RM0004
Reference manual

Programmer's reference manual for Book E processors

Introduction

This reference manual gives an overview of Book E, a version of the PowerPC architecture intended for embedded processors. To ensure application level compatibility with the PowerPC architecture developed by Apple, IBM, and Freescale, Book E incorporates the user level resources defined in the user instruction set architecture (UISA), Book I, of the AIM architectural definition.

Contents

List of tables

List of figures

About this book

The primary objective of this reference is to provide a view of the programming model defined by Book E and the Book E implementation standards (EIS).

Book E is a PowerPC™ architecture definition for embedded processors that ensures binary compatibility with the user instruction set architecture (UISA) portion of the PowerPC architecture as it was jointly developed by Apple, IBM, and Motorola (now Freescale Semiconductor, Inc.).

This book should be used with the user documentation for individual implementations; such documents provide a high-level summary of the information that appears here, as well as implementation-specific features and implementation differences that are not described here.

This document distinguishes between the three levels of the architectural and implementation definition, as follows:

- The Book E architecture —Book E defines a set of user-level instructions and registers that are drawn from the UISA portion of the AIM definition of the PowerPC architecture. Book E also include numerous other supervisor-level registers and instructions as they were defined in the AIM version of the PowerPC architecture for the virtual environment architecture (VEA) and the operating environment architecture (OEA).
  Because Book E defines a much different model for operating system resources such as the MMU and interrupts, it defines many new registers and instructions.
- Book E implementation standards (EIS). In many cases, the Book E architecture definition provides a very general framework, leaving many higher-level details up to the implementation.
  To ensure consistency among its Book E implementations, working standards were defined, providing an additional layer of architecture between Book E and actual devices. This layer includes more specific definitions of Book E features as well as extensions to the architecture, typically in the form of auxiliary processing units (APUs), which define additional registers, instructions, and interrupts that provide specially targeted capabilities. Note that some APUs are implementation-specific and are available only on individual devices. The APUs described here are those that are implemented on multiple processors or families of processors.
  The EIS guarantees that if an APU is implemented, it conforms to the EIS architecture described here.

Information in this book is subject to change without notice, as described in the disclaimers on the title page of this book. As with any technical documentation, it is the readers' responsibility to be sure they are using the most recent version of the documentation.

Audience

It is assumed that the reader has the appropriate general knowledge regarding operating systems, microprocessor system design, and the basic principles of RISC processing to use the information in this manual.

Organization

Following is a summary and a brief description of the major sections of this manual:

- *Part I: Book E and Book E implementation standards*," describes the programming model defined by the PowerPC Book E architecture and the EIS. It consists of the following chapters:
  - *Chapter 1: Overview*," provides a general discussion of the programming, interrupt, cache, and memory management models as they are defined by Book E and the EIS.
  - *Chapter 2: Register model*," is useful for software engineers who need to understand the programming model in general and the functionality of each register.
  - *Chapter 3: Instruction model*," provides an overview of the addressing modes and a description of the instructions. Instructions are organized by function.
  - *Chapter 4: Interrupts and exceptions*," provides an overview of the Book E– and EIS-defined interrupts and exception conditions that can cause them.
  - *Chapter 5: Storage architecture*," describes the cache and MMU portions of the EIS.
  - *Chapter 6: Instruction set*," functions as a handbook for the instruction set. Instructions are sorted by mnemonic. Each instruction description includes the instruction formats and an individualized legend that provides such information as the level or levels of the architecture in which the instruction may be found and the privilege level of the instruction.
- *Part II: EIS-defined extensions to the Book E architecture*," describes the auxiliary procession units (APUs) defined by the EIS. It consists of the following chapters:
  - *Chapter 7: Auxiliary processing units (APUs)*," describes extensions to the Book E architecture defined by the EIS. These include the following:
    - *Chapter 7.1: Integer select APU*"
    - *Chapter 7.2: Performance monitor APU*"
    - *Chapter 7.3: Signal processing engine APU (SPE APU)*"
    - *Chapter 7.4: Embedded vector and scalar single-precision floating-point APUs (SPFP APUs)*"
    - *Chapter 7.5: Machine check APU*"
    - *Chapter 7.6: Debug APU*"
  - *Chapter 8: Storage-related APUs*." describes the following APUs defined by the storage architecture:
    - *Chapter 8.1: Cache line locking APU*"
    - *Chapter 8.2: Direct cache flush APU*"
    - *Chapter 8.3: Cache way partitioning APU*"

Subsequent chapters describe the VLE extension

- *Chapter 9: VLE introduction*"
- *Chapter 10: VLE storage addressing*"
- *Chapter 11: VLE compatibility with the EIS*"
- *Chapter 12: VLE instruction classes*"
- *Chapter 13: VLE instruction set*"
- *Chapter 14: VLE instruction index*"

- The following appendixes are included:
  - *Appendix A: Instruction set listings*," lists all instructions except those defined by the VLE extension instructions by both mnemonic and opcode, and includes a quick reference table with general information, such as the architecture level, privilege level, form, and whether the instruction is optional. VLE instruction opcodes are listed in *Section 13: VLE instruction set*."
  - *Appendix B: Simplified mnemonics for PowerPC instructions*," describes simplified mnemonics, which are provided for easier coding of assembly language programs. Simplified mnemonics are defined for the most frequently used forms of branch conditional, compare, trap, rotate and shift, and certain other instructions defined by the PowerPC™ architecture and by implementations of and extensions to the PowerPC architecture.
  - *Appendix C: Programming examples*," gives examples of how memory synchronization instructions can be used to emulate various synchronization primitives and to provide more complex forms of synchronization. It also describes multiple precision shifts.
  - *Appendix D: Guidelines for 32-bit book E*," provides guidelines used by 32-bit Book E implementations; a set of guidelines is also outlined for software developers. Application software written to these guidelines can be labeled 32-bit Book E applications and can be expected to execute properly on all implementations of Book E, both 32-bit and 64-bit implementations.
  - *Appendix E: Embedded floating-point results*," provides guidelines used by 32-bit Book E implementations; a set of guidelines is also outlined for software developers. Application software written to these guidelines can be labeled 32-bit Book E applications and can be expected to execute properly on all implementations of Book E, both 32-bit and 64-bit implementations.

This book includes a glossary and an index.

Suggested reading

This section lists additional reading that provides background for the information in this manual as well as general information about the architecture.

General information

The following documentation, published by Morgan-Kaufmann Publishers, 340 Pine Street, Sixth Floor, San Francisco, CA, provides useful information about the PowerPC architecture and computer architecture in general:

- *The PowerPC Architecture: A Specification for a New Family of RISC Processors*, Second Edition, by International Business Machines, Inc.

Related documentation

ST documentation is available from the sources listed on the back cover of this manual; the document order numbers are included in parentheses for ease in ordering:

- Reference manuals—These books (formerly called user's manuals) provide details about individual implementations and are intended for use with the EREF.
- Addenda/errata to reference manuals—Because some processors have follow-on parts an addendum is provided that describes the additional features and functionality changes. These addenda are intended for use with the corresponding reference manuals.
- Hardware specifications—Hardware specifications provide specific data regarding bus timing, signal behavior, and AC, DC, and thermal characteristics, as well as other design considerations.
- Technical summaries—Each device has a technical summary that provides an overview of its features. This document is roughly the equivalent to the overview (Chapter 1) of an implementation's reference manual.
- Application notes—These short documents address specific design issues useful to programmers and engineers working with ST processors.

Additional literature is published as new processors become available.

Conventions

This document uses the following notational conventions:

Table 1. Conventions

Convention	Description
cleared/set	When a bit takes the value zero, it is said to be cleared; when it takes a value of one, it is said to be set.
mnemonics	Instruction mnemonics are shown in lowercase bold.
italics	Italics indicate variable command parameters, for example, **bcctr***x*. Book titles in text are set in italics. Internal signals are set in italics, for example, $\overline{qual\ BG}$
0x	Prefix to denote hexadecimal number
0b	Prefix to denote binary number
rA, **r**B	Instruction syntax used to identify what is typically a source GPR
rD	Instruction syntax used to identify a destination GPR
frA, **fr**B, **fr**C	Instruction syntax used to identify a source FPR
frD	Instruction syntax used to identify a destination FPR
REG[FIELD]	Abbreviations for registers are shown in uppercase text. Specific bits, fields, or ranges appear in brackets. For example, MSR[LE] refers to the little-endian mode enable bit in the machine state register.
x	In some contexts, such as signal encodings, an unitalicized x indicates a don't care.
x	An italicized *x* indicates an alphanumeric variable.
n	An italicized *n* indicates an numeric variable.
¬	NOT logical operator
&	AND logical operator
|	OR logical operator
||	Concatenation operator; for example, 010 || 111 is the same as 010111
—	Indicates a reserved field in a register. Although these bits can be written to as ones

Additional conventions used with instruction encodings are described in *Table 195*.

Acronyms and abbreviations

Table 2 contains acronyms and abbreviations that are used in this document.

Table 2. Acronyms and abbreviated terms

Term	Meaning
CR	Condition register
CTR	Count register
DTLB	Data translation lookaside buffer
EA	Effective address
ECC	Error checking and correction
FPR	Floating-point register
FPU	Floating-point unit
GPR	General-purpose register
IEEE	Institute of Electrical and Electronics Engineers
ITLB	Instruction translation lookaside buffer
L2	Secondary cache
LIFO	Last-in-first-out
LR	Link register
LRU	Least recently used
LSB	Least-significant byte
lsb	Least-significant bit
MMU	Memory management unit
MSB	Most-significant byte
msb	Most-significant bit
MSR	Machine state register
NaN	Not a number
NIA	Next instruction address
No-op	No operation
OEA	Operating environment architecture
PTE	Page table entry
RISC	Reduced instruction set computing
RTL	Register transfer language
SIMM	Signed immediate value
SPR	Special-purpose register
TB	Time base register
TLB	Translation lookaside buffer

Table 2. Acronyms and abbreviated terms (continued)

Term	Meaning
UIMM	Unsigned immediate value
UISA	User instruction set architecture
VA	Virtual address
VEA	Virtual environment architecture
VLE	Variable length encoding
XER	Register used primarily for indicating conditions such as carries and overflows for integer operations

Terminology conventions

Table 3 lists certain terms used in this manual that differ from the architecture terminology conventions.

Table 3. Terminology conventions

The architecture specification	This manual
Extended mnemonics	Simplified mnemonics
Privileged mode (or privileged state)	Supervisor level
Problem mode (or problem state)	User level
Real address	Physical address
Relocation	Translation
Out-of-order memory accesses	Speculative memory accesses
Storage (locations)	Memory
Storage (the act of)	Access

Table 4 describes instruction field notation conventions used in this manual.

Table 4. Instruction field conventions

The architecture specification	Equivalent to:
BA, BB, BT	**crb**A, **crb**B, **crb**D (respectively)
BF, BFA	**crf**D, **crf**S (respectively)
D	d
DS	ds
FLM	FM
FRA, FRB, FRC, FRT, FRS	**fr**A, **fr**B, **fr**C, **fr**D, **fr**S (respectively)
FXM	CRM
RA, RB, RT, RS	**r**A, **r**B, **r**D, **r**S (respectively)
SI	SIMM
U	IMM
UI	UIMM
/, //, ///	0...0 (shaded)

Part I: Book E and Book E implementation standards

Part I describes the registers and instructions defined by the Book E architecture and by the Book E implementation standards (EIS). It contains the following chapters:

- *Chapter 1: Overview*," provides a general discussion of the programming, interrupt, cache, and memory management models as they are defined by Book E and the EIS.
- *Chapter 2: Register model*," is useful for software engineers who need to understand the programming model in general and the functionality of each register.
- *Chapter 3: Instruction model*," provides an overview of the addressing modes and a description of the instructions. Instructions are organized by function.
- *Chapter 4: Interrupts and exceptions*," provides an overview of the Book E– and EIS–defined interrupts and exception conditions that can cause them.
- *Chapter 5: Storage architecture*," describes the cache and MMU portions of the EIS.

1 Overview

This document describes the Book E version of the PowerPC™ architecture as it is further defined by the Book E implementation standards (EIS) and implemented on Book E cores.

This chapter includes overviews of the following:

- Features of the Book E version of the PowerPC architecture and implementation-details defined by the EIS
- The Book E and EIS programming model
- The Book E and EIS interrupt model
- The Book E and EIS memory management model
- Architectural compatibility and migration from the original version of the PowerPC architecture as defined by Apple, IBM, and Motorola (referred to as the AIM version of the PowerPC architecture)

1.1 Overview Book E and the Book E implementation standards (EIS)

Book E is a version of the PowerPC architecture intended for embedded processors. To ensure application-level compatibility with the PowerPC architecture developed by Apple, IBM and Freescale, Book E incorporates the user-level resources defined in the user instruction set architecture (UISA), Book I, of the AIM architectural definition.

Because operating systems for embedded processors have different needs than those for desktop systems, Book E defines more flexible interrupt and memory management models. Instead of the segmented memory model defined by the AIM architecture, Book E provides a page-based memory system that supports multiple variable-sized pages managed through translation lookaside buffers (TLBs). Interrupt offsets can be programmed through interrupt-specific interrupt vector offset registers (IVORs). Book E defines the interrupt vector prefix register (IVPR), which is programmed with a prefix value that is concatenated with the IVOR values to place the interrupt vector table anywhere in memory.

As a consequence, some resources defined by the AIM version of the architecture are no longer supported and new ones are provided. For example, segment and block address translation (BAT) registers are gone, and new instructions, registers, and interrupts have been defined for managing page translation and protection through TLBs.

Moreover, the Book E architecture allows greater flexibility. For example, Book E defines the TLB Write Entry (**tlbwe**) and TLB Read Entry (**tlbre**) instructions only very generally, leaving details of their execution and behavior up to the implementation. However, to ensure compatibility among Book E implementations, the Book E implementation standard (EIS) defines more specifically how these instructions work.

1.1.1 Auxiliary processing units (APUs)

Book E supports the use of auxiliary processing units (APUs), which allocate opcode and register space for extending the instruction set without affecting the instruction set defined by Book E. This facilitates the development of special-purpose resources that are useful to some embedded environments but impractical for others. Note that instructions from multiple APUs may be assigned the same opcode numbers of the allocated opcode space.

The EIS defines many APUs. These APUs are not required on all devices, but devices that implement them do so strictly following the EIS architectural definition. In addition, an implementation may also provide an APU that is not a part of the EIS.

APUs may consist of any combination of instructions, optional behavior of Book E–defined instructions, registers, register files, fields within Book E–defined registers, interrupts, or exception conditions within Book E–defined interrupts.

Chapter 7: Auxiliary processing units (APUs)," provides an overview of specific APUs.

1.2 Instruction set

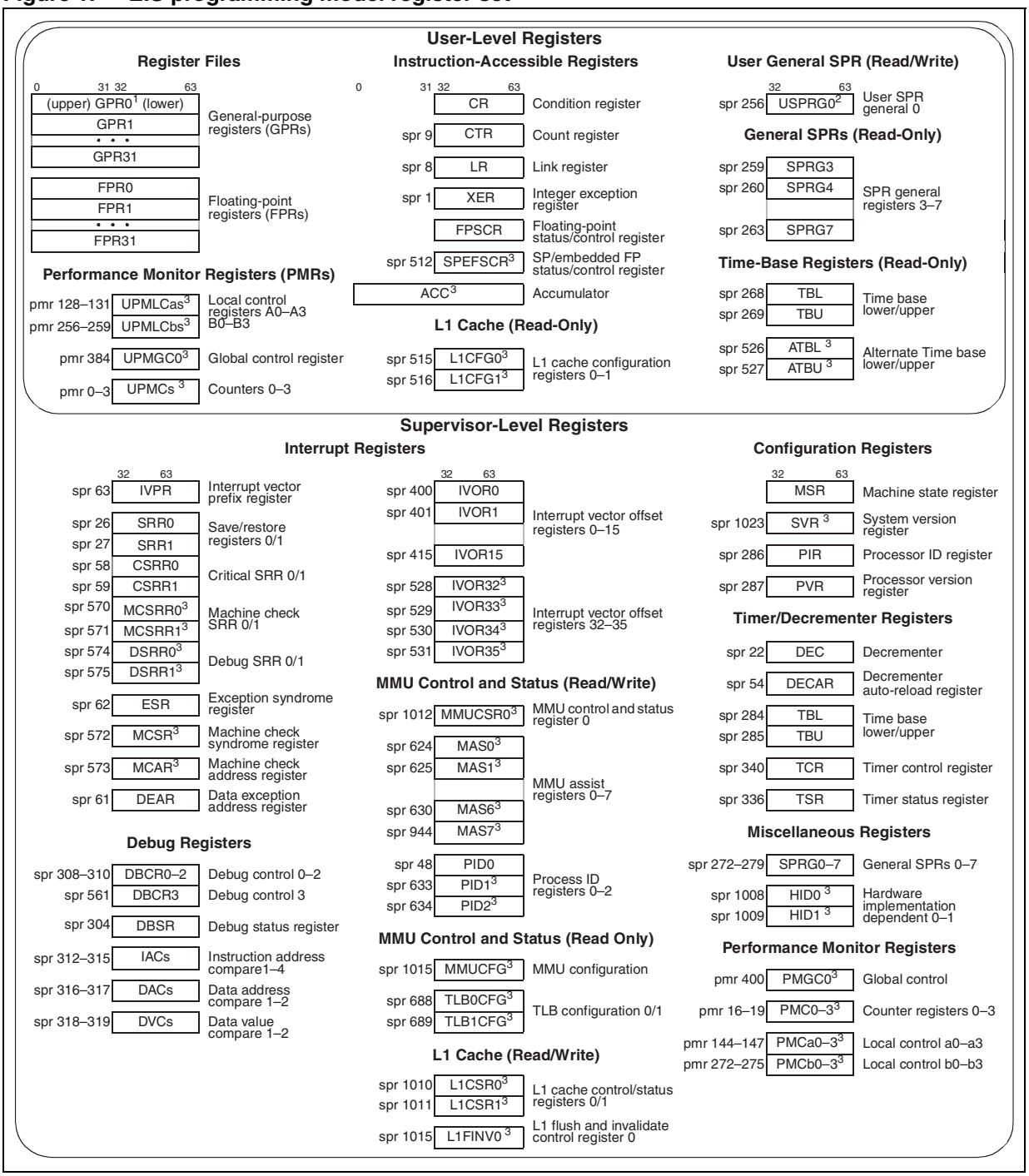
The instruction set of a ST 32-bit Book E–compliant device includes the following:

- The Book E instruction set for 32-bit implementations. This is composed primarily of the user-level instructions defined by the UISA. Some implementations do not include the Book E floating-point instructions or the Load String Word Indexed instruction (**lswx**).
- Instructions defined by EIS APUs. These include the following:
  - Integer select APU. This APU consists of the Integer Select instruction (**isel**), which incorporates an if-then-else statement that selects between two source registers by comparison to a CR bit. This instruction eliminates conditional branches, decreases band latency, and reduces the code footprint.
  - SPE (signal processing engine) APU instructions. SPE instructions treat 64-bit GPRs as a vector of two 32-bit elements (some instructions also read or write 16-bit elements). *Chapter 3.6.1: SPE and embedded floating-point APUs on page 186*," lists SPE APU vector instructions.
  - The embedded vector floating-point APU provides instructions that use the upper and lower words of the 64-bit GPRs for single-precision, vector floating-point calculations.
  - The embedded scalar single-precision APU provides instructions that use the lower 32 bits of the GPRs for single-precision, scalar floating-point calculations.
  - The embedded scalar double-precision APU instructions use the 64-bit GPRs for floating-point calculations.
  - Performance monitor APU—This APU defines two instructions, **mfpmr** and **mtpmr**, used for reading and writing the performance monitor registers (PMRs).
  - Cache block lock and unlock APU, consisting of the following instructions:
    - Data Cache Block Lock Clear (**dcblc**)
    - Data Cache Block Touch and Lock Set (**dcbtls**)
    - Data Cache Block Touch for Store and Lock Set (**dcbtstls**)
    - Instruction Cache Block Lock Clear (**icblc**)
    - Instruction Cache Block Touch and Lock Set (**icbtls**)

1.3 Register set

Note: Devices that implement a particular core may not implement all registers defined by that core.

Figure 1. EIS programming model register set



(1.) The 64-bit GPR registers are accessed by the SPE as separate 32-bit operands by SPE instructions. Only SPE vector instructions can access the upper word. (2.) USPRG0 is a separate physical register from SPRG0. (3.) EIS-defined registers; not part of the Book E architecture.

1.4 Interrupts and exception handling

Book E and the EIS support an extended exception handling model, with nested interrupt capability and extensive interrupt vector programmability. The following sections define the exception model, including an overview of exception handling as implemented in a ST Book E device, a brief description of the exception classes, and an overview of the registers involved.

1.4.1 Exception handling

In general, interrupt processing begins with an exception that occurs due to external conditions, errors, or program execution problems. When the exception occurs, the processor checks to verify that interrupt processing is enabled for that particular exception. If enabled, the interrupt causes the state of the processor to be saved in the appropriate registers, and prepares to begin execution of the handler located at the associated vector address for that particular exception.

Once the handler is executing, the implementation may need to check one or more bits in the exception syndrome register (ESR) or the SPEFSCR, depending on the exception type, to verify the specific cause of the exception and take appropriate action.

The interrupts are described in *Chapter 1.4.4: Interrupt registers*," and in *Table 6*.

1.4.2 Interrupt classes

All interrupts may be categorized as asynchronous/synchronous and critical/noncritical.

- Asynchronous interrupts are caused by events that are independent of instruction execution. The address reported in the save/restore register is that of the instruction that would have executed next had the asynchronous interrupt not occurred.
- Synchronous interrupts are caused directly by the execution or attempted execution of instructions. Synchronous inputs can be precise or imprecise:
  - Synchronous precise interrupts are those that precisely indicate the address of the instruction causing the exception that generated the interrupt or, in some cases, the address of the next instruction in program order. The interrupt type and status bits allow determination of which of the two instructions has been addressed in the appropriate save/restore register.
  - Synchronous imprecise interrupts may indicate the address of the instruction causing the exception that generated the interrupt or some instruction after the instruction causing the interrupt. If the interrupt was caused by either the context synchronizing mechanism or the execution synchronizing mechanism, the address in the appropriate save/restore register is the address of the interrupt forcing instruction. If the interrupt was not caused by either of those mechanisms, the address in the save/restore register is the last instruction to start execution and may not have completed. No instruction following the instruction in the save/restore register has executed.

1.4.3 Interrupt categories

Book E defines critical and noncritical interrupt categories, and the EIS defines the machine check and debug interrupt categories. Each category has a separate set of save and restore registers to which machine state and a return address are automatically written when an interrupt is taken. Each category has a return from interrupt instruction that uses the save and restore registers to reestablish the machine state of the interrupted process and

provides the address within that process at which to resume execution after the interrupt handler completes. Additional resources are provided for masking some of these interrupt categories, as described in the following:

- Debug APU interrupt (if present)—Although Book E defines debug as a critical interrupt, the EIS defines a separate debug APU. Debug save and restore registers (DSRR0/DSRR1) save state when a debug interrupt is taken; **rdci** restores state at the end of the interrupt handler. These interrupts are masked by setting the machine check enable bit, MSR[DE].
- Machine check APU interrupt (if present)—Although Book E defines machine check as a critical interrupt, the EIS defines a separate machine check APU. Machine check save and restore registers (MCSRR0/MCSRR1) save state when a machine check interrupt is taken; **rfmci** restores state at the end of the interrupt handler. These interrupts are masked by setting the machine check enable bit, MSR[ME].
- Noncritical interrupts—First-level interrupts that allow the processor to change program flow to handle conditions generated by external signals, errors, or unusual conditions arising from program execution or from programmable timer-related events. These interrupts are largely identical to those defined by the OEA portion of the Power PC architecture. They use save and restore registers (SRR0/SRR1) to save processor state and the **rfi** instruction to restore state. Asynchronous noncritical interrupts can be masked by the external interrupt enable bit, MSR[EE].
- Critical interrupts—Can be taken during a noncritical interrupt or during regular program flow. They use the critical save and restore registers (CSRR0/CSRR1) to save state when they are taken; they use the **rfci** instruction to restore state. These interrupts can be masked by the critical enable bit, MSR[CE]. Book E defines the critical input and watchdog timer interrupts as critical interrupts.

One interrupt of each category can be reported at a time; when it is taken, no program state is lost. Save/restore register pairs are serially reusable, so program state may be lost when an unordered interrupt is taken. See *Section 4.10: Interrupt ordering and masking*."

1.4.4 Interrupt registers

The registers associated with interrupt and exception handling are described in *Table 5*.

Table 5. Interrupt registers

Register	Description
Non critical interrupt registers	
SRR0	Save/restore register 0—Stores the address of the instruction causing the exception or the address of the instruction that will execute after the **rfi** instruction.
SRR1	Save/restore register 1—Saves machine state on noncritical interrupts and restores machine state after an **rfi** instruction is executed.
Critical interrupt registers	
CSRR0	Critical save/restore register 0—On critical interrupts, CSRR0 stores either the address of the instruction causing the exception or the address of the instruction that will execute after the **rfci** instruction.
CSRR1	Critical save/restore register 1—CSRR1 saves machine state on critical interrupts and restores machine state after an **rfci** instruction is executed.

Table 5. Interrupt registers (continued)

Register	Description
Machine check interrupt registers	
MCSRR0	Machine check save/restore register 0—Stores the address of the instruction that executes after **rfmci** executes.
MCSRR1	Machine check save/restore register 1—MCSRR1 stores machine state on machine check interrupts and restores machine state (if recoverable) after an **rfmci** instruction is executed.
MCAR	Machine check address register—MCAR holds the address of the data or instruction that caused the machine check interrupt. MCAR contents are not meaningful if a signal triggered the machine check interrupt.
Debug interrupt registers	
DSRR0	Debug save/restore register 0—Stores the address of the instruction that executes after **rfdi** executes.
DSRR1	Debug save/restore register 1—Stores machine state on machine check interrupts and restores machine state (if recoverable) after **rfmci** executes.
Syndrome registers	
MCSR	Machine check syndrome register—MCSR saves machine state information on machine check interrupts and restores machine state after an **rfmci** instruction is executed.
ESR	Exception syndrome register—ESR provides a syndrome to differentiate between the different kinds of exceptions that generate the same interrupt type. Upon generation of a specific exception type, the associated bit is set and all other bits are cleared.
SPE and embedded floating-point APU interrupt registers	
SPEFSCR	Signal processing and embedded floating-point status and control register—Provides interrupt control and status as well as various condition bits associated with the operations performed by the SPE APU and the embedded floating-point APUs.
Other interrupt registers	
DEAR	Data exception address register—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.
IVPR	Interrupt vector prefix register—IVPR[32–47] contains the high-order 16 bits of the address of the exception processing routines defined in the IVOR registers.
IVORs	Interrupt vector offset registers—The IVORs contain the low-order offset of the address of the exception processing routines defined in the IVOR registers. See *Table 6*.

Table 6 lists IVOR registers and associated interrupts.

Table 6. Interrupt vector registers and exception conditions

Register	Interrupt
Book E–defined IVORs	
IVOR0	Critical input
IVOR1	Machine check interrupt offset
IVOR2	Data storage interrupt offset
IVOR3	Instruction storage interrupt offset
IVOR4	External input interrupt offset
IVOR5	Alignment interrupt offset
IVOR6	Program interrupt offset
IVOR7	Floating-point unavailable interrupt offset
IVOR8	System call interrupt offset
IVOR9	Auxiliary processor unavailable interrupt offset
IVOR10	Decrementer interrupt offset
IVOR11	Fixed-interval timer interrupt offset
IVOR12	Watchdog timer interrupt offset
IVOR13	Data TLB error interrupt offset
IVOR14	Instruction TLB error interrupt offset
IVOR15	Debug interrupt offset
EIS-Defined IVORs	
IVOR32	SPE APU unavailable interrupt offset
IVOR33	Embedded floating-point data exception interrupt offset
IVOR34	Embedded floating-point round exception interrupt offset
IVOR35	Performance monitor interrupt offset

Each interrupt has an associated interrupt vector address, obtained by concatenating the IVPR and IVOR values (IVPR[32–47]||IVOR*n*[48–59]||0b0000). The resulting address is that of the instruction to be executed when that interrupt occurs. IVPR and IVOR values are indeterminate on reset, and must be initialized by the system software using **mtspr**. For more information, see *Chapter 4: Interrupts and exceptions*."

1.5 Memory management

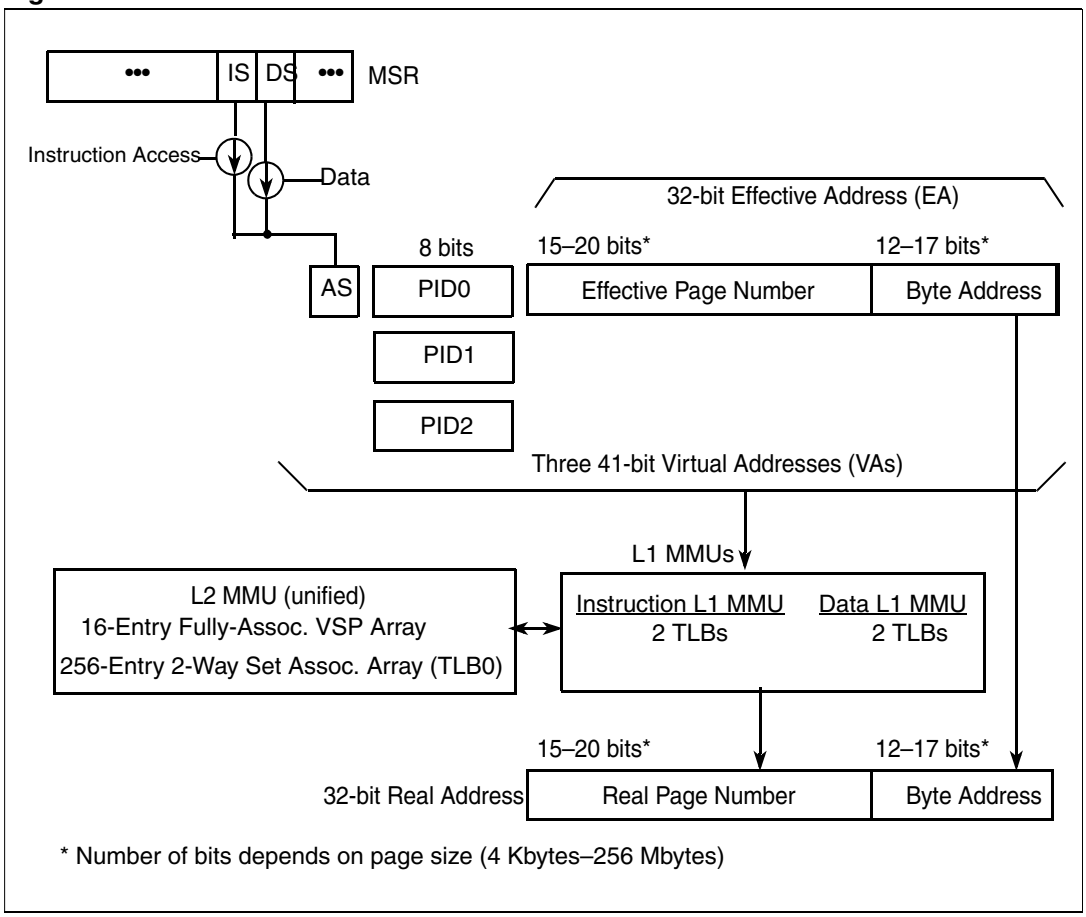
The EIS supports demand-paged virtual memory as well other memory management schemes that depend on precise control of effective-to-physical address translation and flexible memory protection as defined by Book E. The mapping mechanism consists of software-managed TLBs that support variable-sized pages with per-page properties and permissions. The following properties can be configured for each TLB:

- User mode page execute access
- User mode page read access
- User mode page write access
- Supervisor mode page execute access
- Supervisor mode page read access
- Supervisor mode page write access
- Write-through required (W)
- Caching inhibited (I)
- Memory coherence required (M)
- Guarded (G)
- Endianness (E)
- User-definable (U0–U3), a 4-bit implementation-specific field

1.5.1 Address translation

Figure 2 shows a typical translation flow, although each implementation may differ in the specific details. The MMU translates 32-bit effective addresses generated by loads, stores, and instruction fetches into 32-bit real addresses (used for memory bus accesses) using an interim 41-bit virtual address.

Figure 2. Effective-to-Real Address Translation Flow



As *Figure 2* shows, address translation starts with an effective address that is prepended with an address space (AS) value and a process ID to construct a virtual address (VA). The virtual address is then translated into a real address based on the translation information found in the on-chip TLB of the appropriate L1 MMU. The AS bit for the access is selected from the value of MSR[IS] or MSR[DS], for instruction or data accesses, respectively.

The appropriate L1 MMU (instruction or data) is checked for a matching address translation. The instruction L1 MMU and data L1 MMU operate independently and can be accessed in parallel, so that hits for instruction accesses and data accesses can occur in the same clock. If an L1 MMU misses, the request for translation is forwarded to the unified (instruction and data) L2 MMU. If found, the contents of the TLB entry are concatenated with the byte address to obtain the physical address of the requested access. On misses, the L1 TLB entries are replaced from their L2 TLB counterparts using a true LRU algorithm.

1.5.2 MMU assist registers (MAS1–MAS7)

Book E defines SPR numbers for the MMU assist registers, used to hold values either read from or to be written to the TLBs and information required to identify the TLB to be accessed. Book E leaves MAS register bit definitions up to the implementations. To ensure consistency among ST Book E processors, certain aspects of the implementation are defined by the Book E standard; more specific details are left to individual implementations. MAS3 implements the real page number (RPN), the user attribute bits (U0–U3), and

permission bits (UX, SX, UW, SW, UR, SR) that specify user and supervisor read, write, and execute permissions.

Some cores may not does not implement all of the MAS registers.

MAS registers are affected by the following instructions:

- MAS registers are accessed with the **mtspr** and **mfspr** instructions.
- The TLB Read Entry instruction (**tlbre**) causes the contents of a single TLB entry from the L2 MMU to be placed in defined locations in MAS0–MAS3. The TLB entry to be extracted is determined by information written to MAS0 and MAS2 before the **tlbre** instruction is executed.
- The TLB Write Entry instruction (**tlbwe**) causes the information stored in certain locations of MAS0–MAS3 to be written to the TLB specified in MAS0.
- The TLB Search Indexed instruction (**tlbsx**) updates MAS registers conditionally, based on success or failure of a lookup in the L2 MMU. The lookup is specified by the instruction encoding and specific search fields in MAS6. The values placed in the MAS registers may differ, depending on a successful or unsuccessful search.

For TLB miss and certain MMU-related DSI/ISI exceptions, MAS4 provides default values for updating MAS0–MAS2.

1.5.3 Process ID registers (PID0–PID2)

The Book E architecture identifies a single process ID register (PID). The EIS defines additional PIDs to hold values used to construct the virtual addresses for each access. Among these PIDs, PID0 is the Book E–defined PID. These process IDs provide an extended page sharing capability. Which of these three virtual addresses is used for translation is controlled by the TID field of a matching TLB entry, and when TID = 0x00 (identifying a page as globally shared), the PID values are ignored.

A hit to multiple TLB entries in the L1 MMU (even if they are in separate arrays) or a hit to multiple entries in the L2 MMU is considered to be a programming error.

1.5.4 TLB coherency

TLB entries can be invalidated as defined in the Book E architecture. The **tlbivax** instruction invalidates a matching local TLB entry.

1.5.5 Atomic update memory references

Book E supports atomic update memory references for both aligned word forms of data using the load and reserve and store conditional instruction pair, **lwarx** and **stwcx.**. Typically, a load and reserve instruction establishes a reservation and is paired with a store conditional instruction to achieve the atomic operation. However, the programmer is responsible for preserving reservations across context switches and for protecting reservations in multiprocessor implementations.

1.5.6 Memory access ordering

To optimize performance, Book E supports weakly ordered references to memory. Thus, a processor manages the order and synchronization of instructions to ensure proper execution when memory is shared between multiple processes or programs. The cache and data memory control attributes, along with **msync** and **mbar**, provide the required access

control; **msync** and **mbar** are also broadcast to provide the appropriate control in the case of multiprocessor or shared memory systems.

1.5.7 Cache control instructions

Book E cache control instructions perform a full range of cache control functions, including cache locking by line. The EIS defines the following cache locking instructions:

- Data Cache Block Lock Clear (**dcblc**)
- Data Cache Block Touch and Lock Set (**dcbtls**)
- Data Cache Block Touch for Store and Lock Set (**dcbtstls**)
- Instruction Cache Block Lock Clear (**icblc**)
- Instruction Cache Block Touch and Lock Set (**icbtls**)

1.5.8 Programmable page characteristics

Cache and memory attributes are programmable on a per-page basis. In addition to the write-through, caching-inhibited, memory coherency enforce, and guarded characteristics defined by the WIMG bits, Book E defines an endianness bit, E, that selects big- or little-endian byte ordering on a per-page basis.

1.6 Performance monitoring

The EIS provides a performance monitoring capability that supports counting of events such as processor clocks, instruction cache misses, data cache misses, mispredicted branches, and others. The count of these events may be configured to trigger a performance monitor exception. This interrupt is assigned to vector offset register IVOR35.

The register set associated with performance monitoring consists of counter registers, a global control register, and local control registers. These registers are read/write from supervisor mode, and each register is reflected to a corresponding read-only register for user mode. The **mtpmr** and **mfpmr** instructions move data to and from these registers. An overview of the performance monitoring registers is provided in the following sections. For more information, see *Chapter 7.2: Performance monitor APU*."

1.6.1 Global control register

The performance monitor global control register 0 (PMGC0) provides global control of the performance monitor from supervisor mode. From this register all counters may be frozen, unfrozen, or configured to freeze on an enabled condition or event. Additionally, the performance monitoring facility may be disabled or enabled from this register. The PMGC0 contents are reflected to UPMGC0, which may be read from user mode using **mfpmr**.

1.6.2 Performance monitor counter registers

There are four counter registers (PCM0–PCM3) provided in the performance monitor facility. These 32-bit registers hold the current count for software-selectable events and can be programmed to generate an exception on overflow. They can be accessed from supervisor mode using **mtpmr** and **mfpmr**. Their contents are reflected to UPCM0–UPCM3, which can be read from user mode with **mfpmr**.

The exception generated on overflow can be masked by clearing MSR[EE].

1.6.3 Local control registers

For each counter register, there are two corresponding local control registers. These two registers specify which of the 128 available events is to be counted, the action to be taken on overflow, and options for freezing a counter value under given modes or conditions.

- PMLCa0–PMLCa3 provide fields that allow freezing of the corresponding counter in user mode, supervisor mode, or under software control. The overflow condition may be enabled or disabled from these registers. Register contents are reflected to UPMCLa0–UPMLCa3, which can be read from user mode with **mfpmr**.
- PMLCb0–PMLCb3 provide count scaling for each counter register using configurable threshold and multiplier values. The threshold is a 6-bit value and the multiplier is a 3-bit encoded value, allowing 8 multiplier values in the range of 1 to 128. Any counter may be configured to increment only when an event occurs more than [threshold × multiplier] times. The contents of these registers are reflected to UPMCLb0–UPMLCb3, which can be read from user mode with **mfpmr**.

1.7 Legacy support of PowerPC architecture

In general, ST Book E processors support the user-level portion of the AIM architecture. The following subsections highlight the main differences. For specific details, refer to the relevant chapter.

1.7.1 Instruction set compatibility

The following sections generally describe compatibility between Book E and AIM PowerPC instruction sets.

User instruction set

The user mode instruction set defined by the AIM version of the PowerPC architecture is compatible with ST Book E processors with the following exceptions:

- Floating-point functionality provided by the embedded floating-point APUs differs from the AIM defined floating-point ISA. Also, the vector and double-precision floating-point APUs use 64-bit GPRs rather than the FPRs defined by the UISA. Most porting of floating-point operations can be handled by recompiling; however, there are new instructions specific to the APUs.
- String instructions are typically not implemented; therefore, trap emulation must be provided to ensure backward compatibility.

Supervisor instruction set

The supervisor mode instruction set defined by the AIM version of the PowerPC architecture is compatible with the EIS with the following exceptions:

- The MMU architecture is different, so some TLB manipulation instructions have different semantics.
- Instructions that support the BATs and segment registers are not implemented.
- Interrupt vectors are defined by the Book E IVOR*n* and IVPR SPRs.
- Additional instructions are defined for returning from Book E–defined critical interrupts (**rfci**) and APU-specific interrupts.

1.7.2 Memory subsystem

Both Book E and the AIM version of the PowerPC architecture provide separate instruction and data memory resources. The EIS provides additional cache control features, including cache locking.

1.7.3 Interrupt handling

Interrupt handling is generally the same as that defined in the AIM version of the PowerPC architecture, with the following differences: (see *Chapter 1.4*)

- Book E defines a new critical interrupt, providing an extra level of interrupt nesting. The critical interrupt includes external critical and watchdog timer time-out inputs.
- The machine check APU implements the machine check exception differently from the Book E and from the AIM definition. It defines the Return from Machine Check Interrupt instruction, **rfmci**, and two machine check save/restore registers, MCSRR0 and MCSRR1.
- Book E processors can use IVPR and IVORs to set exception vectors individually. To provide compatibility, they can be set to the address offsets defined in the OEA.
- Unlike the AIM version of the PowerPC architecture, Book E does not define a reset vector; execution begins at a fixed virtual address, 0xFFFF_FFFC.
- Some SPRs are different from those defined in the AIM version of the PowerPC architecture, particularly those related to the MMU functions. Much of this information has been moved to a new exception syndrome register (ESR).
- Timer services are generally compatible, although Book E defines a new decrementer auto reload feature and the fixed-interval timer critical interrupt.

1.7.4 Memory management

ST Book E processors implement a straightforward virtual address space that complies with the Book E MMU definition, which eliminates segment registers and block address translation resources. Book E defines resources for fixed 4-Kbyte pages and multiple, variable page sizes that can be configured in a single implementation. TLB management is provided with new instructions and SPRs.

1.7.5 Requirements for system reset generation

Book E does not specify a system reset interrupt as was defined in the AIM version of the PowerPC architecture, but typically, system reset is initiated either by asserting a signal or by software (for example, writing a 1 to DBCR0[34], if MSR[DE] = 1

At reset, instead of invoking a reset interrupt, fetching at address 0xFFFF_FFFC, as defined by Book E. In addition to the Book E reset definition, the EIS and the implementation define specific aspects of MMU page translation and protection mechanisms. Unlike the AIM version of the PowerPC core, as soon as instruction fetching begins, the core is in virtual mode with a hardware-initialized TLB entry.

1.7.6 Little-endian mode

Unlike the AIM version of the PowerPC, where the little-endian mode is controlled on a system basis, Book E supports control of byte ordering on a memory page basis. Additionally, true little-endian mode is supported by byte swapping.

2 Register model

This chapter describes the register model and indicates the architecture level at which each register is defined.

2.1 Overview

Although this chapter organizes registers according to their functionality, they can be differentiated according to how they are accessed, as follows:

- Register files. These user-level registers are accessed explicitly through source and destination operands of computational, load/store, logical, and other instructions. Book E defines two types of register files:
  - General-purpose registers (GPRs), used as source and destination operands for most operations (except Book E–defined floating-point instructions, which use FPRs). See *Chapter 2.3.1: General purpose registers (GPRs)*."
  - Floating-point registers (FPRs), used for Book E–defined floating-point instructions. See *Chapter 2.4.1: Floating-point registers (FPRs)*."
- Special-purpose registers (SPRs)—SPRs are accessed by using the Book E–defined Move to Special-Purpose Register (**mtspr**) and Move from Special-Purpose Register (**mfspr**) instructions. *Chapter 2.2.1: Special-purpose registers (SPRs)*," lists SPRs.
- System-level registers that are not SPRs. These are as follows:
  - Machine state register (MSR). MSR is accessed with the Move to Machine State Register (**mtmsr**) and Move from Machine State Register (**mfmsr**) instructions. See *Chapter 2.6.1: Machine state register (MSR)*."
  - Condition register (CR) bits are grouped into eight 4-bit fields, CR0–CR7, which are set as follows (see *Chapter 2.5.1: Condition register (CR)*"):

    - Specified CR fields can be set by a move to the CR from a GPR (**mtcrf**).

    - A specified CR field can be set by a move to the CR from another CR field (**mcrf**), from the FPSCR (**mcrfs**), or from the XER (**mcrxr**).

    - CR0 can be set as the implicit result of an integer instruction.

    - CR1 can be set as the implicit result of a floating-point instruction.

    - A specified CR field can be set as the result of an integer or floating-point compare instruction (including SPE and SPFP compare instructions).
  - The floating-point status and control register (FPSCR). See *Chapter 2.4.2: Floating-point status and control register (FPSCR)*."
  - The EIS-defined accumulator, which is accessed by signal processing engine (SPE) APU instructions that update the accumulator. See *Chapter 2.14.2: Accumulator (ACC)*."
- Device control registers (DCRs). Book E defines the existence of a DCR address space and the instructions to access them, but does not define particular DCRs. The on-chip DCRs exist architecturally outside the processor core and thus are not part of Book E. The contents of DCR DCRN can be read into a GPR using **mfdcr** r**D,**DCRN. GPR contents can be written into DCR DCRN using **mtdcr** DCRN**,**r**S**. See *Chapter 2.17: Device control registers (DCRs)*."
- Performance monitor registers (PMRs). (Performance monitor APU) Similar to SPRs, PMRs are accessed by using the EIS-defined Move to Performance Monitor Register

(**mtpmr**) and Move from Performance Monitor Register (**mfspr**) instructions. See *Chapter 2.16: Performance monitor registers (PMRs)*."

2.2 Register model for 32-bit Book E implementations

Book E implementations include the following types of software-accessible registers:

- Registers that are accessed as part of instruction execution. These include the following:
  - The following registers are used for integer operations and are described in *Chapter 2.3: Registers for integer operations*":

    - General-purpose registers (GPRs)—Book E defines a set of 32 GPRs used to hold source and destination operands for load, store, arithmetic, and computational instructions, and to read and write to other registers.

    - Integer exception register (XER)—XER bits are set based on the operation of an instruction considered as a whole, not on intermediate results. (For example, the Subtract from Carrying instruction (**subfc**), the result of which is specified as the sum of three values, sets bits in the XER based on the entire operation, not on an intermediate sum.)
  - Registers for floating-point operations. These include the following:

    - Floating-point registers (FPRs)—32 registers used to hold source and destination operands for Book E defined floating-point operations. Note that the embedded floating-point APUs do not implement FPRs; they use GPRs for floating-point operands.

    - Floating-point status and control register (FPSCR)—Used with floating-point operations. These registers are described in *Chapter 2.4: Registers for floating-point operations*."
  - Condition register (CR)—Used to record conditions such as overflows and carries that occur as a result of executing arithmetic instructions (including those implemented by the SPE and SPFP APUs). The CR is described in *Chapter 2.5: Registers for branch operations*."
  - Machine state register (MSR)—Used by the operating system to configure parameters such as user/supervisor mode, address space, and enabling of

asynchronous interrupts. MSR is described in *Chapter 2.6.1: Machine state register (MSR)*."

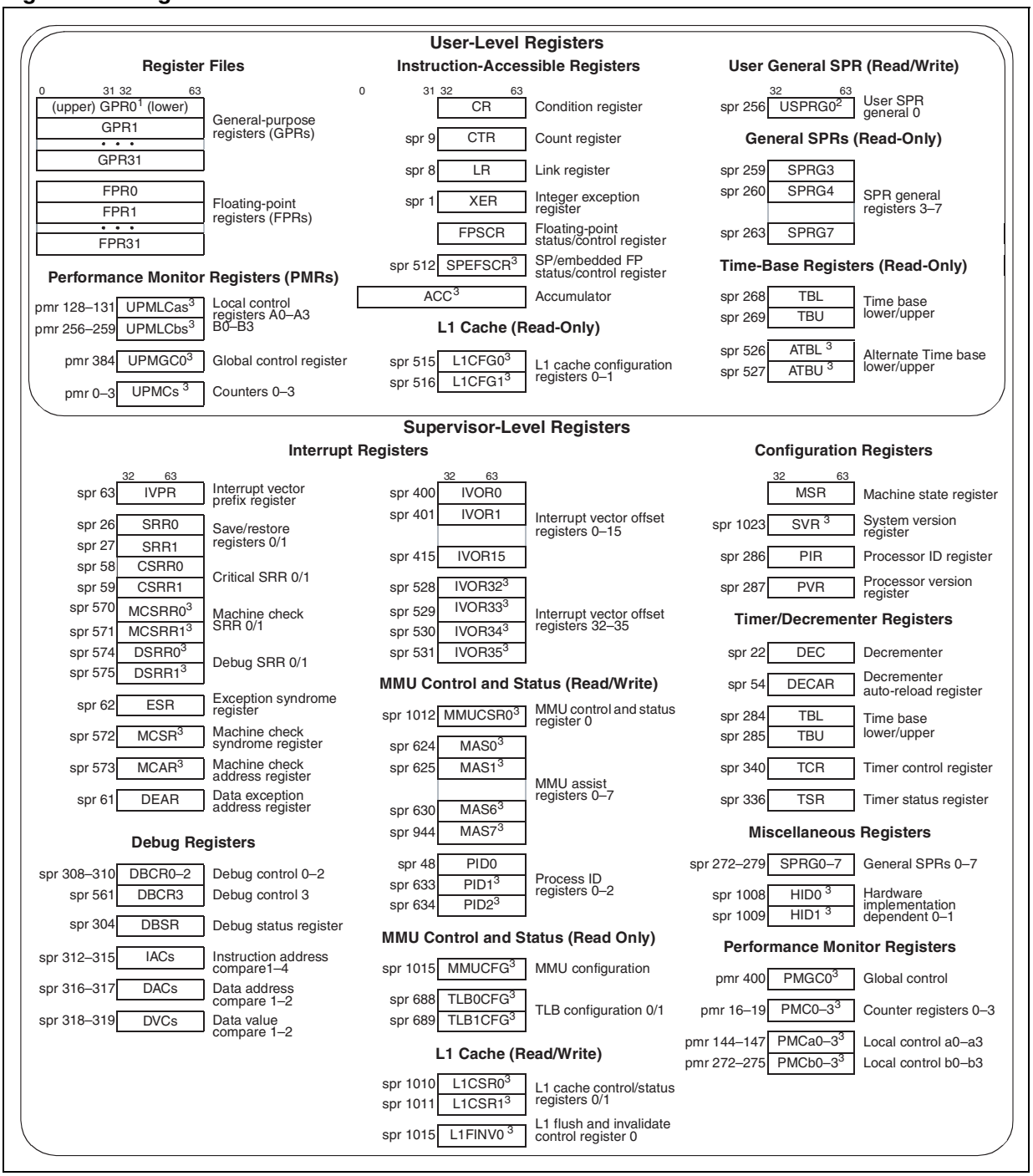
- Special-purpose registers (SPRs).
  - Book E–defined special-purpose registers (SPRs) that are accessed explicitly using **mtspr** and **mfspr** instructions. These registers are listed in *Table 7* in *Chapter 2.2.1: Special-purpose registers (SPRs)*."
  - EIS–defined SPRs that are accessed explicitly using the **mtspr** and **mfspr** instructions. These registers are listed in *Table 8* in *Chapter 2.2.1: Special-purpose registers (SPRs)*."
  - SPRs are described by function in the following sections:
    - *Chapter 2.5: Registers for branch operations*"
    - *Chapter 2.6: Processor control registers*"
    - *Chapter 2.7: Hardware implementation-dependent registers*"
    - *Chapter 2.8: Timer registers*"
    - *Chapter 2.9: Interrupt registers*"
    - *Chapter 2.10: Software use sprs (SPRG0–SPRG7 and USPRG0)*"
    - *Chapter 2.11: L1 cache registers*"
    - *Chapter 2.12: MMU registers*"
    - *Chapter 2.13: Debug registers*"
    - *Chapter 2.14: SPE and SPFP APU registers*"
    - *Chapter 2.15: Alternate time base registers (ATBL and ATBU)*"
- EIS-defined performance monitor registers, described in *Chapter 2.16: Performance monitor registers (PMRs)*." PMRs are like SPRs, but are accessed with EIS-defined move to and move from PMR instructions (**mtpmr** and **mfpmr**).
- EIS-defined device control registers (DCRs). Book E defines a format for implementing device-specific device-control registers. See *Chapter 2.17: Device control registers (DCRs)*."

Book E defines 32- and 64-bit registers. However, except for the 64-bit FPRs, only bits 32–63 of Book E's 64-bit registers (such as LR, CTR, the GPRs, SRR0, and CSRR0) are required to be implemented in hardware in a 32-bit Book E implementation.

Likewise, all Book E integer instructions defined to return a 64-bit result return only bits 32–63 of the result on a 32-bit Book E implementation. SPE APU vector instructions return 64-bit values; SPFP APU instructions return single-precision 32-bit values.

As with the instruction set and other aspects of the architecture, Book E defines some features very specifically, for example, resources that ensure compatibility with implementations of the PowerPC ISA. Other resources are either defined as optional or are defined in a very general way, leaving specific details up to the implementation.

Figure 3. Register model



(1.) The 64-bit GPR registers are accessed by the SPE as separate 32-bit operands by SPE instructions. Only SPE vector instructions can access the upper word.

(2.) USPRG0 is a separate physical register from SPRG0.

(3.) EIS-defined registers; not part of the Book E architecture.

2.2.1 Special-purpose registers (SPRs)

SPRs are on-chip registers that are architecturally part of the processor core. They control the use of the debug facilities, timers, interrupts, memory management unit, and other architected processor resources and are accessed with the **mtspr** and **mfspr** instructions. Unlisted encodings are reserved for future use.

Table 7 summarizes SPRs defined in Book E. The SPR numbers are used in the instruction mnemonics. Bit 5 in an SPR number indicates whether an SPR is accessible from user or supervisor software. An **mtspr** or **mfspr** instruction that specifies an unsupported SPR number is considered an invalid instruction. Invalid instructions are treated as follows:

- If the invalid SPR falls within the range specified as user mode (SPR[5] = 0), an illegal exception is taken.
- If supervisor software attempts to access an invalid supervisor-level SPR (SPR[5] = 1), results are undefined.
- If user software attempts to access an invalid supervisor-level SPR, a privilege exception is taken.

Table 7. Book E special purpose registers (by SPR abbreviation)

SPR Abbreviation	Name	Defined SPR number		Access	Supervisor only
		Decimal	Binary		
CSRR0	*Critical save/restore register 0 (CSRR0)*	58	00001 11010	Read/Write	Yes
CSRR1	*Critical save/restore register 1 (CSRR1)*	59	00001 11011	Read/Write	Yes
CTR	*Count register (CTR)*	9	00000 01001	Read/Write	No
DAC1	*Data address compare registers (DAC1–DAC2)*	316	01001 11100	Read/Write	Yes
DAC2	*Data address compare registers (DAC1–DAC2)*	317	01001 11101	Read/Write	Yes
DBCR0	*Debug control registers (DBCR0–DBCR3) 1*	308	01001 10100	Read/Write	Yes
DBCR1	*Debug control registers (DBCR0–DBCR3) 2*	309	01001 10101	Read/Write	Yes
DBCR2	*Debug control registers (DBCR0–DBCR3) 3*	310	01001 10110	Read/Write	Yes
DBSR	*Debug status register (DBSR)*	304	01001 10000	Read/Clear (1)	Yes
DEAR	*Data exception address register (DEAR)*	61	00001 11101	Read/Write	Yes
DEC	*Decrementer register*	22	00000 10110	Read/Write	Yes
DECAR	*Decrementer auto-reload register (DECAR)*	54	00001 10110	Write-only	Yes
DVC1	*Data value compare registers (DVC1 and DVC2) 1*	318	01001 11110	Read/Write	Yes

Table 7. Book E special purpose registers (by SPR abbreviation) (continued)

SPR Abbreviation	Name	Defined SPR number		Access	Supervisor only
		Decimal	Binary		
DVC2	*Data value compare registers (DVC1 and DVC2) 2*	319	01001 11111	Read/Write	Yes
ESR	*Exception syndrome register (ESR)*	62	00001 11110	Read/Write	Yes
IAC1 IAC2 IAC3 IAC4	*Instruction address compare registers (IAC1–IAC4)*	312 313 314 315	01001 11000 01001 11001 01001 11010 01001 11011	Read/Write	Yes
IVOR0	*Interrupt vector offset registers (IVORs)* Critical input	400	01100 10000	Read/Write	Yes
IVOR1	*Interrupt vector offset registers (IVORs)* Machine check interrupt offset	401	01100 10001	Read/Write	Yes
IVOR10	*Interrupt vector offset registers (IVORs)* Decrementer interrupt offset	410	01100 11010	Read/Write	Yes
IVOR11	*Interrupt vector offset registers (IVORs)* Fixed-interval timer interrupt offset	411	01100 11011	Read/Write	Yes
IVOR12	*Interrupt vector offset registers (IVORs)* Watchdog timer interrupt offset	412	01100 11100	Read/Write	Yes
IVOR13	*Interrupt vector offset registers (IVORs)* Data TLB error interrupt offset	413	01100 11101	Read/Write	Yes
IVOR14	*Interrupt vector offset registers (IVORs)* Instruction TLB error interrupt offset	414	01100 11110	Read/Write	Yes
IVOR15	*Interrupt vector offset registers (IVORs)* Debug interrupt offset	415	01100 11111	Read/Write	Yes
IVOR2	*Interrupt vector offset registers (IVORs)* Data storage interrupt offset	402	01100 10010	Read/Write	Yes
IVOR3	*Interrupt vector offset registers (IVORs)* Instruction storage interrupt offset	403	01100 10011	Read/Write	Yes

Table 7. Book E special purpose registers (by SPR abbreviation) (continued)

SPR Abbreviation	Name	Defined SPR number		Access	Supervisor only
		Decimal	Binary		
IVOR4	*Interrupt vector offset registers (IVORs)* External input interrupt offset	404	01100 10100	Read/Write	Yes
IVOR5	*Interrupt vector offset registers (IVORs)* Alignment interrupt offset	405	01100 10101	Read/Write	Yes
IVOR6	*Interrupt vector offset registers (IVORs)* Program interrupt offset	406	01100 10110	Read/Write	Yes
IVOR7	*Interrupt vector offset registers (IVORs)* Floating-point unavailable interrupt offset	407	01100 10111	Read/Write	Yes
IVOR8	*Interrupt vector offset registers (IVORs)* System call interrupt offset	408	01100 11000	Read/Write	Yes
IVOR9	*Interrupt vector offset registers (IVORs)* APU unavailable interrupt offset	409	01100 11001	Read/Write	Yes
IVPR	*Interrupt vector offset registers (IVORs)* Interrupt vector	63	00001 11111	Read/Write	Yes
LR	*Link register (LR)*	8	00000 01000	Read/Write	No
PID	*Process ID registers (PID0–PIDn)*	48	00001 10000	Read/Write	Yes
PIR	*Processor ID register (PIR)*	286	01000 11110	Read-only	Yes
PVR	*Processor version register (PVR)*	287	01000 11111	Read-only	Yes
SPRG0 SPRG1 SPRG2 SPRG3 SPRG4 SPRG5 SPRG6 SPRG7	*Software use sprs (SPRG0–SPRG7 and USPRG0)*	272 273 274 275 276 277 278 279	01000 10000 01000 10001 01000 10010 01000 10011 01000 10100 01000 10101 01000 10110 01000 10111	Read/Write	Yes
SRR0	*Save/restore register 0 (SRR0)*	26	00000 11010	Read/Write	Yes
SRR1	*Save/restore register 1 (SRR1)*	27	00000 11011	Read/Write	Yes
TBL TBU	*Time base (TBU and TBL)*	284 285	01000 11100 01000 11101	Write-only	Yes

Table 7. Book E special purpose registers (by SPR abbreviation) (continued)

SPR Abbreviation	Name	Defined SPR number		Access	Supervisor only
		Decimal	Binary		
TCR	*Timer control register (TCR)*	340	01010 10100	Read/Write	Yes
TSR	*Timer status register (TSR)*	336	01010 10000	Read/Clear (2)	Yes
USPRG0 USPRG3 USPRG4 USPRG5 USPRG6 USPRG7	*Software use sprs (SPRG0–SPRG7 and USPRG0)*(3)	256 259 260 261 262 263	01000 00000 01000 00011 01000 00100 01000 00101 01000 00110 01000 00111	Read/Write Read-only Read-only Read-only Read-only Read-only	No
UTBL	*Time base (TBU and TBL)*	268	01000 01100	Read-only	No
UTBU	*Time base (TBU and TBL)*	269	01000 01101	Read-only	No
XER	*Integer exception register (XER)*	1	00000 00001	Read/Write	No

1. The DBSR is read using **mfspr**. It cannot be directly written to. Instead, DBSR bits corresponding to 1 bits in the GPR can be cleared using **mtspr**.
2. The TSR is read using **mfspr**. It cannot be directly written to. Instead, TSR bits corresponding to 1 bits in the GPR can be cleared using **mtspr**.
3. User-mode read access to SPRG3 is implementation-dependent

Table 8 lists EIS-defined SPRs. Compilers should recognize the mnemonic name given in this table when parsing instructions.

Table 8. EIS–defined SPRs (by SPR abbreviation)

SPR abbreviation	Name	SPR number	Access	Supervisor only	Section/page
ATBL	Alternate time base lower	526	Read-only	No	*Section 2.15 on page 123*
ATBU	Alternate time base upper	527	Read-only	No	*Section 2.15 on page 123*
DBCR3	Debug control register 3	561	Read/Write	Yes	*on page 115*
DSRR0	Debug save/restore register 0	574	R/W	Yes	*on page 86*
DSRR1	Debug save/restore register 1	575	R/W	Yes	*on page 87*
HID0	Hardware implementation dependent register 0	1008	Read/Write	Yes	*Section 2.7.1 on page 71*
HID1	Hardware implementation dependent register 1	1009	Read/Write	Yes	*Section 2.7.2 on page 74*
IVOR32	SPE/embedded floating-point APU unavailable interrupt offset	528	Read/Write	Yes	*on page 83*
IVOR33	Embedded floating-point data exception interrupt offset	529	Read/Write	Yes	*on page 83*
IVOR34	Embedded floating-point round exception interrupt offset	530	Read/Write	Yes	*on page 83*

Table 8. EIS–defined SPRs (by SPR abbreviation) (continued)

SPR abbreviation	Name	SPR number	Access	Supervisor only	Section/page
IVOR35	Performance monitor	531	Read/Write	Yes	*on page 83*
IVOR36	Processor doorbell interrupt. Defined by processor signalling APU.	532	Read/Write	Yes	*on page 83*
IVOR37	Processor doorbell critical interrupt. Defined by processor signalling APU.	533	Read/Write	Yes	*on page 83*
L1CFG0	L1 cache configuration register 0	515	Read-only	No	*Section 2.11.1 on page 90*
L1CFG1	L1 cache configuration register 1	516	Read-only	No	*Section 2.11.2 on page 92*
L1CSR0	L1 cache control and status register 0	1010	Read/Write	Yes	*Section 2.11.1 on page 90*
L1CSR1	L1 cache control and status register 1	1011	Read/Write	Yes	*Section 2.11.2 on page 92*
L1FINV0	L1 flush and invalidate control register 0	1016	Read/Write	Yes	*Section 2.11.5 on page 96*
MAS0	MMU assist register 0	624	Read/Write	Yes	*Section 2.12.5 on page 101*
MAS1	MMU assist register 1	625	Read/Write	Yes	*Section 2.12.5 on page 101*
MAS2	MMU assist register 2	626	Read/Write	Yes	*Section 2.12.5 on page 101*
MAS3	MMU assist register 3	627	Read/Write	Yes	*Section 2.12.5 on page 101*
MAS4	MMU assist register 4	628	Read/Write	Yes	*Section 2.12.5 on page 101*
MAS5	MMU assist register 5.	629	Read/Write	Yes	*Section 2.12.5 on page 101*
MAS6	MMU assist register 6	630	Read/Write	Yes	*Section 2.12.5 on page 101*
MAS7	MMU assist register 7	944	Read/Write	Yes	*Section 2.12.5 on page 101*
MCAR	Machine check address register	573	Read-only	Yes	*on page 88*
MCARU	Machine check address register upper	569	Read-only	Yes	*on page 88*
MCSR	Machine check syndrome register	572	Read/Write	Yes	*on page 88*
MCSRR0	Machine-check save/restore register 0	570	Read/Write	Yes	*on page 87*
MCSRR1	Machine-check save/restore register 1	571	Read/Write	Yes	*on page 87*
MMUCFG	MMU configuration register	1015	Read-only	Yes	*Section 2.12.3 on page 99*
MMUCSR0	MMU control and status register 0	1012	Read/Write	Yes	*Section 2.12.2 on page 98*

Table 8. EIS–defined SPRs (by SPR abbreviation) (continued)

SPR abbreviation	Name	SPR number	Access	Supervisor only	Section/page
PID0	Process ID register 0. Book E defines only this PID register and refers to as PID, not PID0.	48	Read/Write	Yes	Section 2.12.1 on page 97
PID1	Process ID register 1	633	Read/Write	Yes	Section 2.12.1 on page 97
PID2	Process ID register 2	634	Read/Write	Yes	Section 2.12.1 on page 97
SPEFSCR	Signal processing and embedded floating-point status and control register	512	Read/Write	No	Section 2.14.1 on page 119
SVR	System version register	1023	Read-only	Yes	Section 2.7.5 on page 75
TLB0CFG	TLB configuration register 0	688	Read-only	Yes	Section 2.12.4 on page 100
TLB1CFG	TLB configuration register 1	689	Read-only	Yes	Section 2.12.4 on page 100

2.3 Registers for integer operations

The following sections describe registers defined for integer computational instructions.

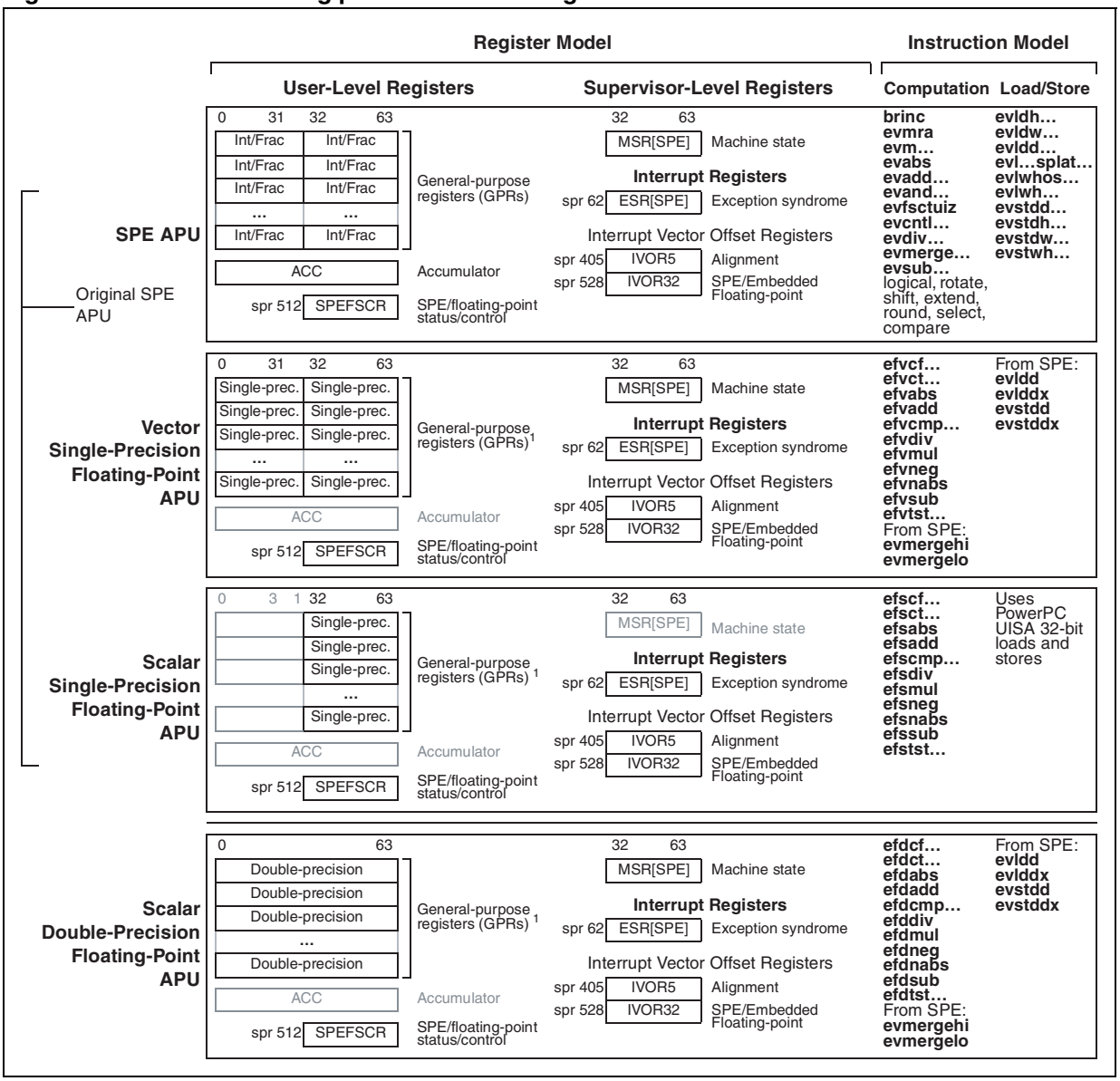
2.3.1 General purpose registers (GPRs)

Book E implementations provide 32 GPRs (GPR0–GPR31) for integer operations. The instruction formats provide 5-bit fields for specifying the GPRs to be used in the execution of the instruction.

The Book E architecture defines 32-bit GPRs for 32-bit implementations; however, several APUs make use of GPRs that are extended to 64 bits to accommodate either vector operands or embedded double-precision floating point operands. The following APUs use the extended 64-bit GPRs:

- The signal processing engine (SPE) APU and the embedded vector single-precision floating-point APU treat the 64-bit operands as consisting of two, 32-bit elements, as shown in *Figure 4*.
- The embedded scalar double-precision floating-point APU treats the GPRs as single 64-bit operands that accommodate IEEE double-precision values.

Figure 4. SPE and floating point APU GPR usage



Gray text indicates that the APU does not use this register or register field.

Formatting of floating-point operands is as defined by IEEE 754, as described in the APU chapter of the EREF.

As shown in *Figure 4*, the embedded scalar single-precision floating-point APU uses 32-bit operands that, like 32-bit Book E instructions, do not affect the upper word of the 64-bit GPRs. For 32-bit implementations that implement 64-bit GPRs, all instructions except SPE APU, embedded vector single-precision APU, and embedded scalar double-precision APU instructions use and return 32-bit values in GPR bits 32–63.

2.3.2 Integer exception register (XER)

Bits in the integer exception register (XER) are set based on the operation of an instruction considered as a whole, not on intermediate results. (For example, the subtract from carrying instruction (**subfc**), the result of which is specified as the sum of three values, sets bits in the XER based on the entire operation, not on an intermediate sum.)

Integer exception register (XER)

SPR1 Access: User read-write

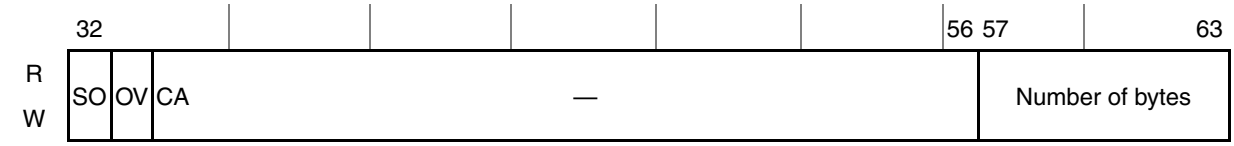


Table 9 describes XER bit definitions.

Table 9. XER field descriptions

Bits	Name	Description
32	SO	Summary overflow. Set when an instruction (except **mtspr**) sets the overflow bit (OV). Once set, SO remains set until it is cleared by **mtspr[XER]** or **mcrxr**. SO is not altered by compare instructions or by other instructions (except **mtspr[XER]** and **mcrxr**) that cannot overflow. Executing **mtspr[XER]**, supplying the values 0 for SO and 1 for OV, causes SO to be cleared and OV to be set.
33	OV	Overflow. X-form add, subtract from, and negate instructions having OE=1 set OV if the carry out of bit 32 is not equal to the carry out of bit 33, and clear OV otherwise to indicate a signed overflow. X-form multiply low word and divide word instructions having OE=1 set OV if the result cannot be represented in 32 bits (**mullwo**, **divwo**, and **divwuo**) and clear OV otherwise. OV is not altered by compare instructions or by other instructions (except **mtspr[XER]** and **mcrxr**) that cannot overflow.
34	CA	Carry. Add carrying, subtract from carrying, add extended, and subtract from extended instructions set CA if there is a carry out of bit 32 and clear it otherwise. CA can be used to indicate unsigned overflow for add and subtract operations that set CA. Shift right algebraic word instructions set CA if any 1 bits are shifted out of a negative operand and clear CA otherwise. Compare instructions and instructions that cannot carry (except Shift Right Algebraic Word, **mtspr[XER]**, and **mcrxr**) do not affect CA.
35–56	—	Reserved, should be cleared.
57–63	No. of Bytes	Supports emulation of load and store string instructions. Specifies the number of bytes to be transferred by a load string indexed or store string indexed instruction.

2.4 Registers for floating-point operations

This section details floating-point registers and their field descriptions.

2.4.1 Floating-point registers (FPRs)

Book E defines 32 floating-point registers (FPR0–FPR31). Floating-point instruction formats provide 5-bit fields for specifying FPRs used in instruction execution.

Each FPR contains 64 bits that support the floating-point format. Instructions that interpret FPR contents as floating-point values use double-precision format for this interpretation.

The computational instructions and the move and select instructions operate on data in FPRs and, except for compare instructions, place the result into an FPR, and optionally place status information into the CR.

Load and store double instructions are provided that transfer 64 bits of data between memory and the FPRs with no conversion. Load single instructions are provided to transfer and convert floating-point values in floating-point single format from memory 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 memory.

Instructions are provided that manipulate the FPSCR and the CR explicitly. Some of these instructions copy data between an FPR and the FPSCR.

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 CR (if Rc = 1), are undefined.

2.4.2 Floating-point status and control register (FPSCR)

The FPSCR, shown below, controls how floating-point exceptions are handled and records status resulting from floating-point operations. FPSCR[32–55] are status bits; FPSCR[56–63] are control bits.

Floating-point status and control register (FPSCR)

Access: User read/write

	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
R W	FX	FEX	VX	OX	UX	ZX	XX	VXSNAN	VXISI	VXIDI	VXZDZ	VXIMZ	VXVC	FR	FI	C
Reset	All zeros															

	48 51	52	53	54	55	56	57	58	59	60	61	62 63
R W	FPCC	—	VXSOFT	VXSQRT	VXCVI	VE	OE	UE	ZE	XE	NI	RN
Reset	All zeros											

The exception bits, FPSCR[35–45,53–55], are sticky; once set they remain set until they are cleared by an **mcrfs, mtfsfi, mtfsf,** or **mtfsb0**. Exception summary bits FPSCR[FX,FEX,VX] are not considered to be exception bits, and only FX is sticky.

FEX and VX are simply the ORs of other FPSCR bits, and so are not listed among the FPSCR bits affected by the various instructions. FPSCR fields are described in *Table 10*.

Table 10. FPSCR field descriptions

Bits	Name	Description
32	FX	Floating-point exception summary. Every floating-point instruction, except **mtfsfi** and **mtfsf**, implicitly sets FX 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.
33	FEX	Floating-point enabled exception summary. FEX 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	VX	Floating-point invalid operation exception summary. VX is the OR of all the invalid operation exception bits. **mcrfs**, **mtfsfi**, **mtfsf**, **mtfsb0**, and **mtfsb1** cannot alter FPSCR[VX] explicitly.
35	OX	Floating-point overflow exception
36	UX	Floating-point underflow exception
37	ZX	Floating-point zero divide exception
38	XX	Floating-point inexact exception. 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 FPSCR[XX] value 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	VXSNAN	Floating-point invalid operation exception (SNaN)
40	VXISI	Floating-point invalid operation exception ($\infty - \infty$)
41	VXIDI	Floating-point invalid operation exception ($\infty \div \infty$)
42	VXZDZ	Floating-point invalid operation exception ($0 \div 0$)
43	VXIMZ	Floating-point invalid operation exception ($\infty \times 0$)
44	VXVC	Floating-point invalid operation exception (invalid compare).
45	FR	Floating-point fraction rounded. The last arithmetic or rounding and conversion instruction incremented the fraction during rounding. This bit is not sticky.
46	FI	Floating-point fraction inexact. The last arithmetic or rounding and conversion instruction either produced an inexact result during rounding or caused a disabled overflow exception. This bit is not sticky. The definition of FPSCR[XX] describes the relationship between FPSCR[FI] and FPSCR[XX].
47–51	FPRF	Floating-point result flags. Set as described below in *Table 10*. For arithmetic, rounding, and conversion instructions, FPRF is set based on the result placed into the target register, except that if any portion of the result is undefined, the value placed into FPRF is undefined.
47	C	Floating-point result class descriptor. Arithmetic, rounding, and conversion instructions may set this bit with the FPCC bits, to indicate the class of the result.

Table 10. FPSCR field descriptions (continued)

Bits	Name	Description
48–51	FPCC	Floating-point condition code. Floating-point Compare instructions set one of the FPCC bits and clear the other three FPCC bits. Arithmetic, rounding, and conversion instructions may set the FPCC bits with the C bit to indicate the class of the result. In this case, the three high-order FPCC bits retain their relational significance indicating that the value is less than, greater than, or equal to zero. 48Floating-point less than or negative (FL or <) 49Floating-point greater than or positive (FG or >) 50Floating-point equal or zero (FE or =) 51Floating-point unordered or NaN (FU or ?)
52	—	Reserved, should be cleared.
53	VXSOFT	Floating-point invalid operation exception (software request). Can be altered only by **mcrfs**, **mtfsfi**, **mtfsf**, **mtfsb0**, or **mtfsb1**.
54	VXSQRT	Floating-point invalid operation exception (invalid square root). Note that VXSQRT is defined even for implementations that do not support either of the two optional instructions that set it, **fsqrt**[**.**] and **frsqrte**[**.**]. Defining it for all implementations gives software a standard interface for handling square root exceptions. If an implementation does not support **fsqrt**[**.**] or **frsqrte**[**.**], software can simulate the instruction and set VXSQRT to reflect the exception.
55	VXCVI	Floating-point invalid operation exception (invalid integer convert)
56	VE	Floating-point invalid operation exception enable
57	OE	Floating-point overflow exception enable
58	UE	Floating-point underflow exception enable
59	ZE	Floating-point zero divide exception enable
60	XE	Floating-point inexact exception enable
61	NI	Floating-point non-IEEE mode. If NI = 1, the remaining FPSCR bits may have meanings other than those given in this document and results of floating-point operations need not conform to the IEEE standard. If the IEEE-conforming result of a floating-point operation would be a denormalized number, the result of that operation is 0 (with the same sign as the denormalized number) if FPSCR[NI] = 1 and other requirements specified in the user's manual for the implementation are met. The other effects of setting NI may differ among implementations. Setting NI is intended to permit results to be approximate and to cause performance to be more predictable and less data-dependent than when NI = 0. For example, in non-IEEE mode, an implementation returns 0 instead of a denormalized number and may return a large number instead of an infinity. In non-IEEE mode an implementation should provide a means for ensuring that all results are produced without software assistance (that is, without causing an enabled exception type program interrupt or a floating-point unimplemented instruction exception type program interrupt and without invoking an emulation assist). The means may be controlled by one or more other FPSCR bits (recall that the other FPSCR bits have implementation-dependent meanings if NI = 1).
62–63	RN	Floating-point rounding control (RN). 00Round to nearest 01Round toward zero 10Round toward +infinity 11Round toward –infinity

Table 11 describes floating-point result flags.

Table 11. Floating-point result flags

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

2.5 Registers for branch operations

This section describes registers used by Book E branch and CR operations.

2.5.1 Condition register (CR)

The 32-bit CR reflects the result of certain operations and provides a mechanism for testing and branching.

Condition register (CR)

Access: User read/write

	32 35	36 39	40 43	44 47	48 51	52 55	56 59	60 63
R W	CR0	CR1	CR2	CR3	CR4	CR5	CR6	CR7
Reset	All zeros							

CR bits are grouped into eight 4-bit fields, CR0–CR7, which are set as follows:

- Specified CR fields can be set by a move to the CR from a GPR (**mtcrf**).
- A specified CR field can be set by a move to the CR from another CR field (**mcrf**), from the FPSCR (**mcrfs**), or from the XER (**mcrxr**).
- CR0 can be set as the implicit result of an integer instruction.
- CR1 can be set as the implicit result of a floating-point instruction.
- A specified CR field can be set as the result of either an integer or a floating-point compare instruction (including SPE and SPFP compare instructions).

Instructions are provided to perform logical operations on individual CR bits and to test individual CR bits (see *Condition register instructions on page 204*").

Note that instructions that access CR bits (for example, Branch Conditional (**bc**), CR logicals, and Move to Condition Register Field (**mtcrf**)) determine the bit position by adding

32 to the operand value. For example, in conditional branch instructions, the BI operand accesses bit BI + 32, as shown in *Table 12*.

Table 12. BI operand settings for CR fields

CR*n* Bits	CR Bits	BI	Description
CR0[0]	32	00000	Negative (LT)—Set when the result is negative. For SPE compare and test instructions: Set if the high-order element of **r**A is equal to the high-order element of **r**B; cleared otherwise.
CR0[1]	33	00001	Positive (GT)—Set when the result is positive (and not zero). For SPE compare and test instructions: Set if the low-order element of **r**A is equal to the low-order element of **r**B; cleared otherwise.
CR0[2]	34	00010	Zero (EQ)—Set when the result is zero. For SPE compare and test instructions: Set to the OR of the result of the compare of the high and low elements.
CR0[3]	35	00011	Summary overflow (SO). Copy of XER[SO] at the instruction's completion. For SPE compare and test instructions: Set to the AND of the result of the compare of the high and low elements.
CR1[0]	36	00100	Copy of FPSCR[FX] at the instruction's completion. Negative (LT) For SPE and SPFP compare and test instructions: Set if the high-order element of **r**A is equal to the high-order element of **r**B; cleared otherwise.
CR1[1]	37	00101	Copy of FPSCR[FEX] at the instruction's completion. Positive (GT) For SPE and SPFP compare and test instructions: Set if the low-order element of **r**A is equal to the low-order element of **r**B; cleared otherwise.
CR1[2]	38	00110	Copy of FPSCR[VX] at the instruction's completion. Zero (EQ) For SPE and SPFP compare and test instructions: Set to the OR of the result of the compare of the high and low elements.
CR1[3]	39	00111	Copy of FPSCR[OX] at the instruction's completion. Summary overflow (SO) For SPE and SPFP compare and test instructions: Set to the AND of the result of the compare of the high and low elements.
CR*n*[0]	40 44 48 52 56 60	01000 01100 10000 10100 11000 11100	Less than or floating-point less than (LT, FL). For integer compare instructions: **r**A < SIMM or **r**B (signed comparison) or **r**A < UIMM or **r**B (unsigned comparison). For floating-point compare instructions: **fr**A < **fr**B. For SPE and SPFP compare and test instructions: Set if the high-order element of **r**A is equal to the high-order element of **r**B; cleared otherwise.

Table 12. BI operand settings for CR fields (continued)

CR*n* Bits	CR Bits	BI	Description
CR*n*[1]	41 45 49 53 57 61	01001 01101 10001 10101 11001 11101	Greater than or floating-point greater than (GT, FG). For integer compare instructions: **r**A > SIMM or **r**B (signed comparison) or **r**A > UIMM or **r**B (unsigned comparison). For floating-point compare instructions: **fr**A > **fr**B. For SPE and SPFP compare and test instructions: Set if the low-order element of **r**A is equal to the low-order element of **r**B; cleared otherwise.
CR*n*[2]	42 46 50 54 58 62	01010 01110 10010 10110 11010 11110	Equal or floating-point equal (EQ, FE). For integer compare instructions: **r**A = SIMM, UIMM, or **r**B. For floating-point compare instructions: **fr**A = **fr**B. For SPE and SPFP compare and test instructions: Set to the OR of the result of the compare of the high and low elements.
CR*n*[3]	43 47 51 55 59 63	01011 01111 10011 10111 11011 11111	Summary overflow or floating-point unordered (SO, FU). For integer compare instructions, this is a copy of XER[SO] at the completion of the instruction. For floating-point compare instructions, one or both of **fr**A and **fr**B is a NaN. For SPE and SPFP vector compare and test instructions: Set to the AND of the result of the compare of the high and low elements.

CR setting for integer instructions

For all integer word instructions in which the Rc bit is defined and set, and for **addic.**, **andi.**, and **andis.**, CR0[32–34] are set by signed comparison of bits 32–63 of the result to zero; CR[35] is copied from the final state of XER[SO]. The Rc bit is not defined for double-word integer operations.

```
if      (target_register)32-63 < 0 then c ← 0b100
else if (target_register)32-63 > 0 then c ← 0b010
else                                    c ← 0b001
CR0 ← c || XERSO
```

The value of any undefined portion of the result is undefined, and the value placed into the first three bits of CR0 is undefined. CR0 bits are interpreted as described in *Table 13*.

Table 13. CR0 bit descriptions

CR bit	Name	Description
32	Negative (LT)	Bit 32 of the result is equal to one.
33	Positive (GT)	Bit 32 of the result is equal to zero, and at least one of bits 33–63 of the result is non-zero.
34	Zero (EQ)	Bits 32–63 of the result are equal to zero.
35	Summary overflow (SO)	This is a copy of the final state of XER[SO] at the completion of the instruction.

Note that CR0 may not reflect the true (infinitely precise) result if overflow occurs.

CR setting for store conditional instructions

CR0 is also set by the integer store conditional instruction, **stwcx.**. See instruction descriptions in *Chapter 3*," for detailed descriptions of how CR0 is set.

CR setting for floating-point instructions

For all floating-point instructions in which the Rc bit is defined and set, CR1 (CR[36–39]) is copied from FPSCR[32–35]. These bits are interpreted as shown in *Table 14*.

Table 14. CR setting for floating-point instructions

Bit	Name	Description
36	FX	Floating-point exception summary. Copy of final state of FPSCR[FX] at instruction completion.
37	FEX	Floating-point enabled exception summary. Copy of final state of FPSCR[FEX] at instruction completion.
38	VX	Floating-point invalid operation exception summary. Copy of final state of FPSCR[VX] at completion.
39	OX	Floating-point overflow exception. Copy of final state of FPSCR[OX] at instruction completion.

CR setting for compare instructions

For compare instructions, a CR field specified by the BI field in the instruction is set to reflect the result of the comparison, as shown in *Table 15*.

Table 15. CR setting for compare instructions

CR*n* bit	Bit expression	CR Bits		BI		Description
		AIM (BI Operand)	Book E	0–2	3–4	
CR*n*[0]	**4 * cr0 + lt** (or **lt**) **4 * cr1 + lt** **4 * cr2 + lt** **4 * cr3+ lt** **4 * cr4 + lt** **4 * cr5 + lt** **4 * cr6 + lt** **4 * cr7 + lt**	0 4 8 12 16 20 24 28	32 36 40 44 48 52 56 60	000 001 010 011 100 101 110 111	00	Less than or floating-point less than (LT, FL). For integer compare instructions: **r**A < SIMM or **r**B (signed comparison) or **r**A < UIMM or **r**B (unsigned comparison). For floating-point compare instructions: frA < frB.
CR*n*[1]	**4 * cr0 + gt** (or **gt**) **4 * cr1 + gt** **4 * cr2 + gt** **4 * cr3+ gt** **4 * cr4 + gt** **4 * cr5 + gt** **4 * cr6 + gt** **4 * cr7 + gt**	1 5 9 13 17 21 25 29	33 37 41 45 49 53 57 61	000 001 010 011 100 101 110 111	01	Greater than or floating-point greater than (GT, FG). For integer compare instructions: rA > SIMM or rB (signed comparison) or **r**A > UIMM or **r**B (unsigned comparison). For floating-point compare instructions: frA > frB.

Table 15. CR setting for compare instructions (continued)

CR*n* bit	Bit expression	CR Bits		BI		Description
		AIM (BI Operand)	Book E	0–2	3–4	
CR*n*[2]	**4 * cr0 + eq** (or **eq**) **4 * cr1 + eq** **4 * cr2 + eq** **4 * cr3+ eq** **4 * cr4 + eq** **4 * cr5 + eq** **4 * cr6 + eq** **4 * cr7 + eq**	2 6 10 14 18 22 26 30	34 38 42 46 50 54 58 62	000 001 010 011 100 101 110 111	10	Equal or floating-point equal (EQ, FE). For integer compare instructions: **r**A = SIMM, UIMM, or **r**B. For floating-point compare instructions: frA = frB.
CR*n*[3]	**4 * cr0 + so/un** (or **so/un**) **4 * cr1 + so**/un **4 * cr2 + so**/un **4 * cr3 + so**/un **4 * cr4 + so/un** **4 * cr5 + so/un** **4 * cr6 + so/un** **4 * cr7 + so/un**	3 7 11 15 19 23 27 31	35 39 43 47 51 55 59 63	000 001 010 011 100 101 110 111	11	Summary overflow or floating-point unordered (SO, FU). For integer compare instructions, this is a copy of XER[SO] at instruction completion. For floating-point compare instructions, one or both of frA and frB is a NaN.

CR bit settings in VLE mode

The VLE extension implements the entire CR, but some comparison operations and all branch instructions are limited to using CR0–CR3. However, all Book E CR field and logical operations are provided.

CR bits are grouped into eight 4-bit fields, CR0–CR7, which are set in one of the following ways.

- Specified CR fields can be set by a move to the CR from a GPR (**mtcrf**).
- A specified CR field can be set by a move to the CR from another CR field (**e_mcrf**).
- CR field 0 can be set as the implicit result of an integer instruction.
- A specified CR field can be set as the result of an integer compare instruction.
- CR field 0 can be set as the result of an integer bit test instruction.

