector by External Process ID Indexed LRU
X	31	199			147	V	stvewx	Store Vector Element Word Indexed
X	31	231			144	V	stvx	Store Vector Indexed
X	31	487			147	V	stvxl	Store Vector Indexed Last
D	36				49	B	stw	Store Word
X	31	662			51	B	stwbrx	Store Word Byte-Reverse Indexed
X	31	150			370	B	stwcx.	Store Word Conditional Indexed
X	31	159			532	E.PD	stwepx	Store Word by External Process ID Indexed
D	37				49	B	stwu	Store Word with Update
X	31	183			49	B	stwux	Store Word with Update Indexed
X	31	151			49	B	stwx	Store Word Indexed
XO	31	40	SR		59	B	subf[o][.]	Subtract From
XO	31	8	SR		60	B	subfc[o][.]	Subtract From Carrying
XO	31	136	SR		61	B	subfe[o][.]	Subtract From Extended
D	8		SR		60	B	subfic	Subtract From Immediate Carrying
XO	31	232	SR		61	B	subfme[o][.]	Subtract From Minus One Extended
XO	31	200	SR		62	B	subfze[o][.]	Subtract From Zero Extended
X	31	598			372	B	sync	Synchronize
X	31	68			70	64	td	Trap Doubleword
D	2				70	64	tdi	Trap Doubleword Immediate
X	31	370		P	453	S	tlbia	TLB Invalidate All
X	31	306	64	H	450	S	tlbie	TLB Invalidate Entry
X	31	274	64	H	452	S	tlbiel	TLB Invalidate Entry Local
X	31	786		P	560, 649	E	tlbivax	TLB Invalidate Virtual Address Indexed
X	31	946		P	560, 650	E	tlbre	TLB Read Entry
X	31	914		P	561, 650	E	tlbsx	TLB Search Indexed
X	31	566		H	453, 561, 651	B	tlbsync	TLB Synchronize
X	31	978		P	562, 651	E	tlbwe	TLB Write Entry
X	31	4			69	B	tw	Trap Word
D	3				69	B	twi	Trap Word Immediate
VX	4	384			160	V	vaddcuw	Vector Add and Write Carry-Out Unsigned Word
VX	4	10			189	V	vaddfp	Vector Add Single-Precision
VX	4	768			160	V	vaddsbs	Vector Add Signed Byte Saturate
VX	4	832			160	V	vaddshs	Vector Add Signed Halfword Saturate
VX	4	896			160	V	vaddsws	Vector Add Signed Word Saturate
VX	4	0			161	V	vaddubm	Vector Add Unsigned Byte Modulo
VX	4	512			162	V	vaddubs	Vector Add Unsigned Byte Saturate
VX	4	64			161	V	vadduhm	Vector Add Unsigned Halfword Modulo
VX	4	576			162	V	vadduhs	Vector Add Unsigned Halfword Saturate