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

CR settings for integer instructions

For all integer word instructions in which the Rc bit is defined and set, and for **addic.**, the first three bits of CR field 0 (CR[32–34]) are set by signed comparison of bits 32–63 of the result to zero, and the fourth bit of CR field 0 (CR[35]) is copied from the final state of XER[SO].

```
if      (target_register)32:63 < 0 then c ← 0b100
else if (target_register)32:63 > 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 in *Table 16*.

Table 16. CR0 encodings

CR bit	Description
32	Negative (LT). Bit 32 of the result is equal to 1.
33	Positive (GT). Bit 32 of the result is equal to 0 and at least one of bits 33–63 of the result is non-zero.
34	Zero (EQ). Bits 32–63 of the result are equal to 0.
35	Summary overflow (SO). This is a copy of the final state XER[SO] at the completion of the instruction.

CR setting for compare instructions supported by the VLE extension

For compare instructions, a CR field specified by the **cr**D operand in the instruction for the **e_cmph**, **e_cmphl**, **e_cmpi**, and **e_cmpli** instructions, or CR0 for the **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 in *Table 17*. A complete description of how the bits are set is given in *Chapter 6*," and in *Integer instructions on page 205*."

Table 17. Condition register setting for compare instructions

CR bit	Description
4×CRD + 32	Less than (LT). For signed-integer compare, GPR(**r**A or **r**X) < SCI8 or SI or GPR(**r**B or **r**Y). For unsigned-integer compare, GPR(**r**A or **r**X) $<_u$ SCI8 or UI or UI5 or GPR(**r**B or **r**Y).
4×CRD + 33	Greater than (GT). For signed-integer compare, GPR(**r**A or **r**X) > SCI8 or SI or UI5 or GPR(**r**B or **r**Y). For unsigned-integer compare, GPR(**r**A or **r**X) $>_u$ SCI8 or UI or UI5 or GPR(**r**B or **r**Y).
4×CRD + 34	Equal (EQ). For integer compare, GPR(**r**A or **r**X) = SCI8 or UI5 or SI or UI or GPR(**r**B or **r**Y).
4×CRD + 35	Summary overflow (SO). For integer compare, this is a copy of the final state of XER[SO] at the completion of the instruction.

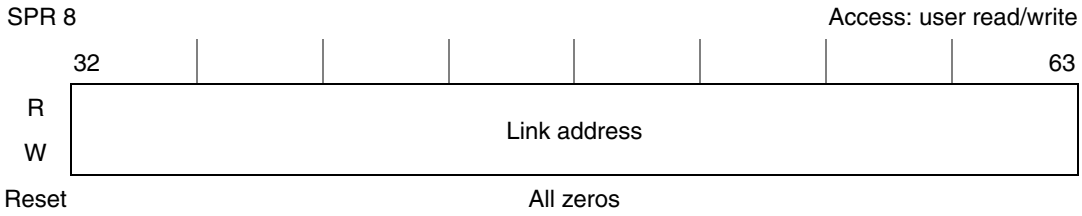
CR setting for the VLE bit test instruction

The Bit Test Immediate instruction, **se_btsti**, also sets CR field 0. See the instruction description and also *Integer instructions on page 205*

2.5.2 Link register (LR)

The link register can be used to provide the branch target address for a Branch Conditional to LR (**bclr***x*) instruction, and it holds the return address after branch and link instructions.

Link register (LR)



The LR contents are read into a GPR using **mfspr**. The contents of a GPR can be written to the LR using **mtspr**. LR[62–63] are ignored by **bclr** instructions.

Link register usage in VLE mode

VLE instructions use the LR as defined in Book E, although the VLE extension defines a subset of all variants of Book E conditional branches involving the LR, as shown in *Table 18*. Note that because VLE instructions can reside on half-word boundaries, in VLE mode, LR[30] is examined when the LR holds an instruction address.

Table 18. Branch to link register instruction comparison

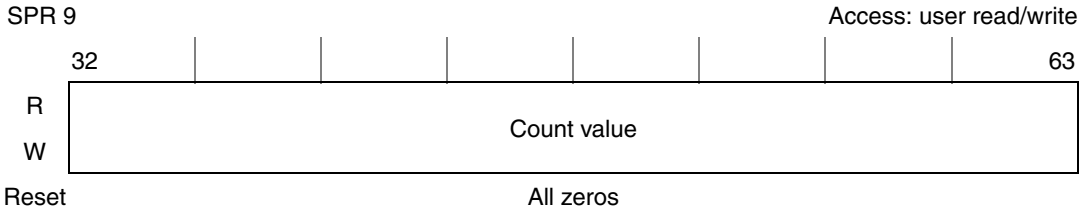
Book E		VLE Subset	
Instruction	**Syntax**	**Instruction**	**Syntax**
Branch Conditional to Link Register Branch Conditional to Link Register & Link	**bclr** BO,BI **bclrl** BO,BI	Branch (Absolute) to Link Register Branch (Absolute) to Link Register & Link	**se_blr** **se_blrl**
Branch Conditional & Link	**e_bcl** BO,BI,BD	Branch Conditional & Link	**e_bcl** BO32,BI32,BD 15
		Branch (Absolute) & Link	**e_bl** BD24 **se_bl** BD8

2.5.3 Count register (CTR)

CTR can be used to hold a loop count that can be decremented and tested during execution of branch instructions that contain an appropriately encoded BO field. If the CTR value is 0 before being decremented, it is –1 afterward. The entire CTR can be used to hold the branch target address for a Branch Conditional to CTR (**bcctr***x*) instruction.

Note that because VLE instructions can reside on half-word boundaries, in VLE mode, CTR[30] is examined when the CTR holds an instruction address.

Count register (CTR)



Count register usage in VLE mode

VLE instructions use the CTR as defined by in Book E, although the VLE extension defines a subset of the variants of Book E conditional branches involving the CTR, as shown in *Table 19*.

Table 19. Branch to count register instruction comparison

Book E		VLE	
Instruction	**Syntax**	**Instruction**	**Syntax**
Branch conditional to count register Branch conditional to count register & link	**bcctr** BO,BI **bcctrl** BO,BI	Branch (absolute) to count register Branch (absolute) to count register & link	**se_bctr** **se_bctrl**

2.6 Processor control registers

This section addresses machine state, processor ID, and processor version registers.

2.6.1 Machine state register (MSR)

The MSR defines the state of the processor (that is, enabling and disabling of interrupts and debugging exceptions, enabling and disabling of address translation for instruction and data memory accesses, enabling and disabling some APUs, and specifying whether the processor is in supervisor or user mode).

MSR contents are automatically saved, altered, and restored by the interrupt-handling mechanism. If a non-critical interrupt is taken, MSR contents are automatically copied into SRR1. If a critical interrupt is taken, MSR contents are automatically copied into CSRR1. When an **rfi** or **rfci** is executed, MSR contents are restored from SRR1 or CSRR1.

The EIS-defined machine check APU defines additional save/restore resources. When a machine check interrupt is taken, MCSRR0 and MCSRR1 hold the return address and MSR information. The return from machine check interrupt instruction, **rfmci**, restores MCSRR1 contents to the MSR.

MSR contents are read into a GPR using **mfmsr**. The contents of a GPR can be written to MSR using **mtmsr**. The write MSR external enable instructions (**wrtee** and **wrteei**) can be used to set or clear MSR[EE] without affecting other MSR bits.

Machine state register (MSR)

Access: supervisor-only

		32	36–37	38	39–43	44	45	46	47	48	49	50	51	52	53	54	55	56–57	58	59	60–61	62–63
Book E/ EIS	R/W	—				AP	WE	CE	—	EE	PR	FP	ME	FE0	—	DE	FE1	—	IS	DS	—	
EIS APUs	R/W	—	UCLE	SPE	—																PMM	—
Reset		All zeros																				

Table 20. MSR field descriptions

Bits	Name	Description
32–36	—	Reserved, should be cleared.(1)
37	UCLE	(Cache-locking APU) User-mode cache lock enable. Used to restrict user-mode cache-line locking by the operating system. 0Any cache lock instruction executed in user-mode takes a cache-locking DSI exception and sets either ESR[DLK] or ESR[ILK]. This allows the operating system to manage and track the locking/unlocking of cache lines by user-mode tasks. 1Cache-locking instructions can be executed in user-mode and they do not take a DSI for cache-locking. (They may still take a DSI for access violations though.)
38	SPE	(SPE, SPFP, DPFP APUs) SPE enable. Enables use of 64-bit extended GPRs used by SPE, single-precision vector, and double-precision floating-point APUs/ 0If software attempts to execute an SPE APU instruction, the SPE APU unavailable exception is taken. 1Software can execute any of the SPE APU instructions. Embedded floating-point instructions require MSR[SPE] to be set. An attempt to execute an embedded floating-point instruction when MSR[SPE] is 0 results in an SPE APU unavailable interrupt.
39–43	—	Reserved, should be cleared. 1
44	AP	APU available. Book E defines the operation of AP as follows: 0The processor cannot execute APU instructions. 1The processor can execute APU instructions.
45	WE	Wait state enable. Allows the core complex to signal a request for power management, according to the states of HID0[DOZE], HID0[NAP], and HID0[SLEEP]. 0The processor is not in wait state and continues processing. No power management request is signaled to external logic. 1The processor enters wait state by ceasing to execute instructions and entering low-power mode. Details of how wait state is entered and exited and how the processor behaves in the wait state are implementation-dependent.
46	CE	Critical enable 0Critical input and watchdog timer interrupts are disabled. 1Critical input and watchdog timer interrupts are enabled.
47	—	Preserved for Book III ILE
48	EE	External enable 0External input, decrementer, fixed-interval timer, and performance monitor interrupts are disabled. 1External input, decrementer, fixed-interval timer, and performance monitor interrupts are enabled.
49	PR	User mode (problem state) 0The processor is in supervisor mode, can execute any instruction, and can access any resource (for example, GPRs, SPRs, and the MSR). 1The processor is in user mode, cannot execute any privileged instruction, and cannot access any privileged resource. PR also affects memory access control.

Table 20. MSR field descriptions (continued)

Bits	Name	Description
50	FP	Floating-point available. 0The processor cannot execute floating-point instructions, including floating-point loads, stores, and moves. 1The processor can execute floating-point instructions.
51	ME	Machine check enable. 0Machine check interrupts are disabled. 1Machine check interrupts are enabled.
52	FE0	Floating-point exception mode 0. The Book E definition of this bit is shown in <Cross Refs>Table 21.
53	—	Allocated for implementation-dependent use.
54	DE	Debug interrupt enable 0Debug interrupts are disabled. 1Debug interrupts are enabled if DBCR0[IDM] = 1. See the description of the DBSR[UDE] in *Chapter 2.13.2*.
55	FE1	Floating-point exception mode 1. The Book E definition of this bit is shown in *Table 21*.
56	—	Reserved, should be cleared. [1]
57	—	Preserved for Book III IP
58	IS	Instruction address space 0The processor directs all instruction fetches to address space 0 (TS = 0 in the relevant TLB entry). 1The processor directs all instruction fetches to address space 1 (TS = 1 in the relevant TLB entry).
59	DS	Data address space 0The processor directs data memory accesses to address space 0 (TS = 0 in the relevant TLB entry). 1The processor directs data memory accesses to address space 1 (TS = 1 in the relevant TLB entry).
60	—	Reserved, should be cleared. [1]
61	PMM	(Performance monitor APU) Performance monitor mark bit. System software can set PMM when a marked process is running to enable statistics gathering only during the execution of the marked process. PMM and MSR[PR] 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 in the PMLCax, the state for which monitoring is enabled, counting is enabled.
62–63	—	Preserved for Book III RI and LE, respectively.

1. An MSR bit that is reserved may be altered by return from interrupt instructions.

The floating-point exception mode bits FE0 and FE1 are described in *Table 21*.

Table 21. Floating-point exception bits—MSR[FE0,FE1]

FE0	FE1	Mode
0	0	Ignore exceptions
0	1	Imprecise nonrecoverable
1	0	Imprecise recoverable
1	1	Precise

2.7 Hardware implementation-dependent registers

Each ST Book E processor implements hardware implementation-dependent registers, HID0 and HID1,which contain fields defined either by the EIS or by the implementation. This section provides architectural information about HID registers and describes only those bits that are defined by the EIS.

Note: 1 Not all processors implement HID fields defined by the EIS. Consult the user documentation.

2 An integrated device may not use all HID fields implemented on an embedded core or may define those fields more specifically. Always begin by looking at the core register descriptions in the reference manual for the integrated device.

2.7.1 Hardware implementation dependent register 0 (HID0)

HID0 is used for configuration and control. Figure below shows the HID0 bits that are defined either generally by the EIS or as part of an EIS-defined APU. Note that not all EIS-compliant device implement all HID0 fields; see the user documentation.

Writing to HID0 typically requires synchronization, as described in *Chapter 2.18.2*."

Hardware implementation dependent register 0 (HID0)

SPR 1008 Access: Supervisor-only

	32	33	34	35	39	40	42	43				47
R W	EMCP	—				PWRMGMT		DPM	EDPM	—	IPR	EN_MAS7_ UPDATE
Reset	All zeros											

	48	49	50	51	55	56	57	58			62	63
R W	EIEC	TBEN	SEL_TBCLK	—	DAPUEN	SGE	—	EIEIO_EN	LWSYNC_EN	—	NOPTST	NOPDST	NOPTI
Reset	All zeros											

HID0 fields are described in *Table 22*.

Table 22. HID0 field descriptions

Bits	Name	Description
32	EMCP	Enable machine check pin. Used to mask machine check exceptions delivered to the core from the machine check input. 0Machine check exceptions from the machine check signal are disabled. 1Machine check exceptions from the machine check signal are enabled. If MSR[ME] = 0, asserting the machine signal check causes a checkstop. If MSR[ME] = 1, asserting the machine check signal causes a machine check exception.
33	—	Implementation dependent.
34	SFR	Sixty-four bit results. Determines how the upper 32 bits of 64-bit registers in a 64-bit implementation are computed when the processor is executing in 32-bit mode (MSR[CM] = 0). 0In 32-bit mode, bits 0–31 of all 64-bit registers are not modified. Explicit 64-bit instructions generate an unimplemented instruction exception when executed. 1In 32-bit mode, bits 0–31 are written with the same value that is written as when the processor is executing in 64-bit mode (except for the LR and any EAs generated that clear bits 0–31. Explicit 64-bit instructions are allowed to execute and do not generate an unimplemented instruction exception unless they would have when the processor is in 64-bit mode.
35–39	—	Implementation dependent.
40–42	PWRMGMT	Power management control. The semantics of PWRMGMT are implementation dependent.
43	DPM	Dynamic power management. Used to enable power-saving by shutting off functional resources not in use. Setting or clearing DPM should not affect performance. 0Dynamic power management is disabled. 1Dynamic power management is enabled.
44	EDPM	Enhanced dynamic power management. Used to enable additional power-saving by shutting off functional resources not in use. Setting EDPM may have adverse effects on performance. 0Enhanced dynamic power management is disabled. 1Enhanced dynamic power management is enabled.
45	—	Implementation dependent.
46	ICR	Interrupt inputs clear reservation. Controls whether external input and critical input interrupts cause an established reservation to be cleared. 0External and critical input interrupts do not affect reservation status. 1External and critical input interrupts, when taken, clear an established reservation.
47	EN_MAS7_UPDATE	Enable hot-wire update of MAS7 register. Implementations that support this bit do not update MAS7 (upper RPN field) when hardware writes MAS registers via a **tlbre**, **tlbsx**, or an interrupt unless this bit is set. This provides a compatibility path for processors that originally offered only 32 bits of physical addressing but have since extended past 32 bits. 0Hardware updates of MAS7 are disabled. 1Hardware updates of MAS7 are enabled.

Table 22. HID0 field descriptions

Bits	Name	Description
48	EIEC	Enable internal error checking. Used to control whether internal processor errors cause a machine check exception. 0 Internal error reporting is disabled. Internally detected processor errors do not generate a machine check interrupt. 1 Internal error reporting is enabled. Internally detected processor errors generate a machine check interrupt.
49	TBEN	Time base enable. Used to control whether the time base increments. 0 The time base is not enabled and will not increment. 1 The time base is enabled and will increment. The rate at which the time base increments is determined by the value of HID0[SEL_TBCLK].
50	SEL_TBCLK	Select time base clock. Used to select the source of the time base clock. 0 The time base is updated based on a core implementation specific rate. 1 The time base is updated based on an external signal to the core
51–54	—	Implementation dependent.
55	DAPUEN	Debug APU enable. Controls whether the debug APU or enhanced debug APU is enabled. 0 The debug APU is disabled. Debug interrupts use CSRR0 and CSRR1 to save state and the **rfci** instruction to return from the debug interrupt. 1 The debug APU is enabled; debug interrupts use DSRR0 and DSRR1 to save state and the **rfdi** instruction to return from the debug interrupt.
56	SGE	Store gathering enable. Turns on store gathering for non-guarded cache inhibited or write-through stores. Details and characteristics of how stores are gathered is implementation dependent. 0 Store gathering is disabled. 1 Store gathering is enabled.
57	—	Implementation dependent.
58	EIEIO_EN	**eieio** synchronization enable. Allows **mbar** instructions to provide the same synchronization semantics as the **eieio** instruction. 0 Synchronization provided by **mbar** is performed in the Book E manner. Additional forms of synchronization, if implemented, are determined by the MO value. 1 Synchronization provided by **mbar** is equivalent to **eieio** synchronization. The MO field is ignored.
59	LWSYNC_EN	Lightweight synchronization enable. Allows **msync** instructions to provide the same synchronization semantics as the **sync** instructions from the PowerPC 2.xx architecture. 0 The synchronization provided by the **msync** instruction is performed in the Book E manner. 1 The synchronization provided by the **msync** instruction is based on the L field defined in PowerPC 2.xx architecture **sync** instruction.
60	—	Implementation dependent.

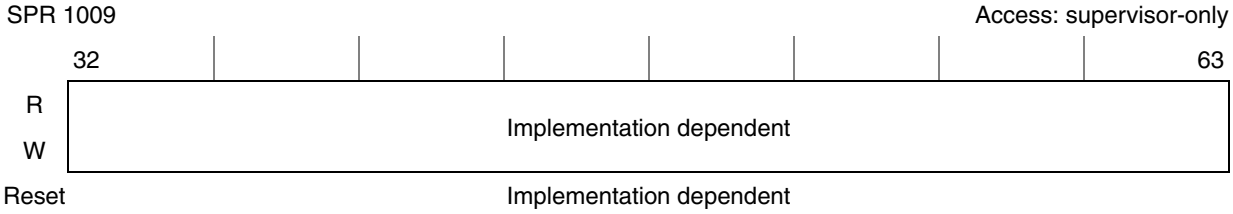
Table 22. HID0 field descriptions

Bits	Name	Description
61	NOPTST	No-op cache touch for store instructions. Controls whether data cache touch for store instructions perform no operation. 0 **dcbtst**, **dstst**, and **dststt** and other forms of cache touch for store instructions operate as defined by the EIS and Book E unless disabled by NOPDST or NOPTI. 1 **dcbtst**, **dstst**, and **dststt** and other forms of cache touch for store instructions are treated as no-ops. Cache line touch for store and lock instructions defined in the cache line locking APU operate as defined.
62	NOPDST	No-op **dst**, **dstt**, **dstst**, and **dststt** instructions. Instructions that start data stream prefetching through the dst instructions produce no-operation. 0 **dst**, **dstt**, **dstst**, and **dststt** operate as defined by the EIS unless disabled by NOPTST or NOPTI. 1 **dst**, **dstt**, **dstst**, and **dststt** are treated as no-ops and all current dst prefetch streams are terminated.
63	NOPTI	No-op cache touch instructions. Data and instruction cache touch instructions perform no operations. 0 **dcbt**, **dcbtst**, **icbt** and other forms of cache touch instructions operate as defined by the EIS and Book E unless disabled by NOPDST or NOPTST. 1 **dcbt**, **dcbtst**, **icbt** and other cache touch instruction forms are treated as no-ops. Cache line touch and lock instructions defined in the cache line locking APU operate as defined.

2.7.2 Hardware implementation dependent register 1 (HID1)

The EIS defines a HID1 register. HID1 contents are implementation dependent. HID1 is used for bus configuration and control. Writing to HID1 requires synchronization, as described in *Chapter 2.18.2: Synchronization requirements for SPRs*."

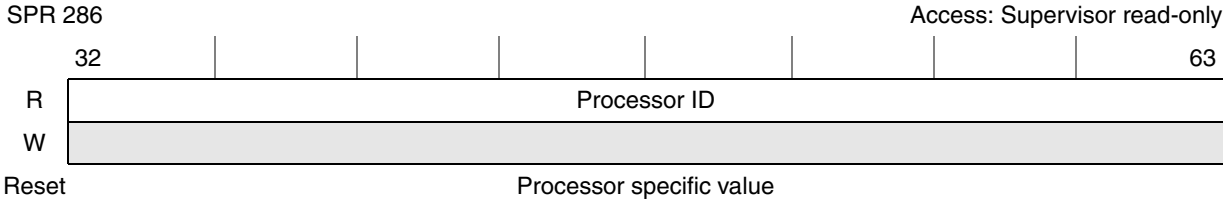
Hardware implementation dependent register 1 (HID1)



2.7.3 Processor ID register (PIR)

The processor ID register (PIR), shown below, contains a value that can be used to distinguish the processor from other processors in the system.

Processor ID register (PIR)



2.7.4 Processor version register (PVR)

The read-only processor version register (PVR), contains a value identifying the version and revision level of the processor. The PVR distinguishes between processors that differ in attributes that may affect software.

Processor version register (PVR)

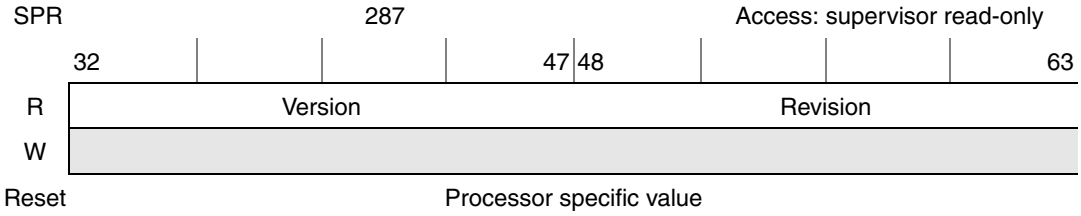


Table 23 describes PVR fields.

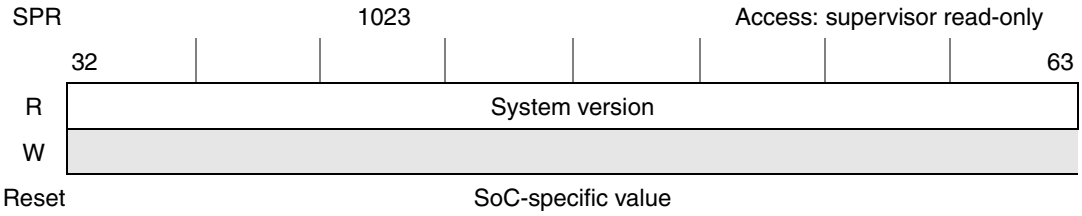
Table 23. PVR field descriptions

Bits	Name	Description
32–47	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.
48–63	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.

2.7.5 System version register (SVR)

The system version register (SVR), contains a read-only SoC-dependent value; consult the documentation for the implementation.

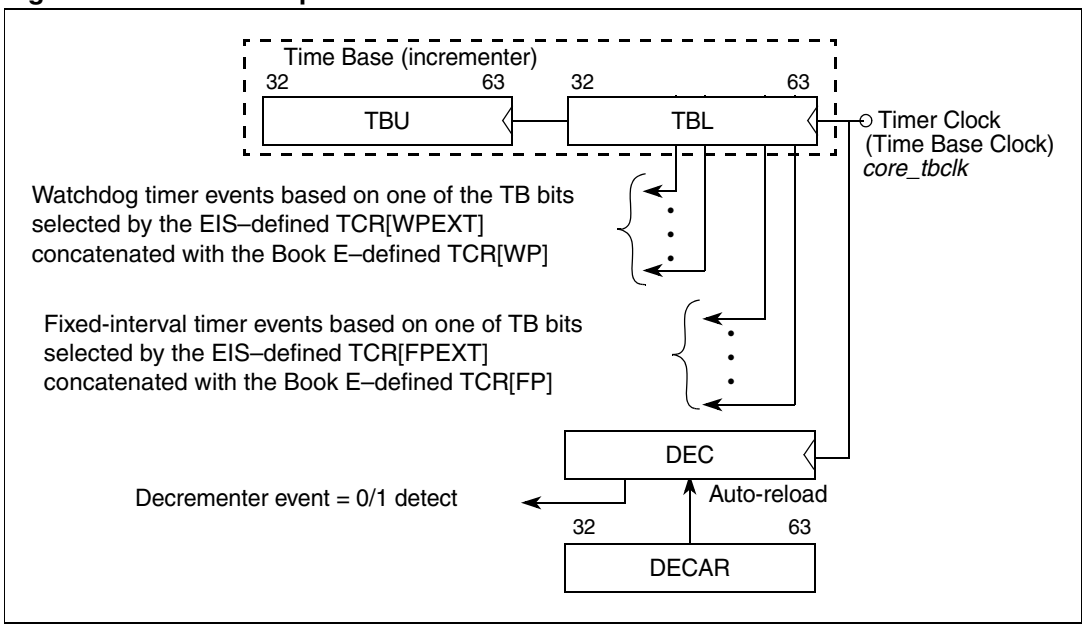
System version register (SVR)



2.8 Timer registers

The time base (TB), decrementer (DEC), fixed-interval timer (FIT), and watchdog timer provide timing functions for the system. The relationship of these timer facilities to each other is shown in *Figure 5* and is described as follows:

Figure 5. Relationship of timer facilities to the time base



- The TB is a long-period counter driven at an implementation-dependent frequency.
- The decrementer, updated at the same rate as the TB, provides a way to signal an exception after a specified period unless one of the following occurs:
  - DEC is altered by software in the interim.
  - The TB update frequency changes.
- The DEC is typically used as a general-purpose software timer.
- The time base for the TB and DEC is selected by the time base enable (TBEN) and select time base clock (SEL_TBCLK) bits in HID0, as follows:
  - If HID0[TBEN] = 1 and HID0[SEL_TBCLK] = 0, the time base is updated every 8 bus clocks.
  - If HID0[TBEN] = 1 and HID0[SEL_TBCLK] = 1, the time base is updated by an implementation-specific clock input).
- Software can select one from of four TB bits to signal a fixed-interval interrupt whenever the bit transitions from 0 to 1. It is typically used to trigger periodic system maintenance functions. Bits that may be selected are implementation-dependent.
- The watchdog timer, also a selected TB bit, provides a way to signal a critical exception when the selected bit transitions from 0 to 1. It is typically used for system error recovery. If software does not respond in time to the initial interrupt by clearing the associated status bits in the TSR before the next expiration of the watchdog timer interval, a watchdog timer-generated processor reset may result, if so enabled.

All timer facilities must be initialized during start-up.

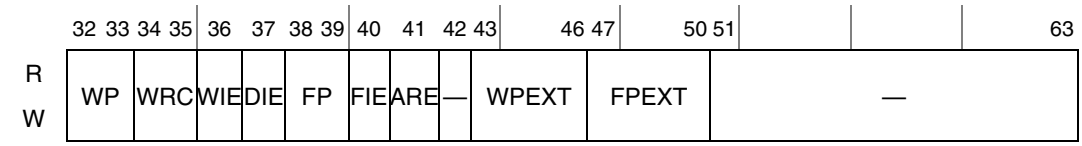
2.8.1 Timer control register (TCR)

The TCR, provides control information for the on-chip timer of the core complex. The core complex implements two fields not specified in Book E: TCR[WPEXT] and TCR[FPEXT].

The 32-bit timer control register (TCR), controls the decrementer. (See *Chapter 2.8.4*.)

Timer control register (TCR)

SPR 340 Access: Supervisor read/write



Reset Processor specific value

Table 24 describes the TCR fields.

Table 24. TCR field descriptions

Bits	Name	Description
32–33	WP	Watchdog timer period. When concatenated with WPEXT, specifies one of 64-bit locations of the time base used to signal a watchdog timer exception on a transition from 0 to 1. WPEXT,WP = 0000_00 selects TBU[32] (the msb of the TB) WPEXT,WP = 1111_11 selects TBL[63] (the lsb of the TB)
34–35	WRC	Watchdog timer reset control. When a watchdog reset event occurs, the value programmed into WRC is reflected on *core_wrs* and into TSR[WRS], but the WRC bits are reset to 00. At this point, software can reprogram WRC. Although WRC can be set by software, it cannot be cleared by software (except by a software-induced reset). Once written to a non-zero value, WRC may no longer be altered by software. 00 No watchdog timer reset will occur. TCR[WRC] resets to 00; it can be set by software, but cannot be cleared by software (except by a software-induced reset). xx Other values: Force processor to be reset on second time-out of watchdog timer. The exact function of any of these settings is implementation-dependent.
36	WIE	Watchdog timer interrupt enable 0 Watchdog timer interrupts disabled 1 Watchdog timer interrupts enabled
37	DIE	Decrementer interrupt enable 0 Decrementer interrupts disabled 1 Decrementer interrupts enabled
38–39	FP	Fixed interval timer period. When concatenated with FPEXT, FP specifies one of 64 bit locations of the time base used to signal a fixed-interval timer exception on a transition from 0 to 1. FPEXT||FP = 0000_00 selects TBU[32] (the msb of the TB) FPEXT||FP = 1111_11 selects TBL[63] (the lsb of the TB)
40	FIE	Fixed interval interrupt enable 0 Fixed interval interrupts disabled 1 Fixed interval interrupts enabled
41	ARE	Auto-reload enable. Controls whether the value in DECAR is reloaded into the DEC when the DEC value reaches 0000_0001. 0 Auto-reload disabled 1 Auto-reload enabled
42	—	Reserved, should be cleared.
43–46	WPEXT	(EIS) Watchdog timer period extension (see the description for WP)

Table 24. TCR field descriptions (continued)

Bits	Name	Description
47–50	FPEXT	(EIS) Fixed-interval timer period extension (see the description for FP)
51–63	—	Reserved, should be cleared.

2.8.2 Timer status register (TSR)

As shown below, the 32-bit TSR contains status on timer events and the most recent watchdog timer-initiated processor reset. All TSR bits function as write-1-to-clear.

Note: Register fields designated as write-1-to-clear are cleared only by writing ones to them. Writing zeros to them has no effect.

Timer status register (TSR)

SPR 336

Access: supervisor w1c

336

	32	33	34 35	36	37	38						63
R	ENW	WIS	WRS	DIS	FIS	—						
W	w1c	w1c	w1c	w1c	w1c							
Reset	All zeros											

Table 25 describes TSR fields.

Table 25. TSR field descriptions

Bits	Name	Description
32	ENW	Enable next watchdog time. When a watchdog timer time-out occurs while WIS = 0 and the next watchdog time-out is enabled (ENW = 1), a watchdog timer exception is generated and logged by setting WIS. This is referred to as a watchdog timer first time out. A watchdog timer interrupt occurs if enabled by TCR[WIE] and MSR[CE]. To avoid another watchdog timer interrupt once MSR[CE] is reenabled (assuming TCR[WIE] is not cleared instead), the interrupt handler must reset TSR[WIS] by executing an **mtspr**, setting WIS and any other bits to be cleared and a 0 in all other bits. The data written to the TSR is not direct data, but a mask. A 1 causes the bit to be cleared; a 0 has no effect. 0 Action on next watchdog timer time-out is to set TSR[ENW]. 1 Action on next watchdog timer time-out is governed by TSR[WIS].
33	WIS	Watchdog timer interrupt status. See the ENW description for more information about how WIS is used. 0 A watchdog timer event has not occurred. 1 A watchdog timer event occurred. When MSR[CE] = 1 and TCR[WIE] = 1, a watchdog timer interrupt is taken.
34–35	WRS	Watchdog timer reset status. Defined at reset (value = 00). Set to TCR[WRC] when a reset is caused by the watchdog timer. 00 No watchdog timer reset has occurred. xx All other values are implementation-dependent.

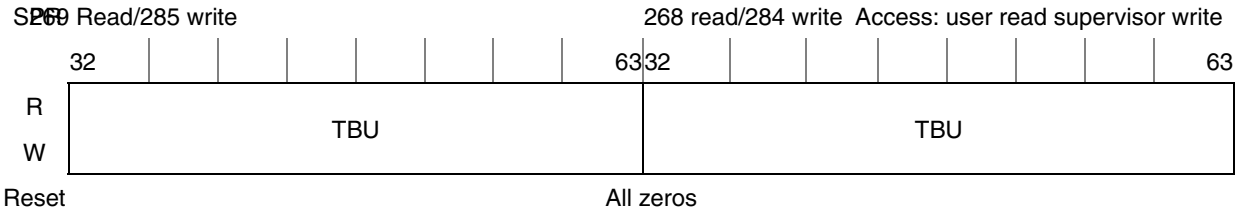
Table 25. TSR field descriptions (continued)

Bits	Name	Description
36	DIS	Decrementer interrupt status. 0 A decrementer event has not occurred. 1 A decrementer event occurred. When MSR[EE] = TCR[DIE] = 1, a decrementer interrupt is taken.
37	FIS	Fixed-interval timer interrupt status. 0 A fixed-interval timer event has not occurred. 1 A fixed-interval timer event occurred. When MSR[EE] = 1 and TCR[FIE]= 1, a fixed-interval timer interrupt is taken.
38–63	—	Reserved, should be cleared.

2.8.3 Time base (TBU and TBL)

The time base (TB), seen below, is composed of two 32-bit registers, the time base upper (TBU) concatenated on the right with the time base lower (TBL). TB provides timing functions for the system. TB is a volatile resource and must be initialized during start-up.

Time base upper/lower registers (TBU/TBL)



The TB is interpreted as a 64-bit unsigned integer that is incremented periodically. Each increment adds 1 to the least-significant bit. The frequency at which the integer is updated is implementation-dependent.

TBL increments until its value becomes 0xFFFF_FFFF ($2^{32} – 1$). At the next increment, its value becomes 0x0000_0000 and TBU is incremented. This process continues until the TBU value becomes 0xFFFF_FFFF and value TBL value becomes 0xFFFF_FFFF (TB is interpreted as 0xFFFF_FFFF_FFFF_FFFF ($2^{64} – 1$)). At the next increment, the TBU value becomes 0x0000_0000 and the TBL value becomes 0x0000_0000. There is no interrupt (or any other indication) when this occurs.

The period depends on the driving frequency. For example, if TB is driven by 100 MHz divided by 32, the TB period is as follows:

$$T_{TB} = 2^{64} \times \frac{32}{10MHz} = 5.90 \times 10^{12} \text{ seconds (approximately 187,000 years)}$$

The TB is implemented such that the following requirements are satisfied:

- Loading a GPR from the TB has no effect on the accuracy of the TB.
- Storing a GPR to the TB replaces the value in the TB with the value in the GPR.

Book E does not specify a relationship between the frequency at which the TB is updated and other frequencies, such as the CPU clock or bus clock in a Book E system. The TB

update frequency is not required to be constant. One of the following is required to ensure that system software can keep time of day and operate interval timers:

- The system provides an (implementation-dependent) interrupt to software whenever the update frequency of the TB changes and a way to determine the current update frequency.
- The update frequency of the TB is under the control of system software.

Note: 1 Disabling the TB or making reading the time base privileged prevents the TB from being used to implement a covert channel in a secure system.

2 If the operating system initializes the TB on power-on to some reasonable value and the update frequency of the TB is constant, the TB 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 TB are monotonically increasing (except when the TB wraps from 2^{64} – 1 to 0). If a trace entry is recorded each time the update frequency changes, the sequence of TB values can be post-processed to become actual time values.
Successive readings of the TB may return identical values.

It is intended that the TB be useful for timing reasonably short sequences of code (a few hundred instructions) and for low-overhead time stamps for tracing.

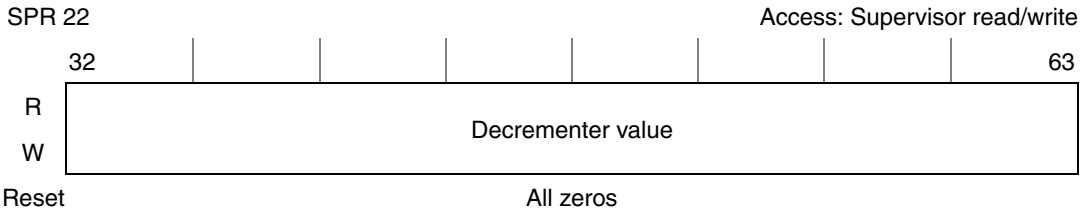
2.8.4 Decrementer register

The 32-bit decrementer (DEC), shown below, is a decrementing counter that is updated at the same rate as the TB. It provides a way to signal a decrementer interrupt after a specified period unless one of the following occurs:

- DEC is altered by software in the interim.
- The TB update frequency changes.

DEC is typically used as a general-purpose software timer. The decrementer auto-reload register is used to automatically reload a programmed value into DEC, as described in *Section 2.8.5: Decrementer auto-reload register (DECAR)*."

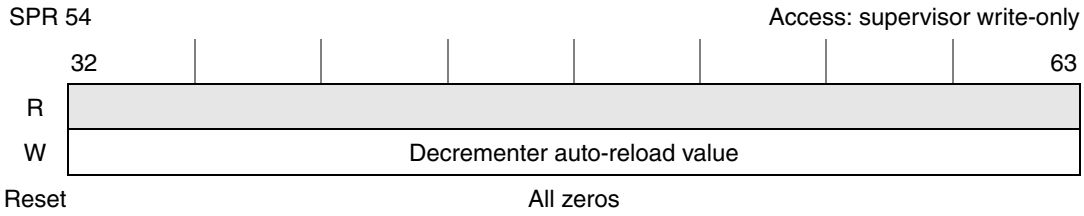
Decrementer register (DEC)



2.8.5 Decrementer auto-reload register (DECAR)

The decrementer auto-reload register is shown in figure below. If the auto-reload function is enabled (TCR[ARE] = 1), the auto-reload value in DECAR is written to DEC when DEC decrements from 0x0000_0001 to 0x0000_0000. Note that writing DEC with zeros by using an **mtspr[DEC]** does not automatically generate a decrementer exception.

Decrementer auto-reload register (DECAR)



2.9 Interrupt registers

Chapter 2.9.1: Interrupt registers defined by book E on page 81," describes registers used for interrupt handling.

2.9.1 Interrupt registers defined by book E

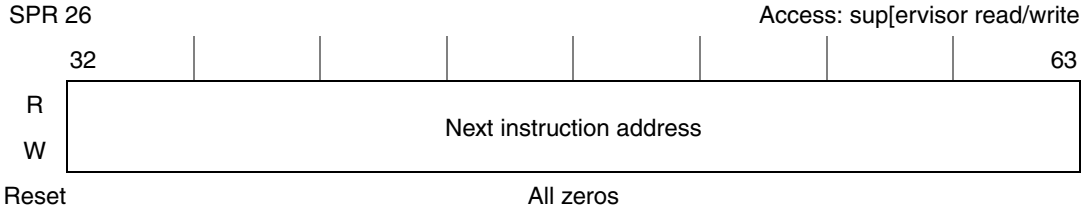
This section describes the following register bits and their fields:

- *Save/restore register 0 (SRR0) on page 81*"
- *Save/restore register 1 (SRR1) on page 81*"
- *Critical save/restore register 0 (CSRR0) on page 82*"
- *Critical save/restore register 1 (CSRR1) on page 82*"
- *Data exception address register (DEAR) on page 82*"
- *Interrupt vector prefix register (IVPR) on page 83*"
- *Interrupt vector offset registers (IVORs) on page 83*"
- *Exception syndrome register (ESR) on page 84*"

Save/restore register 0 (SRR0)

On a noncritical interrupt, SRR0, shown in figure below, holds the address of the instruction where the interrupted process should resume. The instruction is interrupt-specific, although for instruction-caused exceptions, it is typically the address of the instruction that caused the interrupt. When **rfi** executes, instruction execution continues at the address in SRR0.

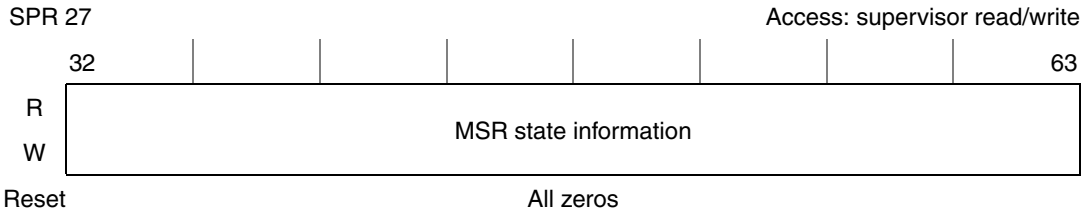
Save/restore register 0 (SRR0)



Save/restore register 1 (SRR1)

SRR1 is provided to save and restore machine state on noncritical interrupts. When a noncritical interrupt is taken, MSR contents are placed in SRR1. When **rfi** executes, SRR1 contents are placed into MSR. SRR1 bits that correspond to reserved MSR bits are also reserved. These registers are not affected by **rfci** or **rfmci**. Reserved MSR bits may be altered by **rfi**, **rfci**, or **rfmci**.

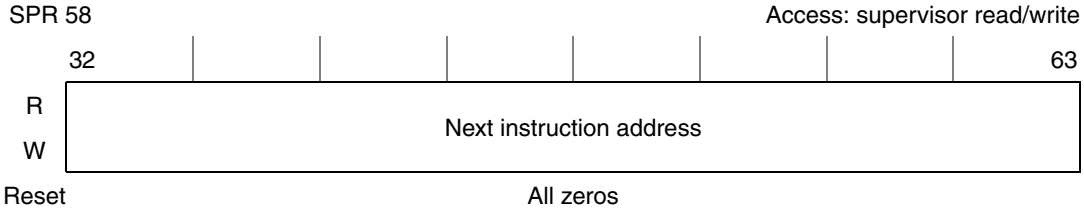
Save/restore register 1 (SRR1)



Critical save/restore register 0 (CSRR0)

CSRR0, is provided to save and restore machine state on critical interrupts. It is used by critical interrupts like SRR0 is used for standard interrupts: to hold the address of the instruction to which control is passed at the end of the interrupt handler. When **rfci** executes, instruction execution continues at the address in CSRR0.

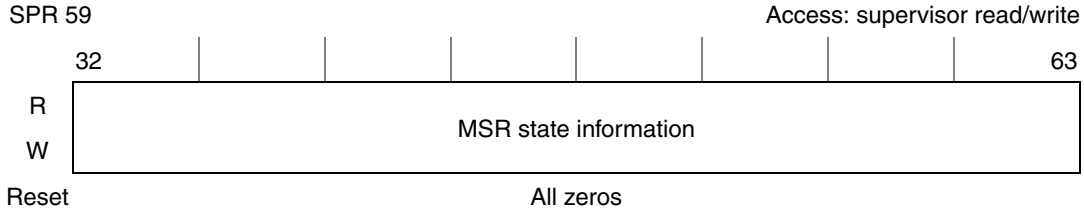
Critical save/restore register 0 (CSRR0)



Critical save/restore register 1 (CSRR1)

CSRR1, is used to save and restore machine state on critical interrupts. When a critical interrupt is taken, MSR contents are placed into CSRR1. When **rfci** executes, CSRR1 contents are restored into the MSR. CSRR1 bits that correspond to reserved MSR bits are also reserved; reserved MSR bits may be altered.

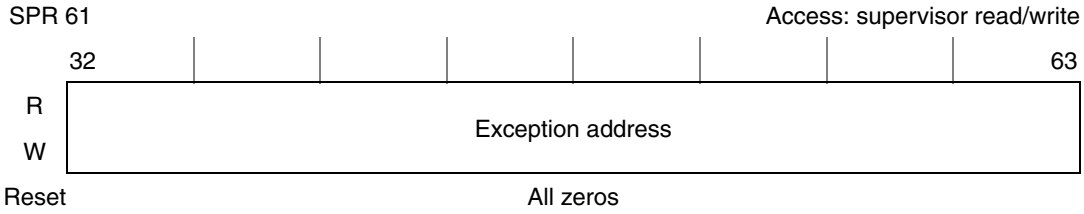
Critical save/restore register 1 (CSRR1)



Data exception address register (DEAR)

DEAR, is loaded with the effective address of a data access (caused by a load, store, or cache management instruction) that results in an alignment, data TLB miss, or DSI exception.

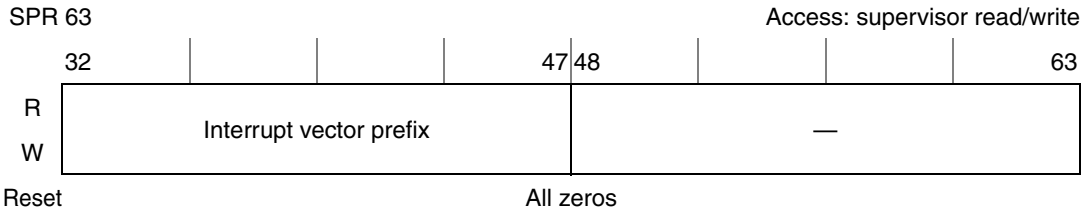
Data exception address register (DEAR)



Interrupt vector prefix register (IVPR)

IVPR is used with IVORs to determine the vector address. IVPR[32–47] provides the high-order 16 bits of the address of the exception processing routines. The 16-bit vector offsets are concatenated to the right of IVPR[32–47] to form the address of the exception processing routine. IVPR[48–63] are reserved.

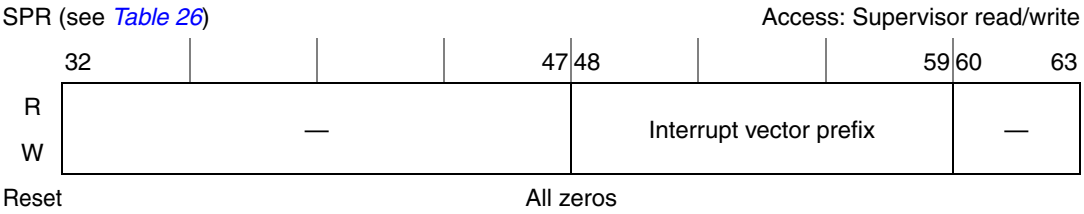
Interrupt vector prefix register (IVPR)



Interrupt vector offset registers (IVORs)

IVORs, hold the quad-word index from the base address provided by the IVPR for each interrupt type.

Interrupt vector offset registers (IVOR)



SPR numbers corresponding to IVOR16–IVOR31 are reserved. IVOR32–IVOR47 and IVOR60–IVOR63 are reserved. SPR numbers for IVOR32–IVOR63 are allocated for implementation-dependent use. IVOR assignments are shown in *Table 26*.

Table 26. IVOR assignments

IVOR Number	SPR	Interrupt type
IVOR0	400	Critical input
IVOR1	401	Machine check
IVOR2	402	Data storage
IVOR3	403	Instruction storage
IVOR4	404	External input
IVOR5	405	Alignment
IVOR6	406	Program
IVOR7	407	Floating-point unavailable
IVOR8	408	System call
IVOR9	409	Auxiliary processor unavailable (optional)
IVOR10	410	Decrementer

Table 26. IVOR assignments (continued)

IVOR Number	SPR	Interrupt type
IVOR11	411	Fixed-interval timer interrupt
IVOR12	412	Watchdog timer interrupt
IVOR13	413	Data TLB error
IVOR14	414	Instruction TLB error
IVOR15	415	Debug
IVOR16–IVOR31	—	Reserved for future architectural use
IVOR36–IVOR63 allocated for implementation dependent use		
IVOR32	528	SPE APU unavailable
IVOR33	529	(Embedded FP APUs) embedded floating-point data exception
IVOR34	530	(Embedded FP APUs) embedded floating-point round exception
IVOR35	531	(Performance monitor APUs) performance monitor

Exception syndrome register (ESR)

The ESR, provides a syndrome to differentiate between different kinds of exceptions that can generate the same interrupt type. When such an interrupt is generated, bits corresponding to the specific exception that generated the interrupt are set and all other ESR bits are cleared. Other interrupt types do not affect ESR contents. The ESR does not need to be cleared by software. *Table 27* shows ESR bit definitions.

EIS storage defines ESR[DLK] and ESR[ILK] to indicate user cache line locking exceptions, ESR[XTE] for precise external transaction errors, and ESR[EPID] external PID load and store exceptions.

The ESR is defined in Book E. Bits architected by EIS storage are defined here.

Exception syndrome register (ESR)

SPR62 Access: Supervisor read/write

		32–35	36	37	38	39	40	41	42	43	44	45	46	47–55	56	57–58	59–61	62	63
Book E	R/W	—	PIL	PPR	PTR	FP	ST	—	DLK0	DLK1	AP	PUO	BO	—					
EIS	R/W	—							DLK	ILK	—				SPE	—	VLEMI	—	MIF	XTE
Reset		All zeros																	

Table 27 describes ESR bit definitions.

Table 27. Exception syndrome register (ESR) definition

Bits	Name	Syndrome	Interrupt types
32–35	—	Reserved, should be cleared. (Defined by Book E as allocated.)	—
36	PIL	Illegal instruction exception	Program
37	PPR	Privileged instruction exception	Program

Table 27. Exception syndrome register (ESR) definition (continued)

Bits	Name	Syndrome	Interrupt types
38	PTR	Trap exception	Program
39	FP	Floating-point operations	Alignment, data storage, data TLB, program
40	ST	Store operation	Alignment, data storage, data TLB error
41	—	Reserved, should be cleared.	—
42	DLK	Defined by cache line locking APU. Instruction cache locking attempt. Set when a DSI occurs because a **dcbtls**, **dcbtstls**, or **dcblc** was executed in user mode (MSR[PR] = 1) while MSR[UCLE] = 0. 0 Default 1 DSI occurred on an attempt to lock line in data cache when MSR[UCLE] = 0.	Data storage
43	ILK	Defined by cache line locking APU. Instruction cache locking attempt. Set when a DSI occurs because an **icbtls** or **icblc** was executed in user mode (MSR[PR] = 1) while MSR[UCLE] = 0. 0 Default 1 DSI occurred on an attempt to lock line in instruction cache when MSR[UCLE] = 0.	Data storage
44	APU	Auxiliary processor operation. Defined by Book E.	Alignment, data storage, data TLB, program
45	PUO	Unimplemented operation exception. Defined by Book E.	Program
46	BO	Byte-ordering exception. Defined by Book E and the VLE extension.	Data storage, instruction storage
47	PIE	Imprecise exception. Defined by Book E.	Program
48–55	—	Reserved.	—
56	SPE	Defined by SPE, embedded floating-point APU. SPE/embedded floating-point exception bit 0 Default 1 Any exception caused by an SPE/embedded floating-point instruction occurred.	Data storage, Data TLB error, Alignment, SPE unavailable, Embedded FP unavailable, Embedded FP data, Embedded FP round
57	—	Reserved, should be cleared	

Table 27. Exception syndrome register (ESR) definition (continued)

Bits	Name	Syndrome	Interrupt types
58	VLEMI	Defined by VLE extension. VLEMI indicates that an interrupt was caused by a VLE instruction. VLEMI is set on an exception associated with execution or attempted execution of a VLE instruction. 0 The instruction page associated with the instruction causing the exception does not have the VLE attribute set or the VLE extension is not implemented. 1 The instruction page associated with the instruction causing the exception has the VLE attribute set and the VLE extension is implemented.	Data storage, Data TLB error, Instruction storage, Program, System Call, Alignment, SPE unavailable, Embedded FP unavailable, Embedded FP data, Embedded FP round
59–61	—	Reserved. Defined by Book E as allocated.	—
62	MIF	Defined by the VLE extension. MIF indicates that an interrupt was caused by a misaligned instruction fetch (NIA_{62} != 0) and the VLE attribute is cleared for the page or the second half of a 32-bit VLE instruction caused an instruction TLB error. 0 Default. 1 NIA_{62} != 0 and the instruction page associated with NIA does not have the VLE attribute set or the second half of a 32-bit VLE instruction caused an instruction TLB error.	Instruction TLB error, Instruction Storage
63	XTE	External transaction error. An external transaction reported an error but the error was handled precisely by the core. The contents of SRR0 contain the address of the instruction that initiated the transaction. 0 Default. No external transaction error was precisely detected. 1 An external transaction reported an error that was precisely detected.	Instruction storage, Data storage

Note: ESR information is incomplete, so system software may need to identify the type of instruction that caused the interrupt and examine the TLB entry and the ESR to fully identify the exception or exceptions. For example, a data storage interrupt may be caused by both a protection violation exception and a byte-ordering exception. System software would have to look beyond ESR[BO], such as the state of MSR[PR] in SRR1 and the TLB entry page protection bits to determine if a protection violation also occurred.

EIS-defined interrupt registers

This section describes machine check save/store and syndrome registers.

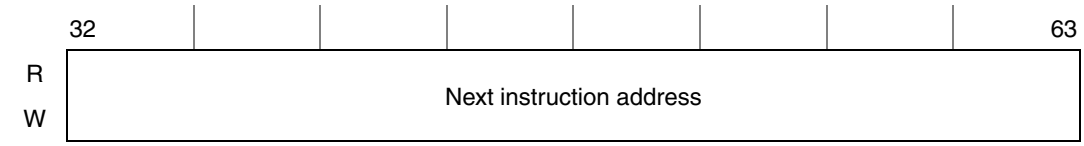
Debug save/restore register 0 (DSRR0)

On a debug interrupt, DSRR0, holds the address of the instruction where the interrupted process should resume. The instruction is interrupt specific. See *Chapter 4.7.16: Debug interrupt on page 271*." When **rfdi** executes, instruction execution continues at the address in DSRR0. DSRR0 and DSRR1 are not affected by **rfi**, **rfci**, or other return from interrupt instructions

Debug save/restore register 0 (DSRR0)

574

SPR 574 Access: Supervisor read/write



Reset Undefined

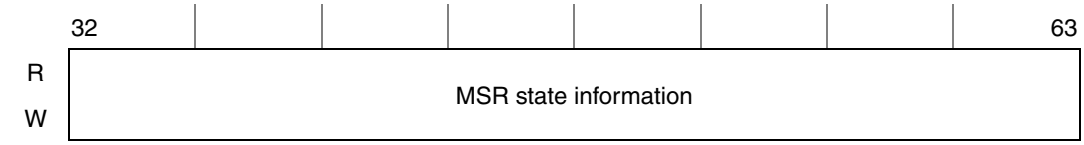
Debug Save/restore register 1 (DSRR1)

DSRR1, is provided to save and restore machine state on debug interrupts. When a debug interrupt is taken, MSR contents are placed into DSRR1. When **rfdi** executes, the contents of DSRR1 are restored into MSR. DSRR1 bits that correspond to reserved MSR bits are also reserved. (See *Section 2.6.1: Machine state register (MSR)*," for more information.) DSRR0 and DSRR1 are not affected by **rfi** or **rfci**. Reserved MSR bits may be altered by **rfi**, **rfci**, or **rfdi**.

Debug save/restore register 1 (DSRR1)

574

SPR 575 Access: Supervisor read/write



Reset Implementation-specific

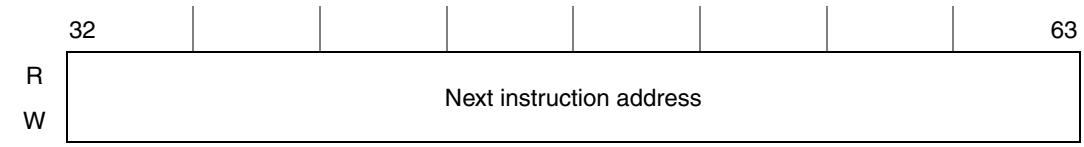
Machine check save/restore register 0 (MCSRR0)

When a machine check interrupt is taken, MCSRR0, is set to the address of the instruction where the interrupted process should resume. The instruction is interrupt-specific, although typically MCSRR0 holds address of the instruction that caused the interrupt. When **rfmci** is executed, instruction execution continues at this address.

Machine check save/restore register 0 (MCSRR0)

574

SPR 570 Access: Supervisor read/write



Reset All zeros

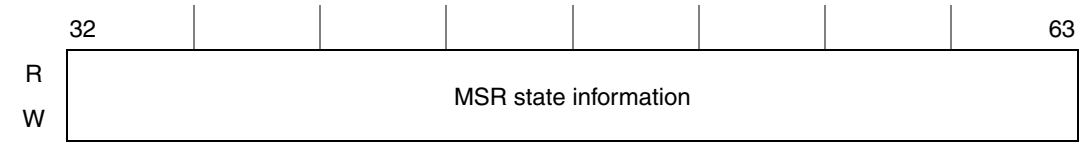
Machine check save/restore register 1 (MCSRR1)

MCSRR1 is used to save and restore machine state on machine check interrupts. When a machine check interrupt is taken, MSR contents are placed into MCSRR1. When **rfmci** executes, MCSRR1 contents are restored to MSR. MCSRR1 bits that correspond to reserved MSR bits are also reserved; reserved MSR bits may be altered.

Machine check save/restore register 1 (MCSRR1)

574

SPR 571 Access: Supervisor read/write



Reset All zeros

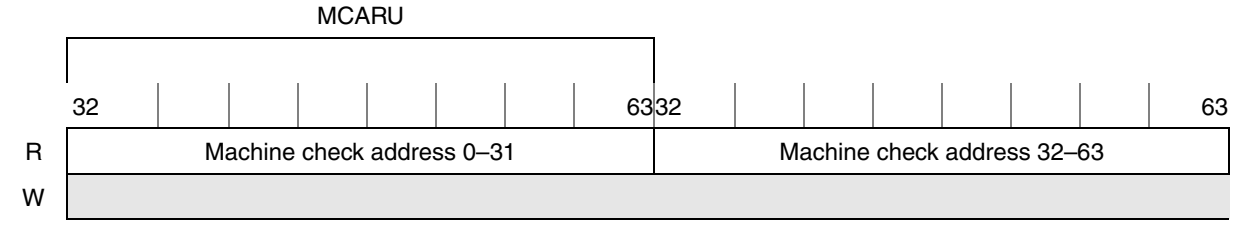
Machine check address register (MCAR/MCARU)

When the core complex takes a machine check interrupt, it updates MCAR, to indicate the address of the data associated with the machine check. Note that if a machine check interrupt is caused by a signal, MCAR contents are not meaningful. Errors that cause MCAR contents to be updated are implementation-dependent. If MCSR[MAV] = 1, the address is an effective address; if MAV = 0, the address is a real address.

Machine check address register (MCAR/MCARU)

SPR MCAR: 573
MCARU: 569

Access: Supervisor read-only



Reset All zeros

For 32-bit implementations that support physical addresses greater than 32 bits, MCARU provides an alias to the upper address bits that reside in MCAR[0–31].

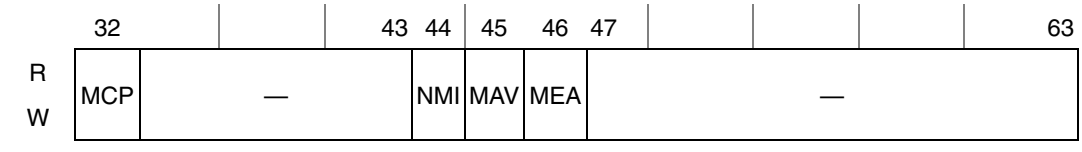
Machine check syndrome register (MCSR)

The MCSR, is used to record the cause of the machine check interrupt. In general, machine check syndrome bits correlating to specific hardware error conditions are implementation dependent. Consult the users manual for a complete definition of machine check error syndromes for a specific processor.

Machine check syndrome register 1 (MCSR)

574

SPR 572 Access: Read/w1c



Reset All zeros

Table 28 describes the MCSR fields.

Table 28. MCSR field descriptions

Bits	Name	Description
32	MCP	Machine check input to core. Processor cores with a machine check input pin (signal) respond to a signal input by producing an asynchronous machine check. The existence of such a signal and how such a signal is generated is implementation dependent and may be tied to a an external pin on the IC package.
33–42	—	Implementation-dependent.
43	NMI	Nonmaskable Interrupt. Set if a non-maskable interrupt (NMI) has been sent to the virtual processor.
44	MAV	MCAR address valid. The address contained in MCAR was updated by the processor and corresponds to the first detected error condition that contained an associated address. Any subsequent machine check errors that have associated addresses are not placed in MCAR unless MAV is 0 when the error is logged. 0 The address in MCAR is not valid. 1 The address in MCAR is valid. Note: Software should read MCAR before clearing MAV. MAV should be cleared before setting MSR[ME].
45	MEA	MCAR effective address. Denotes the type of address in MCAR. MEA has meaning only if MCSR[MAV] is set. 0 The address in MCAR is a physical address. 1 The address in MCAR is an effective address (untranslated).
46–63	—	Implementation-dependent.

Note: The machine check interrupt handler should always write what is read back to the MCSR after the error information has been logged. Writing contents that were read from the MCSR back to the MCSR clears only those status bits that were previously read. Failure to clear all MCSR bits causes an asynchronous machine check interrupt when MSR[ME] is set.

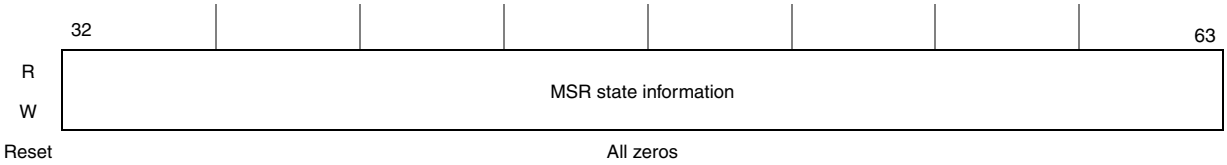
2.10 Software use sprs (SPRG0–SPRG7 and USPRG0)

Software-use SPRs (SPRG0–SPRG7 and USPRG0), have no defined functionality. These are shown below:

- SPRG0–SPRG2—can be accessed only in supervisor mode.
- SPRG3—can be written only in supervisor mode. It is readable in supervisor mode, but whether it can be read in user mode is implementation-dependent.
- SPRG4–SPRG7—can be written only in supervisor mode; readable in supervisor or user mode.
- USPRG0—can be accessed in supervisor or user mode.

Software-use sprs (SPRG0–SPRG7 and USPRG0)

SPR	SPRG0	272	Read/write	Supervisor
	SPRG1	273	Read/write	Supervisor
	SPRG2	274	Read/write	Supervisor
	SPRG3	259	Read-only	User (Implementation-dependent)/supervisor
		275	Read/write	Supervisor
	SPRG4	260	Read-only	User/supervisor
		276	Read/write	Supervisor
	SPRG5	261	Read-only	User/supervisor
		277	Read/write	Supervisor
	SPRG6	262	Read-only	User/supervisor
		278	Read/write	Supervisor
	SPRG7	263	Read-only	User/supervisor
		279	Read/write	Supervisor
	USPRG0	256	Read/write	User/supervisor



Software-use SPRs are read into a GPR by using **mfspr** and are written by using **mtspr**.

2.11 L1 cache registers

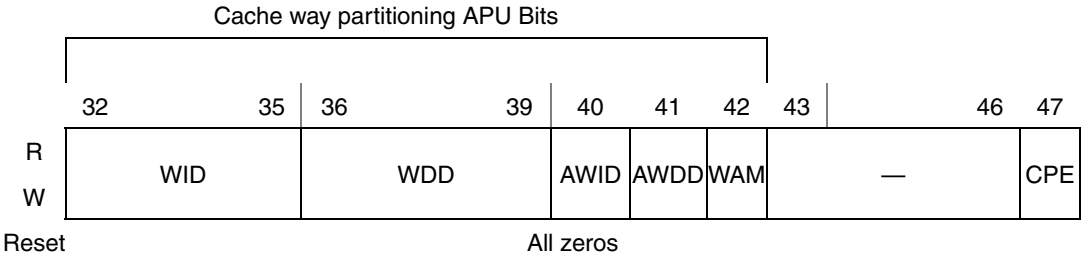
The EIS defines registers that provide control and configuration and status information for the L1 cache implementation.

2.11.1 L1 cache control and status register 0 (L1CSR0)

The L1CSR0, is defined by the EIS. It is used for general control and status of the L1 data cache.

L1 cache control and status register 0 (L1CSR0)

SPR 1010 Supervisor read/write



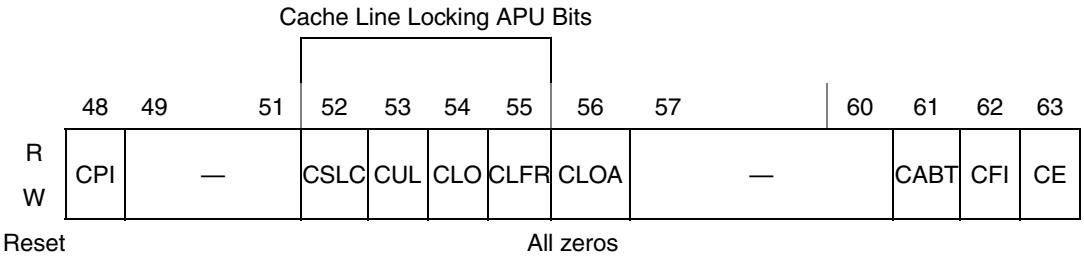


Table 29 describes the L1CSR0 fields.

Table 29. L1CSR0 field descriptions

Bits	Name	Description
32–35	WID	Cache way partitioning APU. Way instruction disable. (bit 32 = way 0, bit 33 = way 1, … bit 35 = way 3). 0 The corresponding way is available for replacement by instruction miss line refills. 1 The corresponding way is not available for replacement by instruction miss line refills.
36–39	WDD	Cache way partitioning APU. Way data disable (bit 36 = way 0, bit 37 = way 1, … bit 39 = way 3). 0 The corresponding way is available for replacement by data miss line refills. 1 The corresponding way is not available for replacement by data miss line refills
40	AWID	Cache way partitioning APU. Additional ways instruction disable. 0 Additional ways beyond 0–3 are available for replacement by instruction miss line fills. 1 Additional ways beyond 0–3 are not available for replacement by instruction miss line fills.
41	AWDD	Cache way partitioning APU. Additional ways data disable. 0 Additional ways beyond 0–3 are available for replacement by data miss line fills. 1 Additional ways beyond 0–3 are not available for replacement by data miss line fills.
42	WAM	Cache way partitioning APU. Way access mode. 0 All ways are available for access. 1 Only ways partitioned for the specific type of access are used for a fetch or read operation.
43-46	—	Reserved for implementation dependent use.
47	CPE DCPE	[Data] Cache parity enable. 0 Parity checking of the cache disabled 1 Parity checking of the cache enabled
48	CPI DCPI	[Data] Cache parity error injection enable. 0 Parity error injection disabled 1 Parity error injection enabled. Note that cache parity must also be enabled (L1CSR0[CPE] = 1) when this bit is set. If DCPE is not set, results are undefined and erratic behavior may occur. It is recommended that an attempt to set this bit when L1CSR0[CPE] = 0 cause the bit not to be set (that is, L1CSR0[CPI] = L1CSR0[CPE] & L1CSR0[CPI]).
49–51	—	Reserved, should be cleared.
52	CSLC DCSLC	[Data]Cache snoop lock clear. Sticky bit set by hardware if a cache line lock was cleared by a snoop operation which caused an invalidation. Note that the lock for that line is cleared whenever the line is invalidated. This bit can be cleared only by software. 0 The cache has not encountered a snoop that invalidated a locked line. 1 The cache has encountered a snoop that invalidated a locked line.
53	CUL DCUL	[Data]Cache unable to lock. Sticky bit set by hardware. This bit can be cleared only by software. 0 Indicates a lock set instruction was effective in the cache 1 Indicates a lock set instruction was not effective in the cache
54	CLO DCLO	[Data]Cache lock overflow. Sticky bit set by hardware. This bit can be cleared only by software. 0 Indicates a lock overflow condition was not encountered in the cache 1 Indicates a lock overflow condition was encountered in the cache

Table 29. L1CSR0 field descriptions (continued)

Bits	Name	Description
55	CLFC DCLFC	[Data]Cache lock bits flash clear. Clearing occurs regardless of the enable (L1CSR0[CE]) value. 0 Default. 1 Hardware initiates a cache lock bits flash clear operation. Cleared when the operation is complete. During a flash clear operation, writing a 1 causes undefined results; writing a 0 has no effect
56	CLOA DCLOA	[Data]Cache lock overflow allocate. Set by software to allow a lock request to replace a locked line when a lock overflow situation exists. Implementation of this bit is optional. 0 Indicates a lock overflow condition does not replace an existing locked line with the requested line 1 Indicates a lock overflow condition replaces an existing locked line with the requested line
57–60	—	Reserved, should be cleared.
61	CABT DCABT	[Data]Cache operation aborted. 0 No cache operation completed improperly 1 Cache operation did not complete properly
62	CFI DCFI	[Data]Cache flash invalidate. Invalidation occurs regardless of the enable (L1CSR0[CE]) value. 0 No cache invalidate. 1 Cache flash invalidate operation. A cache invalidation operation is initiated by hardware. Once complete, this bit is cleared. During an invalidation operation, writing a 1 causes undefined results; writing a 0 has no effect.
63	CE DCE	[Data]Cache enable. 0 The cache is not enabled. (not accessed or updated) 1 Enables cache operation.

2.11.2 L1 cache control and status register 1 (L1CSR1)

L1CSR1, defined as part of the EIS, is used for general control and status of the L1 instruction cache.

L1 cache control and status register 1 (L1CSR1)

011 SPR 1011 Access: supervisor read/write

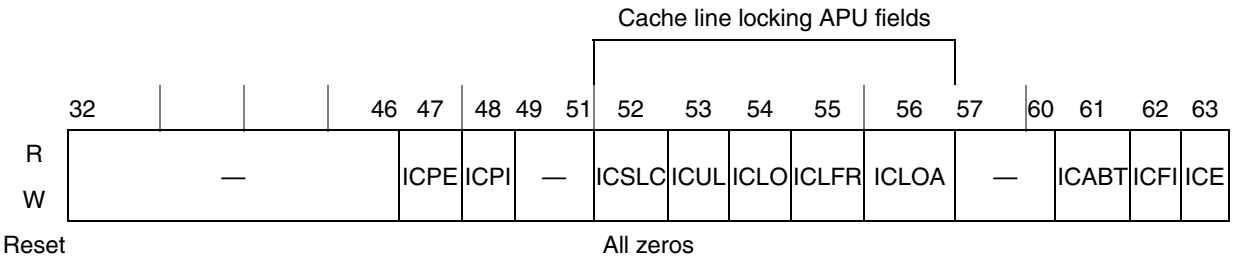


Table 30 describes the L1CSR1 fields.

Table 30. L1CSR1 field descriptions

Bits	Name	Description
32–42	—	Reserved, should be cleared.
43-46	—	Reserved for implementation dependent use.
47	ICPE	Instruction cache parity enable. See *Chapter 4.7.2: Machine check interrupt*." 0 Parity checking of the cache disabled 1 Parity checking of the cache enabled
48	ICPI	Instruction cache parity error injection enable. 0 Parity error injection disabled 1 Parity error injection enabled. Note that cache parity must also be enabled (L1CSR1[ICPE] = 1) when ICPI is set. If L1CSR0[ICPE] is not set the results are undefined and erratic behavior may occur. It is recommended that an attempt to set this bit when L1CSR0[ICPE] = 0 causes the bit not to be set (that is, L1CSR0[ICPI] = L1CSR0[ICPE] & L1CSR0[ICPI]).
49–51	—	Reserved, should be cleared.
52	ICSLC	Cache line locking APU. Instruction cache snoop lock clear. Sticky bit set by hardware if a cache line lock was cleared by a snoop operation that caused an invalidation. Note that the lock for that line is cleared whenever the line is invalidated. This bit can be cleared only by software. 0 The cache has not encountered a snoop that invalidated a locked line. 1 The cache has encountered a snoop that invalidated a locked line.
53	ICUL	Cache line locking APU. Instruction cache unable to lock. Sticky bit set by hardware. This bit can be cleared only by software. 0 Indicates a lock set instruction was effective in the cache 1 Indicates a lock set instruction was not effective in the cache
54	ICLO DCLO	Cache line locking APU. Instruction cache lock overflow. Sticky bit set by hardware. This bit can be cleared only by software. 0 Indicates a lock overflow condition was not encountered in the cache 1 Indicates a lock overflow condition was encountered in the cache
55	ICLFC	Cache line locking APU. Instruction cache lock bits flash clear. Clearing occurs regardless of the enable (L1CSR1[ICE]) value. 0 Default. 1 Hardware initiates a cache lock bits flash clear operation. This bit is cleared when the operation is complete. During a flash clear operation, writing a 1 causes undefined results; writing a 0 has no effect.
56	ICLOA	Cache line locking APU. Instruction cache lock overflow no allocate. Set by software to prevent a lock request from replacing a locked line when a lock overflow situation exists. Implementation of this bit is optional. 0 Indicates a lock overflow condition replaces an existing locked line with the requested line 1 Indicates a lock overflow condition does not replace an existing locked line with the requested line
57–60	—	Reserved, should be cleared.
61	ICABT	Instruction cache operation aborted. 0 No cache operation completed improperly 1 Cache operation did not complete properly

Table 30. L1CSR1 field descriptions (continued)

Bits	Name	Description
62	ICFI	Instruction cache flash invalidate. Invalidation occurs regardless of the enable (L1CSR1[ICE]) value. 0 No cache invalidate. 1 Cache flash invalidate operation. A cache invalidation operation is initiated by hardware. Once complete, this bit is cleared. During an invalidation operation, writing a 1 causes undefined results; writing a 0 has no effect.
63	ICE	Instruction cache enable. 0 The cache is not enabled. (not accessed or updated) 1 Enables cache operation.

2.11.3 L1 cache configuration register 0 (L1CFG0)

The L1CFG0 register, shown below, is defined by the EIS to provide configuration information for the primary (L1) data cache of the processor. If a processor implements a unified cache, L1CFG0 applies to the unified cache and L1CFG1 is not implemented.

L1 cache configuration register 0 (L1CFG0)

SPR 515 Access: user read-only

	32 33	34 35	36	37 38	39	40 41 42	43	44 45	52	63
R	CARCH	CWPA CFAHA	CFISWA	—	CBSIZE	CREPL	CLA	CPA	CNWAY	CSIZE
W										

Reset Implementation-dependent value

Table 31. L1CFG0 field descriptions

Bits	Name	Description
32–33	CARCH	Cache architecture 00 Harvard 01 Unified
34	CWPA	Cache way partitioning APU available. 0 Unavailable 1 Available
35	CFAHA	Cache flush all by hardware available 0 Unavailable 1 Available
36	CFISWA	Direct cache flush APU available. (Cache flush by set and way available.) 0 Unavailable 1 Available
37–38	—	Reserved, should be cleared.

Table 31. L1CFG0 field descriptions (continued)

Bits	Name	Description
39–40	CBSIZE	Cache line size 0032 bytes 0164 bytes 10128 bytes 11Reserved
41–42	CREPL	Cache replacement policy 00 True LRU 01 Pseudo LRU 1x Reserved
43	CLA	Cache line locking APU available 0 Unavailable 1 Available
44	CPA	Cache parity available 0 Unavailable 1 Available
45–52	CNWAY	Cache number of ways minus 1.
53–63	CSIZE	Cache size in Kbytes.

2.11.4 L1 cache configuration register 1 (L1CFG1)

The L1CFG1 register, provides configuration information for the L1 instruction cache. If a processor implements a unified cache, L1CFG0 applies to the unified cache and L1CFG1 is not implemented.

L1 cache configuration register 1 (L1CFG1)

SPR 516 Access: user read-only

	32	38	39	40	41	42	43	44	45	52	53	63
R	—		ICBSIZE		ICREPL		ICLA	ICPA	ICNWAY		ICSIZE	
W												

Reset Implementation-dependent value

Table 32. L1CFG1 field descriptions

Bits	Name	Description
32–38	—	Reserved, should be cleared.
39–40	ICBSIZ	Instruction cache block size 0032 bytes 0164 bytes 10128 bytes 11Reserved

Table 32. L1CFG1 field descriptions (continued)

Bits	Name	Description
41–42	ICREPL	Cache replacement policy 00True LRU 01Pseudo LRU 1xReserved
43	ICLA	Cache line locking APU available 0Unavailable 1Available
44	ICPA	Cache parity available 0Unavailable 1Available
45–52	ICNWAY	Cache number of ways minus 1.
53–63	ICSIZE	Cache size in Kbytes.

2.11.5 L1 flush and invalidate control register 0 (L1FINV0)

The direct cache flush APU defines the L1 flush and invalidate control register 0 (L1FINV0), shown in figure below. The direct cache flush APU allows the programmer to flush and/or invalidate the cache by specifying the cache set and cache way. The direct cache flush APU available bit, L1CFG0[CFISWA], is set for implementations that contain the direct cache flush APU.

To address a specific physical block of the cache, the L1FINV0 is written with the cache set (L1FINV0[CSET]) and cache way (L1FINV0[CWAY]) of the line that is to be flushed. No tag match in the cache is required.

Only the L1 data cache (or unified cache) is manipulated by the direct cache flush APU. The L1 instruction cache or any other caches in the cache hierarchy are not explicitly targeted by this APU. See *Chapter 8.2: Direct cache flush APU on page 850*."

L1 flush and invalidate control register 0 (L1FINV0)

SPR 1016 Access: supervisor read/write

	32	39	40	41	42				58	59	61	62	63
R W	CWAY		—			CSET				—	CCMD		
Reset	All zeros												

Table 33. L1FINV0 fields—L1 direct cache flush

Bits	Name	Descriptions
0–31	—	Reserved, should be cleared.
32–39	CWAY	Cache way. Specifies the cache way to be selected.
40–41	—	Reserved, should be cleared.
42–58	CSET	Cache set. Specifies the cache set to be selected.

Table 33. L1FINV0 fields—L1 direct cache flush

Bits	Name	Descriptions
59–61	—	Reserved, should be cleared.
62–63	CCMD	Cache flush command. 00 Implementation dependent. If implemented, the action performed on the line should be synonymous with a **dcbi** instruction that references the same line. 01 The line specified by CWAY and CSET is flushed if it is modified and valid. It is implementation dependent whether it remains in the cache, or is invalidated. For an implementation, the action performed on the line should be synonymous with a **dcbst** instruction that references the same line. 01 The line specified by CWAY and CSET is flushed if it is modified and valid. It is then invalidated. For an implementation, the action performed on the line should be synonymous with a **dcbf** instruction that references that line. 11 Reserved for future use.

2.12 MMU registers

This section describes the following MMU registers and their fields:

- Process ID registers (PID0–PID2)
- MMU control and status register 0 (MMUCSR0)
- MMU configuration register (MMUCFG)
- TLB configuration registers (TLB*n*CFG)
- MMU assist registers (MAS0–MAS7)

2.12.1 Process ID registers (PID0–PID*n*)

The Book E architecture specifies that a process ID (PID) value be associated with each effective address (instruction or data) generated by the processor.

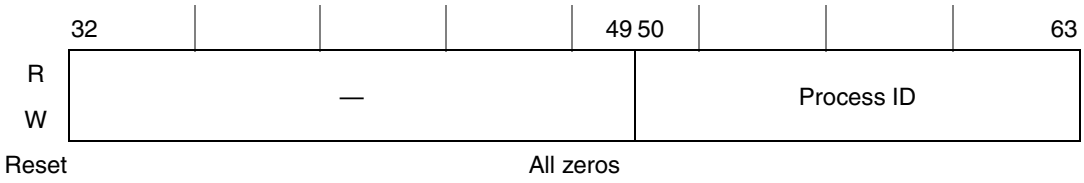
System software uses PIDs to identify TLB entries that the processor uses to translate addresses for loads, stores, and instruction fetches. PID contents are compared to the TID field in TLB entries as part of selecting appropriate TLB entries for address translation. PID values are used to construct virtual addresses for accessing memory. Note that individual processors may not implement all 14 bits of the process ID field.

Book E defines one PID register that holds the PID value for the current process. ST devices may implement from 1 to 15 PID registers. The number of PIDs implemented is indicated by the value of MMUCFG[NPIDS]. Consult the user documentation for the implementation to determine if other PID registers are implemented.

The suggested PID usage is for PID0 to denote private mappings for a process and for 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 space is mapped at the same virtual address in each process.

Process ID registers (PID0–PID2)

SPR 48 (PID0: PID in Book E); Access: Supervisor-only
SPR 633 PID1
SPR 634 PID2 (PID3–PID14 are currently not assigned to SPR numbers)



2.12.2 MMU control and status register 0 (MMUCSR0)

The MMUCSR0 register is used for general control of the L1 and L2 MMUs.

MMU control and status register 0 (MMUCSR0)

SPR 1012 Access: supervisor read/write

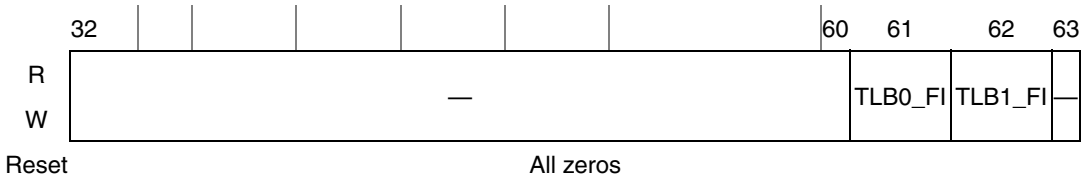


Table 34. MMUCSR0 field descriptions

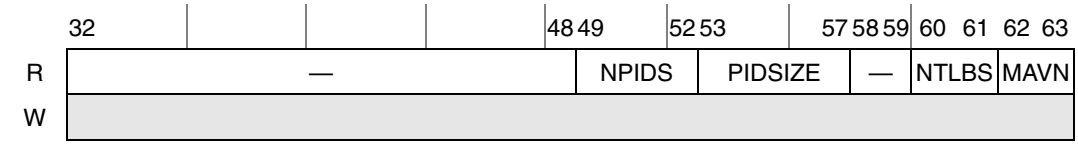
Bits	Name	Description
32–60	—	Reserved, should be cleared.
61	L2TLB0_FI TLB0_FI	TLB0 flash invalidate (write 1 to invalidate) 0 No flash invalidate. Writing a 0 to this bit during an invalidation operation is ignored. 1 TLB0 invalidation operation. Hardware initiates a TLB0 invalidation operation. When this operation is complete, this bit is cleared. Writing a 1 during an invalidation operation causes an undefined operation. If the TLB array supports IPROT, entries that have IPROT set are not invalidated.
62	L2TLB1_FI TLB1_FI	TLB1 flash invalidate (write 1 to invalidate) 0 No flash invalidate. Writing a 0 to this bit during an invalidation operation is ignored. 1 TLB1 invalidation operation. Hardware initiates a TLB1 invalidation operation. When this operation is complete, this bit is cleared. Writing a 1 during an invalidation operation causes an undefined operation. This invalidation typically takes 1 cycle.
63	—	Reserved, should be cleared.

2.12.3 MMU configuration register (MMUCFG)

MMUCFG, shown below, gives configuration information about the implementation's MMU.

MMU configuration register 1 (MMUCFG)

SPR 1015 Access: supervisor read-only



Reset Implementation specific

Table 35. MMUCFG field descriptions

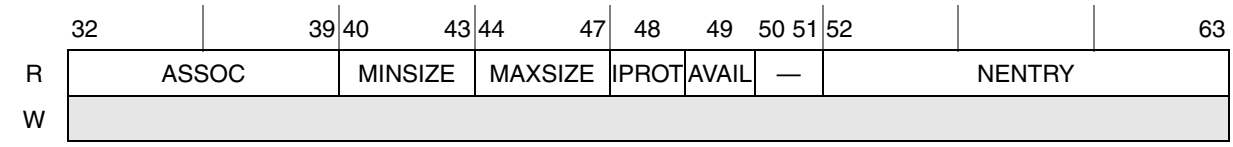
Bits	Name	Description
32–48	—	Reserved, should be cleared.
49–52	NPIDS	Number of PID registers, a 4-bit field that indicates the number of PID registers provided by the processor.
53–57	PIDSIZE	PID register size. The PIDSIZE value is one fewer than the number of bits in each PID register implemented. The processor implements only the least significant PIDSIZE+1 bits in the PID registers.
58–59	—	Reserved, should be cleared.
60–61	NTLBS	Number of TLBs. 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	MAVN	MMU architecture version number. Indicates the version number of the architecture of the MMU implemented by the processor. 00 Version 1.0 01 Reserved 10 Reserved 11 Reserved

2.12.4 TLB configuration registers (TLB*n*CFG)

TLBnCFG registers, shown below, provide information about each specific TLB that is visible to the programming model. TLB0CFG corresponds to TLB0, TLB1CFG corresponds to TLB1, etc.

TLB configuration register *n* (TLB0CFG)

SPR 688 (TLB0CFG) Access: Supervisor read-only
689 (TLB1CFG)



Reset Implementation-specific value

Table 36. TLB*n*CFG field descriptions

Bits	Name	Description
32–39	ASSOC	Associativity of TLB*n*. Number of ways of associativity of TLB array. 0000_0000 Fully associative (A value equal to NENTRY also indicates fully associative.) 0000_0001 1-way set associative 0000_0002 2-way set associative …
40–43	MINSIZE	Minimum page size of TLB*n* 0001 Indicates smallest page size is 4 Kbytes 0002 Indicates smallest page size is 8 Kbytes …
44–47	MAXSIZE	Maximum page size of TLB*n* 0001 Indicates maximum page size is 4 Kbytes 0002 Indicates maximum page size is 8 Kbytes …
48	IPROT	Invalidate protect capability of TLB*n* array. 0 Indicates invalidate protection capability not supported. 1 Indicates invalidate protection capability supported.
49	AVAIL	Page size availability of TLB*n* 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, should be cleared.
52–63	NENTRY	Number of entries in TLB*n*

2.12.5 MMU assist registers (MAS0–MAS7)

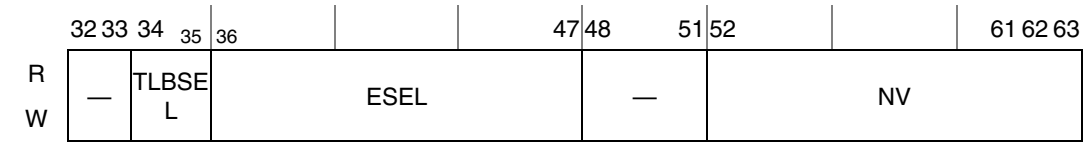
MMU assist registers are defined by the EIS and used by the MMU to manage pages and TLBs. Note that some fields in these registers are redefined by implementations.

MAS register 0 (MAS0)

MAS0, is used for MMU read/write and replacement control.

MAS register 0 (MAS0)

SPR 624 Access: Supervisor read/write



Reset All zeros

Table 37. MAS0 field descriptions

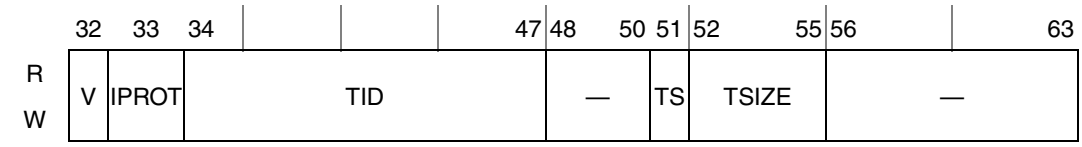
Bits	Name	Comments or function when set
32–33	—	Reserved, should be cleared.
34–35	TLBSEL	Selects TLB for access. 00 TLB0 01 TLB1 10 TLB2 11 TLB3
36–47	ESEL	Entry select. Identifies an entry in the selected array to be used for **tlbwe** and **tlbre**. Valid values for ESEL are from 0 to TLB*n*CFG[ASSOC] - 1. That is, ESEL selects the way from a set of entries determined by MAS3[EPN]. For fully associative TLB arrays, ESEL ranges from 0 to TLB*n*CFG[NENTRY] - 1. ESEL is also updated on TLB error exceptions (misses) and **tlbsx** hit and miss cases.
48–51	—	Reserved, should be cleared.
52–63	NV	Next victim. For those TLBs that support the NV field, provides a hint to software to identify the next victim to be targeted for a TLB miss replacement operation. If the TLB selected by MAS0[TLBSEL] does not support NV, this field is undefined. The computation of NV is implementation-dependent. NV is updated on TLB error exceptions (misses), **tlbsx** hit and miss cases, as shown in *Table 194*, and on execution of **tlbre** if the accessed TLB array supports NV. If NV is updated by a supported TLB array, NV always presents a value that can be used in MAS0[ESEL].

MAS register 1 (MAS1)

Below is the format of MAS1.

MAS register 1 (MAS1) format

SPR 625 Access: Supervisor read/write



Reset All zeros

Table 38. MAS1 field descriptions—descriptor context and configuration control

Bits	Name	Descriptions
32	V	TLB valid bit. 0 This TLB entry is invalid. 1 This TLB entry is valid.
33	IPROT	Invalidate protect. Set to protect this TLB entry from invalidate operations due the execution of **tlbivax**, broadcast invalidations from another processor, or flash invalidations. Note that not all TLB arrays are necessarily protected from invalidation with IPROT. Arrays that support invalidate protection are denoted as such in the TLB configuration registers. 0 Entry is not protected from invalidation. 1 Entry is protected from invalidation.
34–35	—	Reserved, should be cleared.
36–47	TID	Translation identity. 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, should be cleared.
51	TS	Translation space. During translation, TS is compared with AS (MSR[IS] or MSR[DS], depending on the type of access) to select a TLB entry.
52–55	TSIZE	Translation size. Defines the page size of the TLB entry. For TLB arrays that contain fixed-size TLB entries, TSIZE is ignored. For variable page-size TLB arrays, the page size is 4^{TSIZE} Kbytes. TSIZE must be between TLB*n*CFG[MINSIZE] and TLB*n*CFG[MINSIZE]. Note that the EIS standard supports all 16 page sizes defined in Book E. 0001 4 Kbyte 0111 16 Mbyte 0010 16 Kbyte 1000 64 Mbyte 0011 64 Kbyte 1001 256 Mbyte 0100 256 Kbyte 1010 1 Gbyte 0101 1 Mbyte 1011 4 Gbyte 0110 4 Mbyte
56–63	—	Reserved, should be cleared.

MAS register 2 (MAS2)

MAS2, contains fields for specifying the effective page address and the storage attributes for a TLB entry.

MAS register 2 (MAS2)

SPR 626 Access: supervisor read/write

Reset Undefined

Table 39. MAS2 field descriptions—EPN and page attributes

Bits	Name	Description
32–51	EPN	Effective page number. 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 logical address of the page.
52–55	—	Reserved, should be cleared.
56–57	ACM X0	Alternate coherency mode. Allows an implementation to employ multiple coherency methods. If the M attribute (memory coherence required) is not set for a page (M=0), the page has no coherency associated with it and ACM is ignored. If the M attribute is set for a page (M=1), ACM determines the coherency domain (or protocol) used. ACM values are implementation dependent. **Note:** Some previous implementations may have a storage bit in the bit 57 position labeled as X0.
58	VLE X1	VLE mode. Identifies pages which contain instructions from the VLE instruction set. The VLE attribute is only implemented if the processor supports the VLE extension. 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 PowerPC (and associated EIS APUs) instructions. 1 Instructions fetched from the page are decoded and executed as VLE (and associated EIS APUs) instructions.Implementation-dependent page attribute. **Note:** Some implementations have a 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 extension.
59	W	Write-through 0 This page is considered write-back with respect to the caches in the system. 1 All stores performed to this page are written through the caches to main memory.
60	I	Caching-inhibited 0 Accesses to this page are considered cacheable. 1 The page is considered caching-inhibited. All loads and stores to the page bypass the caches and are performed directly to main memory. A read or write to a caching-inhibited page affects only the memory element specified by the operation.
61	M	Memory coherence required 0 Memory coherence is not required. 1 Memory coherence is required. This allows loads and stores to this page to be coherent with loads and stores from other processors (and devices) in the system, assuming all such devices are participating in the coherence protocol.

Table 39. MAS2 field descriptions—EPN and page attributes (continued)

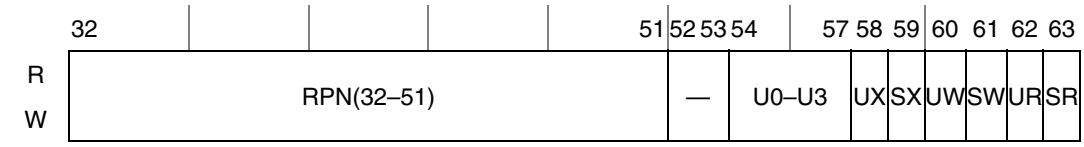
Bits	Name	Description
62	G	Guarded 0 Accesses to this page are not guarded and can be performed before it is known if they are required by the sequential execution model. 1 Loads and stores to this page are performed without speculation (that is, they are known to be required).
63	E	Endianness. Determines endianness for the corresponding page. Little-endian operation is true little endian, which differs from the modified little-endian byte-ordering model optionally available in previous devices that implement the PowerPC architecture. 0 The page is accessed in big-endian byte order. 1 The page is accessed in true little-endian byte order.

MAS register 3 (MAS3)

MAS3 contains fields for specifying the real page address and the permission attributes for a TLB entry.

MAS register 3 (MAS3)

SPR 627 Access: Supervisor read/write



Reset All zeros

Table 40. MAS3 field descriptions–RPN and access control

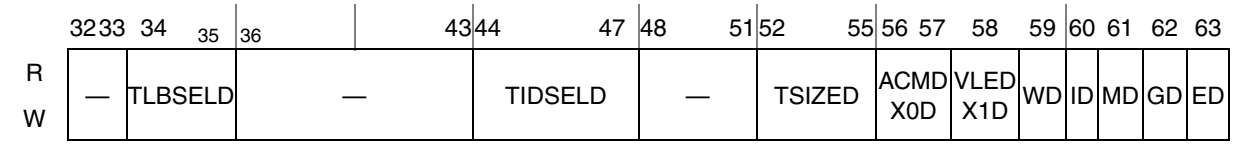
Bits	Name	Description
32–51	RPN[32–51]	Real page number bits 32–51. Depending on page size, only the bits associated with a page boundary are valid. Bits that represent offsets within a page are ignored and should be zero. If the physical address space exceeds 32 bits, RPN[0–31] are accessed through MAS7.
52–53	—	Reserved, should be cleared.
54–57	U0–U3	User bits. Associated with a TLB entry and 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	UX,SX UW,SW UR,SR	Permission bits (UX, SX, UW, SW, UR, SR). User and supervisor read, write, and execute permission bits. Effects of the permission bits are defined by Book E.

MAS register 4 (MAS4)

MAS4,contains fields for specifying default information to be pre-loaded on certain MMU related exceptions.

MAS register 4 (MAS4)

SPR 628 Access: Supervisor read/write



Reset All zeros

The MAS4 fields are described in *Table 41*.

Table 41. MAS4 field descriptions—hardware replacement assist configuration

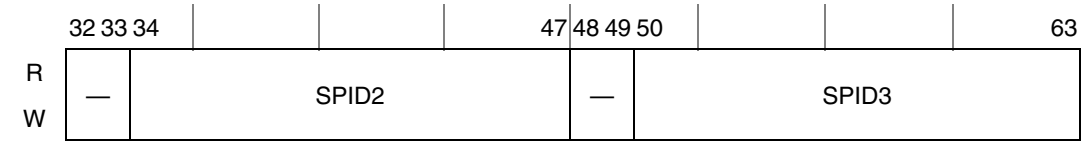
Bits	Name	Description
32–33	—	Reserved, should be cleared.
34–35	TLBSELD	TLBSEL default value. Specifies the default value loaded in MAS0[TLBSEL] on a TLB miss exception.
36–43	—	Reserved, should be cleared.
44–47	TIDSELD	TID default selection value. Specifies which of the current PID registers should be used to load MAS1[TID] on a TLB miss exception. PID registers are addressed as follows: 0000 = PID0 (PID) 0001 = PID1 ... 1110 = PID14 A value that references a non-implemented PID register causes a value of 0 to be placed in MAS1[TID]. See the implementations documentation for a list of supported PIDs.
48–51	—	Reserved, should be cleared.
52–55	TSIZED	Default TSIZE value. Specifies the default value loaded into MAS1[TSIZE] on a TLB miss exception.
56–57	ACMD	Default ACM value Specifies the default value loaded into MAS2[ACM] on a TLB miss exception.
58	VLED	Default VLE value. Specifies the default value loaded into MAS2[VLE] on a TLB miss exception.
59	WD	Default W value. Specifies the default value loaded into MAS2[W] on a TLB miss exception.
60	ID	Default I value. Specifies the default value loaded into MAS2[I] on a TLB miss exception.
61	MD	Default M value. Specifies the default value loaded into MAS2[M] on a TLB miss exception.
62	GD	Default G value. Specifies the default value loaded into MAS2[G] on a TLB miss exception.
63	ED	Default E value. Specifies the default value loaded into MAS2[E] on a TLB miss exception.

MAS register 5 (MAS5)

The optional MAS5 register, contains fields for specifying PID values to be used when searching TLB entries with the **tlbsx** instruction.

MAS register 5 (MAS5)

SPR 629 Access: supervisor read/write



Reset All zeros

Table 42. MAS5 field descriptions—extended search pIDs

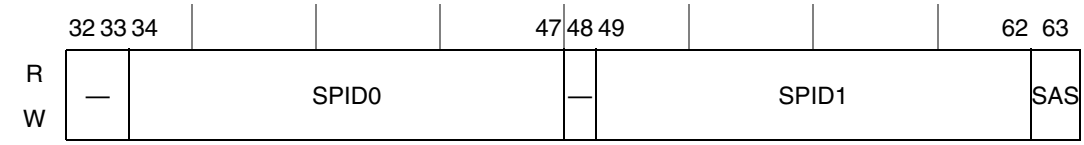
Bits	Name	Description
32–33	—	Reserved, should be cleared.
34–47	SPID2	Search PID2. Specifies the PID2 value used when searching the TLB during execution of **tlbsx**. This field is optional and if implemented is valid for only the number of bits implemented for PID registers.
48–49	—	Reserved, should be cleared.
50–63	SPID3	Search PID3. Specifies the PID3 value used when searching the TLB during execution of **tlbsx**. This field is optional and if implemented is valid for only the number of bits implemented for PID registers.

MAS register 6 (MAS6)

MAS6, contains fields for specifying PID and AS values to be used when searching TLB entries with the **tlbsx** instruction.

MAS register 6 (MAS6)

SPR 630 Access: supervisor read/write



Reset All zeros

Table 43. MAS 6 field descriptions—search pids and search AS

Bits	Name	Description
32–33	—	Reserved, should be cleared.
34–47	SPID0	Search PID0. Specifies the value of PID0 used when searching the TLB during execution of **tlbsx**. SPID0 is valid for only the number of bits implemented for PID registers.
48	—	Reserved, should be cleared.

Table 43. MAS 6 field descriptions—search pids and search AS (continued)

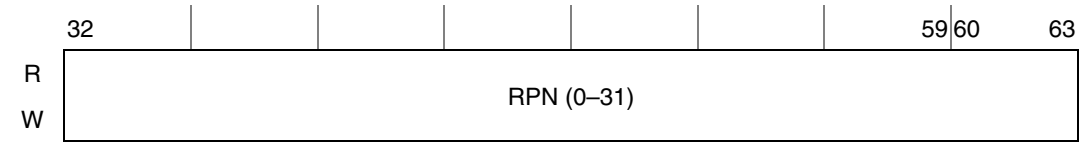
Bits	Name	Description
49–62	SPID1	Search PID1. Specifies the value of PID1 used when searching the TLB during execution of **tlbsx**.SPID1 is optional, and if implemented is valid for only the number of bits implemented for PID registers.
63	SAS	Address space value for searches. Specifies the AS value used when executing **tlbsx** to search the TLB.

MAS register 7 (MAS7)

MAS7, contains the high-order address bits of the RPN only for implementations that support more than 32 bits of physical address.

MAS register 7 (MAS7)

SPR 944 Access: supervisor read/write



Reset All zeros

Table 44. MAS 7 field descriptions—high order RPN

Bits	Name	Description
32–63	RPN[0–31]	Real page number (bits 0–31). RPN[32–63] are accessed through MAS3.

2.13 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.

2.13.1 Debug control registers (DBCR0–DBCR3)

The debug control registers are used to enable debug events, reset the processor, control timer operation during debug events, and set the debug mode of the processor.

Debug control register 0 (DBCR0)

Below is the DBCR0.

Debug control register 0 (DBCR0)

SPR 308 Access: Supervisor-only

	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
R	EDM	IDM	RST		ICMP	BRT	IRPT	TRAP	IAC1	IAC2	IAC3	IAC4	DAC1		DAC2	
W																
Reset	All zeros															

Debug APU

	48	49	56	57	58	59	60	62	63
	RET	—		CIRPT	CRET	VLES	—		FT
Reset	All zeros								

Table 45. DBCR0 field descriptions

Bits	Name	Description
32	EDM	External debug mode. Indicates whether the processor is in external debug mode. 0 The processor is not in external debug mode. 1 The processor is in external debug mode. In some implementations, if EDM = 1, some debug registers are locked and cannot be accessed. Refer to the implementation documentation for any additional implementation-specific behavior.
33	IDM	Internal debug mode. 0 Debug interrupts are disabled. No debug interrupts are taken and debug events are not logged. 1 If MSR[DE] = 1, the occurrence of a debug event or the recording of an earlier debug event in the DBSR when MSR[DE] = 0 or DBCR0[IDM] = 0 causes a debug interrupt. Programming note: Software must clear debug event status in the DBSR in the debug interrupt handler when a debug interrupt is taken before re-enabling interrupts through MSR[DE]. Otherwise, redundant debug interrupts are taken for the same debug event.
34–35	RST	Reset. Book E defines RST such that 00 is always no action and all other settings are implementation 0*x* Default (No action) 1*x* A hard reset is performed on the processor.

Table 45. DBCR0 field descriptions (continued)

Bits	Name	Description
36	ICMP	Instruction completion debug event enable 0 ICMP debug events are disabled. 1 ICMP debug events are enabled. Note: Instruction completion does not cause an ICMP debug event if MSR[DE]=0.
37	BRT	Branch taken debug event enable 0 BRT debug events are disabled. 1 BRT debug events are enabled. Note: Taken branches do not cause a BRT debug event if MSR[DE]=0.
38	IRPT	Interrupt taken debug event enable. 0 IRPT debug events are disabled. 1 IRPT debug events are enabled
39	TRAP	Trap debug event enable 0 TRAP debug events cannot occur. 1 TRAP debug events can occur.
40	IAC1	Instruction address compare 1 debug event enable 0 IAC1 debug events cannot occur. 1 IAC1 debug events can occur.
41	IAC2	Instruction address compare 2 debug event enable. 0 IAC2 debug events cannot occur. 1 IAC2 debug events can occur.
42	IAC3	Defined by Book E as instruction address compare 3 debug event enable 0 IAC3 debug events cannot occur. 1 IAC3 debug events can occur.
43	IAC4	Defined by Book E as instruction address compare 4 debug event enable 0 IAC4 debug events cannot occur. 1 IAC4 debug events can occur.
44–45	DAC1	Data address compare 1 debug event enable 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	DAC2	Data address compare 2 debug event enable 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	RET	Return debug event enable 0 RET debug events cannot occur. 1 RET debug events can occur. Note: An **rfci** does not cause an RET debug event if MSR[DE] = 0 at the time that **rfci** executes.
49–56	—	Reserved, should be cleared.

Table 45. DBCR0 field descriptions (continued)

Bits	Name	Description
57	CIRPT	Debug APU, Critical interrupt taken debug event. A critical interrupt taken debug event occurs when DBCR0[CIRPT] = 1 and a critical interrupt (any interrupt that uses the critical class, that is, uses CSRR0 and CSRR1) occurs. 0 Critical interrupt taken debug events are disabled. 1 Critical interrupt taken debug events are enabled.
58	CRET	Debug APU. Critical interrupt return debug event. 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	VLES	VLE status. (VLE APU). Undefined for IRPT, CIRPT, DEVT[1,2], DCNT[1,2], and UDE events. 0 CRET debug events are disabled. 1 An ICMP, BRT, TRAP, RET, CRET, IAC, or DAC debug event occurred on a VLE instruction.
60–62	—	Reserved
63	FT	Freeze timers on debug event 0 Enable clocking of timers. 1 Disable clocking of timers if any DBSR bit is set (except MRR).

Debug control register 1 (DBCR1)

DBCR1 is shown below.

Debug control register 1 (DBCR1)

SPR 309 Access: supervisor read/write

	32	33	34	35	36	37	38	39	40	41	42	47	48	49	50	51	52	53	54	55	56	57	58	63
R W	IAC1US		IAC1ER		IAC2US		IAC2ER		IAC12M			—	IAC3US		IAC3ER		IAC4US		IAC4ER		IAC34M			—
Reset	All zeros																							

Table 46 provides bit definitions for the DBCR1.

Table 46. DBCR1 field descriptions

Bits	Name	Description
32–33	IAC1US	Instruction address compare 1 user/supervisor mode 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.

Table 46. DBCR1 field descriptions (continued)

Bits	Name	Description
34–35	IAC1ER	Instruction address compare 1 effective/real mode 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	IAC2US	Instruction address compare 2 user/supervisor mode 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	IAC2ER	Instruction address compare 2 effective/real mode 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	IAC12M	Instruction address compare 1/2 mode 00 Exact address compare. IAC1 debug events can occur only if the instruction fetch address equals the value in IAC1. IAC2 debug events can occur only if the instruction fetch address equals the value in IAC2. 01 Address bit match. IAC1 and IAC2 debug events can occur only if the instruction fetch address, ANDed with the contents of IAC2, equals the value in 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 instruction fetch address lies between the values specified in IAC1 and 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 instruction fetch address lies between the values specified in IAC1 and IAC2. If IAC1US≠IAC2US or IAC1ER≠IAC2ER, results are boundedly undefined.
42–47	—	Reserved, should be cleared.
48–49	IAC3US	Instruction address compare 3 user/supervisor mode 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.

Table 46. DBCR1 field descriptions (continued)

Bits	Name	Description
50–51	IAC3ER	Instruction address compare 3 effective/real mode 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	IAC4US	Instruction address compare 4 user/supervisor mode 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	IAC4ER	Instruction address compare 4 effective/real mode 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	IAC34M	Instruction address compare 3/4 mode 00 Exact address compare. IAC3 debug events can occur only if the instruction fetch address equals the value in IAC3. IAC4 debug events can occur only if the instruction fetch address equals the value in IAC4. 01 Address bit match. IAC3 and IAC4 debug events can occur only if the data storage access address, ANDed with the contents of IAC4, equals the value in 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 instruction fetch address lies between the values specified in IAC3 and 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 instruction fetch address lies between the values specified in IAC3 and IAC4. If IAC3US≠IAC4US or IAC3ER≠IAC4ER, results are boundedly undefined.
58–63	—	Reserved, should be cleared.

Debug control register 2 (DBCR2)

DBCR2 is shown below.

Debug control register 2 (DBCR2)

SPR 310 Access: Supervisor read/write

	32 33	34 35	36 37	38 39	40 41	42	43	44 45	46 47	48 55	56 63
R W	DAC1US	D‘AC1ER	DAC2US	DAC2ER	DAC12M	DAC1LNK	DAC2LNK	DVC1M	DVC2M	DVC1BE	DVC2BE
Reset	All zeros										

Table 47. DBCR2 field descriptions

Bits	Name	Description
32–33	DAC1US	Data address compare 1 user/supervisor mode 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	DAC1ER	Data address compare 1 effective/real mode 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	DAC2US	Data address compare 2 user/supervisor mode 00 DAC2 debug events can occur. 01 Reserved 10 DAC2 debug events can occur only if MSR[PR]=0. 11 DAC2 debug events can occur only if MSR[PR]=1.
38–39	DAC2ER	Data address compare 2 effective/real mode 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.

Table 47. DBCR2 field descriptions (continued)

Bits	Name	Description
40–41	DAC12M	Data address compare 1/2 mode 00 Exact address compare. DAC1 debug events can occur only if the data access address equals the value in DAC1. DAC2 debug events can occur only if the data access address equals the value in DAC2. 01 Address bit match. DAC1 and DAC2 debug events can occur only if the data access address, ANDed with the contents of DAC2, equals the value in DAC1, also ANDed with the DAC2 contents. 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 data access address lies between the values specified in DAC1 and 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 data access address lies between the values specified in DAC1 and DAC2. If DAC1US≠DAC2US or DAC1ER≠DAC2ER, results are boundedly undefined.
42	DAC1LNK	Data address compare 1 linked 0 No effect 1 DAC1 debug events are linked to IAC1 debug events. IAC1 debug events do not affect DBSR. When linked to IAC1, DAC1 debug events are conditioned based on whether the instruction also generated an IAC1 debug event.
43	DAC2LNK	Data address compare 2 linked 0 No effect 1 DAC 2 debug events are linked to IAC3 debug events. IAC3 debug events do not affect DBSR. When linked to IAC3, DAC2 debug events are conditioned based on whether the instruction also generated an IAC3 debug event. DAC2 can only be linked if DAC12M specifies exact address compare because DAC2 debug events are not generated in the other compare modes.
44–45	DVC1M	Data value compare 1 mode 00 DAC1 debug events can occur. 01 DAC1 debug events can occur only when all bytes 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 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 in DBCR2[DVC1BE] within at least one of the half words of the data value of the data storage access match their corresponding bytes in DVC1.

Table 47. DBCR2 field descriptions (continued)

Bits	Name	Description
46–47	DVC2M	Data value compare 2 mode 00 DAC2 debug events can occur. 01 DAC2 debug events can occur only when all bytes 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 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 in DBCR2[DVC2BE] within at least one of the half words of the data value of the data storage access match their corresponding bytes in DVC2.
48–55	DVC1BE	Data value compare 1 byte enables. 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	DVC2BE	Data value compare 2 byte enables. Specifies which bytes in the aligned data value being read or written by the storage access are compared to the corresponding bytes in DVC2.

Debug control register 3 (DBCR3)

The debug APU defines the DBCR3, however its contents are implementation specific.

Debug control register 2 (DBCR2)

SPR 561 Access: Supervisor-only

	32							63
R W	Implementation-specific fields							
Reset	Implementation-specific							

2.13.2 Debug status register (DBSR)

The DBSR, provides status debug events information for the most recent processor reset.

Debug status register (DBSR)

SPR: 304 Access: Supervisor: w1c

	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
R	IDE	UDE	MRR		ICMP	BRT	IRPT	TRAP	IAC1	IAC2	IAC3	IAC3	DAC1R	DAC1W	DAC2R	DAC2W
W	w1c	w1c	w1c	w1c	w1c	w1c	w1c	w1c	w1c	w1c	w1c	w1c	w1c	w1c	w1c	w1c
Reset	0	0	undefined		0	0	0	0	0	0	0	0	0	0	0	0

Debug APU

	48	49							56	57	58	59				63
R	RET	—								CIRPT	CRET	—				
W	w1c									w1c	w1c					
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

The DBSR is set through hardware, but is read through software using **mfspr** and cleared by writing ones to them; writing zeros has no effect.

Table 48. DBSR field descriptions

Bits	Name	Description
32	IDE	Imprecise debug event. Set if MSR[DE] = 0 and a debug event causes its respective DBSR bit to be set.
33	UDE	Unconditional debug event. Set if an unconditional debug event occurred. If the $\overline{\text{UDE}}$ signal (level sensitive, active low) is asserted, DBSR[UDE] is affected as follows: MSR[DE] / DBCR0[IDM] / Action X / 0 / No action. 0 / 1 / DBSR[UDE] is set. 1 / 1 / DBSR[UDE] is set and a debug interrupt is taken.
34–35	MRR	Most recent reset. Set when a reset occurs. Undefined at power-up. See the implementation documentation.
36	ICMP	Instruction complete debug event. Set if an instruction completion debug event occurred and DBCR0[ICMP] = 1.
37	BRT	Branch taken debug event. Set if a branch taken debug event occurred (DBCR0[BRT]=1).
38	IRPT	Interrupt taken debug event. Set if an interrupt taken debug event occurred (DBCR0[IRPT]=1).
39	TRAP	Trap instruction debug event. Set if a trap Instruction debug event occurred (DBCR0[TRAP]=1).
40	IAC1	Instruction address compare 1 debug event. Set if an IAC1 debug event occurred (DBCR0[IAC1]=1).
41	IAC2	Instruction address compare 2 debug event. Set if an IAC2 debug event occurred (DBCR0[IAC2]=1).

Table 48. DBSR field descriptions (continued)

Bits	Name	Description
42	IAC3	Instruction address compare 3 debug event. Set if an IAC3 debug event occurred (DBCR0[IAC3]=1).
43	IAC4	Instruction address compare 4 debug event. Set if an IAC4 debug event occurred (DBCR0[IAC4]=1).
44	DAC1R	Data address compare 1 read debug event. Set if a read-type DAC1 debug event occurred (DBCR0[DAC1]=10 or 11).
45	DAC1W	Data address compare 1 write debug event. Set if a write-type DAC1 debug event occurred (DBCR0[DAC1]=01 or 11).
46	DAC2R	Data address compare 2 read debug event.Set if a read-type DAC2 debug event occurred (DBCR0[DAC2]=10 or 11).
47	DAC2W	Data address compare 2 write debug event. Set if a write-type DAC2 debug event occurred (DBCR0[DAC2] =01 or 11).
48	RET	Return debug event. Set if a return debug event occurred (DBCR0[RET]=1).
49–56	—	Reserved, should be cleared.
57	CIRPT	Debug APU. Critical interrupt taken debug event. A critical interrupt taken debug event occurs when DBCR0[CIRPT] = 1 and a critical interrupt (any interrupt that uses the critical class, that is, uses CSRR0 and CSRR1) occurs. 0 No critical interrupt taken debug event has occurred. 1 A critical interrupt taken debug event occurred.
58	CRET	Debug APU. Critical interrupt return debug event. A critical interrupt return debug event occurs when DBCR0[CRET] = 1 and a return from critical interrupt (an **rfci** instruction is executed) occurs. 0 No critical interrupt return debug event has occurred. 1 A critical interrupt return debug event occurred.
59–63	—	Reserved, should be cleared.

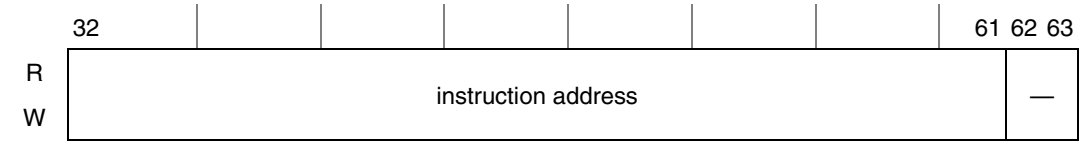
2.13.3 Instruction address compare registers (IAC1–IAC4)

The instruction address compare registers (IAC1–IAC4) are each 64 bits, with bits 62–63 being reserved.

Instruction address compare registers (IAC1–IAC4)

SPR 312 (IAC1)
313 (IAC2)
314 (IAC3)
315 (IAC4)

Access: supervisor read/write



Reset: All zeros

A debug event may be enabled to occur upon an attempt to execute an instruction from an address specified in an IAC, 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. Because all instruction addresses are required to be word-aligned, the two low-order bits of the IACs are reserved and do not participate in the comparison to the instruction address.

2.13.4 Data address compare registers (DAC1–DAC2)

The data address compare registers (DAC1 and DAC2), are each 32 bits. A debug event may be enabled to occur upon loads, stores, or cache operations to an address specified in either 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 DAC2.

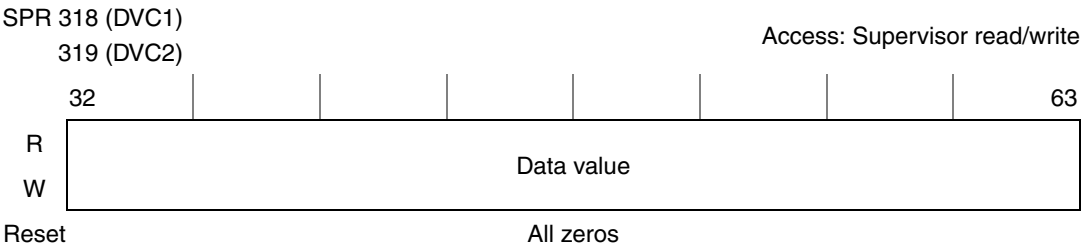
Data address compare registers (DAC1–DAC2)

The contents of DAC1 or DAC2 are compared to the address generated by a data storage access instruction.

2.13.5 Data value compare registers (DVC1 and DVC2)

The data value compare registers (DVC1 and DVC2) are shown below. 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 DVC1 and DVC2. DBCR2[DVC1M] and DBCR2[DVC1BE] control how the contents of DVC1 is compared with the value and DBCR2[DVC2M] and DBCR2[DVC2BE] control how the contents of DVC2 is compared with the value. *Table 47* describes the modes provided.

Data value compare registers (DVC1–DVC2)



2.14 SPE and SPFP APU registers

The SPE and SPFP include the signal processing and embedded floating-point status and control register (SPEFSCR), which is described in *Chapter 2.14.1 on page 119*.", and the SPE implements a 64-bit accumulator, described in *Chapter 2.14.2 on page 122*."

2.14.1 Signal processing, embedded floating-point status, control register (SPEFSCR)

SPEFSCR, is used by the SPE and by the embedded floating-point APUs. Vector floating-point instructions affect both the high element (bits 34-39) and low element floating-point status flags (bits 50–55). Double- and single-precision floating-point instructions affect only the low-element floating-point status flags and leave the high-element floating-point status flags undefined.

Signal processing, embedded floating-point status and control register (SPEFSCR)

SPR: 512 Access: supervisor-only

	High-Word Error Bits										Status Bits					
	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
R W	SOVH	OVH	FGH	FXH	FINVH	FDBZH	FUNFH	FOVFH	—		FINXS	FINVS	FDBZS	FUNFS	FOVFS	MODE
Reset	0	0	undefined		0	0	0	0	0	0	0	0	0	0	0	0

										Enable Bits						
	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
R W	SOV	OV	FG	FX	FINV	FDBZ	FUNF	FOVF	—	FINXE	FINVE	FDBZE	FUNFE	FOVFE	FRMC	
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 49. SPEFSCR field descriptions

Bits	Name	Description
32	SOVH	(SPE APU) Summary integer overflow high. Set when an SPE instruction sets OVH. This is a sticky bit that remains set until it is cleared by an **mtspr** instruction.
33	OVH	(SPE APU) Integer overflow high. Set when an overflow or underflow occurs in the upper word of the result of an SPE instruction.
34	FGH	(FP APUs) Embedded floating-point guard bit high. Used by the floating-point round interrupt handler. FGH is an extension of the low-order bits of the fractional result produced from a floating-point operation on the high word. FGH is zeroed if an overflow, underflow, or invalid input error is detected on the high element of a vector floating-point instruction. Execution of a scalar floating-point instruction leaves FGH undefined.
35	FXH	(SPFP APU) Embedded floating-point inexact bit high. Used by the floating-point round interrupt handler. FXH is an extension of the low-order bits of the fractional result produced from a floating-point operation on the high word. FXH represents the logical OR of all of 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 a vector floating-point instruction. Execution of a scalar floating-point instruction leaves FXH undefined.

Table 49. SPEFSCR field descriptions (continued)

Bits	Name	Description
36	FINVH	(FP APUs) Embedded floating-point invalid operation/input error high. Set under any of the following conditions: Any operand of a high word vector floating-point instruction is Infinity, NaN, or Denorm The operation is a divide and the dividend and divisor are both 0 A conversion to integer or fractional value overflows. Execution of a scalar floating-point instruction leaves FINVH undefined.
37	FDBZH	(FP APUs) Embedded floating-point divide by zero high. Set when a vector floating-point divide instruction is executed with a divisor of 0 in the high word operand and the dividend is a finite non-zero number. Execution of a scalar floating-point instruction leaves FDBZH undefined.
38	FUNFH	(FP APUs) Embedded floating-point underflow high. Set when the execution of a vector floating-point instruction results in an underflow on the high word operation. Execution of a scalar floating-point instruction leaves FUNFH undefined.
39	FOVFH	(FP APUs) Embedded floating-point overflow high. Set when the execution of a vector floating-point instruction results in an overflow on the high word operation. Execution of a scalar floating-point instruction leaves FOVFH undefined.
40–41	—	Reserved, should be cleared.
42	FINXS	(FP APUs) Embedded floating-point inexact sticky flag. Set under the following conditions: – Execution of any scalar or vector floating-point instruction delivers an inexact result for either the low or high element and no floating-point data interrupt is taken for either element – A floating-point instruction results in overflow (FOVF=1 or FOVFH=1), but floating-point overflow exceptions are disabled (FOVFE=0). – A floating-point instruction results in underflow (FUNF=1 or FUNFH=1), but floating-point underflow exceptions are disabled (FUNFE=0), and no floating-point data interrupt occurs. FINXS remains set until it is cleared by software.
43	FINVS	(FP APUs) Embedded floating-point invalid operation sticky flag. The sticky result of any floating-point instruction that causes FINVH or FINV to be set. That is, FINVS <- FINVS | FINV | FINVH. This action may optionally be performed by hardware. To ensure proper operation, software should set this bit on the detection of FINV or FINVH set to one. FINVS remains set until it is cleared by software. (1)
44	FDBZS	(FP APUs) Embedded floating-point divide by zero sticky flag. Set when a floating-point divide instruction sets FDBZH or FDBZ. That is, FDBZS <- FDBZS | FDBZH | FDBZ. FDBZS remains set until it is cleared by software.
45	FUNFS	(FP APUs) Embedded floating-point underflow sticky flag. Defined to be the sticky result of any floating-point instruction that causes FUNFH or FUNF to be set. That is, FUNFS <- FUNFS | FUNF | FUNFH. This action may optionally be performed by hardware. To ensure proper operation, software should set this bit on the detection of FUNF or FUNFH being set. FUNFS remains set until it is cleared by software. 1

Table 49. SPEFSCR field descriptions (continued)

Bits	Name	Description
46	FOVFS	(FP APUs) Embedded floating-point overflow sticky flag. defined to be the sticky result of any floating-point instruction that causes FOVH or FOVF to be set. That is, FOVFS <- FOVFS | FOVF | FOVFH. This action may optionally be performed by hardware. To ensure proper operation, software should set this bit on the detection of FOVF or FOVFH being set. FOVFS remains set until it is cleared by software. [1]
47	MODE	(FP APUs) Embedded floating-point operating mode. Controls the operating mode of the embedded floating-point operations defined in the SPE, and the embedded floating-point APUs. 0 Default hardware results operating mode 1 Reserved.
48	SOV	(SPE APU) Summary integer overflow low. Set when an SPE instruction sets OV. This sticky bit remains set until an **mtspr** writes a 0 to this bit.
49	OV	(SPE APU) Integer overflow low. OV is set when an overflow or underflow occurs in the lower word of the result of an SPE instruction.
50	FG	(FP APUs) Embedded floating-point guard bit (low/scalar) Used by the floating-point round interrupt handler. FG is an extension of the low-order bits of the fractional result produced from a floating-point operation on the low word or any scalar floating-point operation. FG is cleared if an overflow, underflow, or invalid input error is detected on either the low element of a vector floating-point instruction or any scalar floating-point instruction.
51	FX	(FP APUs) Embedded floating-point inexact bit (low/scalar). Used by the floating-point round interrupt handler. FX is an extension of the low-order bits of the fractional result produced from a floating-point operation on the low word or any scalar floating-point instruction. FX represents the logical OR of all of 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 either the low element of a vector floating-point instruction or any scalar floating-point instruction.
52	FINV	(FP APUs) Embedded floating-point invalid operation/input error (low/scalar). Set by the following conditions: – Any operand of a low-word vector or scalar floating-point operation is Infinity, NaN, or Denorm – The operation is a divide and the dividend and divisor are both 0 – A conversion to integer or fractional value overflows
53	FDBZ	(FP APUs) Embedded floating-point divide by zero (low/scalar). Set when a scalar or vector floating-point divide instruction is executed with a divisor of 0 in the low word operand and the dividend is a finite non-zero number.
54	FUNF	(FP APUs) Embedded floating-point underflow (low/scalar). Set when execution of a scalar or vector floating-point instruction results in an underflow on the low word operation.
55	FOVF	(FP APUs) Embedded floating-point overflow (low/scalar). Set when the execution of a scalar or vector floating-point instruction results in an overflow on the low word operation.
56	—	Reserved, should be cleared.

Table 49. SPEFSCR field descriptions (continued)

Bits	Name	Description
57	FINXE	(FP APUs) Embedded floating-point round (inexact) exception enable 0 Exception disabled 1 Exception enabled. A floating-point round interrupt is taken if no other interrupt is taken, and if FG | FGH | FX | FXH (signifying an inexact result) is set as a result of a floating-point operation. If a floating-point instruction operation results in overflow or underflow and the corresponding underflow or overflow exception is disabled, a floating-point round interrupt is taken.
58	FINVE	(FP APUs) Embedded floating-point invalid operation/input error exception enable 0 Exception disabled 1 Exception enabled. A floating-point data interrupt is taken if a floating-point instruction sets FINV or FINVH.
59	FDBZE	(FP APUs) Embedded floating-point divide by zero exception enable 0 Exception disabled 1 Exception enabled. A floating-point data interrupt is taken if a floating-point instruction sets FDBZ or FDBZH.
60	FUNFE	(FP APUs) Embedded floating-point underflow exception enable 0 Exception disabled 1 Exception enabled. A floating-point data interrupt is taken if a floating-point instruction sets FUNF or FUNFH.
61	FOVFE	(FP APUs) Embedded floating-point overflow exception enable 0 Exception disabled 1 Exception enabled. A floating-point data interrupt is taken if a floating-point instruction sets FOVF or FOVFH.
62–63	FRMC	(FP APUs) Embedded floating-point rounding mode control 00 Round to Nearest 01 Round toward Zero 10 Round toward +Infinity. If this mode is not implemented, embedded floating-point round Interrupts are generated for every floating-point instruction for which rounding is indicated. 11 Round toward -Infinity. If this mode is not implemented, embedded floating-point round Interrupts are generated for every floating-point instruction for which rounding is indicated.

1. Software note: Software can detect hardware that manages this sticky bit by performing an operation on a NaN and observing whether hardware sets this sticky bit. In the absence of doing this, if it desired that software written will work on all processors that support embedded floating-point, software should check the appropriate status bits and set the sticky bit itself (if hardware also performs this operation, the action is redundant).

2.14.2 Accumulator (ACC)

The 64-bit architectural accumulator register holds the results of the multiply accumulate (MAC) forms of SPE integer instructions. The accumulator allows back-to-back execution of dependent MAC instructions, something that is found in the inner loops of DSP code such as finite impulse response (FIR) filters. The accumulator is partially visible to the programmer in that its results do not have to be explicitly read to use them. Instead, they are always copied into a 64-bit destination GPR specified as part of the instruction. The

accumulator, however, has to be explicitly cleared when starting a new MAC loop. Based upon the type of instruction, an accumulator can hold either a single 64-bit value or a vector of two 32-bit elements.

The Initialize Accumulator instruction (**evmra**) is provided to initialize the accumulator. This instruction is described in *Chapter 6 on page 330*."

2.15 Alternate time base registers (ATBL and ATBU)

The alternate time base counter (ATB), is formed by concatenating the upper and lower alternate time base registers (ATBU and ATBL). ATBL (SPR 526) provides read-only access to the 64-bit alternate time base counter, which is incremented at an implementation-defined frequency. ATB registers are accessible in both user and supervisor mode.

Like the TB implementation, ATBL is an aliased name for ATB.

Alternate time base register lower (ATBL)

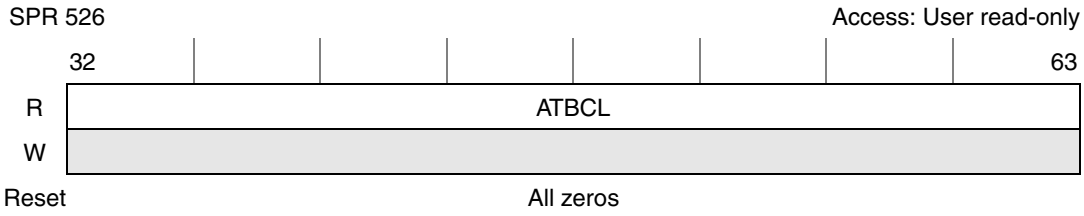


Table 50. ATBL field descriptions

Bits	Name	Description
32–63	ATBCL	Alternate time base counter lower. Lower 32 bits of the alternate time base counter

The ATBU register, provides read-only access to the upper 32 bits of the alternate time base counter. It is accessible in both user and supervisor mode.

Alternate time base register upper (ATBU)

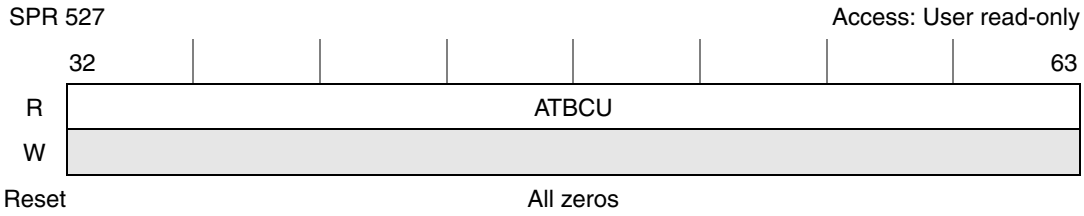


Table 51. ATBU field descriptions

Bits	Name	Description
32–63	ATBCU	Alternate time base counter upper. Upper 32 bits of the alternate time base counter

2.16 Performance monitor registers (PMRs)

The EIS defines a set of register resources used exclusively by the performance monitor. PMRs are similar to the SPRs defined in the Book E architecture and are accessed by **mtpmr** and **mfpmr**, which are also defined by the EIS. *Table 52* lists supervisor-level PMRs. User-level software that attempts to read or write supervisor-level PMRs causes a privilege exception.

Table 52. Performance monitor registers—supervisor level

Abbreviation	Register name	PMR number	pmr[0–4]	pmr[5–9]	Section/page
PMGC0	Performance monitor global control register 0	400	01100	10000	*Chapter 2.16.1*
PMLCa0	Performance monitor local control a0	144	00100	10000	*Chapter 2.16.3*
PMLCa1	Performance monitor local control a1	145	00100	10001	
PMLCa2	Performance monitor local control a2	146	00100	10010	
PMLCa3	Performance monitor local control a3	147	00100	10011	
PMLCb0	Performance monitor local control b0	272	01000	10000	*Chapter 2.16.5*
PMLCb1	Performance monitor local control b1	273	01000	10001	
PMLCb2	Performance monitor local control b2	274	01000	10010	
PMLCb3	Performance monitor local control b3	275	01000	10011	
PMC0	Performance monitor counter 0	16	00000	10000	*Chapter 2.16.7*
PMC1	Performance monitor counter 1	17	00000	10001	
PMC2	Performance monitor counter 2	18	00000	10010	
PMC3	Performance monitor counter 3	19	00000	10011	

User-level PMRs in *Table 53* are read-only and are accessed with **mfpmr**. Attempting to write user-level registers in supervisor or user mode causes an illegal instruction exception.

Table 53. Performance monitor registers—user level (read-only)

Abbreviation	Register name	PMR number	pmr[0–4]	pmr[5–9]	Section/page
UPMGC0	User performance monitor global control register 0	384	01100	00000	*Chapter 2.16.3*
UPMLCa0	User performance monitor local control a0	128	00100	00000	*Chapter 2.16.4*
UPMLCa1	User performance monitor local control a1	129	00100	00001	
UPMLCa2	User performance monitor local control a2	130	00100	00010	
UPMLCa3	User performance monitor local control a3	131	00100	00011	
UPMLCb0	User performance monitor local control b0	256	01000	00000	*Chapter 2.16.6*
UPMLCb1	User performance monitor local control b1	257	01000	00001	
UPMLCb2	User performance monitor local control b2	258	01000	00010	
UPMLCb3	User performance monitor local control b3	259	01000	00011	

Table 53. Performance monitor registers—user level (read-only) (continued)

Abbreviation	Register name	PMR number	pmr[0–4]	pmr[5–9]	Section/page
UPMC0	User performance monitor counter 0	0	00000	00000	*Chapter 2.16.7*
UPMC1	User performance monitor counter 1	1	00000	00001	
UPMC2	User performance monitor counter 2	2	00000	00010	
UPMC3	User performance monitor counter 3	3	00000	00011	

2.16.1 Global control register 0 (PMGC0)

The performance monitor global control register (PMGC0), controls all performance monitor counters.

Performance monitor global control register 0 (PMGC0)/
User performance monitor global control register 0 (UPMGC0)

PMR PMGC0 (PMR400) Access: PMGC0: supervisor-only
UPMGC0 (PMR384) UPMGC0: supervisor/user read-only

	32	33	34	35				50	51	52	53	54	55	56		63
R W	FAC	PMIE	FCECE	—					TBSEL	—			TBEE	—		
Reset	All zeros															

PMGC0 is cleared by a hard reset. Reading this register does not change its contents.

Table 54. PMGC0 field descriptions

Bits	Name	Description
32	FAC	Freeze all counters. When FAC is set by hardware or software, PMLCx[FC] maintains its current value until it is changed by software. 0 The PMCs are incremented (if permitted by other PM control bits). 1 The PMCs are not incremented.
33	PMIE	Performance monitor interrupt enable 0 Performance monitor interrupts are disabled. 1 Performance monitor interrupts are enabled and occur when an enabled condition or event occurs.
34	FCECE	Freeze counters on enabled condition or event 0 The PMCs can be incremented (if permitted by other PM control bits). 1 The PMCs can be incremented (if permitted by other PM control bits) only until an enabled condition or event occurs. When an enabled condition or event occurs, PMGC0[FAC] is set. It is up to software to clear FAC.
35–50	—	Reserved, should be cleared.

Table 54. PMGC0 field descriptions (continued)

Bits	Name	Description
51–52	TBSEL	Time base selector. 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 TB[63] (TBL[31]) 01 TB[55] (TBL[23]) 10 TB[51] (TBL[19]) 11 TB[47] (TBL[15]) Time base transition events can be used to periodically collect information about processor activity. In multiprocessor systems in which TB registers are synchronized among 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 processors in the system. Because the time-base frequency is implementation-dependent, software should invoke a system service program to obtain the frequency before choosing a value for TBSEL.
53–54	—	Reserved, should be cleared.
55	TBEE	Time base transition event exception enable 0 Exceptions from time base transition events are disabled. 1 Exceptions from time base transition events are enabled. A time base transition is signalled to the performance monitor if the TB bit specified in PMGC0[TBSEL] changes from 0 to 1. Time base transition events can be used to freeze the counters (PMGC0[FCECE]) or signal an exception (PMGC0[PMIE]). Changing PMGC0[TBSEL] while PMGC0[TBEE] is enabled may cause a false 0 to 1 transition that signals the specified action (freeze, exception) to occur immediately. Although the interrupt signal condition may occur with MSR[EE] = 0, the interrupt cannot be taken until MSR[EE] = 1.
55–63	—	Reserved, should be cleared.

2.16.2 User global control register 0 (UPMGC0)

The contents of PMGC0 are reflected to UPMGC0, which is read by user-level software. UPMGC0 is read with the **mfpmr** instruction using PMR384.

2.16.3 Local control A registers (PMLCa0–PMLCa3)

The local control A registers 0–3 (PMLCa0–PMLCa3), function as event selectors and give local control for the corresponding performance monitor counters. PMLCa works with the corresponding PMLCb register.

Local control A registers (PMLCa0–PMLCa3)/
User local control A registers (UPMLCa0–UPMLCa3)

PMLCa0 (PMR144) UPMLCa0 (PMR128) Access: PMLCa0–PMLCa3: supervisor-only
PMLCa1 (PMR145) UPMLCa1 (PMR129) UPMLCa0–UPMLCa3: supervisor/user read-only
PMLCa2 (PMR146) UPMLCa2 (PMR130)
PMLCa3 (PMR147) UPMLCa3 (PMR131)

	32	33	34	35	36	37	38	40	41	47	48			63
R W	FC	FCS	FCU	FCM1	FCM0	CE	—		EVENT		—			
Reset	All zeros													

Table 55. PMLCa0–PMLCa3 field descriptions

Bits	Name	Description
32	FC	Freeze counter 0 The PMC is incremented (if permitted by other PM control bits). 1 The PMC is not incremented.
33	FCS	Freeze counter in supervisor state 0 The PMC is incremented (if permitted by other PM control bits). 1 The PMC is not incremented if MSR[PR] = 0.
34	FCU	Freeze counter in user state 0 The PMC is incremented (if permitted by other PM control bits). 1 The PMC is not incremented if MSR[PR] = 1.
35	FCM1	Freeze counter while mark = 1 0 The PMC is incremented (if permitted by other PM control bits). 1 The PMC is not incremented if MSR[PMM] = 1.
36	FCM0	Freeze counter while mark = 0 0 The PMC is incremented (if permitted by other PM control bits). 1 The PMC is not incremented if MSR[PMM] = 0.
37	CE	Condition enable 0 PMC*x* overflow conditions cannot occur. (PMC*x* cannot cause interrupts, cannot freeze counters.) 1 Overflow conditions occur when the most-significant-bit of PMC*x* is equal to one. It is recommended that CE be cleared when counter PMC*x* is selected for chaining.
38–40	—	Reserved, should be cleared.
41–47	EVENT	Event selector. Up to 128 events selectable.
48–63	—	Reserved, should be cleared.

2.16.4 User local control A registers (UPMLCa0–UPMLCa3)

The contents of PMLCa0–PMLCa3 are reflected to UPMLCa0–UPMLCa3, which are read by user-level software with **mfpmr** using PMR numbers in *Table 53*.

2.16.5 Local control B registers (PMLCb0–PMLCb3)

Local control B registers (PMLCb0–PMLCb3), specify a threshold value and a multiple to apply to a threshold event selected for the corresponding performance monitor counter. PMLCb works with the corresponding PMLCa.

Local control B registers (PMLCb0–PMLCb3)/User local control B registers (UPMLCb0–UPMLCb3)

PMR PMR272 (PMLCb0) PMR256 (UPMLCb0)
PMR273 (PMLCb1) PMR257 (UPMLCb1)
PMR274 (PMLCb2) PMR258 (UPMLCb2)
PMR275 (PMLCb3) PMR259 (UPMLCb3)

Access: PMLCb0–PMLCb3 Supervisor read/write
UPMLCb0–UPMLCb3 User read-only

Reset All zeros

Table 56. PMLCb0 –PMLCb3 field descriptions

Bits	Name	Description
32–52	—	Reserved, should be cleared.
53–55	THRESHMUL	Threshold multiple 000 Threshold field is multiplied by 1 (PMLCb*n*[THRESHOLD] * 1) 001 Threshold field is multiplied by 2 (PMLCb*n*[THRESHOLD] * 2) 010 Threshold field is multiplied by 4 (PMLCb*n*[THRESHOLD] * 4) 011 Threshold field is multiplied by 8 (PMLCb*n*[THRESHOLD] * 8) 100 Threshold field is multiplied by 16 (PMLCb*n*[THRESHOLD] * 16) 101 Threshold field is multiplied by 32 (PMLCb*n*[THRESHOLD] * 32) 110 Threshold field is multiplied by 64 (PMLCb*n*[THRESHOLD] * 64) 111 Threshold field is multiplied by 128 (PMLCb*n*[THRESHOLD] * 128)
56–57	—	Reserved, should be cleared.
58–63	THRESHOLD	Threshold. Only events that exceed this value are counted. Events to which a threshold value applies are implementation-dependent as are the dimension (for example duration in cycles) and the granularity with which the threshold value is interpreted. By varying the threshold value, software can profile event characteristics. For example, if PMC1 is configured to count cache misses that last longer than the threshold value, 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.

2.16.6 User local control B registers (UPMLCb0–UPMLCb3)

The contents of PMLCb0–PMLCb3 are reflected to UPMLCb0–UPMLCb3, which are read by user-level software with **mfpmr** using the PMR numbers in *Table 53*.

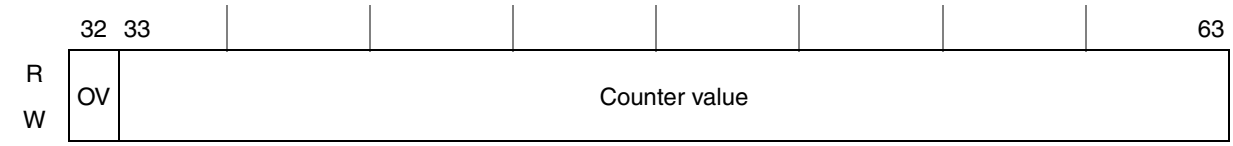
2.16.7 Performance monitor counter registers (PMC0–PMC3)

The performance monitor counter registers PMC0–PMC3, are 32-bit counters that can be programmed to generate interrupt signals when they overflow. Each counter is enabled to count 128 events.

Performance monitor counter registers (PMC0–PMC3)/User performance monitor counter registers (UPMC0–UPMC3)

PMC0 (PMR16) UPMC0 (PMR0)
PMC1 (PMR17) UPMC1 (PMR1)
PMC2 (PMR18) UPMC2 (PMR2)
PMC3 (PMR19) UPMC3 (PMR3)

Access: PMC0–PMC3: Supervisor-only
UPMC0–UPMC3: Supervisor/user read-only



Reset: All zeros

Table 57. PMC0–PMC3 field descriptions

Bits	Name	Description
32	OV	Overflow. When this bit is set, it indicates this counter reaches its maximum value.
33–63	Counter Value	Indicates the number of occurrences of the specified event.

Counters overflow when the high-order bit (the sign bit) becomes set; that is, they reach the value 2,147,483,648 (0x8000_0000). However, an exception is not signaled unless PMGC0[PMIE] and PMLCax[CE] are also set as appropriate.

The interrupts are masked by clearing MSR[EE]. An interrupt that is signaled while MSR[EE] is zero is not taken until MSR[EE] is set. Setting PMGC0[FCECE] forces counters to stop counting when an enabled condition or event occurs.

Software is expected to use **mtpmr** to explicitly set PMCs to non-overflowed values. Setting an overflowed value may cause an erroneous exception. For example, if both PMGC0[PMIE] and PMLCa*x*[CE] are set and the **mtpmr** loads an overflowed value into PMC*x*, an interrupt may be generated without an event counting having taken place.

PMC registers are accessed with **mtpmr** and **mfpmr** using the PMR numbers in *Table 52*.

2.16.8 User performance monitor counter registers (UPMC0–UPMC3)

The contents of PMC0–PMC3 are reflected to UPMC0–UPMC3, which are read by user-level software with the **mfpmr** instruction using the PMR numbers in *Table 53*.

2.17 Device control registers (DCRs)

Book E defines the existence of a DCR address space and the instructions to access them, but does not define particular DCRs. The on-chip DCRs exist architecturally outside the processor core and thus are not part of Book E.

DCRs may control the use of on-chip peripherals, such as memory controllers (specific DCR definitions would be provided in the implementation's user's manual).

The contents of DCR DCRN can be read into a GPR using **mfdcr** **r**D**,**DCRN. GPR contents can be written into DCR DCRN using **mtdcr** DCRN**,r**S.

If DCRs are implemented, they are described as part of the implementation documentation.

2.18 Book E SPR model

This section describes SPR invalid references, synchronization requirements, and preserved, reserved, and allocated registers.

2.18.1 Invalid SPR references

System behavior when an invalid SPR is referenced depends on the privilege level.

- If the invalid SPR is accessible in user mode (SPR[5] = 0), an illegal instruction exception is taken.
- If the invalid SPR is accessible only in supervisor mode (SPR[5] = 1) and the core complex is in supervisor mode (MSR[PR] = 0), the results of the attempted access are boundedly undefined.
- If the invalid SPR address is accessible only in supervisor mode (bit 5 of an SPR number = 1) and the core complex is not in supervisor mode (MSR[PR] = 1), a privilege exception is taken. These results are summarized in *Table 58*.

Table 58. System response to an invalid spr reference

SPR address bit 5	MSR[PR]	Response
0 (User)	x	Illegal exception
1 (Supervisor)	0 (Supervisor)	Boundedly undefined
	1 (User)	Privilege exception

2.18.2 Synchronization requirements for SPRs

Synchronization requirements for accessing certain SPRs are shown in *Table 59*. Except for these SPRs, there are no synchronization requirements for accessing SPRs beyond those stated in Book E. (Note that requirements may be different for different implementations.)

Table 59. Synchronization requirements for sprs

Registers	Instruction	Instruction required before	Instruction required after
DBCR0	**mtspr dbcr0**	None	isync
DBCR1	**mtspr dbcr1**	None	isync

Table 59. Synchronization requirements for sprs (continued)

Registers	Instruction	Instruction required before	Instruction required after
HID0	**mtspr hid0**	None	isync
HID1	**mtspr hid1**	None	isync
L1CSR0	**mtspr l1csr0**	**msync**, **isync**	isync
L1CSR1	**mtspr l1csr1**	None	isync
MAS*n*	**mtspr mas*n***	None	isync
MMUCSR0	**mtspr mmucsr0**	None	isync
PID*n*	**mtspr pid*n***	None	isync
SPEFSCR	**mtspr spefscr**	None	isync

2.18.3 Reserved SPRs

An undefined SPR number in the range 0x000–0x1FF (0–511) that is not preserved is reserved.

2.18.4 Allocated SPRs

SPR numbers allocated for implementation-dependent use are 0x200–0x3FF (512–1023).

Table 60. Allocated SPRs defined by the EIS

SPR	Mnemonic	Register
48	PID0(1)	Process ID register 0. This is not truly an allocated SPR; however, Book E defines only this PID register and refers to it as PID rather than PID0.
512	SPEFSCR	Signal processing and embedded floating-point status and control register
515	L1CFG0	L1 cache configuration register 0
516	L1CFG1	L1 cache configuration register 1
528	IVOR32	SPE APU unavailable exception
529	IVOR33	Embedded floating-point data exception
530	IVOR34	Embedded floating-point round exception
531	IVOR35	Performance monitor Interrupt vector offset register
570	MCSRR0	Machine-check save/restore register 0
571	MCSRR1	Machine-check save/restore register 1
572	MCSR	Machine check syndrome register
573	MCAR	Machine check address register
624	MAS0	MMU assist register 0
625	MAS1	MMU assist register 1
626	MAS2	MMU assist register 2
627	MAS3	MMU assist register 3

Table 60. Allocated SPRs defined by the EIS (continued)

SPR	Mnemonic	Register
628	MAS4	MMU assist register 4
629	MAS5	MMU assist register 5
630	MAS6	MMU assist register 6
633	PID1	Process ID register 1
634	PID2	Process ID register 2
…	PID*n*	Additional PID registers may be defined in this space
688	TLB0CFG	TLB configuration register 0
689	TLB1CFG	TLB configuration register 1
944	MAS7	MMU assist register 7
1008	HID0	Hardware implementation dependent register 0
1009	HID1	Hardware implementation dependent register 1
1010	L1CSR0	L1 cache control and status register 0
1011	L1CSR1	L1 cache control and status register
1012	MMUCSR0	MMU control and status register 0
1015	MMUCFG	MMU configuration register
1023	SVR	System version register

1. An update to a PID register must always be followed by an **isync**.

3 Instruction model

The architecture specifications allow for different processor implementations, which may provide extensions to or deviations from the architectural descriptions. This chapter provides information about the Book E architecture and the Book E implementation standards (EIS), which defines auxiliary processing units (APUs) and other architectural extensions that define additional instructions, registers, and interrupts.
For more information, see *Chapter 7: Auxiliary processing units (APUs) on page 823*."

3.1 Operand conventions

This section describes operand conventions as they are represented in the Book E architecture. These conventions follow the basic descriptions in the classic PowerPC architecture with some changes in terminology. For example, distinctions between user and supervisor-level instructions are maintained, but the designations—UISA, VEA, and OEA—do not apply. Detailed descriptions are provided of conventions used for storing values in registers and memory, accessing processor registers, and representing data in these registers.

3.1.1 Data organization in memory and data transfers

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

Memory operands can be bytes, half words, words, or double words or, for the load/store multiple instruction type and load/store string instructions, a sequence of bytes or words. The address of a memory operand is the address of its first byte (that is, of its lowest-numbered byte). Operand length is implicit for each instruction.

3.1.2 Alignment and misaligned accesses

The operand of a single-register memory access instruction has an alignment boundary equal to its length. An operand's address is misaligned if it is not a multiple of its width.

The concept of alignment is also applied more generally to data in memory. For example, a 12-byte data item is said to be word-aligned if its address is a multiple of four.

Some instructions require their memory operands to have certain alignment. In addition, alignment can affect performance. For single-register memory access instructions, the best performance is obtained when memory operands are aligned.

Instructions are 32 bits (one word) long and must be word-aligned. Note, however, that the VLE extension provides both 16- and 32-bit instructions.
See *VLE instruction alignment and byte ordering on page 217*."

Table 61 lists characteristics for memory operands for single-register memory access instructions.

Table 61. Address characteristics of aligned operands

Operand	Operand Length	Addr[60–63] if Aligned
Byte (or string)	8 bits	xxxx(1)
Half word	2 bytes	xxx0
Word	4 bytes	xx00
Double word	8 bytes	x000

1. An x in an address bit position indicates that the bit can be 0 or 1 independent of the state of other bits in the address.

Note that **lmw**, **stmw**, **lwarx**, and **stwcx.** instructions that are not word aligned cause an alignment exception.

3.2 Instruction set summary

Instructions are divided into the following functional categories:

- Integer instructions—These include arithmetic and logical instructions. See *Integer instructions on page 146*."
- Floating-point instructions—These include floating-point vector and scalar arithmetic instructions. See *Embedded vector and scalar floating-point APU instructions*." Note that some implementations do not support Book E–defined floating-point instructions or registers.
- Load and store instructions—See *Load and store instructions on page 156*."
- Flow control instructions—These include branching instructions, CR logical instructions, trap instructions, and other instructions that affect the instruction flow. See *Branch and flow control instructions on page 163*."
- Processor control instructions—These instructions are used for synchronizing memory accesses. See *Processor control instructions on page 201*."
- Memory synchronization instructions—These instructions are used for memory synchronizing. See *Memory synchronization instructions on page 175*."
- Memory control instructions—These instructions provide control of caches and TLBs. See *Memory control instructions*," and *Supervisor-level memory control instructions*."
- Signal processing instructions—These include a set of vector arithmetic and logic instructions optimized for signal processing. See *Chapter 3.6.1 on page 186*.

Note: Instruction groupings used here do not indicate the execution unit that processes a particular instruction or group of instructions. This information, which is useful for scheduling instructions most effectively, is provided in the execution chapter for the implementation."

Integer instructions operate on word operands. Book E floating-point instructions operate on single-precision and double-precision floating-point operands. The PowerPC architecture uses instructions that are 4 bytes long and word-aligned. It provides for byte, half-word, and word operand loads and stores between memory and a set of 32 general-purpose registers (GPRs). It provides for word and double-word operand loads and stores between memory and a set of 32 floating-point registers (FPRs).

Arithmetic and logical instructions do not read or modify memory. To use the contents of a memory location in a computation and then modify the same or another location, the memory contents must be loaded into a register, modified, and then written to the target location using load and store instructions.

The description of each instruction includes the mnemonic and a formatted list of operands. To simplify assembly language programming, a set of simplified mnemonics and symbols is provided for some of the frequently used instructions; see *Appendix B: Simplified mnemonics for PowerPC instructions on page 1110*," for a complete list of simplified mnemonics. Programs written to be portable across the various assemblers for the PowerPC architecture should not assume the existence of mnemonics not described in that document.

3.2.1 Classes of instructions

Instructions belong to one of the following four classes:

- Defined instructions (See *Defined instruction class on page 135*.)
- Allocated instructions (See *Allocated instruction class on page 136*.)
- Preserved instructions (See *Preserved instruction class on page 137*.)
- Reserved (illegal or no-op) instructions (See *Reserved instruction class on page 138*.)

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

Definition of boundedly undefined

If instructions are encoded with incorrectly set bits in reserved fields, the results on execution can be said to be boundedly undefined. If a user-level program executes the incorrectly coded instruction, the resulting undefined results are bounded in that a spurious change from user to supervisor state is not allowed, and the level of privilege exercised by the program in relation to memory access and other system resources cannot be exceeded. Boundedly undefined results for a given instruction can vary between implementations and between execution attempts in the same implementation.

Defined instruction class

This class of instructions consists of all the instructions defined in Book E. In general, defined instructions are guaranteed to be supported within a Book E system as specified by the architecture, either within the processor implementation itself or within emulation software supported by the system operating software.

For implementations that only provide the 32-bit subset of Book E, emulation of the 64-bit behavior of the defined instructions is not supported. See *Appendix D: Guidelines for 32-bit book E on page 1154*.

Any attempt to execute a defined instruction results in one of the following events:

- An illegal instruction exception-type program interrupt, if an implementation does not recognize the instruction
- An unimplemented instruction exception-type program interrupt, if the instruction is recognized but not supported by the implementation and is not a floating-point instruction
- An unimplemented instruction exception-type program interrupt, if the instruction is recognized but not supported by the implementation, and is a floating-point instruction and floating-point processing is enabled
- The floating-point unavailable interrupt if the instruction is recognized but is not supported by the implementation or is a floating-point instruction and floating-point processing is disabled
- The floating-point unavailable interrupt when floating-point processing is disabled and a floating-point instruction is recognized and is not supported by the implementation
- If an instruction is recognized and supported by the implementation, the processor performs the actions described in the rest of this document. The architected behavior may cause other exceptions.

A defined instruction may be retained by future versions of Book E as a defined instruction, or may be reclassified as a preserved instruction (process of removal from the architecture) and eventually classified as reserved-illegal.

Allocated instruction class

This class of instructions contains the set of instructions (a set of primary opcodes, as well as a set of extended opcodes for certain primary opcodes) used for implementation-specific instructions. *Table 62* lists blocks of opcodes allocated for implementation-dependent use.

Table 62. Allocated instructions

Primary opcode	Extended opcodes
0	All instruction encodings (bits 6–31) except 0x0000_0000(1).
4	All instruction encodings (bits 6–31) SPE and embedded floating-point instructions
19	Extended opcodes (bits 21–30) 0buuuuu_0u11u
31	Extended opcodes (bits 21–30) uuuuu_0u11u
59	Extended opcodes (bits 21–30) uuuuu_0u10u
63	Extended opcodes (bits 21–30) uuuuu_0u10u (except 00000_01100 **frsp**)

1. Instruction encoding 0x0000_0000 is and always will be reserved-illegal.

Allocated instructions are allocated to purposes that are outside the scope of Book E for implementation-dependent and application-specific use.

Any attempt to execute an allocated instruction results in one of the following:

- An illegal instruction exception-type program interrupt, if the instruction is not recognized by the implementation
- An unimplemented instruction exception-type program interrupt, if the instruction is recognized and enabled for execution but the implementation does not support direct

execution of the instruction. This option may be used to allow emulation for unsupported allocated instructions.

- A floating-point unavailable interrupt, if an allocated instruction that extends the floating-point capabilities is recognized and floating-point processing is disabled
- If an allocated instruction is implemented, the processor performs the actions described in the user's manual. Implementation-dependent behavior may cause other exceptions.

An allocated instruction is guaranteed by Book E to remain allocated.

Note: Some allocated instructions may have associated new process state, and, therefore, may provide an associated enable bit, similar in function to MSR[FP] for floating-point instructions. For such instructions, being enabled for execution implies that any associated enable bit is set to allow, or enable, instruction execution. For such instructions, the architecture provides an auxiliary processor unavailable interrupt vector in case execution of such an instruction is attempted when execution is disabled.
For example, MSR[SPE] enables the SPE unavailable interrupt. Other allocated instructions may not have any associated new state and therefore may not require an associated enable bit. If supported by an implementation, such instructions are assumed to be always enabled for execution.

Preserved instruction class

The preserved instruction class supports backward compatibility with the PowerPC architecture. An attempt to execute a preserved instruction results in one of the following:

- If the implementation does not recognize the instruction, an illegal instruction exception-type program interrupt occurs.
- If the instruction is recognized and supported by the implementation, the processor performs the actions defined in the previous version of the architecture.

Future versions of Book E may retain a preserved instruction as a preserved instruction, may reclassify it as an allocated instruction, or may adopt it as part of Book E.

Preserved opcodes are listed in *Table 63*.

Table 63. Preserved instructions

Primary opcode	Extended opcodes
0	No preserved extended opcodes
4	No preserved extended opcodes
19	No preserved extended opcodes
31	Extended opcodes (bits 21–30) 210 0b00110_10010 (**mtsr**) 242 0b00111_10010 (**mtsrin**) 370 0b01011_10010 (**tlbia**) 306 0b01001_10010 (**tlbie**) 371 0b01011_10011 (**mftb**) 595 0b10010_10011 (**mfsr**) 659 0b10100_10011 (**mfsrin**) 310 0b01001_10110 (**eciwx**) 438 0b01101_10110 (**ecowx**)
59	No preserved extended opcodes
63	No preserved extended opcodes

Reserved instruction class

This class of instructions consists of all instruction primary opcodes (and associated extended opcodes, if applicable) that do not belong to either the defined, allocated, or preserved instruction classes.

Reserved instructions are available for future extensions of Book E. That is, some future version of Book E may define any of these instructions to perform new functions or make them available for implementation-dependent use as allocated instructions. There are two types of reserved instructions, reserved-illegal and reserved-nop.

Attempts to execute a reserved-illegal instruction cause an illegal instruction exception-type program interrupt (see *Chapter 4.7.6: Alignment interrupt on page 263*) on implementations conforming to the current version of Book E. Reserved-illegal instructions are, therefore, available for future extensions to Book E that would affect architected state. Such extensions might include new forms of integer or floating-point arithmetic or new forms of load or store instructions that write their result in an architected register.

Attempts to execute a reserved-nop instruction either do not affect implementations conforming to the current version of Book E (that is, treated as a no-operation instruction), or cause an illegal instruction exception-type program interrupt (see *Chapter 4.7.7: Program interrupt on page 265*"). Reserved-nop instructions are available for future architecture extensions that do not affect architected state. Such extensions might include performance-enhancing hints such as new forms of cache touch instructions and could be added while remaining functionally compatible with implementations of previous versions of Book E

A reserved-illegal instruction may be retained by future versions of Book E as a reserved-illegal instruction, may be subsequently reclassified as an allocated instruction, or may even be employed in the role of a subsequently defined instruction.

A reserved-nop instruction may be retained by future versions of Book E as a reserved-nop instruction, may be subsequently reclassified as an allocated instruction, or may even be employed in the role of a subsequently defined instruction that has no effect on architected state.

3.2.2 Instruction forms

This section describes preferred instruction forms, addressing modes, and synchronization.

Preferred instruction forms (no-op)

The Or Immediate (**ori**) instruction has the following preferred form for expressing a no-op:

```
ori 0,0,0
```

Invalid instruction forms

Some of the defined instructions have invalid forms. 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.

Attempts to execute an invalid form of an instruction either causes an illegal instruction type program interrupt or yields boundedly undefined results. Any exceptions to this rule are stated in the instruction descriptions.

Some kinds of invalid form instructions can be deduced just from examining the instruction layout. These are listed below.

- Field shown as reserved but coded as nonzero
- Field shown as containing a particular value but coded as some other value

These invalid forms are not discussed further.

Other invalid instruction forms can be deduced by detecting an invalid encoding of one or more of the instruction operand fields. These kinds of invalid form are identified in the instruction descriptions.

- Branch conditional and branch conditional extended instructions (undefined encoding of BO field)
- Load with update instructions (**r**D = **r**A or **r**A = 0)
- Store with update instructions (**r**A = 0)
- Load multiple instruction (**r**A or **r**B in range of registers to be loaded)
- Load string immediate instructions (**r**A in range of registers to be loaded)
- Load string indexed instructions (**r**D = **r**A or **r**D = **r**B)
- Load/store floating-point with update instructions (**r**A = 0)

3.2.3 Addressing modes

This section describes conventions for addressing memory and for calculating effective addresses (EAs) as defined by the Book E architecture for 32-bit implementations.

Memory addressing

A program references memory using the effective address computed by the processor when it executes a memory access or branch instruction (or other instructions as described in *Chapter : User-level cache instructions on page 180*," and *Chapter : Supervisor-level cache instruction on page 183*," or when it fetches the next sequential instruction.

Memory operands

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

Memory operands may be bytes, half words, words or, for the load/store multiple and load/store string instructions, a sequence of words or bytes. The address of a memory operand is the address of its first byte (that is, of its lowest-numbered byte). Byte ordering can be either big endian or little endian (see *Chapter : Byte ordering on page 141*"). The default byte and bit ordering is big endian.

Operand length is implicit for each instruction with respect to memory alignment. The operand of a scalar memory access 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 memory operand is said to be aligned if it is aligned at its natural boundary; otherwise it is said to be misaligned. For more information about alignment, see *Chapter 3.1.2: Alignment and misaligned accesses on page 133*."

Effective address calculation

The 32-bit address computed by the processor when executing a memory access or branch instruction (or certain other instructions described in *User-level cache instructions on page 180*," *Supervisor-level cache instruction*," and *Supervisor-level tlb management*

instructions on page 183"), or when fetching the next sequential instruction, is called the effective address (EA) and specifies a byte in memory. For a memory access instruction, if the sum of the EA and the operand length exceeds the maximum EA, the memory access is considered to be undefined.

Effective address arithmetic, except for next sequential instruction address computations, wraps around from the maximum address, 2^{32}– 1, to address 0.

Data memory addressing modes

Book E supports the following data memory addressing modes:

- Base+displacement addressing mode—The 16-bit D field is sign-extended and added to the contents of the GPR designated by **r**A or to zero if **r**A = 0. Instructions that use this addressing mode are of the D instruction format.
- Base+index addressing mode—The contents of the GPR designated by **r**B (or the value 0 for **lswi** and **stswi**) are added to the contents of the GPR designated by **r**A or to zero if **r**A = 0. Instructions that use this addressing mode are of the X instruction format.
- Base+displacement extended addressing mode—The 12-bit DE field is sign-extended and added to the contents of the GPR designated by **r**A or to zero if **r**A = 0 to produce the 32-bit EA. Instructions that use this addressing mode are of the DE instruction format.
- Base+displacement extended scaled addressing mode—The 12-bit DES field is concatenated on the right with zeros, sign-extended, and added to the contents of the GPR designated by **r**A or to zero if **r**A = 0 to produce the 32-bit EA. Instructions that use this addressing mode are of the DES instruction format.

In addition, APUs may provide additional addressing modes.

Instruction memory addressing modes

Instruction memory addressing modes correspond with instructions forms, as follows:

- I-form branch instructions—The 24-bit LI field is concatenated on the right with 0b00, sign-extended, and then added to either the address of the branch instruction if AA = 0, or to 0 if AA = 1.
- Taken B-form branch instructions—The 14-bit BD field is concatenated on the right with 0b00, sign-extended, and then added to either the address of the branch instruction if AA = 0, or to 0 if AA = 1.
- Taken XL-form branch instructions—The contents of bits LR[32–61] or CR[32–61] are concatenated on the right with 0b00.
- Sequential instruction fetching (or non-taken branch instructions)—The value 4 is added to the address of the current instruction to form the 32-bit EA of the next instruction. If the address of the current instruction is 0xFFFF_FFFC, the address of the next sequential instruction is undefined.
- Any branch instruction with LK = 1—The value 4 is added to the address of the current instruction and the 32-bit result is placed into the LR. If the address of the current instruction is 0xFFFF_FFFC, the result placed into the LR is undefined.

Although some implementations may support next sequential instruction address computations wrapping from the highest address 0xFFFF_FFFC to 0x0000_0000 as part of the instruction flow, users are strongly encouraged not to depend on this behavior. Doing so can reduce the portability of their software. If code must span this boundary, software should place a non-linking branch at address 0xFFFF_FFFC, which always branches to address

0x0000_0000 (either absolute or relative branches work).
See also *Appendix D: Guidelines for 32-bit book E on page 1154*."

Byte ordering

If scalars (individual data items and instructions) were indivisible, there would be no such concept as byte ordering. It is meaningless to consider the order of bits or groups of bits within the smallest addressable unit of memory, because nothing can be observed about such order. Only when scalars, which the programmer and processor regard as indivisible quantities, can comprise more than one addressable unit of memory does the question of order arise.

For a machine in which the smallest addressable unit of memory is the 64-bit double word, there is no question of the ordering of bytes within double words. All transfers of individual scalars between registers and memory are of double words, and the address of the byte containing the high-order 8 bits of a scalar is no different from the address of a byte containing any other part of the scalar.

For Book E, as for most computer architectures currently implemented, the smallest addressable unit of memory is the 8-bit byte. Many scalars are half words and words (double words in 64-bit implementations) which consist of groups of bytes. When a word-length scalar is moved from a register to memory, the scalar occupies four consecutive byte addresses. It thus becomes meaningful to discuss the order of the byte addresses with respect to the value of the scalar: which byte contains the highest-order eight bits of the scalar, which byte contains the next-highest-order 8 bits, and so on.

Given a scalar that contains multiple bytes, the choice of byte ordering is essentially arbitrary. There are 4! = 24 ways to specify the ordering of 4 bytes within a word but only two of these orderings are sensible:

- The ordering that assigns the lowest address to the highest-order (left-most) 8 bits of the scalar, the next sequential address to the next-highest-order eight bits, and so on. This ordering is called big endian because the big (most-significant) end of the scalar, considered as a binary number, comes first in memory. The 68000 is an example of a processor using this byte ordering.
- The ordering that assigns the lowest address to the lowest-order (right-most) 8 bits of the scalar, the next sequential address to the next-lowest-order eight bits, and so on. This ordering is called little endian because the little (least-significant) end of the scalar, considered as a binary number, comes first in memory. The Intel 8086 is an example of a processor using this byte ordering.

Book E provides support for both big- and little-endian byte ordering in the form of a memory attribute. See *Chapter 5.4.8: Permission attributes on page 315*," and *Chapter 5.2.1: Memory/Cache access attributes on page 283*."

Synchronization requirements

This section describes synchronization requirements for special registers and TLBs. Changing the value in certain system registers and invalidating TLB entries can have the side effect of altering the context in which data addresses and instruction addresses are interpreted, and in which instructions are executed. For example, changing MSR[IS] = 0 to and MSR[IS] = 1 has the side effect of changing address space. Such effects need not occur in program order (program order refers to the execution of instructions in the strict order in which they occur in the program), and therefore may require explicit synchronization by software.

An instruction that alters the context in which data addresses or instruction addresses are interpreted, or in which instructions are executed, is called context altering. This section covers all such context-altering instructions. The required software synchronization for each is shown in *Table 64*.

The notation 'CSI' in the tables means any context-synchronizing instruction (such as, **sc**, **isync**, **rfci**, or **rfi**). A context-synchronizing interrupt (that is, any interrupt except 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.

*Note: Sometimes advantage can be taken of the fact that certain instructions that occur naturally in the program, such as the **rfi/rfci** at the end of an interrupt handler, provide the required synchronization.*

No software synchronization is required before altering the MSR (except perhaps when altering the WE bit: see the tables), because **mtmsr** is execution synchronizing. No software synchronization is required before most of the other alterations shown in the "*Instruction fetch and/or execution*" section in *Table 64*, because all instructions before the context-altering instruction are fetched and decoded before the context-altering instruction executes (the processor must determine whether any of the preceding instructions are context synchronizing)

Table 64 identifies the software synchronization requirements for data access for all context-altering instructions.

Table 64. Synchronization requirements

Context altering instruction or event	Required before	Required after	Notes
Data Accesses			
interrupt	None	None	
mtmsr (DS)	None	CSI	
mtmsr (ME)	None	CSI	(1)
mtmsr (PR)	None	CSI	
mtspr (DAC1, DAC2, DVC1, DVC2)	—	—	(2)
mtspr (DBCR0, DBCR2)	—	—	2
mtspr (DBSR)	—	—	2

Table 64. Synchronization requirements (continued)

Context altering instruction or event	Required before	Required after	Notes
mtspr (PID*n*)	CSI	CSI	
rfci	None	None	
rfi	None	None	
sc	None	None	
tlbivax	CSI	CSI or **msync**	(3),(4)
tlbwe	CSI	CSI or **msync**	3,4
Instruction fetch and/or execution			
Interrupt	None	None	
mtmsr (CE)	None	None	(5)
mtmsr (DE)	None	CSI	
mtmsr (EE)	None	None	3
mtmsr (FE0)	None	CSI	
mtmsr (FE1)	None	CSI	
mtmsr (FP)	None	CSI	
mtmsr (IS)	None	CSI	(6)
mtmsr (ME)	None	CSI	1
mtmsr (PR)	None	CSI	
mtmsr (WE)	—	—	(7)
mtspr (DBCR0, DBCR1)	—	—	2
mtspr (DBSR)	—	—	2
mtspr (DEC)	None	None	(8)
mtspr (IAC1, IAC2, IAC3, IAC4)	—	—	2
mtspr (IVORi)	None	None	
mtspr (IVPR)	None	None	
mtspr (PID)	None	CSI	6
mtspr (TCR)	None	None	8
mtspr (TSR)	None	None	8
rfci	None	None	
rfi	None	None	
sc	None	None	
tlbivax	None	CSI or **msync**	3,4
tlbwe	None	CSI or **msync**	3,4
wrtee	None	None	5
wrteei	None	None	5

1. 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.
2. Synchronization requirements for changing any of the debug registers are implementation-dependent and are specified in the user's manual for the implementation.
3. 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 cause.
   The context synchronizing instruction after the **tlbwe** or **tlbivax** ensures that subsequent storage accesses (data and instruction) use the updated value in any affected TLB entries. It does not ensure that all storage accesses previously translated by the TLB entries being updated have completed with respect to storage; if these completions must be ensured, the tlbwe or tlbivax must be followed by an msync instruction as well as by a context synchronizing instruction.
   The following sequence shows why it is necessary for data accesses to ensure that all storage accesses due to instructions before a 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.
   1. A program issues a load or store to a page.
   2. The same program executes a **tlbwe** or **tlbivax** that invalidates the corresponding TLB entry.
   3. 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.
4. Multiprocessor systems have other requirements to synchronize what is called TLB shoot down' (that is, 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).
5. The effect of changing MSR[EE] or MSR[CE] is immediate.
   If an **mtmsr**, **wrtee**, or **wrteei** clears MSR[EE], an external input, decrementer, or fixed-interval timer interrupt does not occur after the instruction executes.
   If an **mtmsr**, **wrtee**, or **wrteei** 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** executes and before the next instruction is executed in the program that sets MSR[EE].
   If an **mtmsr** clears MSR[CE], a critical input, or watchdog timer interrupt does not occur after the instruction is executed.
   If an **mtmsr** 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 mtmsr executes, and before the next instruction is executed in the program that set MSR[CE].
6. 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.
7. Synchronization requirements for changing the wait state enable are implementation-dependent, and are specified in the user's manual for the implementation.
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.

Context synchronization

An instruction or event is context synchronizing if it satisfies the requirements listed below. Context-synchronizing operations include instructions **isync**, **sc**, **rfi**, **rfci**, **rfdi**, and **rfmci**, and most interrupts.

1. The operation is not initiated or, in the case of **isync**, does not complete until all instructions already in execution have completed to a point at which they have reported all exceptions they cause.
2. The instructions that precede the operation complete execution in the context (including such parameters as privilege level, address space, and memory protection) in which they were initiated.
3. If the operation directly causes an interrupt (for example, **sc** directly causes a system call interrupt) or is an interrupt, the operation is not initiated until no interrupt-causing

exception exists having higher priority than the exception associated with the interrupt. See *Chapter 4.11: Exception priorities on page 278*."

4. The instructions that follow the operation are fetched and executed in the context established by the operation as required by the sequential execution model. (This requirement dictates that any prefetched instructions be discarded and that any effects and side effects of executing them speculatively may also be discarded, except as described in *Memory access ordering on page 290*."

A context-synchronizing operation is necessarily execution synchronizing. Unlike **msync** and **mbar**, such operations do not affect the order of memory accesses with respect to other mechanisms.

Execution synchronization

An instruction is execution synchronizing if it satisfies items 1 and 2 of the definition of context synchronization .**msync** is treated like **isync** with respect to item 1 (that is, the conditions described in item 1 apply to completion of **msync**). Execution synchronizing instructions include **msync**, **mtmsr**, **wrtee**, and **wrteei**. All context-synchronizing instructions are execution synchronizing.

Unlike a context-synchronizing operation, an execution synchronizing instruction need not ensure that the instructions following it execute in the context established by that execution synchronizing instruction. This new context becomes effective sometime after the execution synchronizing instruction completes and before or at a subsequent context-synchronizing operation.

Instruction-related interrupts

Interrupts are caused either directly by the execution of an instruction or by an asynchronous event. In either case, an exception may cause one of several types of interrupts to be invoked.

Examples of interrupts 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 SPR (privileged instruction exception-type program interrupt)
- An attempt by an application program to access an SPR that does not exist (unimplemented operation instruction exception-type program interrupt)
- An attempt by a system program to access an SPR that does not exist (boundedly undefined)
- 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 memory 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 memory with an EA alignment not supported by the implementation (alignment interrupt)
- Execution of a system call instruction (system call interrupt)

- Execution of a **trap** instruction whose trap condition is met (trap type program interrupt)
- Execution of a floating-point instruction when floating-point instructions are unavailable (floating-point unavailable interrupt)
- Execution of a floating-point instruction that causes a floating-point enabled exception to exist (floating-point enabled exception-type program interrupt)
- Execution of a defined instruction that is not implemented by the implementation (illegal instruction exception or unimplemented operation exception-type program interrupt)
- Execution of an allocated instruction that is not implemented by the implementation (illegal instruction exception or unimplemented operation exception-type program interrupt)
- Execution of an allocated instruction when the auxiliary instruction is unavailable (auxiliary processor unavailable interrupt).
- Execution of an allocated instruction that causes an auxiliary enabled exception (enabled exception-type program interrupt).

APUs, such as the SPE, may define additional instruction-caused exceptions and interrupts. 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 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).

Chapter 4: Interrupts and exceptions on page 244 describes interrupt conditions in detail.

3.3 Instruction set overview

This section provides a brief overview of the Book E and Book E instructions.

Note: some instructions have the following optional features:

- CR update—The dot (**.**) suffix on the mnemonic enables the update of the CR.
- Overflow option—The **o** suffix indicates that the overflow bit in the XER is enabled.

3.3.1 Book E user-level instructions

This section discusses the user-level instructions defined in the Book E architecture.

Integer instructions

This section describes the integer instructions. These consist of the following:

- Integer arithmetic instructions
- Integer compare instructions
- Integer logical instructions
- Integer rotate and shift instructions

Integer instructions use the content of the GPRs as source operands and place results into GPRs and the XER and CR fields.

Integer arithmetic instructions

Table 65 lists the integer arithmetic instructions for the PowerPC processors.

Table 65. Integer arithmetic instructions

Name	Mnemonic	Syntax
Add	**add** (**add. addo addo.**)	**r**D,**r**A,**r**B
Add carrying	**addc** (**addc. addco addco.**)	**r**D,**r**A,**r**B
Add extended	**adde** (**adde. addeo addeo.**)	**r**D,**r**A,**r**B
Add immediate	**addi**	**r**D,**r**A,SIMM
Add immediate carrying	**addic**	**r**D,**r**A,SIMM
Add immediate carrying and record	**addic.**	**r**D,**r**A,SIMM
Add immediate shifted	**addis**	**r**D,**r**A,SIMM
Add to minus one extended	**addme** (**addme. addmeo addmeo.**)	**r**D,**r**A
Add to zero extended	**addze** (**addze. addzeo addzeo.**)	**r**D,**r**A
Divide word	**divw** (**divw. divwo divwo.**)	**r**D,**r**A,**r**B
Divide word unsigned	**divwu divwu. divwuo divwuo.**	**r**D,**r**A,**r**B
Multiply high word	**mulhw** (**mulhw.**)	**r**D,**r**A,**r**B
Multiply high word unsigned	**mulhwu** (**mulhwu.**)	**r**D,**r**A,**r**B
Multiply low immediate	**mulli**	**r**D,**r**A,SIMM
Multiply low word	**mullw** (**mullw. mullwo mullwo.**)	**r**D,**r**A,**r**B
Negate	**neg** (**neg. nego nego.**)	**r**D,**r**A
Subtract from	**subf** (**subf. subfo subfo.**)	**r**D,**r**A,**r**B
Subtract from carrying	**subfc** (**subfc. subfco subfco.**)	**r**D,**r**A,**r**B
Subtract from extended	**subfe** (**subfe. subfeo subfeo.**)	**r**D,**r**A,**r**B
Subtract from immediate carrying	**subfic**	**r**D,**r**A,SIMM
Subtract from minus one extended	**subfme** (**subfme. subfmeo subfmeo.**)	**r**D,**r**A
Subtract from zero extended	**subfze** (**subfze. subfzeo subfzeo.**)	**r**D,**r**A

Although there is no subtract immediate instruction, its effect can be achieved by using an **addi** instruction with the immediate operand negated. Simplified mnemonics are provided that include this negation. Subtract instructions subtract the second operand (**r**A) from the third operand (**r**B). Simplified mnemonics are provided in which the third operand is subtracted from the second. See *Appendix B: Simplified mnemonics for PowerPC instructions on page 1110*," for examples.

According to Book E, an implementation that executes instructions with the overflow exception enable bit (OE) set or that sets the carry bit (CA) can either execute these instructions slowly or prevent execution of the subsequent instruction until the operation completes. The summary overflow (SO) and overflow (OV) bits in the XER are set to reflect an overflow condition of a 32-bit result only if the instruction's OE bit is set.

Integer compare instructions

The integer compare instructions algebraically or logically compare the contents of register **r**A with either the zero-extended value of the UIMM operand, the sign-extended value of the SIMM operand, or the contents of **r**B. The comparison is signed for **cmpi** and **cmp** and

unsigned for **cmpli** and **cmpl**. *Table 66* lists integer compare instructions. Note that the L bit must be 0 for 32-bit implementations.

Table 66. Integer 32-Bit compare instructions (L = 0)

Name	Mnemonic	Syntax
Compare	**cmp**	**cr**D,L,**r**A,**r**B
Compare immediate	**cmpi**	**cr**D,L,**r**A,SIMM
Compare logical	**cmpl**	**cr**D,L,**r**A,**r**B
Compare logical immediate	**cmpli**	**cr**D,L,**r**A,UIMM

The **cr**D operand can be omitted if the result of the comparison is to be placed in CR0. Otherwise the target CR field must be specified in **cr**D by using an explicit field number.

For information on simplified mnemonics for the integer compare instructions see *Appendix B: Simplified mnemonics for PowerPC instructions on page 1110*."

Integer logical instructions

The logical instructions shown in *Table 67* perform bit-parallel operations on the specified operands. Logical instructions with the CR updating enabled (uses dot suffix) and instructions **andi.** and **andis.** set CR field CR0 to characterize the result of the logical operation. Logical instructions do not affect XER[SO], XER[OV], or XER[CA].

See *Appendix B*," for simplified mnemonic examples for integer logical operations.

Table 67. Integer logical instructions

Name	Mnemonic	Syntax	Implementation notes
AND	and (and.)	**r**A,**r**S,**r**B	—
AND Immediate	andi**.**	**r**A,**r**S,UIMM	—
AND Immediate Shifted	andis**.**	**r**A,**r**S,UIMM	—
AND with Complement	**andc** (**andc.**)	**r**A,**r**S,**r**B	—
Count Leading Zeros Word	**cntlzw (cntlzw.**)	**r**A,**r**S	—
Equivalent	**eqv** (**eqv.**)	**r**A,**r**S,**r**B	—
Extend Sign Byte	**extsb** (**extsb.**)	**r**A,**r**S	—
Extend Sign Half Word	**extsh** (**extsh.**)	**r**A,**r**S	—
NAND	**nand** (**nand.**)	**r**A,**r**S,**r**B	—
NOR	**nor** (**nor.**)	**r**A,**r**S,**r**B	—
OR	**or** (**or.**)	**r**A,**r**S,**r**B	—
OR Immediate	ori	**r**A,**r**S,UIMM	Book E defines **ori r0,r0,0** as the preferred form for a no-op. The dispatcher may discard this instruction and dispatch it only to the completion queue but not to any execution unit.

Table 67. Integer logical instructions (continued)

Name	Mnemonic	Syntax	Implementation notes
OR Immediate Shifted	oris	**r**A,**r**S,UIMM	—
OR with Complement	orc (**orc.**)	**r**A,**r**S,**r**B	—
XOR	**xor** (**xor.**)	**r**A,**r**S,**r**B	—
XOR Immediate	xori	**r**A,**r**S,UIMM	—
XOR Immediate Shifted	xoris	**r**A,**r**S,UIMM	—

Integer rotate and shift instructions

Rotation operations are performed on data from a GPR, and the result, or a portion of the result, is returned to a GPR. Integer rotate instructions, summarized in *Table 68*, rotate the contents of a register. The result is either inserted into the target register under control of a mask (if a mask bit is set the associated bit of the rotated data is placed into the target register, and if the mask bit is cleared the associated bit in the target register is unchanged) or ANDed with a mask before being placed into the target register. *Appendix B: Simplified mnemonics for PowerPC instructions on page 1110*," lists simplified mnemonics that allow simpler coding of often used functions such as clearing the left- or right-most bits of a register, left or right justifying an arbitrary field, and simple rotates and shifts.

Table 68. Integer rotate instructions

Name	Mnemonic	Syntax
Rotate left word Immediate then AND with mask	**rlwinm** (**rlwinm.**)	**r**A,**r**S,SH,MB,ME
Rotate left word then AND with mask	**rlwnm** (**rlwnm.**)	**r**A,**r**S,**r**B,MB,ME
Rotate left word Immediate then mask insert	**rlwimi** (**rlwimi.**)	**r**A,**r**S,SH,MB,ME

The integer shift instructions (*Table 69*) perform left and right shifts. Immediate-form logical (unsigned) shift operations are obtained by specifying masks and shift values for certain rotate instructions. Simplified mnemonics (shown in *Appendix B: Simplified mnemonics for PowerPC instructions*") are provided to simplify coding of such shifts.

Multiple-precision shifts can be programmed as shown in *C.2: Multiple-precision shifts on page 1148*." The integer shift instructions are summarized in *Table 69*.

Table 69. Integer shift instructions

Name	Mnemonic	Syntax
Shift Left Word	slw (slw.)	**r**A,**r**S,**r**B
Shift Right Word	srw (srw.)	**r**A,**r**S,**r**B
Shift Right Algebraic Word Immediate	srawi (srawi.)	**r**A,**r**S,SH
Shift Right Algebraic Word	sraw (sraw.)	**r**A,**r**S,**r**B

Floating-point instructions

This section describes the floating-point instructions as they are defined by Book E.

The rules followed in assigning new primary and extended opcodes.

- Primary opcode 63 is used for the double-precision arithmetic instructions as well as miscellaneous instructions (for example, FPSCR manipulation instructions). Primary opcode 59 is used for the single-precision arithmetic instructions.
- The single-precision instructions for which there is a corresponding double-precision instruction have the same format and extended opcode as that double-precision instruction.
- In assigning new extended opcodes for primary opcode 63, the following regularities are maintained. In addition, all new X-form instructions in primary opcode 63 have bits 21–22 = 11.
  - Bit 26 = 1 if and only if the instruction is A-form.
  - Bits 26–29 = 0b0000 if and only if the instruction is a comparison or **mcrfs** (if and only if the instruction sets an explicitly designated CR field).
  - Bits 26–28 = 0b001 if and only if the instruction explicitly refers to or sets the FPSCR (that is, is an FPSCR instruction) and is not **mcrfs**.
  - Bits 26–30 = 0b01000 if and only if the instruction is a move register instruction, or any other instruction that does not refer to or set the FPSCR.
- In assigning extended opcodes for primary opcode 59, the following regularities have been maintained. They are based on those rules for primary opcode 63 that apply to the instructions having primary opcode 59. In particular, primary opcode 59 has no FPSCR instructions, so the corresponding rule does not apply.
  - If there is a corresponding instruction with primary opcode 63, its extended opcode is used.
  - Bit 26 = 1 if and only if the instruction is A form.
  - Bits 26–30 = 0b01000 if and only if the instruction is a move register instruction, or any other instruction that does not refer to or set the FPSCR.

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 memory.

Normalized Operand

```
if WORD1:8 > 0 and WORD1:8 < 255 then
  FPR(frD)0:1 ← WORD0:1
  FPR(frD)2 ← ¬WORD1
  FPR(frD)3 ← ¬WORD1
  FPR(frD)4 ← ¬WORD1
  FPR(frD)5:63 ← WORD2:31 || 29 0
```

Denormalized Operand

```
if WORD1:8 = 0 and WORD9:31 ≠ 0 then
  sign ← WORD0
  exp ← -126
  frac0:52 ← 0b0 || WORD9:31 || 29 0
  normalize the operand
    do while frac0 = 0
      frac ← frac1:52 || 0b0
```

```
        exp ← exp - 1
   FPR(frD)0 ← sign
   FPR(frD)1:11 ← exp + 1023
   FPR(frD)12:63 ← frac1:52
```

Zero/Infinity/NaN

```
if WORD1:8 = 255 or WORD1:31 = 0 then
   FPR(frD)0:1 ← WORD0:1
   FPR(frD)2 ← WORD1
   FPR(frD)3 ← WORD1
   FPR(frD)4 ← WORD1
   FPR(frD)5:63 ← WORD2:31 || 29 0
```

For double-precision load floating-point instructions, conversion is not required because the data from memory is copied directly into the FPR.

Many floating-point load instructions have an update form, in which GPR(**r**A) is updated with the EA. For these forms, if **r**A≠0 and **r**A≠**r**D, the EA is placed into GPR(**r**A) and the memory element (byte, half word, word, or double word) addressed by EA is loaded into FPR(**r**D). If **r**A=0 or **r**A=**r**D, the instruction form is invalid.

Floating-point load accesses cause a data storage interrupt if the program is not allowed to read the location. Floating-point load memory accesses cause a data TLB error interrupt if the program attempts to access memory that is unavailable. The floating-point load instruction set is shown in *Table 70*.

Table 70. Floating-point load instruction set

Instruction	Mnemonic	Syntax
Load Floating-Point Double	**lfd**	**fr**D,D(**r**A)
Load Floating-Point Double with Update	**lfdu**	**fr**D,D(**r**A)
Load Floating-Point Double Extended	**lfde**	**fr**D,DES(**r**A)
Load Floating-Point Double with Update Extended	**lfdue**	**fr**D,DES(**r**A)
Load Floating-Point Double Indexed	**lfdx**	**fr**D,**r**A,**r**B
Load Floating-Point Double with Update Indexed	**lfdux**	**fr**D,**r**A,**r**B
Load Floating-Point Double Indexed Extended	**lfdxe**	**fr**D,**r**A,**r**B
Load Floating-Point Double with Update Indexed Extended	**lfduxe**	**fr**D,**r**A,**r**B
Load Floating-Point Single	**lfs**	**fr**D,D(**r**A)
Load Floating-Point Single with Update	**lfsu**	**fr**D,D(**r**A)
Load Floating-Point Single Extended	**lfse**	**fr**D,DES(**r**A)
Load Floating-Point Single with Update Extended	**lfsue**	**fr**D,DES(**r**A)
Load Floating-Point Single Indexed	**lfsx**	**fr**D,**r**A,**r**B
Load Floating-Point Single with Update Indexed	**lfsux**	**fr**D,**r**A,**r**B
Load Floating-Point Single Indexed Extended	**lfsxe**	**fr**D,**r**A,**r**B
Load Floating-Point Single with Update Indexed Extended	**lfsuxe**	**fr**D,**r**A,**r**B

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 optional store floating-point as integer word instruction (**stfiwx**), described in *Chapter 6: Instruction set on page 330*." 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-precision format before storing the operand. The conversion steps are as follows.

Let $WORD_{0:31}$ be the word in memory written to.

```
No Denormalization Required (includes Zero / Infinity / NaN)
if FPR(FRS)1:11 > 896 or FPR(FRS)1:63 = 0 then
   WORD0:1 ← FPR(FRS)0:1
   WORD2:31 ← FPR(FRS)5:34

Denormalization Required
if 874 ≤ FRS1:11 ≤ 896 then
   sign ← FPR(FRS)0
   exp ← FPR(FRS)1:11 - 1023
   frac ← 0b1 || FPR(FRS)12:63
   denormalize operand
      do while exp < -126
         frac ← 0b0 || frac0:62
         exp ← exp + 1
   WORD0 ← sign
   WORD1:8 ← 0x00
   WORD9:31 ← frac1:23
else WORD ← undefined
```

Note that if the value to be stored by a single-precision store floating-point instruction exceeds the maximum number representable in single-precision 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 (that is, the result of a single-precision load floating-point from WORD does 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 is copied directly into memory.

Many floating-point store instructions have an update form, in which GPR(**r**A) is updated with the EA. For these forms, if **r**A≠0, the EA is placed into GPR(**r**A).

Floating-point store accesses cause a data storage interrupt if the program is not allowed to write to the location. Integer store accesses cause a data TLB error interrupt if the program attempts to access memory that is unavailable. Store instructions are shown in *Table 71*.

Book E supports both big-endian and little-endian byte ordering.

Table 71. Floating-point store instructions

Instruction	Mnemonic	Syntax
Store floating-point double	**stfd**	**fr**S,D(**r**A)
Store floating-point double with update	**stfdu**	**fr**S,D(**r**A)
Store floating-point double extended	**stfde**	**fr**S,DES(**r**A)
Store floating-point double with update extended	**stfdue**	**fr**S,DES(**r**A)

Table 71. Floating-point store instructions (continued)

Instruction	Mnemonic	Syntax
Store floating-point double indexed	**stfdx**	**fr**S,**r**A,**r**B
Store floating-point double with update indexed	**stfdux**	**fr**S,**r**A,**r**B
Store floating-point double indexed extended	**stfdxe**	**fr**S,**r**A,**r**B
Store floating-point double with update indexed extended	**stfduxe**	**fr**S,**r**A,**r**B
Store floating-point as integer word indexed	**stfiwx**	**fr**S,**r**A,**r**B
Store floating-point as integer word indexed extended	**stfiwxe**	**fr**S,**r**A,**r**B
Store floating-point single	**stfs**	**fr**S,D(**r**A)
Store floating-point single with update	**stfsu**	**fr**S,D(**r**A)
Store floating-point single extended	**stfse**	**fr**S,DES(**r**A)
Store floating-point single with update extended	**stfsue**	**fr**S,DES(**r**A)
Store floating-point single indexed	**stfsx**	**fr**S,**r**A,**r**B
Store floating-point single with update indexed	**stfsux**	**fr**S,**r**A,**r**B
Store floating-point single indexed extended	**stfsxe**	**fr**S,**r**A,**r**B
Store floating-point single with update indexed extended	**stfsuxe**	**fr**S,**r**A,**r**B

Floating-point move instructions

Described in *Table 72*, these instructions copy data from one FPR 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 (for example, the sign bit of a NaN may be altered by **fneg**, **fabs**, and **fnabs**). These instructions do not alter the FPSCR.

Table 72. Floating-point move instructions

Instruction	Mnemonic	Syntax
Floating Absolute Value	**fabs**[.]	**fr**D,**fr**B
Floating Move Register	**fmr**[.]	**fr**D,**fr**B
Floating Negative Absolute Value	**fnabs**[.]	**fr**D,**fr**B
Floating Negate	**fneg**[.]	**fr**D,**fr**B

Floating-point arithmetic instructions

The following sections describe elementary arithmetic, multiply-add, rounding/conversion, compare, and status/control instructions.

Floating-point elementary arithmetic instructions

Table 73 lists mnemonics and syntax of floating-point elementary arithmetic instructions.

Table 73. Floating-point elementary arithmetic instructions

Instruction	Mnemonic	Syntax
Floating add	**fadd**[.]	**fr**D,**fr**A,**fr**B
Floating add single	**fadds**[.]	**fr**D,**fr**A,**fr**B
Floating divide	**fdiv**[.]	**fr**D,**fr**A,**fr**B
Floating divide single	**fdivs**[.]	**fr**D,**fr**A,**fr**B
Floating multiply	**fmul**[.]	**fr**D,**fr**A,**fr**C
Floating multiply single	**fmuls**[.]	**fr**D,**fr**A,**fr**C
Floating reciprocal estimate single	**fres**[.]	**fr**D,**fr**B
Floating reciprocal square root estimate	**frsqrte**[.]	**fr**D,**fr**B
Floating square root	**fsqrt**[.]	**fr**D,**fr**B
Floating square root single	**fsqrts**[.]	**fr**D,**fr**B
Floating subtract	**fsub**[.]	**fr**D,**fr**A,**fr**B
Floating subtract single	**fsubs**[.]	**fr**D,**fr**A,**fr**B

Floating-point multiply-add instructions

These instructions combine a multiply and an add operation without an intermediate rounding operation. FPSCR status bits, described in *Table 74* are set as follows:

- Overflow, underflow, and inexact exception bits, the FR, FI, and FPRF fields are set based on the final result of the operation, 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, any of the following actions will cause appropriate exception bits to be set:
  - Multiplication of infinity by 0
  - Multiplication of anything by an SNaN
  - Addition of anything with an SNaN

Table 74. Floating-point multiply-add instructions

Instruction	Mnemonic	Instruction
Floating Multiply-Add	**fmadd**[.]	**fr**D,**fr**A,**fr**B,**fr**C
Floating Multiply-Add Single	**fmadds**[.]	**fr**D,**fr**A,**fr**B,**fr**C
Floating Multiply-Subtract	**fmsub**[.]	**fr**D,**fr**A,**fr**B,**fr**C
Floating Multiply-Subtract Single	**fmsubs**[.]	**fr**D,**fr**A,**fr**B,**fr**C
Floating Negative Multiply-Add	**fnmadd**[.]	**fr**D,**fr**A,**fr**B,**fr**C

Table 74. Floating-point multiply-add instructions (continued)

Instruction	Mnemonic	Instruction
Floating Negative Multiply-Add Single	**fnmadds**[.]	**fr**D,**fr**A,**fr**B,**fr**C
Floating Negative Multiply-Subtract	**fnmsub**[.]	**fr**D,**fr**A,**fr**B,**fr**C
Floating Negative Multiply-Subtract Single	**fnmsubs**[.]	**fr**D,**fr**A,**fr**B,**fr**C

Floating-point rounding and conversion instructions

Table 75. Floating-point rounding and conversion instructions

Instruction	Mnemonic	Syntax
Floating Convert from Integer Double Word	**fcfid**	**fr**D,**fr**B
Floating Convert to Integer Double Word	**fctid**	**fr**D,**fr**B
Floating Convert to Integer Double word and round to Zero	**fctidz**	**fr**D,**fr**B
Floating Convert to Integer Word	**fctiw**[.]	**fr**D,**fr**B
Floating Convert to Integer Word and Round to Zero	**fctiwz**[.]	**fr**D,**fr**B
Floating Round to Single-Precision	**frsp**[.]	**fr**D,**fr**B

Floating-point compare instructions

The floating-point compare instructions compare the contents of two FPRs. Comparison ignores the sign of zero (that is, regards +0 as equal to –0). The comparison result can be ordered or unordered. The comparison sets one bit in the designated CR field and clears the other three. The floating-point condition code, FPSCR[FPCC], is set in the same way.

The CR field and the FPCC are set as described in *Table 76*.

Table 76. CR field settings

Bit	Name	Description
0	FL	(**fr**A) < (**fr**B)
1	FG	(**fr**A) > (**fr**B)
2	FE	(**fr**A) = (**fr**B)
3	FU	(**fr**A) ? (**fr**B) (unordered)

The floating-point compare and select instruction set is shown in *Table 77*.

Table 77. Floating-point compare and select instructions

Instruction	Mnemonic	Syntax
Floating Compare Ordered	**fcmpo**	**cr**D,**fr**A,**fr**B
Floating Compare Unordered	**fcmpu**	**cr**D,**fr**A,**fr**B
Floating Select	**fsel** **fsel.**	**fr**D,**fr**A,**fr**B,**fr**C **fr**D,**fr**A,**fr**B,**fr**C

Floating-point status and control register instructions

Every FPSCR instruction synchronizes the effects of all floating-point instructions executed by a given processor. Executing a FPSCR instruction ensures that all floating-point instructions previously initiated by the given processor have completed before the FPSCR instruction is initiated, and that no subsequent floating-point instructions are initiated by the given processor until the FPSCR instruction completes. In particular:

- All exceptions caused by the previously initiated instructions are recorded in the FPSCR before the FPSCR instruction is initiated.
- All invocations of floating-point enabled exception-type program interrupt that will be caused by the previously initiated instructions have occurred before the FPSCR instruction is initiated.
- No subsequent floating-point instruction that depends on or alters the settings of any FPSCR bits is initiated until the FPSCR instruction has completed.

Floating-point load and floating-point store instructions (*Table 78*) are not affected.

Table 78. Floating-point status and control register instructions

Instruction	Mnemonic	Syntax
Move from FPSCR	**mffs** **mffs.**	**fr**D **fr**D
Move to FPSCR Bit 0	**mtfsb0** **mtfsb0.**	**crb**D **crb**D
Move to FPSCR Bit 1	**mtfsb1** **mtfsb1.**	**crb**D **crb**D
Move to FPSCR Fields	**mtfsf** **mtfsf.**	FM,**fr**B FM,**fr**B
Move to FPSCR Field Immediate	**mtfsfi** **mtfsfi.**	**cr**D,IMM **cr**D,IMM

Load and store instructions

Load and store instructions are issued and translated in program order; however, the accesses can occur out of order. Synchronizing instructions are provided to enforce strict ordering. The following load and store instructions are defined:

- Integer load instructions
- Integer store instructions
- Integer load and store with byte-reverse instructions
- Integer load and store multiple instructions
- Memory synchronization instructions
- SPE APU load and store instructions for reading and writing 64-bit GPRs. Some of these instructions are also implemented by processors that support the embedded vector single-precision and embedded scalar double-precision floating-point APUs, which use the extended 64-bit GPRs. See *Chapter 3.6.1 on page 186*."

Self-modifying code

When a processor modifies any memory location that can contain an instruction, software must ensure that the instruction cache is made consistent with data memory and that the

modifications are made visible to the instruction fetching mechanism. This must be done even if the cache is disabled or if the page is marked caching-inhibited.

The following instruction sequence can be used to accomplish this when the instructions being modified are in memory that is memory-coherence required and one processor both modifies the instructions and executes them. (Additional synchronization is needed when one processor modifies instructions that another processor will execute.)

The following sequence synchronizes the instruction stream (using either **dcbst** or **dcbf**):