Form	Opcode Pri	Opcode Ext	Mode Dep.[1]	Priv[1]	Page	Cat[1]	Mnemonic	Instruction
VX	4	128			161	V	vadduwm	Vector Add Unsigned Word Modulo
VX	4	640			162	V	vadduws	Vector Add Unsigned Word Saturate
VX	4	1028			184	V	vand	Vector Logical AND
VX	4	1092			184	V	vandc	Vector Logical AND with Complement
VX	4	1282			175	V	vavgsb	Vector Average Signed Byte
VX	4	1346			175	V	vavgsh	Vector Average Signed Halfword
VX	4	1410			175	V	vavgsw	Vector Average Signed Word
VX	4	1026			176	V	vavgub	Vector Average Unsigned Byte
VX	4	1090			176	V	vavguh	Vector Average Unsigned Halfword
VX	4	1154			176	V	vavguw	Vector Average Unsigned Word
VX	4	842			193	V	vcfsx	Vector Convert From Signed Fixed-Point Word
VX	4	778			193	V	vcfux	Vector Convert From Unsigned Fixed-Point Word
VC	4	966			195	V	vcmpbfp[.]	Vector Compare Bounds Single-Precision
VC	4	198			195	V	vcmpeqfp[.]	Vector Compare Equal To Single-Precision
VC	4	6			181	V	vcmpequb[.]	Vector Compare Equal To Unsigned Byte
VC	4	70			181	V	vcmpequh[.]	Vector Compare Equal To Unsigned Halfword
VC	4	134			182	V	vcmpequw[.]	Vector Compare Equal To Unsigned Word
VC	4	454			196	V	vcmpgefp[.]	Vector Compare Greater Than or Equal To Single-Precision
VC	4	710			196	V	vcmpgtfp[.]	Vector Compare Greater Than Single-Precision
VC	4	774			182	V	vcmpgtsb[.]	Vector Compare Greater Than Signed Byte
VC	4	838			182	V	vcmpgtsh[.]	Vector Compare Greater Than Signed Halfword
VC	4	902			182	V	vcmpgtsw[.]	Vector Compare Greater Than Signed Word
VC	4	518			183	V	vcmpgtub[.]	Vector Compare Greater Than Unsigned Byte
VC	4	582			183	V	vcmpgtuh[.]	Vector Compare Greater Than Unsigned Halfword
VC	4	646			183	V	vcmpgtuw[.]	Vector Compare Greater Than Unsigned Word
VX	4	970			192	V	vctsxs	Vector Convert To Signed Fixed-Point Word Saturate
VX	4	906			192	V	vctuxs	Vector Convert To Unsigned Fixed-Point Word Saturate
VX	4	394			197	V	vexptefp	Vector 2 Raised to the Exponent Estimate Floating-Point
VX	4	458			197	V	vlogefp	Vector Log Base 2 Estimate Floating-Point
VA	4	46			190	V	vmaddfp	Vector Multiply-Add Single-Precision
VX	4	1034			191	V	vmaxfp	Vector Maximum Single-Precision
VX	4	258			177	V	vmaxsb	Vector Maximum Signed Byte
VX	4	322			177	V	vmaxsh	Vector Maximum Signed Halfword
VX	4	386			177	V	vmaxsw	Vector Maximum Signed Word
VX	4	2			178	V	vmaxub	Vector Maximum Unsigned Byte
VX	4	66			178	V	vmaxuh	Vector Maximum Unsigned Halfword
VX	4	130			178	V	vmaxuw	Vector Maximum Unsigned Word
VA	4	32			168	V	vmhaddshs	Vector Multiply-High-Add Signed Halfword Saturate
VA	4	33			168	V	vmhraddshs	Vector Multiply-High-Round-Add Signed Halfword Saturate
VX	4	1098			191	V	vminfp	Vector Minimum Single-Precision
VX	4	770			179	V	vminsb	Vector Minimum Signed Byte
VX	4	834			179	V	vminsh	Vector Minimum Signed Halfword
VX	4	898			179	V	vminsw	Vector Minimum Signed Word
VX	4	514			180	V	vminub	Vector Minimum Unsigned Byte
VX	4	578			180	V	vminuh	Vector Minimum Unsigned Halfword
VX	4	642			180	V	vminuw	Vector Minimum Unsigned Word
VA	4	34			169	V	vmladduhm	Vector Multiply-Low-Add Unsigned Halfword Modulo
VX	4	12			154	V	vmrghb	Vector Merge High Byte
VX	4	76			154	V	vmrghh	Vector Merge High Halfword
VX	4	140			154	V	vmrghw	Vector Merge High Word
VX	4	268			155	V	vmrglb	Vector Merge Low Byte
VX	4	332			155	V	vmrglh	Vector Merge Low Halfword
VX	4	396			155	V	vmrglw	Vector Merge Low Word
VA	4	37			170	V	vmsummbm	Vector Multiply-Sum Mixed Byte Modulo
VA	4	40			170	V	vmsumshm	Vector Multiply-Sum Signed Halfword Modulo
VA	4	41			171	V	vmsumshs	Vector Multiply-Sum Signed Halfword Saturate
VA	4	36			169	V	vmsumubm	Vector Multiply-Sum Unsigned Byte Modulo
VA	4	38			171	V	vmsumuhm	Vector Multiply-Sum Unsigned Halfword Modulo