```
dcbst (or dcbf)|update memory
msync  |wait for update
icbi   |remove (invalidate) copy in instruction cache
msync  |ensure the ICBI invalidate is complete
isync  |remove copy in own instruction buffer
```

These operations are required because the data cache is a write-back cache. Because instruction fetching bypasses the data cache, changes to items in the data cache cannot be reflected in memory until the fetch operations complete. The **msync** after the **icbi** is required to ensure that the **icbi** invalidation has completed in the instruction cache.

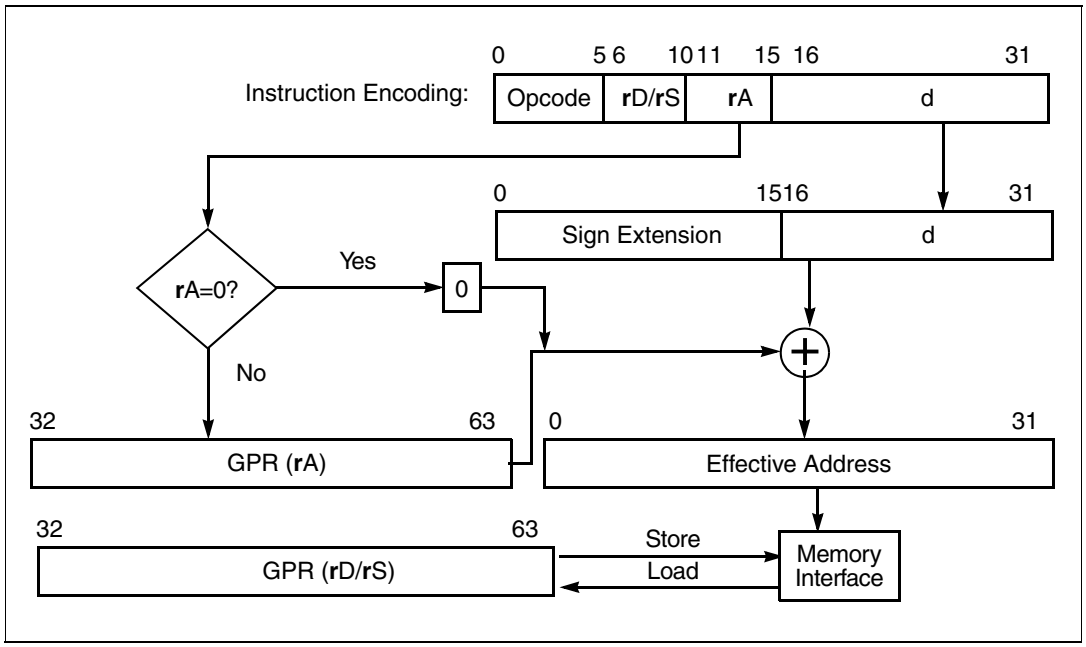
Special care must be taken to avoid coherency paradoxes in systems that implement unified secondary caches, and designers should carefully follow the guidelines for maintaining cache coherency discussed in the user's manual.

Integer load and store address generation

Integer load and store operations generate EAs using register indirect with immediate index mode, register indirect with index mode, or register indirect mode, which are described as follows:

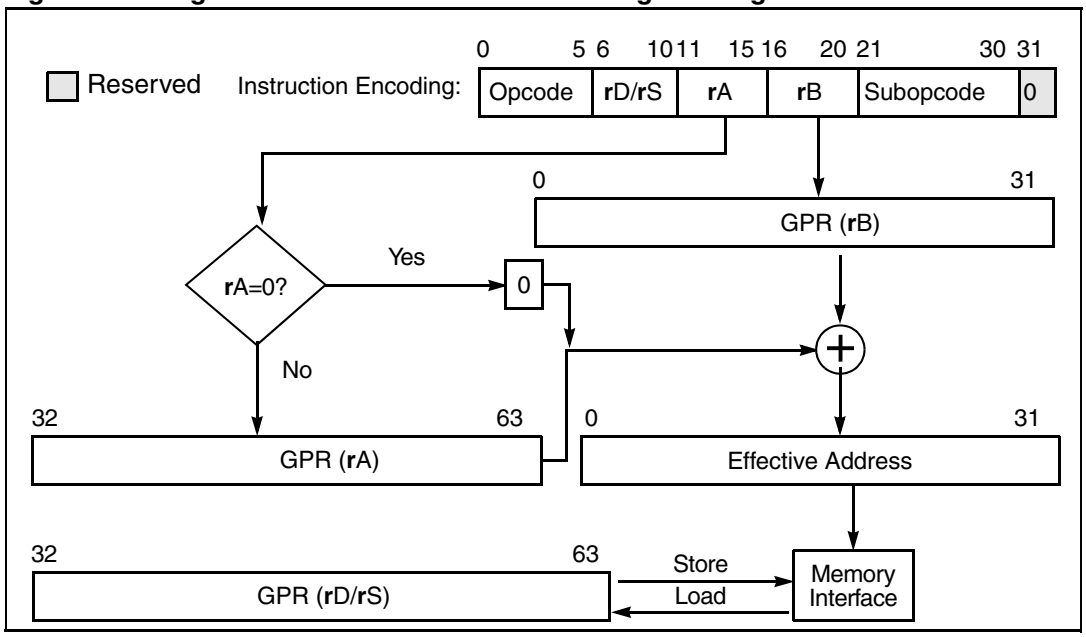
- Register indirect with immediate index addressing for integer loads and stores. Instructions using this addressing mode contain a signed 16-bit immediate index (d operand), which is sign extended and added to the contents of a general-purpose register specified in the instruction (**r**A operand), to generate the EA. If **r0** is specified, a value of zero is added to the immediate index (d operand) in place of the contents of **r0**. The option to specify **r**A or 0 is shown in the instruction descriptions as (**r**A|0). *Figure 6* shows how an EA is generated using this mode.

Figure 6. Register indirect with immediate index addressing for integer loads/stores



- Register indirect with index addressing for integer loads and stores. Instructions using this mode cause the contents of two GPRs (specified as operands **r**A and **r**B) to be added in the EA generation. A zero in place of the **r**A operand causes a zero to be added to the GPR contents specified in operand **r**B. The option to specify **r**A or 0 is shown in the instruction descriptions as (**r**A|0). *Figure 7* shows how an EA is generated using this mode.

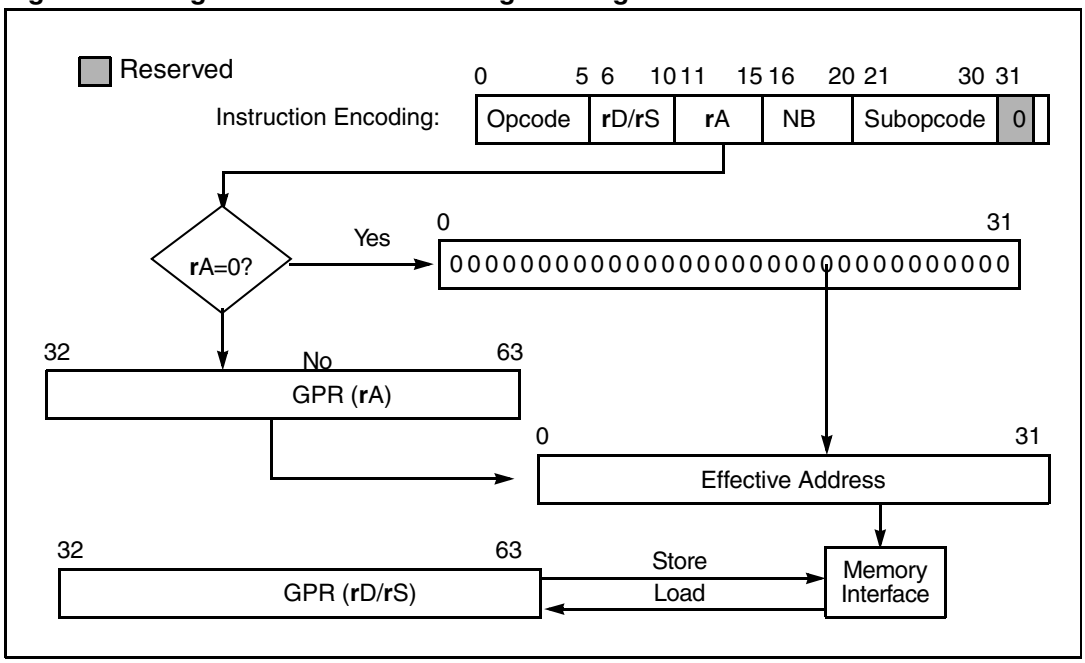
Figure 7. Register indirect with index addressing for integer loads/stores



- Register indirect addressing for integer loads and stores. Instructions using this addressing mode use the contents of the GPR specified by the **r**A operand as the EA.

A zero in the **r**A operand generates an EA of zero. The option to specify **r**A or 0 is shown in the instruction descriptions as (**r**A|0). *Figure 8* shows how an EA is generated using this mode.

Figure 8. Register indirect addressing for integer loads/stores



See *Effective address calculation on page 139*," for information about calculating EAs. Note that in some implementations, operations that are not naturally aligned can suffer performance degradation. *Chapter 4.7.6: Alignment interrupt on page 263*, for additional information about load and store address alignment interrupts.

Register indirect integer load instructions

For integer load instructions, the byte, half word, or word addressed by the EA is loaded into **r**D. Many integer load instructions have an update form, in which **r**A is updated with the generated EA. For these forms, if **r**A ≠ 0 and **r**A ≠ **r**D (otherwise invalid), the EA is placed into **r**A and the memory element (byte, half word, or word) addressed by the EA is loaded into **r**D. Note that the Book E architecture defines load with update instructions with operand **r**A = 0 or **r**A = **r**D as invalid forms.

Integer load instructions

Table 79. Integer load instructions

Name	Mnemonic	Syntax
Load Byte and Zero	lbz	**r**D,**d(r**A**)**
Load Byte and Zero Indexed	lbzx	**r**D,**r**A,**r**B
Load Byte and Zero with Update	lbzu	**r**D,**d(r**A**)**
Load Byte and Zero with Update Indexed	lbzux	**r**D,**r**A,**r**B
Load Half Word and Zero	lhz	**r**D,**d(r**A**)**
Load Half Word and Zero Indexed	lhzx	**r**D,**r**A,**r**B

Table 79. Integer load instructions (continued)

Name	Mnemonic	Syntax
Load Half Word and Zero with Update	lhzu	**r**D,**d(r**A**)**
Load Half Word and Zero with Update Indexed	lhzux	**r**D,**r**A,**r**B
Load Half Word Algebraic	lha	**r**D,**d(r**A**)**
Load Half Word Algebraic Indexed	lhax	**r**D,**r**A,**r**B
Load Half Word Algebraic with Update	lhau	**r**D,**d(r**A**)**
Load Half Word Algebraic with Update Indexed	lhaux	**r**D,**r**A,**r**B
Load Word and Zero	lwz	**r**D,**d(r**A**)**
Load Word and Zero Indexed	lwzx	**r**D,**r**A,**r**B
Load Word and Zero with Update	lwzu	**r**D,**d(r**A**)**
Load Word and Zero with Update Indexed	lwzux	**r**D,**r**A,**r**B

Integer store instructions

For integer store instructions, the **r**S contents are stored into the byte, half word, word or double word in memory addressed by the EA. Many store instructions have an update form in which **r**A is updated with the EA. For these forms, the following rules apply:

- If **r**A ≠ 0, the EA is placed into **r**A.
- If **r**S = **r**A, the contents of register **r**S are copied to the target memory element and the generated EA is placed into **r**A (**r**S).

The Book E architecture defines store with update instructions with **r**A = 0 as an invalid form. In addition, it defines integer store instructions with the CR update option enabled (Rc field, bit 31, in the instruction encoding = 1) to be an invalid form. *Table 80* summarizes integer store instructions.

Table 80. Integer store instructions

Name	Mnemonic	Syntax
Store Byte	stb	**r**S,**d(r**A**)**
Store Byte Indexed	stbx	**r**S,**r**A,**r**B
Store Byte with Update	stbu	**r**S,**d(r**A**)**
Store Byte with Update Indexed	stbux	**r**S,**r**A,**r**B
Store Half Word	sth	**r**S,**d(r**A**)**
Store Half Word Indexed	sthx	**r**S,**r**A,**r**B
Store Half Word with Update	sthu	**r**S,**d(r**A**)**
Store Half Word with Update Indexed	sthux	**r**S,**r**A,**r**B
Store Word	stw	**r**S,**d(r**A**)**
Store Word Indexed	stwx	**r**S,**r**A,**r**B
Store Word with Update	stwu	**r**S,**d(r**A**)**
Store Word with Update Indexed	stwux	**r**S,**r**A,**r**B

Integer load and store with byte-reverse instructions

Table 81 describes integer load and store with byte-reverse instructions. These books were defined in part to support the original PowerPC definition of little-endian byte ordering. Note that Book E supports true little endian on a per-page basis. For more information, see *Byte ordering on page 141*."

Table 81. Integer load and store with byte-reverse instructions

Name	Mnemonic	Syntax
Load Half Word Byte-Reverse Indexed	lhbrx	**r**D,**r**A,**r**B
Load Word Byte-Reverse Indexed	lwbrx	**r**D,**r**A,**r**B
Store Half Word Byte-Reverse Indexed	sthbrx	**r**S,**r**A,**r**B
Store Word Byte-Reverse Indexed	stwbrx	**r**S,**r**A,**r**B

Integer load and store multiple instructions

The load/store multiple instructions are used to move blocks of data to and from the GPRs. The load multiple and store multiple instructions can have operands that require memory accesses crossing a 4-Kbyte page boundary. As a result, these instructions can be interrupted by a data storage interrupt associated with the address translation of the second page.

Note: If one of these instructions is interrupted, it may be restarted, requiring multiple memory accesses.

The Book E architecture defines the Load Multiple Word (**lmw**) instruction (*Table 82*) with **r**A in the range of registers to be loaded as an invalid form. Load and store multiple accesses must be word aligned; otherwise, they cause an alignment exception.

Table 82. Integer load and store multiple instructions

Name	Mnemonic	Syntax
Load Multiple Word	lmw	**r**D,**d(r**A**)**
Store Multiple Word	stmw	**r**S,**d(r**A**)**

Integer load and store string instructions

The integer load and store string instructions allow movement of data from memory to registers or from registers to memory without concern for alignment. These instructions can be used for a short move between arbitrary memory locations or to initiate a long move between misaligned memory fields. However, in some implementations, these instructions are likely to have greater latency and take longer to execute, perhaps much longer, than a sequence of individual load or store instructions that produce the same results.

Table 83 summarizes the integer load and store string instructions.

Table 83. Integer load and store string instructions

Name	Mnemonic	Syntax
Load String Word Immediate	lswi	**r**D,**r**A,NB
Load String Word Indexed	lswx	**r**D,**r**A,**r**B
Store String Word Immediate	stswi	**r**S,**r**A,NB
Store String Word Indexed	stswx	**r**S,**r**A,**r**B

Load string and store string instructions can involve operands that are not word-aligned.

Floating-point load and store address generation

Floating-point load and store operations, listed in *Table 84*, generate EAs using the register indirect with immediate index addressing mode and register indirect with index addressing mode. Floating-point loads and stores are not supported for direct-store accesses. The use of floating-point loads and stores for direct-store accesses results in an alignment interrupt.

There are two forms of the floating-point load instruction—single-precision and double-precision operand formats. Because the FPRs support only the floating-point double-precision format, single-precision floating-point load instructions convert single-precision data to double-precision format before loading an operand into an FPR.

The floating-point load and store indexed instructions (**lfsx**, **lfsux**, **lfdx**, **lfdux**, **stfsx**, **stfsux**, **stfdx**, and **stfdux**) are invalid when the Rc bit is one.

The PowerPC architecture defines load with update with **r**A = 0 as an invalid form.

Table 84. Floating-point load instructions

Name	Mnemonic	Syntax
Load Floating-Point Single	lfs	**fr**D,**d(r**A**)**
Load Floating-Point Single Indexed	lfsx	**fr**D,**r**A,**r**B
Load Floating-Point Single with Update	lfsu	**fr**D,**d(r**A**)**
Load Floating-Point Single with Update Indexed	lfsux	**fr**D,**r**A,**r**B
Load Floating-Point Double	lfd	**fr**D,**d(r**A**)**
Load Floating-Point Double Indexed	lfdx	**fr**D,**r**A,**r**B
Load Floating-Point Double with Update	lfdu	**fr**D,**d(r**A**)**
Load Floating-Point Double with Update Indexed	lfdux	**fr**D,**r**A,**r**B

Floating-point store instructions

This section describes floating-point store instructions. There are three basic forms of the store instruction—single-precision, double-precision, and integer. The integer form is supported by the optional **stfiwx** instruction. Because the FPRs support only double-precision format for floating-point data, single-precision floating-point store instructions convert double-precision data to single-precision format before storing the operands. *Table 85* summarizes the floating-point store instructions.

Table 85. Floating-point store instructions

Name	Mnemonic	Syntax
Store Floating-Point Single	stfs	**fr**S,**d(r**A**)**
Store Floating-Point Single Indexed	stfsx	**fr**S,**r** B
Store Floating-Point Single with Update	stfsu	**fr**S,**d(**rA**)**
Store Floating-Point Single with Update Indexed	stfsux	**fr**S,**r** B
Store Floating-Point Double	stfd	**fr**S,**d(r**A**)**
Store Floating-Point Double Indexed	stfdx	**fr**S,**r**B

Table 85. Floating-point store instructions (continued)

Name	Mnemonic	Syntax
Store Floating-Point Double with Update	stfdu	**fr**S,**d(**rA**)**
Store Floating-Point Double with Update Indexed	stfdux	**fr**S,**r**B
Store Floating-Point as Integer Word Indexed(1)	stfiwx	**fr**S,**r**B

1. The **stfiwx** instruction is optional to the Book E architecture.

Some floating-point store instructions require conversions in the LSU. *Table 86* shows conversions the LSU makes when executing a Store Floating-Point Single instruction.

Table 86. Store floating-point single behavior

FPR Precision	Data Type	Action
Single	Normalized	Store
Single	Denormalized	Store
Single	Zero, infinity, QNaN	Store
Single	SNaN	Store
Double	Normalized	If (exp ≤ 896) then denormalize and store, else store
Double	Denormalized	Store zero
Double	Zero, infinity, QNaN	Store
Double	SNaN	Store

Table 87 shows the conversions made when performing a Store Floating-Point Double instruction. Most entries in the table indicate that the floating-point value is simply stored. Only in a few cases are any other actions taken.

Table 87. Store floating-point double behavior

FPR Precision	Data Type	Action
Single	Normalized	Store
Single	Denormalized	Normalize and store
Single	Zero, infinity, QNaN	Store
Single	SNaN	Store
Double	Normalized	Store
Double	Denormalized	Store
Double	Zero, infinity, QNaN	Store
Double	SNaN	Store

Branch and flow control instructions

Some branch instructions can redirect instruction execution conditionally based on the value of bits in the CR.

Branch instruction address calculation

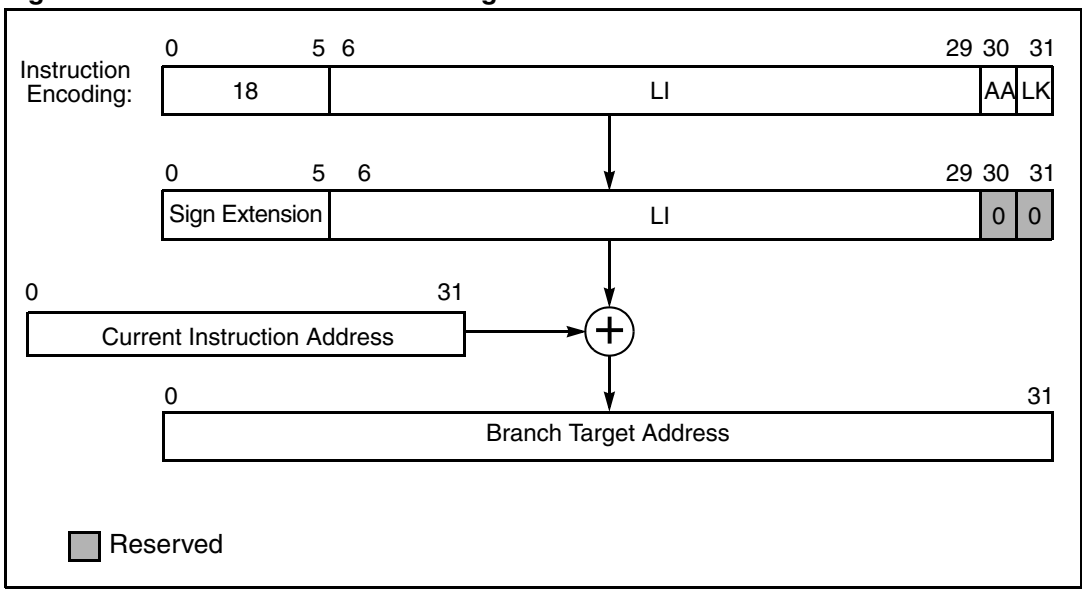
Branch instructions can alter the sequence of instruction execution. Instruction addresses are always assumed to be word aligned; the Book E processors ignore the two low-order bits of the generated branch target address. Branch instructions compute the EA of the next instruction address using the following addressing modes:

- Branch relative
- Branch conditional to relative address
- Branch to absolute address
- Branch conditional to absolute address
- Branch conditional to link register (LR)
- Branch conditional to count register (CTR)

Branch relative addressing mode

Instructions that use branch relative addressing generate the next instruction address by sign extending and appending 0b00 to the immediate displacement operand LI, and adding the resultant value to the current instruction address. Branches using this mode have the absolute addressing option disabled (AA field, bit 30, in the instruction encoding = 0). The LR update option can be enabled (LK field, bit 31, in the instruction encoding = 1). This causes the EA of the instruction following the branch instruction to be placed in the LR. *Figure 9* shows how the branch target address is generated using this mode.

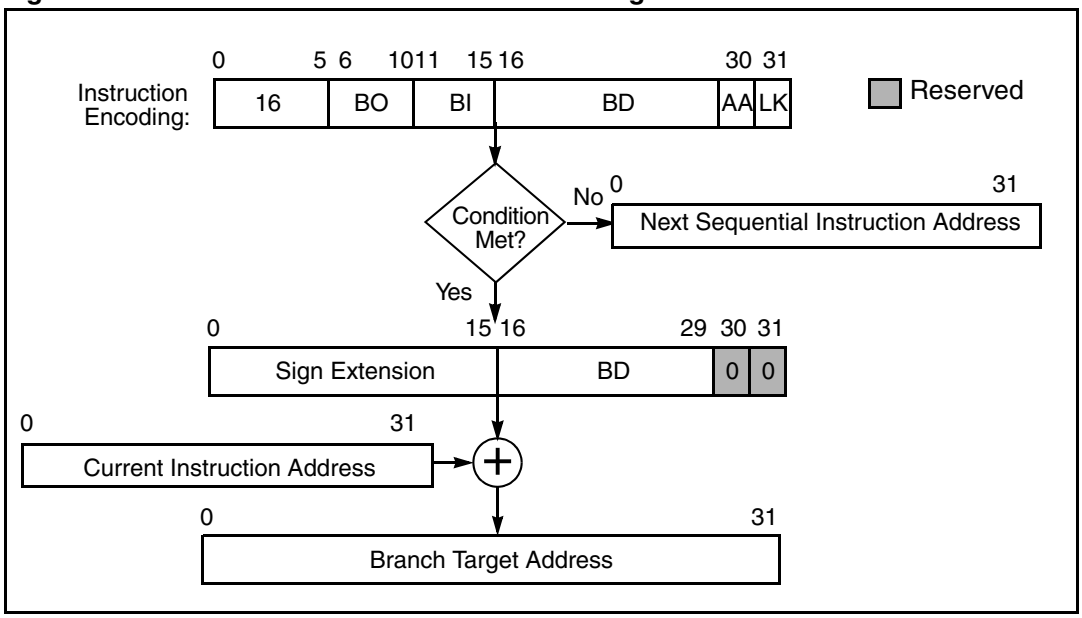
Figure 9. Branch relative addressing



Branch conditional to relative addressing mode

If branch conditions are met, instructions that use the branch conditional to relative addressing mode generate the next instruction address by sign extending and appending results to the immediate displacement operand (BD) and adding the resultant value to the current instruction address. Branches using this mode have the absolute addressing option disabled (AA field, bit 30, in the instruction encoding = 0). The LR update option can be enabled (LK field, bit 31, in the instruction encoding = 1). This option causes the EA of the instruction following the branch instruction to be placed in the LR. *Figure 10* shows how the branch target address is generated using this mode.

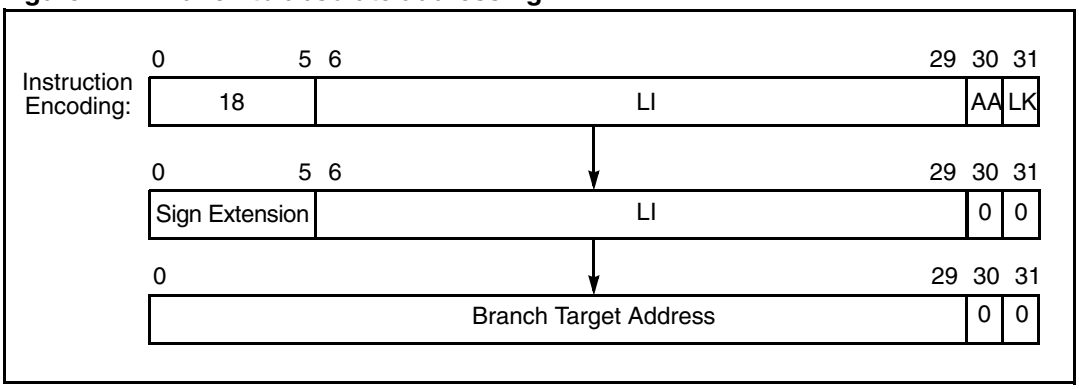
Figure 10. Branch conditional relative addressing



Branch to absolute addressing mode

Instructions that use branch to absolute addressing mode generate the next instruction address by sign extending and appending 0b00 to the LI operand. Branches using this addressing mode have the absolute addressing option enabled (AA field, bit 30, in the instruction encoding = 1). The LR update option can be enabled (LK field, bit 31, in the instruction encoding = 1). This option causes the EA of the instruction following the branch instruction to be placed in the LR. *Figure 11* shows how the branch target address is generated using this mode.

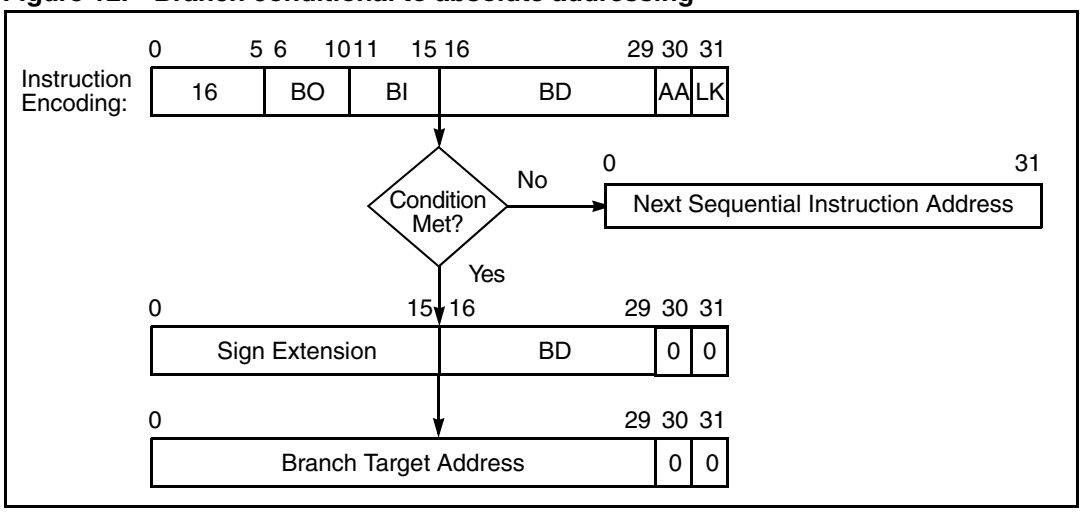
Figure 11. Branch to absolute addressing



Branch conditional to absolute addressing mode

If the branch conditions are met, instructions that use the branch conditional to absolute addressing mode generate the next instruction address by sign extending and appending 0b00 to the BD operand. Branches using this addressing mode have the absolute addressing option enabled (AA field, bit 30, in the instruction encoding = 1). The LR update option can be enabled (bit 31 (LK) in the instruction encoding = 1). This option causes the EA of the instruction following the branch instruction to be placed in the LR. *Figure 12* shows how the branch target address is generated using this mode.

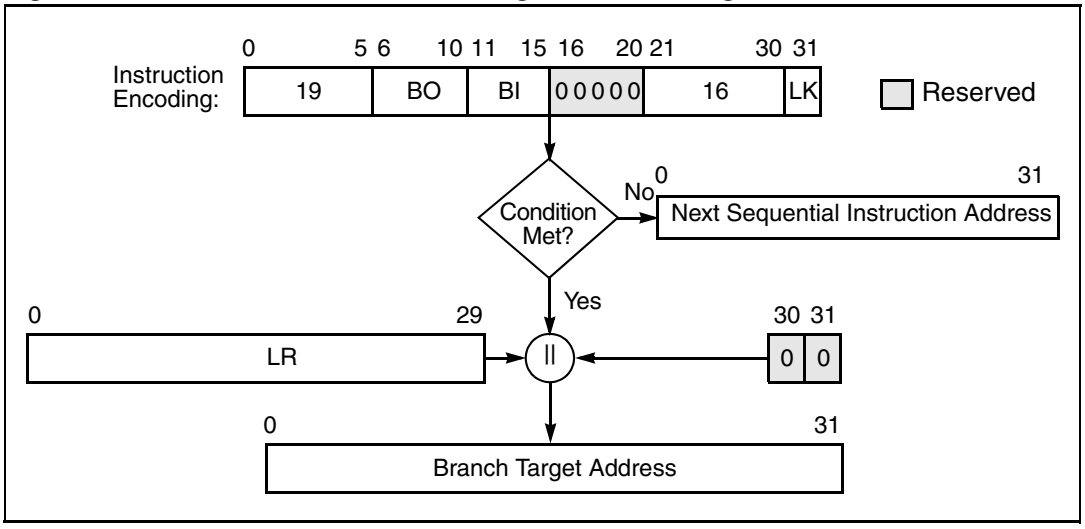
Figure 12. Branch conditional to absolute addressing



Branch conditional to link register addressing mode

If the branch conditions are met, the branch conditional to LR instruction generates the next instruction address by fetching the contents of the LR and clearing the two low-order bits to zero. The LR update option can be enabled (LK field, bit 31, in the instruction encoding = 1). This option causes the EA of the instruction following the branch instruction to be placed in the LR. *Figure 13* shows how the branch target address is generated using this mode.

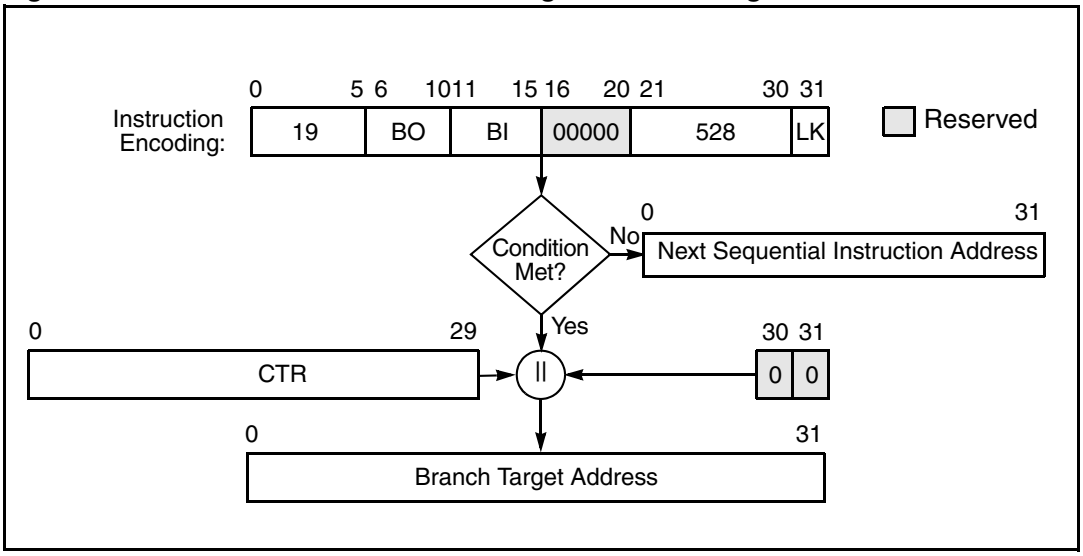
Figure 13. Branch conditional to link register addressing



Branch conditional to count register addressing mode

If the branch conditions are met, the branch conditional to count register instruction generates the next instruction address by fetching the contents of the count register (CTR) and clearing the two low-order bits to zero. The LR update option can be enabled (LK field, bit 31, in the instruction encoding = 1). This option causes the EA of the instruction following the branch instruction to be placed in the LR. *Figure 14* shows how the branch target address is generated when using this mode.

Figure 14. Branch conditional to count register addressing



Conditional branch control

Note: Some processors do not implement the static branch prediction defined in Book E and described here. For those processors, the BO operand is ignored for branch prediction.

For branch conditional instructions, the BO operand specifies the conditions under which the branch is taken. The first four bits of the BO operand specify how the branch is affected by or affects the condition and count registers. The fifth bit, shown in *Table 89* as having the value *y*, is used by some implementations for branch prediction as described below.

Table 88. BO bit descriptions

BO Bits	Description
0	Setting this bit causes the CR bit to be ignored.
1	Bit value to test against
2	Setting this causes the decrement to not be decremented.
3	Setting this bit reverses the sense of the CTR test.
4	Used for the *y* bit, which provides a hint about whether a conditional branch is likely to be taken and may be used by some implementations to improve performance.

The encodings for the BO operands are shown in *Table 89*.

Table 89. BO operand encodings

BO	Description
0000*y*	Decrement the CTR, then branch if the decremented CTR ≠ 0 and the condition is FALSE.
0001*y*	Decrement the CTR, then branch if the decremented CTR = 0 and the condition is FALSE.
001*zy*	Branch if the condition is FALSE.
0100*y*	Decrement the CTR, then branch if the decremented CTR ≠ 0 and the condition is TRUE.
0101*y*	Decrement the CTR, then branch if the decremented CTR = 0 and the condition is TRUE.
011*zy*	Branch if the condition is TRUE.
1*z*00*y*	Decrement the CTR, then branch if the decremented CTR ≠ 0.
1*z*01*y*	Decrement the CTR, then branch if the decremented CTR = 0.
1*z*1*zz*	Branch always.

In this table, z indicates a bit that is ignored. Note that the z bits should be cleared, as they may be assigned a meaning in some future version of the architecture.
The y bit provides a hint about whether a conditional branch is likely to be taken and may be used by some implementations to improve performance.

The branch always encoding of the BO operand does not have a *y* bit.

Clearing the *y* bit indicates a predicted behavior for the branch instruction as follows:

- For **bc***x* with a negative value in the displacement operand, the branch is taken.
- In all other cases (**bc***x* with a non-negative value in the displacement operand, **bclr***x*, or **bcctr***x*), the branch is not taken.

Setting the *y* bit reverses the preceding indications.

The sign of the displacement operand is used as described above even if the target is an absolute address. The default value for the *y* bit should be 0 and should be set to 1 only if software has determined that the prediction corresponding to *y* = 1 is more likely to be correct than the prediction corresponding to *y* = 0. Software that does not compute branch predictions should clear the *y* bit.

In most cases, the branch should be predicted to be taken if the value of the following expression is 1, and predicted to fall through if the value is 0.

((BO[0] & BO[2]) | S) ≈ BO[4]

In the expression above, S (bit 16 of the branch conditional instruction coding) is the sign bit of the displacement operand if the instruction has a displacement operand and is 0 if the operand is reserved. BO[4] is the *y* bit, or 0 for the branch always encoding of the BO operand. (Advantage is taken of the fact that, for **bclr***x* and **bcctr***x*, bit 16 of the instruction is part of a reserved operand and therefore must be 0.)

The 5-bit BI operand in branch conditional instructions specifies which CR bit represents the condition to test. The CR bit selected is BI +32, as shown in *Table 17*.

If the branch instructions contain immediate addressing operands, the target addresses can be computed sufficiently ahead of the branch instruction that instructions can be fetched along the target path. If the branch instructions use the link and count registers, instructions along the target path can be fetched if the link or count register is loaded sufficiently ahead of the branch instruction.

Branching can be conditional or unconditional, and optionally a branch return address is created by storing the EA of the instruction following the branch instruction in the LR after the branch target address has been computed. This is done regardless of whether the branch is taken.

Branch instructions

Table 90 lists branch instructions provided by the Book E processors. A set of simplified mnemonics and symbols is provided for the most frequently used forms of branch conditional, compare, trap, rotate and shift, and certain other instructions. See *Appendix B: Simplified mnemonics for PowerPC instructions on page 1110*."

Table 90. Branch instructions

Name	Mnemonic	Syntax
Branch	**b** (**ba bl bla**)	target_addr
Branch Conditional	**bc** (**bca bcl bcla**)	BO,BI,target_addr
Branch Conditional to Link Register	**bclr** (**bclrl**)	BO,BI
Branch Conditional to Count Register	**bcctr** (**bcctrl**)	BO,BI

Note that the EIS defines the Integer Select instruction, **isel**, which can be used to more efficiently handle sequences with multiple conditional branches. Its syntax is given in *Chapter 3.6.2*." A detailed description including an example of how **isel** can be used can be found in *Chapter 7.1.2 on page 824*."

Condition register (cr) logical Instructions

CR logical instructions, shown in *Table 91*, and the Move Condition Register Field (**mcrf**) instruction are also defined as flow control instructions.

Table 91. Condition register logical instructions

Name	Mnemonic	Syntax
Condition Register AND	**crand**	**crb**D,**crb**A,**crb**B
Condition Register OR	**cror**	**crb**D,**crb**A,**crb**B
Condition Register XOR	**crxor**	**crb**D,**crb**A,**crb**B
Condition Register NAND	**crnand**	**crb**D,**crb**A,**crb**B
Condition Register NOR	**crnor**	**crb**D,**crb**A,**crb**B
Condition Register Equivalent	**creqv**	**crb**D,**crb**A,**crb**B
Condition Register AND with Complement	**crandc**	**crb**D,**crb**A,**crb**B
Condition Register OR with Complement	**crorc**	**crb**D,**crb**A,**crb**B
Move Condition Register Field	**mcrf**	**crf**D,**crf**S

Note that if the LR update option is enabled for any of these instructions, the Book E architecture defines these forms of the instructions as invalid.

Trap instructions

The trap instructions shown in *Table 92* test for a specified set of conditions. If any of the conditions tested by a trap instruction are met, the system trap type program interrupt is taken. For more information, see *Chapter 4.7.7: Program interrupt on page 265*." If the tested conditions are not met, instruction execution continues normally. See *Appendix B: Simplified mnemonics for PowerPC instructions on page 1110*."

Table 92. Trap instructions

Name	Mnemonic	Syntax
Trap Word Immediate	**twi**	TO,**r**A,SIMM
Trap Word	**tw**	TO,**r**A,**r**B

System linkage instruction

The system call (**sc**) instruction permits a program to call on the system to perform a service; see *Table 93* and *System linkage instructions on page 182*."

Table 93. System linkage instruction

Name	Mnemonic	Syntax
System Call	**sc**	—

Executing this instruction causes the system call interrupt handler to be invoked. For more information, see *Chapter 4.7.9*."

Processor control instructions

Processor control instructions are used to read from and write to the CR, machine state register (MSR), and special-purpose registers (SPRs).

Move to/from condition register instructions

Table 94 summarizes the instructions for reading from or writing to the CR.

Table 94. Move to/from condition register instructions

Name	Mnemonic	Syntax
Move to Condition Register Fields	**mtcrf**	CRM,**r**S
Move to Condition Register from XER	**mcrxr**	**cr**D
Move from Condition Register	**mfcr**	**r**D

Move to/from special-purpose register instructions

Table 95 lists the **mtspr** and **mfspr** instructions.

Table 95. Move to/from special-purpose register instructions

Name	Mnemonic	Syntax
Move to Special-Purpose Register	**mtspr**	SPR,**r**S
Move from Special-Purpose Register	**mfspr**	**r**D,SPR

Table 96 summarizes all SPRs defined in Book E, indicating which are user-level access. The SPR number column lists register numbers used in the instruction mnemonics.

Table 96. Book E special-purpose registers (by SPR abbreviation)

SPR	Name	Defined SPR number		Access	Supervisor only	Section/ page
		Decimal	Binary			
CSRR0	Critical save/restore register 0	58	00001 11010	Read/Write	Yes	*on page 82*
CSRR1	Critical save/restore register 1	59	00001 11011	Read/Write	Yes	*on page 82*
CTR	Count register	9	00000 01001	Read/Write	No	*on page 68*
DAC1	Data address compare 1	316	01001 11100	Read/Write	Yes	*Chapter 2.13.4*
DAC2	Data address compare 2	317	01001 11101	Read/Write	Yes	*Chapter 2.13.4*
DBCR0	Debug control register 0	308	01001 10100	Read/Write	Yes	*on page 108*
DBCR1	Debug control register 1	309	01001 10101	Read/Write	Yes	*on page 110*
DBCR2	Debug control register 2	310	01001 10110	Read/Write	Yes	*on page 113*
DBSR	Debug status register	304	01001 10000	Read/Clear(1)	Yes	*Chapter 2.13.2*
DEAR	Data exception address register	61	00001 11101	Read/Write	Yes	*on page 82*
DEC	Decrementer	22	00000 10110	Read/Write	Yes	*Chapter 2.8.4*
DECAR	Decrementer auto-reload	54	00001 10110	Write-only	Yes	*Chapter 2.8.5*
DVC1	Data value compare 1	318	01001 11110	Read/Write	Yes	*Chapter 2.13.5*
DVC2	Data value compare 2	319	01001 11111			
ESR	Exception syndrome register	62	00001 11110	Read/Write	Yes	*on page 84*
IAC1	Instruction address compare 1	312	01001 11000	Read/Write	Yes	*Chapter 2.13.3*
IAC2	Instruction address compare 2	313	01001 11001	Read/Write	Yes	*Chapter 2.13.3*
IAC3	Instruction address compare 3	314	01001 11010	Read/Write	Yes	*Chapter 2.13.3*
IAC4	Instruction address compare 4	315	01001 11011	Read/Write	Yes	*Chapter 2.13.3*
IVOR0	Critical input	400	01100 10000	Read/Write	Yes	*on page 83*
IVOR1	Critical input interrupt offset	401	01100 10001	Read/Write	Yes	*on page 83*
IVOR2	Data storage interrupt offset	402	01100 10010	Read/Write	Yes	*on page 83*
IVOR3	Instruction storage interrupt offset	403	01100 10011	Read/Write	Yes	*on page 83*
IVOR4	External input interrupt offset	404	01100 10100	Read/Write	Yes	*on page 83*
IVOR5	Alignment interrupt offset	405	01100 10101	Read/Write	Yes	*on page 83*

Table 96. Book E special-purpose registers (by SPR abbreviation) (continued)

SPR	Name	Defined SPR number		Access	Supervisor only	Section/ page
		Decimal	Binary			
IVOR6	Program interrupt offset	406	01100 10110	Read/Write	Yes	*on page 83*
IVOR7	Floating-point unavailable interrupt offset	407	01100 10111	Read/Write	Yes	*on page 83*
IVOR8	System call interrupt offset	408	01100 11000	Read/Write	Yes	*on page 83*
IVOR9	Auxiliary processor unavailable interrupt offset	409	01100 11001	Read/Write	Yes	*on page 83*
IVOR10	Decrementer interrupt offset	410	01100 11010	Read/Write	Yes	*on page 83*
IVOR11	Fixed-interval timer interrupt offset	411	01100 11011	Read/Write	Yes	*on page 83*
IVOR12	Watchdog timer interrupt offset	412	01100 11100	Read/Write	Yes	*on page 83*
IVOR13	Data TLB error interrupt offset	413	01100 11101	Read/Write	Yes	*on page 83*
IVOR14	Instruction TLB error interrupt offset	414	01100 11110	Read/Write	Yes	*on page 83*
IVOR15	Debug interrupt offset	415	01100 11111	Read/Write	Yes	*on page 83*
IVPR	Interrupt vector	63	00001 11111	Read/Write	Yes	*Chapter 2.13.3*
LR	Link register	8	00000 01000	Read/Write	No	*Chapter 2.5.2*
PID	Process ID register (2)	48	00001 10000	Read/Write	Yes	*Chapter 2.12.1*
PIR	Processor ID register	286	01000 11110	Read only	Yes	*Chapter 2.7.3*
PVR	Processor version register	287	01000 11111	Read only	Yes	*Chapter 2.7.4*
SPRG0	SPR general 0	272	01000 10000	Read/Write	Yes	*Chapter 2.10*
SPRG1	SPR general 1	273	01000 10001	Read/Write	Yes	*Chapter 2.10*
SPRG2	SPR general 2	274	01000 10010	Read/Write	Yes	*Chapter 2.10*
SPRG3	SPR general 3	259	01000 00011	Read only	No(3)	*Chapter 2.10*
		275	01000 10011	Read/Write	Yes	*Chapter 2.10*
SPRG4	SPR general 4	260	01000 00100	Read only	No	*Chapter 2.10*
		276	01000 10100	Read/Write	Yes	*Chapter 2.10*
SPRG5	SPR general 5	261	01000 00101	Read only	No	*Chapter 2.10*
		277	01000 10101	Read/Write	Yes	*Chapter 2.10*
SPRG6	SPR general 6	262	01000 00110	Read only	No	*Chapter 2.10*
		278	01000 10110	Read/Write	Yes	*Chapter 2.10*
SPRG7	SPR general 7	263	01000 00111	Read only	No	*Chapter 2.10*
		279	01000 10111	Read/Write	Yes	*Chapter 2.10*
SRR0	Save/restore register 0	26	00000 11010	Read/Write	Yes	*on page 81*
SRR1	Save/restore register 1	27	00000 11011	Read/Write	Yes	*on page 81*

Table 96. Book E special-purpose registers (by SPR abbreviation) (continued)

SPR	Name	Defined SPR number		Access	Supervisor only	Section/ page
		Decimal	Binary			
TBL	Time base lower	268	01000 01100	Read only	No	*Chapter 2.8.3*
		284	01000 11100	Write-only	Yes	*Chapter 2.8.3*
TBU	Time base upper	269	01000 01101	Read only	No	*Chapter 2.8.3*
		285	01000 11101	Write-only	Yes	*Chapter 2.8.3*
TCR	Timer control register	340	01010 10100	Read/Write	Yes	*Chapter 2.8.1*
TSR	Timer status register	336	01010 10000	Read/Clear(4)	Yes	*Chapter 2.8.2*
USPRG 0	User SPR general 0(5)	256	01000 00000	Read/Write	No	*Chapter 2.10*
XER	Integer exception register	1	00000 00001	Read/Write	No	*Chapter 2.3.2*

1. The DBSR is read using **mfspr**. It cannot be directly written to. Instead, DBSR bits corresponding to 1 bits in the GPR can be cleared using **mtspr**.
2. Implementations may support more than one PID. If multiple PIDs are implemented, the Book E–defined PID is implemented as PID0.
3. User-mode read access to SPRG3 is implementation-dependent.
4. The TSR is read using **mfspr**. It cannot be directly written to. Instead, TSR bits corresponding to 1 bits in the GPR can be cleared using mtspr.
5. USPRG0 is a separate physical register from SPRG0.

Table 97 lists EIS-specific SPRs, indicating which can be accessed by user-level software. Compilers should recognize SPR names when parsing instructions.

Table 97. Implementation-specific SPRs (by SPR abbreviation)

SPR	Name	SPR number	Access	Supervisor only	Section/page
ATBL	Alternate time base lower	526	Read-only	No	*Chapter 2.15*
ATBU	Alternate time base upper	527	Read-only	No	*Chapter 2.15*
DSRR0	Debug save/restore register 0	574	R/W	Yes	*on page 86*
DSRR1	Debug save/restore register 1	575	R/W	Yes	*on page 86*
IVOR32	SPE/embedded floating-point APU unavailable interrupt offset	528	Read/Write	Yes	*on page 83*
IVOR33	Embedded floating-point data exception interrupt offset	529	Read/Write	Yes	*on page 83*
IVOR34	Embedded floating-point round exception interrupt offset	530	Read/Write	Yes	*on page 83*
IVOR35	Performance monitor	531	Read/Write	Yes	*on page 83*
L1CFG0	L1 cache configuration register 0	515	Read-only	No	*Chapter 2.11.3*
L1CFG1	L1 cache configuration register 1	516	Read-only	No	*Chapter 2.11.3*
L1CSR0	L1 cache control and status register 0	1010	Read/Write	Yes	*Chapter 2.11.1*
L1CSR1	L1 cache control and status register 1	1011	Read/Write	Yes	*Chapter 2.11.2*

Table 97. Implementation-specific SPRs (by SPR abbreviation) (continued)

SPR	Name	SPR number	Access	Supervisor only	Section/page
L1FINV0	L1 flush and invalidate control register 0	1016	Read/Write	Yes	*Chapter 2.11.5*
MAS0	MMU assist register 0	624	Read/Write	Yes	*on page 101*
MAS1	MMU assist register 1	625	Read/Write	Yes	*on page 101*
MAS2	MMU assist register 2	626	Read/Write	Yes	*on page 101*
MAS3	MMU assist register 3	627	Read/Write	Yes	*on page 104*
MAS4	MMU assist register 4	628	Read/Write	Yes	*on page 104*
MAS5	MMU assist register 5.	629	Read/Write	Yes	*on page 104*
MAS6	MMU assist register 6	630	Read/Write	Yes	*on page 104*
MAS7	MMU assist register 7	944	Read/Write	Yes	*on page 107*
MCAR	Machine check address register	573	Read-only	Yes	*on page 107*
MCSR	Machine check syndrome register	572	Read/Write	Yes	*on page 88*
MCSRR0	Machine-check save/restore register 0	570	Read/Write	Yes	*on page 88*
MCSRR1	Machine-check save/restore register 1	571	Read/Write	Yes	*on page 88*
MMUCFG	MMU configuration register	1015	Read-only	Yes	*Chapter 2.12.3*
MMUCSR0	MMU control and status register 0	1012	Read/Write	Yes	*Chapter 2.12.2*
PID0	Process ID register 0. Book E defines only this PID register and refers to as PID, not PID0.	48	Read/Write	Yes	*Chapter 2.12.1*
PID1	Process ID register 1	633	Read/Write	Yes	*Chapter 2.12.1*
PID2	Process ID register 2	634	Read/Write	Yes	*Chapter 2.12.1*
SPEFSCR	Signal processing and embedded floating-point status and control register	512	Read/Write	No	*Chapter 2.14.1*
SVR	System version register	1023	Read-only	Yes	*Chapter 2.7.5*
TLB0CFG	TLB configuration register 0	688	Read-only	Yes	*Chapter 2.12.4*
TLB1CFG	TLB configuration register 1	689	Read-only	Yes	*Chapter 2.12.4*

Memory synchronization instructions

Memory synchronization instructions control the order in which memory operations complete with respect to asynchronous events and the order in which memory operations are seen by other mechanisms that access memory. See *Table 98* for a summary.

Table 98. Memory synchronization instructions

Name	Mnemonic	Syntax	EIS notes
Instruction synchronize	**isync**	—	Refetch serializing. An **isync** waits for previous instructions (including any interrupts they generate) to complete before **isync** executes, which purges all instructions from the processor and refetches the next instruction. **isync** does not wait for pending stores in the store queue to complete. Any subsequent instruction sees all effects of instructions before the **isync**. Because it prevents execution of subsequent instructions until preceding instructions complete, if an **isync** follows a conditional branch that depends on the value returned by a preceding load, the load on which the branch depends is performed before any loads caused by instructions after the **isync** even if the effects of the dependency are independent of the value loaded (for example, the value is compared to itself and the branch tests selected, CR*n*[EQ]), and even if the branch target is the next sequential instruction to be executed.
Load word and reserve indexed	lwarx	**r**D,**r**A,**r**B	**lwarx** with **stwcx.** can emulate semaphore operations such as test and set, compare and swap, exchange memory, and fetch and add. Both instructions must use the same EA. Reservation granularity is implementation-dependent. Executing **lwarx** and **stwcx.** to a page marked write-through (WIMG = 10*xx*) or when the data cache is locked may cause a data storage interrupt. If the location is not word-aligned, an alignment interrupt occurs.
Memory barrier	**mbar**	MO	**mbar** provides a pipelined memory barrier. (Note that **mbar** uses the same opcode as **eieio**, which is not defined by Book E.) The behavior of **mbar** is affected by the MO field (bits 6–10) of the instruction. MO = 0—**mbar** behaves identically to **msync**. MO = 1—**mbar** is a weaker, faster memory barrier; see the user's manual for implementation-specific behavior.

Table 98. Memory synchronization instructions (continued)

Name	Mnemonic	Syntax	EIS notes
Memory synchronize	msync	—	Provides an ordering function for the effects of all instructions executed by the processor executing the **msync**. Executing an **msync** ensures that all previous instructions complete before it completes and that no subsequent instructions are initiated until after it completes. It also creates a memory barrier, which orders the storage accesses associated with these instructions. **msync** cannot complete before storage accesses associated with previous instructions are performed. **msync** is execution synchronizing. Note the following: **msync** is used to ensure that all stores into a data structure caused by store instructions executed in a critical section of a program are performed with respect to another processor before the store that releases the lock is performed with respect to that processor. **mbar** is preferable in many cases. On ST Book E devices: Unlike a context-synchronizing operations, **msync** does not discard prefetched instructions.
Store word conditional indexed	stwcx.	**r**S,**r**A,**r**B	**lwarx** with **stwcx.** can emulate semaphore operations such as test and set, compare and swap, exchange memory, and fetch and add. Both instructions must use the same EA. Reservation granularity is implementation-dependent. Executing **lwarx** and **stwcx.** to a page marked write-through (WIMG = 10*xx*) or cache-inhibited (WIMG = 01*xx*) when the data cache is locked may cause a data storage interrupt. If the location is not word-aligned, an alignment interrupt occurs.

Atomic update primitives using lwarx and stwcx.

The **lwarx** and **stwcx.** instructions together permit atomic update of a memory location. Book E provides word and double word forms of each of these instructions. Described here is the operation of **lwarx** and **stwcx.**

A specified memory location that may be modified by other processors or mechanisms requires memory coherence. If the location is in write-through required or caching inhibited memory, the implementation determines whether these instructions function correctly or cause the system data storage error handler to be invoked.

Note the following:

- The memory coherence required attribute on other processors and mechanisms ensures that their stores to the specified location will cause the reservation created by the **lwarx** to be cancelled.
- **Warning:** Support for load and reserve and store conditional instructions for which the specified location is in caching-inhibited memory is being phased out of Book E. It is likely not to be provided on future implementations. New programs should not use these instructions to access caching inhibited memory.

A **lwarx** instruction is a load from a word-aligned location with the following side effects.

- A reservation for a subsequent **stwcx.** instruction is created.
- The memory coherence mechanism is notified that a reservation exists for the location accessed by the **lwarx**.

The **stwcx.** is a store to a word-aligned location that is conditioned on the existence of the reservation created by the **lwarx** and on whether both instructions specify the same location. To emulate an atomic operation, both **lwarx** and **stwcx.** must access the same location. **lwarx** and **stwcx.** are ordered by a dependence on the reservation, and the program is not required to insert other instructions to maintain the order of memory accesses caused by these two instructions.

A **stwcx.** performs a store to the target location only if the location accessed by the **lwarx** that established the reservation has not been stored into by another processor or mechanism between supplying a value for the **lwarx** and storing the value supplied by the **stwcx.**. If the instructions specify different locations, the store is not necessarily performed. CR0 is modified to indicate whether the store was performed, as follows:

CR0[LT,GT,EQ,SO] = 0b00 || store_performed || XER[SO]

If a **stwcx.** completes but does not perform the store because a reservation no longer exists, CR0 is modified to indicate that the **stwcx.** completed without altering memory.

A **stwcx.** that performs its store is said to succeed.

Examples using **lwarx** and **stwcx.** are given in *Appendix C: Programming examples on page 1143*."

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 on another processor may return a stale value. However, a subsequent **lwarx** from the given location on the other processor followed by a successful **stwcx.** on 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).

Reservations

The ability to emulate an atomic operation using **lwarx** and **stwcx.** is based on the conditional behavior of **stwcx.**, the reservation set 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 memory called a reservation granule. The size of the reservation granule is implementation-dependent, but is a multiple of 4 bytes for **lwarx**. The reservation granule associated with EA contains the real address to which the 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.') When one processor holds a reservation and another processor performs a store, the first processor's reservation is cleared if the store affects any bytes in the reservation granule.

*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, which 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 is subsequently restored. The use of **lwarx** and **stwcx.** improves on such a compare and swap, because the reservation reliably binds **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**.*

A processor has at most one reservation at any time. Book E states that a reservation is established by executing a **lwarx** and is lost (or may be lost, in the case of the fourth and fifth bullets) if any of the following occurs.

- The processor holding the reservation executes another **lwarx**; this clears the first reservation and establishes a new one.
- The processor holding the reservation executes any **stwcx.**, regardless of whether the specified address matches that of the **lwarx**.
- Another processor executes a store or **dcbz** to the same reservation granule.
- Another processor executes a **dcbtst**, **dcbst**, or **dcbf** to the same reservation granule; whether the reservation is lost is undefined.
- Another processor executes a **dcba** to the reservation granule. The reservation is lost if the instruction causes the target block to be newly established in the data cache or to be modified; otherwise, whether the reservation is lost is undefined.
- Some other mechanism modifies a location in the same reservation granule.
- Other implementation-specific conditions may also cause the reservation to be cleared, See the core reference manual.

Interrupts are not guaranteed to clear reservations. (However, system software invoked by interrupts may clear reservations.)

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 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 issuing a **stwcx.** to a dummy writable word-aligned location as part of the context switch, thereby clearing any reservation established by the old context. Executing **stwcx.** to a word-aligned location is enough to clear the reservation, regardless of whether it was set by **lwarx**.

Forward progress

Forward progress in loops that use **lwarx** and **stwcx.** is achieved by a cooperative effort among hardware, operating system software, and application software.

Book E guarantees one of the following when a processor executes a **lwarx** to obtain a reservation for location X and then a **stwcx.** to store a value to location X:

1. The **stwcx.** succeeds and the value is written to location X.
2. The **stwcx.** fails because some other processor or mechanism modified location X.
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 memory or context switches. 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. Although Book E alone cannot provide such a

guarantee, the conditions in cases 1 and 2 are necessary for a guarantee. An implementation and operating system can build on them to provide such a guarantee.

Note that Book E does not guarantee fairness. In competing for a reservation, two processors can indefinitely lock out a third.

Reservation loss due to granularity

Lock words should be allocated such that contention for the locks and updates to nearby data structures do not cause excessive reservation losses due to false indications of sharing that can occur due to the reservation granularity.

A processor holding a reservation on any word in a reservation granule loses its reservation if some other processor stores anywhere in that granule. Such problems can be avoided only by ensuring that few such stores occur. This can most easily be accomplished by allocating an entire granule for a lock and wasting all but one word.

Reservation granularity may vary for each implementation. There are no architectural restrictions bounding the granularity implementations must support, so reasonably portable code must dynamically allocate aligned and padded memory for locks to guarantee absence of granularity-induced reservation loss.

Memory control instructions

Memory control instructions can be classified as follows:

- User- and supervisor-level cache management instructions.
- Supervisor-level–only translation lookaside buffer management instructions

This section describes the user-level cache management instructions. See *Supervisor-level memory control instructions on page 183*," for information about supervisor-level cache and translation lookaside buffer management instructions.

This section does not describe the cache-locking APU instructions, which are described in *Chapter 3.6.4: Cache locking APU on page 200*."

Cache management instructions

Cache management instructions obey the sequential execution model except as described in the example in this section of managing coherence between the instruction and data caches.

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 from or a store to the addressed byte with respect to address translation, memory protection, and the memory access ordering done by **msync**, **mbar**, and the other means described in *Memory access ordering on page 290*."

If caches are combined, the same value should be given for an instruction cache attribute and the corresponding data cache attribute.

Each implementation provides an efficient way for software to ensure that all blocks that are considered to be modified in the data cache have been copied to main memory before the processor enters any power-saving mode in which data cache contents are not maintained. The means are described in the reference manual for the implementation.

It is permissible for an implementation to treat any or all of the cache touch instructions (**icbt**, **dcbt**, or **dcbtst**) as no-operations, even if a cache is implemented.

The instruction cache is not necessarily kept consistent with the data cache or with main memory. When instructions are modified, software must ensure that the instruction cache is made consistent with data memory and that the modifications are made visible to the instruction fetching mechanism. The following instruction sequence can be used to accomplish this when the instructions being modified are in memory that is memory coherence required and one program both modifies the instructions and executes them. (Additional synchronization is needed when one program modifies instructions that another program will execute.) In this sequence, location 'instr' is assumed to contain modified instructions.

```
dcbst     instr     # update block in main memory
msync               # order update before invalidation
icbi      instr     # invalidate copy in instr cache
msync               # order invalidation before discarding prefetched instructions
isync               # discard prefetched instructions
```

Note: Because the optimal instruction sequence may vary between systems, many operating systems provide a system service to perform the function described above.
As stated above, the EA is translated using translation resources used for data accesses, even though the block being invalidated was copied into the instruction cache based on translation resources used for instruction fetches.

User-level cache instructions

The instructions listed in *Table 99* help user-level programs manage on-chip caches if they are implemented. The following sections describe how these operations are treated with respect to the caches. The EIS supports the following CT values, defined by the EIS:

- CT = 0 indicates the L1 cache.
- CT = 1 indicates the I/O cache. (Note that some versions of the e500 documentation refer to the I/O cache as a frontside L2 cache.)
- CT = 2 indicates a backside L2 cache.

As with other memory-related instructions, the effects of cache management instructions on memory are weakly-ordered. If the programmer must ensure that cache or other instructions have been performed with respect to all other processors and system mechanisms, an **msync** must be placed after those instructions.

Chapter 3.6.4," describes cache-locking APU instructions.

Table 99. User-level cache instructions

Name	Mnemonic	Syntax	Descriptions
Data cache block allocate	**dcba**	**r**A,**r**B	This instruction is treated as a store with respect to any memory barriers, synchronization, translation and protection, and debug address comparisons. A no-op occurs if the cache is disabled or locked, if the page is marked write-through or cache-inhibited, or if a TLB protection violation occurs. An implementation may chose to no-op the instruction.
Data cache block flush	**dcbf**	**r**A,**r**B	This instruction is treated as a load with respect to any memory barriers, synchronization, translation and protection, and debug address comparisons.

Table 99. User-level cache instructions (continued)

Name	Mnemonic	Syntax	Descriptions
Data cache block set to zero	**dcbz**	**r**A,**r**B	This instruction is treated as a store with respect to any memory barriers, synchronization, translation and protection, and debug address comparisons. If the block containing the byte addressed by EA is in the data cache, all bytes of the block are cleared. If the block containing the byte addressed by EA is not in the data cache and is in storage that is not caching inhibited, the block is established in the data cache without fetching the block from main storage and all bytes of the block are cleared. If the block containing the byte addressed by EA is not in the data cache and is in storage that is not caching inhibited and cannot be established in the cache, then one of the following occurs: All bytes of the area of main storage that corresponds to the addressed block are set to zero An alignment interrupt is taken If the block containing the byte addressed by EA is in storage that is caching inhibited or write through required, one of the following occurs: All bytes of the area of main storage that corresponds to the addressed block are set to zero An alignment interrupt is taken.
Data cache block store	**dcbst**	**r**A,**r**B	This instruction is treated as a load with respect to any memory barriers, synchronization, translation and protection, and debug address comparisons.
Data cache block touch (1)	**dcbt**	CT,**r**A,**r**B	This instruction is treated as a load with respect to any memory barriers, synchronization, translation and protection, and debug address comparisons. A no-op occurs if the cache is disabled or locked, if the page is marked write-through or cache-inhibited, or if a TLB protection violation occurs. An implementation may chose to no-op the instruction.
Data cache block touch for store 1	**dcbtst**	CT,**r**A,**r**B	Depending on the implementation, this instruction is treated as a load or store with respect to any memory barriers, synchronization, translation and protection, and debug address comparisons. A no-op occurs if the cache is disabled or locked, if the page is marked write-through or cache-inhibited, or if a TLB protection violation occurs. An implementation may chose to no-op the instruction.

Table 99. User-level cache instructions (continued)

Name	Mnemonic	Syntax	Descriptions
Instruction cache block invalidate	icbi	**r**A,**r**B	This instruction is treated as a load with respect to any memory barriers, synchronization, translation and protection, and debug address comparisons.
Instruction cache block touch	icbt	CT,**r**A,**r**B	This instruction is treated as a load with respect to any memory barriers, synchronization, translation and protection, and debug address comparisons. A no-op occurs if the cache is disabled or locked, if the page is marked write-through or cache-inhibited, or if a TLB protection violation occurs. An implementation may chose to no-op the instruction.

1. A program that uses **dcbt** and **dcbtst** improperly is less efficient. To improve performance, HID0[NOPTI] can be set, which causes **dcbt** and **dcbtst** to be no-oped at the cache. They do not cause bus activity and cause only a 1-clock execution latency. The default state of this bit is zero, which enables the use of these instructions.

3.3.2 Supervisor level instructions

The Book E architecture includes the structure of the memory management model, supervisor-level registers, and the interrupt model. This section describes the supervisor-level instructions defined by the EIS.

System linkage instructions

This section describes the system linkage instructions (see *Table 100*). The user-level **sc** instruction lets a user program call on the system to perform a service and causes the processor to take a system call interrupt. The supervisor-level **rfi** instruction is used for returning from an interrupt handler. The **rfci** instruction is used for critical interrupts. The EIS defines the **rfmci** for machine check interrupts and **rfdi** for debug APU interrupts.

Table 100. System linkage instructions—supervisor-level

Name	Mnemonic	Syntax	Implementation notes
Return from interrupt	**rfi**	—	**rfi** is context-synchronizing
Return from debug interrupt	**rfdi**	—	Debug interrupt APU. When **rfdi** is executed, the values in the debug save and restore registers (DSRR0 and DSRR1) are restored. **rfdi** is context-synchronizing.
Return from machine check interrupt	**rfmci**	—	Machine check interrupt APU. When **rfmci** is executed, the values in the machine check interrupt save and restore registers (MCSRR0 and MCSRR1) are restored. **rfmci** is context-synchronizing.
Return from critical interrupt	**rfci**	—	When **rfci** executes, the values in the critical interrupt save and restore registers (CSRR0 and CSRR1) are restored. **rfci** is context-synchronizing.
System call	**sc**	—	The **sc** instruction is context-synchronizing.

Table 101 lists instructions for accessing the MSR.

Table 101. Move to/from machine state register instructions

Name	Mnemonic	Syntax	Description
Move from machine state register	**mfmsr**	**r**D	—
Move to machine state register	**mtmsr**	**r**S	—
Write MSR external enable	**wrtee**	**r**S	Bit 48 of the contents of **r**S is placed into MSR[EE]. Other MSR bits are unaffected.
Write MSR external enable immediate	**wrteei**	E	The value of E is placed into MSR[EE]. Other MSR bits are unaffected.

Certain encodings of the SPR field of **mtspr** and **mfspr** instructions (shown in *Table 95*) provide access to supervisor-level SPRs. *Table 96* lists encodings for architecture-defined SPRs. Encodings for EIS-defined, supervisor-level SPRs are listed in *Table 102*. Simplified mnemonics are provided for **mtspr** and **mfspr**. *Appendix C: Programming examples on page 1143*," describes context synchronization requirements when altering certain SPRs.

Supervisor-level memory control instructions

Memory control instructions include the following:

- Cache management instructions (supervisor-level and user-level)
- Translation lookaside buffer management instructions

This section describes supervisor-level memory control instructions. *Memory control instructions on page 179*," describes user-level memory control instructions.

Supervisor-level cache instruction

Table 102 lists the only supervisor-level cache management instruction.

Table 102. Supervisor-Level cache management instruction

Name	Mnemonic	Syntax	Implementation notes
Data cache block invalidate	**dcbi**	**r**A,**r**B	This instruction is treated as a store with respect to any memory barriers, synchronization, translation and protection, and debug address comparisons. An implementation may first perform a **dcbst** operation before invalidating the cache block if the memory is marked as coherency required (WIMG = xx1x).

See *User-level cache instructions on page 180*," for cache instructions that provide user-level programs the ability to manage the on-chip caches.

Supervisor-level tlb management instructions

The address translation mechanism is defined in terms of TLBs and page table entries (PTEs) Book E processors use to locate the logical-to-physical address mapping for a particular access. See *Chapter 5.4: Storage model on page 301*," for more information about TLB operations. *Table 103* summarizes the operation of the TLB instructions.

Table 103. TLB management instructions

Name	Mnemonic	Syntax	Implementation Notes
TLB invalidate virtual address indexed	tlbivax	**r**A, **r**B	A TLB invalidate operation is performed whenever **tlbivax** is executed. **tlbivax** invalidates any TLB entry that corresponds to the virtual address calculated by this instruction as long as IPROT is not set; this includes invalidating TLB entries contained in TLBs on other processors and devices in addition to the processor executing **tlbivax**. Thus an invalidate operation is broadcast throughout the coherent domain of the processor executing **tlbivax**. See *Chapter 5.4 on page 301*."
TLB read entry	tlbre	—	**tlbre** causes the contents of a single TLB entry to be extracted from the MMU and be placed in the corresponding MAS register fields. The entry extracted is specified by the TLBSEL, ESEL and EPN fields of MAS0 and MAS2. The contents extracted from the MMU are placed in MAS0–MAS3 and MAS7. See *Chapter 5.4.9 on page 317*."
TLB search indexed	tlbsx	**r**A, **r**B	**tlbsx** updates MAS conditionally based on the success or failure of a lookup in the MMU. The lookup is controlled by the EA provided by GPR[**r**B] specified in the instruction encoding and MAS6[SAS,SPID]. The values placed into MAS0–MAS3 and MAS7 differ, depending on whether a successful or unsuccessful search occurred. Note that RA=0 is a preferred form for **tlbsx** and that some ST implementations take an illegal instruction exception program interrupt if RA != 0.
TLB synchronize	tlbsync	—	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 **msync** instruction executed by the same processor. Executing a tlbsync instruction ensures that all of the following occurs: All TLB invalidations caused by **tlbivax** instructions preceding the **tlbsync** will have completed on any other processor before any storage accesses associated with data accesses caused by instructions following the **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 msync instruction, to the extent required by the associated memory coherence required attributes, before the **mbar** or **msync** instruction's memory barrier is created. See *Chapter 5.4.9 on page 317*."
TLB Write Entry	tlbwe	—	**tlbwe** causes the contents of certain fields of MAS0, MAS1, MAS2, and MAS3 to be written into a TLB entry specified by the TLBSEL, ESEL, and EPN fields of MAS0 and MAS2. If MAS7 is implemented, execution of **tlbwe** causes any MAS7[RPN] to be written to the selected TLB entry. See *Chapter 5.4.9 on page 317*."

3.3.3 Recommended simplified mnemonics

The description of each instruction includes the mnemonic and a formatted list of operands. Book E–compliant assemblers support the mnemonics and operand lists. To simplify assembly language programming, a set of simplified mnemonics and symbols is provided for some of the most frequently used instructions; refer to *Appendix B: Simplified mnemonics for PowerPC instructions on page 1110*, for a complete list. Programs written to be portable across the various assemblers for the Book E architecture should not assume the existence of mnemonics not described in this document.

3.3.4 Book E instructions with implementation-specific features

Book E defines several instructions in a general way, leaving the details of the execution up to the implementation. These are listed in *Table 104*. This section describes how the EIS further defines those instructions. See the user documentation for additional implementation-specific behavior.

Table 104. Implementation-specific instructions summary

Name	Mnemonic	Syntax	Category
Move from APID Indirect	**mfapidi**	—	Optional. If not implemented, attempted execution causes an illegal instruction exception type program interrupt.
Move from Device Control Register	**mfdcr**	—	
Move to Device Control Register	**mtdcr**	—	
TLB Invalidate Virtual Address Indexed	**tlbivax**	rA, rB	These are described generally in *Supervisor-level tlb management instructions on page 183.*
TLB Read Entry	**tlbre**	—	
TLB Search Indexed	**tlbsx**	rA, rB	
TLB Write Entry	**tlbwe**	—	

A list of user-level instructions defined by both the classic PowerPC architecture and Book E can be found in *Chapter 3.7*."

3.3.5 EIS instructions

The EIS defines the instructions listed in *Table 105* (with cross references to more detailed descriptions) that extend the Book E instruction set in accordance with Book E. SPE and embedded floating-point APU instructions are listed in *Table 108* and *Table 117*.

Table 105. EIS-defined instructions (except SPE and SPFP instructions)

Name	Mnemonic	Syntax	Section #/page
Data Cache Block Lock Clear	**dcblc**	CT, **r**A, **r**B	*Chapter 3.6.4*
Data Cache Block Touch and Lock Set	**dcbtls**	CT, **r**A, **r**B	
Data Cache Block Touch for Store and Lock Set	**dcbtstls**	CT, **r**A, **r**B	
Instruction Cache Block Lock Clear	**icblc**	CT, **r**A, **r**B	
Instruction Cache Block Touch and Lock Set	**icbtls**	CT, **r**A, **r**B	
Integer Select	**isel**	**r**D, **r**A, **r**B, **cr**b	*Chapter 3.6.2*

Table 105. EIS-defined instructions (except SPE and SPFP instructions) (continued)

Name	Mnemonic	Syntax	Section #/page
Move from Performance Monitor Register	**mfpmr**	**r**D,PMRN	*Chapter 3.6.3*
Move to Performance Monitor Register	**mtpmr**	PMRN,**r**S	
Return from Machine Check Interrupt	**rfmci**	—	*Chapter 3.6.5*
Return from Debug Interrupt	**rfdi**	—	*Chapter 3.6.5*

3.3.6 Context synchronization

Context synchronization is achieved by post- and presynchronizing instructions. An instruction is presynchronized by completing all instructions before dispatching the presynchronized instruction. Post-synchronizing is implemented by not dispatching any later instructions until the post-synchronized instruction is completely finished.

3.4 Instruction fetching

In general, instructions are prefetched from the cache on a cache hit and from memory on a cache miss. Prefetched instructions may not be executed if the instruction stream is redirected after instructions are fetched and before they are scheduled for execution.

3.5 Memory synchronization

The **msync** instruction provides a memory barrier throughout the memory hierarchy. It waits for preceding data memory accesses to reach the point of coherency (that is, visible to the entire memory hierarchy); then it is broadcast. No subsequent instructions in the stream are initiated until after **msync** completes. Note that **msync** uses the same opcode as the **sync** instruction. The **msync** instruction is described in *Memory synchronization instructions on page 175*."

See *Memory access ordering on page 290*," for detailed information.

3.6 EIS-specific instructions

This section described EIS-defined instructions that are part of APUs or other extensions to the Book E architecture.

3.6.1 SPE and embedded floating-point APUs

The SPE and the embedded vector single-precision and embedded scalar double-precision APUs provide an extended GPR file with 32, 64-bit registers. The 32-bit Book E instructions operate on the lower (least significant) 32 bits of the 64-bit register. SPE APU vector instructions and embedded vector SPFP treat 64-bit registers as containing two 32-bit elements or four 16-bit elements as described in *SPE APU instructions on page 188*." The embedded double-precision floating-point APU uses the extended GPRs to hold single, IEEE-compliant double-precision operands.

However, like 32-bit Book E instructions, scalar SPFP APU floating-point instructions use bits 32–63 of the GPRs to hold 32-bit single-precision operands, as described in *Embedded vector and scalar floating-point APU instructions on page 196*."

There is no record form of SPE or embedded floating-point instructions. Vector compare instructions store the result of the comparison into the CR. The meaning of the CR bits is now overloaded for vector operations. Vector compare instructions specify a CR field and two source registers as well as the type of compare: greater than, less than, or equal. Two bits in the CR field are written with the result of the vector compare, one for each element. The two defined bits could be used either by a vector select instruction or by a UISA branch instruction.

A partially visible accumulator register is architected for the integer and fractional multiply accumulate SPE instructions. It is described in *Chapter 2.14.2 on page 122*."

Full descriptions of these instructions can be found in *Chapter 13 on page 891*."

SPE APU instruction architecture

This section describes the instruction formats and instructions defined by the SPE APU.

Signed fractions

In signed fractional format, the N-bit operand is represented in a 1.[N–1] format (1 sign bit, N–1 fraction bits). Signed fractional numbers are in the following range:

$$-1.0 \leq SF \leq 1.0 - 2^{-(N-1)}$$

The real value of the binary operand SF[0:N-1] is as follows:

$$SF = -1.0 \bullet SF(0) + \sum_{i=1}^{N-1} SF(i) \bullet 2^{-i}$$

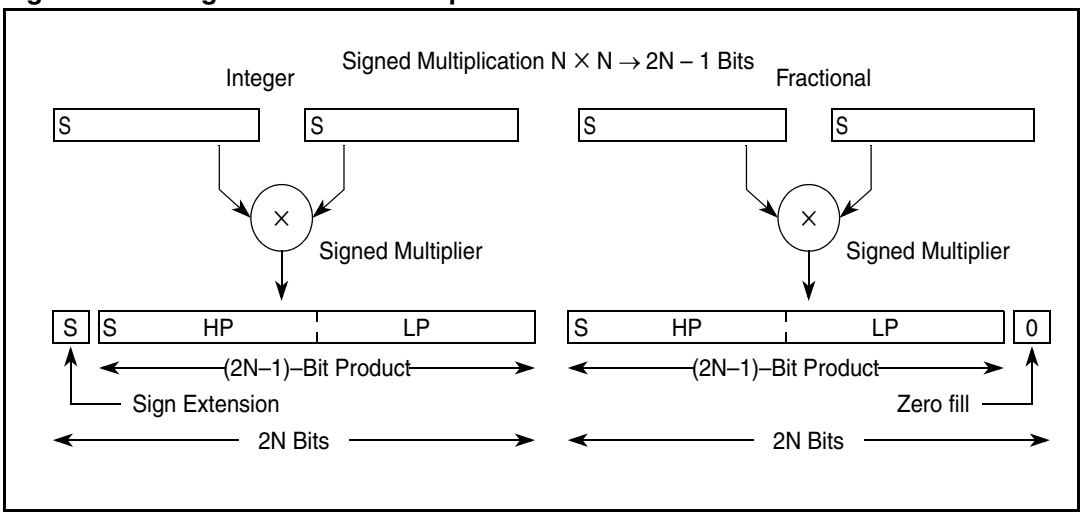
The most negative and positive numbers representable in fractional format are as follows:

- The most negative number is represented by SF(0) = 1 and SF[1:N–1] = 0 (that is, N=32; 0x8000_0000 = –1.0).
- The most positive number is represented by SF(0) = 0 and SF[1:N–1] = all 1s (that is, N=32; 0x7FFF_FFFF = 1.0 - $2^{-(N-1)}$).

SPE APU—integer and fractional operations

Figure 15 shows data formats for signed integer and fractional multiplication. Note that low word versions of signed saturate and signed modulo fractional instructions are not supported. Attempting to execute an opcode corresponding to these instructions causes boundedly undefined results.

Figure 15. Integer and fractional operations



SPE APU instructions

SPE APU instructions treat 64-bit GPRs as being composed of a vector of two 32-bit elements. (Some instructions also read or write 16-bit elements.) The SPE APU supports a number of forms of multiply and multiply-accumulate operations, and of add and subtract to accumulator operations. The SPE supports signed and unsigned forms, and optional fractional forms. For these instructions, the fractional form does not apply to unsigned forms because integer and fractional forms are identical for unsigned operands.

Table 106 shows how SPE APU vector multiply instruction mnemonics are structured.

Table 106. SPE APU vector multiply instruction mnemonic structure

Prefix	Multiply element		Data Type element		Accumulate element	
evm	ho	half odd (16x16->32)	usi	unsigned saturate integer	a	write to ACC
	he	half even (16x16->32)	umi	unsigned modulo integer	aa	write to ACC & added ACC
	hog	half odd guarded (16x16->32)	ssi	signed saturate integer	an	write to ACC & negate ACC
	heg	half even guarded (16x16->32)	**ssf** (1)	signed saturate fractional	aaw	write to ACC & ACC in words
	wh	word high (32x32->32)	smi	signed modulo integer	anw	write to ACC & negate ACC in words
	wl	word low (32x32->32)	**smf**[1]	signed modulo fractional		
	wh g	word high guarded (32x32->32)				
	wlg	word low guarded (32x32->32)				
	w	word (32x32->64)				

1. Low word versions of signed saturate and signed modulo fractional instructions are not supported. Attempting to execute an opcode corresponding to these instructions causes boundedly undefined results.

Table 107 defines mnemonic extensions for these instructions.

Table 107. Mnemonic extensions for multiply-accumulate instructions

Extension	Meaning	Comments
Multiply form		
he	Half word even	16×16→32
heg	Half word even guarded	16×16→32, 64-bit final accumulator result
ho	Half word odd	16×16→32
hog	Half word odd guarded	16×16→32, 64-bit final accumulator result
w	Word	32×32→64
wh	Word high	32×32→32, high-order 32 bits of product
wl	Word low	32×32→32, low-order 32 bits of product
Data type		
smf	Signed modulo fractional	(Wrap, no saturate)
smi	Signed modulo integer	(Wrap, no saturate)
ssf	Signed saturate fractional	
ssi	Signed saturate integer	
umi	Unsigned modulo integer	(Wrap, no saturate)
usi	Unsigned saturate integer	
Accumulate options		
a	Update accumulator	Update accumulator (no add)
aa	Add to accumulator	Add result to accumulator (64-bit sum)
aaw	Add to accumulator (words)	Add word results to accumulator words (pair of 32-bit sums)
an	Add negated	Add negated result to accumulator (64-bit sum)
anw	Add negated to accumulator (words)	Add negated word results to accumulator words (pair of 32-bit sums)

Table 108 lists SPE APU instructions.

Table 108. SPE APU vector instructions

Instruction	Mnemonic	Syntax
Bit Reversed Increment	**brinc**	**r**D,**r**A,**r**B
Initialize Accumulator	**evmra**	**r**D,**r**A
Multiply Half Words, Even, Guarded, Signed, Modulo, Fractional and Accumulate	**evmhegsmfaa**	**r**D,**r**A,**r**B
Multiply Half Words, Even, Guarded, Signed, Modulo, Fractional and Accumulate Negative	**evmhegsmfan**	**r**D,**r**A,**r**B
Multiply Half Words, Even, Guarded, Signed, Modulo, Integer and Accumulate	**evmhegsmiaa**	**r**D,**r**A,**r**B
Multiply Half Words, Even, Guarded, Signed, Modulo, Integer and Accumulate Negative	**evmhegsmian**	**r**D,**r**A,**r**B

Table 108. SPE APU vector instructions (continued)

Instruction	Mnemonic	Syntax
Multiply Half Words, Even, Guarded, Unsigned, Modulo, Integer and Accumulate	**evmhegumiaa**	**r**D,**r**A,**r**B
Multiply Half Words, Even, Guarded, Unsigned, Modulo, Integer and Accumulate Negative	**evmhegumian**	**r**D,**r**A,**r**B
Multiply Half Words, Odd, Guarded, Signed, Modulo, Fractional and Accumulate	**evmhogsmfaa**	**r**D,**r**A,**r**B
Multiply Half Words, Odd, Guarded, Signed, Modulo, Fractional and Accumulate Negative	**evmhogsmfan**	**r**D,**r**A,**r**B
Multiply Half Words, Odd, Guarded, Signed, Modulo, Integer and Accumulate	**evmhogsmiaa**	**r**D,**r**A,**r**B
Multiply Half Words, Odd, Guarded, Signed, Modulo, Integer and Accumulate Negative	**evmhogsmian**	**r**D,**r**A,**r**B
Multiply Half Words, Odd, Guarded, Unsigned, Modulo, Integer and Accumulate	**evmhogumiaa**	**r**D,**r**A,**r**B
Multiply Half Words, Odd, Guarded, Unsigned, Modulo, Integer and Accumulate Negative	**evmhogumian**	**r**D,**r**A,**r**B
Vector Absolute Value	**evabs**	**r**D,**r**A
Vector Add Immediate Word	**evaddiw**	**r**D,**r**B,UIMM
Vector Add Signed, Modulo, Integer to Accumulator Word	**evaddsmiaaw**	**r**D,**r**A,**r**B
Vector Add Signed, Saturate, Integer to Accumulator Word	**evaddssiaaw**	**r**D,**r**A
Vector Add Unsigned, Modulo, Integer to Accumulator Word	**evaddumiaaw**	**r**D,**r**A
Vector Add Unsigned, Saturate, Integer to Accumulator Word	**evaddusiaaw**	**r**D,**r**A
Vector Add Word	**evaddw**	**r**D,**r**A,**r**B
Vector AND	**evand**	**r**D,**r**A,**r**B
Vector AND with Complement	**evandc**	**r**D,**r**A,**r**B
Vector Compare Equal	**evcmpeq**	**cr**D,**r**A,**r**B
Vector Compare Greater Than Signed	**evcmpgts**	**cr**D,**r**A,**r**B
Vector Compare Greater Than Unsigned	**evcmpgtu**	**cr**D,**r**A,**r**B
Vector Compare Less Than Signed	**evcmplts**	**cr**D,**r**A,**r**B
Vector Compare Less Than Unsigned	**evcmpltu**	**cr**D,**r**A,**r**B
Vector Convert Floating-Point from Signed Fraction	**evfscfsf**	**r**D,**r**B
Vector Convert Floating-Point from Signed Integer	**evfscfsi**	**r**D,**r**B
Vector Convert Floating-Point from Unsigned Fraction	**evfscfuf**	**r**D,**r**B
Vector Convert Floating-Point from Unsigned Integer	**evfscfui**	**r**D,**r**B
Vector Convert Floating-Point to Signed Fraction	**evfsctsf**	**r**D,**r**B
Vector Convert Floating-Point to Signed Integer	**evfsctsi**	**r**D,**r**B
Vector Convert Floating-Point to Signed Integer with Round toward Zero	**evfsctsiz**	**r**D,**r**B
Vector Convert Floating-Point to Unsigned Fraction	**evfsctuf**	**r**D,**r**B

Table 108. SPE APU vector instructions (continued)

Instruction	Mnemonic	Syntax
Vector Convert Floating-Point to Unsigned Integer	**evfsctui**	**r**D,**r**B
Vector Convert Floating-Point to Unsigned Integer with Round toward Zero	**evfsctuiz**	**r**D,**r**B
Vector Count Leading Sign Bits Word	**evcntlsw**	**r**D,**r**A
Vector Count Leading Zeros Word	**evcntlzw**	**r**D,**r**A
Vector Divide Word Signed	**evdivws**	**r**D,**r**A,**r**B
Vector Divide Word Unsigned	**evdivwu**	**r**D,**r**A,**r**B
Vector Equivalent	**eveqv**	**r**D,**r**A,**r**B
Vector Extend Sign Byte	**evextsb**	**r**D,**r**A
Vector Extend Sign Half Word	**evextsh**	**r**D,**r**A
Vector Floating-Point Absolute Value	**evfsabs**	**r**D,**r**A
Vector Floating-Point Add	**evfsadd**	**r**D,**r**A,**r**B
Vector Floating-Point Compare Equal	**evfscmpeq**	**cr**D,**r**A,**r**B
Vector Floating-Point Compare Greater Than	**evfscmpgt**	**cr**D,**r**A,**r**B
Vector Floating-Point Compare Less Than	**evfscmplt**	**cr**D,**r**A,**r**B
Vector Floating-Point Divide	**evfsdiv**	**r**D,**r**A,**r**B
Vector Floating-Point Multiply	**evfsmul**	**r**D,**r**A,**r**B
Vector Floating-Point Negate	**evfsneg**	**r**D,**r**A
Vector Floating-Point Negative Absolute Value	**evfsnabs**	**r**D,**r**A
Vector Floating-Point Subtract	**evfssub**	**r**D,**r**A,**r**B
Vector Floating-Point Test Equal	**evfststeq**	**cr**D,**r**A,**r**B
Vector Floating-Point Test Greater Than	**evfststgt**	**cr**D,**r**A,**r**B
Vector Floating-Point Test Less Than	**evfststlt**	**cr**D,**r**A,**r**B
Vector Load Double into Half Words	**evldh**	**r**D,**d(r**A**)**
Vector Load Double into Half Words Indexed	**evldhx**	**r**D,**r**A,**r**B
Vector Load Double into Two Words	**evldw**	**r**D,**d(r**A**)**
Vector Load Double into Two Words Indexed	**evldwx**	**r**D,**r**A,**r**B
Vector Load Double Word into Double Word	**evldd**	**r**D,**d(r**A**)**
Vector Load Double Word into Double Word Indexed	**evlddx**	**r**D,**r**A,**r**B
Vector Load Half Word into Half Word Odd Signed and Splat	**evlhhossplat**	**r**D,**d(r**A**)**
Vector Load Half Word into Half Word Odd Signed and Splat Indexed	**evlhhossplatx**	**r**D,**r**A,**r**B
Vector Load Half Word into Half Word Odd Unsigned and Splat	**evlhhousplat**	**r**D,**d(r**A**)**
Vector Load Half Word into Half Word Odd Unsigned and Splat Indexed	**evlhhousplatx**	**r**D,**r**A,**r**B
Vector Load Half Word into Half Words Even and Splat	**evlhhesplat**	**r**D,**d(r**A**)**
Vector Load Half Word into Half Words Even and Splat Indexed	**evlhhesplatx**	**r**D,**r**A,**r**B
Vector Load Word into Half Words and Splat	**evlwhsplat**	**r**D,**d(r**A**)**

Table 108. SPE APU vector instructions (continued)

Instruction	Mnemonic	Syntax
Vector Load Word into Half Words and Splat Indexed	**evlwhsplatx**	**r**D,**r**A,**r**B
Vector Load Word into Half Words Odd Signed (with sign extension)	**evlwhos**	**r**D,**d(r**A**)**
Vector Load Word into Half Words Odd Signed Indexed (with sign extension)	**evlwhosx**	**r**D,**r**A,**r**B
Vector Load Word into Two Half Words Even	**evlwhe**	**r**D,**d(r**A**)**
Vector Load Word into Two Half Words Even Indexed	**evlwhex**	**r**D,**r**A,**r**B
Vector Load Word into Two Half Words Odd Unsigned (zero-extended)	**evlwhou**	**r**D,**d(r**A**)**
Vector Load Word into Two Half Words Odd Unsigned Indexed (zero-extended)	**evlwhoux**	**r**D,**r**A,**r**B
Vector Load Word into Word and Splat	**evlwwsplat**	**r**D,**d(r**A**)**
Vector Load Word into Word and Splat Indexed	**evlwwsplatx**	**r**D,**r**A,**r**B
Vector Merge High	**evmergehi**	**r**D,**r**A,**r**B
Vector Merge High/Low	**evmergehilo**	**r**D,**r**A,**r**B
Vector Merge Low	**evmergelo**	**r**D,**r**A,**r**B
Vector Merge Low/High	**evmergelohi**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Even, Signed, Modulo, Fractional	**evmhesmf**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Even, Signed, Modulo, Fractional and Accumulate into Words	**evmhesmfaaw**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Even, Signed, Modulo, Fractional and Accumulate Negative into Words	**evmhesmfanw**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Even, Signed, Modulo, Fractional, Accumulate	**evmhesmfa**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Even, Signed, Modulo, Integer	**evmhesmi**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Even, Signed, Modulo, Integer and Accumulate into Words	**evmhesmiaaw**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Even, Signed, Modulo, Integer and Accumulate Negative into Words	**evmhesmianw**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Even, Signed, Modulo, Integer, Accumulate	**evmhesmia**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Even, Signed, Saturate, Fractional	**evmhessf**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Even, Signed, Saturate, Fractional and Accumulate into Words	**evmhessfaaw**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Even, Signed, Saturate, Fractional and Accumulate Negative into Words	**evmhessfanw**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Even, Signed, Saturate, Fractional, Accumulate	**evmhessfa**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Even, Signed, Saturate, Integer and Accumulate into Words	**evmhessiaaw**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Even, Signed, Saturate, Integer and Accumulate Negative into Words	**evmhessianw**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Even, Unsigned, Modulo, Integer	**evmheumi**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Even, Unsigned, Modulo, Integer and Accumulate into Words	**evmheumiaaw**	**r**D,**r**A,**r**B

Table 108. SPE APU vector instructions (continued)

Instruction	Mnemonic	Syntax
Vector Multiply Half Words, Even, Unsigned, Modulo, Integer and Accumulate Negative into Words	**evmheumianw**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Even, Unsigned, Modulo, Integer, Accumulate	**evmheumia**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Even, Unsigned, Saturate, Integer and Accumulate into Words	**evmheusiaaw**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Even, Unsigned, Saturate, Integer and Accumulate Negative into Words	**evmheusianw**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Odd, Signed, Modulo, Fractional	**evmhosmf**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Odd, Signed, Modulo, Fractional and Accumulate into Words	**evmhosmfaaw**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Odd, Signed, Modulo, Fractional and Accumulate Negative into Words	**evmhosmfanw**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Odd, Signed, Modulo, Fractional, Accumulate	**evmhosmfa**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Odd, Signed, Modulo, Integer	**evmhosmi**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Odd, Signed, Modulo, Integer and Accumulate into Words	**evmhosmiaaw**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Odd, Signed, Modulo, Integer and Accumulate Negative into Words	**evmhosmianw**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Odd, Signed, Modulo, Integer, Accumulate	**evmhosmia**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Odd, Signed, Saturate, Fractional	**evmhossf**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Odd, Signed, Saturate, Fractional and Accumulate into Words	**evmhossfaaw**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Odd, Signed, Saturate, Fractional and Accumulate Negative into Words	**evmhossfanw**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Odd, Signed, Saturate, Fractional, Accumulate	**evmhossfa**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Odd, Signed, Saturate, Integer and Accumulate into Words	**evmhossiaaw**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Odd, Signed, Saturate, Integer and Accumulate Negative into Words	**evmhossianw**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Odd, Unsigned, Modulo, Integer	**evmhoumi**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Odd, Unsigned, Modulo, Integer and Accumulate into Words	**evmhoumiaaw**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Odd, Unsigned, Modulo, Integer and Accumulate Negative into Words	**evmhoumianw**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Odd, Unsigned, Modulo, Integer, Accumulate	**evmhoumia**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Odd, Unsigned, Saturate, Integer and Accumulate into Words	**evmhousiaaw**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Odd, Unsigned, Saturate, Integer and Accumulate Negative into Words	**evmhousianw**	**r**D,**r**A,**r**B
Vector Multiply Word High Signed, Modulo, Fractional	**evmwhsmf**	**r**D,**r**A,**r**B

Table 108. SPE APU vector instructions (continued)

Instruction	Mnemonic	Syntax
Vector Multiply Word High Signed, Modulo, Fractional and Accumulate	**evmwhsmfa**	**r**D,**r**A,**r**B
Vector Multiply Word High Signed, Modulo, Integer	**evmwhsmi**	**r**D,**r**A,**r**B
Vector Multiply Word High Signed, Modulo, Integer and Accumulate	**evmwhsmia**	**r**D,**r**A,**r**B
Vector Multiply Word High Signed, Saturate, Fractional	**evmwhssf**	**r**D,**r**A,**r**B
Vector Multiply Word High Signed, Saturate, Fractional and Accumulate	**evmwhssfa**	**r**D,**r**A,**r**B
Vector Multiply Word High Unsigned, Modulo, Integer	**evmwhumi**	**r**D,**r**A,**r**B
Vector Multiply Word High Unsigned, Modulo, Integer and Accumulate	**evmwhumia**	**r**D,**r**A,**r**B
Vector Multiply Word Low Signed, Modulo, Integer and Accumulate in Words	**evmwlsmiaaw**	**r**D,**r**A,**r**B
Vector Multiply Word Low Signed, Modulo, Integer and Accumulate Negative in Words	**evmwlsmianw**	**r**D,**r**A,**r**B
Vector Multiply Word Low Signed, Saturate, Integer and Accumulate in Words	**evmwlssiaaw**	**r**D,**r**A,**r**B
Vector Multiply Word Low Signed, Saturate, Integer and Accumulate Negative in Words	**evmwlssianw**	**r**D,**r**A,**r**B
Vector Multiply Word Low Unsigned, Modulo, Integer	**evmwlumi**	**r**D,**r**A,**r**B
Vector Multiply Word Low Unsigned, Modulo, Integer and Accumulate	**evmwlumia**	**r**D,**r**A,**r**B
Vector Multiply Word Low Unsigned, Modulo, Integer and Accumulate in Words	**evmwlumiaaw**	**r**D,**r**A,**r**B
Vector Multiply Word Low Unsigned, Modulo, Integer and Accumulate Negative in Words	**evmwlumianw**	**r**D,**r**A,**r**B
Vector Multiply Word Low Unsigned, Saturate, Integer and Accumulate in Words	**evmwlusiaaw**	**r**D,**r**A,**r**B
Vector Multiply Word Low Unsigned, Saturate, Integer and Accumulate Negative in Words	**evmwlusianw**	**r**D,**r**A,**r**B
Vector Multiply Word Signed, Modulo, Fractional	**evmwsmf**	**r**D,**r**A,**r**B
Vector Multiply Word Signed, Modulo, Fractional and Accumulate	**evmwsmfa**	**r**D,**r**A,**r**B
Vector Multiply Word Signed, Modulo, Fractional and Accumulate	**evmwsmfaa**	**r**D,**r**A,**r**B
Vector Multiply Word Signed, Modulo, Fractional and Accumulate Negative	**evmwsmfan**	**r**D,**r**A,**r**B
Vector Multiply Word Signed, Modulo, Integer	**evmwsmi**	**r**D,**r**A,**r**B
Vector Multiply Word Signed, Modulo, Integer and Accumulate	**evmwsmia**	**r**D,**r**A,**r**B
Vector Multiply Word Signed, Modulo, Integer and Accumulate	**evmwsmiaa**	**r**D,**r**A,**r**B
Vector Multiply Word Signed, Modulo, Integer and Accumulate Negative	**evmwsmian**	**r**D,**r**A,**r**B
Vector Multiply Word Signed, Saturate, Fractional	**evmwssf**	**r**D,**r**A,**r**B
Vector Multiply Word Signed, Saturate, Fractional and Accumulate	**evmwssfa**	**r**D,**r**A,**r**B
Vector Multiply Word Signed, Saturate, Fractional and Accumulate	**evmwssfaa**	**r**D,**r**A,**r**B
Vector Multiply Word Signed, Saturate, Fractional and Accumulate Negative	**evmwssfan**	**r**D,**r**A,**r**B
Vector Multiply Word Unsigned, Modulo, Integer	**evmwumi**	**r**D,**r**A,**r**B
Vector Multiply Word Unsigned, Modulo, Integer and Accumulate	**evmwumia**	**r**D,**r**A,**r**B

Table 108. SPE APU vector instructions (continued)

Instruction	Mnemonic	Syntax
Vector Multiply Word Unsigned, Modulo, Integer and Accumulate	**evmwumiaa**	**r**D,**r**A,**r**B
Vector Multiply Word Unsigned, Modulo, Integer and Accumulate Negative	**evmwumian**	**r**D,**r**A,**r**B
Vector NAND	**evnand**	**r**D,**r**A,**r**B
Vector Negate	**evneg**	**r**D,**r**A
Vector NOR	**evnor**	**r**D,**r**A,**r**B
Vector OR	**evor**	**r**D,**r**A,**r**B
Vector OR with Complement	**evorc**	**r**D,**r**A,**r**B
Vector Rotate Left Word	**evrlw**	**r**D,**r**A,**r**B
Vector Rotate Left Word Immediate	**evrlwi**	**r**D,**r**A,UIMM
Vector Round Word	**evrndw**	**r**D,**r**A
Vector Select	**evsel**	**r**D,**r**A,**r**B,**cr**S
Vector Shift Left Word	**evslw**	**r**D,**r**A,**r**B
Vector Shift Left Word Immediate	**evslwi**	**r**D,**r**A,UIMM
Vector Shift Right Word Immediate Signed	**evsrwis**	**r**D,**r**A,UIMM
Vector Shift Right Word Immediate Unsigned	**evsrwiu**	**r**D,**r**A,UIMM
Vector Shift Right Word Signed	**evsrws**	**r**D,**r**A,**r**B
Vector Shift Right Word Unsigned	**evsrwu**	**r**D,**r**A,**r**B
Vector Splat Fractional Immediate	**evsplatfi**	**r**D,SIMM
Vector Splat Immediate	**evsplati**	**r**D,SIMM
Vector Store Double of Double	**evstdd**	**r**S,**d(r**A**)**
Vector Store Double of Double Indexed	**evstddx**	**r**S,**r**A,**r**B
Vector Store Double of Four Half Words	**evstdh**	**r**S,**d(r**A**)**
Vector Store Double of Four Half Words Indexed	**evstdhx**	**r**S,**r**A,**r**B
Vector Store Double of Two Words	**evstdw**	**r**S,**d(r**A**)**
Vector Store Double of Two Words Indexed	**evstdwx**	**r**S,**r**A,**r**B
Vector Store Word of Two Half Words from Even	**evstwhe**	**r**S,**d(r**A**)**
Vector Store Word of Two Half Words from Even Indexed	**evstwhex**	**r**S,**r**A,**r**B
Vector Store Word of Two Half Words from Odd	**evstwho**	**r**S,**d(r**A**)**
Vector Store Word of Two Half Words from Odd Indexed	**evstwhox**	**r**S,**r**A,**r**B
Vector Store Word of Word from Even	**evstwwex**	**r**S,**d(r**A**)**
Vector Store Word of Word from Even Indexed	**evstwwex**	**r**S,**r**A,**r**B
Vector Store Word of Word from Odd	**evstwwo**	**r**S,**d(r**A**)**
Vector Store Word of Word from Odd Indexed	**evstwwox**	**r**S,**r**A,**r**B
Vector Subtract from Word	**evsubfw**	**r**D,**r**A,**r**B
Vector Subtract Immediate from Word	**evsubifw**	**r**D,UIMM,**r**B

Table 108. SPE APU vector instructions (continued)

Instruction	Mnemonic	Syntax
Vector Subtract Signed, Modulo, Integer to Accumulator Word	**evsubfsmiaaw**	**r**D,**r**A
Vector Subtract Signed, Saturate, Integer to Accumulator Word	**evsubfssiaaw**	**r**D,**r**A
Vector Subtract Unsigned, Modulo, Integer to Accumulator Word	**evsubfumiaaw**	**r**D,**r**A
Vector Subtract Unsigned, Saturate, Integer to Accumulator Word	**evsubfusiaaw**	**r**D,**r**A
Vector XOR	**evxor**	**r**D,**r**A,**r**B

Embedded vector and scalar floating-point APU instructions

The embedded floating-point operations are IEEE-compliant with software exception handlers and offer a simpler exception model than the floating-point instructions defined by the PowerPC ISA. Instead of FPRs, these instructions use GPRs to offer improved performance for converting between floating-point, integer, and fractional values. Sharing GPRs allows vector floating-point instructions to use SPE load and store instructions.

The SPFP APUs are described as follows:

- Vector SPFP instructions operate on a vector of two 32-bit, single-precision floating-point numbers that reside in the upper and lower halves of the 64-bit GPRs. These instructions are listed in *Table 117* alongside their scalar equivalents.
- Scalar SPFP instructions operate on single 32-bit operands that reside in the lower 32-bits of the GPRs. These instructions are listed in *Table 117*.
- Scalar DPFP instructions operate on single 64-bit double-precision operands that reside in the 64-bit GPRs. These instructions are listed in *Table 109*.

Note: Note that the vector and scalar versions of the instructions have the same syntax.

Table 109. Vector and scalar floating-point APU instructions

Instruction	Single-precision		Double-precision scalar	Syntax
	Scalar	Vector		
Convert Floating-Point Double- from Single-Precision	—	—	efdcfs	**r**D,**r**B
Convert Floating-Point from Signed Fraction	**efscfsf**	**evfscfsf**	**efdcfsf**	**r**D,**r**B
Convert Floating-Point from Signed Integer	**efscfsi**	**evfscfsi**	**efdcfsi**	**r**D,**r**B
Convert Floating-Point from Unsigned Fraction	**efscfuf**	**evfscfuf**	**efdcfuf**	**r**D,**r**B
Convert Floating-Point from Unsigned Integer	**efscfui**	**evfscfui**	**efdcfui**	**r**D,**r**B
Convert Floating-Point Single- from Double-Precision	—	—	efscfd	**r**D,**r**B
Convert Floating-Point to Signed Fraction	**efsctsf**	**evfsctsf**	**efdctsf**	**r**D,**r**B
Convert Floating-Point to Signed Integer	**efsctsi**	**evfsctsi**	**efdctsi**	**r**D,**r**B
Convert Floating-Point to Signed Integer with Round toward Zero	**efsctsiz**	**evfsctsiz**	**efdctsiz**	**r**D,**r**B
Convert Floating-Point to Unsigned Fraction	**efsctuf**	**evfsctuf**	**efdctuf**	**r**D,**r**B
Convert Floating-Point to Unsigned Integer	**efsctui**	**evfsctui**	**efdctui**	**r**D,**r**B
Convert Floating-Point to Unsigned Integer with Round toward Zero	**efsctuiz**	**evfsctuiz**	**efdctuiz**	**r**D,**r**B

Table 109. Vector and scalar floating-point APU instructions (continued)

Instruction	Single-precision		Double-precision scalar	Syntax
	Scalar	Vector		
Floating-Point Absolute Value	**efsabs** (1)	**evfsabs**	**efdabs**	**r**D,**r**A
Floating-Point Add	**efsadd**	**evfsadd**	**efdadd**	**r**D,**r**A,**r**B
Floating-Point Compare Equal	**efscmpeq**	**evfscmpeq**	**efdcmpeq**	**cr**D,**r**A,**r**B
Floating-Point Compare Greater Than	**efscmpgt**	**evfscmpgt**	**efdcmpgt**	**cr**D,**r**A,**r**B
Floating-Point Compare Less Than	**efscmplt**	**evfscmplt**	**efdcmplt**	**cr**D,**r**A,**r**B
Floating-Point Divide	**efsdiv**	**evfsdiv**	**efddiv**	**r**D,**r**A,**r**B
Floating-Point Multiply	**efsmul**	**evfsmul**	**efdmul**	**r**D,**r**A,**r**B
Floating-Point Negate	**efsneg** 1	**evfsneg**	**efdneg**	**r**D,**r**A
Floating-Point Negative Absolute Value	**efsnabs** 1	**evfsnabs**	**efdnabs**	**r**D,**r**A
Floating-Point Subtract	**efssub**	**evfssub**	**efdsub**	**r**D,**r**A,**r**B
Floating-Point Test Equal	**efststeq**	**evfststeq**	**efdtsteq**	**cr**D,**r**A,**r**B
Floating-Point Test Greater Than	**efststgt**	**evfststgt**	**efdtstgt**	**cr**D,**r**A,**r**B
Floating-Point Test Less Than	**efststlt**	**evfststlt**	**efdtstlt**	**cr**D,**r**A,**r**B
SPE Double Word Load/Store Instructions				
Vector Load Double Word into Double Word	—	**evldd**	**evldd**	**r**D,**d(r**A**)**
Vector Load Double Word into Double Word Indexed	—	**evlddx**	**evlddx**	**r**D,**r**A,**r**B
Vector Merge High	—	**evmergehi**	**evmergehi**	**r**D,**r**A,**r**B
Vector Merge Low	—	**evmergelo**	**evmergelo**	**r**D,**r**A,**r**B
Vector Store Double of Double	—	**evstdd**	**evstdd**	**r**S,**d(r**A**)**
Vector Store Double of Double Indexed	—	**evstddx**	**evstddx**	**r**S,**r**A,**r**B

On some cores, floating-point operations that produce a result of zero may generate an incorrect sign.

1. Exception detection for these instructions is implementation dependent. On some devices, Infinities, NaNs, and Denorms are always be treated as Norms. No exceptions are taken if SPEFSCR[FINVE] = 1.

3.6.2 Integer select (isel) APU

The integer select APU consists of the **isel** instruction, a conditional register move that helps eliminate branches. *Section 7.1: Integer select APU*," describes the use of **isel**.

Table 110. Integer select APU instruction

Name	Mnemonic	Syntax
Integer Select	isel	**r**D,**r**A,**r**B,**cr**B

3.6.3 Performance monitor APU

The EIS defines the performance monitor as an APU. Software communication with the performance monitor APU is achieved through performance monitor registers (PMRs) rather

than SPRs. New instructions are provided to move to and from these PMRs. Performance monitor APU instructions are described in *Table 111*.

Table 111. Performance monitor APU instructions

Name	Mnemonic	Syntax
Move from performance monitor register	**mfpmr**	**r**D,PMRN
Move to performance monitor register	**mtpmr**	PMRN,**r**S

The Book E implementation standards defines a set of register resources used exclusively by the performance monitor. PMRs are similar to the SPRs defined in the Book E architecture and are accessed by **mtpmr** and **mfpmr**, which are also defined by the EIS. *Table 112* lists supervisor-level PMRs. User-level software that attempts to read or write supervisor-level PMRs causes a privilege exception.

Table 112. Performance monitor registers—supervisor level

Abbreviation	Register name	PMR number	pmr[0–4]	pmr[5–9]	Section/page
PMGC0	Performance monitor global control register 0	400	01100	10000	*Chapter 2.16.1*
PMLCa0	Performance monitor local control a0	144	00100	10000	*Chapter 2.16.3*
PMLCa1	Performance monitor local control a1	145	00100	10001	
PMLCa2	Performance monitor local control a2	146	00100	10010	
PMLCa3	Performance monitor local control a3	147	00100	10011	
PMLCb0	Performance monitor local control b0	272	01000	10000	*Chapter 2.16.5*
PMLCb1	Performance monitor local control b1	273	01000	10001	
PMLCb2	Performance monitor local control b2	274	01000	10010	
PMLCb3	Performance monitor local control b3	275	01000	10011	

Table 112. Performance monitor registers—supervisor level (continued)

Abbreviation	Register name	PMR number	pmr[0–4]	pmr[5–9]	Section/page
PMC0	Performance monitor counter 0	16	00000	10000	*Chapter 2.16.7*
PMC1	Performance monitor counter 1	17	00000	10001	
PMC2	Performance monitor counter 2	18	00000	10010	
PMC3	Performance monitor counter 3	19	00000	10011	

User-level PMRs in *Table 113* are read-only and are accessed with **mfpmr**. Attempting to write user-level registers in supervisor or user mode causes an illegal instruction exception.

Table 113. Performance monitor registers—user level (read-only)

Abbreviation	Register Name	PMR Number	pmr[0–4]	pmr[5–9]	Section/Page
UPMGC0	User performance monitor global control register 0	384	01100	00000	*Chapter 2.16.2*
UPMLCa0	User performance monitor local control a0	128	00100	00000	*Chapter 2.16.4*
UPMLCa1	User performance monitor local control a1	129	00100	00001	
UPMLCa2	User performance monitor local control a2	130	00100	00010	
UPMLCa3	User performance monitor local control a3	131	00100	00011	
UPMLCb0	User performance monitor local control b0	256	01000	00000	*Section 2.16.6 on page 129*
UPMLCb1	User performance monitor local control b1	257	01000	00001	
UPMLCb2	User performance monitor local control b2	258	01000	00010	
UPMLCb3	User performance monitor local control b3	259	01000	00011	
UPMC0	User performance monitor counter 0	0	00000	00000	*Section 2.16.8 on page 129*
UPMC1	User performance monitor counter 1	1	00000	00001	
UPMC2	User performance monitor counter 2	2	00000	00010	
UPMC3	User performance monitor counter 3	3	00000	00011	

3.6.4 Cache locking APU

This section describes the instructions in the cache locking APU, which consists of the instructions described in *Table 114*. Lines are locked into the cache by software using a series of touch and lock set 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 **r**A and **r**B operands to these instructions form a EA identifying the line to be locked. The CT field indicates which cache in the cache hierarchy should be targeted. These instructions are similar to the **dcbt**, **dcbtst**, and **icbt** instructions, but locking instructions can 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, lines are unlocked from the cache by software using a series of lock-clear instructions. The following instructions are provided to lock instructions into the instruction cache:

- **dcblc**—Data Cache Block Lock Clear
- **icblc**—Instruction Cache Block Lock Clear

The **r**A and **r**B operands to these instructions form an EA identifying the line to be unlocked. The CT field indicates which cache in the cache hierarchy should be targeted.

Additionally, software may clear all the locks in the cache. For the primary cache, this is accomplished by setting the CLFC (DCLFC, ICLFC) bit in L1CSR0 (L1CSR1).

Cache lines can also be implicitly unlocked in the following ways:

- A locked line is invalidated if it is targeted by a **dcbi**, **dcbf**, or **icbi** instruction.
- A snoop hit on a locked line that requires the line to be invalidated. This can occur because the data the line contains has been modified external to the processor, or another processor has explicitly invalidated the line.
- The entire cache containing the locked line is flash invalidated.

An implementation is not required to unlock lines if data is invalidated in the cache. Although the data may be invalidated (and thus not in the cache), the cache can keep the lock associated with that cache line present and fill the line from the memory subsystem when the next access occurs. If the implementation does not clear locks when the associated line is invalidated, the method of locking is said to be persistent. An implementation may choose to implement locks as persistent or not persistent; the preferred method is persistent.

Table 114. Cache locking APU instructions

Name	Mnemonic	Syntax	Description
Data Cache Block Lock Clear	**dcblc**	CT,**r**A,**r**B	Treated as a load with respect to any memory barriers, synchronization, translation and protection, and debug address comparisons.
Data Cache Block Touch and Lock Set	**dcbtls**	CT,**r**A,**r**B	Treated as a load with respect to any memory barriers, synchronization, translation and protection, and debug address comparisons.

Table 114. Cache locking APU instructions (continued)

Name	Mnemonic	Syntax	Description
Data Cache Block Touch for Store and Lock Set	**dcbtstls**	CT,**r**A,**r**B	It is implementation dependent whether this instruction is treated as a load or store with respect to any memory barriers, synchronization, translation and protection, and debug address comparisons.
Instruction Cache Block Lock Clear	**icblc**	CT,**r**A,**r**B	Treated as a load with respect to any memory barriers, synchronization, translation and protection, and debug address comparisons.
Instruction Cache Block Touch and Lock Set	**icbtls**	CT,**r**A,**r**B	Treated as a load with respect to any memory barriers, synchronization, translation and protection, and debug address comparisons.

The cache-locking APU defines a flash clear for all data cache lock bits (using L1CSR0[CLFR]) and in the instruction cache (using L1CSR1[ICLFR]). This allows system software to clear all data cache locking bits without knowing the addresses of the lines locked.

3.6.5 Machine check APU

The machine check APU defines a separate interrupt type for machine check interrupts. It provides additional save and restore SPRs (MCSRR and MCSRR1). The Return from Machine Check Interrupt instruction (**rfmci**), is described in *Table 115*.

Table 115. Machine check APU instruction

Name	Mnemonic	Syntax	Implementation notes
Return from machine check interrupt	**rfmci**	—	Restores MCSRR0 and MCSRR1 values; context-synchronizing.

3.6.6 VLE extension

This section lists instructions defined or supported by the VLE extension. Unless otherwise noted, instructions that are not prefixed with **e**_ or **se**_ have identical encodings and semantics as in Book E or in the EIS. Book E–defined instructions listed in the tables in this section can be executed when the processor is in VLE mode; Book E instructions not listed cannot.

A complete list of supported instructions is provided in *Instruction listings on page 217*."

Processor control instructions

This section lists processor control instructions that can be executed when a processor is in VLE mode. These instructions are grouped as follows:

- *System linkage instructions on page 201"*
- *Processor control register manipulation instructions on page 202"*
- *Instruction synchronization instruction on page 202*"

System linkage instructions

The **se_sc**, **se_rfi**, **se_rfci**, and **se_rfdi** system linkage instructions, shown in *Table 116*, enable a program to call on the system to perform a service (that is, invoke a system call interrupt), and enable the system to return from performing a service or from processing an interrupt.

Table 116. System linkage instruction set index

Mnemonic	Instruction	Reference
se_sc	System Call	*Page -954*
se_rfci`	Return from critical interrupt	*Page -949*
se_rfdi	Return from debug interrupt	*Page -859*
se_rfi	Return from interrupt	*Page -950*

Processor control register manipulation instructions

In addition to the Book E processor control register manipulation instructions, the VLE extension provides 16-bit forms of instructions to move to/from the LR and CTR, listed in *Table 117*

Table 117. System register manipulation instruction set index

Mnemonic	Instruction	Reference
se_mfctr rX	Move From Count Register	*Page -938*
mfdcr rD,DCRN	Move From Device Control Register	*Book E*
se_mflr rX	Move From Link Register	*Page -939*
mfmsr rD	Move From Machine State Register	*Book E*
mfspr rD,SPRN	Move From Special Purpose Register	*Book E*
se_mtctr rX	Move To Count Register	*Page -942*
mtdcr DCRN,rS	Move To Device Control Register	*Book E*
se_mtlr rX	Move To Link Register	*Page -943*
mtmsr rS	Move To Machine State Register	*Book E*
mtspr SPRN,rS	Move To Special Purpose Register	*Book E*
wrtee rA	Write MSR External Enable	*Book E*
wrteei E	Write MSR External Enable Immediate	*Book E*

Instruction synchronization instruction

Table 118 lists the VLE-defined **se_isync** instruction.

Table 118. Instruction Synchronization Instruction Set Index

Mnemonic	Instruction	Reference
se_isync	Instruction Synchronize	*Page -929*

Branch operation instructions

This section lists branch instructions that can be executed when a processor is in VLE mode. It also describes the registers that support them.

Registers for branch operations

The sections listed in the following describe the registers that support branch operations:

- *Chapter 2.5.1: Condition register (CR) on page 61*"
- *Chapter 2.5.2: Link register (LR) on page 66*"
- *Chapter 2.5.3: Count register (CTR) on page 67*"

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 *Chapter 10.2: Instruction memory addressing modes on page 854*."

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 whether or not 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 E's BO field.

If the BO32 field specifies that the CTR is to be decremented, CTR[32–63] are decremented. If BO[16,32] specifies a condition that must be TRUE or FALSE, that condition is obtained from the contents of CR[BI+32]. (Note that CR bits are numbered 32–63. BI refers to the BI field in the branch instruction encoding. For example, specifying BI = 2 refers to CR[34].)

Encodings for the BO32 field for the VLE extension are shown in *Table 120*.

Table 119. VLE extension BO32 encodings

BO32	Description
00	Branch if the condition is FALSE.
01	Branch if the condition is TRUE.

Table 119. VLE extension BO32 encodings

BO32	Description
10	Decrement CTR[32–63], then branch if the decremented CTR[32–63]≠0.
11	Decrement CTR[32–63], then branch if the decremented CTR[32–63] = 0.

The encoding for the BO16 field for the VLE extension is shown in *Table 120*.

Table 120. VLE extension BO16 encodings

BO16	Description
0	Branch if the condition is FALSE.
1	Branch if the condition is TRUE.

The various branch instructions supported by the VLE extension are shown in *Table 121*.

Table 121. Branch instruction set index

Mnemonic	Instruction	Reference
e_b BD24 **e_bl** BD24	Branch Branch & Link	*Page -903*
se_b BD8 **se_bl** BD8	Branch Branch & Link	*Page -903*
e_bc BO32,BI32,BD15 **se_bc** BO16,BI16,BD8 **e_bcl** BO32,BI32,BD15	Branch Conditional Branch Conditional Branch Conditional & Link	*Page -904*
se_bctr **se_bctrl**	Branch to Count Register Branch to Count Register & Link	*Page -906*
se_blr **se_blrl**	Branch to Link Register Branch to Link Register & Link	*Page -908*

Condition register instructions

Condition register instructions are provided to transfer values to/from various portions of the CR. The VLE extension does not introduce any additional functionality beyond that defined in Book E for CR operations, but does remap the CR-logical and **mcrf** instruction functionality into major opcode 31. These instructions operate identically to the Book E instructions, but are encoded differently. *Table 122* lists condition register instructions supported in VLE mode.

Table 122. Condition register instruction set index

Mnemonic	Instruction	Reference
e_crand crbD,**crb**A,**crb**B	Condition Register AND	*Page -920*
e_crandc crbD,**crb**A,**crb**B	Condition Register AND with Complement	*Page -920*
e_creqv crbD,**crb**A,**crb**B	Condition Register Equivalent	*Page -920*
e_crnand crbD,**crb**A,**crb**B	Condition Register NAND	*Page -921*
e_crnor crbD,**crb**A,**crb**B	Condition Register NOR	*Page -922*

Table 122. Condition register instruction set index

Mnemonic	Instruction	Reference
e_cror crbD,**crb**A,**crb**B	Condition Register OR	*Page -923*
e_crorc crbD,**crb**A,**crb**B	Condition Register OR with Complement	*Page -923*
e_crxor crbD,**crb**A,**crb**B	Condition Register XOR	*Page -925*
e_mcrf crD,**cr**S	Move Condition Register Field	*Page -936*
mcrxr crD	Move to Condition Register from Integer Exception Register	*Book E*
mfcr rD	Move From condition register	*Book E*
mtcrf FXM,**r**S	Move to Condition Register Fields	*Book E*

Integer instructions

This section lists the integer instructions supported by the VLE extension.

Integer load instructions

The integer load instructions, listed in *Table 123*, compute the EA of the memory to be accessed as described in *Chapter 10.1: Data memory addressing modes on page 854*."

The byte, half word, or word in memory addressed by EA is loaded into GPR(**r**D) or GPR(**r**Z).

The VLE extension supports both big- and little-endian byte ordering for data accesses.

Some integer load instructions have an update form in which GPR(**r**A) is updated with the EA. For these forms, if **r**A ≠ 0 and **r**A ≠ **r**D, the EA is placed into GPR(**r**A) and the memory element (byte, half word, word, or double word) addressed by EA is loaded into GPR(**r**D). If **r**A = 0 or **r**A = **r**D, the instruction form is invalid. This is the same behavior as specified for load with update instructions in Book E.

Table 123. Basic integer load instruction set index

Mnemonic	Instruction	Reference
e_lbz rD,D**(r**A**)** **e_lbzu r**D,D8**(r**A**)** **se_lbz r**Z,SD4**(r**X**)**	Load Byte and Zero Load Byte and Zero with Update Load Byte and Zero (16-bit form)	*Page -930*
lbzx rD,**r**A,**r**B **lbzux r**D,**r**A,**r**B	Load Byte and Zero Indexed Load Byte and Zero with Update Indexed	*Book E*
e_lha rD,D**(r**A**)** **e_lhau r**D,D8**(r**A**)**	Load Halfword Algebraic Load Halfword Algebraic with Update	*Page -931*
lhax rD,**r**A,**r**B **lhaux r**D,**r**A,**r**B	Load Halfword Algebraic Indexed Load Halfword Algebraic with Update Indexed	*Book E*

Table 123. Basic integer load instruction set index

Mnemonic	Instruction	Reference
e_lhz rD,D(**r**A) **e_lhzu** rD,D8(**r**A) **se_lhz** rZ,SD4(**r**X)	Load Halfword and Zero Load Halfword and Zero with Update Load Halfword and Zero (16-bit form)	*Page -932*
lhzx rD,**r**A,**r**B **lhzux** rD,**r**A,**r**B	Load Halfword and Zero Indexed Load Halfword and Zero with Update Indexed	*Book E*
e_lwz rD,D(**r**A) **e_lwzu** rD,D8(**r**A) **se_lwz** rZ,SD4(**r**X)	Load Word and Zero Load Word and Zero with Update Load Word and Zero (16-bit form)	*Page -935*
lwzx rD,**r**A,**r**B **lwzux** rD,**r**A,**r**B	Load Word and Zero Indexed Load Word and Zero with Update Indexed	*Book E*

Integer load byte-reversed instructions are listed in *Table 124*.

Table 124. Integer load byte-reverse instruction set index

Mnemonic	Instruction	Reference
lhbrx rD,**r**A,**r**B	Load Halfword Byte-Reverse Indexed	*Book E*
lwbrx rD,**r**A,**r**B	Load Word Byte-Reverse Indexed	*Book E*

The VLE-defined integer load multiple instruction is listed in *Table 125*.

Table 125. Integer load multiple instruction set index

Mnemonic	Instruction	Reference
e_lmw rD,D8(**r**A)	Load Multiple Word	*Page -934*

The VLE-defined integer load and reserve instruction is listed in *Table 126*.

Table 126. Integer load and reserve instruction set index

Mnemonic	Instruction	Reference
lwarx rD,**r**A,**r**B	Load Word And Reserve Indexed	*Book E*

Integer store instructions

The integer store instructions compute the EA of the memory to be accessed as described in *Chapter 10.1: Data memory addressing modes on page 854*."

The contents of GPR(**r**S) or GPR(**r**Z) are stored into the byte, half word, or word in memory addressed by EA.

The VLE extension supports both big- and little-endian byte ordering for data accesses.

Some integer store instructions have an update form, in which GPR(**r**A) is updated with the EA. For these forms, the following rules (from Book E) apply.

- If **r**A ≠ 0, the EA is placed into GPR(**r**A).
- If **r**S = **r**A, the contents of GPR(**r**S) are copied to the target memory element and then EA is placed into GPR(**r**A).

The basic integer store instructions are listed in *Table 127*.

Table 127. Basic integer store instruction set index

Mnemonic	Instruction	Reference
e_stb **r**S,D(**r**A) **e_stbu** **r**S,D8(**r**A) **se_stb** **r**Z,SD4(**r**X)	Store Byte Store Byte with Update Store Byte (16-bit form)	*Page -958*
stbx **r**S,**r**A,**r**B **stbux** **r**S,**r**A,**r**B	Store Byte Indexed Store Byte with Update Indexed	*Book E*
e_sth **r**S,D(**r**A) **e_sthu** **r**S,D8(**r**A) **se_sth** **r**Z,SD4(**r**X)	Store Halfword Store Halfword with Update Store Halfword (16-bit form)	*Page -959*
sthx **r**S,**r**A,**r**B **sthux** **r**S,**r**A,**r**B	Store Halfword Indexed Store Halfword with Update Indexed	*Book E*
e_stw **r**S,D(**r**A) **e_stwu** **r**S,D8(**r**A) **se_stw** **r**Z,SD4(**r**X)	Store Word Store Word with Update Store Word (16-bit form)	*Page -961*
stwx **r**S,**r**A,**r**B **stwux** **r**S,**r**A,**r**B	Store Word Indexed Store Word with Update Indexed	*Book E*

The integer store byte-reverse instructions are listed in *Table 128*.

Table 128. Integer store byte-reverse instruction set index

Mnemonic	Instruction	Reference
sthbrx **r**S,**r**A,**r**B	Store Halfword Byte-Reverse Indexed	*Book E*
stwbrx **r**S,**r**A,**r**B	Store Word Byte-Reverse Indexed	*Book E*

The integer store multiple instruction is listed in *Table 129*.

Table 129. Integer store multiple instruction set index

Mnemonic	Instruction	Reference
e_stmw **r**S,D8(**r**A)	Store Multiple Word	*Page -960*

The integer store conditional instruction is listed in *Table 130*.

Table 130. Integer store conditional instruction set index

Mnemonic	Instruction	Reference
stwcx. **r**S,**r**A,**r**B	Store Word Conditional Indexed	Book E

Integer arithmetic instructions

The integer 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 integer arithmetic instructions treat source operands as signed, two's complement integers unless the instruction is explicitly identified as performing an unsigned operation.

The **e_add2i.** instruction and the OIM5-form instruction, **se_subi.**, set the first three bits of CR0 to characterize bits 32–63 of the result. These bits are set by signed comparison of bits 32–63 of the result to zero.

e_addic[.] and **e_subfic**[.] always set CA to reflect the carry out of bit 32.

The integer arithmetic instructions are listed in *Table 131*.

Table 131. Integer arithmetic instruction set index

Mnemonic	Instruction	Reference
add rD,rA,rB **add.** rD,rA,rB **addo** rD,rA,rB **addo.** rD,rA,rB	Add	*Book E*
se_add rX,rY	Add	*Page -897*
addc rD,rA,rB **addc.** rD,rA,rB **addco** rD,rA,rB **addco.** rD,rA,rB	Add Carrying	*Book E*
adde rD,rA,rB **adde.** rD,rA,rB **addeo** rD,rA,rB **addeo.** rD,rA,rB	Add Extended	*Book E*
e_addi rD,rA,SCI8 **e_addi.** rD,rA,SCI8 **e_add16i** rD,rA,SI **e_add2i.** rD,SI **se_addi** rX,OIMM	Add Immediate	*Page -898*
e_addic rD,rA,SCI8 **e_addic.** rD,rA,SCI8	Add Immediate Carrying	*Page -900*
e_add2is rD,SI	Add Immediate Shifted	*Page -898*
divw rD,rA,rB **divw.** rD,rA,rB **divwo** rD,rA,rB **divwo.** rD,rA,rB	Divide Word	*Book E*
divwu rD,rA,rB **divwu.** rD,rA,rB **divwuo** rD,rA,rB **divwuo.** rD,rA,rB	Divide Word Unsigned	*Book E*

Table 131. Integer arithmetic instruction set index (continued)

Mnemonic	Instruction	Reference
mulhw rD,rA,rB **mulhw.** rD,rA,rB	Multiply High Word	*Book E*
mulhwu rD,rA,rB **mulhwu.** rD,rA,rB	Multiply High Word Unsigned	*Book E*
e_mulli rD,rA,SCI8 **e_mull2i** rD,SI	Multiply Low Immediate	*Page -944*
mullw rD,rA,rB **mullw.** rD,rA,rB **mullwo** rD,rA,rB **mullwo.** rD,rA,rB	Multiply Low Word	*Book E*
se_mullw rX,rY	Multiply Low Word	*Page -945*
neg rD,rA **se_neg** rX **neg.** rD,rA **nego** rD,rA **nego.** rD,rA	Negate	*Page -946*
se_sub rX,rY	Subtract	*Page -962*
subf rD,rA,rB **subf.** rD,rA,rB **subfo** rD,rA,rB **subfo.** rD,rA,rB	Subtract From	*Book E*
se_subf rX,rY	Subtract From	*Page -963*
subfc rD,rA,rB **subfc.** rD,rA,rB **subfco** rD,rA,rB **subfco.** rD,rA,rB	Subtract From Carrying	*Book E*
e_subfic rD,rA,SCI8 **e_subfic.** rD,rA,SCI8	Subtract From Immediate Carrying	*Page -964*
se_subi rX,OIMM **se_subi.** rX,OIMM	Subtract Immediate	*Page -965*

Integer logical and move instructions

Logical instructions perform bit-parallel operations on 32-bit operands or move register or immediate values into registers. The move instructions move values into a GP from either another GPR, or an immediate value.

The X-form logical instructions with Rc = 1 and the SCI8-form logical instructions with Rc = 1 set the first three bits of CR field 0 as described in *Integer arithmetic instructions on page 208*." The logical instructions do not change XER[SO,OV,CA].

The integer logical instructions are listed in *Table 132*.

Table 132. Integer logical instruction set index

Mnemonic	Instruction	Reference
and[.] rA,rS,rB **se_and**[.] rX,rY	AND	*Page -901*
andc[.] rA,rS,rB **se_andc** rX,rY	AND with Complement	*Page -901*
e_andi[.] rA,rS,SCI8 **se_andi** rX,UI5 **e_and2i.** rD,UI	AND Immediate	*Page -901*
e_and2is. rD,UI	AND Immediate Shifted	*Page -901*
se_bclri rX,UI5	Bit Clear	*Page -905*
se_bgeni rX,UI5	Bit Generate	*Page -907*
se_bmski rX,UI5	Bit Mask Generate	*Page -909*
se_bseti rX,UI5	Bit Set	*Page -910*
cntlzw rA,rS **cntlzw.** rA,rS	Count Leading Zeros Word	*Book E*
eqv rA,rS,rB **eqv.** rA,rS,rB	Equivalent	*Book E*
extsb rA,rS **extsb.** rA,rS **se_extsb** rX	Extend Sign Byte	*Page -926*
extsh rA,rS **extsh.** rA,rS **se_extsh** rX	Extend Sign Halfword	*Page -926*
se_extzb rX	Extend with Zeros Byte	*Page -927*
se_extzh rX	Extend with Zeros Halfword	*Page -927*
e_li rD,LI20 **se_li** rX,UI7	Load Immediate	*Page -933*
e_lis rD,UI	Load Immediate Shifted	*Page -933*
se_mfar rX,arY	Move from Alternate Register	*Page -937*
se_mr rX,rY	Move Register	*Page -940*
se_mtar arX,rY	Move to Alternate Register	*Page -941*
nand rA,rS,rB **nand.** rA,rS,rB	NAND	*Book E*
nor rA,rS,rB **nor.** rA,rS,rB	NOR	*Book E*
or rA,rS,rB **or.** rA,rS,rB **se_or** rX,rY	OR	*Page -948*
se_not rX	NOT	*Page -947*

Table 132. Integer logical instruction set index (continued)

Mnemonic	Instruction	Reference
orc r**A**,r**S**,r**B** **orc.** r**A**,r**S**,r**B**	OR with Complement	*Book E*
e_ori[.] r**A**,r**S**,SCI8 **e_or2i** r**D**,UI	OR Immediate	*Page -966*
e_or2is r**D**,UI	OR Immediate Shifted	*Page -966*
xor r**A**,r**S**,r**B** **xor.** r**A**,r**S**,r**B**	XOR	*Book E*
e_xori[.] r**A**,r**S**,SCI8	XOR Immediate	*Page -966*

Integer compare and bit test instructions

The integer compare instructions compare the contents of GPR(rA) with one of the following:

- The value of the SCI8 field
- 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 GPR(**r**B) or GPR(**r**Y).

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**.

When operands are treated as 32-bit signed quantities, GPR*n*[32] is the sign bit. When operands are treated as 16-bit signed quantities, GPR*n*[48] is the sign bit.

For 32-bit implementations, the L field must be zero.

Compare instructions set one of the left-most three bits of the designated CR field and clears the other two. XER[SO] is copied to bit 3 of the designated CR field.

The CR field is set as shown in *Table 133*.

Table 133. CR settings for compare instructions

Bit	Name	Description
0	LT	(**r**A or **r**X) < SCI8, SI, UI5, or GPR(**r**B or **r**Y) (signed comparison) (**r**A or **r**X) $<_u$ SCI8, UI, UI5 or GPR(**r**B or **r**Y) (unsigned comparison)
1	GT	(**r**A or **r**X) > SCI8, SI, UI5, or GPR(**r**B or **r**Y) (signed comparison) (**r**A or **r**X) $>_u$ SCI8, UI, UI5 or GPR(**r**B or **r**Y) (unsigned comparison)
2	EQ	(**r**A or **r**X) = SCI8, SI, UI, UI5, or GPR(**r**B or **r**Y)
3	SO	Summary overflow from the XER

The integer bit test instruction tests the bit specified by the UI5 instruction field and sets the CR0 field as shown in *Table 134*.

Table 134. CR settings for integer bit test instructions

Bit	Name	Description
0	LT	Always cleared
1	GT	RX_{ui5} == 1
2	EQ	RX_{ui5} == 0
3	SO	Summary overflow from the XER

Table 135 is an index for integer compare and bit test operations.

Table 135. Integer compare and bit test instruction set index

Mnemonic	Instruction	Reference
se_btsti rX,UI5	Bit Test Immediate	*Page -911*
cmp crD,L,rA,rB **se_cmp rX,rY**	Compare	*Page -912*
e_cmph crD,rA,rB **se_cmph rX,rY**	Compare Halfword	*Page -914*
e_cmph16i rA,SI16	Compare Halfword Immediate	*Page -914*
e_cmphl crD,rA,rB **se_cmphl rX,rY**	Compare Halfword Logical	*Page -916*
e_cmphl16i rA,UI16	Compare Halfword Logical Immediate	*Page -916*
e_cmpi crD,rA,SCI8 **e_cmp16i rA**,SI16 **se_cmpi rX**,UI5	Compare Immediate	*Page -912*
cmpl crD,L,rA,rB **se_cmpl rX,rY**	Compare Logical	*Page -918*
e_cmpli crD,rA,SCI8 **e_cmpl16i rA**,UI16 **se_cmpli rX**,UI5	Compare Logical Immediate	*Page -918*

Integer select instruction

The **isel** 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 integer select instruction is listed in *Table 136*.

Table 136. Integer select instruction set index

Mnemonic	Instruction	Reference
isel rD,rA,rB,crb	Integer Select	EIS

Integer trap instructions

Trap instructions test for a specified set of conditions by comparing the contents of one GPR with a second GPR. If any of the conditions tested by a Trap instruction are met, a trap

exception type program interrupt is invoked. If none of the tested conditions are met, instruction execution continues normally.

The contents of GPR(**r**A) are compared with the contents of GPR(**r**B). For **twi** and **tw**, only the contents of bits 32–63 of **r**A (and **r**B) participate in the comparison.

This comparison results in five conditions that are ANDed with TO. If the result is not 0, the trap exception type program interrupt is invoked. These conditions are as shown in *Table 137*.

Table 137. Integer trap conditions

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

The integer trap instruction is listed in *Table 138*.

Table 138. Integer trap instruction set index

Mnemonic	Instruction	Reference
tw TO,**r**A,**r**B	Trap Word	Book E

Integer rotate and shift instructions

Instructions are provided that perform shifts and rotates on data from a GPR and return the result, or a portion of the result, to a GPR.

The rotation operations rotate a 32-bit quantity left by a specified number of bit positions. Bits that exit from position 32 enter at position 63.

The $rotate_{32}$ operation is used to rotate a given 32-bit quantity.

Some rotate and shift instructions employ a mask generator. The mask is 32 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 32 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

$mask_{32:mstop}$ = 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 bits 32–63 of the mask.

The use of the mask is described in following sections.

The rotate word and shift word instructions with Rc = 1 set the first three bits of CR field 0 as described in Book E. 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.

The instructions in *Table 139* rotate the contents of a register. Depending on the instruction type, the amount of the rotation is either specified as an immediate, or contained in a GPR.

Table 139. Integer rotate instruction set index

Mnemonic	Instruction	Reference
e_rlw r**A**,r**S**,r**B**	Rotate Left Word	*Page -951*
e_rlwi r**A**,r**S**,SH	Rotate Left Word Immediate	*Page -951*

The instructions in *Table 140* rotate the contents of a register. Depending on the instruction type, the result of the rotation is either 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; if a 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 32-*n*, where *n* is the number of bits by which to rotate right. They allow right-rotation of the contents of bits 32–63 of a register to be performed (in concept) by a left-rotation of 32-*n*, where *n* is the number of bits by which to rotate right.

Table 140. Integer rotate with mask instruction set index

Mnemonic	Instruction	Reference
e_rlwimi r**A**,r**S**,SH,MB,ME	Rotate Left Word Immediate then Mask Insert	*Page -952*
e_rlwinm r**A**,r**S**,SH,MB,ME	Rotate Left Word Immediate then AND with Mask	*Page -953*

The integer shift instructions are listed in *Table 141*.

Table 141. Integer shift instruction set index

Mnemonic	Instruction	Reference
slw r**A**,r**S**,r**B** **slw.** r**A**,r**S**,r**B** **se_slw** r**X**,r**Y**	Shift Left Word	*Page -955*
e_slwi r**A**,r**S**,SH **se_slwi** r**X**,UI5	Shift Left Word Immediate	*Page -955*
sraw r**A**,r**S**,r**B** **sraw.** r**A**,r**S**,r**B** **se_sraw** r**X**,r**Y**	Shift Right Algebraic Word	*Page -956*
srawi r**A**,r**S**,SH **srawi.** r**A**,r**S**,SH **se_srawi** r**X**,UI5	Shift Right Algebraic Word Immediate	*Page -956*

Table 141. Integer shift instruction set index

Mnemonic	Instruction	Reference
srw rA,rS,rB **srw.** rA,rS,rB **se_srw** rX,rY	Shift Right Word	*Page -957*
e_srwi rA,rS,SH **se_srwi** rX,UI5	Shift Right Word Immediate	*Page -957*

Storage control instructions

This section lists storage control instructions, which include the following:

Storage synchronization instructions on page 216”

Cache management instructions on page 216”

TLb management instructions on page 216”

Storage synchronization instructions

The memory synchronization instructions implemented by the VLE extension are identical to those defined in Book E.

The storage synchronization instructions are listed in *Table 142*.

Table 142. Storage synchronization instruction set index

Mnemonic	Instruction	Reference
mbar	Memory Barrier	Book E
msync	Memory Synchronize	Book E

Cache management instructions

Cache management instructions implemented by the VLE extension are identical to those defined in Book E.

The cache management instructions are listed in *Table 143*.

Table 143. Cache management instruction set index

Mnemonic	Instruction	Reference
dcba r**A**,r**B**	Data Cache Block Allocate	Book E
dcbf r**A**,r**B**	Data Cache Block Flush	Book E
dcbi r**A**,r**B**	Data Cache Block Invalidate	Book E
dcbst r**A**,r**B**	Data Cache Block Store	Book E
dcbt CT,r**A**,r**B**	Data Cache Block Touch	Book E
dcbtst CT,r**A**,r**B**	Data Cache Block Touch for Store	Book E
dcbz r**A**,r**B**	Data Cache Block set to Zero	Book E
icbi r**A**,r**B**	Instruction Cache Block Invalidate	Book E
icbt CT,r**A**,r**B**	Instruction Cache Block Touch	Book E

TLb management instructions

The TLB management instructions implemented by the VLE extension are identical to those defined in Book E and in the EIS. The TLB management instructions are listed in *Table 144*.

Table 144. TLB management instruction set index

Mnemonic	Instruction	Reference
tlbivax rA,rB	TLB Invalidate Virtual Address Indexed	Book E
tlbre	TLB Read Entry	Book E
tlbsx rA,rB	TLB Search Indexed	Book E
tlbsync	TLB Synchronize	Book E
tlbwe	TLB Write Entry	Book E

VLE instruction alignment and byte ordering

An instruction fetched from memory must be placed in the pipeline with its bytes in the proper order. Otherwise, the instruction decoder cannot recognize it. Book E allows instructions to be placed into memory marked as either big- or little-endian. This is manageable because Book E instructions are always word-size aligned on word boundaries. The VLE extension includes both half-word– and word-length instructions are aligned on half-word boundaries. Because of this, only big-endian instruction memory is supported when executing from a page of VLE instructions. Attempts to execute VLE instructions from a page marked as little-endian generate an instruction storage interrupt byte-ordering exception.

Instruction listings

This section lists instructions either defined or supported by the VLE extension.

Table 145 lists instructions by instruction name.

Table 145. Instructions listed by name

Instruction	Mnemonic	Reference
Add	**add** rD,**r**A,**r**B **add.** rD,**r**A,**r**B **addo** rD,**r**A,**r**B **addo.** rD,**r**A,**r**B	Book E
Add Carrying	**addc** rD,**r**A,**r**B **addc.** rD,**r**A,**r**B **addco** rD,**r**A,**r**B **addco.** rD,**r**A,**r**B	Book E
Add Extended	**adde** rD,**r**A,**r**B **adde.** rD,**r**A,**r**B **addeo** rD,**r**A,**r**B **addeo.** rD,**r**A,**r**B	Book E
AND with Complement	**andc**[.] rA,**r**S,**r**B **se_andc** rX,**r**Y	Book E *Page -901*
AND	**and**[.] rA,**r**S,**r**B **se_and**[.] rX,**r**Y	Book E *Page -901*
Compare	**cmp** crD,L,**r**A,**r**B **se_cmp** rX,**r**Y	Book E *Page -912*

Table 145. Instructions listed by name (continued)

Instruction	Mnemonic	Reference
Compare Logical	**cmpl cr**D,L,**r**A,**r**B **se_cmpl r**X,**r**Y	Book E *Page -918*
Count Leading Zeros Word	**cntlzw r**A,**r**S **cntlzw. r**A,**r**S	Book E
Data Cache Block Allocate	**dcba r**A,**r**B	Book E
Data Cache Block Flush	**dcbf r**A,**r**B	Book E
Data Cache Block Invalidate	**dcbi r**A,**r**B	Book E
Data Cache Block Store	**dcbst r**A,**r**B	Book E
Data Cache Block Touch	**dcbt** CT,**r**A,**r**B	Book E
Data Cache Block Touch for Store	**dcbtst** CT,**r**A,**r**B	Book E
Data Cache Block set to Zero	**dcbz r**A,**r**B	Book E
Divide Word	**divw r**D,**r**A,**r**B **divw. r**D,**r**A,**r**B **divwo r**D,**r**A,**r**B **divwo. r**D,**r**A,**r**B	Book E
Divide Word Unsigned	**divwu r**D,**r**A,**r**B **divwu. r**D,**r**A,**r**B **divwuo r**D,**r**A,**r**B **divwuo. r**D,**r**A,**r**B	Book E
Equivalent	**eqv r**A,**r**S,**r**B **eqv. r**A,**r**S,**r**B	Book E
Extend Sign Byte	**extsb r**A,**r**S **extsb. r**A,**r**S **se_extsb r**X	Book E Book E *Page -926*
Extend Sign Halfword	**extsh r**A,**r**S **extsh. r**A,**r**S **se_extsh r**X	Book E Book E *Page -926*
Add Immediate Shifted	**e_add2is r**D,SI	*Page -897*
Add Immediate	**e_addi r**D,**r**A,SCI8 **e_addi. r**D,**r**A,SCI8 **e_add16i r**D,**r**A,SI **e_add2i. r**D,SI **se_addi r**X,OIMM	*Page -897*
Add Immediate Carrying	**e_addic r**D,**r**A,SCI8 **e_addic. r**D,**r**A,SCI8	*Page -900*
AND Immediate Shifted	**e_and2is. r**D,UI	*Page -901*
AND Immediate	**e_andi**[.] **r**A,**r**S,SCI8 **se_andi r**X,UI5 **e_and2i. r**D,UI	*Page -901*

Table 145. Instructions listed by name (continued)

Instruction	Mnemonic	Reference
Branch Conditional Branch Conditional Branch Conditional & Link	**e_bc** BO32,BI32,BD15 **se_bc** BO16,BI16,BD8 **e_bcl** BO32,BI32,BD15	*Page -904*
Branch Branch & Link	**e_b** BD24 **e_bl** BD24	*Page -903*
Compare Halfword	**e_cmph crD,rA,rB** **se_cmph rX,rY**	*Page -914*
Compare Halfword Immediate	**e_cmph16i rA**,SI16	*Page -914*
Compare Halfword Logical	**e_cmphl crD,rA,rB** **se_cmphl rX,rY**	*Page -916*
Compare Halfword Logical Immediate	**e_cmphl16i rA**,UI16	*Page -916*
Compare Immediate	**e_cmpi crD,rA**,SCI8 **e_cmp16i rA**,SI16 **se_cmpi rX**,UI5	*Page -912*
Compare Logical Immediate	**e_cmpli crD,rA**,SCI8 **e_cmpl16i rA**,UI16 **se_cmpli rX**,UI5	*Page -918*
Condition Register AND	**e_crand crbD,crbA,crbB**	*Page -920*
Condition Register AND with Complement	**e_crandc crbD,crbA,crbB**	*Page -920*
Condition Register Equivalent	**e_creqv crbD,crbA,crbB**	*Page -920*
Condition Register NAND	**e_crnand crbD,crbA,crbB**	*Page -921*
Condition Register NOR	**e_crnor crbD,crbA,crbB**	*Page -922*
Condition Register OR	**e_cror crbD,crbA,crbB**	*Page -923*
Condition Register OR with Complement	**e_crorc crbD,crbA,crbB**	*Page -924*
Condition Register XOR	**e_crxor crbD,crbA,crbB**	*Page -925*
Load Byte and Zero Load Byte and Zero with Update Load Byte and Zero (16-bit form)	**e_lbz rD**,D**(rA)** **e_lbzu rD**,D8**(rA)** **se_lbz rZ**,SD4**(rX)**	*Page -930*
Load Halfword Algebraic Load Halfword Algebraic with Update	**e_lha rD**,D**(rA)** **e_lhau rD**,D8**(rA)**	*Page -931*
Load Halfword and Zero Load Halfword and Zero with Update Load Halfword and Zero (16-bit form)	**e_lhz rD**,D**(rA)** **e_lhzu rD**,D8**(rA)** **se_lhz rZ**,SD4**(rX)**	*Page -932*
Load Immediate	**e_li rD**,LI20 **se_li rX**,UI7	*Page -933*
Load Immediate Shifted	**e_lis rD**,UI	*Page -933*
Load Multiple Word	**e_lmw rD**,D8**(rA)**	*Page -934*

Table 145. Instructions listed by name (continued)

Instruction	Mnemonic	Reference
Load Word and Zero Load Word and Zero with Update Load Word and Zero (16-bit form)	**e_lwz** rD,D(**r**A) **e_lwzu** rD,D8(**r**A) **se_lwz** rZ,SD4(**r**X)	*Page -935*
Move Condition Register Field	**e_mcrf cr**D,**cr**S	*Page -936*
Multiply Low Immediate	**e_mulli** rD,**r**A,SCI8 **e_mull2i** rD,SI	*Page -944*
OR Immediate Shifted	**e_or2is** rD,UI	*Page -948*
OR Immediate	**e_ori**[.] rA,**r**S,SCI8 **e_or2i** rD,UI	*Page -948*
Rotate Left Word	**e_rlw** rA,**r**S,**r**B	*Page -951*
Rotate Left Word Immediate	**e_rlwi** rA,**r**S,SH	*Page -951*
Rotate Left Word Immediate then Mask Insert	**e_rlwimi** rA,**r**S,SH,MB,ME	*Page -952*
Rotate Left Word Immediate then AND with Mask	**e_rlwinm** rA,**r**S,SH,MB,ME	*Page -953*
Shift Left Word Immediate	**e_slwi** rA,**r**S,SH **se_slwi** rX,UI5	*Page -955*
Shift Right Word Immediate	**e_srwi** rA,**r**S,SH **se_srwi** rX,UI5	*Page -957*
Store Byte Store Byte with Update Store Byte (16-bit form)	**e_stb** rS,D(**r**A) **e_stbu** rS,D8(**r**A) **se_stb** rZ,SD4(**r**X)	*Page -958*
Store Halfword Store Halfword with Update Store Halfword (16-bit form)	**e_sth** rS,D(**r**A) **e_sthu** rS,D8(**r**A) **se_sth** rZ,SD4(**r**X)	*Page -959*
Store Multiple Word	**e_stmw** rS,D8(**r**A)	*Page -960*
Store Word Store Word with Update Store Word (16-bit form)	**e_stw** rS,D(**r**A) **e_stwu** rS,D8(**r**A) **se_stw** rZ,SD4(**r**X)	*Page -961*
Subtract From Immediate Carrying	**e_subfic** rD,**r**A,SCI8 **e_subfic.** rD,**r**A,SCI8	*Page -964*
XOR Immediate	**e_xori**[.] rA,**r**S,SCI8	*Page -966*
Instruction Cache Block Invalidate	**icbi** rA,**r**B	Book E
Instruction Cache Block Touch	**icbt** CT,**r**A,**r**B	Book E
Integer Select	**isel** rD,**r**A,**r**B,**cr**b	EIS
Load Byte and Zero Indexed Load Byte and Zero with Update Indexed	**lbzx** rD,**r**A,**r**B **lbzux** rD,**r**A,**r**B	Book E
Load Halfword Algebraic Indexed Load Halfword Algebraic with Update Indexed	**lhax** rD,**r**A,**r**B **lhaux** rD,**r**A,**r**B	Book E
Load Halfword Byte-Reverse Indexed	**lhbrx** rD,**r**A,**r**B	Book E

Table 145. Instructions listed by name (continued)

Instruction	Mnemonic	Reference
Load Halfword and Zero Indexed Load Halfword and Zero with Update Indexed	**lhzx** rD,rA,rB **lhzux** rD,rA,rB	Book E
Load Word And Reserve Indexed	**lwarx** rD,rA,rB	Book E
Load Word Byte-Reverse Indexed	**lwbrx** rD,rA,rB	Book E
Load Word and Zero Indexed Load Word and Zero with Update Indexed	**lwzx** rD,rA,rB **lwzux** rD,rA,rB	Book E
Memory Barrier	**mbar**	Book E
Move to Condition Register from Integer Exception Register	**mcrxr** crD	Book E
Move From condition register	**mfcr** rD	Book E
Move From Device Control Register	**mfdcr** rD,DCRN	Book E
Move From Machine State Register	**mfmsr** rD	Book E
Move From Special Purpose Register	**mfspr** rD,SPRN	Book E
Memory Synchronize	**msync**	Book E
Move to Condition Register Fields	**mtcrf** FXM,rS	Book E
Move To Device Control Register	**mtdcr DCRN,**rS	Book E
Move To Machine State Register	**mtmsr** rS	Book E
Move To Special Purpose Register	**mtspr SPRN,**rS	Book E
Multiply High Word	**mulhw** rD,rA,rB **mulhw.** rD,rA,rB	Book E
Multiply High Word Unsigned	**mulhwu** rD,rA,rB **mulhwu.** rD,rA,rB	Book E
Multiply Low Word	**mullw** rD,rA,rB **mullw.** rD,rA,rB **mullwo** rD,rA,rB **mullwo.** rD,rA,rB	Book E
NAND	**nand** rA,rS,rB **nand.** rA,rS,rB	Book E
Negate	**neg** rD,rA **se_neg** rX **neg.** rD,rA **nego** rD,rA **nego.** rD,rA	Book E *Page -946* Book E Book E Book E
NOR	**nor** rA,rS,rB **nor.** rA,rS,rB	Book E
OR	**or** rA,rS,rB **or.** rA,rS,rB **se_or** rX,rY	Book E Book E *Page -948*

Table 145. Instructions listed by name (continued)

Instruction	Mnemonic	Reference
OR with Complement	**orc** rA,rS,rB **orc.** rA,rS,rB	Book E
Add	**se_add** rX,rY	*Page -897*
Bit Clear	**se_bclri** rX,UI5	*Page -905*
Branch to Count Register Branch to Count Register & Link	**se_bctr** **se_bctrl**	*Page -906*
Bit Generate	**se_bgeni** rX,UI5	*Page -907*
Branch to Link Register Branch to Link Register & Link	**se_blr** **se_blrl**	*Page -908*
Bit Mask Generate	**se_bmski** rX,UI5	*Page -909*
Bit Set	**se_bseti** rX,UI5	*Page -910*
Branch Branch & Link	**se_b** BD8 **se_bl** BD8	*Page -903*
Bit Test Immediate	**se_btsti** rX,UI5	*Page -911*
Extend with Zeros Byte	**se_extzb** rX	*Page -927*
Extend with Zeros Halfword	**se_extzh** rX	*Page -927*
Instruction Synchronize	**se_isync**	*Page -929*
Move from Alternate Register	**se_mfar** rX,**ar**Y	*Page -937*
Move From Count Register	**se_mfctr** rX	*Page -938*
Move From Link Register	**se_mflr** rX	*Page -939*
Move Register	**se_mr** rX,rY	*Page -940*
Move to Alternate Register	**se_mtar** arX,rY	*Page -941*
Move To Count Register	**se_mtctr** rX	*Page -942*
Move To Link Register	**se_mtlr** rX	*Page -943*
Multiply Low Word	**se_mullw** rX,rY	*Page -945*
NOT	**se_not** rX	*Page -947*
Subtract	**se_sub** rX,rY	*Page -962*
Subtract From	**se_subf** rX,rY	*Page -963*
Subtract Immediate	**se_subi** rX,OIMM **se_subi.** rX,OIMM	*Page -965*
Shift Left Word	**slw** rA,rS,rB **slw.** rA,rS,rB **se_slw** rX,rY	Book E Book E *Page -955*
Shift Right Algebraic Word	**sraw** rA,rS,rB **sraw.** rA,rS,rB **se_sraw** rX,rY	Book E Book E *Page -956*

Table 145. Instructions listed by name (continued)

Instruction	Mnemonic	Reference
Shift Right Algebraic Word Immediate	**srawi** rA,rS,SH **srawi.** rA,rS,SH **se_srawi** rX,UI5	Book E Book E *Page -956*
Shift Right Word	**srw** rA,rS,rB **srw.** rA,rS,rB **se_srw** rX,rY	Book E Book E *Page -957*
Store Byte Indexed Store Byte with Update Indexed	**stbx** rS,rA,rB **stbux** rS,rA,rB	Book E
Store Halfword Byte-Reverse Indexed	**sthbrx** rS,rA,rB	Book E
Store Halfword Indexed Store Halfword with Update Indexed	**sthx** rS,rA,rB **sthux** rS,rA,rB	Book E
Store Word Byte-Reverse Indexed	**stwbrx** rS,rA,rB	Book E
Store Word Conditional Indexed	**stwcx.** rS,rA,rB	Book E
Store Word Indexed Store Word with Update Indexed	**stwx** rS,rA,rB **stwux** rS,rA,rB	Book E
Subtract From	**subf** rD,rA,rB **subf.** rD,rA,rB **subfo** rD,rA,rB **subfo.** rD,rA,rB	Book E
Subtract From Carrying	**subfc** rD,rA,rB **subfc.** rD,rA,rB **subfco** rD,rA,rB **subfco.** rD,rA,rB	Book E
TLB Invalidate Virtual Address Indexed	**tlbivax** rA,rB	Book E
TLB Read Entry	**tlbre**	Book E
TLB Search Indexed	**tlbsx** rA,rB	Book E
TLB Synchronize	**tlbsync**	Book E
TLB Write Entry	**tlbwe**	Book E
Trap Word	**tw** TO,rA,rB	Book E
Write MSR External Enable	**wrtee** rA	Book E
Write MSR External Enable Immediate	**wrteei E**	Book E
XOR	**xor** rA,rS,rB **xor.** rA,rS,rB	Book E

Table 145 lists instructions that can be executed in VLE mode by mnemonic.

Table 146. Instructions listed by mnemonic

Mnemonic	Instruction	Reference
add rD,rA,rB **add.** rD,rA,rB **addo** rD,rA,rB **addo.** rD,rA,rB	Add	Book E
addc rD,rA,rB **addc.** rD,rA,rB **addco** rD,rA,rB **addco.** rD,rA,rB	Add Carrying	Book E
adde rD,rA,rB **adde.** rD,rA,rB **addeo** rD,rA,rB **addeo.** rD,rA,rB	Add Extended	Book E
andc[.] rA,rS,rB	AND with Complement	Book E
and[.] rA,rS,rB	AND	Book E
cmp crD,L,rA,rB	Compare	Book E
cmpl crD,L,rA,rB	Compare Logical	Book E
cntlzw rA,rS **cntlzw.** rA,rS	Count Leading Zeros Word	Book E
dcba rA,rB	Data Cache Block Allocate	Book E
dcbf rA,rB	Data Cache Block Flush	Book E
dcbi rA,rB	Data Cache Block Invalidate	Book E
dcbst rA,rB	Data Cache Block Store	Book E
dcbt CT,rA,rB	Data Cache Block Touch	Book E
dcbtst CT,rA,rB	Data Cache Block Touch for Store	Book E
dcbz rA,rB	Data Cache Block set to Zero	Book E
divw rD,rA,rB **divw.** rD,rA,rB **divwo** rD,rA,rB **divwo.** rD,rA,rB	Divide Word	Book E
divwu rD,rA,rB **divwu.** rD,rA,rB **divwuo** rD,rA,rB **divwuo.** rD,rA,rB	Divide Word Unsigned	Book E
eqv rA,rS,rB **eqv.** rA,rS,rB	Equivalent	Book E
extsb rA,rS **extsb.** rA,rS	Extend Sign Byte	Book E
extsh rA,rS **extsh.** rA,rS	Extend Sign Halfword	Book E

Table 146. Instructions listed by mnemonic (continued)

Mnemonic	Instruction	Reference
e_add2is rD,SI	Add Immediate Shifted	Page -897
e_addi rD,rA,SCI8 **e_addi.** rD,rA,SCI8 **e_add16i** rD,rA,SI **e_add2i.** rD,SI	Add Immediate	Page -897
e_addic rD,rA,SCI8 **e_addic.** rD,rA,SCI8	Add Immediate Carrying	Page -900
e_and2is. rD,UI	AND Immediate Shifted	Page -901
e_andi[.] rA,rS,SCI8 **e_and2i.** rD,UI	AND Immediate	Page -901
e_bc BO32,BI32,BD15 **e_bcl** BO32,BI32,BD15	Branch Conditional Branch Conditional & Link	-904Page
e_b BD24 **e_bl** BD24	Branch Branch & Link	Page -903
e_cmph crD,rA,rB	Compare Halfword	Page -914
e_cmph16i rA,SI16	Compare Halfword Immediate	Page -914
e_cmphl crD,rA,rB	Compare Halfword Logical	Page -916
e_cmphl16i rA,UI16	Compare Halfword Logical Immediate	Page -916
e_cmpi crD,rA,SCI8 **e_cmp16i** rA,SI16	Compare Immediate	Page -912
e_cmpli crD,rA,SCI8 **e_cmpl16i** rA,UI16	Compare Logical Immediate	Page -918
e_crand crbD,crbA,crbB	Condition Register AND	Page -920
e_crandc crbD,crbA,crbB	Condition Register AND with Complement	Page -920
e_creqv crbD,crbA,crbB	Condition Register Equivalent	Page -920
e_crnand crbD,crbA,crbB	Condition Register NAND	Page -921
e_crnor crbD,crbA,crbB	Condition Register NOR	Page -922
e_cror crbD,crbA,crbB	Condition Register OR	Page -923
e_crorc crbD,crbA,crbB	Condition Register OR with Complement	Page -924
e_crxor crbD,crbA,crbB	Condition Register XOR	Page -925
e_lbz rD,D(rA) **e_lbzu** rD,D8(rA)	Load Byte and Zero Load Byte and Zero with Update	Page -930
e_lha rD,D(rA) **e_lhau** rD,D8(rA)	Load Halfword Algebraic Load Halfword Algebraic with Update	Page -931
e_lhz rD,D(rA) **e_lhzu** rD,D8(rA)	Load Halfword and Zero Load Halfword and Zero with Update	Page -932
e_li rD,LI20	Load Immediate	Page -933
e_lis rD,UI	Load Immediate Shifted	Page -933

Table 146. Instructions listed by mnemonic (continued)

Mnemonic	Instruction	Reference
e_lmw rD,D8(**r**A)	Load Multiple Word	*Page -935*
e_lwz rD,D(**r**A) **e_lwzu** rD,D8(**r**A)	Load Word and Zero Load Word and Zero with Update	*Page -936*
e_mcrf **cr**D,**cr**S	Move Condition Register Field	*Page -944*
e_mulli rD,**r**A,SCI8 **e_mull2i** rD,SI	Multiply Low Immediate	*Page -948*
e_or2is rD,UI	OR Immediate Shifted	*Page -948*
e_ori[.] rA,**r**S,SCI8 **e_or2i** rD,UI	OR Immediate	*Page -951*
e_rlw rA,**r**S,**r**B	Rotate Left Word	*Page -951*
e_rlwi rA,**r**S,SH	Rotate Left Word Immediate	*Page -952*
e_rlwimi rA,**r**S,SH,MB,ME	Rotate Left Word Immediate then Mask Insert	*Page -953*
e_rlwinm rA,**r**S,SH,MB,ME	Rotate Left Word Immediate then AND with Mask	*Page -955*
e_slwi rA,**r**S,SH	Shift Left Word Immediate	*Page -935*
e_srwi rA,**r**S,SH	Shift Right Word Immediate	Book E
e_stb rS,D(**r**A) **e_stbu** rS,D8(**r**A)	Store Byte Store Byte with Update	*Page -958*
e_sth rS,D(**r**A) **e_sthu** rS,D8(**r**A)	Store Halfword Store Halfword with Update	*Page -959*
e_stmw rS,D8(**r**A)	Store Multiple Word	*Page -960*
e_stw rS,D(**r**A) **e_stwu** rS,D8(**r**A)	Store Word Store Word with Update	*Page -961*
e_subfic rD,**r**A,SCI8 **e_subfic.** rD,**r**A,SCI8	Subtract From Immediate Carrying	*Page -964*
e_xori[.] rA,**r**S,SCI8	XOR Immediate	*Page -966*
icbi rA,**r**B	Instruction Cache Block Invalidate	Book E
icbt CT,**r**A,**r**B	Instruction Cache Block Touch	Book E
isel rD,**r**A,**r**B,**cr**b	Integer Select	EIS
lbzx rD,**r**A,**r**B **lbzux** rD,**r**A,**r**B	Load Byte and Zero Indexed Load Byte and Zero with Update Indexed	Book E
lhax rD,**r**A,**r**B **lhaux** rD,**r**A,**r**B	Load Halfword Algebraic Indexed Load Halfword Algebraic with Update Indexed	Book E
lhbrx rD,**r**A,**r**B	Load Halfword Byte-Reverse Indexed	Book E
lhzx rD,**r**A,**r**B **lhzux** rD,**r**A,**r**B	Load Halfword and Zero Indexed Load Halfword and Zero with Update Indexed	Book E
lwarx rD,**r**A,**r**B	Load Word And Reserve Indexed	Book E
lwbrx rD,**r**A,**r**B	Load Word Byte-Reverse Indexed	Book E

Table 146. Instructions listed by mnemonic (continued)

Mnemonic	Instruction	Reference
lwzx rD,rA,rB **lwzux** rD,rA,rB	Load Word and Zero Indexed Load Word and Zero with Update Indexed	Book E
mbar	Memory Barrier	Book E
mcrxr crD	Move to Condition Register from Integer Exception Register	Book E
mfcr rD	Move From condition register	Book E
mfdcr rD,DCRN	Move From Device Control Register	Book E
mfmsr rD	Move From Machine State Register	Book E
mfspr rD,SPRN	Move From Special Purpose Register	Book E
msync	Memory Synchronize	Book E
mtcrf FXM,rS	Move to Condition Register Fields	Book E
mtdcr DCRN,rS	Move To Device Control Register	Book E
mtmsr rS	Move To Machine State Register	Book E
mtspr SPRN,rS	Move To Special Purpose Register	Book E
mulhw rD,rA,rB **mulhw.** rD,rA,rB	Multiply High Word	Book E
mulhwu rD,rA,rB **mulhwu.** rD,rA,rB	Multiply High Word Unsigned	Book E
mullw rD,rA,rB **mullw.** rD,rA,rB **mullwo** rD,rA,rB **mullwo.** rD,rA,rB	Multiply Low Word	Book E
nand rA,rS,rB **nand.** rA,rS,rB	NAND	Book E
neg rD,rA **neg.** rD,rA **nego** rD,rA **nego.** rD,rA	Negate	Book E
nor rA,rS,rB **nor.** rA,rS,rB	NOR	Book E
or rA,rS,rB **or.** rA,rS,rB	OR	Book E
orc rA,rS,rB **orc.** rA,rS,rB	OR with Complement	Book E
se_add rX,rY	Add	*Page -897*
se_addi rX,OIMM	Add Immediate	*Page -897*
se_andc rX,rY	AND with Complement	*Page -901*
se_andi rX,UI5	AND Immediate	*Page -901*

Table 146. Instructions listed by mnemonic (continued)

Mnemonic	Instruction	Reference
se_and[.] rX,rY	AND	Page -901
se_bc BO16,BI16,BD8	Branch Conditional	Page -904
se_bclri rX,UI5	Bit Clear	Page -905
se_bctr **se_bctrl**	Branch to Count Register Branch to Count Register & Link	Page -905
se_bgeni rX,UI5	Bit Generate	Page -906
se_blr **se_blrl**	Branch to Link Register Branch to Link Register & Link	Page -907
se_bmski rX,UI5	Bit Mask Generate	Page -908
se_bseti rX,UI5	Bit Set	Page -909
se_b BD8 **se_bl** BD8	Branch Branch & Link	Page -910
se_btsti rX,UI5	Bit Test Immediate	Page -903
se_cmp rX,rY	Compare	Page -912
se_cmph rX,rY	Compare Halfword	Page -914
se_cmphl rX,rY	Compare Halfword Logical	Page -916
se_cmpi rX,UI5	Compare Immediate	Page -912
se_cmpl rX,rY	Compare Logical	Page -918
se_cmpli rX,UI5	Compare Logical Immediate	Page -918
se_extsb rX	Extend Sign Byte	Page -926
se_extsh rX	Extend Sign Halfword	Page -926
se_extzb rX	Extend with Zeros Byte	Page -927
se_extzh rX	Extend with Zeros Halfword	Page -927
se_isync	Instruction Synchronize	Page -929
se_lbz rZ,SD4**(**rX**)**	Load Byte and Zero (16-bit form)	Page -930
se_lhz rZ,SD4**(**rX**)**	Load Halfword and Zero (16-bit form)	Page -932
se_li rX,UI7	Load Immediate	Page -933
se_lwz rZ,SD4**(**rX**)**	Load Word and Zero (16-bit form)	Page -935
se_mfar rX,**ar**Y	Move from Alternate Register	Page -937
se_mfctr rX	Move From Count Register	Page -938
se_mflr rX	Move From Link Register	Page -939
se_mr rX,rY	Move Register	Page -940
se_mtar **ar**X,rY	Move to Alternate Register	Page -941
se_mtctr rX	Move To Count Register	Page -942
se_mtlr rX	Move To Link Register	Page -943

Table 146. Instructions listed by mnemonic (continued)

Mnemonic	Instruction	Reference
se_mullw rX,rY	Multiply Low Word	*Page -945*
se_neg rX	Negate	*Page -946*
se_not rX	NOT	*Page -947*
se_or rX,rY	OR	*Page -948*
se_slw rX,rY	Shift Left Word	*Page -955*
se_slwi rX,UI5	Shift Left Word Immediate	*Page -955*
se_sraw rX,rY	Shift Right Algebraic Word	*Page -956*
se_srawi rX,UI5	Shift Right Algebraic Word Immediate	*Page -956*
se_srw rX,rY	Shift Right Word	*Page -957*
se_srwi rX,UI5	Shift Right Word Immediate	*Page -957*
se_stb rZ,SD4**(**rX**)**	Store Byte (16-bit form)	*Page -958*
se_sth rZ,SD4**(**rX**)**	Store Halfword (16-bit form)	*Page -959*
se_stw rZ,SD4**(**rX**)**	Store Word (16-bit form)	*Page -961*
se_sub rX,rY	Subtract	*Page -962*
se_subf rX,rY	Subtract From	*Page -963*
se_subi rX,OIMM **se_subi.** rX,OIMM	Subtract Immediate	*Page -965*
slw rA,rS,rB **slw.** rA,rS,rB	Shift Left Word	Book E
sraw rA,rS,rB **sraw.** rA,rS,rB	Shift Right Algebraic Word	Book E
srawi rA,rS,SH **srawi.** rA,rS,SH	Shift Right Algebraic Word Immediate	Book E
srw rA,rS,rB **srw.** rA,rS,rB	Shift Right Word	Book E
stbx rS,rA,rB **stbux** rS,rA,rB	Store Byte Indexed Store Byte with Update Indexed	Book E
sthbrx rS,rA,rB	Store Halfword Byte-Reverse Indexed	Book E
sthx rS,rA,rB **sthux** rS,rA,rB	Store Halfword Indexed Store Halfword with Update Indexed	Book E
stwbrx rS,rA,rB	Store Word Byte-Reverse Indexed	Book E
stwcx. rS,rA,rB	Store Word Conditional Indexed	Book E
stwx rS,rA,rB **stwux** rS,rA,rB	Store Word Indexed Store Word with Update Indexed	Book E

Table 146. Instructions listed by mnemonic (continued)

Mnemonic	Instruction	Reference
subf rD,rA,rB **subf.** rD,rA,rB **subfo** rD,rA,rB **subfo.** rD,rA,rB	Subtract From	Book E
subfc rD,rA,rB **subfc.** rD,rA,rB **subfco** rD,rA,rB **subfco.** rD,rA,rB	Subtract From Carrying	Book E
tlbivax rA,rB	TLB Invalidate Virtual Address Indexed	Book E
tlbre	TLB Read Entry	Book E
tlbsx rA,rB	TLB Search Indexed	Book E
tlbsync	TLB Synchronize	Book E
tlbwe	TLB Write Entry	Book E
tw TO,rA,rB	Trap Word	Book E
wrtee rA	Write MSR External Enable	Book E
wrteei E	Write MSR External Enable Immediate	Book E
xor rA,rS,rB **xor.** rA,rS,rB	XOR	Book E

3.7 Instruction listing

Table 147 lists instructions defined in Book E, in the PowerPC architecture, and by the EIS. A check mark (√) or text in a column indicates that the instruction is defined or implemented. The EIS-specific instructions are indicated by the name of the APU or architectural extension that defines the instruction.

Table 147. List of instructions

Mnemonic	Book E	Classic	EIS	Mnemonic	Book E	Classic	EIS
addc[**o**][.]	√	√		**e_cmpli**			VLE
adde[**o**][.]	√	√		**e_crand**			VLE
addi	√	√		**e_crandc**			VLE
addic[.]	√	√		**e_creqv**			VLE
addis	√	√		**e_crnand**			VLE
addme[**o**][.]	√	√		**e_crnor**			VLE
addze[**o**][.]	√	√		**e_cror**			VLE
add[**o**].]	√	√		**e_crorc**			VLE
andc[.]	√	√		**e_crxor**			VLE
andi.	√	√		**e_lbz**			VLE

Table 147. List of instructions (continued)

Mnemonic	Book E	Classic	EIS	Mnemonic	Book E	Classic	EIS
andis.	√	√		e_lbzu			VLE
and[.]	√	√		e_lha			VLE
b	√	√		e_lhau			VLE
ba	√	√		e_lhz			VLE
bc	√	√		e_lhzu			VLE
bca	√	√		e_li			VLE
bcctr	√	√		e_lis			VLE
bcctrl	√	√		e_lmw			VLE
bcl	√	√		e_lwz			VLE
bcla	√	√		e_lwzu			VLE
bclr	√	√		e_mcrf			VLE
bclrl	√	√		e_mull2i			VLE
bl	√	√		e_mulli			VLE
bla	√	√		e_or2i			VLE
brinc			SPE APU	e_or2is			VLE
cmp	√	√		e_ori[.]			VLE
cmpi	√	√		e_rlw			VLE
cmpl	√	√		e_rlwi			VLE
cmpli	√	√		e_rlwimi			VLE
cntlzw[.]	√	√		e_rlwinm			VLE
crand	√	√		e_slwi			VLE
crandc	√	√		e_srwi			VLE
creqv	√	√		e_stb			VLE
crnand	√	√		e_stbu			VLE
crnor	√	√		e_sth			VLE
cror	√	√		e_sthu			VLE
crorc	√	√		e_stmw			VLE
crxor	√	√		e_stw			VLE
dcba	√	√		e_stwu			VLE
dcbf	√	√		e_subfic			VLE
dcbi	√	√		e_subfic.			VLE
dcblc			Cache locking	e_xori[.]			VLE
dcbst	√	√		fabs[.]	√	√	

Table 147. List of instructions (continued)

Mnemonic	Book E	Classic	EIS	Mnemonic	Book E	Classic	EIS
dcbt	√	√		**fadds**[.]	√	√	
dcbtls			Cache locking	**fadd**[.]	√	√	
dcbtst	√	√		**fcfid**[.]	√	√	
dcbtstls			Cache locking	**fcmpo**	√	√	
dcbz	√	√		**fcmpu**	√	√	
divwu[o][.]	√	√		**fctidz**[.]	√	√	
divw[o][.]	√	√		**fctid**[.]	√	√	
eciwx		√		**fctiwz**[.]	√	√	
ecowx		√		**fctiw**[.]	√	√	
efsabs			Scalar SPFP	**fdivs**[.]	√	√	
efsadd			Scalar SPFP	**fdiv**[.]	√	√	
efscfsf			Scalar SPFP	**fmadds**[.]	√	√	
efscfsi			Scalar SPFP	**fmadd**[.]	√	√	
efscfuf			Scalar SPFP	**fmr**[.]	√	√	
efscfui			Scalar SPFP	**fmsubs**[.]	√	√	
efscmpeq			Scalar SPFP	**fmsub**[.]	√	√	
efscmpgt			Scalar SPFP	**fmuls**[.]	√	√	
efscmplt			Scalar SPFP	**fmul**[.]	√	√	
efsctsf			Scalar SPFP	**fnabs**[.]	√	√	
efsctsi			Scalar SPFP	**fneg**[.]	√	√	
efsctsiz			Scalar SPFP	**fnmadds**[.]	√	√	
efsctuf			Scalar SPFP	**fnmadd**[.]	√	√	
efsctui			Scalar SPFP	**fnmsubs**[.]	√	√	
efsctuiz			Scalar SPFP	**fnmsub**[.]	√	√	

Table 147. List of instructions (continued)

Mnemonic	Book E	Classic	EIS	Mnemonic	Book E	Classic	EIS
efsdiv			Scalar SPFP	**fres**[.]	√	√	
efsmul			Scalar SPFP	**frsp**[.]	√	√	
efsnabs			Scalar SPFP	**frsqrte**[.]	√	√	
efsneg			Scalar SPFP	**fsel**[.]	√	√	
efssub			Scalar SPFP	**fsqrts**[.]	√	√	
efststeq			Scalar SPFP	**fsqrt**[.]	√	√	
efststgt			Scalar SPFP	**fsubs**[.]	√	√	
efststlt			Scalar SPFP	**fsub**[.]	√	√	
eieio	Now **mbar**	√		**icbi**	√	√	
eqv[.]	√	√		**icblc**			Cache locking
evabs			SPE APU	**icbt**	√		
evaddiw			SPE APU	**icbtls**			Cache locking
evaddsmiaaw			SPE APU	**isel**			Integer select
evaddssiaaw			SPE APU	**isync**	√	√	
evaddumiaaw			SPE APU	**lbz**	√	√	
evaddusiaaw			SPE APU	**lbzu**	√	√	
evaddw			SPE APU	**lbzux**	√	√	
evand			SPE APU	**lbzx**	√	√	
evandc			SPE APU	**ld**		√	
evcmpeq			SPE APU	**ldarx**		√	
evcmpgts			SPE APU	**ldu**		√	

Table 147. List of instructions (continued)

Mnemonic	Book E	Classic	EIS	Mnemonic	Book E	Classic	EIS
evcmpgtu			SPE APU	ldux		√	
evcmplts			SPE APU	ldx		√	
evcmpltu			SPE APU	lfd	√	√	
evcntlsw			SPE APU	lfdu	√	√	
evcntlzw			SPE APU	lfdux	√	√	
evdivws			SPE APU	lfdx	√	√	
evdivwu			SPE APU	lfs	√	√	
eveqv			SPE APU	lfsu	√	√	
evextsb			SPE APU	lfsux	√	√	
evextsh			SPE APU	lfsx	√	√	
evfsabs			Vector SPFP	lha	√	√	
evfsadd			Vector SPFP	lhau	√	√	
evfscfsf			Vector SPFP	lhaux	√	√	
evfscfsi			Vector SPFP	lhax	√	√	
evfscfuf			Vector SPFP	lhbrx	√	√	
evfscfui			Vector SPFP	lhz	√	√	
evfscmpeq			Vector SPFP	lhzu	√	√	
evfscmpgt			Vector SPFP	lhzux	√	√	
evfscmplt			Vector SPFP	lhzx	√	√	
evfsctsf			Vector SPFP	lmw	√	√	
evfsctsi			Vector SPFP	lswi	√	√	

Table 147. List of instructions (continued)

Mnemonic	Book E	Classic	EIS	Mnemonic	Book E	Classic	EIS
evfsctsiz			Vector SPFP	lswx	√	√	
evfsctuf			Vector SPFP	lwa		√	
evfsctui			Vector SPFP	lwarx	√	√	
evfsctuiz			Vector SPFP	lwaux		√	
evfsdiv			Vector SPFP	lwax		√	
evfsmul			Vector SPFP	lwbrx	√	√	
evfsnabs			Vector SPFP	lwz	√	√	
evfsneg			Vector SPFP	lwzu	√	√	
evfssub			Vector SPFP	lwzux	√	√	
evfststeq			Vector SPFP	lwzx	√	√	
evfststgt			Vector SPFP	mbar	√		
evfststlt			Vector SPFP	mcrf	√	√	
evldd			SPE APU	mcrfs	√	√	
evlddx			SPE APU	mcrxr	√	√	
evldh			SPE APU	mfapidi	√		
evldhx			SPE APU	mfcr	√	√	
evldw			SPE APU	mfdcr	√		
evldwx			SPE APU	mffs[.]	√	√	
evlhhesplat			SPE APU	mfmsr	√	√	
evlhhesplatx			SPE APU	mfpmr			Performance monitor

Table 147. List of instructions (continued)

Mnemonic	Book E	Classic	EIS	Mnemonic	Book E	Classic	EIS
evlhhossplat			SPE APU	**mfspr**	√	√	
evlhhossplatx			SPE APU	**mfsr**		√	
evlhhousplat			SPE APU	**mfsrin**		√	
evlhhousplatx			SPE APU	**mftb**		√	
evlwhe			SPE APU	**msync**	√		
evlwhex			SPE APU	**mtcrf**	√	√	
evlwhos			SPE APU	**mtdcr**	√		
evlwhosx			SPE APU	**mtfsb0**[.]	√	√	
evlwhou			SPE APU	**mtfsb1**[.]	√	√	
evlwhoux			SPE APU	**mtfsfi**[.]	√	√	
evlwhsplat			SPE APU	**mtfsf**[.]	√	√	
evlwhsplatx			SPE APU	**mtmsr**	√	√	
evlwwsplat			SPE APU	**mtmsrd**		64-bit only	
evlwwsplatx			SPE APU	**mtpmr**			Performance monitor
evmergehi			SPE APU	**mtspr**	√	√	
evmergehilo			SPE APU	**mtsr**		√	
evmergelo			SPE APU	**mtsrd**		√	
evmergelohi			SPE APU	**mtsrdin**		√	
evmhegsmfaa			SPE APU	**mtsrin**		√	
evmhegsmfan			SPE APU	**mulhd.**		√	

Table 147. List of instructions (continued)

Mnemonic	Book E	Classic	EIS	Mnemonic	Book E	Classic	EIS
evmhegsmiaa			SPE APU	mulhdu.		√	
evmhegsmian			SPE APU	mulhwu[.]	√	√	
evmhegumiaa			SPE APU	mulhw[.]	√	√	
evmhegumian			SPE APU	mulld.		√	
evmhesmf			SPE APU	mulldo.		√	
evmhesmfa			SPE APU	mulli	√	√	
evmhesmfaaw			SPE APU	mullw[o][.]	√	√	
evmhesmfanw			SPE APU	nand[.]	√	√	
evmhesmi			SPE APU	neg[o][.]	√	√	
evmhesmia			SPE APU	nor[.]	√	√	
evmhesmiaaw			SPE APU	orc[.]	√	√	
evmhesmianw			SPE APU	ori	√	√	
evmhessf			SPE APU	oris	√	√	
evmhessfa			SPE APU	or[.]	√	√	
evmhessfaaw			SPE APU	rfci	√		
evmhessfanw			SPE APU	rfi	√	√	
evmhessiaaw			SPE APU	rfid		√	
evmhessianw			SPE APU	rfmci			Machine check
evmheumi			SPE APU	rldcl.		√	
evmheumia			SPE APU	rldcr.		√	
evmheumiaaw			SPE APU	rldic.		√	

Table 147. List of instructions (continued)

Mnemonic	Book E	Classic	EIS	Mnemonic	Book E	Classic	EIS
evmheumianw			SPE APU	rldicl.		√	
evmheusiaaw			SPE APU	rldicr.		√	
evmheusianw			SPE APU	rldimi.		√	
evmhogsmfaa			SPE APU	rlwimi[.]	√	√	
evmhogsmfan			SPE APU	rlwinm[.]	√	√	
evmhogsmiaa			SPE APU	rlwnm[.]	√	√	
evmhogsmian			SPE APU	sc	√	√	
evmhogumiaa			SPE APU	se_add			VLE
evmhogumian			SPE APU	se_addi			VLE
evmhosmf			SPE APU	se_andc			VLE
evmhosmfa			SPE APU	se_andi			VLE
evmhosmfaaw			SPE APU	se_and[.]			VLE
evmhosmfanw			SPE APU	se_b			VLE
evmhosmi			SPE APU	se_bc			VLE
evmhosmia			SPE APU	se_bclri			VLE
evmhosmiaaw			SPE APU	se_bctr			VLE
evmhosmianw			SPE APU	se_bctrl			VLE
evmhossf			SPE APU	se_bgeni			VLE
evmhossfa			SPE APU	se_bl			VLE
evmhossfaaw			SPE APU	se_blr			VLE
evmhossfanw			SPE APU	se_blrl			VLE

Table 147. List of instructions (continued)

Mnemonic	Book E	Classic	EIS	Mnemonic	Book E	Classic	EIS
evmhossiaaw			SPE APU	se_bmski			VLE
evmhossianw			SPE APU	se_bseti			VLE
evmhoumi			SPE APU	se_btsti			VLE
evmhoumia			SPE APU	se_cmp			VLE
evmhoumiaaw			SPE APU	se_cmph			VLE
evmhoumianw			SPE APU	se_cmphl			VLE
evmhousiaaw			SPE APU	se_cmpi			VLE
evmhousianw			SPE APU	se_cmpl			VLE
evmra			SPE APU	se_cmpli			VLE
evmwhsmf			SPE APU	se_extsb			VLE
evmwhsmfa			SPE APU	se_extsh			VLE
evmwhsmi			SPE APU	se_extzb			VLE
evmwhsmia			SPE APU	se_extzh			VLE
evmwhssf			SPE APU	se_isync			VLE
evmwhssfa			SPE APU	se_lbz			VLE
evmwhumi			SPE APU	se_lhz			VLE
evmwhumia			SPE APU	se_li			VLE
evmwlsmiaaw			SPE APU	se_lwz			VLE
evmwlsmianw			SPE APU	se_mfar			VLE
evmwlssiaaw			SPE APU	se_mfctr			VLE
evmwlssianw			SPE APU	se_mflr			VLE

Table 147. List of instructions (continued)

Mnemonic	Book E	Classic	EIS	Mnemonic	Book E	Classic	EIS
evmwlumi			SPE APU	se_mr			VLE
evmwlumia			SPE APU	se_mtar			VLE
evmwlumiaaw			SPE APU	se_mtctr			VLE
evmwlumianw			SPE APU	se_mtlr			VLE
evmwlusiaaw			SPE APU	se_mullw			VLE
evmwlusianw			SPE APU	se_neg			VLE
evmwsmf			SPE APU	se_not			VLE
evmwsmfa			SPE APU	se_or			VLE
evmwsmfaa			SPE APU	se_slw			VLE
evmwsmfan			SPE APU	se_slwi			VLE
evmwsmi			SPE APU	se_sraw			VLE
evmwsmia			SPE APU	se_srawi			VLE
evmwsmiaa			SPE APU	se_srw			VLE
evmwsmian			SPE APU	se_srwi			VLE
evmwssf			SPE APU	se_stb			VLE
evmwssfa			SPE APU	se_sth			VLE
evmwssfaa			SPE APU	se_stw			VLE
evmwssfan			SPE APU	se_sub			VLE
evmwumi			SPE APU	se_subf			VLE
evmwumia			SPE APU	se_subi			VLE
evmwumiaa			SPE APU	se_subi.			VLE

Table 147. List of instructions (continued)

Mnemonic	Book E	Classic	EIS	Mnemonic	Book E	Classic	EIS
evmwumian			SPE APU	slbia		√	
evnand			SPE APU	slbie		√	
evneg			SPE APU	sldi		√	
evnor			SPE APU	slw[.]	√	√	
evor			SPE APU	srad.		√	
evorc			SPE APU	sradi.		√	
evrlw			SPE APU	srawi[.]	√	√	
evrlwi			SPE APU	sraw[.]	√	√	
evrndw			SPE APU	srd.		√	
evsel			SPE APU	srw[.]	√	√	
evslw			SPE APU	stb	√	√	
evslwi			SPE APU	stbu	√	√	
evsplatfi			SPE APU	stbux	√	√	
evsplati			SPE APU	stbx	√	√	
evsrwis			SPE APU	std		√	
evsrwiu			SPE APU	stdcx.		√	
evsrws			SPE APU	stdu		√	
evsrwu			SPE APU	stdux		√	
evstdd			SPE APU	stdx		√	
evstddx			SPE APU	stfd	√	√	
evstdh			SPE APU	stfdu	√	√	

Table 147. List of instructions (continued)

Mnemonic	Book E	Classic	EIS	Mnemonic	Book E	Classic	EIS
evstdhx			SPE APU	**stfdux**	√	√	
evstdw			SPE APU	**stfdx**	√	√	
evstdwx			SPE APU	**stfiwx**	√	√	
evstwhe			SPE APU	**stfs**	√	√	
evstwhex			SPE APU	**stfsu**	√	√	
evstwho			SPE APU	**stfsux**	√	√	
evstwhox			SPE APU	**stfsx**	√	√	
evstwwex			SPE APU	**sth**	√	√	
evstwwex			SPE APU	**sthbrx**	√	√	
evstwwo			SPE APU	**sthu**	√	√	
evstwwox			SPE APU	**sthux**	√	√	
evsubfsmiaaw			SPE APU	**sthx**	√	√	
evsubfssiaaw			SPE APU	**stmw**	√	√	
evsubfumiaaw			SPE APU	**stswi**	√	√	
evsubfusiaaw			SPE APU	**stswx**	√	√	
evsubfw			SPE APU	**stw**	√	√	
evsubifw			SPE APU	**stwbrx**	√	√	
evxor			SPE APU	**stwcx.**	√	√	
extsb[.]	√	√		**stwu**	√	√	
extsh[.]	√	√		**stwux**	√	√	
extsw.		64-bit only		**stwx**	√	√	
e_add16i			VLE	**subfc[o][.]**	√	√	
e_add2i.			VLE	**subfe[o][.]**	√	√	

Table 147. List of instructions (continued)

Mnemonic	Book E	Classic	EIS	Mnemonic	Book E	Classic	EIS
e_add2is			VLE	subfic	√	√	
e_addi			VLE	subfme[o][.]	√	√	
e_addi.			VLE	subfze[o][.]	√	√	
e_addic			VLE	subf[o][.]	√	√	
e_addic.			VLE	sync	Now msync	√	
e_and2i.			VLE	tlbia		√	
e_and2is.			VLE	tlbie		√	
e_andi[.]			VLE	tlbivax	√		
e_b			VLE	tlbre	√		
e_bc			VLE	tlbsx	√		
e_bcl			VLE	tlbsync	√	√	
e_bl			VLE	tlbwe	√		
e_cmp16i			VLE	tw	√	√	
e_cmph			VLE	twi	√	√	
e_cmph16i			VLE	wrtee	√		
e_cmphl			VLE	wrteei	√		
e_cmphl16i			VLE	xori[.]	√	√	
e_cmpi			VLE	xor[.]	√	√	
e_cmpl16i			VLE				

4 Interrupts and exceptions

This chapter provides a general description of the Book E exception and interrupt models as they are implemented on ST processors. It identifies and describes the portions of the interrupt model that are defined by the Book E architecture and by the Book E implementation standards (EIS).

Note: : Terminology

The Book E architecture has defined additional resources for interrupt handling. As a result, the terms 'interrupt' and 'exception' differ somewhat from their use in previous ST documentation, such as the Programming Environments Manual. Use of these terms is now as follows:

- An interupt is the action in which the processor saves its context (typically the machine state register (MSR) and next instruction address) and begins execution at a predetermined interrupt handler address with a modified MSR.

- An exception is the event that, if enabled, causes the processor to take an interrupt. Book E describes exceptions as being generated by signals from internal and external peripherals, instructions, the internal timer facility, debug events, or error conditions.

4.1 Overview

Book E defines are two categories of interrupts, noncritical and critical, for which separate resources are provided to save state when the interrupt is taken and to restore state when the interrupt handler returns control to the interrupted program.

Using the model provided by the Book E architecture, the EIS defines additional interrupt types which may be implemented on ST Book E devices. These are described in *Table 148*.

Table 148. Interrupt types

Category	Description	Programming resources
Book E defined		
Noncritical interrupts	First-level interrupts that let the processor change program flow to handle conditions generated by external signals, errors, or unusual conditions arising from program execution or from programmable timer-related events. These interrupts are largely identical to those defined by the OEA.	SRR0/SRR1 SPRs and **rfi** instruction. Asynchronous noncritical interrupts can be masked by the external interrupt enable bit, MSR[EE].
Critical interrupts	Book E–defined. Critical input, watchdog timer, and debug interrupts. these interrupts can be taken during a noncritical interrupt or during regular program flow. Book E defines the critical input, watchdog timer, debug, and machine check interrupts as critical interrupts. The EIS defines additional resources for machine check and debug interrupts.	Critical save and restore SPRs (CSRR0/CSRR1) and the **rfci** instruction. Critical input and watchdog timer critical interrupts can be masked by the critical enable bit, MSR[CE]. Debug events can be masked by the debug enable bit MSR[DE].

Table 148. Interrupt types

Category	Description	Programming resources
EIS defined (consult implementation documentation to determine whether these interrupts are implemented)		
Machine check interrupt	The EIS-defined machine check APU provides a separate set of resources for the machine check interrupt, which is similar to the Book E–defined critical interrupt type.	Machine check save and restore SPRs (MCSRR0/MCSRR1) and the **rfmci** instruction. Can be masked by the machine check enable bit, MSR[ME].
Debug interrupt	The EIS-defined debug APU provides a separate set of resources for the debug interrupt, which is similar to the Book E–defined critical interrupt type.	Debug save and restore SPRs (DSRR0/DSRR1) and the **rfdi** instruction. Can be masked by the machine check enable bit, MSR[DE]. The debug APU extends the Book E debug register model for more detailed control of debug resources.

All interrupts except EIS-defined interrupts are ordered within the two categories of noncritical and critical, such that only one interrupt of each category is reported, and when an interrupt is processed (taken), no program state is lost. Because save/restore register pairs are serially reusable, care must be taken to preserve program state that may be lost when an unordered interrupt is taken. (See *Chapter 4.10*.")

All interrupts except the machine check interrupt are context synchronizing as defined in *Context synchronization on page 144*." 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 1 and 2 of *Context synchronization on page 144*," but satisfies items 3 and 4.

4.2 EIs interrupt definitions

This section gives an overview of additions and modifications to the Book E interrupt model defined by the EIS. Specific details are also provided throughout this chapter. Except for the following, the core complex reports exceptions as specified in Book E:

- The machine check exception differs as follows:
  - It is not processed as a critical interrupt, but uses MCSRR0 and MCSRR1 for saving the return address and the MSR in case the machine check is recoverable.
  - Return From Machine Check Interrupt instruction (**rfmci**) is implemented to support the return to the address saved in MCSRR0.
  - A machine check syndrome register, MCSR, logs the cause of the machine check (instead of ESR).

  The core complex reports the machine check exception as described in *Chapter 4.7.2*."
- The following interrupts are defined for use with the embedded floating-point and signal-processing (SPE) APUs:
  - SPE/embedded floating-point unavailable interrupt. IVOR32 (SPR 528) contains the vector offset.
    See *SPE/embedded floating-point APU unavailable interrupt on page 272*."
  - Embedded floating-point data interrupt. IVOR33 (SPR 529) contains the vector offset. See *Embedded floating-point data interrupt on page 272*."
  - Embedded floating-point round interrupt. IVOR34 (SPR 530) contains the vector offset. See *Embedded floating-point round interrupt on page 273*."

The following additional bits are defined to support SPE and SPFP exceptions:

- MSR[38] is defined as the vector available bit (SPE). If this bit is clear and software attempts to execute any of the SPE instructions, the SPE unavailable interrupt is taken. If this bit is set, software can execute any SPE instructions.

Note: SPFP instructions require MSR[SPE] to be set. An attempt to execute an SPFP instruction when MSR[SPE] is 0 causes an SPE APU unavailable interrupt. Embedded vector and scalar floating-point APU instructions on page 196," lists affected instructions.

- ESR[SPE], the SPE exception bit, is set when the processor reports an exception related to the execution of SPFP or SPE instructions.

- The debug exception implementation does not support IAC3, IAC4, DAC3, and DAC4 comparisons.
- The core complex supports instruction address compare (IAC1 and IAC2) and data address compare (DAC1 and DAC2) for effective addresses only. Real-address support is not provided.
- Some implementations do not support the Book E-defined floating-point unavailable and auxiliary processor unavailable interrupts.
- Data value compare (DVC) debug exceptions are not supported.
- The interrupt priorities differ from those specified in Book E as described in *Chapter 4.11*."
- Alignment exceptions. Vector operations can cause alignment exceptions as described in *Chapter 4.7.6*."
- Book E and the machine check APU define sources of externally generated interrupts.

4.2.1 Recoverability from interrupts

All interrupts except some machine check interrupts are recoverable. The state of the core complex (return address and MSR contents) is saved when a machine check interrupt is taken. The conditions that cause a machine check may or may not prohibit recovery.

4.3 Interrupt registers

Table 149 summarizes registers used for interrupt handling. These registers are described in detail in *Chapter 2*."

Table 149. Interrupt registers defined by the PowerPC architecture

Register	Description
Book E Interrupt Registers	
Save/restore register 0 (SRR0)	On a noncritical 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 noncritical interrupt or the address of the instruction to return to after a noncritical interrupt is serviced.
Save/restore register 1 (SRR1)	When a noncritical interrupt is taken, MSR contents are placed into SRR1. When **rfi** is executed, SRR1 contents are placed into the MSR. SRR1 bits that correspond to reserved MSR bits are also reserved. Note that an MSR bit that is reserved may be altered by **rfi**.
Critical save/restore register 0 (CSRR0)	When a critical interrupt is taken, 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.
Critical save/restore register 1 (CSRR1)	When a critical interrupt is taken, MSR contents are placed into CSRR1. When **rfci** is executed, CSRR1 contents are placed into the MSR. CSRR1 bits that correspond to reserved MSR bits are also reserved. Note that an MSR bit that is reserved may be altered by **rfci**.
Data exception address register (DEAR)	DEAR contains the address referenced by a load, store, or cache management instruction that caused an alignment, data TLB miss, or data storage interrupt.
Interrupt vector prefix register (IVPR)	IVPR[32–47] provides the high-order 48 bits of the address of the interrupt handling routine for each interrupt type. The 16-bit vector offsets are concatenated to the right of IVPR to form the address of the interrupt handling routine. IVPR[48–63] are reserved.

Table 149. Interrupt registers defined by the PowerPC architecture (continued)

Register	Description
Exception syndrome register (ESR)	Provides a syndrome to differentiate between exceptions that can generate the same interrupt type. When one of these types of interrupts is generated, bits corresponding to the specific exception that generated the interrupt are set and all other ESR bits are cleared. Other interrupt types do not affect the ESR. ESR does not need to be cleared by software. *Exception syndrome register (ESR) on page 84*," shows ESR bit definitions. An implementation may define additional ESR bits to identify implementation-specific or architected interrupt types; the EIS defines ESR[ILK] and ESR[SPE]. *Note: System software may need to identify the type of instruction that caused the interrupt and examine the TLB entry and ESR to fully identify the exception or exceptions. For example, because both protection violation and byte-ordering exception conditions may be present, and either causes a data storage interrupt, system software would have to look beyond ESR[BO], such as the state of MSR[PR] in SRR1 and the TLB entry page protection bits, to determine if a protection violation also occurred.* The EIS defines ESR[56] as the SPE exception bit (SPE). It is set when the processor reports an exception related to the execution of an SPFP or SPE instruction. Note that the EIS definition of the machine check interrupt uses the machine check syndrome register (MCSR) rather than the ESR.

Table 149. Interrupt registers defined by the PowerPC architecture (continued)

<table>
<tr><th>Register</th><th>Description</th></tr>
<tr><td>Interrupt vector offset registers (IVORs)</td><td>Holds the quad-word index from the base address provided by the IVPR for each interrupt type. IVOR0–IVOR15 are provided for defined interrupt types. SPR numbers corresponding to IVOR16–IVOR31 are reserved. IVOR[32–47,60–63] are reserved. SPR numbers for IVOR32–IVOR63 are allocated for implementation-dependent use. (IVOR32–IVOR34 (SPR 528–530) are used by interrupts defined by the EIS.) IVOR assignments are shown below.

Book E–defined interrupts
IVOR Number — Interrupt Type
IVOR0 — Critical input
IVOR1 — Machine check
IVOR2 — Data storage
IVOR3 — Instruction storage
IVOR4 — External input
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
IIVOR16–IVOR31 — Reserved

EIS-defined interrupts (IVOR32–IVOR63)
IVOR Number — Interrupt Type
IVOR32 — SPE APU unavailable
IVOR33 — Embedded floating-point data
IVOR34 — Embedded floating-point round
IVOR35 — Performance monitor
IVOR36 — Processor doorbell
IVOR37 — Processor doorbell critical</td></tr>
<tr><td>Machine state register (MSR)</td><td>MSR[38] is defined as the vector available bit (SPE). It functions as follows:
0: If software attempts to execute an instruction that tries to access the upper word of a 64-bit GPR, an SPE APU unavailable interrupt is taken.
1: Software can execute any embedded floating-point or SPE instructions.</td></tr>
<tr><td colspan="2">EIS-Specific Interrupt Registers</td></tr>
<tr><td>Machine check save/restore register 0 (MCSRR0)</td><td>When a machine check interrupt is taken, 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 the instruction that caused the machine check interrupt, or the address of the instruction to return to after a machine check interrupt is serviced.</td></tr>
<tr><td>Machine check save/restore register 1 (MCSRR1)</td><td>When a machine check interrupt is taken, MSR contents are placed into MCSRR1. When rfmci is executed, MCSRR1 contents are restored to MSR. MCSRR1 bits that correspond to reserved MSR bits are also reserved. Note that an MSR bit that is reserved may be altered by rfmci.</td></tr>
</table>

Table 149. Interrupt registers defined by the PowerPC architecture (continued)

Register	Description
Machine check syndrome register (MCSR)	When a machine check interrupt is taken, MCSR is updated to differentiate among machine check conditions. MCSR also indicates whether a machine check condition is recoverable. ABIST status is logged in MCSR[48–54]. These read-only bits do not initiate machine check (or any other interrupt). An ABIST bit being set indicates an error being detected in the corresponding module. Processors that do not implement the machine check APU use the Book E–defined ESR for this purpose. *Machine check syndrome register (MCSR) on page 88*," shows MCSR bit definitions.
Machine check address register (MCAR)	When a machine check interrupt is taken, MCAR is updated with the address of the data associated with the machine check. Note that if a machine check interrupt is caused by a signal, the MCAR contents are not meaningful. See *Machine check address register (MCAR/MCARU) on page 88*."

4.4 Exceptions

Exceptions are caused directly by instruction execution or by an asynchronous event. In either case, the exception may cause one of several types of interrupts to be invoked.

The following examples are of exceptions caused directly by instruction execution:

- 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 or to access a privileged SPR (privileged instruction exception-type program interrupt)
- In general, an attempt by an application program to access a nonexistent SPR (unimplemented operation instruction exception-type program interrupt). Note the following behavior defined by the EIS:
  - If MSR[PR] = 1 (user mode), SPR bit 5 = 0 (user-accessible SPR), and the SPR number is invalid, an illegal instruction exception is taken.
  - If MSR[PR] = 0 (supervisor mode) and the SPR number is invalid, an illegal instruction exception is taken.
  - If MSR[PR] = 1, SPR bit 5 = 1, and invalid SPR address (supervisor-only SPR), a privileged instruction exception-type program interrupt is taken.
- 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 location that is either unavailable (instruction or data TLB error interrupt) or not permitted (instruction or data storage interrupt)
- An attempt to access a location with an effective address alignment not supported by the implementation (alignment interrupt)
- Execution of a System Call (**sc**) instruction (system call interrupt)
- Execution of a **trap** instruction whose trap condition is met (trap interrupt type)
- Execution of a floating-point instruction when floating-point instructions are unavailable (floating-point unavailable interrupt)
- Execution of a floating-point instruction that causes a floating-point enabled exception to exist (enabled exception-type program interrupt)
- Execution of a defined instruction that is not implemented (illegal instruction exception or unimplemented operation exception-type program interrupt)
- Execution of an allocated instruction that is not implemented (illegal instruction exception or unimplemented operation exception-type program interrupt)
- Execution of an allocated instruction when the auxiliary instruction is unavailable (auxiliary unavailable interrupt)
- Execution of an allocated instruction that causes an auxiliary enabled exception (enabled exception-type program interrupt)

Invocation of an interrupt is precise, except that if one of the imprecise modes for invoking a floating-point enabled exception-type program interrupt is in effect, the invocation may be imprecise. When the interrupt is invoked imprecisely, the excepting instruction does not appear to complete before the next instruction starts (because the invocation of the interrupt required to complete execution has not occurred).

4.5 Interrupt classes

All interrupts except machine check are categorized by two independent characteristics:

- Critical/noncritical. Some interrupt types demand immediate attention even if other interrupt types being processed have not had the opportunity to save the machine state (that is, return address and captured state of the MSR). To enable taking a critical interrupt immediately after a noncritical interrupt is taken (that is, before the machine state is saved), two sets of save/restore register pairs are provided. Critical interrupts use CSRR0/CSRR1, and noncritical interrupts use SRR0/SRR1.
- Asynchronous/synchronous. Asynchronous interrupts are caused by events external to instruction execution; synchronous interrupts are caused by instruction execution and are either precise or imprecise. *Table 150* describes asynchronous and synchronous interrupts.

Table 150. Asynchronous and synchronous interrupts

Class	Description
Asynchronous	Caused by events independent from instruction execution. For asynchronous interrupts, the address reported to the interrupt handling routine is the address of the instruction that would have executed next, had the asynchronous interrupt not occurred.

Table 150. Asynchronous and synchronous interrupts (continued)

Class	Description
Synchronous, Precise	Caused directly by instruction execution. Synchronous interrupts are precise or imprecise. These interrupts precisely indicate the address of the instruction causing the exception or, for certain synchronous, precise interrupt types, the address of the immediately following instruction. When the execution or attempted execution of an instruction causes a synchronous, precise interrupt, the following conditions exist at the interrupt point: Whether SRR0 or CSRR0 addresses the instruction causing the exception or the next instruction is determined by the interrupt type and status bits. An interrupt is generated such that all instructions before the instruction causing the exception appear to have completed with respect to the executing processor. However, some accesses associated with these preceding instructions may not have been performed with respect to other processors and mechanisms. The exception-causing instruction may appear not to have begun execution (except for causing the exception), may be partially executed, or may have completed, depending on the interrupt type. See *Chapter 4.9*." Architecturally, no instruction beyond the exception-causing instruction executed.
Synchronous, Imprecise	Imprecise interrupts may indicate the address of the instruction causing the exception that generated the interrupt or some instruction after that instruction. When 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 exception-causing instruction or some instruction following the exception-causing instruction 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 context synchronization forces the imprecise interrupt due to an instruction that causes another exception that generates an interrupt (for example, alignment or data storage interrupt), SRR0 addresses the interrupt-forcing instruction, which may have partially executed (see *Chapter 4.9*"). If execution synchronization forces an imprecise interrupt due to an execution-synchronizing instruction other than **msync** or **isync**, SRR0 or CSRR0 addresses the interrupt-forcing instruction, which appears not to have begun execution (except for its forcing the imprecise interrupt). If the interrupt is forced by **msync** or **isync**, SRR0 or CSRR0 may address **msync** or **isync**, or the following instruction. If context or execution synchronization forces an imprecise interrupt, the instruction addressed by SRR0 or CSRR0 may have partially executed (see *Chapter 4.9*"). No instruction following the instruction addressed by SRR0 or CSRR0 has executed.

4.5.1 Requirements for system reset generation

Book E does not specify a system reset interrupt as was defined in the AIM version of the PowerPC architecture. A system reset is typically initiated in one of the following ways:

- Assertion of a signal that resets the internal state of the core complex
- By writing a 1 to DBCR0[34], if MSR[DE] = 1

4.6 Interrupt processing

Associated with each kind of interrupt is an interrupt vector, the address of the initial instruction that is executed when an 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 causes an interrupt to be generated and it has been determined that the interrupt can be taken, the following steps are performed:

1. SRR0 (for noncritical class interrupts) or CSRR0 (for critical class interrupts) or MCSRR0 for machine check interrupts is loaded with an instruction address that depends on the type of interrupt; see the specific interrupt description for details.
2. The ESR or MCSR is loaded with information specific to the exception type. Note that many interrupt types can only be caused by a single type of exception event, and thus do not need nor use an ESR setting to indicate the cause of the interrupt.
3. SRR1 (for noncritical class interrupts) or CSRR1 (for critical class interrupts) or MCSRR1 for machine check interrupts is loaded with a copy of the MSR contents.
4. New MSR values take effect beginning with the first instruction following the interrupt. The MSR is updated as follows:
   - MSR[SPE,WE,EE,PR,FP,FE0,FE1,IS,DS] are cleared by all interrupts.
   - MSR[CE,DE] are cleared by critical class interrupts and unchanged by noncritical class interrupts.
   - MSR[ME] is cleared by machine check interrupts and unchanged by other interrupts.
   - Other defined MSR bits are unchanged by all interrupts.

   MSR fields are described in *Chapter 2.6.1: Machine state register (MSR) on page 68*."
5. Instruction fetching and execution resumes, using the new MSR value, at a location specific to the interrupt type (IVPR[32–47] || IVOR*n*[48–59] || 0b0000)

   The IVOR*n* for the interrupt type is described in *Table 151*. IVPR and IVOR contents are indeterminate upon reset and must be initialized by system software.

Interrupts do 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 a noncritical interrupt handling routine, executing **rfi** causes the MSR to be restored from the SRR1 contents and instruction execution to resume at the address contained in SRR0. Likewise, **rfci** and **rfmci** perform the same function at the end of critical and machine check interrupt handling routines respectively, using the critical and machine check save/restore registers.

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.**—Clears outstanding reservations to prevent pairing a **lwarx** in the old process with a **stwcx.** in the new one*

***msync**—Ensures that memory operations of an interrupted process complete with respect to other processors before that process begins executing on another processor*

***rfi**, **rfci**, **rfmci**, or **isync**—Ensures that instructions in the new process execute in the new context*

4.7 Interrupt definitions

Table 151 summarizes each interrupt type, the various exception types that may cause that interrupt, the interrupt classification, which ESR bits can be set, which MSR bits can mask the interrupt type, and which IVOR is used to specify the vector address.

Table 151. Interrupt and exception types

IVOR	Interrupt Type	Exception Type	Exception Class(1)	ESR(2)	Mask Bits	Notes	Page
IVOR0	Critical input	Critical input	A, C	—	MSR[CE]	(3)	*4.7.1 on page 258*
IVOR1	Machine check	Machine check	C	—	MSR[ME]	(4),(5)	*4.7.2 on page 259*
IVOR2	Data storage (DSI)	Access	SP	[SPE],[ST], [FP,AP]	—	(6)	*4.7.3 on page 260*
		Load reserve or store conditional to write- through required location (W = 1)	SP	[ST]	—	6	
		Cache locking	SP	{DLK_0,DLK_1} [DLK,ILK],[ST]	—	(7)	
		Byte ordering	SP	[ST],[FP,AP],BO	—	—	
IVOR3	Instruction storage (ISI)	Access	SP	—	—	—	*4.7.4 on page 262*
		Byte ordering	SP	BO	—	—	
IVOR4	External input		A	—	MSR[EE]	3	*4.7.5 on page 263*
IVOR5	Alignment		SP	[ST],[FP,AP], [SPE,AP,ST]	—	—	*4.7.6 on page 263*
IVOR6	Program	Illegal	SP	PIL	—	—	*4.7.7 on page 265*
		Privileged	SP	PPR,[AP]	—	—	
		Trap	SP	PTR	—	—	
		Floating-point enabled	SP, SI	FP,[PIE]	MSR[FE0,FE1]	(8),(9)	
		Auxiliary processor enabled	SP	AP	—	9	
		Unimplemented op	SP	PUO,[FP,AP]	—	11	
IVOR7	Floating-point unavailable		SP		—		*4.7.8 on page 267*
IVOR8	System call		SP	—	—	—	*4.7.9 on page 267*
IVOR9	Auxiliary processor unavailable		SP		—		*4.7.10 on page 267*

Table 151. Interrupt and exception types (continued)

IVOR	Interrupt Type	Exception Type	Exception Class(1)	ESR(2)	Mask Bits	Notes	Page
IVOR10	Decrementer		A	—	MSR[EE], TCR[DIE]	—	*4.7.11 on page 268*
IVOR11	Fixed interval timer		A	—	MSR[EE], TCR[FIE]	—	*4.7.12 on page 268*
IVOR12	Watchdog		A, C	—	MSR[CE], TCR[WIE]	—	*4.7.13 on page 269*
IVOR13	Data TLB error	Data TLB miss	SP	[SPE],[ST], [FP,AP]	—	—	*4.7.14 on page 269*
IVOR14	Instruction TLB error	Instruction TLB miss	SP	—	—	—	*4.7.15 on page 270*
IVOR15	Debug	Trap (synchronous)	A, SP, C	—	MSR[DE], DBCR0[IDM]	—	*4.7.16 on page 271*
		Instruction address compare (synchronous)	A, SP, C	—	MSR[DE], DBCR0[IDM]	—	
		Data address compare (synchronous)	A, SP, C	—	MSR[DE], DBCR0[IDM]	—	
		Instruction complete	SP, C	—	MSR[DE], DBCR0[IDM]	(10)	
		Branch taken	SP, C	—	MSR[DE], DBCR0[IDM]	10	
		Return from interrupt	SP, C	—	MSR[DE], DBCR0[IDM]	—	
		Interrupt taken	SI, C	—	MSR[DE], DBCR0[IDM]	—	
		Unconditional debug event	SI, C	—	MSR[DE], DBCR0[IDM]	—	
IVOR32	SPE / Embedded FP APU unavailable	SPE APU unavailable	SP	—	—	(11)	*on page 272*
IVOR33	Embedded FP data	Embedded FP data exception	SP	—	—	11	*on page 272*
IVOR34	Embedded FP round	Embedded FP round exception	SP	—	—	11	*on page 273*

1. A = asynchronous, C = critical, SI = synchronous, imprecise, SP = synchronous, precise
2. In general, when an interrupt causes an ESR bit or bits to be set (or cleared) as indicated in the table, it also causes all other ESR bits to be cleared. Special rules may apply for implementation-specific ESR bits

   Legend: xxx (no brackets) means ESR[xxx] is set.
   [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] may be set, or possibly both.
3. Although not part of Book E, system interrupt controllers commonly provide independent mask and status bits for critical input and external input interrupt sources.

4. Machine check interrupts are not asynchronous or synchronous. See *Chapter 4.7.2*."
5. Machine check status information is commonly provided as part of the system implementation but is not part of Book E.
6. Software must examine the instruction and the subject TLB entry to determine the exact cause of the interrupt.
7. Cache locking and cache locking exceptions are implementation-dependent.
8. The precision of the floating-point enabled exception type is controlled by MSR[FE0,FE1], as described in <Cross Refs>Table 161. See *Chapter 4.7.7*." Also, exception status on the exact cause is available in the FPSCR. (See *Chapter 2.4.2: Floating-point status and control register (FPSCR) on page 58*.")

   The precision of the auxiliary processor enabled exception type program interrupt is implementation-dependent.
9. Auxiliary processor exception status is commonly provided as part of the implementation and is not part of Book E.
10. Instruction complete and branch taken debug events are defined only for MSR[DE] = 1 for internal debug mode DBCR0[IDM] = 1. In other words, for internal debug mode with MSR[DE] = 0, instruction complete and branch taken debug events cannot occur, no DBSR status bits are set, and no subsequent imprecise debug interrupt can occur.
11. EIS-defined exception

4.7.1 Critical input interrupt

A critical input interrupt occurs when no higher priority exception exists, a critical input exception is presented to the interrupt mechanism, and MSR[CE] = 1. The specific definition of a critical input exception is implementation-dependent but is typically caused by assertion of an asynchronous signal that is part of the system. In addition to MSR[CE], implementations may provide other ways to mask the critical input interrupt.

CSRR0, CSRR1, and MSR are updated as shown in *Table 152*.

Table 152. Critical input interrupt register settings

Register	Setting
CSRR0	Set to the effective address of the next instruction to be executed
CSRR1	Set to the MSR contents at the time of the interrupt
MSR	ME is unchanged. All other MSR bits are cleared.

Instruction execution resumes at address IVPR[32–47] || IVOR0[48–59] || 0b0000.

Critical interrupt input signals are level sensitive; to guarantee that the core complex can take a critical input interrupt, the critical input interrupt signal must be asserted until the interrupt is taken. Otherwise, whether the core complex takes an critical interrupt depends on whether MSR[CE] is set when the critical interrupt signal is asserted.

Note: To avoid redundant critical input interrupts, software must take any actions required by the implementation to clear any critical input exception status before reenabling MSR[CE].

4.7.2 Machine check interrupt

The EIS defines the machine check APU, which differs from the Book E definition of the machine check interrupt as follows:

- Book E defines machine check interrupts as critical interrupts, but the machine check APU treats them as a distinct interrupt type.
- Machine check is no longer a critical interrupt but uses MCSRR0 and MCSRR1 to save the return address and the MSR in case the machine check is recoverable.
- Return from machine check interrupt instruction (**rfmci**) is implemented to support the return to the address saved in MCSRR0.
- An address related to the machine check may be stored in MCAR, according to *Table 153*.
- A machine check syndrome register, MCSR, is used to log the cause of the machine check (instead of ESR). The MCSR is described in *Table 153*.

The following general information applies to both the Book E and EIS definitions. A machine check interrupt occurs when no higher priority exception exists, a machine check exception is presented to the interrupt mechanism, and MSR[ME] = 1. Specific causes of machine check exceptions are implementation-dependent, as are the details of the actions taken on a machine check interrupt.

Machine check interrupts are typically caused by a hardware or memory subsystem failure or by an attempt to access an invalid address. They may be caused indirectly by execution of an instruction, but may not be recognized or reported until long after the processor has executed past the instruction that caused the machine check. As such, machine check interrupts are not thought of as synchronous or asynchronous nor as precise or imprecise.

The following general rules apply:

- No instruction after the one whose address is reported to the machine check interrupt handler in MCSRR0 has begun execution.
- 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 instructions certain to complete appear to have done so within the context existing before the machine check interrupt. No further interrupts (other than possible additional machine check interrupts) occur as a result of those instructions.

If MSR[ME] is cleared, the processor enters checkstop state immediately on detecting the machine check condition.

When a machine check interrupt is taken, registers are updated as shown in *Table 153*.

Table 153. Machine check interrupt settings

Register	Setting
CSRR0(1)	Set to an instruction address. As closely as possible, set to the effective address of an instruction that was executing or about to be executing when the machine check exception occurred.
CSRR1 1	Set to the MSR contents at the time of the interrupt
MSR	UCLE, SPE, WE, CE, EE, PR, FP, ME, FE0, FE1, DE, IS, DS, PMM, and RI are cleared.
ESR	Implementation-dependent. The EIS uses the MCSR rather than the ESR.
Machine Check APU Registers	

Table 153. Machine check interrupt settings (continued)

Register	Setting
MCSRR0	On a best-effort basis, the core complex sets this to an effective address of some instruction that was executing or about to be executing when the machine check condition occurred.
MCSRR1	MSR[37–38,46–55,57–59,61–63] are loaded with equivalent MSR bits. All other bits are reserved.
MCAR/ MCARU	When a machine check interrupt is taken, the machine check address register is updated with the address of the data associated with the machine check. Note that if a machine check interrupt is caused by a signal, the MCAR contents are not meaningful. See *Machine check address register (MCAR/MCARU) on page 88.* MCARU is an alias to the upper 32 bits of MCAR.
MCSR	Set according to the machine check condition. See *Table 20*.

1. These registers are used if the machine check APU is not implemented.

Instruction execution resumes at address IVPR[32–47] || IVOR1[48–59] || 0b0000.

Note: 1 If a memory subsystem error causes a machine check interrupt, the subsystem may return incorrect data, which may be placed into registers or on-chip caches.

*2 For implementations on which a machine check interrupt is caused by referring to an invalid physical address, executing **dcbz** or **dcba** can cause a delayed machine check interrupt by establishing a data cache block associated with an invalid physical address. A machine check interrupt can occur later if and when an attempt is made to write that block to main memory, 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 **dcbst** or **dcbf**.*

4.7.3 Data storage interrupt

A data storage interrupt (DSI) occurs when no higher priority exception exists and a data storage exception is presented to the interrupt mechanism. *Table 154* describes exception conditions for a data storage interrupt as defined by Book E.

Table 154. Data storage interrupt exception conditions

Exception	Cause
Read access control exception	Occurs when either of the following conditions exists: ● In user mode (MSR[PR] = 1), a load or load-class cache management instruction attempts to access a memory location that is not user-mode read enabled (page access control bit UR = 0). ● In supervisor mode (MSR[PR] = 0), a load or load-class cache management instruction attempts to access a location that is not supervisor-mode read enabled (page access control bit SR = 0).
Write access control exception	Occurs when either of the following conditions exists: ● In user mode (MSR[PR] = 1), a store or store-class cache management instruction attempts to access a location that is not user-mode write enabled (page access control bit UW = 0). ● In supervisor mode (MSR[PR] = 0), a store or store-class cache management instruction attempts to access a location that is not supervisor-mode write enabled (page access control bit SW = 0).

Table 154. Data storage interrupt exception conditions (continued)

Exception	Cause
Byte-ordering exception	The implementation cannot access data in the byte order specified by the page's endian attribute. *Note: Note: The byte-ordering exception is provided to assist implementations that cannot support dynamically switching byte ordering between consecutive accesses, the byte order for a class of accesses, or misaligned accesses using a specific byte order.* Load/store accesses that cross a page boundary such that endianness changes cause a byte-ordering exception.
Cache locking exception	(EIS) The locked state of one or more cache lines has the potential to be altered. This exception is implementation-dependent. A cache locking exception occurs with the execution of **icbtls**, **icblc**, **dcbtls**, **dcbtstls**, or **dcblc** when (MSR[PR] = 1) and (MSR[UCLE] = 0). ESR is set as follows: ● For **icbtls** and **icblc**, ESR[ILK] is set. ● For **dcbtls**, **dcbtstls**, or **dcblc**, ESR[DLK] is set. Book E refers to this as a cache-locking exception.
Storage synchronization exception	Occurs when either of the following conditions exists: ● 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, the instruction executes correctly.) ● A store conditional instruction produces an effective address for which a normal store would cause a data storage interrupt but the processor does not have the reservation from a load and reserve instruction. Book E states that it is implementation-dependent whether a data storage interrupt occurs. The EIS defines that the data storage interrupt is taken.

Instructions **icbt**, **dcbt**, **dcbtst**, and **dcba**, and **lswx** or **stswx** with a length of zero cannot cause a data storage interrupt, regardless of the effective address.

Note: ***icbi*** *and* ***icbt*** *are treated as loads from the addressed byte with respect to address translation and protection. They use MSR[DS], not MSR[IS], to determine translation for their operands. Instruction storage interrupts and instruction TLB error interrupts are associated with instruction fetching and not execution. Data storage interrupts and data TLB error interrupts are associated with the execution of instruction cache management instructions.*

When a data storage interrupt occurs, the processor suppresses execution of the instruction causing the data storage exception.

SRR0, SRR1, ESR, MSR, and DEAR, are updated as follows:

Table 155. Data Storage Interrupt Register Settings

Register	Setting
SRR0	Set to the effective address of the instruction causing the interrupt
SRR1	Set to the MSR contents at the time of the interrupt
ESR	FPSet if the instruction causing the interrupt is a floating-point load or store; otherwise cleared STSet if the instruction causing the interrupt is a store or store-class cache management instruction; otherwise cleared DLKDLK is set when a DSI occurs because **dcbtls**, **dcbtstls**, or **dcblc** is executed in user mode and MSR[UCLE] = 0. APSet if the instruction causing the interrupt is an auxiliary processor load or store; otherwise cleared BOSet if the instruction caused a byte-ordering exception; otherwise cleared All other defined ESR bits are cleared.
MSR	CE, ME, and DE are unchanged. All other MSR bits are cleared.
DEAR	Set to the effective address of a byte that lies both within the range of bytes being accessed by the access or cache management instruction and within the page whose access caused the exception

Instruction execution resumes at address IVPR[32–47] || IVOR2[48–59] || 0b0000.

4.7.4 Instruction storage interrupt

An instruction storage interrupt occurs when no higher priority exception exists and an instruction storage exception is presented to the interrupt mechanism. Instruction storage exception conditions are described in *Table 156*.

Table 156. Instruction storage interrupt exception conditions

Exception	Cause
Execute access control exception	In user mode, an instruction fetch attempts to access a memory location that is not user mode execute enabled (page access control bit UX = 0). In supervisor mode, an instruction fetch attempts to access a memory location that is not supervisor mode execute enabled (page access control bit SX = 0).
Byte-ordering exception	The implementation cannot fetch the instruction in the byte order specified by the page's endian attribute. The EIS defines that accesses that cross a page boundary such that endianness changes cause a byte-ordering exception.

Note that Book E provides this exception to assist implementations that cannot dynamically switch byte ordering between consecutive accesses, do not support the byte order for a class of accesses, or do not support misaligned accesses using a specific byte order.

When an instruction storage interrupt occurs, the processor suppresses execution of the instruction causing the exception.

SRR0, SRR1, MSR, and ESR are updated as shown in *Table 157*.

Table 157. Instruction storage interrupt register settings

Register	Setting
SRR0	Set to the effective address of the instruction causing the instruction storage interrupt
SRR1	Set to the MSR contents at the time of the interrupt
MSR	CE, ME, and DE are unchanged. All other MSR bits are cleared.
ESR	BO is set if the instruction fetch caused a byte-ordering exception; otherwise cleared. All other defined ESR bits are cleared.

Note: Permissions violation and byte-ordering exceptions are not mutually exclusive. Even if ESR[BO] is set, system software must examine the TLB entry accessed by the fetch to determine whether a permissions violation also may have occurred.

Instruction execution resumes at address IVPR[32–47] || IVOR3[48–59] || 0b0000.

4.7.5 External input interrupt

An external input interrupt occurs when no higher priority exception exists, an external input exception is presented to the interrupt mechanism, and MSR[EE] = 1. The specific definition of an external input exception is implementation-dependent and is typically caused by assertion of an asynchronous signal that is part of the processing system.

To guarantee that the core complex can take an external interrupt, the external interrupt pin must be asserted until the interrupt is taken. Otherwise, whether the external interrupt is taken depends on whether MSR[EE] is set when the external interrupt signal is asserted.

In addition to MSR[EE], implementations may provide other ways to mask this interrupt.

SRR0, SRR1, and MSR are updated as shown in *Table 158*.

Table 158. External input interrupt register settings

Register	Setting
SRR0	Set to the effective address of the next instruction to be executed
SRR1	Set to the MSR contents at the time of the interrupt
MSR	CE, ME, and DE are unchanged. All other MSR bits are cleared.

Instruction execution resumes at address IVPR[32–47] || IVOR4[48–59] || 0b0000.

Note: To avoid redundant external input interrupts, software must take any actions required to clear any external input exception status before reenabling MSR[EE].

4.7.6 Alignment interrupt

An alignment interrupt occurs when no higher priority exception exists and an alignment exception is presented to the interrupt mechanism. An alignment exception may occur when an implementation cannot perform a data 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.
- A **dcbz** operand is in write-through-required or caching-inhibited memory, or **dcbz** is executed in an implementation with no data cache or a write-through data cache.
- The operand of a store, except store conditional, is in write-through required memory.

The EIS defines the following alignment exception conditions:

- Execution of a **dcbz** references a page marked as write-through or cache inhibited.
- A load multiple word instruction (**lmw**) reads an address that is not a multiple of four.
- A **lwarx** or **stwcx**. instruction references an address that is not a multiple of four.
- SPFP and SPE APU instructions are not aligned on a natural boundary. A natural boundary is defined by the size of the data element being accessed.
- A vector operation reports an exception if the physical address of the following instructions is not aligned to the 64-bit boundary: **evldd**, **evlddx**, **evldw**, **evldwx**, **evldh**, **evldhx**, **evstdd**, **evstddx**, **evstdw**, **evstdwx**, **evstdh**, and **evstdhx**. *Table 159* describes additional ESR settings.

For **lmw** and **stmw** with a non–word-aligned operand and for load and reserve and store conditional instructions with an misaligned operand, an implementation may yield boundedly undefined results instead of causing an alignment interrupt. A store conditional to a write-through required location may either cause an alignment or data storage interrupt or may 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**, correct execution means clearing each byte of the block in main memory.

Note: Book E does not support use of an misaligned effective address by load and reserve and store conditional instructions. If an misaligned effective address is specified, the alignment interrupt handler should treat the instruction as a programming error and must not attempt to emulate the instruction.

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 shown in *Table 159*.

Table 159. Alignment interrupt register settings

Register	Setting
SRR0	Set to the effective address of the instruction causing the alignment interrupt
SRR1	Set to the MSR contents at the time of the interrupt
MSR	CE, ME, and DE are unchanged. All other MSR bits are cleared.
DEAR	Set to the EA of a byte that is both within the range of the bytes being accessed by the memory access or cache management instruction, and within the page whose access caused the alignment exception
ESR	FP Set if the instruction causing the interrupt is a floating-point load or store; otherwise cleared ST Set if the instruction causing the interrupt is a store; otherwise cleared AP Set if the instruction causing the interrupt is an auxiliary processor load or store; otherwise cleared The following bits may be affected for vector alignment exception conditions: SPE Set AP Set (May not be supported on all processors) ST Set only if the instruction causing the exception is a store and is cleared for a load All other defined ESR bits are cleared.

Instruction execution resumes at address IVPR[32–47] || IVOR5[48–59] || 0b0000.

4.7.7 Program interrupt

A program interrupt occurs when no higher priority exception exists and a program exception is presented to the interrupt mechanism. A program interrupt is caused when any of the following exceptions occurs during execution of an instruction.

Table 160. Program interrupt exception conditions

Exception	Cause
Floating-point enabled exception	Caused when (MSR[FE0] | MSR[FE1]) & FPSCR[FEX] = 1. FPSCR[FEX] is set 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 FPSCR control bits. See *Chapter 2.4.2: Floating-point status and control register (FPSCR) on page 58*." Whether the interrupt is precise or imprecise is determined by MSR[FE0,FE1], as described in *Table 20*.
Auxiliary processor enabled exception	Implementation dependent
Illegal instruction exception	Always occurs when execution of any of the following kinds of instructions is attempted. – A reserved-illegal instruction – In user mode, an **mtspr** or **mfspr** that specifies an SPRN value with SPRN[5] = 0 (user-mode accessible) that represents an unimplemented SPR – (EIS) If an invalid SPR address is accessible only in supervisor mode and the processor is in supervisor mode (MSR[PR] = 0), results are undefined. – (EIS) If the invalid SPR address is accessible only in the supervisor mode and the processor is in user mode (MSR[PR] = 1), a privileged instruction exception is taken. 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. See the user's manual for the implementation. – An instruction that is in invalid form (boundedly undefined results). – An **lswx** instruction for which **r**A or **r**B is in the range of registers to be loaded (boundedly undefined results) – A reserved no-op instruction (no-operation performed is preferred). – A defined or allocated instruction that is not implemented (unimplemented operation exception). Unimplemented Book E instructions take an illegal instruction exception. – The EIS defines that an attempt to execute a 64-bit Book E instruction causes an illegal instruction exception.
Privileged instruction exception	Occurs when MSR[PR] = 1 and execution is attempted of any of the following: – A privileged instruction – An **mtspr** or **mfspr** instruction that specifies a privileged SPR (SPRN[5] = 1) – (EIS) An **mtpmr** or **mfpmr** instruction that specifies a privileged PMR (PMRN[5] = 1)
Trap exception	Occurs when any of the conditions specified in a trap instruction are met.
Unimplemented operation exception	May occur when a defined or allocated instruction is encountered that is not implemented. Otherwise an illegal instruction exception occurs. See the reference manual for the implementation.

Whether a floating-point enabled interrupt is precise or imprecise is determined by MSR[FE0,FE1], as described in *Table 161*.

Table 161. MSR[FE0,FE1] settings

FE0,FE1	Description
01,10	Imprecise. When such a program interrupt is taken, if the address saved in SRR0 is not that of the instruction that caused the exception (that is, the instruction that caused FPSCR[FEX] to be set), ESR[PIE] is set. Note that some implementations may ignore these bit settings and treat all affected interrupts as precise.
11	Precise.
00	The interrupt is masked and the interrupt subsequently occurs if and when floating-point enabled exception-type program interrupts are enabled by setting either or both FE0,FE1 and also causes ESR[PIE] to be set.

SRR0, SRR1, MSR, and ESR are updated as shown in *Table 162*.

Table 162. Program interrupt register settings

Register	Description
SRR0	For all program interrupts except an enabled exception when in an imprecise mode (see *Table 164*), set to the EA of the instruction that caused the interrupt. For an imprecise enabled exception, set to the EA of the excepting instruction or of some subsequent instruction that has not been executed (in which case ESR[PIE] is set). If the instruction is **msync** or **isync**, SRR0 does not point more than 4 bytes beyond the **msync** or **isync**. If FPSCR[FEX] = 1 but both MSR[FE0,FE1] = 00, an enabled exception-type program interrupt occurs before or at the next synchronizing event if [FE0,FE1] are altered by any instruction so that the expression (MSR[FE0] | MSR[FE1]) & FPSCR[FEX] is 1. When this occurs, ESR[PIE] is set and SRR0 is loaded with the EA of the instruction that would have executed next, not with the EA of the instruction that modified MSR causing the interrupt.
SRR1	Set to the MSR contents at the time of the interrupt.
MSR	CE, ME, and DE are unchanged. All other MSR bits are cleared.
ESR	PIL Set if an illegal instruction exception-type program interrupt; otherwise cleared. PPR Set if a privileged instruction exception-type program interrupt; otherwise cleared. PTR Set if a trap exception-type program interrupt; otherwise cleared. PUO Set if an unimplemented operation exception-type program interrupt; otherwise cleared. FP Set if the instruction causing the interrupt is a floating-point instruction; otherwise cleared. PIE Set 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 (that is, the instruction that caused FPSCR[FEX] to be set); otherwise cleared. AP Set if the instruction causing the interrupt is an auxiliary processor instruction; otherwise cleared. All other defined ESR bits are cleared.

Instruction execution resumes at address IVPR[32–47] || IVOR6[48–59] || 0b0000.

4.7.8 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.

When a floating-point unavailable interrupt occurs, the processor suppresses execution of the instruction causing the floating-point unavailable interrupt.

SRR0, SRR1, and MSR are updated as shown in *Table 163*.

Table 163. Floating-point unavailable interrupt register settings

Register	Description
SRR0	Set to the effective address of the instruction that caused the interrupt.
SRR1	Set to the MSR contents at the time of the interrupt.
MSR	CE, ME, and DE are unchanged. All other MSR bits are cleared.

Instruction execution resumes at address IVPR[32–47]||IVOR7[48–59]||0b0000.

4.7.9 System call interrupt

A system call interrupt occurs when no higher priority exception exists and a System Call (**sc**) instruction is executed. SRR0, SRR1, and MSR are updated as shown in *Table 164*.

Table 164. System call interrupt register settings

Register	Description
SRR0	Set to the effective address of the instruction after the **sc** instruction.
SRR1	Set to the MSR contents at the time of the interrupt.
MSR	CE, ME, and DE are unchanged. All other MSR bits are cleared.

Instruction execution resumes at address IVPR[32–47] || IVOR8[48–59] || 0b0000.

4.7.10 Auxiliary processor unavailable interrupt

An auxiliary processor unavailable interrupt occurs when no higher priority exception exists, 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 and its configuration are implementation-dependent. See the reference manual for the implementation.

When an auxiliary processor unavailable interrupt occurs, the processor suppresses execution of the instruction causing the auxiliary processor unavailable interrupt.

Registers SRR0, SRR1, and MSR are updated as shown in *Table 165*.

Table 165. Auxiliary processor unavailable interrupt register settings

Register	Setting
SRR0	Set to the effective address of the instruction that caused the interrupt.
SRR1	Set to the MSR contents at the time of the interrupt.
MSR	CE, ME, and DE are unchanged. All other MSR bits are cleared.

Instruction execution resumes at address IVPR[32–47]||IVOR9[48–59]||0b0000.

4.7.11 Decrementer Interrupt

A decrementer interrupt occurs when no higher priority exception exists, a decrementer exception exists (TSR[DIS] = 1) & the interrupt is enabled (TCR[DIE] = 1 and MSR[EE] = 1).

MSR[EE] also enables external input and fixed-interval timer interrupts.

SRR0, SRR1, MSR, and TSR are updated as shown in *Table 166*.

Table 166. Decrementer interrupt register settings

Register	Setting
SRR0	Set to the effective address of the next instruction to be executed.
SRR1	Set to the MSR contents at the time of the interrupt.
MSR	CE, ME, and DE are unchanged. All other MSR bits are cleared.
TSR	DIS is set.

Instruction execution resumes at address IVPR[32–47] || IVOR10[48–59] || 0b0000.

*Note: To avoid redundant decrementer interrupts, before reenabling MSR[EE], the interrupt handling routine must clear TSR[DIS] by writing a word to TSR using **mtspr** with a 1 in any bit position to be cleared and 0 in all others. The data written to the TSR is not direct data, but a mask. Writing a 1 to this bit causes it to be cleared; writing a 0 has no effect.*

4.7.12 Fixed-interval timer interrupt

A fixed-interval timer interrupt occurs when no higher priority exception exists, a fixed-interval timer exception exists (TSR[FIS] = 1), and the interrupt is enabled (TCR[FIE] = 1 and MSR[EE] = 1).

The fixed-interval timer period is determined by TCR[FP], which, when concatenated with TCR[FPEXT], specifies one of 64 bit locations of the time base used to signal a fixed-interval timer exception on a transition from 0 to 1.

TCR[FPEXT],TCR[FP] = 000000 selects TBU[32]. TCR[FPEXT],TCR[FP] = 111111 selects TBL[63].

Note: MSR[EE] also enables external input and decrementer interrupts.

SRR0, SRR1, MSR, and TSR are updated as shown in *Table 167*.

Table 167. Fixed-interval timer interrupt register settings

Register	Setting
SRR0	Set to the effective address of the next instruction to be executed.
SRR1	Set to the MSR contents at the time of the interrupt.
MSR	CE, ME, and DE are unchanged. All other MSR bits are cleared.
TSR	FIS is set.

Instruction execution resumes at address IVPR[32–47] || IVOR11[48–59] || 0b0000.

*Note: To avoid redundant fixed-interval timer interrupts, before reenabling MSR[EE], the interrupt handling routine must clear TSR[FIS] by writing a word to TSR using **mtspr** with a 1 in any bit position to be cleared and 0 in all others. The data written to the TSR is not direct data, but a mask. Writing a 1 causes the bit to be cleared; writing a 0 has no effect.*

4.7.13 Watchdog timer interrupt

A watchdog timer interrupt occurs when no higher priority exception exists, a watchdog timer exception exists (TSR[WIS] = 1), and the interrupt is enabled (TCR[WIE] = 1 and MSR[CE] = 1).

MSR[CE] also enables the critical input interrupt.

CSRR0, CSRR1, MSR, and TSR are updated as shown in *Table 168*.

Table 168. Watchdog timer interrupt register settings

Register	Setting
CSRR0	Set to the effective address of the next instruction to be executed.
CSRR1	Set to the MSR contents at the time of the interrupt.
MSR	ME is unchanged; all other MSR bits are cleared.
TSR	WIS is set.

Instruction execution resumes at address IVPR[32–47] || IVOR12[48–59] || 0b0000.

*Note: To avoid redundant watchdog timer interrupts, before reenabling MSR[CE], the interrupt handling routine must clear TSR[WIS] by writing a word to TSR using **mtspr** with a 1 in any bit position to be cleared and 0 in all others. The data written to the TSR is not direct data, but a mask. Writing a 1 to this bit causes it to be cleared; writing a 0 has no effect.*

4.7.14 Data tlb error interrupt

A data TLB error interrupt occurs when no higher priority exception exists and the exception described in *Table 169* is presented to the interrupt mechanism.

Table 169. Data tlb error interrupt exception conditions

Exception	Description
Data TLB miss exception	Virtual addresses associated with an instruction fetch do not match any valid TLB entry.

If a store conditional instruction produces an effective address for which a normal store would cause a data TLB error interrupt, but the processor does not have the reservation from a load and reserve instruction, Book E defines it as implementation-dependent whether a data TLB error interrupt occurs. The EIS defines that the interrupt is taken.

When a data TLB error interrupt occurs, the processor suppresses execution of the instruction causing the data TLB error exception.

SRR0, SRR1, MSR, DEAR, and ESR are updated as shown in *Table 170*.

Table 170. Data tlb error interrupt register settings

Register	Setting
SRR0	Set to the effective address of the instruction causing the data TLB error interrupt.
SRR1	Set to the MSR contents at the time of the interrupt.
MSR	CE, ME, and DE are unchanged. All other MSR bits are cleared.
DEAR	Set to the EA of a byte that is both within the range of the bytes being accessed by the memory access or cache management instruction and within the page whose access caused the data TLB error exception.
ESR	STSet if the instruction causing the interrupt is a store, **dcbi**, or **dcbz** instruction; otherwise cleared. FPSet if the instruction causing the interrupt is a floating-point load or store; otherwise cleared. APSet if the instruction causing the interrupt is an auxiliary processor load or store; otherwise cleared. All other defined ESR bits are cleared.
MAS*n*	See *Table 193*.

Table 192 shows MAS register settings for data and instruction TLB error interrupts. *MAS register updates for exceptions, tlbsx, and tlbre on page 328*," describes how these values are set as defined by the EIS.

Instruction execution resumes at address IVPR[32–47] || IVOR13[48–59] || 0b0000.

4.7.15 Instruction tlb error interrupt

An instruction TLB error interrupt occurs when no higher priority exception exists and the exception described in *Table 171* is presented to the interrupt mechanism.

Table 171. Instruction TLB error interrupt exception conditions

Exception	Description
Instruction TLB miss exception	The virtual addresses associated with a fetch do not match any valid TLB entry.

When an instruction TLB error interrupt occurs, the processor suppresses execution of the instruction causing the instruction TLB miss exception.

SRR0, SRR1, and MSR are updated as shown in *Table 172*.

Table 172. Instruction TLB error interrupt register settings

Register	Setting
SRR0	Set to the effective address of the instruction causing the instruction TLB error interrupt.
SRR1	Set to the MSR contents at the time of the interrupt.
MSR	CE, ME, and DE are unchanged. All other MSR bits are cleared.
MAS*n*	See *Table 192*.

Instruction execution resumes at address IVPR[32–47] || IVOR14[48–59] || 0b0000.

4.7.16 Debug interrupt

A debug interrupt occurs when no higher priority interrupt exists, 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 DBSR bit to be set.

Table 173. Debug interrupt register settings

Register	Setting
CSRR0	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 (IAC registers), data address compare (DAC1R, DAC1W, DAC2R, and DAC2W), data value compare (DVC1 and 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 instruction that would have executed after the **rfi**, **rfci**, or **rfmci** that caused the debug interrupt. – For debug exceptions that occur while debug interrupts are disabled (DBCR0[IDM] = 0 or MSR[DE] = 0), a debug interrupt occurs at the next synchronizing event if DBCR0[IDM] and MSR[DE] are modified such that they are both set and if the debug exception status is still set in the DBSR. When this occurs, CSRR0 holds the address of the instruction that would have executed next, not the address of the instruction that modified DBCR0 or MSR and thus caused the interrupt.
CSRR1	Set to the MSR contents at the time of the interrupt.
MSR	ME is unchanged. All other MSR bits are cleared.
DBSR	Set to indicate type of debug event. (See *Chapter 2.13.2: Debug status register (DBSR) on page 116*.)

CSRR0, CSRR1, MSR, and DBSR are updated as shown in *Table 173*.

Instruction execution resumes at address IVPR[32–47] || IVOR15[48–59] || 0b0000.

4.7.17 EIS-defined interrupts

The interrupts in this section are defined by the EIS.

SPE/embedded floating-point APU unavailable interrupt

An SPE APU unavailable interrupt is taken if MSR[SPE] is cleared and an SPE, embedded scalar double-precision or embedded vector single-precision floating-point instruction is executed. It is not used by the embedded scalar single-precision floating-point APU.

When an SPE unavailable interrupt occurs, the processor suppresses execution of the instruction causing the interrupt. *Table 174* describes register settings. If the SPE/embedded floating-point unavailable interrupt occurs, the processor suppresses execution of the instruction causing the exception. The SRR0, SRR1, MSR, and ESR registers are modified as shown in *Table 174*.

Table 174. SPE/embedded floating-point APU unavailable interrupt register settings

Register	Setting
SRR0	Set to the effective address of the instruction causing the interrupt.
SRR1	Set to the MSR contents at the time of the interrupt.
MSR	CE, ME, and DE are unchanged. All other bits are cleared.
ESR	SPE (bit 24) is set. All other ESR bits are cleared.

Instruction execution resumes at address IVPR–47] || IVOR32[48–59] || 0b0000.

Embedded floating-point data interrupt

An embedded floating-point data interrupt is generated in the following cases:

- SPEFSCR[FINVE] = 1 and either SPEFSCR[FINVH,FINV] = 1
- SPEFSCR[FDBZE] = 1and either SPEFSCR[FDBZH,FDBZ] = 1
- SPEFSCR[FUNFE] = 1 and either SPEFSCR[FUNFH,FUNF] = 1
- SPEFSCR[FOVFE] = 1 and either SPEFSCR[FOVFH,FOVF] = 1

Note that although SPEFSCR status bits can be updated by using **mtspr**, interrupts occur only if they are set as the result of an arithmetic operation.

When an embedded floating-point data interrupt occurs, the processor suppresses execution of the instruction causing the interrupt. *Table 175* shows register settings.

Table 175. Embedded floating-point data interrupt register settings

Register	Setting
SRR0	Set to the effective address of the instruction causing the interrupt.
SRR1	Set to the MSR contents at the time of the interrupt.
MSR	CE, ME, and DE are unchanged. All other bits are cleared.
ESR	SPE (bit 24) is set. All other ESR bits are cleared.
SPEFSCR	One or more of the FINVH, FINV, FDBZH, FDBZ, FUNFH, FUNF, FOVFH, or FOVF bits are set to indicate the interrupt type.

Instruction execution resumes at address IVPR[32–47] || IVOR33[48–59] || 0b0000.

Embedded floating-point round interrupt

The embedded floating-point round interrupt is taken on any of the following conditions:

- SPEFSCR[FINXE] = 1 and any of the SPEFSCR[FGH,FXH,FG,FX] bits = 1
- SPEFSCR[FRMC] = 0b10 (+∞)
- SPEFSCR[FRMC] = 0b11 (–∞)

Note that although these SPEFSCR status bits can be updated by using an **mtspr**[SPEFSCR], interrupts occur only if they are set as the result of an arithmetic operation.

When an embedded floating-point round interrupt occurs, the unrounded (truncated) result is placed in the target register. *Table 176* describes register settings.

Table 176. Embedded floating-point round interrupt register settings

Register	Setting
SRR0	Set to the effective address of the instruction following the instruction causing the interrupt.
SRR1	Set to the MSR contents at the time of the interrupt.
MSR	CE, ME, and DE are unchanged. All other MSR bits are cleared.
ESR	SPE (bit 24) is set. All other ESR bits are cleared.
SPEFSCR	FGH, FXH, FG, FX, and FRMC are set appropriately to indicate the interrupt type.

Instruction execution resumes at address IVPR[32–47] || IVOR34[48–59] || 0b0000.

4.8 Performance monitor interrupt

The performance monitor provides a performance monitor interrupt that is triggered by an enabled condition or event. An enabled condition or event is as follows:

A PMC*x* register overflow condition occurs with the following settings:

- PMLCa*x*[CE] = 1; that is, for the given counter the overflow condition is enabled.
- PMC*x*[32] = 1; that is, the given counter indicates an overflow.

For a performance monitor interrupt to be signaled on an enabled condition or event, PMGC0[PMIE] must be set.

The performance monitor can also freeze the performance monitor counters triggered by an enabled condition or event. For the performance monitor counters to freeze on an enabled condition or event, PMGC0[FCECE] must be set.

Although the interrupt condition could occur with MSR[EE] = 0, the interrupt cannot be taken until MSR[EE] = 1. If a counter overflows while PMGC0[FCECE] = 0, PMLCa[CE] = 1, and MSR[EE] = 0, it is possible for the counter to wrap around to all zeros again without the performance monitor interrupt being taken.

The priority of the performance monitor interrupt is below that of the fixed-interval interrupt and above that of the decrementer interrupt.

4.9 Partially executed instructions

In general, the PowerPC architecture permits load and store instructions to be partially executed, interrupted, and then restarted from the beginning upon return from the interrupt. To guarantee that a particular load or store instruction completes without being interrupted and restarted, software must mark the memory as guarded and use an elementary (non-string or non-multiple) load or store aligned on an operand-sized boundary.

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 a target register **r**D or **fr**D has been altered.
- For update forms of load or store, the update register, **r**A, will not have been altered.

The following effects are permissible when certain instructions are partially executed and then restarted:

- For any store, bytes at the target 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 or not.
- For any load, bytes at the addressed 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 registers in the range to be loaded may have been altered. Including the addressing registers **r**A and possibly **r**B in the range to be loaded is a programming error, and thus the rules for partial execution do not protect these registers against overwriting.

In no case is access control violated.

As previously stated, elementary, aligned, guarded loads and stores are the only access instructions guaranteed not to 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, or guarded):
   - Any asynchronous interrupt
   - Machine check
   - Program (imprecise mode floating-point enabled)
   - Program (imprecise mode auxiliary processor enabled)
   - Decrementer
   - Fixed-interval timer
   - Watchdog timer
   - Debug (unconditional debug event)
2. Misaligned elementary load or store, or any multiple or string:

   All of the above listed under item 1, plus the following:
   - Alignment
   - Data storage (if the access crosses a protection boundary)
   - Debug (data address compare, data value compare)

The **mtcrf** and **mfcr** instructions can also be partially executed due to the occurrence of any of the interrupts listed under item 1 at the time **mtcrf** or **mfcr** was executing.

- All instructions before **mtcrf** or **mfcr** have completed execution. Some memory accesses generated by these preceding instructions may not have completed.
- No subsequent instruction has begun execution.
- The **mtcrf** or **mfcr** instruction, whose address was saved in SRR0/CSRR0 at the time of the interrupt, may appear not to have begun or may have partially executed.

4.10 Interrupt ordering and masking

Multiple exceptions that can each generate an interrupt can exist simultaneously. However, the PowerPC architecture does not provide for reporting multiple simultaneous interrupts of the same class (critical or noncritical). Therefore, the PowerPC architecture defines that interrupts must be ordered with one another and provides a way to mask certain persistent interrupt types.

When an interrupt type is masked (disabled) and an event causes an exception that would normally generate an interrupt of that type, the exception persists as a status bit in a register (which register depends upon the exception type) but no interrupt is generated. Later, if the interrupt type is enabled (unmasked) and the exception status has not been cleared by software, the interrupt due to the original exception event is finally generated. (A typical implementation has such a mechanism for certain debug events. A signal that triggers an asynchronous interrupt, such as external input, must be asserted until they are taken. There is no mechanism for saving the external interrupt if the signal is negated before the interrupt is taken. All interrupts are level-sensitive except for machine check, which is edge-triggered.)

All asynchronous interrupt types and some synchronous interrupt types can be masked. An example of a maskable synchronous interrupt type is the floating-point enabled exception-type program interrupt. The execution of a floating-point instruction that causes FPSCR[FEX] to be set is considered an exception event, regardless of the setting of MSR[FE0,FE1]. If MSR[FE0,FE1] are both 0, the floating-point enabled exception-type program interrupt is masked, but the exception persists in FPSCR[FEX]. Later, if MSR[FE0,FE1] are enabled, the interrupt is generated.

The PowerPC architecture allows implementations to avoid situations in which an interrupt would cause state information (saved in save/restore registers) from a previous interrupt to be overwritten and lost. As a first step, upon any noncritical class interrupt, hardware automatically disables further asynchronous, noncritical class interrupts (external input) by clearing MSR[EE]. Likewise, upon any critical class interrupt, hardware automatically disables further asynchronous interrupts, both critical and noncritical, by clearing MSR[CE] and MSR[EE]. Critical input, watchdog timer, and debug interrupts are disabled by clearing MSR[CE,DE]. Note that machine check interrupts, while considered neither asynchronous nor synchronous, are not maskable by MSR[CE,DE,EE] and could be presented in a situation that could cause loss of state information.

This first step of clearing MSR[EE] (and MSR[CE,DE] for critical class interrupts) prevents subsequent asynchronous interrupts from overwriting save/restore registers before software can save their contents. On any interrupt, hardware also automatically clears MSR[WE,PR,FP,FE0,FE1,IS,DS], which helps avoid subsequent interrupts of certain other types. However, guaranteeing that these interrupt types do not occur also requires system software to avoid executing instructions that could cause (or enable) a subsequent interrupt, if SRR1 contents have not been saved.

4.10.1 Guidelines for system software

Table 177 lists actions system software must avoid before saving save/restore register contents.

Table 177. Operations to avoid

Operation	Reason
Reenabling MSR[EE] (or MSR[CE,DE] in critical class interrupt handlers)	Prevents any asynchronous interrupts, snd (in the case of MSR[DE]) any debug interrupts, including synchronous and asynchronous types
Branching (or sequential execution) to addresses not mapped by the TLB, mapped without UX = 1 or SX = 1 permission, or causing large address or instruction address overflow exceptions.	Prevents instruction storage, instruction TLB error, and instruction address overflow interrupts
Load, store, or cache management instructions to addresses not mapped by the TLB or not having required access permissions.	Prevents data storage and data TLB error interrupts
Execution of system call (**sc**) or trap (**tw**, **twi**, **td**, **tdi**)	Prevents system call and trap exception-type program interrupts
Execution of any floating-point instruction	Prevents floating-point unavailable interrupts. Note that this interrupt would occur upon execution of any floating-point instruction, due to the automatic clearing of MSR[FP]. However, even if software were to reenable MSR[FP], floating-point instructions must still be avoided to prevent program interrupts due to various possible program interrupt exceptions (floating-point enabled, unimplemented operation).
Reenabling of MSR[PR]	Prevents privileged instruction exception-type program interrupts. Alternatively, software could reenable MSR[PR] but avoid executing any privileged instructions.
Execution of any auxiliary processor instruction	Prevents auxiliary processor unavailable, auxiliary processor enabled type, and unimplemented operation type program interrupts
Execution of any illegal instructions	Prevents illegal instruction exception-type program interrupts
Execution of any instruction that could cause an alignment interrupt	Prevents alignment interrupts, including string or multiple instructions and misaligned elementary load or store instructions. *Chapter 4.7.6: Alignment interrupt*," lists instructions that cause alignment interrupts.

If the machine check APU is not implemented, machine check interrupts are a special case. Machine checks are critical interrupts, but normal critical interrupts (critical input, watchdog timer, and debug) do not automatically disable machine checks. Machine checks are disabled by clearing MSR[ME], and only a machine check interrupt itself automatically clears this bit. Thus there is always the risk that a machine check interrupt could occur

within a normal critical interrupt handler before it saves the save/restore registers' contents. In such a case, the interrupt may not be recoverable.

It is unnecessary for hardware or software to avoid critical-class interrupts from within noncritical-class interrupt handlers (hence hardware does not automatically clear MSR[CE,ME,DE] on a noncritical interrupt), since the two interrupt classes use different save/restore registers. However, because a critical class interrupt can occur within a noncritical handler before the noncritical handler saves SRR0/SRR1, hardware and software must cooperate to avoid both critical and noncritical class interrupts from within critical class interrupt handlers. Therefore, within the critical class interrupt handler, both pairs of save/restore registers may contain data necessary to system software.

4.10.2 Interrupt order

Enabled interrupt types for which simultaneous exceptions can exist are prioritized as follows:

1. Synchronous (non-debug) interrupts:
   - Data storage
   - Instruction storage
   - Alignment
   - Program
   - Floating-point unit unavailable
   - Auxiliary processor unavailable
   - System call
   - Data TLB error
   - Instruction TLB error

   Only one of the above synchronous interrupt types may have an existing exception generating it at a given time. This is guaranteed by the exception priority mechanism (see *Chapter 4.11: Exception priorities*") and the sequential execution model.
2. Machine check
3. Debug
4. Critical input
5. Watchdog timer
6. External input
7. Fixed-interval timer
8. Decrementer

Although, as indicated above, noncritical, synchronous exception types listed under item 1 are generated with higher priority than critical interrupt types in items 2–5, noncritical interrupts are immediately followed by the highest priority existing critical interrupt type, without executing any instructions at the noncritical interrupt handler. This is because noncritical interrupt types do not automatically disable MSR mask bits for critical interrupt types (CE and ME). In all other cases, a particular interrupt type listed above automatically disables subsequent interrupts of the same type, as well as all lower priority interrupt types.

4.11 Exception priorities

Book E requires all synchronous (precise and imprecise) exceptions to be reported in program order, as required by the sequential execution model. The one exception to this rule is the case of multiple synchronous imprecise exceptions. Upon a synchronizing event, all previously executed instructions are required to report any synchronous imprecise interrupt-generating exceptions, and the interrupt is then generated with all of those exception types reported cumulatively in the ESR and in any status registers associated with the particular exception type (such as the FPSCR).

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 is 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 *Chapter 4.10.2: Interrupt order*." The exception priority mechanism also prevents certain debug exceptions from existing in combination with certain other synchronous interrupt-generating exceptions.

The EIS defines priorities for all exceptions including those defined in optional APUs. Interrupt types are defined as either synchronous (the interrupt is as a direct result of an instruction in execution) or asynchronous, (the interrupt results from an event external to the execution of a particular instruction or an instruction removes a gating condition to a pending exception). Except for machine check interrupts, which can be either synchronous or asynchronous, interrupts are either synchronous or asynchronous exclusively.

Because asynchronous interrupts may temporally be sampled either before or after an instruction is completed, an implementation can order asynchronous interrupts among only asynchronous interrupts and order synchronous interrupts among only synchronous interrupt. The distinction is important because synchronous interrupts that require post-completion actions (such as system call or debug instruction complete exceptions) cannot be separated from the completion of the instruction. Therefore, asynchronous interrupts cannot be sampled during the completion and post-completion synchronous exceptions for a given instruction.

The relative priorities for asynchronous exceptions is given in *Table 178* and for synchronous exceptions in and *Table 179*. In many cases, certain exceptions cannot occur at the same time (for example, program-trap and program-Illegal cannot occur on the same instruction). In general those exceptions are grouped at the same relative priority.

Table 178. EIS asynchronous exception priorities

Relative priority	Exception	Interrupt level(1)	Interrupt nature	Pre/post completion (2)	Comments
0	Machine check	Machine check	Asynch/synch	pre for synch	Asynchronous exceptions may come from processor or from an external source.
1	Debug—UDE	Critical/debug	Asynch	N/A	Generally used for an externally generated high priority attention signal.
	Debug—IDE	Critical/debug	Asynch	N/A	Usually taken after MSR[DE] goes from 0 to 1 via **rfdi**/**rfci** or **mtmsr**.
	Debug—interrupt taken	Critical/debug	Asynch	N/A	Debug interrupt taken after original interrupt changed NIA and MSR.
	Debug—critical interrupt taken	Debug	Asynch	N/A	Debug interrupt taken after original critical interrupt has changed NIA and MSR.
2	Critical input	Critical	Asynch	N/A	
3	Watchdog	Critical	Asynch	N/A	
4	External input	Base	Asynch	N/A	
18	Fixed interval timer	Base	Asynch	N/A	
19	Decrementer	Base	Asynch	N/A	
20	Performance Monitor	Base	Asynch	N/A	

1. The interrupt level defines the set of save/restore registers used when the interrupt is taken—base (SRR0/SRR1), critical (CSRR0/CSRR1), debug (DSRR0/DSRR1), and machine check (MCSRR0/MCSRR1).
2. Pre- or post-completion refers to whether the exception occurs before an instruction completes (pre) and the corresponding interrupt points to the instruction causing the exception, or if the instruction completes (post) and the corresponding interrupt points to the next instruction to be executed.

Table 179. EIS synchronous exception priorities

Relative priority	Exception	Interrupt level(1)	Interrupt nature	Pre/post completion (2)	Comments
5	Debug–instruction address compare	Critical/debug	Synch	pre	
6	ITLB	Base	Synch	pre	
	ISI	Base	Synch	pre	
7	FP unavailable	Base	Synch	pre	
	AltiVec unavailable	Base	Synch	pre	Defined by the AltiVec APU.
	SPE unavailable	Base	Synch	pre	Defined by the SPE APU.
	Embedded floating-point unavailable	Base	Synch	pre	Defined by the embedded floating point APUs.
8	Debug—trap	Critical/debug	Synch	pre	
9	Program—illegal instruction	Base	Synch	pre	
	Program—unimplemented operation	Base	Synch	pre	
	Program—privileged instruction	Base	Synch	pre	
	Program—Trap	Base	Synch	pre	
	Program—FP enabled	Base	Synch	pre	An FP enabled interrupt may be imprecise.
10	(Alignment)	Base	Synch	pre	Alignment may be handled at either priority.
11	DTLB	Base	Synch	pre	
	Data storage	Base	Synch	pre	
12	Alignment	Base	Synch	pre	Alignment may be handled at either priority.

Table 179. EIS synchronous exception priorities (continued)

Relative priority	Exception	Interrupt level(1)	Interrupt nature	Pre/post completion (2)	Comments
13	System call	Base	Synch	post	Points SRR0 to instruction after **sc** (post completion).
	Embedded FP data	Base	Synch	pre	Defined by the SPE APU.
	Embedded FP round	Base	Synch	post	Points SRR0 to the instruction after the one causing the exception (post completion). Defined by SPE APU.
	AltiVec Assist	Base	Synch	pre	Defined by the AltiVec APU.
14	Debug—return from interrupt	Critical/debug	Synch	pre	
	Debug—Return from critical interrupt	Debug	Synch	pre	Defined by the enhanced debug APU.
	Debug—branch taken	Critical/debug	Synch	pre	
15	Debug—DAC	Critical/debug	Synch	pre or post	Preferred method is pre-completion.
16	Debug—DVC	Critical/debug	Synch	pre or post	Preferred method is pre-completion.
17	Debug—instruction complete	Critical/debug	Synch	post	Points [CD]SRR0 to next instruction (post completion).

1. The interrupt level defines the set of save/restore registers used when the interrupt is taken—base (SRR0/SRR1), critical (CSRR0/CSRR1), debug (DSRR0/DSRR1), and machine check (MCSRR0/MCSRR1).
2. Pre- or post-completion refers to whether the exception occurs before an instruction completes (pre) and the corresponding interrupt points to the instruction causing the exception, or if the instruction completes (post) and the corresponding interrupt points to the next instruction to be executed.

5 Storage architecture

This chapter describes the cache and MMU portions of the Book E implementation standards (EIS). Note that not all features that are defined by the EIS storage architecture are supported on all ST EIS processors; consult the user documentation. This chapter is organized into three section:

- *Chapter 5.2: Memory and cache coherency*"
- *Chapter 5.3: Cache model*"
- *Chapter 5.4: Storage model*"

5.1 Overview

The Book E architecture memory and cache definitions support a wide variety of embedded implementations. To provide such flexibility, Book E defines many features in a very general way, leaving specific details up to the implementation. To ensure consistency among its Book E cores and devices, ST has defined more specific implementation standards. However, these standards still leave many details up to individual implementations. To provide context for those features, this chapter describes aspects of the memory hierarchy and the memory management model defined by Book E; it also describes the ST EIS.

Note: This chapter describes some features (in particular, registers) in a very general way that does not include some details that are important to the programmer. There are also small differences in how some features are defined here and how they are implemented. For implementation-specific details, see the user documentation.

Throughout this chapter, references to load instructions include cache management and other instructions that are stated in the instruction descriptions to be treated as a load, and references to store instructions include the cache management and other instructions that are treated as a store.

The following APUs, which are part of the EIS storage architecture, are defined in *Chapter 8: Storage-related APUs on page 848*":

- Cache line locking APU
- Cache way partitioning APU
- Direct cache flush APU

These APUs may be implemented independently of each other. They are defined together in a single specification because it is likely that an implementation will include more than one of these APUs.

5.2 Memory and cache coherency

The primary objective of a coherent memory system is to provide the same image of memory to all devices using the system. Coherency allows synchronization and cooperative use of shared resources. Otherwise, multiple copies of data corresponding to a memory location, some containing outdated values, could exist in a system, resulting in errors when the outdated values are used. Each memory-sharing device must follow rules for managing the state of its cache. This section describes the coherency mechanisms of the Book E architecture and the cache coherency protocols that the ST Book E devices support.

Unless specifically noted, the discussion of coherency in this section applies to the core complex data cache only. The instruction cache is not snooped for general coherency with other caches; however, it is snooped when the Instruction Cache Block Invalidate (**icbi**) instruction is executed by this processor or any processor in the system.

5.2.1 Memory/Cache access attributes

Some memory characteristics can be set on a page basis by using the WIMGE bits in the translation lookaside buffer (TLB) entries. These bits allow both uniprocessor and multiprocessor system designs to exploit numerous system-level performance optimizations. The WIMGE attributes control the following:

- Write-through (W bit)
- Caching-inhibited (I bit)
- Memory-coherency-required (M bit)
- Guarded (G bit)
- Endianness (E bit)

In addition to the WIMGE bits, the Book E MMU model defines the following attributes on a page basis:

- User-definable (U0, U1, U2, U3)

The EIS defines the following optional attributes, which are manipulated by software through MMU assist register 2 (MAS2):

- Alternate coherency mode (ACM). The ACM attribute, programmed through MAS2[ACM], allows an implementation to employ multiple coherency methods and 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 for a page (M = 1), the ACM attribute determines the coherency domain (or protocol) used. ACM values are implementation dependent.
- Variable length encoding (VLE). The VLE attribute, MAS2[VLE], identifies pages that contain instructions from the VLE instruction set. If VLE = 0, instructions fetched from the page are decoded and executed as PowerPC (and associated EIS APUs) instructions. If VLE = 1, instructions fetched from the page are decoded and executed as Power Embedded instructions.

Consult the user documentation to determine whether the EIS-defined attributes are implemented.

The WIMGE attributes are programmed by the operating system for each page. The W and I attributes control how the processor performing an access uses its own cache. The M attribute ensures that coherency is maintained for all copies of the addressed memory location. The G attribute prevents speculative loading from the addressed memory location. (An operation is said to be performed speculatively if, at the time that it is performed, it is not known to be required by the sequential execution model.) The E attribute defines the order in which the bytes that comprise a multiple-byte data object are stored in memory (big- or little-endian).

The WIMGE attributes occupy 5 bits in the TLB entries for page address translation. The operating system writes the WIMGE bits for each page into the TLB entries in system memory as it maps translations. For more information, see *TLB entries on page 319*.”

All combinations of these attributes are supported except those that simultaneously specify a region as write-through and caching-inhibited. Write-through and caching-inhibited attributes are mutually exclusive because the write-through attribute permits the data to be in the data cache while the caching-inhibited attribute does not.

Memory that is write-through or caching-inhibited is not intended for general-purpose programming. For example, **lwarx** and **stwcx.** instructions may cause the system DSI exception handler to be invoked if they specify a location in memory having either of these attributes. Some implementations take a data storage interrupt if the location is write-through but does not take the interrupt if the location is cache-inhibited. Note that, except that the guarded bit does not prevent instruction prefetches, the definitions of the WIMG bits are unchanged

Write-through attribute

A page marked W = 0 is considered to be write-back. If some store instructions executed by a given processor access locations in a block as write-through and other store instructions executed by the same processor access locations in that block as write-back, software must ensure that the block cannot be accessed by another processor or mechanism in the system.

A store to a write-through (W = 1) memory location is performed in main memory and may cause additional memory locations 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. A store to write-through memory cannot cause a block to be put in a modified state in the data cache.

Also, if a store instruction that accesses a block in a location marked as write-through is executed when the block is already considered to be modified in the data cache, the block may continue to be considered to be modified in the data cache even if the store causes all modified locations in the block to be written to main memory. In some processors, accesses caused by separate store instructions that specify locations in write-through memory may be combined into one access. This is called store-gathering. Such combining does not occur if the store instructions are separated by an **msync** or an **mbar**.

Caching-inhibited attribute

A load instruction that specifies a location in caching-inhibited (I = 1) memory is performed to main memory and may cause additional locations in main memory to be accessed unless the specified location is also guarded. An instruction fetch from caching-inhibited memory may cause additional words in main memory to be accessed. No copy of the accessed locations is placed into the caches.

In some processors, nonoverlapping accesses caused by separate load instructions that specify locations in caching-inhibited memory may be combined into one access, as may nonoverlapping accesses caused by separate store instructions to caching-inhibited memory (that is, store-gathering). Such combining does not occur if the load or store instructions are separated by an **msync** instruction, or by an **mbar** instruction if the memory is also guarded.

Memory-coherence-required attribute

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 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 memory.

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 sub-sequence 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. Although a block's size is implementation-dependent, it is usually larger than a word and is often the size of a cache block.

When memory coherence is not required (M = 0), the hardware need not enforce data coherence for memory accesses initiated by the processor. When memory coherence is required (M = 1), the hardware must enforce data coherence for memory accesses initiated by the processor. Hardware support for the memory-coherence-required attribute is optional for implementations that do not support multiprocessing.

Guarded attribute

When the guarded bit is set, the page is designated as guarded. This setting can be used to protect certain memory areas from read accesses made by the processor that are not dictated directly by the program. If areas of physical memory are not fully populated (in other words, there are holes in the physical memory map within this area), this setting can protect the system from undesired accesses caused by speculative (referred to as 'out of order' in the architecture specification, and described in *Definition of apeculative and out-of-order memory accesses on page 285*") load operations that could lead to the generation of the machine check exception. Also, the guarded bit can be used to prevent speculative load operations from occurring to certain peripheral devices that produce undesired results when accessed in this way.

Definition of apeculative and out-of-order memory accesses

In the architecture definition, the term 'out of order' replaced the term 'speculative' with respect to memory accesses to avoid a conflict between the word's meaning in the context of execution of instructions past unresolved branches. The architecture's use of out of order in this context could in turn be confused with the notion of loads and stores being reordered in a weakly ordered memory system.

In the context of memory accesses, this document uses the terms 'speculative' and 'out of order' as follows:

- Speculative memory access—An access to memory that occurs before it is known to be required by the sequential execution model.
- Out-of-order memory access—A memory access performed ahead of one that may have preceded it in the sequential model, such as is allowed by a weakly ordered memory model.

Performing operations speculatively

An operation is said to be nonspeculative if it is guaranteed to be required by the sequential execution model. Any other operation is said to be performed speculatively, which the architecture specification refers to as out of order.

Operations are performed speculatively by hardware 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 needed depends on whether control flow is diverted away from the instruction by an event such as an exception, branch, trap, system call, return from interrupt instruction, or anything else that changes the context in which the instruction is executed.

Typically, the hardware performs operations speculatively when it has resources that would otherwise be idle, so the operation incurs little or no cost. If subsequent events such as branches or exceptions indicate that the operation would not have been performed, the processor abandons any results of the operation except as described below.

Most operations can be performed speculatively, as long as the machine appears to follow the sequential execution model. Certain speculative operations are restricted, as follows:

- Stores—A store instruction cannot execute speculatively in a manner such that the alteration of the target location can be observed by other processors or mechanisms.
- Accessing guarded memory—The restrictions for this case are given in *Speculative accesses to guarded memory on page 286*."

No error of any kind other than a machine check exception may be reported due to an operation that is performed speculatively, until such time as it is known that the operation is required by the sequential execution model. The only other permitted side effect (other than machine check) of performing an operation speculatively is that nonguarded memory locations that could be fetched into a cache by nonspeculative execution may be fetched speculatively into that cache.

Guarded memory

Memory is said to be well behaved if the corresponding physical memory 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 speculatively from well-behaved memory without causing undesired side effects.

Memory is said to be guarded if the G bit is set for the page. In general, memory that is not well-behaved should be guarded. Because such memory may represent an I/O device or include nonexistent locations, a speculative access to such memory may cause an I/O device to perform incorrect operations or may cause a machine check.

Note that if separate store instructions access memory that is both caching-inhibited and guarded, the accesses are performed in the order specified by the program. If an aligned load or store that is not a string or multiple access to caching-inhibited, guarded memory has accessed main memory and an external, decrementer, or imprecise-mode floating-point enabled exception is pending, the load or store is completed before the exception is taken.

Speculative accesses to guarded memory

Accesses for load instructions from guarded memory may be performed speculatively if a copy of the target location is in a cache; in this case, the location may be accessed from the cache or from main memory.

Note that software should ensure that only well-behaved memory is loaded into a cache, either by marking as caching-inhibited (and guarded) all memory that may not be well-behaved or by marking such memory caching-allowed (and guarded) and referring only to cache blocks that are well-behaved.

Instrubction accesses: guarded memory and no-execute memory

The G bit is ignored for instruction fetches, and instructions are speculatively fetched from guarded pages. To prevent speculative fetches from pages that do not contain instructions and are not well-behaved, the page should be designated as no-execute (with the UX/SX page permission bits cleared). If the effective address of the current instruction is mapped to no-execute memory, an ISI exception is generated.

Endianness

Objects may be loaded from or stored to memory in byte, half-word, word, or double-word units. For a particular data length, the load and store operations are symmetrical; a store followed by a load of the same data object yields an unchanged value. Book E makes no guarantees about the order in which the bytes that comprise multiple-byte data objects are stored into memory. The endianness (E) page attribute distinguishes between memory that is big or little endian, as described in the following subsections.

Except for instruction fetches, it is always permitted to access the same location using two effective addresses with different E bit settings. Instruction pages must be flushed from any caches before the E bit can be changed for those addresses. See *Byte ordering on page 141*," for more information about endianness.

Big-endian pages

If a stored multiple-byte object is probed by reading its component bytes one at a time using load-byte instructions, the store order may be perceived. If such probing shows that the lowest memory address contains the highest-order byte of the multiple-byte scalar, the next-higher sequential address the next-least-significant byte, and so on, the multiple-byte object is stored in big-endian form. Big-endian memory is defined on a page basis by the memory/cache attribute, E = 0.

Note that strings are not multiple-byte scalars but are interpreted as a series of single-byte scalars. Bytes in a string are loaded from memory using a load string word instruction, starting at the lowest-numbered address, and placed into the target register or registers starting at the left-most byte of the least-significant word. Bytes in a string are stored using a store string word instruction from the source register, starting at the left-most byte of the least-significant word, and placed into memory, starting at the lowest-numbered address.

Little-endian pages

Alternatively, if the probing shows that the lowest memory address contains the lowest-order byte of the multiple-byte scalar, the next-higher sequential address the next-most-significant byte, and so on, the multiple-byte object is stored in little-endian form. Little-endian memory is defined on a page basis by the memory/cache attribute, E = 1, and for Book E devices is defined as true little-endian memory.

Structure mapping examples

The following C programming example defines the data structure *S* used in this section to demonstrate how the bytes that comprise each element (a, b, c, d, e, and f) are mapped into memory. The structure contains scalars (shown in hexadecimal in the comments) and a sequence of characters, shown in single quotation marks.