Form	Opcode		Mode Dep.[1]	Priv[1]	Page	Cat[1]	Mnemonic	Instruction
	Pri	Ext						
VA	4	39			172	V	vmsumuhs	Vector Multiply-Sum Unsigned Halfword Saturate
VX	4	776			166	V	vmulesb	Vector Multiply Even Signed Byte
VX	4	840			166	V	vmulesh	Vector Multiply Even Signed Halfword
VX	4	520			166	V	vmuleub	Vector Multiply Even Unsigned Byte
VX	4	584			166	V	vmuleuh	Vector Multiply Even Unsigned Halfword
VX	4	264			167	V	vmulosb	Vector Multiply Odd Signed Byte
VX	4	328			167	V	vmulosh	Vector Multiply Odd Signed Halfword
VX	4	8			167	V	vmuloub	Vector Multiply Odd Unsigned Byte
VX	4	72			167	V	vmulouh	Vector Multiply Odd Unsigned Halfword
VA	4	47			190	V	vnmsubfp	Vector Negative Multiply-Subtract Single-Precision
VX	4	1284			184	V	vnor	Vector Logical NOR
VX	4	1156			184	V	vor	Vector Logical OR
VA	4	43			157	V	vperm	Vector Permute
VX	4	782			149	V	vpkpx	Vector Pack Pixel
VX	4	398			150	V	vpkshss	Vector Pack Signed Halfword Signed Saturate
VX	4	270			150	V	vpkshus	Vector Pack Signed Halfword Unsigned Saturate
VX	4	462			150	V	vpkswss	Vector Pack Signed Word Signed Saturate
VX	4	334			150	V	vpkswus	Vector Pack Signed Word Unsigned Saturate
VX	4	14			151	V	vpkuhum	Vector Pack Unsigned Halfword Unsigned Modulo
VX	4	142			151	V	vpkuhus	Vector Pack Unsigned Halfword Unsigned Saturate
VX	4	78			151	V	vpkuwum	Vector Pack Unsigned Word Unsigned Modulo
VX	4	206			151	V	vpkuwus	Vector Pack Unsigned Word Unsigned Saturate
VX	4	266			198	V	vrefp	Vector Reciprocal Estimate Single-Precision
VX	4	714			194	V	vrfim	Vector Round to Single-Precision Integer toward -Infinity
VX	4	522			194	V	vrfin	Vector Round to Single-Precision Integer Nearest
VX	4	650			194	V	vrfip	Vector Round to Single-Precision Integer toward +Infinity
VX	4	586			194	V	vrfiz	Vector Round to Single-Precision Integer toward Zero
VX	4	4			185	V	vrlb	Vector Rotate Left Byte
VX	4	68			185	V	vrlh	Vector Rotate Left Halfword
VX	4	132			185	V	vrlw	Vector Rotate Left Word
VX	4	330			198	V	vrsqrtefp	Vector Reciprocal Square Root Estimate Single-Precision
VA	4	42			157	V	vsel	Vector Select
VX	4	452			158	V	vsl	Vector Shift Left
VX	4	260			186	V	vslb	Vector Shift Left Byte
VA	4	44			158	V	vsldoi	Vector Shift Left Double by Octet Immediate
VX	4	324			186	V	vslh	Vector Shift Left Halfword
VX	4	1036			158	V	vslo	Vector Shift Left by Octet
VX	4	388			186	V	vslw	Vector Shift Left Word
VX	4	524			156	V	vspltb	Vector Splat Byte
VX	4	588			156	V	vsplth	Vector Splat Halfword
VX	4	780			156	V	vspltisb	Vector Splat Immediate Signed Byte
VX	4	844			156	V	vspltish	Vector Splat Immediate Signed Halfword
VX	4	908			156	V	vspltisw	Vector Splat Immediate Signed Word
VX	4	652			156	V	vspltw	Vector Splat Word
VX	4	708			159	V	vsr	Vector Shift Right
VX	4	772			188	V	vsrab	Vector Shift Right Algebraic Byte
VX	4	836			188	V	vsrah	Vector Shift Right Algebraic Halfword
VX	4	900			188	V	vsraw	Vector Shift Right Algebraic Word
VX	4	516			187	V	vsrb	Vector Shift Right Byte
VX	4	580			187	V	vsrh	Vector Shift Right Halfword
VX	4	1100			159	V	vsro	Vector Shift Right by Octet
VX	4	644			187	V	vsrw	Vector Shift Right Word
VX	4	1408			163	V	vsubcuw	Vector Subtract and Write Carry-Out Unsigned Word
VX	4	74			189	V	vsubfp	Vector Subtract Single-Precision
VX	4	1792			163	V	vsubsbs	Vector Subtract Signed Byte Saturate
VX	4	1856			163	V	vsubshs	Vector Subtract Signed Halfword Saturate
VX	4	1920			163	V	vsubsws	Vector Subtract Signed Word Saturate
VX	4	1024			164	V	vsububm	Vector Subtract Unsigned Byte Modulo