```
struct {
        int       a;      /* 0x1112_1314                word*/
        double    b;      /* 0x2122_2324_2526_2728double word*/
        char *    c;      /* 0x3132_3334                word*/
        char      d[7];   /* 'L','M','N','O','P','Q','R' array of bytes*/
```

```
        short   e;      /* 0x5152          half word*/
        int     f;      /* 0x6162_6364     word*/
} S;
```

big-endian mapping of the structure ia shown below.

Big-endian mapping of structure

Contents	11	12	13	14	(x)	(x)	(x)	(x)
Address	00	01	02	03	04	05	06	07
Contents	21	22	23	24	25	26	27	28
Address	08	09	0A	0B	0C	0D	0E	0F
Contents	31	32	33	34	‘L’	‘M’	‘N’	‘O’
Address	10	11	12	13	14	15	16	17
Contents	‘P’	‘Q’	‘R’	(x)	51	52	(x)	(x)
Address	18	19	1A	1B	1C	1D	1E	1F
Contents	61	62	63	64	(x)	(x)	(x)	(x)
Address	20	21	22	23	24	25	26	27

Note that the MSB of each scalar is at the lowest address. The mapping uses padding (indicated by (x)) to align the scalars—4 bytes between elements a and b, 1 byte between d and e, and 2 bytes between e and f. Note that the padding is determined by the compiler, not the architecture.

The structure using little-endian mapping, showing double words laid out with addresses increasing from right to left.

Little-endian mapping of structure *S*—alternate view

Contents	(x)	(x)	(x)	(x)	11	12	13	14
Address	07	06	05	04	03	02	01	00
Contents	21	22	23	24	25	26	27	28
Address	0F	0E	0D	0C	0B	0A	09	08
Contents	‘O’	‘N’	‘M’	‘L’	31	32	33	34
Address	17	16	15	14	13	12	11	10
Contents	(x)	(x)	51	52	(x)	‘R’	‘Q’	‘P’
Address	1F	1E	1D	1C	1B	1A	19	18
Contents	(x)	(x)	(x)	(x)	61	62	63	64
Address	27	26	25	24	23	22	21	20

Mismatched memory cache attributes

Accesses to the same memory location using two effective addresses for which the write-through required attribute (W bit) differs meet the memory coherence requirements described in *Write-through attribute on page 284*," 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 write-through attribute is the same for all the accesses.

Loads, stores, **dcbz** instructions, and instruction fetches to the same memory location using two effective addresses for which the caching-inhibited attribute (I bit) differs must meet the requirement that a copy of the target location of an access to caching-inhibited memory not be in the cache. Violation of this requirement is considered a programming error; software must ensure that the location has not previously been brought into the cache or, if it has, that it has been flushed from the cache. If the programming error occurs, the result of the access is boundedly undefined. It is not considered a programming error if the target location of any other cache management instruction to caching-inhibited memory is in the cache.

Accesses to the same memory location using two effective addresses for which the memory coherence attribute (M bit) differs may require explicit software synchronization before accessing the location with M = 1 if the location has previously been accessed with M = 0. Any such requirement is system-dependent. For example, in some systems that use bus snooping, no software synchronization may be required. In some directory-based systems, software may be required to execute **dcbf** instructions on each processor to flush all cache entries accessed with M = 0 before accessing those locations with M = 1.

Accesses to the same memory location using two effective addresses for which the guarded attribute (G bit) differs are always permitted.

Except for instruction fetches, accesses to the same memory location using two effective addresses for which the endian storage attribute (E bit) differs are always permitted as described in *Endianness on page 287*." Instruction memory locations must be flushed before the endian attribute can be changed for those addresses.

The requirements on mismatched user-defined memory attributes (U0–U3) is implementation-dependent.

Coherency paradoxes and WIMGE

Care must be taken with respect to the use of the WIMGE bits if coherent memory support is desired. Careless programming of these bits may create situations that present coherency paradoxes to the processor. These paradoxes can occur within a single processor or across several processors. It is important to note that, in the presence of a paradox, the operating system software is responsible for correctness.

In particular, a coherency paradox can occur when the state of these bits is changed without appropriate precautions (such as flushing the pages that correspond to the changed bits from the caches of all processors in the system) or when the address translations of aliased real addresses specify different values for certain WIMGE bit values. For more information, see *Mismatched memory cache attributes on page 289*."

Support for M = 1 memory is optional. Cache attribute settings where both W = 1 and I = 1 are not supported. For all supported combinations of the W, I, and M bits, both G and E may be 0 or 1.

The default setting of the WIMGE bits is 0b01000.

Self-modifying code

When a processor modifies any memory location that can contain an instruction, software must ensure that the instruction cache is made consistent with data memory and that the modifications are made visible to the instruction fetching mechanism. This must be done even if the cache is disabled or if the page is marked caching-inhibited.

The following instruction sequence can be used to accomplish this when the instructions being modified are in memory that is memory-coherence required and one processor both modifies the instructions and executes them. (Additional synchronization is needed when one processor modifies instructions that another will execute.)

The following sequence synchronizes the instruction stream (using either **dcbst** or **dcbf**):

```
dcbst (or dcbf)|update memory
msync          |wait for update
icbi           |remove (invalidate) copy from instruction cache
msync          |ensure that ICBI invalidation at icache has completed
isync          |remove copy in own instruction buffer
```

5.2.2 Shared memory

The architecture supports sharing memory between programs, between different instances of the same program, and between processors and other mechanisms. It also supports access to a memory location by one or more programs using different effective addresses. In these cases, memory is shared in blocks that are an integral number of pages. When one physical memory location has different effective addresses, the addresses are said to be aliases. Each application can be granted separate access privileges to aliased pages.

Lock acquisition and import barriers on page 294,” gives examples of how **msync** and **mbar** are used to control memory access ordering when memory is shared among programs.

Memory access ordering

The memory model in Book E for memory access ordering is weakly consistent. This provides an opportunity for improved performance over a model with stronger consistency rules but places the responsibility on the program to ensure that ordering or synchronization instructions are properly placed for correct execution of the program.

The order in which a processor accesses memory, the order in which those accesses are performed with respect to other processors or mechanisms, and the order in which they are performed in main memory may all be different. *Table 180* describes how the architecture defines requirements for ordering of loads and stores.

Table 180. Load and store ordering

Type of Access	Architecture definition
Load ordering with respect to other loads	The architecture guarantees that loads that are both caching-inhibited (I = 1) and guarded (G = 1) are not reordered with respect to one another.
	If a load instruction depends on the value returned by a preceding load (because the value is used to compute the effective address specified by the second load), the corresponding memory accesses are performed in program order with respect to any processor or mechanism to the extent required by the associated memory coherence required attributes (that is, the memory coherence required attribute, if any, associated with each access). This applies even if the dependency does not affect program logic (for example, the value returned by the first load is ANDed with zero and then added to the effective address specified by the second load).
Store ordering with respect to other stores	If two store instructions specify memory locations that are both caching inhibited and guarded, the corresponding memory accesses are performed in program order with respect to any processor or mechanism. Otherwise, stores are weakly ordered with respect to one another.
Store ordering with respect to loads	The architecture specifies that an **msync** or **mbar** must be used to ensure sequential ordering of loads with respect to stores.

When a processor (P1) executes **msync** or **mbar**, a memory barrier is created that separates applicable memory accesses into two groups, G1 and G2. G1 includes all applicable memory accesses associated with instructions preceding the barrier-creating instruction, and G2 includes all applicable memory accesses associated with instructions following the barrier-creating instruction.

Table shows an example using a two-processor system.

Table 181. Memory barrier when coherency is required (M = 1)

<table>
<tr><th>Processor 1 (P1)</th><th>Memory access groups G1 and G2</th><th>Processor 2 (P2)</th></tr>
<tr><td>Instruction 1</td><td rowspan="4">G1: Memory accesses generated by P1 before the memory barrier</td><td rowspan="4">When memory coherence is required, G1 accesses that affect P2 are also performed before the memory barrier.</td></tr>
<tr><td>Instruction 2</td></tr>
<tr><td>Instruction 3</td></tr>
<tr><td>Instruction 4</td></tr>
<tr><td colspan="2">Instruction 5 (msync or mbar)—Memory barrier</td><td>Barrier generated by P1 does not order P2 instructions or associated accesses with respect to other P2 instructions and associated accesses.</td></tr>
<tr><td>Instruction 6</td><td rowspan="5">G2: Memory accesses generated by P1 after the memory barrier</td><td rowspan="5">When memory coherence is required, G2 accesses that affect P2 are also performed after the memory barrier.</td></tr>
<tr><td>Instruction 7</td></tr>
<tr><td>Instruction 8</td></tr>
<tr><td>Instruction 9</td></tr>
<tr><td>Instruction 10</td></tr>
</table>

The memory barrier ensures that all memory accesses in G1 are performed with respect to any processor or mechanism, to the extent required by the associated memory coherence required attributes (that is, the memory-coherence required attribute, if any, associated with each access), before any memory accesses in G2 are performed with respect to that processor or mechanism.

The ordering enforced by a memory barrier is said to be cumulative if it also orders memory accesses that are performed by processors and mechanisms other than P1, as follows:

- G1 includes all applicable memory accesses by any such processor or mechanism that have been performed with respect to P1 before the memory barrier is created.
- G2 includes all applicable memory 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 G2.

Table 182 shows an example of a cumulative memory barrier in a two-processor system.

Table 182. Cumulative memory barrier

<table>
<tr><th>Processor 1 (P1)</th><th>Memory access groups G1 and G2</th><th>Processor 2 (P2)</th></tr>
<tr><td>P1 Instruction 1</td><td rowspan="4">G1: Memory accesses generated by P1 and P2 that affect P1. Includes accesses generated by executing P2 instructions L–O (assuming that the access generated by instruction O occurs before P1's msync is executed).</td><td>P2 Instruction L</td></tr>
<tr><td>P1 Instruction 2</td><td>P2 Instruction M</td></tr>
<tr><td>P1 Instruction 3</td><td>P2 Instruction N</td></tr>
<tr><td>P1 Instruction 4</td><td>P2 Instruction O</td></tr>
<tr><td colspan="2" rowspan="3">P1 Instruction 5 (msync)—Cumulative memory barrier applies to all accesses except those associated with fetching instructions following msync.</td><td>P2 Instruction P</td></tr>
<tr><td>P2 Instruction Q</td></tr>
<tr><td>P2 Instruction R</td></tr>
<tr><td>P1 Instruction 6</td><td rowspan="6">G2: Memory accesses generated by P1 and P2. Includes accesses generated by P2 instructions P–X (assuming that the access generated by instruction P occurs after P1's msync is executed) performed after a load instruction executed by P2 has returned the value stored by a store that is in G2.
The msync memory barrier does not affect accesses associated with instruction fetching that occur after the msync.</td><td>P2 Instruction S</td></tr>
<tr><td>P1 Instruction 7</td><td>P2 Instruction T</td></tr>
<tr><td>P1 Instruction 8</td><td>P2 Instruction U</td></tr>
<tr><td>P1 Instruction 9</td><td>P2 Instruction V</td></tr>
<tr><td>P1 Instruction 10</td><td>P2 Instruction W</td></tr>
<tr><td>P1 Instruction 11</td><td>P2 Instruction X</td></tr>
</table>

A memory barrier created by **msync** is cumulative and applies to all accesses except those associated with fetching instructions following the **msync**. See the definition of **mbar** in *Memory synchronization instructions on page 175*," for a description of the corresponding properties of the memory barrier created by that instruction.

Programming considerations

Because stores cannot be performed out of program order, as described in Book E, if a store instruction depends on the value returned by a preceding load (because the value the load returns is needed to compute either the effective address specified by the store or the value to be stored), the corresponding accesses are guaranteed to be performed in program order. The same applies whether or not the store instruction executes is dependent upon a conditional branch that in turn depends on the value returned by a preceding load. For example, if a conditional branch depends on a preceding load and that branch chooses

between a path that includes a store instruction if the condition is met, that dependent store is not performed unless and until the condition determined by the load is met.

Because instructions following an **isync** cannot execute until all instructions preceding **isync** have completed, if an **isync** follows a conditional branch instruction that depends on the value returned by a preceding load instruction, that load is performed before any loads caused by instructions following the **isync**. This is true even if the effects of the dependency are independent of the value loaded (for example, the value is compared to itself and the branch tests CR*n*[EQ]), and even if the branch target is the next sequential instruction.

Except for the cases described above and earlier in this section, data and control dependencies do not order memory accesses. Examples include the following:

- If a load specifies the same memory location as a preceding store and the location 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 memory subsystem) and not be visible to other processors and mechanisms. As a result, 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 is performed, a conditional branch instruction that depends on the CR0 value set by a store conditional instruction does not order that store with respect to memory accesses caused by instructions that follow the branch.

  For example, in the following sequence, the **stw** is the **bc** instruction's target:

  stwcx.
  bc
  stw

  To complete, the **stwcx.** must update the architected CR0 value, even though its store may not have been performed. The architecture does not require that the store generated by the **stwcx.** must be performed before the store generated by the **stw**.
- Because processors may predict branch target addresses and branch condition resolution, control dependencies (branches, for example) do not order memory 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.

Some processors implement nonarchitected duplicates of architected resources such as GPRs, CR fields, and the LR, so resource dependencies (for example, specification of the same target register for two load instructions) do not force ordering of memory accesses.

Examples of correct uses of dependencies, **msync**, and **mbar** to order memory accesses can be found in hi."

Because the memory model is weakly consistent, the sequential execution model as applied to instructions that cause memory 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 memory accesses caused by the first instruction have been performed. However, for a sequence of atomic accesses to the same memory location for which memory coherence is 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 one for which caching is inhibited.

Because caching-inhibited memory accesses are performed in main memory, memory barriers and dependencies on load instructions order such accesses with respect to any processor or mechanism even if the memory is not marked as requiring memory coherence.

Programming examples

Example 1 shows cumulative ordering of memory accesses preceding a memory barrier, Example 2 shows cumulative ordering of memory accesses following a memory barrier. In both examples, assume that locations X, Y, and Z initially contain the value 0. In both, cumulative ordering dictates that the value loaded from location X by processor C is 1.

E**xample 1:**

- Processor A stores the value 1 to location X.
- Processor B loads from location X obtaining the value 1, executes an **msync**, then stores the value 2 to location Y.
- Processor C loads from location Y obtaining the value 2, executes an **msync**, then loads from location X.

Example 2:

- Processor A stores the value 1 to location X, executes an **msync**, 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 an **msync**, then loads from location X.

Lock acquisition and import barriers

An import barrier is an instruction or instruction sequence that prevents memory accesses caused by instructions following the barrier from being performed before memory 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. An **msync** can always be used as an import barrier, but the approaches shown below generally yield better performance because they order only the relevant memory accesses.

Acquire lock and import shared memory

If **lwarx** and **stwcx.** are used to obtain the lock, an import barrier can be constructed by placing an **isync** immediately following the loop containing the **lwarx** and **stwcx.**. The following example uses the compare and swap primitive (see *Chapter C.1.1: Synchronization primitives on page 1144*") to acquire the lock.

This example assumes 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:lwarxr6,0,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 is resolved to be not taken.

Obtain pointer and import shared memory

If **lwarx** and **stwcx.** 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 (see *Section C.1.1: Synchronization primitives*") 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 **lwarx** loads the pointer value into GPR 5. The load from data1 may be performed out of order 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 out of order, 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** 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**.

Atomic memory references

The Book E architecture defines the Load Word and Reserve Indexed (**lwarx**) and the store word conditional indexed (**stwcx.**) instructions to provide an atomic update function for a single, aligned word of memory. These instructions can be used to develop a rich set of multiprocessor synchronization primitives. Note that atomic memory references constructed using **lwarx**/**stwcx.** instructions depend on the presence of a coherent memory system for correct operation. These instructions should not be expected to provide atomic access to noncoherent memory.

The **lwarx** instruction performs a load word from memory operation and creates a reservation for the same reservation granule that contains the accessed word. Reservation granularity is implementation-dependent.

The **lwarx** instruction makes a nonspecific reservation with respect to the executing processor and a specific reservation with respect to other masters. This means that any subsequent **stwcx.** executed by the same processor, regardless of address, cancels the reservation. Also, any bus write or invalidate operation from another processor to an address that matches the reservation address cancels the reservation.

5.3 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 Book E, for example in the descriptions of the cache management instructions in *Chapter 3: Instruction model on page 133*." Book E allows the following additional cache models are defined by the EIS:

- Unified cache, in which a cache is shared by both instructions and data
- Multi-level caches, which must support the programming model implied by a Harvard-style cache.

A processor is not required to maintain copies of storage locations in the instruction cache that are consistent with modifications to those storage locations (that is, modifications by store instructions).

In general, a location in the data cache is considered to be modified in that cache if the location has been modified (for example, by a store instruction) and the modified data has not been written to main storage. The only exception to this rule is described in *Write-through attribute on page 284*."

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). 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 attributes associated with the specified storage location.

Cache management instructions allow the program to do the following.

- Give a hint that a block of storage should be copied to the instruction cache, so that the copy of the block is more likely to be in the cache when subsequent accesses to the block occur, thereby reducing delays (**icbt**)
- Invalidate the copy of storage in an instruction cache block (**icbi**)
- Discard prefetched instructions (**isync**)
- Invalidate the copy of storage in a data cache block (**dcbi**)
- Give a hint that a block of storage should be copied to the data cache, so that the copy of the block is more likely to be in the cache when subsequent accesses to the block occur, thereby reducing delays (**dcbt**, **dcbtst**)
- Allocate a data cache block and set the contents of that block to zeros, but no-operation if no access is allowed to the data cache block and do not cause any exceptions (**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**).

5.3.1 Cache programming model

This section summarizes the register and instructions defined to support the cache model. Full descriptions of these resources are provided in *Chapter 2: Register model on page 46*, and *Chapter 3: Instruction model on page 133*.

Cache model registers

The EIS cache model implements the following registers and register fields:

- Machine state register (MSR). Defines the processor state (that is, enabling and disabling of interrupts and debugging exceptions, enabling and disabling of address translation for instruction and data memory accesses, enabling and disabling some APUs, and specifying whether the processor is in supervisor or user mode). EIS storage defines the user cache locking enable bit (MSR[UCLE]) as part of the cache line locking APU.

  Book E and the EIS define the MSR fields described in *Table 183*. The MSR is described in detail in *Chapter 2.6.1: Machine state register (MSR) on page 68*."

Table 183. Storage related MSR fields

Bits	Name	Description
37	UCLE	(EIS-defined) User-mode cache lock enable. Used to restrict user-mode cache-line locking by the operating system. 0Any cache lock instruction executed in user-mode takes a cache-locking exception and data storage interrupt and sets either ESR[DLK] or ESR[ILK]. This allows the operating system to manage and track the locking/unlocking of cache lines by user-mode tasks. 1Cache-locking instructions can be executed in user-mode and they do not take a DSI for cache-locking (they may still take a DSI for access violations though).
58	IS	(Book E–defined) Instruction address space 0The processor directs all instruction fetches to address space 0 (TS = 0 in the relevant TLB entry). 1The processor directs all instruction fetches to address space 1 (TS = 1 in the relevant TLB entry).
59	DS	(Book E–defined) Data address space 0The processor directs data memory accesses to address space 0 (TS = 0 in the relevant TLB entry). 1The processor directs data memory accesses to address space 1 (TS = 1 in the relevant TLB entry).

- Exception syndrome register (ESR). The ESR provides a syndrome to differentiate between different kinds of exceptions that can generate the same interrupt type. When such an interrupt is generated, bits corresponding to the exception that generated the interrupt are set and all other ESR bits are cleared. Other interrupt types do not affect ESR contents. The ESR does not need to be cleared by software.
- Book E and the EIS defines the storage-related ESR fields described in *Table 184*. The ESR is described in detail in *Exception syndrome register (ESR) on page 84*."

Table 184. Exception syndrome register (ESR) definition

Bits	Name	Syndrome	Interrupt types
39	FP	(Book E–defined) Floating-point operations	Alignment, data storage, data TLB, program
40	ST	(Book E–defined) Store operation	Alignment, data storage, data TLB error
42	DLK	Defined by cache line locking APU. Instruction cache locking attempt. Set when a DSI occurs because a **dcbtls**, **dcbtstls**, or **dcblc** was executed in user mode (MSR[PR] = 1) while MSR[UCLE] = 0. 0Default 1DSI occurred on an attempt to lock line in data cache when MSR[UCLE] = 0.	Data storage
43	ILK	Defined by cache line locking APU. Instruction cache locking attempt. Set when a DSI occurs because an **icbtls** or **icblc** was executed in user mode (MSR[PR] = 1) while MSR[UCLE] = 0. 0Default 1DSI occurred on an attempt to lock line in instruction cache when MSR[UCLE] = 0.	Data storage
44	APU	(Book E–defined) Auxiliary processor operation.	Alignment, data storage, data TLB, program
46	BO	Byte-ordering exception. Defined by Book E and the VLE extension.	Data storage, instruction storage

Table 184. Exception syndrome register (ESR) definition (continued)

Bits	Name	Syndrome	Interrupt types
56	SPE	Defined by SPE, embedded floating-point APU. SPE/embedded floating-point exception bit 0 Default 1 Any exception caused by an SPE/embedded floating-point instruction occurred.	Data storage, Data TLB error, Alignment, SPE unavailable, Embedded FP unavailable, Embedded FP data, Embedded FP round
58	VLEMI	Defined by VLE extension. VLEMI indicates that an interrupt was caused by a VLE instruction. VLEMI is set on an exception associated with execution or attempted execution of a VLE instruction. 0 The instruction page associated with the instruction causing the exception does not have the VLE attribute set or the VLE extension is not implemented. 1 The instruction page associated with the instruction causing the exception has the VLE attribute set and the VLE extension is implemented.	Data storage, Data TLB error, Instruction storage, Program, System Call, Alignment, SPE unavailable, Embedded FP unavailable, Embedded FP data, Embedded FP round
62	MIF	Defined by the VLE extension. MIF indicates that an interrupt was caused by a misaligned instruction fetch (NIA_{62} != 0) and the VLE attribute is cleared for the page or the second half of a 32-bit VLE instruction caused an instruction TLB error. 0Default. 1NIA_{62} != 0 and the instruction page associated with NIA does not have the VLE attribute set or the second half of a 32-bit VLE instruction caused an instruction TLB error.	Instruction TLB error, Instruction Storage
63	XTE	External transaction error. An external transaction reported an error but the error was handled precisely by the core. SRR0 holds the address of the instruction that initiated the transaction. 0Default. No external transaction error was precisely detected. 1An external transaction reported an error that was precisely detected.	Instruction storage, Data storage

- L1 cache control and status registers (L1CSR0–L1CSR1).
  - L1CSR0 provides general control and status for the processor's primary data cache. If a processor implements a unified L1 cache, L1CSR0 applies to the unified cache and L1CSR1 is not implemented. See *Chapter 2.11.1: L1 cache control and status register 0 (L1CSR0) on page 90*."
  - L1CSR1 provides general control and status for the processor's primary instruction cache. If a processor implements a unified L1 cache, L1CSR0 applies

to the unified cache and L1CSR1 is not implemented. See *Chapter 2.11.2: L1 cache control and status register 1 (L1CSR1) on page 92*."

- L1 cache configuration registers (L1CFG0)
  - L1CFG0 provides configuration information for the processor's primary data cache. If a processor implements a unified cache, L1CFG0 applies to the unified cache and L1CFG1 is not implemented. See *Chapter 2.11.3: L1 cache configuration register 0 (L1CFG0) on page 94*."
  - L1CFG1 provides configuration information for the processor's primary instruction cache. If a processor implements a unified cache, L1CFG0 applies to the unified cache and L1CFG1 is not implemented. L1CFG1 allows software to identify the organization and capabilities of the primary instruction cache. *Chapter 2.11.4: L1 cache configuration register 1 (L1CFG1) on page 95*."

Cache model instructions

The Book E PowerPC architecture defines instructions for controlling both the instruction and data caches (when they exist).

- Data Cache Block Touch (**dcbt**)
- Data Cache Block Touch for Store (**dcbtst**)
- Data Cache Block Zero (**dcbz**)
- Data Cache Block Store (**dcbst**)
- Data Cache Block Flush (**dcbf**)
- Data Cache Block Allocate (**dcba**)
- Data Cache Block Invalidate (**dcbi**)
- Instruction Cache Block Invalidate (**icbi**)
- Instruction Synchronize (**isync**)
- Instruction Cache Block Touch (**icbt**)

These instructions are described in *User-level cache instructions on page 180*," and *Supervisor-level cache instruction on page 183*." Note that the behavior of many of these instructions is determined by the value of the cache target operand (CT). See *CT instruction field on page 301*." *Permission control and cache management instructions on page 316*," describes conditions in which cache control instructions can generate protection violations.

The cache block locking APU, defined by the EIS, adds the following instructions:

- Data Cache Block Lock Clear (**dcblc**)
- Data Cache Block Touch and Lock Set (**dcbtls**)
- Data Cache Block Touch for Store and Lock Set (**dcbtstls**)
- Instruction Cache Block Lock Clear (**icblc**)
- Instruction Cache Block Touch and Lock Set (**icbtls**)

These instructions are described in *Chapter 8.1: Cache line locking APU on page 848*."

CT instruction field

Instructions having a CT (cache target) field for specifying a cache hierarchy use the value 0 to specify the primary cache. ST devices interpret this operand as follows:

- CT = 0 indicates the L1 cache.
- CT = 1 indicates the I/O cache. (Note that some versions of the e500 documentation refer to the I/O cache as a frontside L2 cache.)
- CT = 2 indicates a backside L2 cache.

5.3.2 Primary (L1) cache model

This section describes the L1 cache model defined by the EIS.

Types

Primary caches may separate instruction and data caches into two separate structures (commonly known as Harvard architecture), or they may provide a unified cache combining instructions and data. Caches are physically tagged.

Storage attributes and coherency

Primary data caches must support the storage attributes defined by Book E with the following advisory:

Note: The primary data cache may be implemented not to snoop (that is, not coherent with transactions outside the processor). System software is then responsible to maintain coherency. Thus the setting of the M attribute is meaningless. The preferred implementation provides snooping for primary data caches.

Primary instruction caches must support the storage attributes defined by Book E with the following advisory:

- The guarded attribute should be ignored for instruction fetch accesses. To prevent speculative fetch accesses to guarded memory, software should mark those pages as no-execute.
- The cache may be implemented not to snoop (that is, not coherent with transactions outside the processor). System software is then responsible to maintain coherency. The preferred implementation does not provide snooping for primary instruction caches.

As with other memory-related instructions, the effects of cache management instructions on memory are weakly-ordered. If the programmer must ensure that cache or other instructions have been performed with respect to all other processors and system mechanisms, an **msync** must be placed after those instructions.

5.4 Storage model

This section describes the storage model as it is defined by Book E and by the EIS.

5.4.1 Storage programming model

This section summarizes the register and instructions defined to support the cache model. Full descriptions of these resources are provided in *Chapter 2: Register model on page 46*," and *Chapter 3: Instruction model on page 133*."

Storage model registers

This section provides an overview of the registers used for programming the MMU. Full descriptions are provided in *Chapter 2.12: MMU registers on page 97*." These registers consist of the following:

- Process ID registers (PID0–PID2) are used by system software to identify TLB entries that are used by the processor to accomplish address translation for loads, stores, and instruction fetches. Book E defines one PID register (PID synonymous with PID0). The EIS defines 14 additional PID registers, PID1 through PID14. A implementation may choose to provide any number of PIDs up to a maximum of 15. 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 (see *Chapter 5.4.6*").
- MMU assist registers (MAS0–MAS7) are used to transfer data to and from the TLB arrays. Software uses **mfspr** and **mtspr** to read and write MAS registers. Executing **tlbre** causes the TLB entry specified by MAS0[TLBSEL,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,ESEL] and MAS2[EPN] to be written with the MAS register contents. Hardware can also updated MAS registers on the occurrence of an instruction or data TLB error interrupt or as the result of a **tlbsx**.

  All MAS registers are supervisor level, and all except MAS5 and MAS7 must be implemented. MAS7 is not required if the processor supports 32 bits or less of physical address. Implementing MAS5 is implementation dependent.

  Processors are required to implement only the necessary bits of any multiple-bit MAS register field 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 two TLB arrays would likely implement only the lower-order MAS0[TLBSEL] bits.

  - MAS0, contains fields for identifying and selecting a TLB entry.
  - MAS1, contains fields for selecting a TLB entry during translation.
  - MAS2, contains fields for specifying the effective page address and the storage attributes for a TLB entry.
  - MAS3, contains fields for specifying the real page address and the permission attributes for a TLB entry.
  - MAS4, contains fields for specifying default information to be pre-loaded on certain MMU-related exceptions.
  - The optional MAS5 register, contains fields for specifying PID values to be used when searching TLB entries with the **tlbsx** instruction.
  - MAS6, contains fields for specifying PID and AS values used when the **tlbsx** instruction is used to search TLB entries.

  MAS7, contains the high-order address bits of the RPN for implementations that support more than 32 bits of physical address. Implementations that support 32 bits or fewer do not implement MAS7.

- MMU configuration register (MMUCFG), provides configuration information about the MMU.
- TLB configuration registers (TLB*n*CFG). One TLB*n*CFG register, is implemented to provide information about each TLB implemented. TLB0CFG corresponds to TLB0, TLB1CFG corresponds to TLB1, etc.
- MMU control and status register (MMUCSR0), is used for general control of the MMU including flash invalidation of the TLB arrays and page sizes for programmable fixed size arrays. For TLB arrays with programmable fixed sizes, the TLB*n*_PS fields allow software to specify the page size.

Storage model instructions

The address translation mechanism is defined in terms of TLBs and page table entries (PTEs) Book E processors use to locate the logical-to-physical address mapping for a particular access. *Table 103 on page 184* describes the operation of the TLB instructions, which are summarized as follows:

- TLB Invalidate Virtual Address Indexed (**tlbivax**)
- TLB Read Entry (**tlbre**)
- TLB Search Indexed (**tlbsx**)
- TLB Synchronize (**tlbsync**)
- TLB Write Entry (**tlbwe**)

5.4.2 The storage architecture

This section describes the storage model as it is defined by Book E and by the ST EIS.

Book E storage architecture

The memory management approach defined by the Book E EIS is suited for desktop applications and has the simplicity and flexibility necessary for embedded applications. Book E supports demand-paged virtual memory as well as a variety of other management schemes that depend on precise control of effective-to-real address translation and flexible

memory protection. Address translation misses and protection faults cause precise exceptions. Sufficient information is available to correct the fault and restart the faulting instruction.

Each program on a 32-bit implementation can access 2^{32} bytes of effective address (EA) space, subject to limitations imposed by the operating system. In a typical Book E system, each program's EA space is a subset of a larger virtual address (VA) space managed by the operating system.

Each effective (logical) address is translated to a real (physical) address before being used to access physical memory or an I/O device. Hardware does this by using the address translation mechanism described in *Chapter 5.4.6*." The operating system manages the physically addressed resources of the system by setting up the tables used by the address translation mechanism.

The Book E architecture divides the effective address space into pages. The page represents the granularity of effective address translation, permission control, and memory/cache attributes. Up to 12 page sizes (1, 4, 16, 64, or 256 Kbytes; 1, 4, 16, 64, or 256 Mbytes; or 1 Gbyte) may be simultaneously supported. For an effective-to-real address translation to exist, a valid entry for the page containing the effective address must be in a translation lookaside buffer (TLB). Addresses for which no TLB entry exists cause TLB miss exceptions (instruction or data TLB error interrupts).

The instruction addresses generated by a program and the addresses used by load, store, and cache management instructions are effective addresses. However, in general, the physical memory space 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 enough virtual pages for an application. If a sufficient set is maintained, paging activity is minimized, therefore maximizing performance.

The operating system can restrict access to virtual pages by selectively granting permissions for user-state read, write, and execute, and supervisor-state read, write, and execute on a per-page basis. These permissions can be set up for a particular system (for example, program code might be execute-only, data structures may be mapped as read/write/no-execute) and can also be changed by the operating system based on application requests and operating system policies.

EIS storage architecture

The standard for Book E MMUs establishes a common way of implementing Book E processors to provide a programming model that is consistent across all products in the family. Having a standard reduces the software efforts required in porting to a new processor because the common programming model minimizes implementation differences. Thus, the standard defines configuration information for features such as TLBs, caches, and other entities that have standard forms, but differing attributes (like cache sizes and associativity) such that a single software implementation can be created that works efficiently for all implementations of a class.

The Book E MMU standard defines functions and structures that are visible to the execution model of the processor. These consist of the following definitions:

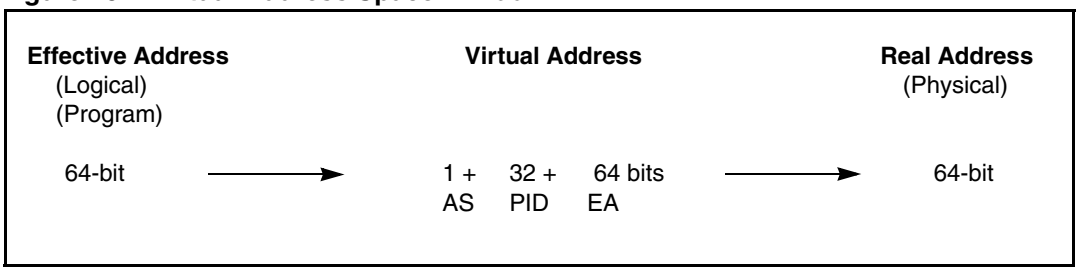
- The TLB, from a programming point of view, consists of zero or more TLB arrays, each of which may have differing characteristics.
- The logical-to-physical address translation mechanism
- Methods and effects of changing and manipulating TLB arrays
- Configuration information available to the operating system that describes the structure and form of the TLB arrays and translation mechanism

To assist or accelerate translation, implementations may contain other TLB structures not visible to the programming model. These structures and the methods for using them are not explicitly defined in the architecture or the ST standard, but they may be considered at the operating system level because they may affect an implementation's performance.

5.4.3 Virtual address (VA)

Book E defines a virtual address space composed of the effective address of an access, the 1-bit current address space (AS) of the access and the 32-bit process ID (PID) of an access, as shown in *Figure 16*. The following subsections describe the selection of AS and PID for an effective address, both used to construct the virtual address for an access.

Figure 16. Virtual Address Space in Book E

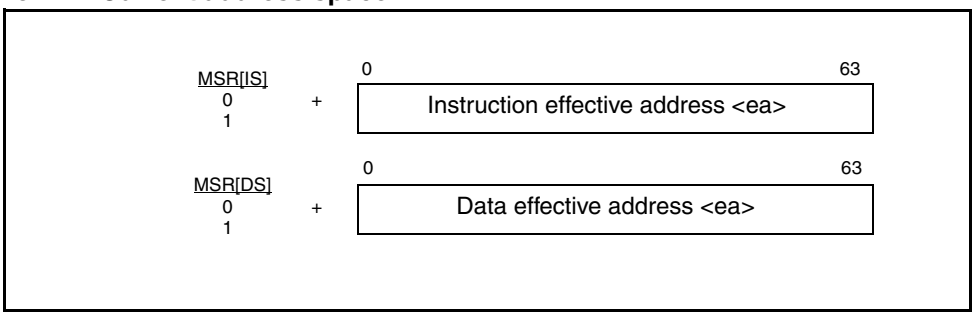


5.4.4 Address spaces

Instruction accesses are generated by sequential instruction fetches or due to a change in program flow (branches and interrupts). Data accesses are generated by load, store, and cache management instructions.

The Book E architecture defines two address spaces for instruction accesses and two address spaces for data accesses. The current address space for instruction or data accesses is determined by the value of MSR[IS] and MSR[DS], respectively, as shown in *Figure 17*.

Figure 17. Current address space



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 (**icbt**, **icbtls**, **icbtlc**) the value of the AS bit is taken from the contents of MSR[DS].

The address space indicator (MSR[IS] or MSR[DS], as appropriate) is used in addition to the effective address generated by the processor for translation into a physical address by the TLB mechanism.

Because MSR[IS] and MSR[DS] are cleared when an interrupt occurs, an address space value of zero can be used to denote interrupt-related address spaces, or possibly all system software address spaces; an address space value of one can be used to denote non–interrupt-related address spaces, or possibly all user address spaces.

Software Note: Although system software is free to use address space bits as it sees fit, on an interrupt, the MSR[IS] and MSR[DS] are cleared. This encourages software to use address space 0 for system software and address space 1 for user software.

Instruction address spaces

The two effective instruction address spaces are defined by the value of MSR[IS], and instruction fetch addresses are translated from the effective address space specified by the current value of MSR[IS]. Changing the value of MSR[IS] is considered a context-altering operation, requiring a context synchronization operation to follow it. When a context synchronizing event occurs, any prefetched instructions are discarded and instructions are refetched using the then-current state of MSR[IS] and the then-current program counter. See *Context synchronization on page 144*," for more information on the definition of context synchronizing events.

Instructions are not fetched from memory designated by the TLB mechanism as no-execute (UX = 0 or SX = 0). If the effective address of the current instruction is mapped to no-execute memory, an instruction storage interrupt (ISI) is generated.

Note that mapping a page as no-execute does not affect instruction caches in the system (or any instructions resident in unified caches). Thus, if an instruction is loaded into a cache when its effective address is mapped to execute permitted memory, and the execute permissions for that page are later changed to no-execute, any instructions fetched before the no-execute mapping remain in the cache until explicitly evicted by an **icbi** instruction or through the cache's replacement policy. However, attempted execution of such instructions still results in an ISI. Thus, for example, the operating system can change the designation of an application's instruction pages to no-execute without having to first flush instruction cache blocks that map to these pages.

Data address spaces

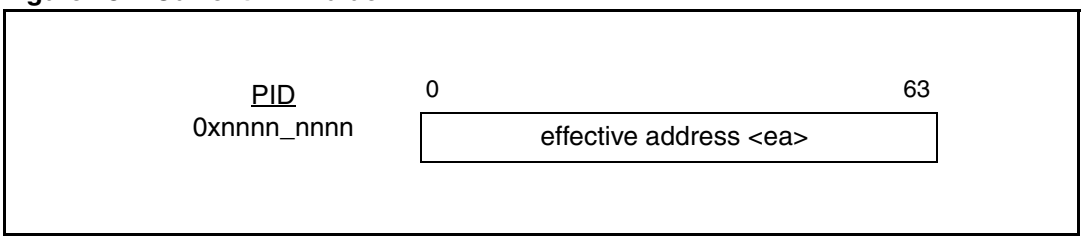
The two effective data address spaces are defined by the value of MSR[DS] and data is accessed to/from the effective address space specified by the current value of MSR[DS]. As is the case with MSR[IS], changing the value of MSR[DS] is considered a context-altering operation, requiring a context synchronization operation to follow it. When a context synchronizing event occurs, subsequent accesses are made using the new state of MSR[DS] (see *Context synchronization on page 144*").

Data can be read from a page, provided the user read (UR) permission bit is set in the TLB for a user access, or the supervisor read (SR) bit is set for a supervisor access. Likewise, data write access permissions are determined by the user write (UW) and supervisor write (SW) permission bits. If permissions are violated, the appropriate interrupt is taken.

5.4.5 Process ID

As described in *Chapter 2.12.1: Process ID registers (PID0–PIDn) on page 97*," Book E defines that a PID value be associated with each effective address (instruction or data) generated by the processor. At the Book E level, one 32-bit PID register maintains the PID value for the current process. This value is used to construct a virtual address for accessing memory.

Figure 18. Current PID Value



System software uses PIDs to identify TLB entries that the processor uses to translate addresses for loads, stores, and instruction fetches. PID contents are compared to the TID field in TLB entries as part of selecting appropriate TLB entries for address translation. PID values are used to construct virtual addresses for accessing memory. Note that individual processors may not implement all 14 bits of the process ID field.

Book E defines one PID register that holds the PID value for the current process. ST devices may implement from 1 to 15 PID registers. The number of PIDs implemented is indicated by the value of MMUCFG[NPIDS]. Consult the user documentation for the implementation to determine if other PID registers are implemented.

PID registers are more fully described in *Chapter 2.12.1: Process ID registers (PID0–PIDn) on page 97*."

Software Note: The suggested PID usage is for PID0 to denote private mappings for a process and for 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.

Process ID (PID) registers

The Book E architecture specifies that a process ID (PID) value be associated with each EA (instruction or data) generated by the processor.

System software uses PIDs to identify TLB entries that the processor uses to translate addresses for loads, stores, and instruction fetches. PID contents are compared to the TID field in TLB entries as part of selecting appropriate TLB entries for address translation. PID values are used to construct virtual addresses for accessing memory. Note that individual processors may not implement all 14 bits of the process ID field.

Book E defines one PID register that holds the PID value for the current process. ST devices may implement from 1 to 15 PID registers. The number of PIDs implemented is indicated by the value of MMUCFG[NPIDS]. Consult the user documentation for the implementation to determine if other PID registers are implemented.

The 15 PID registers supported by the EIS are implemented as SPR registers set by system software, and collectively reflect the process ID of the currently executing context. The system maintains multiple PID values in order to allow the sharing of TLB entries for pages that are shared among multiple execution contexts. For example, system software may

assign PID0 to contain the unique process ID (for private mappings for the current processes) and may assign PID1 to contain the unique process ID for a common set of shared libraries.

Note that Book E defines the value of all zeros for a TID field in a TLB entry as an entry that is globally shared. Thus, when PID values (up to 12 bits for ST devices) are compared to the TID fields in the TLB arrays for matches, if a TLB entry contains all zeros in the TID field, it globally matches all PID values. PID registers are more fully described in *Chapter 2.12.1: Process ID registers (PID0–PIDn) on page 97*."

Address space identifiers

The AS bit is the address space identifier. Thus there are two possible address spaces, 0 and 1. The value of the AS bit 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 (**icbt**, **icbtls**, **icbtlc**) the value of the AS bit is taken from the contents of MSR[DS]. The AS bit is defined by Book E.

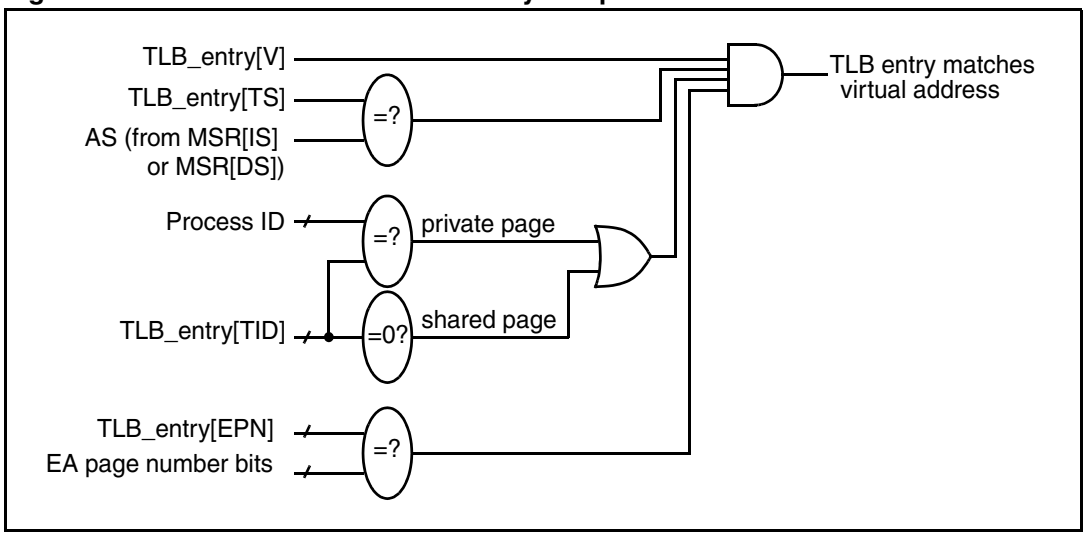
Note: Although system software is free to use address space bits as it sees fit, it should be noted that on interrupt, the MSR[IS] and MSR[DS] bits are cleared. This encourages software to use address space 0 for system software and address space 1 for user software.

5.4.6 Address translation

The effective address (EA) is the untranslated address for an instruction fetch address or for a data address that is calculated as a result of a load, store, or cache management instruction. The EA, concatenated with the MSR[IS] or MSR[DS] address space (AS) value, is compared to the appropriate number of bits of the EPN field (depending on the page size) and the TS field of the TLB entry. If a match occurs, that TLB entry is a candidate for a translation match. In addition to a match in the EPN field and TS, a matching TLB entry must match with the current process ID of the access.

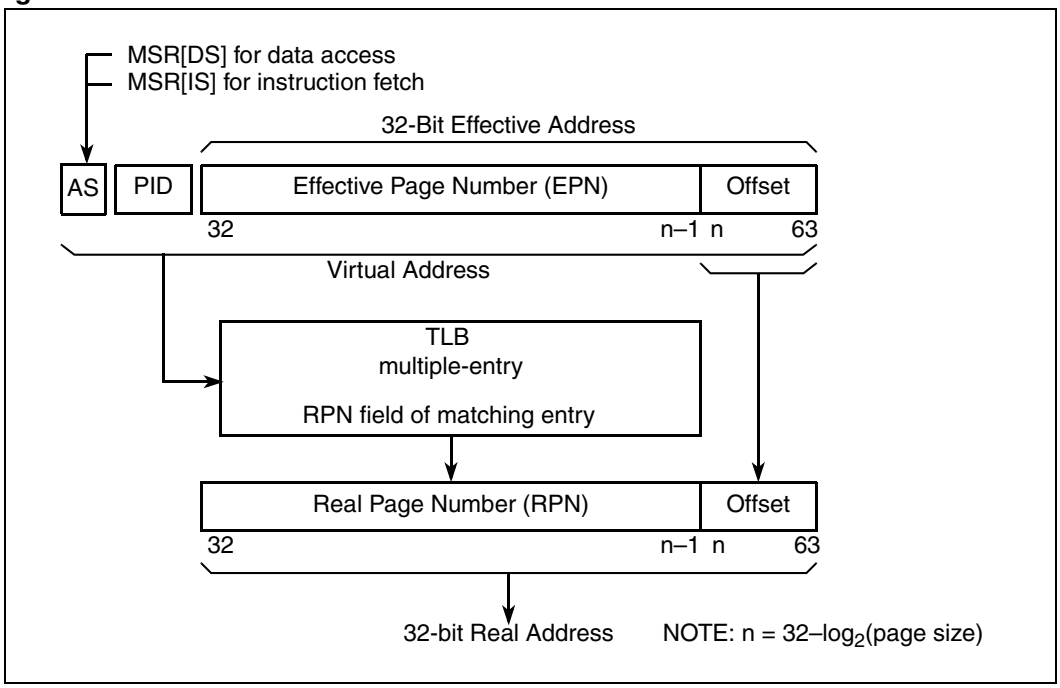
Figure 19 shows the translation match logic for the effective address plus its attributes (collectively called the virtual address) and how it is compared with the corresponding fields in the TLB entries.

Figure 19. Virtual address and TLB-entry comparison



The generation of the physical address occurs as shown in *Figure 20*.

Figure 20. Effective-to-real address translation



The EA combines with the AS and each PID register to form one virtual address for each unique PID register value. Also. an implicit virtual address is formed using a PID value of 0. Thus the following virtual addresses (VAs) are formed:

VA0 ← AS || 0 || EA
VA1 ← AS || PID0 || EA

...

VAn+1 ← AS || PIDn || EA

Note that a PID register containing a 0 value (or the same value as another PID register) forms a non-unique VA. Duplicate VAs are ignored.

Each of the unique VAs are compared to all the valid TLB entries by comparing specific fields of each TLB entry to each of the VAs. The fields of each valid (TLB[V] = 1) TLB entry are combined to form a set of matching TLB address (TAs):

$$TA \leftarrow TLB_{TS} \,||\, TLB_{TID} \,||\, TLB_{EPN} \,||\, {}^{12}0$$

Each TA is compared to all VAs under a mask based on the page size (TLB[SIZE]) of the TLB entry. The mask of the comparison of the EA and EPN portions of the virtual and translation addresses is computed as follows:

$$mask \leftarrow \sim(1024 << (2 * TLB_{SIZE})) - 1)$$

where the number of bits in the mask is equal to the number of bits in a TA (or VA). If a TA matches any VA the TLB entry is said to match. If more than one TA/VA match occurs, it is considered a serious programming error and the results are undefined. The recommended behavior is that a machine check interrupt is taken.

Once a match occurs the matching TLB is used for access control, storage attributes, and effective to real address translation. Access control, storage attributes, and address translation are defined by Book E (additional storage attributes are defined within this document).

5.4.7 Address translation and the ST EIS

Translating an effective address to a real address is defined by Book E to require four elements:

- The address space value. Depending on the type of translation (instruction or data), MSR[IS] or MSR[DS] is used.
- The TLB entries in the TLB arrays
- The effective address being translated

The following subsections describe these elements as they are further defined by the EIS.

Match criteria for TLB entries

TLB arrays contain TLB entries that are used to match any address presented for translation. All TLB entries for any given implementation are candidates for any given translation. The TLB itself is unordered with respect to the various elements used in address translations, and regardless of implementation, should be considered to perform the translation comparison with all entries in parallel.

There should be only one valid matching translation for a given effective address, PID value, and address space value. If the TLB contains more than one matching entry, it is considered a programming error, and the behavior of any such translation is undefined. In this case, the processor is likely to enter checkstop state or take a machine check interrupt.

The following fields are compared in the TLB entries:

- V—The matching entry must have the V bit set.
- TS—The address space identifier used for translation. The appropriate bit of MSR[IS] or MSR[DS] must match the TS bit for a matching entry.
- TID—The contents of a PID register must match the TID field of a matching entry, or the TID field must be all zeros for a matching entry.
- EPN—The appropriate number of bits (depending on the page size) of the effective address being translated is compared to the EPN field of the TLB entry.

If a match occurs on all the fields listed above, the physical address is formed by replacing the effective page number in the effective address with the value in the RPN field of the matching TLB entry. The number of bits in the page number depends on the page size for that TLB entry.

Translation algorithms

The following algorithm describes how translation operates at the ST Book E level:

```
ea = effective address
if translation is an instruction address then
   as = MSR[IS]
else                              // data address translation
   as = MSR[DS]
for all TLB entries
   if ! TLB_V then
      next                        // compare next TLB entry
   if as != TLB_TS then
      next
   if TLB_TID == 0 then
      goto pid_match
   for all PID registers
      if this PID register == TLB_TID then
         goto pid_match
   endfor
   next                           // no PIDs matched

pid_match:                        // translation match
   mask = ~((1024 << (2 * TLB_TSIZE)) - 01)
   if (ea & mask) != TLB_EPN then
      next                        // no address match
   real address = TLB_RPN | (ea & ~mask)  // real address computed
end translation -- success
endfor
end translation -- tlbmiss
```

The algorithm for the granting of permission is as follows:

```
if MSR_PR == 0 then
   x = TLB_SX
   r = TLB_SR
   w = TLB_SW
else
   x = TLB_UX
   r = TLB_UR
```

```
    w = TLB_UW

if instruction fetch address then
    if x == 0 then
        Instruction Storage Interrupt
else                                // data access
    if data read (load) then
        if r == 0 then
            Data Storage Interrupt
    else                            // write access (store)
        if w == 0 then
            Data Storage Interrupt
```

Access control

If address translation results in a match (hit), the matching TLB entry is used to perform access control (permission checks). These checks are based on the privilege level of the access (MSR[PR]) and the type of access (fetch for execute, read for loads, and write for stores). The TLB entry's permission bits (TLB[US,SX,UW,SW,UR,SR]) determine if the operation should succeed. If permission is denied, execution of the instruction is suppressed and an instruction storage interrupt or data storage interrupt occurs as defined in Book E. Software uses the ESR, SRR0, and the DEAR to determine the type of operation attempted and then must perform a TLB search if updating the TLB is desired.

The algorithm for determining access control is as follows:

```
if MSR_PR = 0 then
    x ← TLB_SX
    r ← TLB_SR
    w ← TLB_SW
else
    x ← TLB_UX
    r ← TLB_UR
    w ← TLB_UW

if instruction_fetch & x = 0 then
    take instruction storage interrupt
else if load &       r = 0 then
    take data storage interrupt
else if store &      w = 0 then
    take data storage interrupt
else
    access permitted
```

Physical (real) address generation

If permission checking is successful, the real address is formed by combining the TLB[RPN] with the lower order offset bits of the EA based on the page size of the TLB entry.

```
mask ← ~(1024 << (2 * TLB_SIZE)) - 1)
real_address ← ((TLB_RPN << 12) & mask) | (EA & ~mask)
```

Where mask contains the same number of bits as a real address. The real address is then used to access the memory subsystem using the TLB[ACM,VLE,W,I,M,G,E] fields from the TLB entry to determine how the location should be accessed.

Page size and effective address bits compared

The page size defined for a TLB entry determines how many bits of the effective address are compared with the corresponding EPN field in the TLB entry as shown in *Table 185*.

Table 185. Page size and EPN field comparison

SIZE Field	Page Size (4^{SIZE}Kbytes)	EA to EPN Comparison (Bits 32–53; 2×SIZE)
0b0000	1 Kbyte	EA[32–53] = ? EPN[32–53]
0b0001	4 Kbyte	EA[32–51] = ? EPN[32–51]
0b0010	16 Kbyte	EA[32–49] = ? EPN[0–49]
0b0011	64 Kbyte	EA[32–47] = ? EPN[32–47]
0b0100	256 Kbyte	EA[32–45] = ? EPN[32–45]
0b0101	1 Mbyte	EA[32–43] = ? EPN[32–43]
0b0110	4 Mbyte	EA[32–41] = ? EPN[32–41]
0b0111	16 Mbyte	EA[32–39] = ? EPN[32–39]
0b1000	64 Mbyte	EA[32–37] = ? EPN[32–37]
0b1001	256 Mbyte	EA[32–35] = ? EPN[32–35]
0b1010	1 Gbyte	EA[32–33] = ? EPN[32–33]

Permission attribute comparison

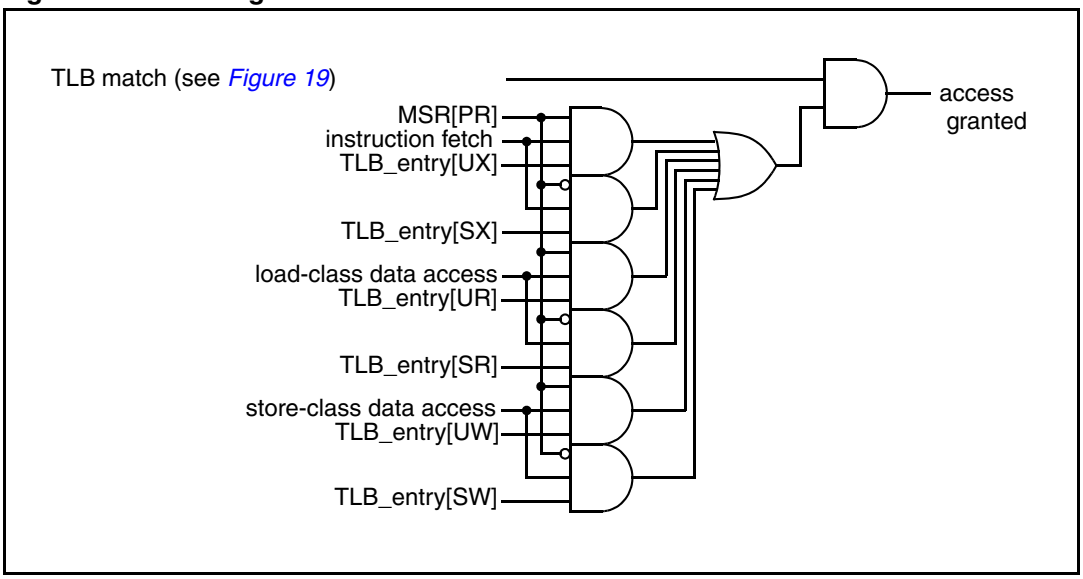
As part of the translation process, the selected TLB entry provides the access permission bits (UX, SX, UW, SW, UR, SR), and memory/cache attributes (U0, U1, U2, U3, W, I, M, G, and E) for the access. These bits specify whether or not the access is allowed and how the access is to be performed.

If a matching TLB entry has been identified, Book E provides an access permission mechanism that selectively grants shared access, grants execute access, grants read access, grants write access, and prohibits access to areas of memory based on a number of criteria. Book E defines the permission bits in TLB entries as follows:

- SR—Supervisor read permission
- SW—Supervisor write permission
- SX—Supervisor execute permission
- UR—User read permission
- UW—User write permission
- UX—User execute permission

If the virtual address translation comparison with TLB entries was successful, the permission bits for the matching entry are checked as shown in *Figure 21*. If the access is not allowed by the access permission mechanism, the processor generates an instruction or data storage interrupt (ISI or DSI).

Figure 21. Granting of Access Permission



The permission attributes defined by Book E are defined in detail in *Chapter 5.4.8*."

Page size and real address generation

If no virtual address match occurs, the translation fails and a TLB miss exception occurs. Depending on the access type (instruction or data address), either the instruction TLB error interrupt or the data TLB error interrupt is taken.

Otherwise, the real page number (RPN) field of the matching TLB entry provides the translation for the effective address of the 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 = 32 - $\log_2$(page size). Note that the untranslated bits must be zero in the RPN field.

Table 186. Real address generation

Size field	Page size (4^{SIZE}Kbytes)	RPN bits required to be equal to 0	Real address
0b0000	1 Kbyte	none	RPN[32–53] || EA[54–63]
0b0001	4 Kbyte	RPN[52–53] = 0	RPN[32–51] || EA[52–63]
0b0010	16 Kbyte	RPN[50–53] = 0	RPN[32–49] || EA[50–63]
0b0011	64 Kbyte	RPN[48–53] = 0	RPN[32–47] || EA[48–63]
0b0100	256 Kbyte	RPN[46–53] = 0	RPN[32–45] || EA[46–63]
0b0101	1 Mbyte	RPN[44–53] = 0	RPN[32–43] || EA[44–63]
0b0110	4 Mbyte	RPN[42–53] = 0	RPN[32–41] || EA[42–63]
0b0111	16 Mbyte	RPN[40–53] = 0	RPN[32–39] || EA[40–63]
0b1000	64 Mbyte	RPN[38–53] = 0	RPN[32–37] || EA[38–63]
0b1001	256 Mbyte	RPN[36–53] = 0	RPN[32–35] || EA[36–63]
0b1010	1 Gbyte	RPN[34–53] = 0	RPN[32–33] || EA[34–63]

5.4.8 Permission attributes

The permission attributes defined in Book E are shown in *Table 187* and described in the following subsections.

Table 187. Permission control for instruction, data read, and data write accesses

Access type	MSR[PR]	TLB[UX]		TLB[SX]		TLB[UR]		TLB[SR]		TLB[UW]		TLB[SW]	
		0	1	0	1	0	1	0	1	0	1	0	1
Instruction fetch	0	—	—	ISI	√	—	—	—	—	—	—	—	—
	1	ISI	√	—	—	—	—	—	—	—	—	—	—
Data read (load)	0	—	—	—	—	—	—	DSI	√	—	—	—	—
	1	—	—	—	—	DSI	√	—	—	—	—	—	—
Data write (store)	0	—	—	—	—	—	—	—	—	—	—	DSI	√
	1	—	—	—	—	—	—	—	—	DSI	√	—	—

Execute access permission

The UX and SX bits of the TLB entry control execute access to the corresponding page.

Instructions may be fetched and executed from a page in memory if MSR[PR] = 1 (user mode) if the UX access control bit for that page is set. If the UX access control bit is cleared, instructions from that page are not fetched and they are not placed into any cache while the processor is in user mode.

Instructions may be fetched and executed from a page in memory if MSR[PR] = 0 (supervisor mode) and the SX access control bit for that page is set. If the SX access control bit is cleared, instructions from that page are not fetched and are not placed into any cache while the processor is in supervisor mode.

If the sequential execution model calls for the execution of an instruction from a page that is not enabled for execution (that is, UX = 0 when MSR[PR] = 1 or SX = 0 when MSR[PR] = 0), an execute access control exception-type instruction storage interrupt (ISI) is taken.

Read access permission

The UR and SR bits of the TLB entry control read access to the corresponding page.

Load operations (including load-class cache management instructions) are permitted from a page in memory while the processor is in user mode (MSR[PR] = 1) if the UR access control bit for that page is set. If the UR access control bit is cleared, execution of the load instruction is suppressed and a read access control exception-type data storage interrupt (DSI) is taken.

Similarly, load operations (including load-class cache management instructions) are permitted from a page in memory if MSR[PR] = 0 (supervisor mode) and the SR access control bit for that page is set. If the SR access control bit is cleared, execution of the load instruction is suppressed and a read access control exception-type data storage interrupt (DSI) is taken.

Write access permission

The UW and SW bits of the TLB entry control write access to the corresponding page.

Store operations (including store-class cache management instructions) are permitted to a page in memory if MSR[PR] = 1 (user mode) and the UW access control bit for that page is set. If the UW access control bit is cleared, execution of the store instruction is suppressed and a write access control exception-type data storage interrupt (DSI) is taken.

Similarly, store operations (including store-class cache management instructions) are permitted to a page in memory if MSR[PR] = 0 (supervisor mode) and the SW access control bit for that page is set. If the SW access control bit is cleared, execution of the store instruction is suppressed and a write access control exception-type data storage interrupt (DSI) is taken.

Permission control and cache management instructions

The **dcbi** and **dcbz** instructions are treated as stores because they can change data (or cause loss of data by invalidating a modified line). As such, they both can cause write access control exception-type DSIs.

The **dcba** instruction is treated as a store because it can also change data. As such, it can also cause a write access control exception. However, these exceptions do not result in a data storage interrupt and if a permission violation occurs, the instruction execution completes, but the allocate operation is merely cancelled (essentially, a no-op).

The **icbi** instruction is treated as a load with respect to permissions checking. As such, it can cause a read access control exception-type data storage interrupt.

The **dcbt**, **dcbtst**, and **icbt** instructions are treated as loads with respect to permissions checking. As such, they can cause read access control exceptions. However, such exceptions do not result in data storage interrupts and if a permission violation occurs, the instruction execution completes, but the operation is cancelled (essentially, a no-op).

The **dcbf** and **dcbst** instructions are treated as loads with respect to permissions checking. Flushing or storing a line from the cache is not considered a store because the store has already been performed to update the cache and the **dcbf** or **dcbst** instruction is only updating the copy in main memory. Like load instructions, these instructions can cause read access control exception-type data storage interrupts.

Table 188 summarizes exception cases caused by the cache management instructions due to permissions violations.

Table 188. Permission control and cache instructions

Instruction	Can cause read permission violation exception?	Can cause write permission violation exception?
dcba	No	Yes(1)
dcbf	Yes	No
dcbi	No	Yes
dcbst	Yes	No
dcbt	Yes[1]	No
dcbtst	Yes[1]	No
dcbz	No	Yes
icbi	Yes	No
icbt	Yes[1]	No

1. **dcba**, **dcbt**, **dcbtst**, and **icbt** may cause a read access control exception but does not result in a data storage interrupt (DSI).

Permissions control and string instructions

When the string length is zero, neither **lswx** nor **stswx** can cause data storage interrupts due to permissions violations.

Use of permissions to maintain page history

The Book E architecture TLB entry definition does not define bits for maintaining page history information. The U0–U3 bits in the TLB entries can be used by software for storing history information, but implementations may ignore these bits internally.

Page changed bit status can be implemented in the system software by disabling write permissions to all pages. The first attempt to write to the page results in a data storage interrupt. At this point, system software can record the page changed bit in memory, update the TLB entry permission to allow writes to that page, and return to the user program allowing further writes to the page to proceed without exception.

Crossing page boundaries

Care must be taken with single instruction accesses (load/stores) that cross page boundaries. Examples are **lmw** and **stmw** instructions and misaligned accesses on implementations that support misaligned load/stores. Architecturally, each of the parts of the access that cross the natural boundary of the access size (half word, word, double word) are treated separately with respect to exception conditions. Additionally, these types of instructions may optionally partially complete. For example, a store word instruction that crosses a page boundary because it is misaligned to the last half word of a page might actually store the first 16 bits because the access was permitted, but produce a DSI or data TLB error exception because the second 16 bits in the next page were not valid or they were protected. An implementation may choose to suppress the first 16-bit store or perform it.

5.4.9 Translation lookaside buffer (TLB) arrays

The MMU contains up to four TLB arrays, which are on-chip storage areas for holding TLB entries. A TLB entry contains effective-to-physical address mappings for loads, stores, and instruction fetches. A TLB array must contain TLB entries that share the same characteristics and contains zero or more TLB entries. Each TLB entry has specific fields that can be addressed by the corresponding fields in the MMU assist registers (see *Chapter 2.12.5: MMU assist registers (MAS0–MAS7) on page 101*"). Each implemented TLB array has an associated configuration register (TLB*n*CFG) describing the size and attributes of the TLB entries in that array. See *Chapter 2.12.4: TLB configuration registers (TLBnCFG) on page 100*."

The architected fields of a TLB entry are described in *Table 189*.

Table 189. TLB entry

Name	Description
V	Valid bit. A 1-bit entry that specifies whether this TLB entry is valid for translation. (1 = valid).
TID	Translation ID. Identifies which process ID (PID) that this TLB entry is valid for. Translation IDs are compared with Process IDs (PIDs) during translation to identify which TLB entry to use for translating an address. A TID value of 0 is considered global and matches all PID values.
TS	Translation space. Identifies which address space that this TLB entry is valid for. The translation space field is compared with MSR[IS] for instruction accesses and the MSR[DS] bit for data accesses. This allows for an efficient change of address space when a transition from user mode to supervisor mode occurs. This is a 1 bit field.
SIZE	Page size. Describes the size of the page. An implementation is not required to support variable page sizes or any particular page size. If a TLB array does not support variable size pages (that is, TLB*n*CFG[AVAIL] _ 0) then this field is ignored. Page sizes range from 4 Kbytes to 1 Tbyte in powers of 4. EIS does not support 1-Kbyte page sizes defined in Book E. Page size encoding is defined by Book E.
EPN	Effective page number. Describes the logical or effective starting address of the page. The number of bits that are valid (used in translation) depends on the size of the page and if the processor is a 32- or 64-bit implementation. This field is used to compare with the EA being translated to identify which TLB entry to use for translation. For 32-bit implementations, this field is 2 to 20 bits, depending on the page size (SIZE).
RPN	Real page number. Describes the physical starting address of the page. The real page number is substituted for the effective page number from the address being translated which results in the real address. This field is 2 to 52 bits depending on the page size and the number of bits of real address supported by the implementation.
WIMGE	Storage attributes. Describe the characteristics of any memory/fetch accesses to the page and the subsequent treatment of those data items with respect to the memory subsystem (caches and bus transactions). The WIMGE bits are defined by Book E.
ACM	Alternate coherency mode. Optional. An implementation may optionally support additional coherency models. If such coherency models are provided, they are encoded in this field. ACM values are implementation dependent. The Alternate Coherence Mode is used only when the M bit (from WIMGE) is set.
VLE	Variable length encoding. Optional. If an implementation supports the VLE extension, clearing VLE causes instruction access to this page to decode and execute as PowerPC Book E (and EIS APUs) instructions, and setting VLE causes instruction access to this page to decode and execute as VLE (and EIS APU) instructions.
SR,SW,SX, UR,UW,UX	Permissions. User and supervisor read, write, and execute permission bits. Supervisor and user permission bits are defined by Book E.
U0,U1,U2,U3	User bits. Implementation dependent. Consult the user's manual for any implementation usage. It is strongly recommended to leave these as storage associated with a TLB entry to be used by system software.
IPROT	Invalidation protection. Invalidation protection. This entry is protected from all TLB invalidation mechanisms except the explicit writing of a 0 to the V bit.

5.4.10 TLB management

TLB entries are managed by software using the set of MAS registers, which are used to move data between software and the TLB entries, to identify TLB entries, and to provide

default values when translation or protection faults occur. See *Chapter 2.12.5: MMU assist registers (MAS0–MAS7) on page 101*."

TLB configuration information

Information about the configuration for a given TLB implementation is available to system software by reading the contents of the MMU configuration SPRs. These SPRs describe the architectural version of the MMU, the number of TLB arrays, and the characteristics of each TLB array. MMU architecture version number 1 is defined as these SPRs with the field definitions as described in this section.

- MMU configuration register (MMUCFG), implemented by all ST Book E processors, contains basic information about the MMU architecture for each device. TLB configuration registers (TLB*n*CFG). Implemented by all ST Book E processors for each of the TLBs specified in MMUCFG[NTLBS]. They contain configuration information about each particular TLB. See *Chapter 2.12.3: MMU configuration register (MMUCFG) on page 99*."
- The TLB*n*CFG number assignment is the same as the value in MAS0[TLBSEL]. For example, TLB0CFG provides configuration information about TLB0, and TLB1CFG provides configuration information about TLB1. See *Chapter 2.12.4: TLB configuration registers (TLBnCFG) on page 100*."

TLB entries

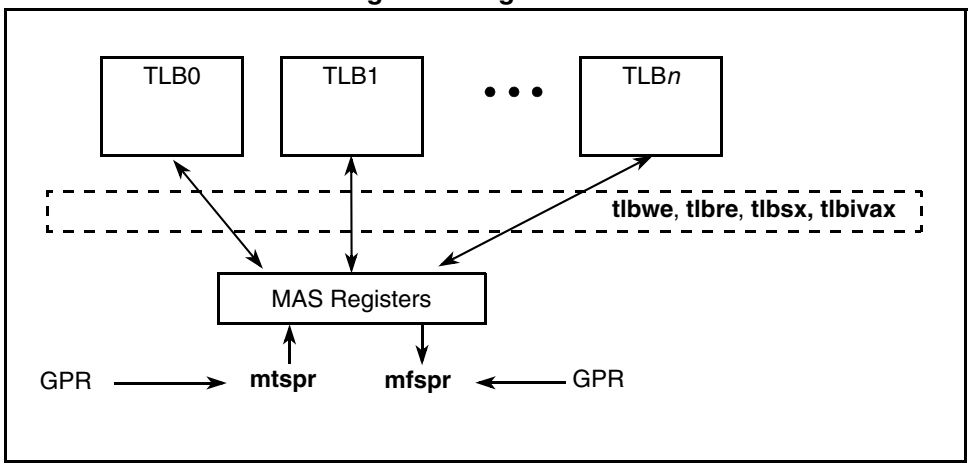
The software-visible TLB is subdivided into zero or more TLB arrays. Each array must contain TLB entries that share the same characteristics. Each TLB array contains one or more TLB entries. Each entry has specific fields that correspond to fields in the seven MMU assist (MAS) registers, described in *Chapter 2.12.5: MMU assist registers (MAS0–MAS7) on page 101*." Some TLB fields are architected in Book E and others are architected in the EIS. Note that Book E architected fields may have restrictions or enhancements imposed by the EIS for the Book E implementations.

The IPROT TLB entry, architected by the EIS, designates TLB entries as protected from certain kinds of invalidation. TLB invalidation and the IPROT field are described further in *Invalidating TLB entries on page 321*."

Reading and writing TLB entries

All TLB entries are updated by executing **tlbwe** instructions. At the time of **tlbwe** execution, the MMU assist registers (MAS0–MAS6), a set of SPRs defined by the EIS, are used to index a specific TLB entry. The MAS registers also contain the information that is written to the indexed entry, such that they serve as the ports into the TLBs, as shown in *Figure 22*. The contents of the MAS registers are described in *Chapter 2.12.5: MMU assist registers (MAS0–MAS7) on page 101*.

Figure 22. TLBs accessed through MAS registers and TLB instructions



Similarly, TLB entries are 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 contain the contents of the indexed TLB entry. To read or write TLB entries, the MAS registers are first loaded by system software using **mtspr** instructions and then the desired **tlbre** or **tlbwe** instructions must be executed.

Note that RA = 0 is a preferred form for **tlbsx** and that some Book E implementations take an illegal instruction exception program interrupt if RA ≠ 0.

Reading TLB entries

TLB entries are 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**, the MAS registers contain the contents of the indexed TLB entry.

Selection of the TLB entry to read is performed by setting MAS0[TLBSEL], MAS0[ESEL] and MAS2[EPN] to indicate the entry to read. MAS0[TLBSEL] selects which TLB the entry should be read from (0 to 3) and MAS2[EPN] selects the set of entries from which MAS0[ESEL] selects an entry. For fully associative TLBs, MAS2[EPN] is not required since the value in MAS0[ESEL] fully identifies the TLB entry. Valid values for MAS0[ESEL] are from 0 to associativity - 1.

The selected TLB entry is then used to update the following fields of the MAS registers: V, IPROT, TID, TS, TSIZE, EPN, ACM, VLE, WIMGE, RPN, U0—U3, & permissions. If the TLB array supports NV, it is used to update the NV field in the MAS registers, otherwise the contents of NV field are undefined. The update of MAS registers as a result of a **tlbre** instruction is summarized in *Table 191*.

No operands are given for the **tlbre** instruction and the Book E defined implementation dependent field should be treated as a reserved field.

Specifying invalid values for MAS0[TLBSEL] and MAS0[ESEL] produce boundedly undefined results.

Writing TLB entries

TLB entries are 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 TLB entry contents of the MAS registers are written to the indexed TLB entry.

Selection of the TLB entry to write is performed by setting MAS0[TLBSEL], MAS0[ESEL] and MAS2[EPN] to indicate the entry to write. MAS0[TLBSEL] selects which TLB the entry should be written from (0 to 3) and MAS2[EPN] selects the set of entries from which MAS0[ESEL] selects an entry. For fully associative TLBs, MAS2[EPN] is not used to identify a TLB entry since the value in MAS0[ESEL] fully identifies the TLB entry. Valid values for MAS0[ESEL] are from 0 to associativity minus 1.

The selected TLB entry is then written with following fields of the MAS registers: V, IPROT, TID, TS, TSIZE, EPN, ACM, VLE, WIMGE, RPN, U0—U3, and permissions. If the TLB array supports NV, it is written with the NV value.

The effects of updating the TLB entry are not guaranteed to be visible to the programming model until the completion of a context synchronizing operation. Writing a TLB entry that is used by the programming model prior to a context synchronizing operation produces undefined behavior.

No operands are given for the **tlbwe** instruction and the Book E defined implementation dependent field should be treated as a reserved field.

Specifying invalid values for MAS0[TLBSEL] and MAS0[ESEL] produce boundedly undefined results.

*Note: Writing TLB entries should be followed by an **isync** or an **rfi** before the new entries are to be used by the programming model.*

Invalidating TLB entries

TLB entries may be invalidated by any of the following methods:

- A TLB entry can be invalidated as the result of a **tlbwe** instruction that clears MAS0[V] in the entry.
- As a result of a **tlbivax** instruction or from a received broadcast invalidation resulting from a **tlbivax** on another processor in an SMP system.
- As a result of a flash invalidate.

In both multiprocessor and uniprocessor systems, invalidations can occur on a wider set of TLB entries than intended. This are called generous invalidations That is, a virtual address presented for invalidation may invalidate not only the targeted TLB, but also may 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 broadcast globally to all processors in the system. Hardware may impose other limitations.

The architecture ensures that the intended TLB is invalidated, but does not guarantee that it is the only one. A TLB entry invalidated by clearing the V bit of the TLB entry by use of a **tlbwe** is guaranteed to invalidate only the addressed TLB entry. However, invalidates occurring from **tlbivax** instructions or from the multiprocessor broadcasts as a result of **tlbivax** instructions may cause generous invalidates.

The architecture provides a method to protect against generous invalidations. This is important, because certain virtual memory regions (most notably, the code memory region that serves as the exception handler for MMU faults) must be properly mapped to for forward progress to occur. If this region does not have a valid mapping, an MMU exception cannot be handled because the first address of the interrupt handler causes another MMU exception. To prevent this, the architecture specifies an IPROT bit for TLB entries. Setting the MAS0[PROT] protects the corresponding TLB entry from invalidations resulting from **tlbivax** instructions, as a result of broadcast invalidation from another processor in an SMP

system, or from flash invalidations. TLB entries with the IPROT field set can be invalidated only by explicitly writing the TLB entry and specifying a 0 for MAS1[V].

Note: Software Note: Not all TLB arrays in a given implementation implement the IPROT attribute. It is likely that implementations that are suitable for demand page environments implement it for only a single array, while not implementing it for other arrays.

Software Note: Operating systems must use great care when using protected (IPROT) TLB entries, particularly in SMP systems. An SMP system that contains TLB entries on other processors requires a cross-processor interrupt or some other synchronization mechanism to assure that each processor performs the required invalidation by writing its own TLB entries.

Invalidations using tlbivax:

The **tlbivax** instruction provides a virtual address as a target for invalidation. EA[0–51] are used to find a TLB entry with a matching EPN field. The page size of the TLB entry is used to mask the low order bits in the comparison. The comparison is performed only for TLB entries in the specified TLB array, do not have the IPROT attribute set (if supported by the TLB array), and are valid. The AS bit does not participate in the comparison. The EA specified by the **r**A and **r**B operands in the **tlbivax** instruction contains fields in the lower order bits to augment the invalidation to specific TLB arrays and to flash invalidate those arrays. Note that TLB entry invalidations resulting from **tlbivax** instructions do not invalidate any entry that has IPROT = 1 unless the specified TLB array does not support the IPROT attribute. The encoding of the EA used by **tlbivax** is shown in *Table 190*.

*Note: Software 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.*

*Software Note: The preferred form of the **tlbivax** instruction contains the entire EA in **r**B and zero in **r**A. Some implementations may take an Unimplemented Instruction exception if **r**A is non-zero.*

EA format for **tlbivax ia shown below**.

EA Format for tlbivax

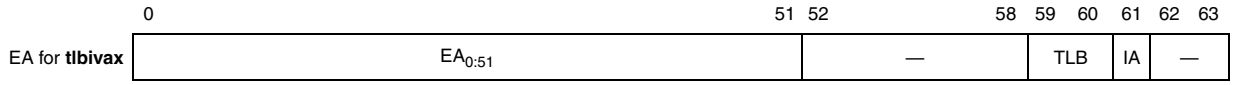


Table 190 describes EA fields for **tlbivax**.

Table 190. Fields for EA format of tlbivax

Field	Name	Comments or function when set
0–51	$EA_{0:51}$	The upper bits of the address to invalidate.
52–58	—	Reserved, should be cleared.
59–60	TLB	Selects TLB array for invalidation. 00TLB0 01TLB1 10TLB2 11TLB3
61	IA	Invalidate all entries in selected TLB array.
62–63	—	Reserved, should be cleared.

Flash invalidations using MMUCSR0:

All entries in a TLB array may be flash invalidated using the MMUCSR0 register. Flash invalidation of an array is started when the corresponding flash invalidate bit is set in MMUCSR0 (MMUCSR0[TLB*n*_FI]). The flash invalidation is complete when the corresponding flash invalidate bit is cleared by the processor. Writing a 0 to a flash invalidate bit in MMUCSR0 has no effect. Note that TLB entry invalidations resulting from MMUCSR0 flash invalidations do not invalidate any entry that has IPROT = 1 unless the specified TLB array does not support the IPROT attribute.

Searching TLB Entries

Software may search the MMU by using the **tlbsx** instruction that is provided by Book E. 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.

To properly execute a search for a TLB, software loads MAS5 and MAS6 registers with PID and AS values to search for. These are MAS6[SPID0], MAS6[SPID1], MAS5[SPID2], MAS5[SPID3], MAS6[SAS]. Software then executes a **tlbsx** instruction. The search performs the same TA to VA comparison described in *Chapter 5.4.6*," except that the PID and AS values are taken from the MAS registers. If a matching, valid TLB entry is found, the MAS register are loaded with the information from that TLB entry as if the TLB entry was read from a **tlbre** instruction. Software can examine the MAS1[V] bit to determine if the search was successful. Successful searches cause the valid bit to be set. *Table 191* summarizes the update of MAS registers as a result of a **tlbsx** instruction.

The preferred form of the **tlbsx** is **r**A = 0. Some implementations may take an unimplemented instruction exception or an illegal instruction exception if **r**A != 0.

TLB replacement hardware assist

The architecture provides mechanisms to accelerate software in creating and updating TLB entries when MMU related exceptions occur. This is called TLB replacement hardware assist. Hardware updates the MAS registers on the occurrence of a data TLB error interrupt or instruction TLB error interrupt.

When a TLB error exception (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 TLB 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, MAS0[NV] 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 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 remaining fields are located in MAS3, requiring only the single MAS register manipulation by software before writing the TLB entry.

For ISI and 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 191*.

Table 191. MAS register update summary

MAS field updated	Value loaded on event			
	TLB error interrupt	tlbsx hit	tlbsx miss	tlbre
MAS0[TLBSEL]	MAS4[TLBSELD]	TLB array that hit	MAS4[TLBSELD]	—
MAS0[ESEL]	***if*** MAS4[TLBSELD] supports next victim ***then*** hardware hint, ***else*** undefined	Number of entry that hit	***if*** MAS4[TLBSELD] supports next victim ***then*** hardware hint, ***else*** undefined	—
MAS0[NV]	***if*** MAS4[TLBSELD] supports next victim ***then*** next hardware hint, ***else*** undefined	***if*** MAS4[TLBSELD] supports next victim ***then*** hardware hint, ***else*** undefined	***if*** MAS4[TLBSELD] supports next victim ***then*** next hardware hint, ***else*** undefined	***if*** MAS4[TLBSELD] supports next victim ***then*** hardware hint, ***else*** undefined
MAS1[V]	1	1	0	TLB[V]
MAS1I[PROT]	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]	$EA_{0:51}$	TLB[EPN]	—	TLB[EPN]
MAS2[ACM]	MAS4[ACMD]	TLB[ACM]	MAS4[ACMD]	TLB[ACM]
MAS2[VLE]	MAS4[VLED]	TLB[VLE]	MAS4[VLED]	TLB[VLE]

Table 191. MAS register update summary (continued)

MAS field updated	Value loaded on event			
	TLB error interrupt	tlbsx hit	tlbsx miss	tlbre
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]	—	TLB[U0,U1,U2,U3]	—	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	—	—	—	—
MAS5	—	—	—	—
MAS6[SPID0]	PID0	—	—	—
MAS6[SPID1]	—	—	—	—
MAS6[SAS]	MSR[IS] or MSR[DS]	—	—	—
MAS7[RPN]	0	TLB[RPN] (bits 0–31)	0	TLB[RPN] (bits 0–31)

5.4.11 MAS registers and exception handling

When translation-related exceptions occur, hardware preloads the MAS registers with information that the interrupt handler likely needs to handle the fault. For a TLB miss exception, some MAS register fields are loaded with default information specified in MAS4. System software should set up the default information in MAS4 before allowing exceptions. In most cases, system software sets this up once depending on its scheme for handling page faults. This simplifies translation-related exception handling. The following subsections detail specific MAS register fields and the contents loaded for each exception type.

TLB miss exception types

The Book E architecture defines that a TLB miss exception is caused when a virtual address for an access does not match with that of any on-chip TLB entry. This condition causes one of the following:

- An instruction TLB error interrupt
- A data TLB error interrupt

Instruction TLB error interrupt settings

An instruction TLB error interrupt occurs when the virtual address associated with an instruction address (fetch) does not match any valid entry in the TLB (that is, the address for the instruction cannot be translated). In addition to the values automatically written to the

MAS registers (described in *TLB miss exception MAS register settings on page 326*"), SRR0 contains the address of the instruction that caused the instruction TLB error. This SRR0 value is used to identify the EA for handling the exception as well as the address to return to when system software has resolved the exception condition by writing a new TLB entry.

Data TLB error interrupt settings

A data TLB error interrupt occurs when the virtual address associated with a data reference from a load, store, or cache management instruction does not match any valid entry in the TLB (that is, the address of the data item of a load or store instruction cannot be translated). In addition to the values automatically written to the MAS registers (described in *TLB miss exception MAS register settings on page 326*"), the effective address of the data access that caused the exception is automatically loaded in the data exception address register (DEAR). Also, SRR0 contains the address of the instruction that caused the data TLB error and its value is used to identify the address to return to when system software has resolved the exception condition (by writing a new TLB entry).

TLB miss exception MAS register settings

When either an instruction or data TLB error interrupt occurs, the TLB information and selection fields of the MAS registers are loaded with default values from other MAS registers to assist in processing the exception. The intention is that the common case of a page fault generally requires only system software to load the RPN (corresponding to the physical address that will be used for this page), and the access permissions and the defaults can be used for the remaining MAS fields.

The processor may use the next victim (NV) field from the TLB array to select which TLB entry should be used for the new translation. The method used to select the candidate TLB for replacement (the next victim) is implementation-dependent and may vary on different Book E implementations. In any case, software is free to choose any TLB entry for the replacement (software can overwrite the value in MAS0[ESEL]).

The EIS defines the fields set in the MAS registers at exception time for an instruction or data TLB error interrupt as shown in *Table 192*.

Table 192. MAS settings for an instruction or data TLB error interrupt

MAS Field	Value
TLBSEL	Set to value in TLBSELD (default). This defines the TLB array to be used for the new TLB entry that will be written.
ESEL	May be set to an implementation-dependent value, usually based on the NV field of the array selected by TLBSELD, if that TLB supports the NV function. If the selected TLB does not support NV, the value loaded into ESEL is undefined.
NV	Set to an implementation-dependent value to select which TLB entry to replace on the next TLB miss. The NV field of the TLB array is updated by the value of NV in the MAS registers when **tlbwe** is executed.
V	Set
IPROT	Cleared
TID	Set to the contents of the PID register referenced by TIDSELD. That is, if TIDSELD contains the value 1, the contents of PID1 are written to the TID field.
TS	Set to the value of the IS or DS bit in the MSR at the time of the exception (that is, the MSR that described the context that was running when the exception occurred).
TSIZE	Set to TSIZED
EPN	Set to the effective page number of the instruction or data address causing the exception. The number of bits of the page number is implementation-dependent, but should be consistent with the TLB array selected if the TLB has a fixed page size. If the TLB array selected by TLBSELD contains variable-sized pages, the value for EPN is undefined.
WIMGE and X bits	Set to corresponding default values in the MAS registers
RPN	Cleared
Permissions	SR, UR, UW, SW, UX, SX cleared to 0 (no permissions); note that U0–U3 are unchanged.

All other MAS register values are unchanged.

Permissions violation exception types

The Book E architecture also defines that a permissions violation exception is caused when an effective address for an access matches with a TLB entry but the permission attributes in the matching TLB entry do not allow the access to proceed, as described in *Chapter 5.4.8*." This condition causes an instruction storage interrupt (ISI) or a data storage interrupt (DSI)

Instruction storage interrupt settings

An instruction storage interrupt occurs when the effective address associated with an instruction address (fetch) matches a valid entry in the TLB but one of the permission bits in the TLB does not allow the instruction fetch. In addition to the values automatically written to the MAS registers (described in *Permissions violation mas register settings on page 328*"), SRR0 contains the address of the instruction that caused the instruction TLB error and this value is used to identify the effective address for handling the exception as well as the address to return to when system software has resolved the exception condition (by writing a new TLB entry).

Data storage interrupt settings

A data storage interrupt occurs when the EA associated with a data reference from a load or store instruction matches with a valid entry in the TLB but one of the permission bits in the TLB does not allow the data access. In addition to the values automatically written to the MAS registers (described in *Permissions violation mas register settings on page 328*"), the effective address of the data access that caused the exception is contained in the data exception address register (DEAR). This address is used by system software to identify the address that caused the exception. Also, SRR0 contains the address of the instruction that caused the data storage interrupt and its value is used to identify the address to return to when system software has resolved the exception condition by writing a new TLB entry.

Permissions violation mas register settings

When either an instruction or data storage interrupt occurs, only the SPIDx and the SAS fields are automatically loaded into the MAS registers to assist in processing the interrupt. System software is required to then execute a **tlbsx** instruction to load the MAS registers with the TLB entry associated with the instruction address. System software may then make any desired changes to the TLB entry prior to writing it. *Table 193* describes the fields set in the MAS registers at exception time for instruction or data storage interrupts.

Table 193. MAS settings for permissions violation ISI or DSI

Field	Setting
SPID0	Set to PID0
SPID1	Set to PID1
SPID2	Set to PID2
SPID3	Set to PID3
SAS	Set to the value of MSR[IS] (for an instruction storage interrupt) or MSR[DS] (for a data storage interrupt) at the time of the exception (that is, the MSR that described the context that was running when the exception occurred).

All other MAS register values are unchanged.

MAS register updates for exceptions, tlbsx, and tlbre

Table 194 summarizes MAS register fields updates from the perspective of the EIS as a result of various events. Note that the implementations further define how certain MAS fields are set on exceptions.

Table 194. MMU assist register field updates—EIS definition

MAS*n* bit/field	Value loaded for each case						
	ITLB/DTLB error	tlbsx hit	tlbsx miss	ISI	DSI	tlbre	tlbwe
TLBSEL	TLBSELD	Which TLB hit	TLBSELD	—	—	—	—
ESEL	If TLBSELD supports NV: TLB0[NV] else, undefined	Number of entry that hit	If TLBSELD supports NV: TLB0[NV] else, undefined	—	—	—	—
NV	If TLBSELD supports NV: next NV (array) else, undefined	If TLBSEL supports NV: NV (array) else, undefined	If TLBSELD supports NV: next NV (array) else, undefined	—	—	If TLBSEL supports NV: NV (array) else, undefined	—
V	1	1	0	—	—	V (array)	—
IPROT	0	If TLB that hits supports IPROT: matched IPROT value; else, 0	0	—	—	If TLB that hits supports IPROT: matched IPROT value; else, 0	—
TID	PID*n* values selected by TIDSELD	TID (array)	SPID0	—	—	TID(array)	—
TS	MSR[IS/DS]	SAS	SAS	—	—	TS (array)	—
TSIZE[0–3]	TSIZED	TSIZE (array)	TSIZED	—	—	TSIZE (array)	—
EPN[32–51]	If TLBSELD has fixed page size: EPN of access else, undefined	EPN (array)	—	—	—	EPN (array)	—
X0, X1 WIMGE	X0D, X1D WIMGED	X0, X1 (array) WIMGE (array)	X0D, X1D WIMGED	—	—	X0, X1 (array) WIMGE (array)	—
RPN[32–51]	Zeros	RPN(array)	Zeros	—	—	RPN (array)	—
PERMIS	Zeros	PERMIS (array)	Zeros	—	—	PERMIS (array)	—
TLBSELD	—	—	—	—	—	—	—
TIDSELD	—	—	—	—	—	—	—
TSIZED	—	—	—	—	—	—	—
WIMGED	—	—	—	—	—	—	—

6 Instruction set

This chapter describes the following instructions:

- Book E instructions defined for 32-bit implementations. This includes instructions not implemented in all Book E devices.
- Instructions defined by the EIS, except for the instructions defined by the VLE extension. Full descriptions of these instructions are provided in *Chapter 13: VLE instruction set on page 891*."

6.1 Notation

The following definitions and notation are used throughout this chapter in the instruction descriptions.

Table 195. Notation conventions

Symbol	Meaning
X_p	Bit p of register/field X
X_{field}	The bits composing a defined field of X. For example, X_{sign}, X_{exp}, and X_{frac} represent the sign, exponent, and fractional value of a floating-point number X
$X_{p:q}$	Bits p through q of register/field X
$X_{p\ q\ ...}$	Bits p, q,... of register/field X
¬X	The one's complement of the contents of X
Field i	Bits 4×i through 4×i+3 of a register
.	As the last character of an instruction mnemonic, this character indicates that the instruction records status information in certain fields of the condition register as a side effect of execution, as described in *Chapter 2.5.1: Condition register (CR) on page 61*."
||	Describes the concatenation of two values. For example, 010 || 111 is the same as 010111.
x^n	x raised to the n^{th} power
^nx	The replication of x, n times (i.e., x concatenated to itself n–1 times). $^n0$ and $^n1$ are special cases: $^n0$ means a field of n bits with each bit equal to 0. Thus ^{50} is equivalent to 0b0_0000. $^n1$ means a field of n bits with each bit equal to 1. Thus ^{51} is equivalent to 0b1_1111.
/, //, ///,	A reserved field in an instruction or in a register. Each bit and field in instructions, in status and control registers (such as the XER or FPSCR), and in SPRs is either defined, allocated, or reserved, as described in *Chapter 3.2.1: Classes of instructions on page 135*."

6.2 Instruction fields

Table 196 describes instruction fields.

Table 196. Instruction field descriptions

Field	Description
AA (30)	Absolute address bit. 0 The immediate field represents an address relative to the current instruction address. For I-form branch instructions the effective address of the branch target is the value $^{32}0$ || (CIA+EXTS(LI||0b00))$_{32–63}$. For B-form branch instructions the effective address of the branch target is the value $^{32}0$ || (CIA+EXTS(BD||0b00))$_{32–63}$. For I-form branch extended instructions the effective address of the branch target is the value CIA+EXTS(LI||0b00). For B-form branch extended instructions the effective address of the branch target is the value CIA+EXTS(BD||0b00). 1 The immediate field represents an absolute address. For I-form branch instructions the effective address of the branch target is the value $^{32}0$ || EXTS(LI||0b00)$_{32–63}$. For B-form branch instructions the effective address of the branch target is the value $^{32}0$ || EXTS(BD||0b00)$_{32–63}$. For I-form branch extended instructions the effective address of the branch target is the value EXTS(LI||0b00). For B-form branch extended instructions the effective address of the branch target is the value EXTS(BD||0b00).
crbA (11–15)	Used to specify a condition register bit to be used as a source
crbB (16–20)	Used to specify a condition register bit to be used as a source
crD (6–8)	Used to specify a CR or FPSCR field to be used as a target
crS (11–13)	Used to specify a CR or FPSCR field to be used as a source
BI (11–15)	Used to specify a condition register bit to be used as the condition of a branch conditional instruction
BO (6–10)	Used to specify options for branch conditional instructions. See *Branch and flow control instructions on page 163*."
crbD (6–10)	Used to specify a CR or FPSCR bit to be used as a target
CT (6–10)	Used by cache touch instructions (**dcbt**, **dcbtst**, and **icbt**) to specify the target portion of the cache facility to place the prefetched data or instructions and is implementation-dependent
D (16–31)	Immediate field used to specify a 16-bit signed two's complement integer that is sign-extended to 64 bits
DCRN(16–20||11–15)	Used to specify a device control register for the **mtdcr** and **mfdcr** instructions
E (15)	Immediate field used to specify a 1-bit value used by **wrteei** to place in MSR[EE] (external input enable bit)
FM (7–14)	Field mask used to identify FPSCR fields that are to be updated by the **mtfsf** instruction

Table 196. Instruction field descriptions (continued)

Field	Description
frA (11–15)	Used to specify an FPR to be used as a source
frB (16–20)	Used to specify an FPR to be used as a source
frC (21–25)	Used to specify an FPR to be used as a source
frS (6–10)	Used to specify an FPR to be used as a source
frD (6–10)	Used to specify an FPR to be used as a target
CRM (12–19)	Field mask used to identify the condition register fields to be updated by the **mtcrf** instruction
LI (6–29)	Immediate field specifying a 24-bit signed two's complement integer that is concatenated on the right with 0b00 and sign-extended to 64 bits
LK (31)	LINK bit. Indicates whether the link register (LR) is set. 0Do not set the LR. 1Set the LR. The sum of the value 4 and the address of the branch instruction is placed into the LR.
MB (21–25) and ME (26–30)	Fields used in M-form rotate instructions to specify a 64-bit mask consisting of 1 bits from bit MB+32 through bit ME+32 inclusive and 0 bits elsewhere.
MO (6–10)	Used to specify the subset of memory accesses ordered by a Memory Barrier instruction (**mbar**).
NB (16–20)	Used to specify the number of bytes to move in an immediate Move Assist instruction
OPCD (0–5)	Primary opcode field
rA (11–15)	Used to specify a GPR to be used as a source or as a target
rB (16–20)	Used to specify a GPR to be used as a source
Rc (31)	Record bit. 0Do not alter the condition register. 1Set condition register field 0 or field 1.
RS (6–10)	Used to specify a GPR to be used as a source
rD (6–10)	Used to specify a GPR to be used as a target
SH (16–20)	Used to specify a shift amount in rotate word immediate and shift word immediate instructions
SIMM (16–31)	Immediate field used to specify a 16-bit signed integer
SPRN (16–20||11–15)	Used to specify an SPR for **mtspr** and **mfspr** instructions
TO (6–10)	Used to specify the conditions on which to trap. The encoding is described in *Table 92: Trap instructions on page 170*."
U (16–19)	Immediate field used as the data to be placed into a field in the FPSCR
UIMM (16–31)	Immediate field used to specify a 16-bit unsigned integer
XO (21–29, 21–30, 22–30, 26–30, 27–29, 27–30, 28–31)	Extended opcode field

6.3 Description of instruction operations

The operation of most instructions is described by a series of statements using a semiformal language at the register transfer level (RTL), which uses the general notation given in *Table 195* and *Table 196* and the RTL-specific conventions in *Table 197*. See the example in *Figure 23*. Some of this notation is used in the formal descriptions of instructions.

The RTL descriptions cover the normal execution of the instruction, except that 'standard' setting of the condition register, integer exception register, and floating-point status and control register are not always shown. (Non-standard setting of these registers, such as the setting of condition register field 0 by the **stwcx.** instruction, is shown.) The RTL descriptions do not cover all cases in which exceptions may occur, or for which the results are boundedly undefined, and may not cover all invalid forms.

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

Table 197. RTL notation

Notation	Meaning
$\leftarrow$	Assignment
$\leftarrow_f$	Assignment in which the data may be reformatted in the target location
$\neg$	NOT logical operator (one's complement)
+	Two's complement addition
–	Two's complement subtraction, unary minus
×	Multiplication
÷	Division (yielding quotient)
$+_{dp}$	Floating-point addition, double precision
$-_{dp}$	Floating-point subtraction, double precision
$\times_{dp}$	Floating-point multiplication, double precision
$\div_{dp}$	Floating-point division quotient, double precision
$+_{sp}$	Floating-point addition, single precision
$-_{sp}$	Floating-point subtraction, single precision
$\times_{sf}$	Signed fractional multiplication. Result of multiplying two quantities having bit lengths *x* and *y* taking the least significant x+y–1 bits of the product and concatenating a 0 to the least significant bit forming a signed fractional result of x+y bits.
$\times_{si}$	Signed integer multiplication
$\times_{sp}$	Floating-point multiplication, single precision
$\div_{sp}$	Floating-point division, single precision
$\times_{fp}$	Floating-point multiplication to infinite precision (no rounding)
$\times_{ui}$	Unsigned integer multiplication
FPSquareRoot-Double(x)	Floating-point $\sqrt{x}$, result rounded to double-precision

Table 197. RTL notation (continued)

Notation	Meaning
FPSquareRoot-Single(x)	Floating-point $\sqrt{x}$, result rounded to single-precision
FPReciprocal-Estimate(x)	Floating-point estimate of $\frac{1}{x}$
FPReciprocal-SquareRoot-Estimate(x)	Floating-point estimate of $\frac{1}{\sqrt{x}}$
Allocate-DataCache-Block(x)	If the block containing the byte addressed by x does not exist in the data cache, allocate a block in the data cache and set the contents of the block to 0.
Flush-DataCache-Block(x)	If the block containing the byte addressed by x exists in the data cache and is dirty, the block is written to main memory and is removed from the data cache.
Invalidate-DataCache-Block(x)	If the block containing the byte addressed by x exists in the data cache, the block is removed from the data cache.
Store-DataCache-Block(x)	If the block containing the byte addressed by x exists the data cache and is dirty, the block is written to main memory but may remain in the data cache.
Prefetch-DataCache-Block(x,y)	If the block containing the byte addressed by x does not exist in the portion of the data cache specified by y, the block in memory is copied into the data cache.
Prefetch-ForStore-DataCache-Block(x,y)	If the block containing the byte addressed by x does not exist in the portion of the data cache specified by y, the block in memory is copied into the data cache and made exclusive to the processor executing the instruction.
ZeroDataCache-Block(x)	The contents of the block containing the byte addressed by x in the data cache is cleared.
Invalidate-Instruction-CacheBlock(x)	If the block containing the byte addressed by x is in the instruction cache, the block is removed from the instruction cache.
Prefetch-Instruction-CacheBlock(x,y)	If the block containing the byte addressed by x does not exist in the portion of the instruction cache specified by y, the block in memory is copied into the instruction cache.
=, ≠	Equals, Not Equals relations
<, ≤, >, ≥	Signed comparison relations
$<_u$, $>_u$	Unsigned comparison relations
?	Unordered comparison relation
&, |	AND, OR logical operators
⊕, ≡	Exclusive OR, Equivalence logical operators ((a≡b) = (a⊕¬b))
>>, <<	Shift right or left logical
ABS(x)	Absolute value of x
APID(x)	Returns an implementation-dependent information on the presence and status of the auxiliary processing extensions specified by x
CEIL(x)	Least integer ≥ x

Table 197. RTL notation (continued)

Notation	Meaning
DCREG(x)	Device control register x
DOUBLE(x)	Result of converting x from floating-point single format to floating-point double format
EXTS(x)	Result of extending x on the left with signed bits
EXTZ(x)	Result of extending x on the left with zeros
FPR(x)	Floating-point register x
GPR(x)	General-purpose register x
MASK(x, y)	Mask having 1s in bit positions x through y (wrapping if x>y) and 0s elsewhere
MEM(x,1)	Contents of the byte of memory located at address x
MEM(x,y) (for y={2,4,8})	Contents of y bytes of memory starting at address x. If big-endian memory, the byte at address x is the MSB and the byte at address x+y–1 is the LSB of the value being accessed. If little-endian memory, the byte at address x is the LSB and the byte at address x+y–1 is the MSB of the value being accessed.
MOD(x,y)	Modulo y of x (remainder of x divided by y)
ROTL32(x, y)	Result of rotating the value 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
SPREG(x)	Special-purpose register x
TRAP	Invoke a trap-type program interrupt
characterization	Reference to the setting of status bits in a standard way that is explained in the text
undefined	An undefined value. The value may vary between implementations and between different executions on the same implementation.
CIA	Current instruction address, the address of the instruction being described in RTL. Used by relative branches to set the next instruction address (NIA) and by branch instructions with LK=1 to set the LR. CIA does not correspond to any architected register.
NIA	Next instruction address, the 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, 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. NIA does not correspond to any architected register.
if … then … else …	Conditional execution, indenting shows range; else is optional

Table 197. RTL notation (continued)

Notation	Meaning
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.

Precedence rules for RTL operators are summarized in *Table 198*. 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 198. 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
←	None

6.3.1 SPE APU saturation and bit-reverse models

For saturation and bit reversal, the pseudo RTL is provided here to more accurately describe those functions that are referenced in the instruction pseudo RTL.

Saturation

```
SATURATE(overflow, carry, saturated_underflow, saturated_overflow, value)
if overflow then
   if carry then
      return saturated_underflow
   else
      return saturated_overflow
else
   return value
```

Bit reverse

```
BITREVERSE(value)
result ← 0
mask ← 1
shift ← 31
cnt ← 32
while cnt > 0 then do
   t ← data & 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
```

6.3.2 Embedded floating-point conversion models

The embedded floating-point instructions defined by the signal processing engine (SPE) APU and the single-precision floating-point (SPFP) APUs contain floating-point conversion to and from integer and fractional type instructions. The floating-point to-and-from non–floating-point conversion model pseudo RTL is provided here as a group of functions that is called from the individual instruction pseudo RTL descriptions.

Table 199. Conversion models

Function	Name	Reference
Common functions		
Round a 32-bit value	Round32(fp,guard,sticky)	*on page 339*
Round a 64-bit value	Round64(fp,guard,sticky)	*Chapter*
Signal floating-point error	SignalFPError	*on page 339*
Is a 32-bit value a NaN or Infinity?	Isa32NaNorInfinity(fp)	*on page 339*
Floating-point conversions		
Convert from single-precision floating-point to integer word with saturation	CnvtFP32ToI32Sat(fp,signed,upper_lower,round,fractional)	*on page 351*
Convert from double-precision floating-point to integer word with saturation	CnvtFP64ToI32Sat(fp,signed,round,fractional)	*on page 353*
Convert from double-precision floating-point to integer double word with saturation	CnvtFP64ToI64Sat(fp,signed,round)	*on page 355*
Convert to single-precision floating-point from integer word with saturation	CnvtI32ToFP32Sat(v,signed,upper_lower,fractional)	*on page 356*

Table 199. Conversion models

Function	Name	Reference
Convert to double-precision floating-point from integer double word with saturation	CnvtI64ToFP64Sat(v,signed)	*on page 358*
Integer Saturate		
Integer saturate	SATURATE(ovf,carry,neg_sat,pos_sat,value)	*Chapter*

Common embedded floating-point functions

This section includes common functions used by the functions in subsequent sections.

```
32-Bit NaN or Infinity Test
// Determine if fp value is a NaN or Infinity
Isa32NaNorInfinity(fp)
return (fp_exp = 255)
Isa32NaN(fp)
return ((fp_exp = 255) & (fp_frac ≠ 0))
Isa32Infinity(fp)
return ((fp_exp = 255) & (fp_frac = 0))
// Determine if fp value is denormalized
Isa32Denorm(fp)
return ((fp_exp = 0) & (fp_frac ≠ 0))
// Determine if fp value is a NaN or Infinity
Isa64NaNorInfinity(fp)
return (fp_exp = 2047)
Isa64NaN(fp)
return ((fp_exp = 2047) & (fp_frac ≠ 0))
Isa64Infinity(fp)
return ((fp_exp = 2047) & (fp_frac = 0))
// Determine if fp value is denormalized
Isa64Denorm(fp)
return ((fp_exp = 0) & (fp_frac ≠ 0))


Signal Floating-Point Error
// Signal a Floating-Point Error in the SPEFSCR
SignalFPError(upper_lower, bits)
if (upper_lower = UPPER) then
   bits ← bits << 15
SPEFSCR ← SPEFSCR | bits
bits ← (FG | FX)
if (upper_lower = UPPER) then
   bits ← bits << 15
SPEFSCR ← SPEFSCR & ¬bits


Round a 32-Bit Value
// Round a 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 Value
// Round a 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
```

Convert from single-precision floating-point to integer word with saturation

```
// Convert 32-bit floating point to integer/factional
//   signed = SIGN or UNSIGN
//   upper_lower = UPPER or LOWER
//   round = ROUND or TRUNC
//   fractional = F (fractional) or I (integer)

CnvtFP32ToI32Sat(fp, signed, upper_lower, round, fractional)

FP32format fp;

if (Isa32NaNorInfinity(fp)) then   // SNaN, QNaN, +-INF
    SignalFPError(upper_lower, FINV)
    if (Isa32NaN(fp)) then
        return 0x00000000    // all NaNs
    if (signed = SIGN) 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 = UNSIGN) & (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 = SIGN) 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 = SIGN) then
            shift ← shift + 1

    if (fp_exp > max_exp) then
        SignalFPError(upper_lower, FOVF)  // overflow
        if (signed = SIGN) then
            if (fp_sign = 1) then
                return 0x80000000
            else
                return 0x7fffffff
        else
            return 0xffffffff

    result ← 0b1 || fp_frac || 0b00000000  // add U to frac
    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 = UPPER) 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 = ROUND) & (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 = SIGN) then
   if (fp_sign = 1) then
      result ← ¬result + 1

return result
```

Convert from double-precision floating-point to integer word with saturation

```
// Convert 64-bit floating point to integer/fractional
//   signed = SIGN or UNSIGN
//   round = ROUND or TRUNC
//   fractional = F (fractional) or I (integer)

CnvtFP64ToI32Sat(fp, signed, round, fractional)

FP64format fp;

if (Isa64NaNorInfinity(fp)) then   // SNaN, QNaN, +-INF
   SignalFPError(LOWER, FINV)
   if (Isa64NaN(fp)) then
      return 0x00000000    // all NaNs
   if (signed = SIGN) 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(LOWER, FINV)
   return 0x00000000    // regardless of sign

if ((signed = UNSIGN) & (fp_sign = 1)) then
   SignalFPError(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 ← 1054
   shift ← 1054 - fp_exp
   if (signed ← SIGN) 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 = SIGN) then
      shift ← shift + 1

if (fp_exp > max_exp) then
   SignalFPError(LOWER, FOVF)  // overflow
   if (signed = SIGN) 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 = ROUND) & (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 = SIGN) then
   if (fp_sign = 1) then
      result ← ¬result + 1

return result
```

Convert from double-precision floating-point to integer double word with saturation

```
// Convert 64-bit floating point to integer/fractional
//   signed = SIGN or UNSIGN
//   round = ROUND or TRUNC

CnvtFP64ToI64Sat(fp, signed, round)

FP64format fp;

if (Isa64NaNorInfinity(fp)) then   // SNaN, QNaN, +-INF
   SignalFPError(LOWER, FINV)
   if (Isa64NaN(fp)) then
      return 0x00000000_00000000    // all NaNs
   if (signed = SIGN) 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(LOWER, FINV)
   return 0x00000000_00000000    // regardless of sign

if ((signed = UNSIGN) & (fp_sign = 1)) then
   SignalFPError(LOWER, 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 = SIGN) then
   if ((fp_exp ≠ 1086) | (fp_frac ≠ 0) | (fp_sign ≠ 1)) then
      max_exp ← max_exp - 1

if (fp_exp > max_exp) then
   SignalFPError(LOWER, FOVF)  // overflow
   if (signed = SIGN) then
      if (fp_sign = 1) then
         return 0x80000000_00000000
      else
         return 0x7fffffff_ffffffff
   else
      return 0xffffffff_ffffffff

result ← 0b1 || fp_frac || 0b00000000000 // add U to frac
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 = ROUND) & (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 = SIGN) then
    if (fp_sign = 1) then
        result ← ¬result + 1

return result
```

Convert to single-precision floating-point from integer word with saturation

```
// Convert from integer/factional to 32-bit floating point
//   signed = SIGN or UNSIGN
//   upper_lower = UPPER or LOWER
//   fractional = F (fractional) or I (integer)

CnvtI32ToFP32Sat(v, signed, upper_lower, fractional)

FP32format result;

result_sign ← 0
if (v = 0) then
    result ← 0
    if (upper_lower = UPPER) then
        SPEFSCR_FGH ← 0
        SPEFSCR_FXH ← 0
    else
        SPEFSCR_FG ← 0
        SPEFSCR_FX ← 0
else
    if (signed = SIGN) then
        if (v_0 = 1) then
            v ← ¬v + 1
            result_sign ← 1
    if (fractional = F) then    // fractional bit pos alignment
        maxexp ← 127
        if (signed = UNSIGN) then
```

```
        maxexp ← maxexp - 1
    else
      maxexp ← 158    // integer bit pos alignment
    sc ← 0
    while (v0 = 0)
      v ← v << 1
      sc ← sc + 1
    v0 ← 0    // clear U bit
    resultexp ← maxexp - sc
    guard ← v24
    sticky ← (v25:31 ≠ 0)

    // Report sticky and guard bits

    if (upper_lower = UPPER) then
      SPEFSCRFGH ← guard
      SPEFSCRFXH ← sticky
    else
      SPEFSCRFG ← guard
      SPEFSCRFX ← sticky

    if (guard | sticky) then
      SPEFSCRFINXS ← 1

  // Round the result

    resultfrac ← v1:23
    result ← Round32(result, guard, sticky)

  return result
```

Convert to double-precision floating-point from integer word with saturation

```
  // Convert from integer/factional to 64-bit floating point
  //   signed = SIGN or UNSIGN
  //   fractional = F (fractional) or I (integer)

  CnvtI32ToFP64Sat(v, signed, fractional)

  FP64format result;

  resultsign ← 0
  if (v = 0) then
    result ← 0
    SPEFSCRFG ← 0
    SPEFSCRFX ← 0
  else
    if (signed = SIGN) then
      if (v[0] = 1) then
        v ← ¬v + 1
        resultsign ← 1
    if (fractional = F) then    // fractional bit pos alignment
      maxexp ← 1023
      if (signed = UNSIGN) then
        maxexp ← maxexp - 1
```

```
    else
       maxexp ← 1054    // integer bit pos alignment
    sc ← 0
    while (v0 = 0)
       v ← v << 1
       sc ← sc + 1
    v0 ← 0    // clear U bit
    resultexp ← maxexp - sc
// Report sticky and guard bits
    SPEFSCRFG ← 0
    SPEFSCRFX ← 0

    resultfrac ← v1:31 || 21 0
return result
```

Convert to double-precision floating-point from integer double word with saturation

```
// Convert from 64 integer to 64-bit floating point
//   signed = SIGN or UNSIGN

CnvtI64ToFP64Sat(v, signed)

FP64format result;

resultsign ← 0
if (v = 0) then
   result ← 0
   SPEFSCRFG ← 0
   SPEFSCRFX ← 0
else
   if (signed = SIGN) then
      if (v0 = 1) then
         v ← ¬v + 1
         resultsign ← 1
   maxexp ← 1054
   sc ← 0

   while (v0 = 0)
      v ← v << 1
      sc ← sc + 1
   v0 ← 0    // clear U bit
   resultexp ← maxexp - sc
   guard ← v53
   sticky ← (v54:63 ≠ 0)

// Report sticky and guard bits

   SPEFSCRFG ← guard
   SPEFSCRFX ← sticky
   if (guard | sticky) then
      SPEFSCRFINXS ← 1
```

```
// Round the result

   result_frac ← v_1:52
   result ← Round64(result, guard, sticky)

return result
```

6.3.3 Integer saturation models

```
// Saturate after addition

SATURATE(ovf, carry, neg_sat, pos_sat, value)

if ovf then
   if carry then
      return neg_sat
   else
      return pos_sat
else
   return value
```

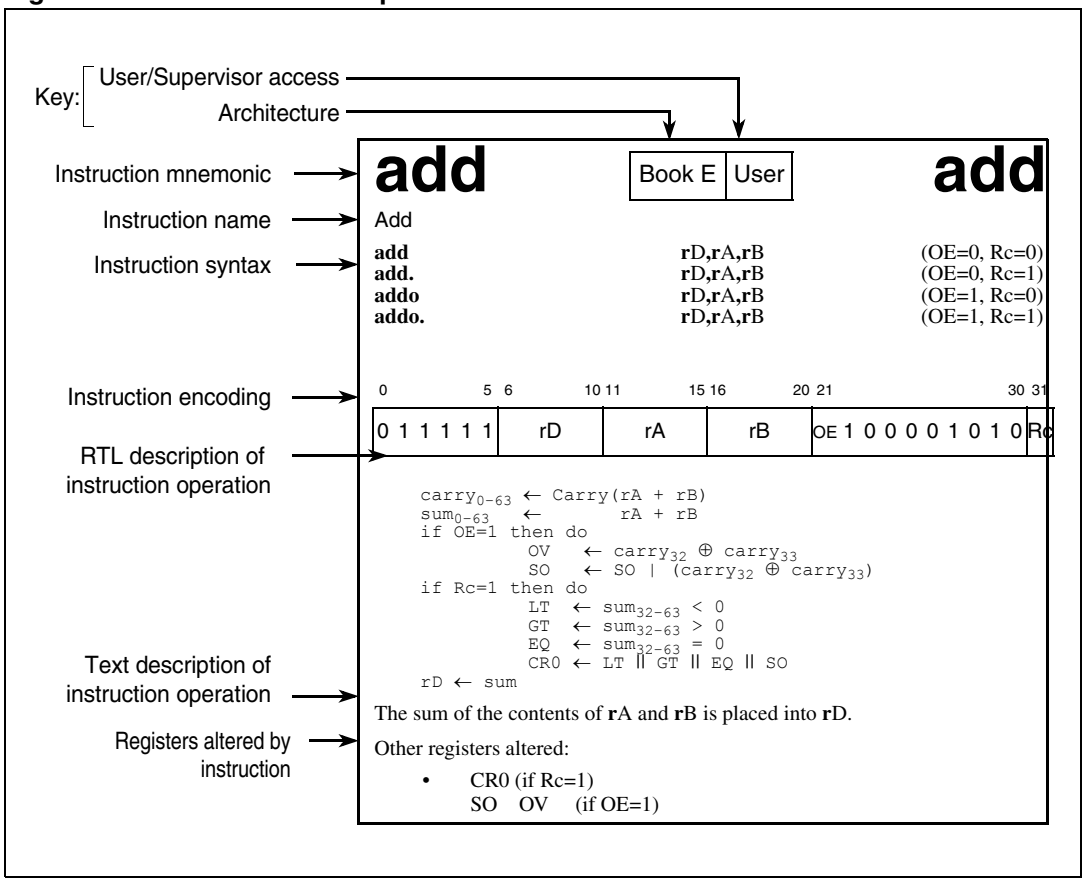
6.3.4 Embedded floating-point results

Appendix E: Embedded floating-point results on page 1156," summarizes results of various types of SPE and SPFP floating-point operations on various combinations of input operands.

6.4 Instruction set

The rest of this chapter describes individual instructions, which are listed in alphabetical order by mnemonic. *Figure 23* shows the format for instruction description pages.

Figure 23. Instruction description

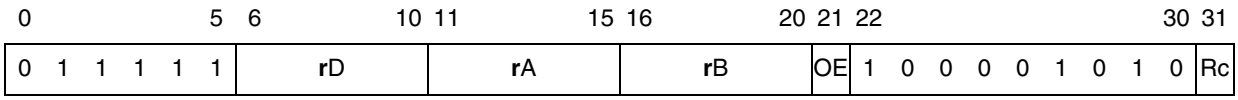


Note: The execution unit that executes the instruction may not be the same for all processors.

add | Book E | User | **add**

Add

add	**r**D,**r**A,**r**B	(OE=0, Rc=0)
add.	**r**D,**r**A,**r**B	(OE=0, Rc=1)
addo	**r**D,**r**A,**r**B	(OE=1, Rc=0)
addo.	**r**D,**r**A,**r**B	(OE=1, Rc=1)



$carry_{0:63} \leftarrow Carry(rA + rB)$
$sum_{0:63} \leftarrow rA + rB$
if OE=1 then do
$OV \leftarrow carry_{32} \oplus carry_{33}$
$SO \leftarrow SO \mid (carry_{32} \oplus carry_{33})$
if Rc=1 then do
$LT \leftarrow sum_{32:63} < 0$
$GT \leftarrow sum_{32:63} > 0$
$EQ \leftarrow sum_{32:63} = 0$
$CR0 \leftarrow LT \mid\mid GT \mid\mid EQ \mid\mid SO$
$rD \leftarrow sum$

$carry_{0:63} \leftarrow Carry(rA + rB)$
$sum_{0:63} \leftarrow rA + rB$
if OE=1 then do
$OV \leftarrow carry_{32} \oplus carry_{33}$
$SO \leftarrow SO \mid (carry_{32} \oplus carry_{33})$
if Rc=1 then do
$LT \leftarrow sum_{32:63} < 0$
$GT \leftarrow sum_{32:63} > 0$
$EQ \leftarrow sum_{32:63} = 0$
$CR0 \leftarrow LT \mid\mid GT \mid\mid EQ \mid\mid SO$
$rD \leftarrow sum$

The sum of the contents of **r**A and **r**B is placed into **r**D.

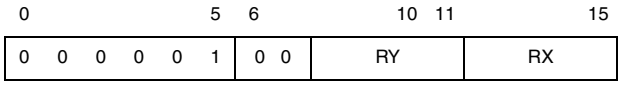
Other registers altered:

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

_add*x* | VLE | User | _ **add*x***

Add

se_add ‘ **rX,rY**



$sum_{32:63} \leftarrow GPR(RX) + GPR(RY)$

$GPR(RX) \leftarrow sum_{32:63}$

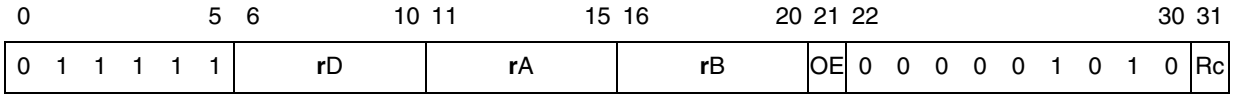
The sum of the contents of GPR(**r**X) and the contents of GPR(**r**Y) is placed into GPR(**r**X).

Special Registers Altered: None

addc | Book E | User | **addc**

Add Carrying

addc	**r**D,**r**A,**r**B	(OE=0, Rc=0)
addc.	**r**D,**r**A,**r**B	(OE=0, Rc=1)
addco	**r**D,**r**A,**r**B	(OE=1, Rc=0)
addco.	**r**D,**r**A,**r**B	(OE=1, Rc=1)



$carry_{0:63} \leftarrow Carry(rA + rB)$
$sum_{0:63} \leftarrow rA + rB$
if OE=1 then do
$OV \leftarrow carry_{32} \oplus carry_{33}$
$SO \leftarrow SO \mid (carry_{32} \oplus carry_{33})$
if Rc=1 then do
$LT \leftarrow sum_{32:63} < 0$
$GT \leftarrow sum_{32:63} > 0$
$EQ \leftarrow sum_{32:63} = 0$
$CR0 \leftarrow LT \mid\mid GT \mid\mid EQ \mid\mid SO$
$rD \leftarrow sum$
$CA \leftarrow carry_{32}$

The sum of the contents of **r**A and **r**B is placed into **r**D.

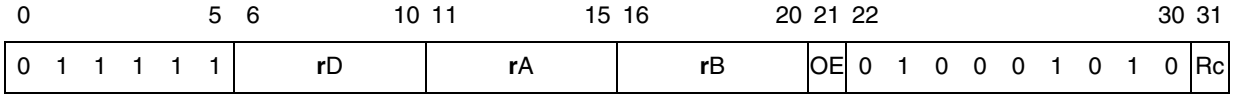
Other registers altered:

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

adde | Book E | User | **adde**

Add Extended

adde	**r**D,**r**A,**r**B	(OE=0, Rc=0)
adde.	**r**D,**r**A,**r**B	(OE=0, Rc=1)
addeo	**r**D,**r**A,**r**B	(OE=1, Rc=0)
addeo.	**r**D,**r**A,**r**B	(OE=1, Rc=1)



if E=0 then Cin ← CA
$carry_{0:63}$ ← Carry(rA + rB + Cin)
$sum_{0:63}$ ← rA + rB + Cin
if OE=1 then do
OV ← $carry_{32} \oplus carry_{33}$
SO ← SO | ($carry_{32} \oplus carry_{33}$)
if Rc=1 then do
LT ← $sum_{32:63} < 0$
GT ← $sum_{32:63} > 0$
EQ ← $sum_{32:63} = 0$
CR0 ← LT || GT || EQ || SO
rD ← sum
CA ← $carry_{32}$

For **adde**[**o**][**.**], the sum of the contents of **r**A, the contents of **r**B, and CA is placed into **r**D.

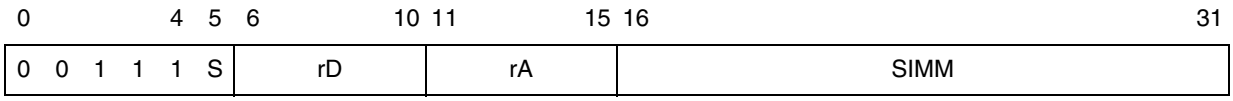
Other registers altered:

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

addi | Book E | User | **addi**

Add immediate [shifted]

addi **r**D,**r**A,SIMM (S=0)
addis **r**D,**r**A,SIMM (S=1)



if rA=0 then a ← $^{64}0$ else a ← rA
if s=0 then b ← EXTS(SIMM)
if s=1 then b ← EXTS(SIMM || $^{16}0$)
rD ← a + b

If **addi** and **r**A=0, the sign-extended value of the SIMM field is placed into **r**D.

If **addi** and **r**A≠0, the sum of the contents of **r**A and the sign-extended value of field SIMM is placed into **r**D.

If **addis** and **r**A=0, the sign-extended value of the SIMM field, concatenated with 16 zeros, is placed into **r**D.

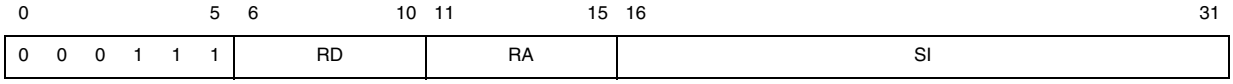
If **addis** and **r**A≠0, the sum of the contents of **r**A and the sign-extended value of the SIMM field concatenated with 16 zeros, is placed into **r**D.

Other registers altered: None

_addi*x* | VLE | User | **_addi*x***

Add [2 operand] Immediate [Shifted] [and Record]

e_add16i **r**D,**r**A,SI

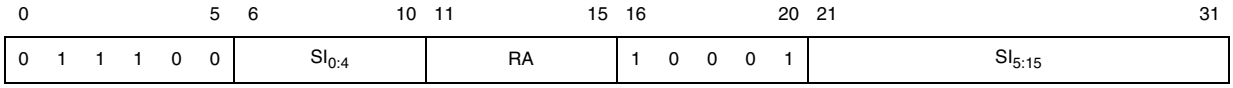


a ← GPR(RA)
b ← EXTS(SI)
GPR(RD) ← a + b

The sum of the contents of GPR(**r**A) and the sign-extended value of field SI is placed into GPR(**r**D).

Special Registers Altered: None

e_add2i. **r**A,SI

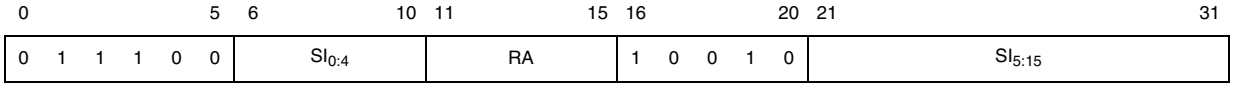


$SI \leftarrow SI_{0:4} \,||\, SI_{5:15}$
$sum_{32:63} \leftarrow GPR(RA) + EXTS(SI)$
$LT \leftarrow sum_{32:63} < 0$
$GT \leftarrow sum_{32:63} > 0$
$EQ \leftarrow sum_{32:63} = 0$
$CR0 \leftarrow LT \,||\, GT \,||\, EQ \,||\, SO$
$GPR(RA) \leftarrow sum_{32:63}$

The sum of the contents of GPR(**r**A) and the sign-extended value of SI is placed into GPR(**r**A).

Special Registers Altered: CR0

e_add2is **r**A,SI

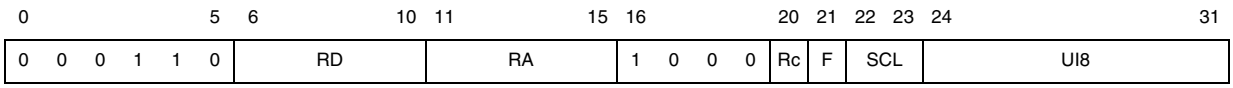


$SI \leftarrow SI_{0:4} \,||\, SI_{5:15}$
$sum_{32:63} \leftarrow GPR(RD) + (SI \,||\, {}^{16}0)$
$GPR(RA) \leftarrow sum_{32:63}$

The sum of the contents of GPR(**r**A) and the value of SI concatenated with 16 zeros is placed into GPR(**r**Aarav2006).

Special Registers Altered: None

e_addi **r**D,**r**A,SCI8 (Rc = 0)
e_addi. **r**D,**r**A,SCI8 (Rc = 1)

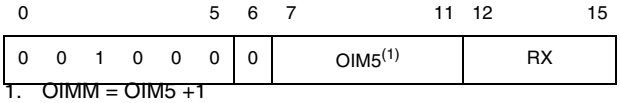


```
imm ← SCI8(F,SCL,UI8)
sum32:63  ←      GPR(RA) + imm
if Rc=1 then do
  LT  ← sum32:63 < 0
  GT  ← sum32:63 > 0
  EQ  ← sum32:63 = 0
  CR0 ← LT || GT || EQ || SO
GPR(RD) ← sum32:63
```

The sum of the contents of GPR(**r**A) and the value of SCI8 is placed into GPR(**r**D).

Special Registers Altered: CR0 (if Rc = 1)

se_addi **r**X,OIMM



1. OIMM = OIM5 +1