Form	Opcode		Mode Dep.[1]	Priv[1]	Page	Cat[1]	Mnemonic	Instruction
	Pri	Ext						
VX	4	1536			165	V	vsububs	Vector Subtract Unsigned Byte Saturate
VX	4	1088			164	V	vsubuhm	Vector Subtract Unsigned Halfword Modulo
VX	4	1600			164	V	vsubuhs	Vector Subtract Unsigned Halfword Saturate
VX	4	1152			164	V	vsubuwm	Vector Subtract Unsigned Word Modulo
VX	4	1664			165	V	vsubuws	Vector Subtract Unsigned Word Saturate
VX	4	1672			173	V	vsum2sws	Vector Sum across Half Signed Word Saturate
VX	4	1800			174	V	vsum4sbs	Vector Sum across Quarter Signed Byte Saturate
VX	4	1608			174	V	vsum4shs	Vector Sum across Quarter Signed Halfword Saturate
VX	4	1544			174	V	vsum4ubs	Vector Sum across Quarter Unsigned Byte Saturate
VX	4	1928			173	V	vsumsws	Vector Sum across Signed Word Saturate
VX	4	846			152	V	vupkhpx	Vector Unpack High Pixel
VX	4	526			152	V	vupkhsb	Vector Unpack High Signed Byte
VX	4	590			152	V	vupkhsh	Vector Unpack High Signed Halfword
VX	4	974			153	V	vupklpx	Vector Unpack Low Pixel
VX	4	654			153	V	vupklsb	Vector Unpack Low Signed Byte
VX	4	718			153	V	vupklsh	Vector Unpack Low Signed Halfword
VX	4	1220			184	V	vxor	Vector Logical XOR
X	31	62			375	WT	wait	Wait
X	31	131		S	528	E	wrtee	Write MSR External Enable
X	31	163		S	528	E	wrteei	Write MSR External Enable Immediate
X	31	316	SR		73	B	xor[.]	XOR
D	26				72	B	xori	XOR Immediate
D	27				72	B	xoris	XOR Immediate Shifted

[1] See the key to the mode dependency and privilege columns on page 839 and the key to the category column in Section 1.3.5 of Book I.

Mode Dependency and Privilege Abbreviations

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.

Key to Mode Dependency Column

Mode Dep.	Description
CT	If the instruction tests the Count Register, it tests the low-order 32 bits in 32-bit mode and all 64 bits in 64-bit mode.
SR	The setting of status registers (such as XER and CR0) is mode-dependent.
32	The instruction can be executed only in 32-bit mode.
64	The instruction can be executed only in 64-bit mode.

Key to Privilege Column

Priv.	Description
P	Denotes a privileged instruction.
O	Denotes an instruction that is treated as privileged or nonprivileged (or hypervisor, for ***mtspr***), depending on the SPR number.
H	Denotes an instruction that can be executed only in hypervisor state.

Index

D

E

F

G

H

I

N

O

P

Q

R

U

V

W

X

Z

Numerics

Last Page - End of Document