```
GPR(RX) ← GPR(RX) + (27 0 || OFFSET(OIM5))
```

The sum of the contents of GPR(**r**X) and the zero-extended offset value of OIM5 (a final value in the range 1–32), is placed into GPR(**r**X).

Special Registers Altered: None

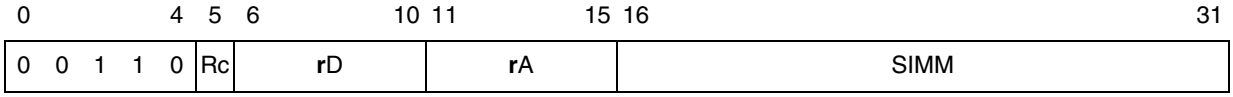
addic

Book E	User

addic

Add immediate carrying [and record]

addic **r**D,**r**A,SIMM (Rc=0)
addic. **r**D,**r**A,SIMM (Rc=1)



$carry_{0:63} \leftarrow Carry(rA + EXTS(SIMM))$
$sum_{0:63} \leftarrow rA + EXTS(SIMM)$
if Rc=1 then do
  $LT \leftarrow sum_{32:63} < 0$
  $GT \leftarrow sum_{32:63} > 0$
  $EQ \leftarrow sum_{32:63} = 0$
  $CR0 \leftarrow LT \,||\, GT \,||\, EQ \,||\, SO$
$rD \leftarrow rA+EXTS(SIMM)$
$CA \leftarrow carry_{32}$

The sum of the contents of **r**A and the sign-extended value of the SIMM field is placed into **r**D.

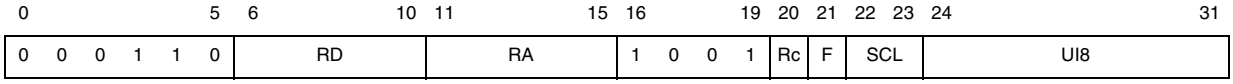
Other registers altered:

- CA
  CR0 (if Rc=1)

_addic*x* | VLE | User | **_addic*x***

Add Immediate Carrying [and Record]

e_addic **r**D,**r**A,SCI8 (Rc = 0)
e_addic. **r**D,**r**A,SCI8 (Rc = 1)



imm ← SCI8(F,SCL,UI8)
$carry_{32:63}$ ← Carry(GPR(RA) + imm)
$sum_{32:63}$ ← GPR(RA) + imm
if Rc=1 then do
  LT ← $sum_{32:63}$ < 0
  GT ← $sum_{32:63}$ > 0
  EQ ← $sum_{32:63}$ = 0
  CR0 ← LT || GT || EQ || SO
GPR(RD) ← $sum_{32:63}$
CA ← $carry_{32}$

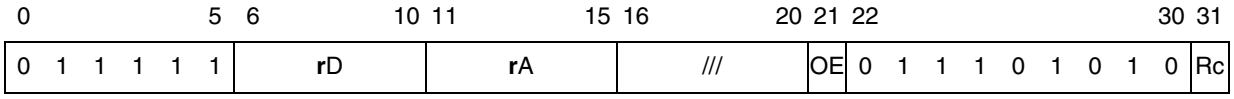
The sum of the contents of GPR(**r**A) and the value of SCI8 is placed into GPR(**r**D).

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

addme | Book E | User | **addme**

Add to minus one extended

addme	**r**D,**r**A	(OE=0, Rc=0)
addme.	**r**D,**r**A	(OE=0, Rc=1)
addmeo	**r**D,**r**A	(OE=1, Rc=0)
addmeo.	**r**D,**r**A	(OE=1, Rc=1)



if E=0 then Cin ← CA
$carry_{0:63}$ ← Carry(rA + Cin + 0xFFFF_FFFF_FFFF_FFFF)
$sum_{0:63}$ ← rA + Cin + 0xFFFF_FFFF_FFFF_FFFF
if OE=1 then do
OV ← $carry_{32} \oplus carry_{33}$
SO ← SO | ($carry_{32} \oplus carry_{33}$)
if Rc=1 then do
LT ← $sum_{32:63} < 0$
GT ← $sum_{32:63} > 0$
EQ ← $sum_{32:63} = 0$
CR0 ← LT || GT || EQ || SO
rD ← sum
CA ← $carry_{32}$

For **addme**[**o**][**.**], the sum of the contents of **r**A, CA, and $^{64}1$ is placed into **r**D.

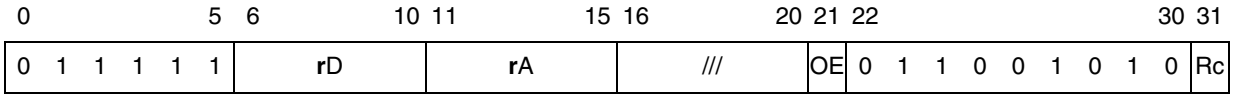
Other registers altered:

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

addze | Book E | User | **addze**

Add to zero extended

addze	**r**D,**r**A	(OE=0, Rc=0)
addze.	**r**D,**r**A	(OE=0, Rc=1)
addzeo	**r**D,**r**A	(OE=1, Rc=0)
addzeo.	**r**D,**r**A	(OE=1, Rc=1)



if E=0 then Cin ← CA
$carry_{0:63}$ ← Carry(rA + Cin)
$sum_{0:63}$ ← rA + Cin
if OE=1 then do
OV ← $carry_{32} \oplus carry_{33}$
SO ← SO | ($carry_{32} \oplus carry_{33}$)
if Rc=1 then do
LT ← $sum_{32:63} < 0$
GT ← $sum_{32:63} > 0$
EQ ← $sum_{32:63} = 0$
CR0 ← LT || GT || EQ || SO
rD ← sum
CA ← $carry_{32}$

For **addze**[**o**][**.**], the sum of the contents of **r**A and CA is placed into **r**D.

Other registers altered:

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

and

Book E	User

and

AND [Immediate [Shifted] | with Complement]

and **r**A,**r**S,**r**B (Rc=0)
and. **r**A,**r**S,**r**B (Rc=1)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	1	1	1	1	1	**r**S					**r**A					**r**B					0	0	0	0	0	1	1	1	0	0	Rc

andi. **r**A,**r**S,UIMM (S=0, Rc=1)
andis. **r**A,**r**S,UIMM (S=1, Rc=1)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	1	1	1	0	S	**r**S					**r**A					UIMM															

andc **r**A,**r**S,**r**B (Rc=0)
andc. **r**A,**r**S,**r**B (Rc=1)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	1	1	1	1	1	**r**S					**r**A					**r**B					0	0	0	0	1	1	1	1	0	0	Rc

if ‘andi.’ then b ← $^{48}0$ || UIMM
if ‘andis.’ then b ← $^{32}0$ || UIMM || $^{16}0$
if ‘and[.]’ then b ← rB
if ‘andc[.]’ then b ← ¬rB
$result_{0:63}$ ← rS & b
if Rc=1 then do
$LT \leftarrow result_{32:63} < 0$
$GT \leftarrow result_{32:63} > 0$
$EQ \leftarrow result_{32:63} = 0$
CR0 ← LT || GT || EQ || SO
rA ← result

For **andi.**, the contents of **r**S are ANDed with $^{48}0$ || UIMM.

For **andis.**, the contents of **r**S are ANDed with $^{32}0$ || UIMM || $^{16}0$.

For **and**[**.**], the contents of **r**S are ANDed with the contents of **r**B.

For **andc**[**.**], the contents of **r**S are ANDed with the one’s complement of the contents of **r**B.

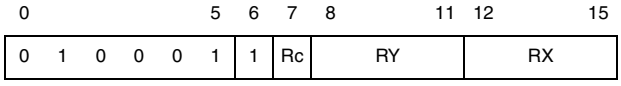
The result is placed into **r**A.

Other registers altered: CR0 (if Rc=1)

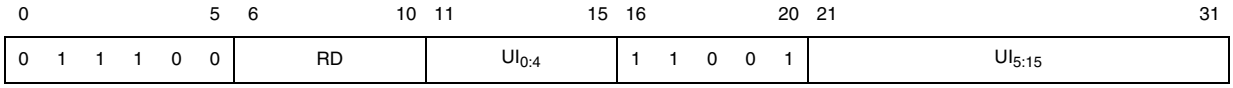
_and*x* | VLE | User | **_and*x***

AND [2 operand] [Immediate | with Complement] [and Record]

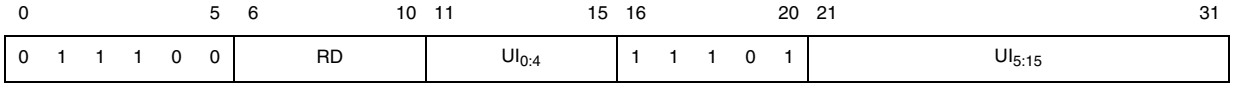
se_and **r**X,**r**Y (Rc = 0)
se_and. **r**X,**r**Y (Rc = 1)



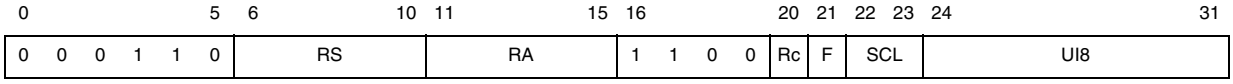
e_and2i. **r**D,UI



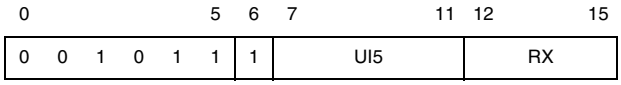
e_and2is. **r**D,UI



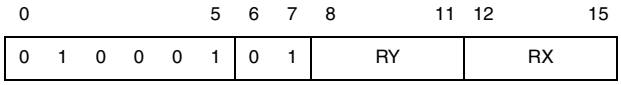
e_andi **r**A,**r**S,SCI8 (Rc = 0)
e_andi. **r**A,**r**S,SCI8 (Rc = 1)



se_andi **r**X,UI5



se_andc **r**X,**r**Y



if ‘e_andi[.]’ then b ← SCI8(F,SCL,UI8)
if ‘se_andi’ then b ← UI5
if ‘se_and[.]’ then b ← GPR(RY)
if ‘se_andc’ then b ← ¬GPR(RY)
if ‘e_and2i.’ then b ← $^{16}0$ || $UI_{0:4}$ || $UI_{5:15}$
if ‘e_and2is.’ then b ← $UI_{0:4}$ || $UI_{5:15}$ || $^{16}0$
$result_{32:63}$ ← GPR(RS or RD or RX) & b
if Rc=1 then do
  LT ← $result_{32:63}$ < 0
  GT ← $result_{32:63}$ > 0
  EQ ← $result_{32:63}$ = 0
  CR0 ← LT || GT || EQ || SO
if ‘se_and[ci]’ then GPR(RX) ← $result_{32:63}$ else GPR(RA or RD) ← $result_{32:63}$

For **e_andi**[**.**], the contents of GPR(**r**S) are ANDed with the value of SCI8.

For **e_and2i.**, the contents of GPR(**r**D) are ANDed with $^{16}0$ || UI.

For **e_and2is.**, the contents of GPR(**r**D) are ANDed with UI || $^{16}0$.

For **se_andi**, the contents of GPR(**r**X) are ANDed with the value of UI5.

For **se_and**[**.**], the contents of GPR(**r**X) are ANDed with the contents of GPR(**r**Y).

For **se_andc**, the contents of GPR(**r**X) are ANDed with the one's complement of the contents of GPR(**r**Y).

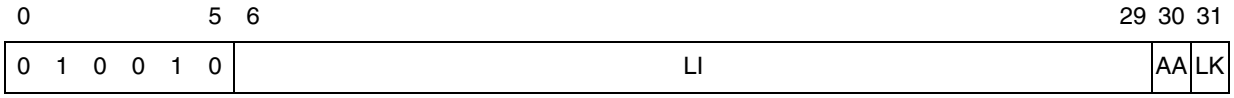
The result is placed into GPR(**r**A) or GPR(**r**X) (**se_and**[**ic**][**.**])

Special Registers Altered: CR0 (if Rc = 1)

b | Book E | User | **b**

Branch [and Link] [Absolute]

b	LI	(AA=0, LK=0)
ba	LI	(AA=1, LK=0)
bl	LI	(AA=0, LK=1)
bla	LI	(AA=1, LK=1)



if AA=1 then a ← $^{64}0$ else a ← CIA
if E=0 then NIA ← $^{32}0$ || (a + EXTS(LI||0b00))$_{32:63}$
if LK=1 then LR ← CIA + 4

The branch target effective address (BTEA) is calculated as follows:

- For 32-bit implementations, BTEA is bits 32–63 of the sum of the current instruction address (CIA), or 32 zeros if AA=1, and the sign-extended value of the LI instruction field concatenated with 0b00

BTEA is the address of the next instruction to be executed.

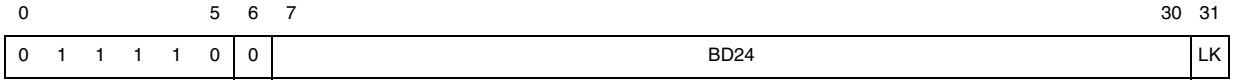
If LK=1, the sum CIA+4 is placed into the LR.

Other registers altered: LR (if LK=1)

_bx | VLE | User | **_bx**

Branch [and Link]

e_b BD24 (LK = 0)
e_bl BD24 (LK = 1)



a ← CIA
NIA ← (a + EXTS(BD24||0b0))$_{32:63}$
if LK=1 then LR ← CIA + 4

Let the BTEA be calculated as follows:

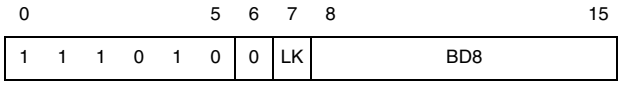
- For **e_b**[**l**], let BTEA be the sum of the CIA and the sign-extended value of the BD24 instruction field concatenated with 0b0.

The BTEA is the address of the next instruction to be executed.

If LK = 1, the sum CIA+4 is placed into the LR.

Special Registers Altered: LR (if LK = 1)

se_b BD8 (LK = 0)
se_bl BD8 (LK = 1)



a ← CIA
NIA ← (a + EXTS(BD8||0b0))$_{32:63}$
if LK=1 then LR ← CIA + 2

Let the BTEA be calculated as follows:

- For **se_b**[**l**], let BTEA be the sum of the CIA and the sign-extended value of the BD8 instruction field concatenated with 0b0.

The BTEA is the address of the next instruction to be executed.

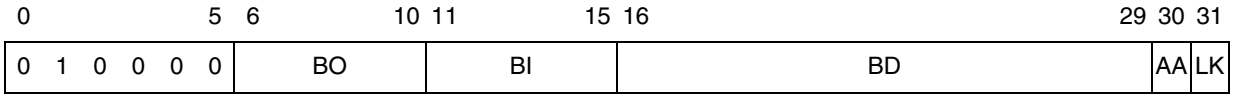
If LK = 1, the sum CIA+2 is placed into the LR.

Special Registers Altered: LR (if LK = 1)

bc | Book E | User | **bc**

Branch conditional [and link] [absolute]

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



if $\neg BO_2$ then $CTR_{32:63} \leftarrow CTR_{32:63} - 1$
ctr_ok $\leftarrow BO_2 \mid ((CTR_{32:63} \neq 0) \oplus BO_3)$
cond_ok $\leftarrow BO_0 \mid (CR_{BI+32} \equiv BO_1)$
if ctr_ok & cond_ok then
  if AA=1 then a $\leftarrow {}^{64}0$ else a $\leftarrow$ CIA
  if E=0 then NIA $\leftarrow {}^{32}0 \mid\mid (a + EXTS(BD \mid\mid 0b00))_{32:63}$
else NIA $\leftarrow$ CIA + 4
if LK=1 then LR $\leftarrow$ CIA + 4

The branch target effective address (BTEA) is calculated as follows:

- For 32-bit implementations, BTEA is bits 32–63 of the sum of the current instruction address (CIA), or 32 zeros if AA=1, and the sign-extended value of the LI instruction field concatenated with 0b00

The BO instruction field specifies any conditions that must be met for the branch to be taken, as defined in *Conditional branch control on page 167*." The sum BI+32 specifies the CR bit to be used.

The BI field specifies the CR bit used as the condition of the branch, as shown in *Table 200*.

Table 200. BI operand settings for CR fields

CR*n* Bits	CR Bits	BI	Description
CR0[0]	32	00000	Negative (LT)—Set when the result is negative.
CR0[1]	33	00001	Positive (GT)—Set when the result is positive (and not zero).
CR0[2]	34	00010	Zero (EQ)—Set when the result is zero.
CR0[3]	35	00011	Summary overflow (SO). Copy of XER[SO] at the instruction's completion.
CR1[0]	36	00100	Copy of FPSCR[FX] at the instruction's completion.
CR1[1]	37	00101	Copy of FPSCR[FEX] at the instruction's completion.
CR1[2]	38	00110	Copy of FPSCR[VX] at the instruction's completion.
CR1[3]	39	00111	Copy of FPSCR[OX] at the instruction's completion.
CR*n*[0]	40 44 48 52 56 60	01000 01100 10000 10100 11000 11100	Less than or floating-point less than (LT, FL). For integer compare instructions: **r**A < SIMM or **r**B (signed comparison) or **r**A < UIMM or **r**B (unsigned comparison). For floating-point compare instructions:**fr**A < **fr**B.
CR*n*[1]	41 45 49 53 57 61	01001 01101 10001 10101 11001 11101	Greater than or floating-point greater than (GT, FG). For integer compare instructions: **r**A > SIMM or **r**B (signed comparison) or **r**A > UIMM or **r**B (unsigned comparison). For floating-point compare instructions:**fr**A > **fr**B.
CR*n*[2]	42 46 50 54 58 62	01010 01110 10010 10110 11010 11110	Equal or floating-point equal (EQ, FE). For integer compare instructions: **r**A = SIMM, UIMM, or **r**B. For floating-point compare instructions: **fr**A = **fr**B.
CR*n*[3]	43 47 51 55 59 63	01011 01111 10011 10111 11011 11111	Summary overflow or floating-point unordered (SO, FU). For integer compare instructions, this is a copy of XER[SO] at the completion of the instruction. For floating-point compare instructions, one or both of **fr**A and **fr**B is a NaN.

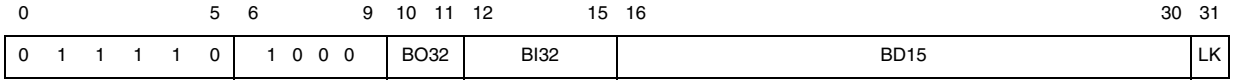
If the branch conditions are met, the BTEA is the address of the next instruction to be executed.
If LK=1, the sum CIA + 4 is placed into the LR.
Other registers altered:

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

_bc*x* | VLE | User | **_bc*x***

Branch Conditional [and Link]

e_bc BO32,BI32,BD15 (LK = 0)
e_bcl BO32,BI32,BD15 (LK = 1)



if $BO32_0$ then $CTR_{32:63} \leftarrow CTR_{32:63} - 1$
ctr_ok $\leftarrow \neg BO32_0 \mid ((CTR_{32:63} \neq 0) \oplus BO32_1)$
cond_ok $\leftarrow BO32_0 \mid (CR_{BI32+32} \equiv BO32_1)$
if ctr_ok & cond_ok then
NIA $\leftarrow (CIA + EXTS(BD15 \parallel 0b0))_{32:63}$
else NIA $\leftarrow CIA + 4$
if LK=1 then LR $\leftarrow CIA + 4$

Let the BTEA be calculated as follows:

- For **e_bc**[**l**], let BTEA be the sum of the CIA and the sign-extended value of the BD15 instruction field concatenated with 0b0.

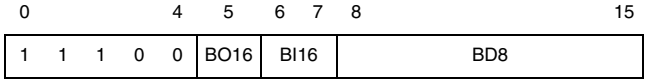
BO32 specifies any conditions that must be met for the branch to be taken, as defined in *Chapter 12.2.2: Branch instructions on page 864*." The sum BI32+32 specifies the CR bit. Only CR[32–47] may be specified.

If the branch conditions are met, the BTEA is the address of the next instruction to be executed.

If LK = 1, the sum CIA + 4 is placed into the LR.

Special Registers Altered: CTR (if $BO32_0 = 1$)
LR (if LK = 1)

se_bc BO16,BI16,BD8



cond_ok $\leftarrow (CR_{BI16+32} \equiv BO16)$
if cond_ok then
NIA $\leftarrow (CIA + EXTS(BD8 \parallel 0b0))_{32:63}$
else NIA $\leftarrow CIA + 2$

Let the BTEA be calculated as follows:

- For **se_bc**, BTEA is the sum of the CIA and the sign-extended value of the BD8 instruction field concatenated with 0b0.

BO16 specifies any conditions that must be met for the branch to be taken, as defined in *Chapter 12.2.2: Branch instructions*." The sum BI16+32 specifies CR bit; only CR[32–35] may be specified.

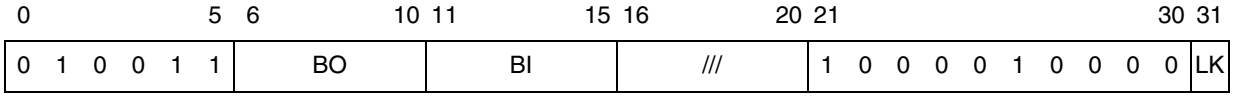
If the branch conditions are met, the BTEA is the address of the next instruction to be executed.

Special Registers Altered: None

bcctr | Book E | User | **bcctr**

Branch conditional to count register [and link]

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



cond_ok ← BO_0 | (CR_{BI+32} ≡ BO_1)
if cond_ok & E=0 then NIA ← $^{32}0$ || $CTR_{32:61}$ || 0b00
if ¬cond_ok then NIA ← CIA + 4
if LK=1 then LR ← CIA + 4

The branch target effective address (BTEA) is calculated as follows:

- For **bcctr**[**l**], BTEA is the contents of CTR[32–61] concatenated with 0b00.

BO specifies conditions that must be met for the branch to be taken. BI+32 specifies the CR bit to be used; see *Table 201*.

Table 201. BI operand settings for CR fields

CR*n* Bits	CR Bits	BI	Description
CR0[0]	32	00000	Negative (LT)—Set when the result is negative.
CR0[1]	33	00001	Positive (GT)—Set when the result is positive (and not zero).
CR0[2]	34	00010	Zero (EQ)—Set when the result is zero.
CR0[3]	35	00011	Summary overflow (SO). Copy of XER[SO] at the instruction's completion.
CR1[0]	36	00100	Copy of FPSCR[FX] at the instruction's completion.
CR1[1]	37	00101	Copy of FPSCR[FEX] at the instruction's completion.
CR1[2]	38	00110	Copy of FPSCR[VX] at the instruction's completion.
CR1[3]	39	00111	Copy of FPSCR[OX] at the instruction's completion.
CR*n*[0]	40 44 48 52 56 60	01000 01100 10000 10100 11000 11100	Less than or floating-point less than (LT, FL). For integer compare instructions: **r**A < SIMM or **r**B (signed comparison) or **r**A < UIMM or **r**B (unsigned comparison). For floating-point compare instructions:**fr**A < **fr**B.
CR*n*[1]	41 45 49 53 57 61	01001 01101 10001 10101 11001 11101	Greater than or floating-point greater than (GT, FG). For integer compare instructions: **r**A > SIMM or **r**B (signed comparison) or **r**A > UIMM or **r**B (unsigned comparison). For floating-point compare instructions: **fr**A > **fr**B.
CR*n*[2]	42 46 50 54 58 62	01010 01110 10010 10110 11010 11110	Equal or floating-point equal (EQ, FE). For integer compare instructions: **r**A = SIMM, UIMM, or **r**B. For floating-point compare instructions: frA = frB.
CR*n*[3]	43 47 51 55 59 63	01011 01111 10011 10111 11011 11111	Summary overflow or floating-point unordered (SO, FU). For integer compare instructions, this is a copy of XER[SO] at the completion of the instruction. For floating-point compare instructions, one or both of **fr**A and **fr**B is a NaN.

If the condition is met, the BTEA is the address of the next instruction to be executed.

If LK=1, the sum CIA + 4 is placed into the LR.

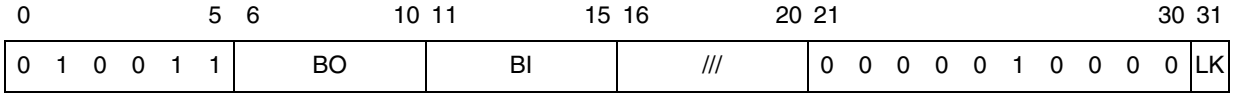
If the decrement and test CTR option is specified (BO[2]=0), the instruction form is invalid.

Other registers altered: LR (if LK=1)

bclr | Book E | User | **bclr**

Branch conditional to link register [and link]

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



if $\neg BO_2$ then $CTR_{32:63} \leftarrow CTR_{32:63} - 1$
ctr_ok $\leftarrow BO_2 \mid ((CTR_{32:63} \neq 0) \oplus BO_3)$
cond_ok $\leftarrow BO_0 \mid (CR_{BI+32} \equiv BO_1)$
if ctr_ok & cond_ok & E=0 then NIA $\leftarrow {}^{32}0 \parallel LR_{32:61} \parallel 0b00$
if $\neg$(ctr_ok & cond_ok) then NIA $\leftarrow$ CIA + 4
if LK=1 then LR $\leftarrow$ CIA + 4

The branch target effective address (BTEA) is calculated as follows:

- For **bclr**[**l**], BTEA is the contents of LR[32–61] concatenated with 0b00.

The BO field specifies any conditions that must be met for the branch to be taken, as defined in *Conditional branch control on page 167*." The sum BI+32 specifies the CR bit to be used.
The BI field specifies the CR bit used as the condition of the branch, as shown in *Table 202*.

Table 202. BI operand settings for CR fields

CR*n* Bits	CR Bits	BI	Description
CR0[0]	32	00000	Negative (LT)—Set when the result is negative.
CR0[1]	33	00001	Positive (GT)—Set when the result is positive (and not zero).
CR0[2]	34	00010	Zero (EQ)—Set when the result is zero.
CR0[3]	35	00011	Summary overflow (SO). Copy of XER[SO] at the instruction's completion.
CR1[0]	36	00100	Copy of FPSCR[FX] at the instruction's completion.
CR1[1]	37	00101	Copy of FPSCR[FEX] at the instruction's completion.
CR1[2]	38	00110	Copy of FPSCR[VX] at the instruction's completion.
CR1[3]	39	00111	Copy of FPSCR[OX] at the instruction's completion.
CR*n*[0]	40 44 48 52 56 60	01000 01100 10000 10100 11000 11100	Less than or floating-point less than (LT, FL). For integer compare instructions: **r**A < SIMM or **r**B (signed comparison) or **r**A < UIMM or **r**B (unsigned comparison). For floating-point compare instructions: **fr**A < **fr**B.
CR*n*[1]	41 45 49 53 57 61	01001 01101 10001 10101 11001 11101	Greater than or floating-point greater than (GT, FG). For integer compare instructions: **r**A > SIMM or **r**B (signed comparison) or **r**A > UIMM or **r**B (unsigned comparison). For floating-point compare instructions: **fr**A > **fr**B.
CR*n*[2]	42 46 50 54 58 62	01010 01110 10010 10110 11010 11110	Equal or floating-point equal (EQ, FE). For integer compare instructions: **r**A = SIMM, UIMM, or **r**B. For floating-point compare instructions: **fr**A = **fr**B.
CR*n*[3]	43 47 51 55 59 63	01011 01111 10011 10111 11011 11111	Summary overflow or floating-point unordered (SO, FU). For integer compare instructions, this is a copy of XER[SO] at the completion of the instruction. For floating-point compare instructions, one or both of **fr**A and **fr**B is a NaN.

If the condition is met, the BTEA is the address of the next instruction to be executed.
If LK=1, the sum CIA + 4 is placed into the LR.
Other registers altered:

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

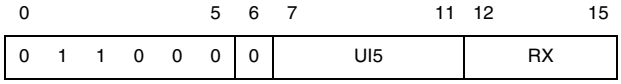
_bclri

VLE	User

_bclri

Bit Clear Immediate

se_bclri **r**X,UI5



a ← UI5
b ← $^{a}1$ || 0 || $^{31-a}1$
$result_{32:63}$ ← GPR(RX) & b
GPR(RX) ← $result_{32:63}$

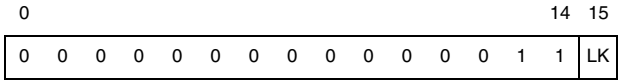
For **se_bclri**, the bit of GPR(**r**X) specified by the value of UI5 is cleared and all other bits in GPR(**r**X) remain unaffected.

Special Registers Altered: None

_bctr*x* | VLE | User | **_bctr*x***

Branch to Count Register [and Link]

se_bctr (LK = 0)
se_bctrl (LK = 1)



NIA ← $CTR_{32:62}$ || 0b0
if LK=1 then LR ← CIA + 2

Let the BTEA be calculated as follows:

- For **se_bctr[l]**, let BTEA be bits 32–62 of the contents of the CTR concatenated with 0b0.

The BTEA is the address of the next instruction to be executed.

If LK = 1, the sum CIA + 2 is placed into the LR.

Special Registers Altered: LR (if LK = 1)

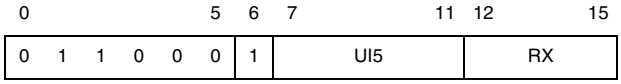
_bgeni

VLE	User

_bgeni

Bit Generate Immediate

se_bgeni **r**X,UI5



a ← UI5
b ← $^{a}0$ || 1 || $^{31-a}0$
GPR(RX) ← b

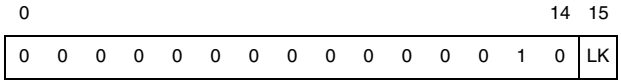
For **se_bgeni**, a constant value consisting of a single ‘1’ bit surrounded by ‘0’s is generated and the value is placed into GPR(**r**X). The position of the ‘1’ bit is specified by the UI5 field.

Special Registers Altered: None

_blr*x* | VLE | User | **_blr*x***

Branch to Link Register [and Link]

se_blr (LK = 0)
se_blrl (LK = 1)



NIA ← $LR_{32:62}$ || 0b0
if LK=1 then LR ← CIA + 2

Let the BTEA be calculated as follows:

- For **se_blr[l]**, let BTEA be bits 32–62 of the contents of the LR concatenated with 0b0.

The BTEA is the address of the next instruction to be executed.

If LK = 1, the sum CIA + 2 is placed into the LR.

Special Registers Altered: LR (if LK = 1)

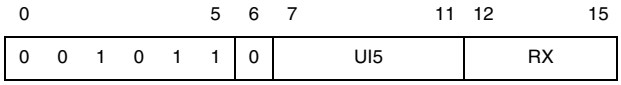
_bmaski

VLE	User

_bmaski

Bit Mask Generate Immediate

se_bmaski **r**X,UI5



a ← UI5
if a = 0 then b ← $^{32}1$ else b ← $^{32-a}0 \,||\, {}^{a}1$
GPR(RX) ← b

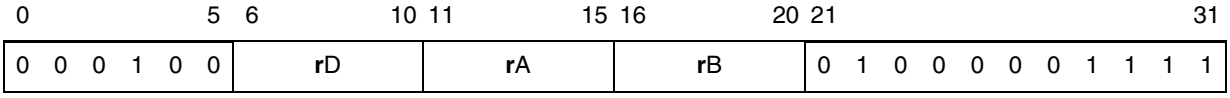
For **se_bmaski**, a constant value consisting of a mask of low-order '1' bits that is zero-extended to 32 bits is generated, and the value is placed into GPR(**r**X). The number of low-order '1' bits is specified by the UI5 field. If UI5 is 0b00000, a value of all '1's is generated

Special Registers Altered: None

brinc | SPE APU | User | **brinc**

Bit reversed increment

brinc **r**D,**r**A,**r**B



n ← MASKBITS // Imp dependent # of mask bits
mask ← $rB_{64-n:63}$ // Least sig. n bits of register
a ← $rA_{64-n:63}$
d ← bitreverse(1 + bitreverse(a | (¬ mask)))
rD ← $rA_{0:63-n}$ || (d & mask)

brinc provides a way for software to access FFT data in a bit-reversed manner. **r**A contains the index into a buffer that contains data on which FFT is to be performed. **r**B 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 half word 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. *Table 203* shows example values of masks for different data sizes and number of data.

Table 203. Data samples and sizes

Number of data samples	Data size			
	Byte	Half word	Word	Double word
8	000...00000111	000...00001110	000...000011100	000...0000111000
16	000...00001111	000...00011110	000...000111100	000...0001111000
32	000...00011111	000...00111110	000...001111100	000...0011111000
64	000...00111111	000...01111110	000...011111100	000...0111111000

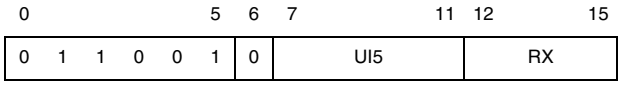
_bseti

VLE	User

_bseti

Bit Set Immediate

se_bseti **r**X,UI5



a ← UI5
b ← $^{a}0$ || 1 || $^{31-a}0$
result$_{32:63}$ ← GPR(RX) | b
GPR(RX) ← result$_{32:63}$

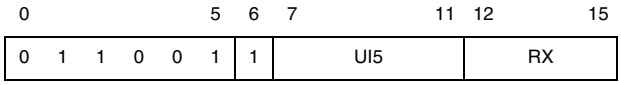
For **se_bseti**, the bit of GPR(**r**X) specified by the value of UI5 is set, and all other bits in GPR(**r**X) remain unaffected.

Special Registers Altered: None

_btsti VLE User **_btsti**

Bit Test Immediate

se_btsti **r**X,UI5



a ← UI5
b ← $^{a}0$ || 1 || $^{31-a}0$
c ← GPR(RX) & b
if c = $^{32}0$ then d ← 0b001 else d ← 0b010
$CR_{0:3}$ ← d || XER_{SO}

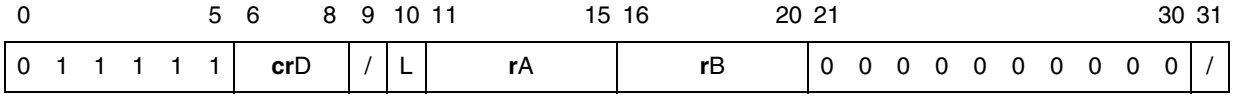
For **se_btsti**, the bit of GPR(**r**X) specified by the value of UI5 is tested for equality to '1'. The result of the test is recorded in the CR. EQ is set if the tested bit is clear, LT is cleared, and GT is set to the inverse value of EQ.

Special Registers Altered: CR[0–3]

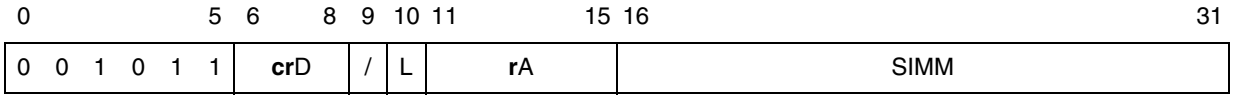
cmp | Book E | User | **cmp**

Compare [immediate]

cmp **cr**D,L,**r**A,**r**B



cmpi **cr**D,L,**r**A,SIMM



if L=0 then a ← EXTS($rA_{32:63}$)
else a ← rA
if 'cmpi' then b ← EXTS(SIMM)
if 'cmp' & L=0 then b ← EXTS($rB_{32:63}$)
if 'cmp' & L=1 then b ← rB
if a < b then c ← 0b100
if a > b then c ← 0b010
if a = b then c ← 0b001
$CR_{4\times \mathbf{cr}D+32:4\times \mathbf{cr}D+35}$ ← c || XER_{SO}

If **cmp** and L=0, the contents of **r**A[32–63] are compared with the contents of **r**B[32–63], treating the operands as signed integers.

If **cmpi** and L=0, the contents of **r**A[32–63] are compared with the sign-extended value of the SIMM field, treating the operands as signed integers.

The result of the comparison is placed into CR field **cr**D.

Other registers altered: CR field **cr**D

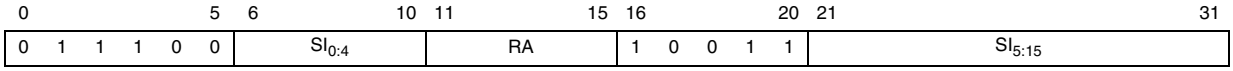
_cmp

VLE	User

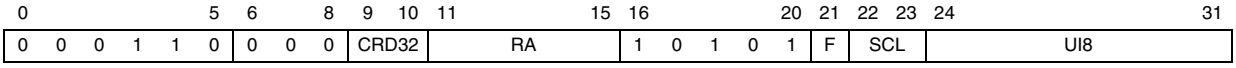
cmp

Compare [Immediate]

e_cmp16i **r**A,SI



e_cmpi **cr**D32,**r**A,SCI8



a ← GPR(RA)$_{32:63}$
if ‘e_cmpi’ then b ← SCI8(F,SCL,UI8)
if ‘e_cmp16i’ then b ← EXTS(SI$_{0:4}$ || SI$_{5:15}$)
if a < b then c ← 0b100
if a > b then c ← 0b010
if a = b then c ← 0b001
if ‘e_cmpi’ then CR$_{4\times CRD32+32:4\times CRD32+35}$ ← c || XER$_{SO}$ // only CR0-CR3
if ‘e_cmp16i’ then CR$_{32:35}$ ← c || XER$_{SO}$ // only CR0

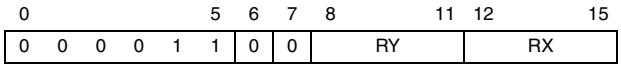
If **e_cmpi**, GPR(**r**A) contents are compared with the value of SCI8, treating operands as signed integers.

If **e_cmp16i**, GPR(**r**A) contents are compared with the sign-extended value of the SI field, treating operands as signed integers.

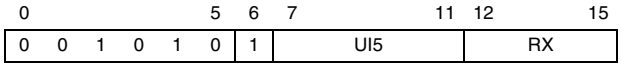
The result of the comparison is placed into CR field **cr**D (**cr**D32). For **e_cmpi**, only CR0–CR3 may be specified. For **e_cmp16i**, only CR0 may be specified.

Special Registers Altered: CR field **cr**D (**cr**D32) (CR0 for **e_cmp16i**)

se_cmp **r**X,**r**Y



se_cmpi **r**X,UI5



a ← GPR(RX)$_{32:63}$
if ‘se_cmpi’ then b ← ^{27}0 || UI5
if ‘se_cmp’ then b ← GPR(RY)$_{32:63}$
if a < b then c ← 0b100
if a > b then c ← 0b010
if a = b then c ← 0b001
CR$_{0:3}$ ← c || XER$_{SO}$

If **se_cmp**, the contents of GPR(**r**X) are compared with the contents of GPR(**r**Y), treating the operands as signed integers. The result of the comparison is placed into CR field 0.

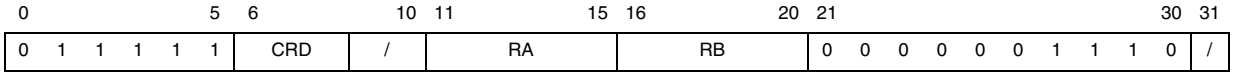
If **se_cmpi**, the contents of GPR(**r**X) are compared with the value of the zero-extended UI5 field, treating the operands as signed integers. The result of the comparison is placed into CR field 0.

Special Registers Altered: CR[0–3]

_cmph | VLE | User | **_cmph**

Compare Halfword [Immediate]

e_cmph **cr**D,**r**A,**r**B

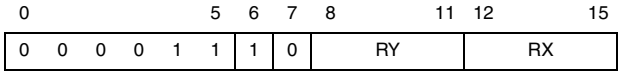


a ← EXTS(GPR(RA)$_{48:63}$)
b ← EXTS(GPR(RB)$_{48:63}$)
if a < b then c ← 0b100
if a > b then c ← 0b010
if a = b then c ← 0b001
$CR_{4\times CRD+32:4\times CRD+35}$ ← c || XER_{SO}

For **e_cmph**, the contents of the low-order 16 bits of GPR(**r**A) and GPR(**r**B) are compared, treating the operands as signed integers. The result of the comparison is placed into CR field CRD.

Special Registers Altered: CR field CRD

se_cmph **r**X,**r**Y

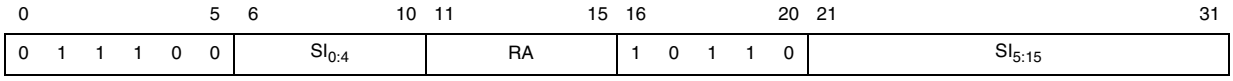


a ← EXTS(GPR(RX)$_{48:63}$)
b ← EXTS(GPR(RY)$_{48:63}$)
if a < b then c ← 0b100
if a > b then c ← 0b010
if a = b then c ← 0b001
$CR_{0:3}$ ← c || XER_{SO}

For **se_cmph**, the contents of the low-order 16 bits of GPR(**r**X) and GPR(**r**Y) are compared, treating the operands as signed integers. The result of the comparison is placed into CR field 0.

Special Registers Altered: CR[0–3]

e_cmph16i **r**A,SI



a ← EXTS(GPR(RA)$_{48:63}$)
b ← EXTS($SI_{0:4}$ || $SI_{5:15}$)
if a < b then c ← 0b100
if a > b then c ← 0b010
if a = b then c ← 0b001
$CR_{32:35}$ ← c || XER_{SO} // only CR0

The contents of the lower 16-bits of GPR(**r**A) are sign-extended and compared with the sign-extended value of the SI field, treating the operands as signed integers.

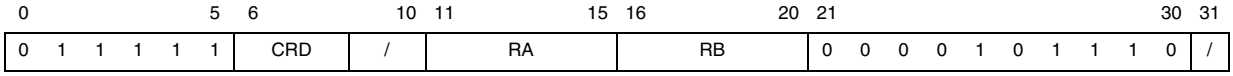
The result of the comparison is placed into CR0.

Special Registers Altered: CR0

_cmphl | VLE | User | **_cmphl**

Compare Halfword Logical [Immediate]

e_cmphl **cr**D,**r**A,**r**B

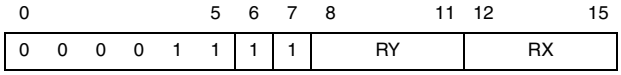


a ← EXTZ(GPR(RA)$_{48:63}$)
b ← EXTZ(GPR(RB)$_{48:63}$)
if a < b then c ← 0b100
if a > b then c ← 0b010
if a = b then c ← 0b001
$CR_{4\times CRD+32:4\times CRD+35}$ ← c || XER_{SO}

For **e_cmphl**, the contents of the low-order 16 bits of GPR(**r**A) and GPR(**r**B) are compared, treating the operands as unsigned integers. The result of the comparison is placed into CR field CRD.

Special Registers Altered: CR field CRD

se_cmphl **r**X,**r**Y

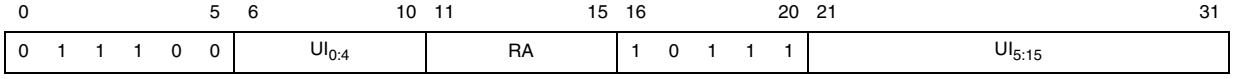


a ← GPR(RX)$_{48:63}$
b ← GPR(RY)$_{48:63}$
if a < b then c ← 0b100
if a > b then c ← 0b010
if a = b then c ← 0b001
$CR_{0:3}$ ← c || XER_{SO}

For **se_cmphl**, the contents of the low-order 16 bits of GPR(**r**X) and GPR(**r**Y) are compared, treating the operands as unsigned integers. The result of the comparison is placed into CR field 0.

Special Registers Altered: CR[0–3]

e_cmphl16i **r**A,UI



a ← $^{16}0$ || GPR(RA)$_{48:63)}$
b ← $^{16}0$ || $UI_{0:4}$ || $UI_{5:15}$
if a < b then c ← 0b100
if a > b then c ← 0b010
if a = b then c ← 0b001
$CR_{32:35}$ ← c || XER_{SO} // only CR0

The contents of the lower 16-bits of GPR(**r**A) are zero-extended and compared with the zero-extended value of the UI field, treating the operands as unsigned integers.

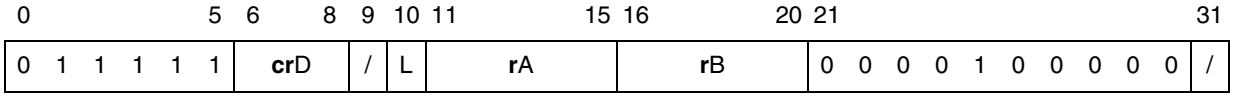
The result of the comparison is placed into CR0.

Special Registers Altered: CR0

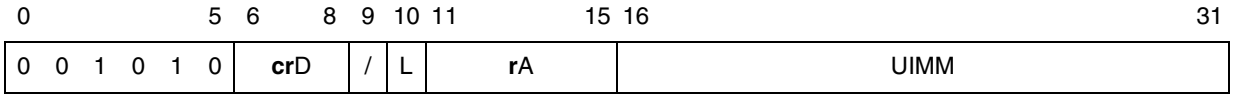
cmpl | Book E | User | **cmpl**

Compare logical [immediate]

cmpl **cr**D,L,**r**A,**r**B



cmpli **cr**D,L,**r**A,UIMM



if L=0 then a ← $^{32}0$ || $rA_{32:63}$
else a ← rA
if 'cmpli' then b ← $^{48}0$ || UIMM
if 'cmpl' & L=0 then b ← $^{32}0$ || $rB_{32:63}$
if 'cmpl' & L=1 then b ← rB
if a $<_u$ b then c ← 0b100
if a $>_u$ b then c ← 0b010
if a = b then c ← 0b001
$CR_{4\times crD+32:4\times crD+35}$ ← c || XER_{SO}

If **cmpl** and L=0, the contents of **r**A[32–63] are compared with the contents of **r**B[32–63], treating the operands as unsigned integers.

If **cmpli** and L=0, the contents of **r**A[32–63] are compared with the zero-extended value of the UIMM field, treating the operands as unsigned integers.

The result of the comparison is placed into CR field **cr**D.

Other registers altered: CR field **cr**D

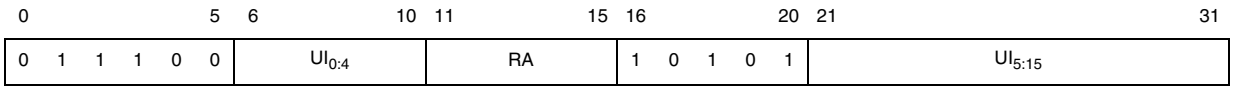
_cmpl

VLE	User

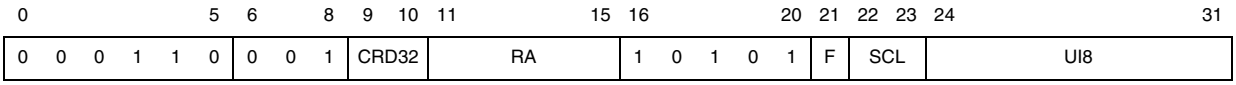
_cmpl

Compare Logical [Immediate]

e_cmpl16i **r**A,UI



e_cmpli **cr**D32,**r**A,SCI8



a ← GPR(RA)$_{32:63}$
if ‘e_cmpli’ then b ← SCI8(F,SCL,UI8)
if ‘e_cmpl16i’ then b ← ^{16}0 || UI$_{0:4}$ || UI$_{5:15}$
if a <$_u$ b then c ← 0b100
if a >$_u$ b then c ← 0b010
if a = b then c ← 0b001
if ‘e_cmpli’ then CR$_{4\times CRD32+32:4\times CRD32+35}$ ← c || XER$_{SO}$ // only CR0-CR3
if ‘e_cmp16i’ then CR$_{32:35}$ ← c || XER$_{SO}$ // only CR0

If **e_cmpi**, the contents of bits 32–63 of GPR(**r**A) are compared with the value of SCI8, treating the operands as unsigned integers.

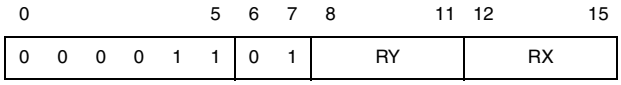
L must be 0 for 32-bit implementations

If **e_cmpl16i**, the contents of GPR(**r**A) are compared with the zero-extended value of the UI field, treating the operands as unsigned integers.

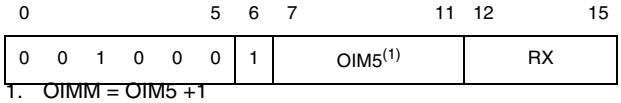
The result of the comparison is placed into CR field CRD (CRD32). For **e_cmpli**, only CR0–CR3 may be specified. For **e_cmpl16i**, only CR0 may be specified.

Special Registers Altered: CR field CRD (CRD32) (CR0 for **e_cmpl16i**)

se_cmpl **r**X,**r**Y



se_cmpli **r**X,OIMM



1. OIMM = OIM5 +1

a ← GPR(RX)$_{32:63}$
if ‘se_cmpli’ then b ← ^{27}0 || OFFSET(OIM5)
if ‘se_cmpl’ then b ← GPR(RY)$_{32:63}$
if a <$_u$ b then c ← 0b100
if a >$_u$ b then c ← 0b010
if a = b then c ← 0b001
CR$_{0:3}$ ← c || XER$_{SO}$

If **se_cmpl**, the contents of GPR(**r**X) are compared with the contents of GPR(**r**Y), treating the operands as unsigned integers. The result of the comparison is placed into CR field 0.

If **se_cmpli**, the contents of GPR(**r**X) are compared with the value of the zero-extended offset value of the OIM5 field (a final value in the range 1–32), treating the operands as unsigned integers. The result of the comparison is placed into CR field 0.

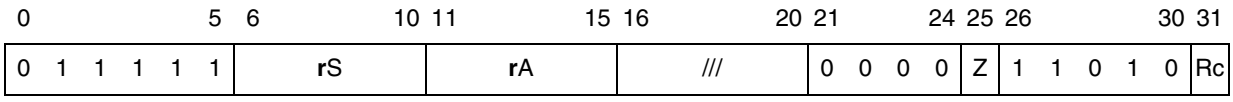
Special Registers Altered: CR[0–3]

cntlzw | Book E | User | **cntlzw**

Count leading zeros (word)

cntlzw **r**A,**r**S (Z=0, Rc=0)

cntlzw. **r**S (Z=0, Rc=1)




```
if ‘cntlzd’ then n ← 0 else n ← 32
i ← 0
do while n < 64
  if rS_n = 1 then leave
  n ← n + 1
  i ← i + 1
rA ← i
if Rc=1 then do
  GT ← i > 0
  EQ ← i = 0
  CR0 ← 0b0 || GT || EQ || SO
```

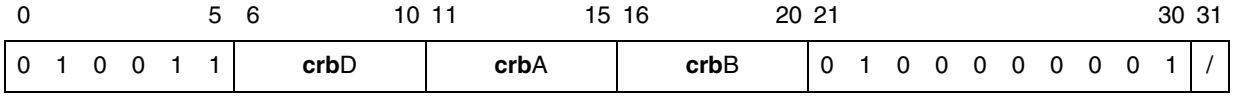
For **cntlzw[.]**, a count of the number of consecutive zero bits starting at **r**S[32] is placed into **r**A. This number ranges from 0 to 32, inclusive. If Rc=1, CR field 0 is set to reflect the result.

Other registers altered: CR0 (if Rc=1)

crand | Book E | User | **crand**

Condition register AND

crand **crb**D,**crb**A,**crb**B



$CR_{crbD+32} \leftarrow CR_{crbA+32}$ & $CR_{crbB+32}$

The content of bit **crb**A+32 of CR is ANDed with the content of bit **crb**B+32 of CR, and the result is placed into bit **crb**D+32 of CR.

Other registers altered: CR

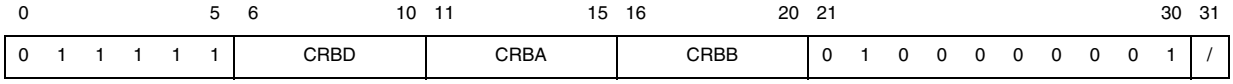
_crand

VLE	User

_crand

Condition Register AND

e_crand **crb**D,**crb**A,**crb**B



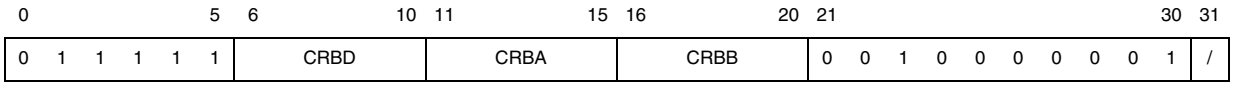
$CR_{BT+32} \leftarrow CR_{BA+32}$ & CR_{BB+32}

The content of bit CRBA+32 of the CR is ANDed with the content of bit CRBB+32 of the CR, and the result is placed into bit CRBD+32 of the CR.

Special Registers Altered: CR

Condition Register AND with Complement

e_crandc **crb**D,**crb**A,**crb**B



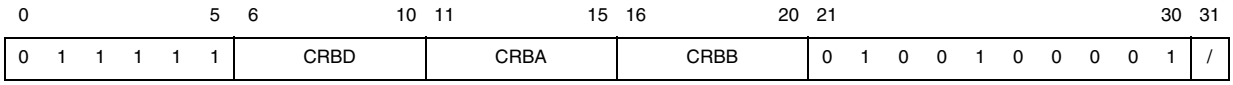
$CR_{BT+32} \leftarrow CR_{BA+32}$ & $\neg CR_{BB+32}$

The content of bit CRBA+32 of the CR is ANDed with the one's complement of the content of bit CRBB+32 of the CR, and the result is placed into bit CRBD+32 of the CR.

Special Registers Altered: CR

CR Equivalent

e_creqv **crb**D,**crb**A,**crb**B



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

The content of bit CRBA+32 of the CR is XORed with the content of bit CRBB+32 of the CR, and the one's complement of result is placed into bit CRBD+32 of the CR.

Special Registers Altered: CR

crandc | Book E | User | **crandc**

Condition register AND with complement

crandc **crb**D,**crb**A,**crb**B

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	1	0	0	1	1	**crb**D					**crb**A					**crb**B					0	0	1	0	0	0	0	0	0	1	/

$CR_{\mathbf{crb}D+32} \leftarrow CR_{\mathbf{crb}A+32} \,\&\, \neg CR_{\mathbf{crb}B+32}$

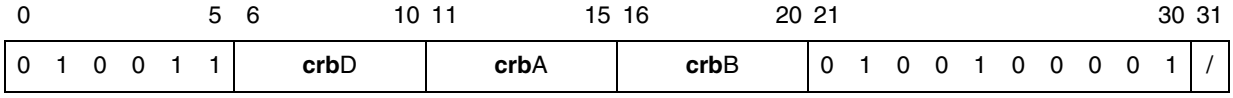
The content of bit **crb**A+32 of CR is ANDed with the one's complement of the content of bit **crb**B+32 of CR, and the result is placed into bit **crb**D+32 of CR.

Other registers altered: CR

creqv | Book E | User | **creqv**

Condition register equivalent

creqv **crb**D,**crb**A,**crb**B



$CR_{crbD+32} \leftarrow CR_{crbA+32} \equiv CR_{crbB+32}$

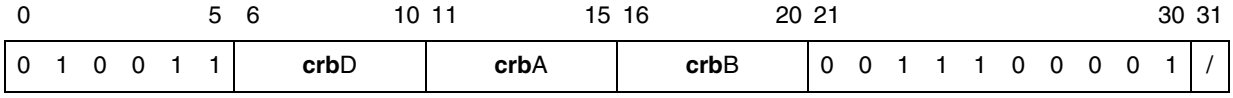
The content of bit **crb**A + 32 of CR is XORed with the content of bit **crb**B + 32 of CR, and the one's complement of result is placed into bit **crb**D+32 of CR.

Other registers altered: CR

crnand | Book E | User | **crnand**

Condition register NAND

crnand **crb**D,**crb**A,**crb**B



$CR_{crbD+32} \leftarrow \neg(CR_{crbA+32} \& CR_{crbB+32})$

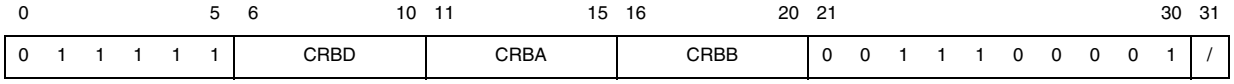
The content of bit **crb**A+32 of CR is ANDed with the content of bit **crb**B+32 of CR, and the one's complement of the result is placed into bit **crb**D+32 of CR.

Other registers altered: CR

_crnand | VLE | User | **_crnand**

Condition Register NAND

e_crnand **crb**D,**crb**A,**crb**B



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

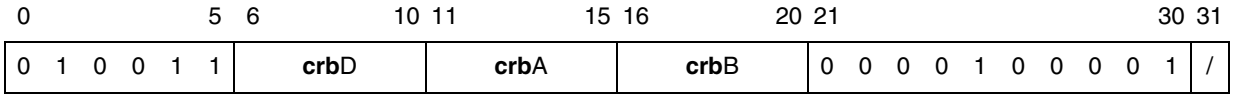
The content of bit CRBA+32 of the CR is ANDed with the content of bit CRBB+32 of the CR, and the one's complement of the result is placed into bit CRBD+32 of the CR.

Special Registers Altered: CR

crnor | Book E | User | **crnor**

Condition register NOR

crnor **crb**D,**crb**A,**crb**B



$CR_{crbD+32} \leftarrow \neg(CR_{crbA+32} \mid CR_{crbB+32})$

The content of bit **crb**A+32 of CR is ORed with the content of bit **crb**B+32 of CR, and the one's complement of the result is placed into bit **crb**D+32 of CR.

Other registers altered: CR

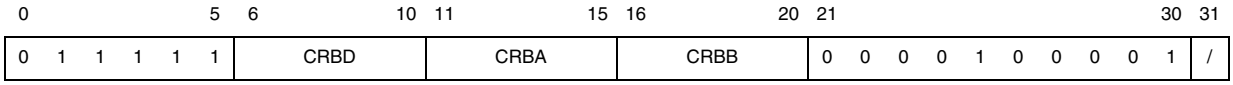
_crnor

VLE	User

_crnor

Condition Register NOR

e_crnor crbD,crbA,crbB



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

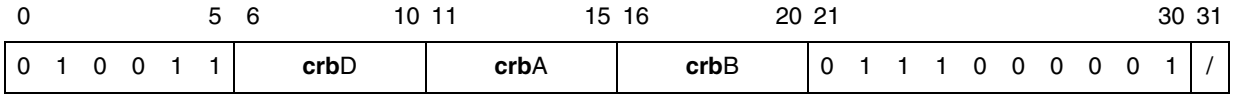
The content of bit CRBA+32 of the CR is ORed with the content of bit CRBB+32 of the CR, and the one's complement of the result is placed into bit CRBD+32 of the CR.

Special Registers Altered: CR

cror | Book E | User | **cror**

Condition register OR

cror **crb**D,**crb**A,**crb**B



$CR_{crbD+32} \leftarrow CR_{crbA+32} \mid CR_{crbB+32}$

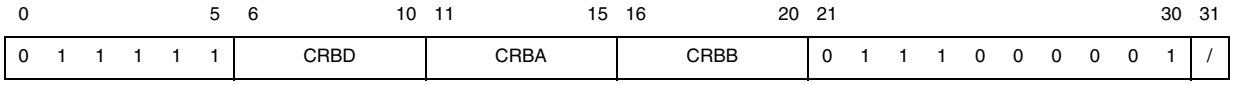
The content of bit **crb**A+32 of CR is ORed with the content of bit **crb**B+32 of CR, and the result is placed into bit **crb**D+32 of CR.

Other registers altered: CR

_cror | VLE | User | **_cror**

Condition Register OR

e_cror **crb**D,**crb**A,**crb**B



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

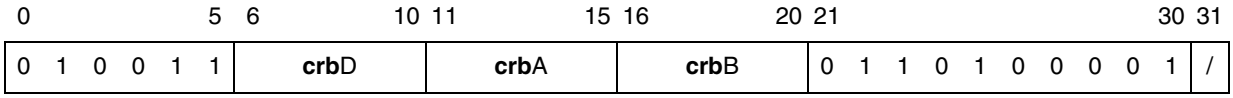
The content of bit CRBA+32 of the CR is ORed with the content of bit CRBB+32 of the CR, and the result is placed into bit CRBD+32 of the CR.

Special Registers Altered: CR

crorc | Book E | User | **crorc**

Condition register OR with complement

crorc **crb**D,**crb**A,**crb**B



$CR_{\mathbf{crb}D+32} \leftarrow CR_{\mathbf{crb}A+32} \mid \neg CR_{\mathbf{crb}B+32}$

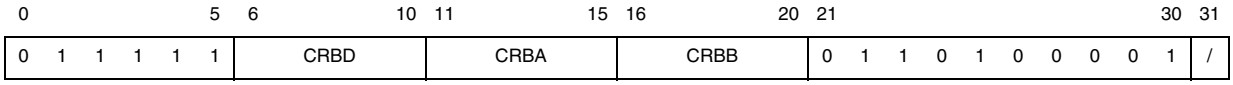
The content of bit **crb**A+32 of CR is ORed with the one's complement of the content of bit **crb**B+32 of CR, and the result is placed into bit **crb**D+32 of CR.

Other registers altered: CR

_crorc | VLE | User | **_crorc**

Condition Register OR with Complement

e_crorc **crb**D,**crb**A,**crb**B



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

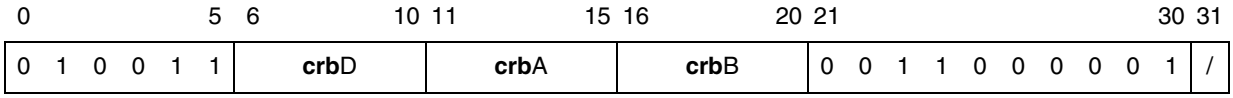
The content of bit CRBA+32 of the CR is ORed with the one's complement of the content of bit CRBB+32 of the CR, and the result is placed into bit CRBD+32 of the CR.

Special Registers Altered: CR

crxor | Book E | User | **crxor**

Condition register XOR

crxor **crb**D,**crb**A,**crb**B



$CR_{crbD+32} \leftarrow CR_{crbA+32} \oplus CR_{crbB+32}$

The content of bit **crb**A+32 of CR is XORed with the content of bit **crb**B+32 of CR, and the result is placed into bit **crb**D+32 of CR.

Other registers altered: CR

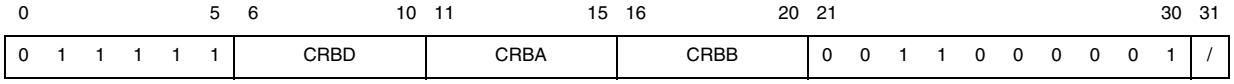
_crxor

VLE	User

_crxor

Condition Register XOR

e_crxor **crb**D,**crb**A,**crb**B



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

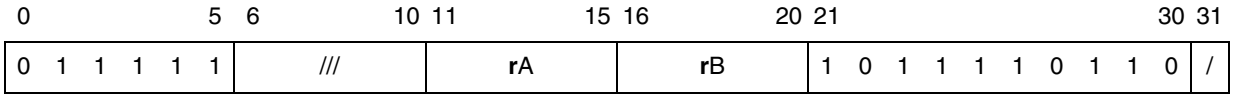
The content of bit CRBA+32 of the CR is XORed with the content of bit CRBB+32 of the CR, and the result is placed into bit CRBD+32 of the CR.

Special Registers Altered: CR

dcba | Book E | User | **dcba**

Data cache block allocate

dcba **r**A,**r**B



if rA=0 then a ← $^{64}0$ else a ← rA
EA ← $^{32}0$ || $(a + rB)_{32:63}$
AllocateDataCacheBlock(EA)

EA calculation:

Addressing Mode	EA for **r**A=0	EA for **r**A≠0
	$^{32}0$ || $\mathbf{r}B_{32:63}$	$^{32}0$ || $(\mathbf{r}A+\mathbf{r}B)_{32:63}$

dcba is a hint that performance would likely improve if the block containing the byte addressed by EA is established in the data cache without fetching the block from main memory, because the program is likely to soon store into a portion of the block and the contents of the rest of the block are not meaningful to the program. If the hint is honored, the contents of the block are undefined when the instruction completes. The hint is ignored if the block is caching-inhibited.

If the block containing the byte addressed by EA is in memory that is memory-coherence required and the block exists in a data cache of any other processors, it is kept coherent in those caches.

This instruction is treated as a storeexcept that an interrupt is not taken for a translation or protection violation.

This instruction may establish a block in the data cache without verifying that the associated real address is valid. This can cause a delayed machine check interrupt.

Other registers altered: None

dcbf | Book E | User | **dcbf**

Data cache block flush

dcbf **r**A,**r**B

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	1	1	1	1	1	///					**r**A					**r**B					0	0	0	1	0	1	0	1	1	0	/

if rA=0 then a ← $^{64}0$ else a ← rA
EA ← $^{32}0$ || $(a + rB)_{32:63}$
FlushDataCacheBlock(EA)

EA calculation:

Addressing Mode	EA for **r**A=0	EA for **r**A≠0
	$^{32}0$ || $\mathbf{r}B_{32:63}$	$^{32}0$ || $(\mathbf{r}A+\mathbf{r}B)_{32:63}$

If the block containing the byte addressed by EA is in memory 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 memory. Additional locations in the block may also be written to main memory. The block is invalidated in the data caches of all processors.

If the block containing the byte addressed by EA is in memory 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 memory. Additional locations in the block may also be written to main memory. The block is invalidated in the data cache of this processor.

On some implementations, HID1[ABE] must be set to allow management of external L2 caches (for implementations with L2 caches) as well as other L1 caches in the system.

The function of this instruction is independent of whether the block containing the byte addressed by EA is in memory that is write-through required or caching-inhibited.

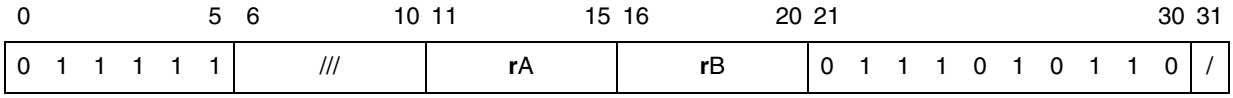
This instruction is treated as a load. See *Cache management instructions on page 216*.”

Other registers altered: None

dcbi | Book E | User | **dcbi**

Data cache block invalidate

dcbi **r**A,**r**B



if rA=0 then a ← $^{64}0$ else a ← rA
EA ← $^{32}0$ || $(a + rB)_{32:63}$
InvalidateDataCacheBlock(EA)
EA calculation: Addressing Mode EA for **r**A=0 EA for **r**A≠0
$^{32}0$ || $\mathbf{r}B_{32:63}$ $^{32}0$ || $(\mathbf{r}A+\mathbf{r}B)_{32:63}$

If the block containing the byte addressed by EA is in is coherence-required memory and any block containing the addressed byte is any processors' data cache is invalidated in those 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 memory and additional locations in the block may be written to main memory.

If the block containing the byte addressed by EA is not coherence-required memory 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, any locations in the block considered modified in that data cache are written to main memory; additional locations in the block may be written to main memory.

dcbi is treated as a store on implementations that invalidate a block without first writing to main memory 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.

Additional information about this instruction is as follows.

- The data cache block size for **dcbi** is the same as for **dcbf**.
- 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.

Other registers altered: None

dcblc | Cache locking APU | User | **dcblc**

Data cache block lock clear

dcblc CT,**r**A,**r**B Form: X

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	1	1	1	1	1	CT					**r**A					**r**B					0	1	1	0	0	0	0	1	1	0	/

if **r**A = 0 then a ← $^{64}0$ else a ← GPR(**r**A)
if Mode32 then EA ← $^{32}0$ || (a + GPR(**r**B))$_{32:63}$
if Mode64 then EA ← a + GPR(**r**B)
DataCacheBlockClearLock(CT, EA)
EA calculation: EA for **r**A=0EA for **r**A≠0
$^{32}0$ || GPR(rB)$_{32:63}$ $^{32}0$ || (GPR(rA)+GPR(rB))$_{32:63}$

The data cache specified by CT has the cache line corresponding to EA unlocked allowing the line to participate in the normal replacement policy.

Cache lock clear instructions remove locks previously set by cache lock set instructions.

User-level cache instructions on page 180," lists supported CT values. An implementation may use other CT values to enable software to target specific, implementation-dependent portions of its cache hierarchy or structure.

The instruction is treated as a load with respect to translation and memory protection and can cause DSI and DTLB error interrupts accordingly.

An unable-to-unlock condition is said to occur any of the following conditions exist:

- The target address is marked cache-inhibited, or the storage attributes of the address uses 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.
- The target address is not in the cache or is present in the cache but is not locked.

If an unable-to-unlock condition occurs, no cache operation is performed.

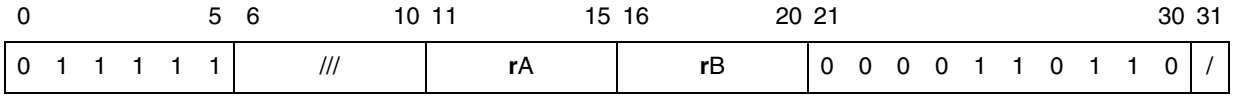
EIS Specifics

Clearing and then setting L1CSR0[CLFR] allows system software to clear all L1 data cache locking bits without knowing the addresses of the lines locked.

dcbst | Book E | User | **dcbst**

Data Cache Block Store

dcbst **r**A,**r**B



if rA=0 then a ← $^{64}0$ else a ← rA
EA ← $^{32}0$ || $(a + rB)_{32:63}$
StoreDataCacheBlock(EA)
EA calculation: Addressing ModeEA for **r**A=0EA for **r**A≠0
$^{32}0$ || $\mathbf{r}B_{32:63}$ $^{32}0$ || $(\mathbf{r}A+\mathbf{r}B)_{32:63}$

If the block containing the byte addressed by EA is in memory 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 memory. Additional locations in the block may be written to main memory. The block ceases to be considered to be modified in that data cache.

If the block containing the byte addressed by EA is in memory that is not memory-coherence required and 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, those locations are written to main memory. Additional locations in the block may be written to main memory. The block ceases to be considered to be modified in that cache.

The function of this instruction is independent of whether the block containing the byte addressed by EA is in memory that is write-through required or caching-inhibited.

This instruction is treated as a load.

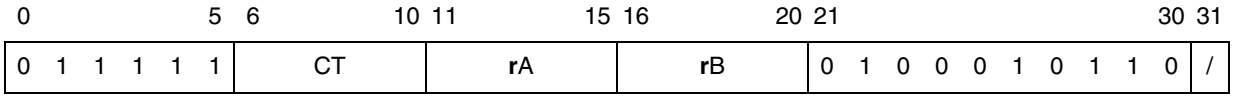
On some implementations, HID1[ABE] must be set to allow management of external L2 caches (for implementations with L2 caches) as well as other L1 caches in the system.

Other registers altered: None

dcbt | Book E | User | **dcbt**

Data cache block touch

dcbt CT,**r**A,**r**B



if rA=0 then a ← $^{64}0$ else a ← rA
EA ← $^{32}0$ || $(a + rB)_{32:63}$
PrefetchDataCacheBlock(CT, EA)
EA calculation: Addressing ModeEA for **r**A=0EA for **r**A≠0
$^{32}0$ || $\mathbf{r}B_{32:63}$ $^{32}0$ || $(\mathbf{r}A+\mathbf{r}B)_{32:63}$

User-level cache instructions on page 180," lists supported CT values. An implementation may use other CT values to enable software to target specific, implementation-dependent portions of its cache hierarchy or structure.

Implementations should perform no operation when CT specifies a value not supported by the implementation.

The hint is ignored if the block is caching-inhibited.

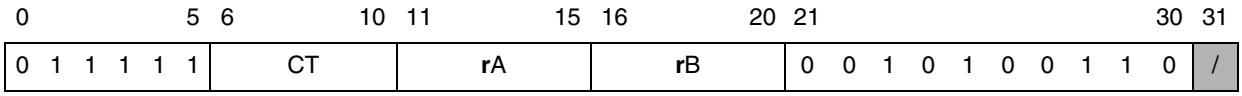
This instruction is treated as a load except that an interrupt is not taken for a translation or protection violation.

Other registers altered: None

dcbtls | Cache locking APU | User | **dcbtls**

Data cache block touch and lock set

dcbtls CT,**r**A,**r**B Form: X



if **r**A = 0 then a ← $^{64}0$ else a ← GPR(**r**A)
if Mode32 then EA ← $^{32}0$ || (a + GPR(**r**B))$_{32:63}$
if Mode64 then EA ← a + GPR(**r**B)
PrefetchDataCacheBlockLockSet(CT, EA)

EA calculation: EA for **r**A=0 EA for **r**A≠0
$^{32}0$ || GPR(rB)$_{32:63}$ $^{32}0$ || (GPR(rA)+GPR(rB))$_{32:63}$

The data cache specified by CT has the cache line corresponding to EA loaded and locked into the cache. If the line already exists in the cache, it is locked without being refetched.

Cache touch and lock set instructions let software lock cache lines into the cache to provide lower latency for critical cache accesses and more deterministic behavior. Locked lines do not participate in the normal replacement policy when a line must be victimized for replacement.

User-level cache instructions on page 180," lists supported CT values. An implementation may use other CT values to enable software to target specific, implementation-dependent portions of its cache hierarchy or structure.

The instruction is treated as a load with respect to translation and memory protection and can cause DSI and DTLB error interrupts accordingly.

An unable to lock condition is said to occur any of the following conditions exist:

- The target address is marked cache-inhibited, or the storage attributes of the address uses 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.

If an unable to lock condition occurs, no cache operation is performed and LICSR0[DCUL] is set appropriately.

Overlocking is said to exist is all available ways for a given cache index are already locked. If overlocking occurs for **dcbtls** and if the lock was targeted for the primary cache (CT = 0), the requested line is not locked into the cache. When overlock occurs, L1CSR1[DCLO] is set. If L1CSR1[DCLOA] is set, the requested line is locked into the cache and implementation dependent line currently locked in the cache is evicted.

The results of overlocking and unable to lock conditions for caches other than the primary cache and secondary cache are defined as part of the architecture for the specific cache hierarchy designated by CT.

Other registers altered:

- L1CSR0[DCUL] if unable to lock occurs
- L1CSR0[DCLO] (L2CSR[L2CLO]) if lock overflow occurs

dcbtst | Book E | User | **dcbtst**

Data cache block touch for store

dcbtst CT,**r**A,**r**B

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	1	1	1	1	1	CT					**r**A					**r**B					0	0	1	1	1	1	0	1	1	0	/

if rA=0 then a ← $^{64}0$ else a ← rA
EA ← $^{32}0$ || $(a + rB)_{32:63}$
PrefetchForstoreDataCacheBlock(CT, EA)
EA calculation: Addressing ModeEA for **r**A=0EA for **r**A≠0
$^{32}0$ || $\mathbf{r}B_{32:63}$ $^{32}0$ || $(\mathbf{r}A+\mathbf{r}B)_{32:63}$

If CT=0, this instruction is a hint that performance would likely be improved if the block containing the byte addressed by EA is fetched into the data cache, because the program will probably soon store into the addressed byte.

User-level cache instructions on page 180," lists supported CT values. An implementation may use other CT values to enable software to target specific, implementation-dependent portions of its cache hierarchy or structure.

Implementations should perform no operation when CT specifies a value not supported by the implementation.

The hint is ignored if the block is caching-inhibited.

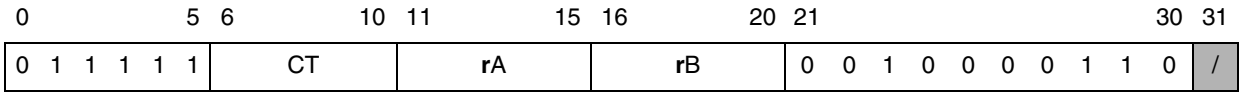
This instruction is treated as a load , except that an interrupt is not taken for a translation or protection violation.

Other registers altered: None

dcbtstls | Cache locking APU | User | **dcbtstls**

Data cache block touch for store and lock set

dcbtstls CT,**r**A,**r**B Form: X



if **r**A = 0 then a ← $^{64}0$ else a ← GPR(**r**A)
if Mode32 then EA ← $^{32}0$ || (a + GPR(**r**B))$_{32:63}$
if Mode64 then EA ← a + GPR(**r**B)
PrefetchDataCacheBlockLockSet(CT, EA)
EA calculation: EA for **r**A=0 EA for **r**A≠0
$^{32}0$ || GPR(rB)$_{32:63}$ $^{32}0$ || (GPR(rA)+GPR(rB))$_{32:63}$

The data cache specified by CT has the cache line corresponding to EA loaded and locked into the cache. If the line already exists in the cache, it is locked without refetching from memory.

Cache touch and lock set instructions allow software to lock lines into the cache to shorten latency for critical cache accesses and more deterministic behavior. Lines locked in the cache do not participate in the normal replacement policy when a line must be victimized for replacement.

User-level cache instructions on page 180," lists supported CT values. An implementation may use other CT values to enable software to target specific, implementation-dependent portions of its cache hierarchy or structure.

Table 114 describes how this instruction is treated with respect to translation and memory protection.

For unable-to-lock conditions, described in *Unable-to-lock conditions on page 849*," no cache operation is performed and LICSR0[DCUL] is set.

Overlocking occurs when all available ways for a given cache index are already locked. If an overlocking condition occurs for a **dcbtstls** instruction and if the lock was targeted for the primary cache or secondary cache (CT = 0 or CT = 2), the requested line is not locked into the cache. When overlock occurs, L1CSR1[DCLO] (L2CSR[L2CLO] for CT = 2) is set. If L1CSR1[DCLOA] is set (or L2CSR[L2CLOA] for CT = 2), the requested line is locked into the cache and implementation dependent line currently locked in the cache is evicted. If system software wants to precisely determine if an overlock event has occurred in the L1 data cache, it must perform the following code sequence:

```
dcbtstls
msync
mfspr (L1CSR0)
(check L1CSR0[DCUL] bit for data cache index unable-to-lock condition)
(check L1CSR0[DCLO] bit for data cache index overlock condition)
```

Results of overlocking and unable-to-lock conditions for caches other than the primary and secondary cache are defined as part of the architecture for the cache hierarchy designated by CT.

Other registers altered:

- L1CSR0[DCUL] if unable to lock occurs
- L1CSR0[DCLO] (L2CSR[L2CLO]) if lock overflow occurs

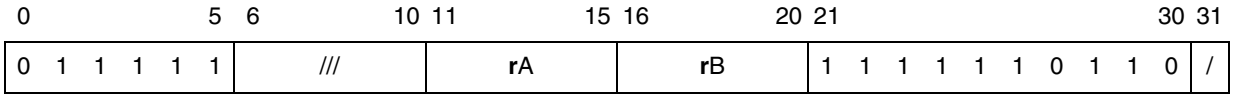
EIS specifics:

Clearing and then setting L1CSR0[CLFR] allows system software to clear all data cache locking bits without knowing the addresses of the lines locked.

dcbz | Book E | User | **dcbz**

Data cache block set to zero

dcbz **r**A,**r**B



if rA=0 then a ← $^{64}0$ else a ← rA
EA ← $^{32}0$ || $(a + rB)_{32:63}$
ZeroDataCacheBlock(EA)
EA calculation: Addressing ModeEA for **r**A=0EA for **r**A≠0
$^{32}0$ || $\mathbf{r}B_{32:63}$ $^{32}0$ || $(\mathbf{r}A+\mathbf{r}B)_{32:63}$

If the block containing the addressed byte is in the data cache, all bytes of the block are cleared.

If the block containing the byte addressed by EA is not in the data cache and is in memory that is not caching-inhibited, the block is established in the data cache without fetching the block from main memory, and all bytes of the block are cleared.

If the block containing the byte addressed by EA is not in the data cache and is in storage that is not caching inhibited and cannot be established in the cache, then one of the following occurs:

- All bytes of the area of main storage that corresponds to the addressed block are set to zero
- An alignment interrupt is taken

If the block containing the byte addressed by EA is in storage that is caching inhibited or write through required, one of the following occurs:

- All bytes of the area of main storage that corresponds to the addressed block are set to zero
- An alignment interrupt is taken.

If the block containing the byte addressed by EA is in memory-coherence required memory and the block exists in any other processors' data cache, it is kept coherent in those caches.

dcbz may establish a block in the data cache without verifying that the associated real address is valid. This can cause a delayed machine check interrupt.

dcbz is treated as a store.

- On some implementations, HID1[ABE] must be set to allow management of external L2 caches (for implementations with L2 caches) as well as other L1 caches in the system.
- **dcbz** may cause a cache-locking exception on some implementations. See the user documentation.

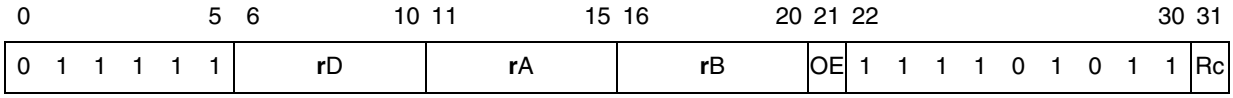
Other registers altered: None

Programming note: If the block containing the byte addressed by EA is in memory that is caching-inhibited or write-through required, the alignment interrupt handler should clear all bytes of the area of main memory that corresponds to the addressed block.

divw | Book E | User | **divw**

Divide word

divw	**r**D,**r**A,**r**B	(OE=0, Rc=0)
divw.	**r**D,**r**A,**r**B	(OE=0, Rc=1)
divwo	**r**D,**r**A,**r**B	(OE=1, Rc=0)
divwo.	**r**D,**r**A,**r**B	(OE=1, Rc=1)



$dividend_{0:31} \leftarrow rA_{32:63}$
$divisor_{0:31} \leftarrow rB_{32:63}$
$quotient_{0:31} \leftarrow dividend \div divisor$
if OE=1 then do
  $OV \leftarrow ((rA_{32:63}=-2^{31}) \& (rB_{32:63}=-1)) \mid (rB_{32:63}=0)$
  $SO \leftarrow SO \mid OV$
if Rc=1 then do
  $LT \leftarrow quotient < 0$
  $GT \leftarrow quotient > 0$
  $EQ \leftarrow quotient = 0$
  $CR0 \leftarrow LT \parallel GT \parallel EQ \parallel SO$
$rD_{32:63} \leftarrow quotient$
$rD_{0:31} \leftarrow undefined$

The 32-bit quotient of the contents of **r**A[32–63] divided by the contents of **r**B[32–63] is placed into **r**D[32–63]. **r**D[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 the following:

dividend = (quotient × divisor) + r

Here, $0 \le r < |divisor|$ if the dividend is nonnegative and $-|divisor| < r \le 0$ if it is negative.

If any of the following divisions is attempted, the contents of **r**D are undefined as are (if Rc=1) the contents of the CR0[LT,GT,EQ]. In these cases, if OE=1, OV is set.

0x8000_0000 ÷ –1
<anything> ÷ 0

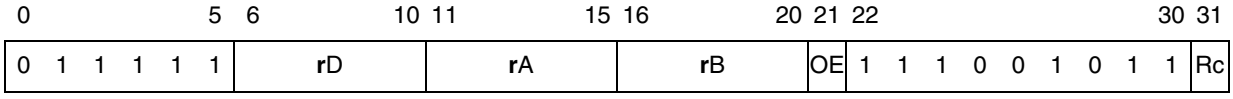
Other registers altered:

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

divwu | Book E | User | **divwu**

Divide word unsigned

divwu	**r**D,**r**A,**r**B	(OE=0, Rc=0)
divwu.	**r**D,**r**A,**r**B	(OE=0, Rc=1)
divwuo	**r**D,**r**A,**r**B	(OE=1, Rc=0)
divwuo.	**r**D,**r**A,**r**B	(OE=1, Rc=1)



$dividend_{0:31} \leftarrow rA_{32:63}$
$divisor_{0:31} \leftarrow rB_{32:63}$
$quotient_{0:31} \leftarrow dividend \div divisor$
if OE=1 then do
  $OV \leftarrow (rB_{32:63}=0)$
  $SO \leftarrow SO \mid OV$
if Rc=1 then do
  $LT \leftarrow quotient < 0$
  $GT \leftarrow quotient > 0$
  $EQ \leftarrow quotient = 0$
  $CR0 \leftarrow LT \mid\mid GT \mid\mid EQ \mid\mid SO$
$rD_{32:63} \leftarrow quotient$
$rD_{0:31} \leftarrow$ undefined

The 32-bit quotient of the contents of **r**A[32–63] divided by the contents of **r**B[32–63] is placed into **r**D[32–63]. **r**D[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 the following:

dividend = (quotient × divisor) + r

Here, $0 \leq r < divisor$.

If an attempt is made to perform the following division, the contents of **r**D are undefined as are (if Rc=1) the contents of the LT, GT, and EQ bits of CR0. In this case, if OE=1 OV is set.

<anything> ÷ 0

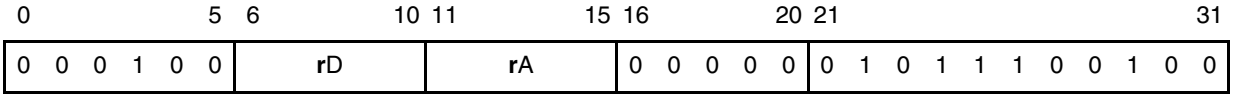
Other registers altered:

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

efdabs | Scalar DPFP APU | User | **efdabs**

Floating-point double-precision absolute value

efdabs **r**D,**r**A



$rD_{0:63} \leftarrow 0b0 \parallel rA_{1:63}$

The sign bit of **r**A is set to 0 and the result is placed into **r**D.

Exceptions:

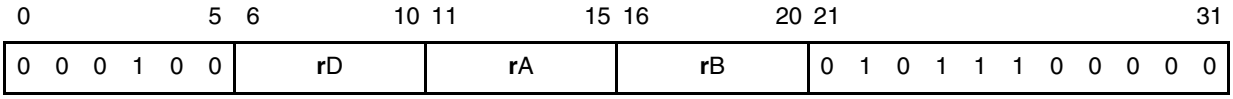
Exception detection for embedded floating-point absolute value operations is implementation dependent. An implementation may choose to not detect exceptions and carry out the sign bit operation. If the implementation does not detect exceptions, or if exception detection is disabled, the computation can be carried out in one of two ways, as a sign bit operation ignoring the rest of the contents of the source register, or by examining the input and appropriately saturating the input prior to performing the operation.

If an implementation chooses to handle exceptions, the exception is handled as follows: If **r**A is Infinity, Denorm, or NaN, SPEFSCR[FINV] is set, and FG and FX are cleared. If floating-point invalid input exceptions are enabled, an interrupt is taken and the destination register is not updated.

efdadd | Scalar DPFP APU | User | **efdadd**

Floating-point double-precision add

efdadd **r**D,**r**A,**rB**



$rD_{0:63} \leftarrow rA_{0:63} +_{dp} rB_{0:63}$

rA is added to **rB** and the result is stored in **r**D. If **r**A is NaN or infinity, the result is either *pmax* ($a_{sign}==0$), or *nmax* ($a_{sign}==1$). Otherwise, If **rB** is NaN or infinity, the result is either *pmax* ($b_{sign}==0$), or *nmax* ($b_{sign}==1$). Otherwise, if an overflow occurs, *pmax* or *nmax* (as appropriate) is stored in **r**D. If an underflow occurs, +0 (for rounding modes RN, RZ, RP) or -0 (for rounding mode RM) is stored in **r**D.

Exceptions:

If the contents of **r**A or **rB** are Infinity, Denorm, or NaN, SPEFSCR[FINV] is set. If SPEFSCR[FINVE] is set, an interrupt is taken, and the destination register is not updated. Otherwise, if an overflow occurs, SPEFSCR[FOVF] is set, or if an underflow occurs, SPEFSCR[FUNF] is set. If either underflow or overflow exceptions are enabled and the corresponding bit is set, an interrupt is taken. If any of these interrupts are taken, the destination register is not updated.

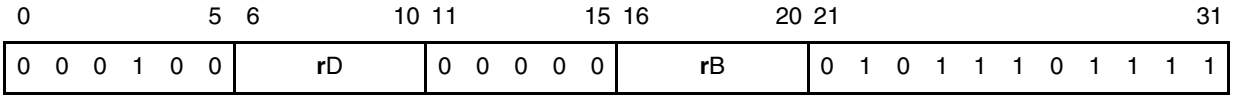
If the result of this instruction is inexact or if an overflow occurs but overflow exceptions are disabled, and no other interrupt is taken, SPEFSCR[FINXS] is set. If the floating-point inexact exception is enabled, an interrupt is taken using the floating-point round interrupt vector. In this case, the destination register is updated with the truncated result, the FG and FX bits are properly updated to allow rounding to be performed in the interrupt handler.

FG and FX are cleared if an overflow, underflow, or invalid operation/input error is signaled, regardless of enabled exceptions.

efdcfs | Scalar DPFP APU | User | **efdcfs**

Floating-point double-precision convert from single-precision

efdcfs **r**D,**rB**




```
FP32format f;
FP64format result;
f ← rB32:63
if (fexp = 0) & (ffrac = 0)) then
   result ← fsign || 63 0   // signed zero value
else if Isa32NaNorInfinity(f) | Isa32Denorm(f) then
   SPEFSCRFINV ← 1
   result ← fsign || 0b11111111110 || 52 1   // max value
else if Isa32Denorm(f) then
   SPEFSCRFINV ← 1
   result ← fsign || 63 0
else
   resultsign ← fsign
   resultexp ← fexp - 127 + 1023
   resultfrac ← ffrac || 29 0

rD0:63 = result
```

The single-precision floating-point value in the low element of **rB** is converted to a double-precision floating-point value and the result is placed into **r**D. The rounding mode is not used since this conversion is always exact.

Exceptions:

If the low element of **rB** is Infinity, Denorm, or NaN, SPEFSCR[FINV] is set. If SPEFSCR[FINVE] is set, an interrupt is taken, and the destination register is not updated.

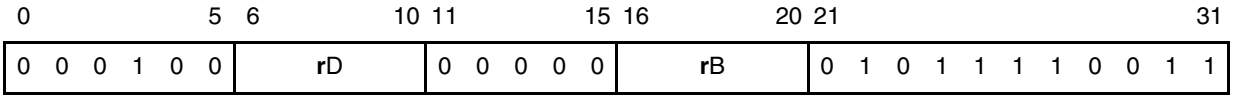
FG and FX are always cleared.

Note: Architecture Note: This instruction is optional if neither the embedded scalar single-precision floating-point APU or the embedded vector single-precision floating-point APU are implemented.

efdcfsf | Scalar DPFP APU | User | **efdcfsf**

Convert floating-point double-precision from signed fraction

efdcfsf **rD,rB**



$rD_{0:63} \leftarrow$ CnvtI32ToFP64($rB_{32:63}$, SIGN, F)

The signed fractional low element in **rB** is converted to a double-precision floating-point value using the current rounding mode and the result is placed into **r**D.

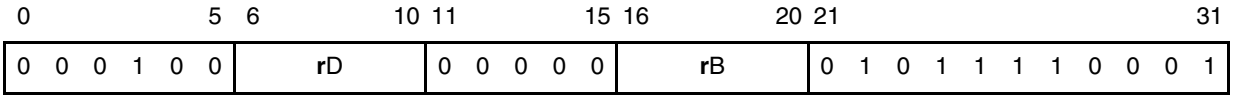
Exceptions:

None.

efdcfsi | Scalar DPFP APU | User | **efdcfsi**

Convert floating-point double-precision from signed integer

efdcfsi **rD,rB**



$rD_{0:63} \leftarrow CnvtSI32ToFP64(rB_{32:63}, SIGN, I)$

The signed integer low element in **rB** is converted to a double-precision floating-point value using the current rounding mode and the result is placed into **r**D.

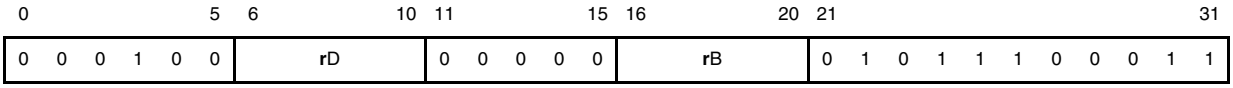
Exceptions:

None.

efdcfsid | Scalar DPFP APU | User | **efdcfsid**

Convert floating-point double-precision from signed integer doubleword

efdcfsid **rD,rB**



$rD_{0:63} \leftarrow$ CnvtI64ToFP64($rB_{0:63}$, SIGN)

The signed integer doubleword in **rB** is converted to a double-precision floating-point value using the current rounding mode and the result is placed into **r**D.

Exceptions:

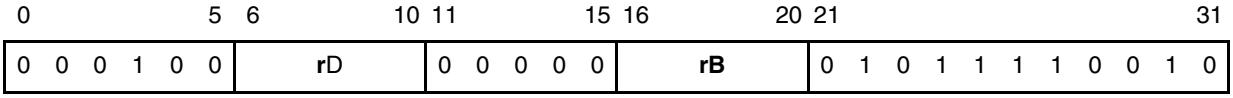
This instruction can signal an inexact status and set SPEFSCR[FINXS] if the conversion is not exact. If the floating-point inexact exception is enabled, an interrupt is taken using the floating-point round interrupt vector. In this case, the destination register is updated with the truncated result, the FG and FX bits are properly updated to allow rounding to be performed in the interrupt handler.

This instruction may only be implemented for 64-bit implementations.

efdcfuf | Scalar DPFP APU | User | **efdcfuf**

Convert floating-point double-precision from unsigned fraction

efdcfuf **rD,rB**



$rD_{0:63} \leftarrow$ CnvtI32ToFP64($rB_{32:63}$, UNSIGN, F)

The unsigned fractional low element in **rB** is converted to a double-precision floating-point value using the current rounding mode and the result is placed into **r**D.

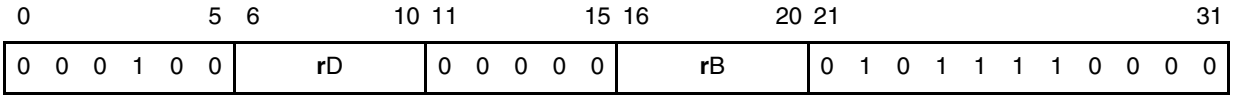
Exceptions:

None.

efdcfui | Scalar DPFP APU | User | **efdcfui**

Convert floating-point double-precision from unsigned integer

efdcfui **rD,rB**



$rD_{0:63} \leftarrow$ CnvtSI32ToFP64($rB_{32:63}$, UNSIGN, I)

The unsigned integer low element in **rB** is converted to a double-precision floating-point value using the current rounding mode and the result is placed into **r**D.

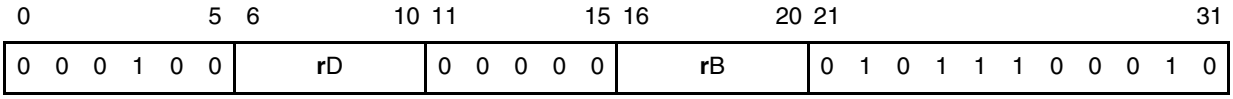
Exceptions:

None.

efdcfuid | Scalar DPFP APU | User | **efdcfuid**

Convert floating-point double-precision from unsigned integer doubleword

efdcfuid **rD,rB**



$rD_{0:63} \leftarrow$ CnvtI64ToFP64($rB_{0:63}$, UNSIGN)

The unsigned integer doubleword in **rB** is converted to a double-precision floating-point value using the current rounding mode and the result is placed into **r**D.

Exceptions:

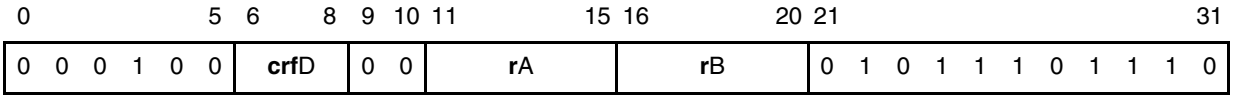
This instruction can signal an inexact status and set SPEFSCR[FINXS] if the conversion is not exact. If the floating-point inexact exception is enabled, an interrupt is taken using the floating-point round interrupt vector. In this case, the destination register is updated with the truncated result, the FG and FX bits are properly updated to allow rounding to be performed in the interrupt handler.

This instruction may only be implemented for 64-bit implementations.

efdcmpeq | Scalar DPFP APU | User | **efdcmpeq**

Floating-point double-precision compare equal

efdcmpeq **crf**D,**rA**,**rB**



al ← $rA_{0:63}$
bl ← $rB_{0:63}$
if (al = bl) then cl ← 1
else cl ← 0
$CR_{4*crD:4*crD+3}$ ← undefined || cl || undefined || undefined

rA is compared against **rB**. If **r**A is equal to **rB**, the bit in the **crf**D is set, otherwise it is cleared. Comparison ignores the sign of 0 (+0 = -0).

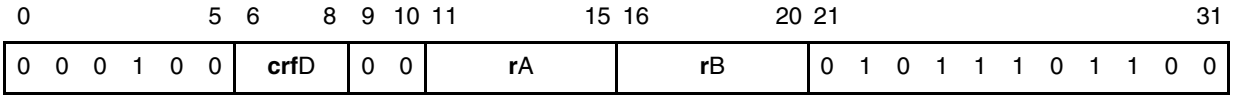
Exceptions:

If the contents of **r**A or **rB** are Infinity, Denorm, or NaN, SPEFSCR[FINV] is set, and the FGH FXH, FG and FX bits are cleared. If floating-point invalid input exceptions are enabled, an interrupt is taken and the condition register is not updated. Otherwise, the comparison proceeds after treating NaNs, Infinities, and Denorms as normalized numbers, using their values of '*e*' and '*f*' directly.

efdcmpgt | Scalar DPFP APU | User | **efdcmpgt**

Floating-point double-precision compare greater than

efdcmpgt **crf**D,**r**A,**rB**



al ← $rA_{0:63}$
bl ← $rB_{0:63}$
if (al > bl) then cl ← 1
else cl ← 0
$CR_{4*crD:4*crD+3}$ ← undefined || cl || undefined || undefined

rA is compared against **rB**. If **r**A is greater than **rB**, the bit in the **crf**D is set, otherwise it is cleared. Comparison ignores the sign of 0 (+0 = -0).

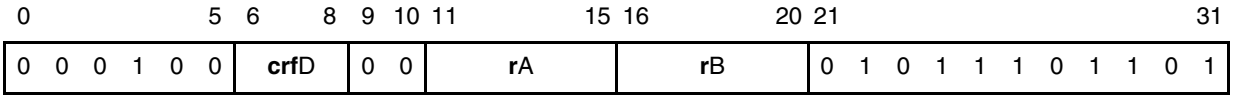
Exceptions:

If the contents of **r**A or **rB** are Infinity, Denorm, or NaN, SPEFSCR[FINV] is set, and the FGH FXH, FG and FX bits are cleared. If floating-point invalid input exceptions are enabled, an interrupt is taken and the condition register is not updated. Otherwise, the comparison proceeds after treating NaNs, Infinities, and Denorms as normalized numbers, using their values of '*e*' and '*f*' directly.

efdcmplt efdcmplt

Floating-point double-precision compare less than

efdcmplt **crf**D,**r**A,**rB**



$al \leftarrow rA_{0:63}$
$bl \leftarrow rB_{0:63}$
if (al < bl) then cl ← 1
else cl ← 0
$CR_{4*crD:4*crD+3} \leftarrow$ undefined || cl || undefined || undefined

rA is compared against **rB**. If **r**A is less than **rB**, the bit in the **crf**D is set, otherwise it is cleared. Comparison ignores the sign of 0 (+0 = -0).

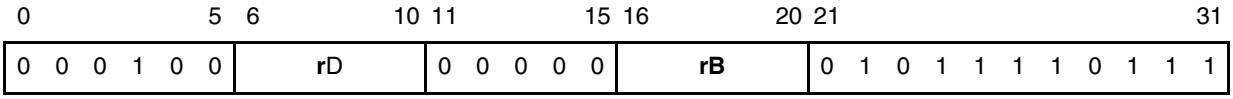
Exceptions:

If the contents of **r**A or **rB** are Infinity, Denorm, or NaN, SPEFSCR[FINV] is set, and the FGH FXH, FG and FX bits are cleared. If floating-point invalid input exceptions are enabled, an interrupt is taken and the condition register is not updated. Otherwise, the comparison proceeds after treating NaNs, Infinities, and Denorms as normalized numbers, using their values of '*e*' and '*f*' directly.

efdctsf | Scalar DPFP APU | User | **efdctsf**

Convert floating-point double-precision to signed fraction

efdctsf **rD,rB**



$rD_{32:63} \leftarrow$ CnvtFP64ToI32Sat($rB_{0:63}$, SIGN, ROUND, F)

The double-precision floating-point value in **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.

Exceptions:

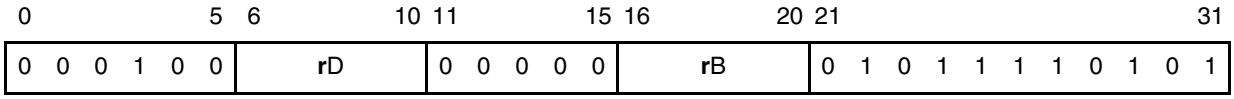
If the contents of **rB** are Infinity, Denorm, or NaN, or if an overflow occurs, SPEFSCR[FINV] is set, and the FG, and FX bits are cleared. If SPEFSCR[FINVE] is set, an interrupt is taken, and the destination register is not updated.

This instruction can signal an inexact status and set SPEFSCR[FINXS] if the conversion is not exact. If the floating-point inexact exception is enabled, an interrupt is taken using the floating-point round interrupt vector. In this case, the destination register is updated with the truncated result, the FG and FX bits are properly updated to allow rounding to be performed in the interrupt handler.

efdctsi | Scalar DPFP APU | User | **efdctsi**

Convert floating-point double-precision to signed integer

efdctsi rD,rB



$rD_{32:63} \leftarrow$ CnvtFP64ToI32Sat($rB_{0:63}$, SIGN, ROUND, I)

The double-precision floating-point value in **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.

Exceptions:

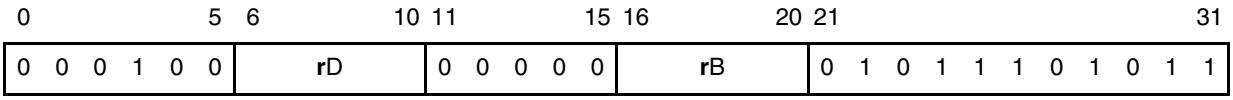
If the contents of **rB** are Infinity, Denorm, or NaN, or if an overflow occurs, SPEFSCR[FINV] is set, and the FG, and FX bits are cleared. If SPEFSCR[FINVE] is set, an interrupt is taken, the destination register is not updated, and no other status bits are set.

This instruction can signal an inexact status and set SPEFSCR[FINXS] if the conversion is not exact. If the floating-point inexact exception is enabled, an interrupt is taken using the floating-point round interrupt vector. In this case, the destination register is updated with the truncated result, the FG and FX bits are properly updated to allow rounding to be performed in the interrupt handler.

efdctsidz | Scalar DPFP APU | User | **efdctsidz**

Convert floating-point double-precision to signed integer doubleword with round toward zero

efdctsidz rD,rB



$rD_{0:63} \leftarrow$ CnvtFP64ToI64Sat($rB_{0:63}$, SIGN, TRUNC)

The double-precision floating-point value in **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.

Exceptions:

If the contents of **rB** are Infinity, Denorm, or NaN, or if an overflow occurs, SPEFSCR[FINV] is set, and the FG, and FX bits are cleared. If SPEFSCR[FINVE] is set, an interrupt is taken, the destination register is not updated, and no other status bits are set.

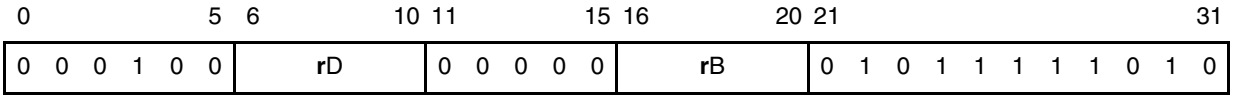
This instruction can signal an inexact status and set SPEFSCR[FINXS] if the conversion is not exact. If the floating-point inexact exception is enabled, an interrupt is taken using the floating-point round interrupt vector. In this case, the destination register is updated with the truncated result, the FG and FX bits are properly updated to allow rounding to be performed in the interrupt handler.

This instruction may only be implemented for 64-bit implementations.

efdctsiz | Scalar DPFP APU | User | **efdctsiz**

Convert floating-point double-precision to signed integer with round toward zero

efdctsiz rD,rB



$rD_{32:63} \leftarrow$ CnvtFP64ToI32Sat($rB_{0:63}$, SIGN, TRUNC, I

The double-precision floating-point value in **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.

Exceptions:

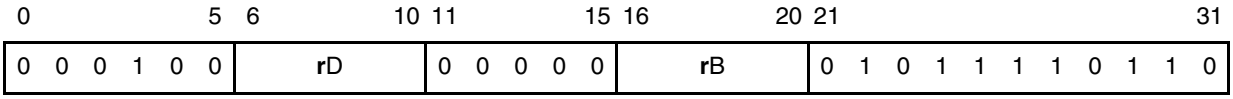
If the contents of **rB** are Infinity, Denorm, or NaN, or if an overflow occurs, SPEFSCR[FINV] is set, and the FG, and FX bits are cleared. If SPEFSCR[FINVE] is set, an interrupt is taken, the destination register is not updated, and no other status bits are set.

This instruction can signal an inexact status and set SPEFSCR[FINXS] if the conversion is not exact. If the floating-point inexact exception is enabled, an interrupt is taken using the floating-point round interrupt vector. In this case, the destination register is updated with the truncated result, the FG and FX bits are properly updated to allow rounding to be performed in the interrupt handler.

efdctuf | Scalar DPFP APU | User | **efdctuf**

Convert floating-point double-precision to unsigned fraction

efdctuf **rD,rB**



$rD_{32:63} \leftarrow$ CnvtFP64ToI32Sat($rB_{0:63}$, UNSIGN, ROUND, F)

The double-precision floating-point value in **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.

Exceptions:

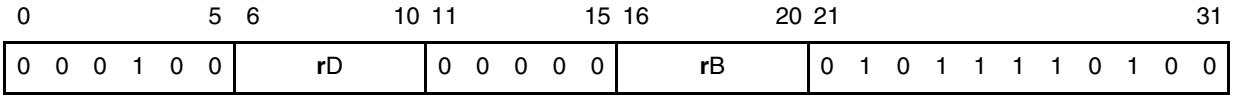
If the contents of **rB** are Infinity, Denorm, or NaN, or if an overflow occurs, SPEFSCR[FINV] is set, and the FG, and FX bits are cleared. If SPEFSCR[FINVE] is set, an interrupt is taken, and the destination register is not updated.

This instruction can signal an inexact status and set SPEFSCR[FINXS] if the conversion is not exact. If the floating-point inexact exception is enabled, an interrupt is taken using the Floating-Point Round Interrupt vector. In this case, the destination register is updated with the truncated result, the FG and FX bits are properly updated to allow rounding to be performed in the interrupt handler.

efdctui | Scalar DPFP APU | User | **efdctui**

Convert floating-point double-precision to unsigned integer

efdctui rD,rB



$rD_{32:63} \leftarrow$ CnvtFP64ToI32Sat($rB_{0:63}$, UNSIGN, ROUND, I

The double-precision floating-point value in **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.

Exceptions:

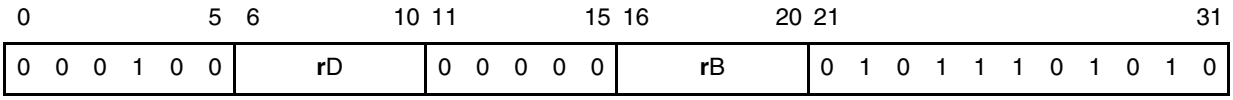
If the contents of **rB** are Infinity, Denorm, or NaN, or if an overflow occurs, SPEFSCR[FINV] is set, and the FG, and FX bits are cleared. If SPEFSCR[FINVE] is set, an interrupt is taken, and the destination register is not updated.

This instruction can signal an inexact status and set SPEFSCR[FINXS] if the conversion is not exact. If the floating-point inexact exception is enabled, an interrupt is taken using the floating-point round interrupt vector. In this case, the destination register is updated with the truncated result, the FG and FX bits are properly updated to allow rounding to be performed in the interrupt handler.

efdctuidz | Scalar DPFP APU | User | **efdctuidz**

Convert floating-point double-precision to unsigned integer doubleword with round toward zero

efdctuidz **rD,rB**



$rD_{0:63} \leftarrow$ CnvtFP64ToI64Sat($rB_{0:63}$, UNSIGN, TRUNC)

The double-precision floating-point value in **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.

Exceptions:

If the contents of **rB** are Infinity, Denorm, or NaN, or if an overflow occurs, SPEFSCR[FINV] is set, and the FG, and FX bits are cleared. If SPEFSCR[FINVE] is set, an interrupt is taken, and the destination register is not updated.

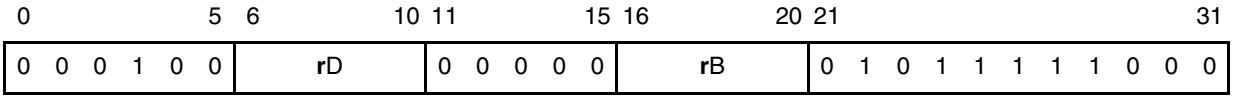
This instruction can signal an inexact status and set SPEFSCR[FINXS] if the conversion is not exact. If the floating-point inexact exception is enabled, an interrupt is taken using the floating-point round interrupt vector. In this case, the destination register is updated with the truncated result, the FG and FX bits are properly updated to allow rounding to be performed in the interrupt handler.

This instruction may only be implemented for 64-bit implementations.

efdctuiz | Scalar DPFP APU | User | **efdctuiz**

Convert floating-point double-precision to unsigned integer with round toward zero

efdctuiz **rD,rB**



$rD_{32:63} \leftarrow$ CnvtFP64ToI32Sat($rB_{0:63}$, UNSIGN, TRUNC, I)

The double-precision floating-point value in **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.

Exceptions:

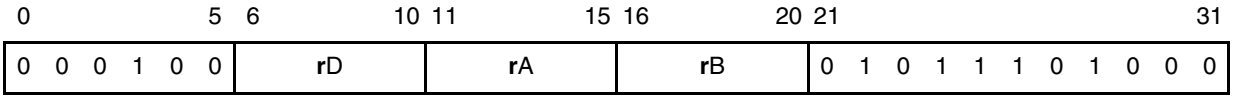
If the contents of **rB** are Infinity, Denorm, or NaN, or if an overflow occurs, SPEFSCR[FINV] is set, and the FG, and FX bits are cleared. If SPEFSCR[FINVE] is set, an interrupt is taken, and the destination register is not updated.

This instruction can signal an inexact status and set SPEFSCR[FINXS] if the conversion is not exact. If the floating-point inexact exception is enabled, an interrupt is taken using the floating-point round interrupt vector. In this case, the destination register is updated with the truncated result, the FG and FX bits are properly updated to allow rounding to be performed in the interrupt handler.

efddiv | Scalar DPFP APU | User | **efddiv**

Floating-point double-precision divide

efddiv **r**D,**r**A,**rB**



$rD_{0:63} \leftarrow rA_{0:63} \div_{dp} rB_{0:63}$

rA is divided by **rB** and the result is stored in **r**D. If **rB** is a NaN or infinity, the result is a properly signed zero. Otherwise, if **rB** is a zero (or a denormalized number optionally transformed to zero by the implementation), or if **r**A is either NaN or infinity, the result is either *pmax* ($a_{sign}==b_{sign}$), or *nmax* ($a_{sign}!=b_{sign}$). Otherwise, if an overflow occurs, *pmax* or *nmax* (as appropriate) is stored in **r**D. If an underflow occurs, +0 or -0 (as appropriate) is stored in **r**D.

Exceptions:

If the contents of **r**A or **rB** are Infinity, Denorm, or NaN, or if both **r**A and **rB** are +/-0, SPEFSCR[FINV] is set. If SPEFSCR[FINVE] is set, an interrupt is taken, and the destination register is not updated. Otherwise, if the content of **rB** is +/-0 and the content of **r**A is a finite normalized non-zero number, SPEFSCR[FDBZ] is set. If floating-point divide by zero Exceptions are enabled, an interrupt is then taken. Otherwise, if an overflow occurs, SPEFSCR[FOVF] is set, or if an underflow occurs, SPEFSCR[FUNF] is set. If either underflow or overflow exceptions are enabled and the corresponding bit is set, an interrupt is taken. If any of these interrupts are taken, the destination register is not updated.

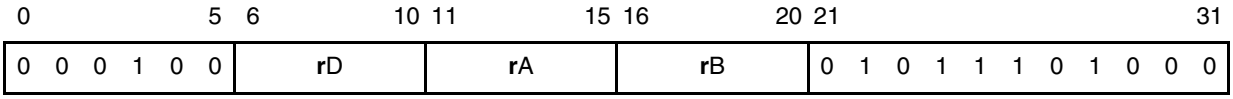
If the result of this instruction is inexact or if an overflow occurs but overflow exceptions are disabled, and no other interrupt is taken, SPEFSCR[FINXS] is set. If the floating-point inexact exception is enabled, an interrupt is taken using the floating-point round interrupt vector. In this case, the destination register is updated with the truncated result, the FG and FX bits are properly updated to allow rounding to be performed in the interrupt handler.

FG and FX are cleared if an overflow, underflow, divide by zero, or invalid operation/input error is signaled, regardless of enabled exceptions.

efdmul | Scalar DPFP APU | User | **efdmul**

Floating-point double-precision multiply

efdmul **r**D,**r**A,**rB**



$rD_{0:63} \leftarrow rA_{0:63} \times_{dp} rB_{0:63}$

rA is multiplied by **rB** and the result is stored in **r**D. If **r**A or **rB** are zero (or a denormalized number optionally transformed to zero by the implementation), the result is a properly signed zero. Otherwise, if **r**A or **rB** are either NaN or infinity, the result is either *pmax* ($a_{sign}==b_{sign}$), or *nmax* ($a_{sign}!=b_{sign}$). Otherwise, if an overflow occurs, *pmax* or *nmax* (as appropriate) is stored in **r**D. If an underflow occurs, +0 or -0 (as appropriate) is stored in **r**D.

Exceptions:

If the contents of **r**A or **rB** are Infinity, Denorm, or NaN, SPEFSCR[FINV] is set. If SPEFSCR[FINVE] is set, an interrupt is taken, and the destination register is not updated. Otherwise, if an overflow occurs, SPEFSCR[FOVF] is set, or if an underflow occurs, SPEFSCR[FUNF] is set. If either underflow or overflow exceptions are enabled and the corresponding bit is set, an interrupt is taken. If any of these interrupts are taken, the destination register is not updated.

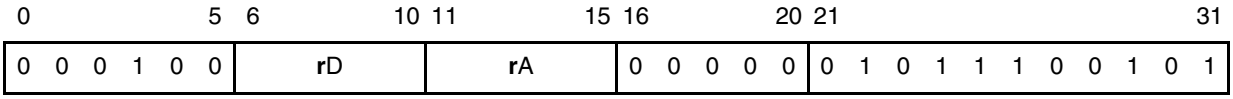
If the result of this instruction is inexact or if an overflow occurs but overflow exceptions are disabled, and no other interrupt is taken, SPEFSCR[FINXS] is set. If the floating-point inexact exception is enabled, an interrupt is taken using the floating-point round interrupt vector. In this case, the destination register is updated with the truncated result, the FG and FX bits are properly updated to allow rounding to be performed in the interrupt handler.

FG and FX are cleared if an overflow, underflow, or invalid operation/input error is signaled, regardless of enabled exceptions.

efdnabs | Scalar DPFP APU | User | **efdnabs**

Floating-point double-precision negative absolute value

efdnabs **r**D,**r**A



$rD_{0:63} \leftarrow 0b1 \parallel rA_{1:63}$

The sign bit of **r**A is set to 1 and the result is placed into **r**D.

Exceptions:

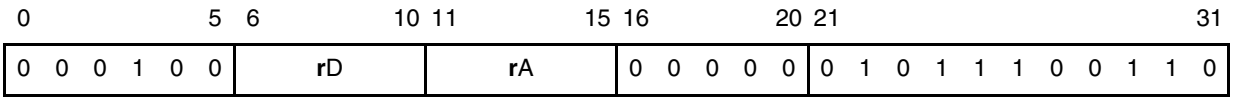
Exception detection for embedded floating-point absolute value operations is implementation dependent. An implementation may choose to not detect exceptions and carry out the sign bit operation. If the implementation does not detect exceptions, or if exception detection is disabled, the computation can be carried out in one of two ways, as a sign bit operation ignoring the rest of the contents of the source register, or by examining the input and appropriately saturating the input prior to performing the operation.

If an implementation chooses to handle exceptions, the exception is handled as follows: If **r**A is Infinity, Denorm, or NaN, SPEFSCR[FINV] is set, and FG and FX are cleared. If floating-point invalid input exceptions are enabled, an interrupt is taken and the destination register is not updated.

efdneg | Scalar DPFP APU | User | **efdneg**

Floating-point double-precision negate

efdneg rD,rA



$rD_{0:63} \leftarrow \neg rA_0 \,||\, rA_{1:63}$

The sign bit of **r**A is complemented and the result is placed into **r**D.

Exceptions:

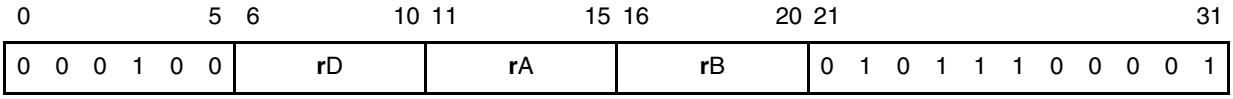
Exception detection for embedded floating-point absolute value operations is implementation dependent. An implementation may choose to not detect exceptions and carry out the sign bit operation. If the implementation does not detect exceptions, or if exception detection is disabled, the computation can be carried out in one of two ways, as a sign bit operation ignoring the rest of the contents of the source register, or by examining the input and appropriately saturating the input prior to performing the operation.

If an implementation chooses to handle exceptions, the exception is handled as follows: If **r**A is Infinity, Denorm, or NaN, SPEFSCR[FINV] is set, and FG and FX are cleared. If floating-point invalid input exceptions are enabled, an interrupt is taken and the destination register is not updated.

efdsub | Scalar DPFP APU | User | **efdsub**

Floating-point double-precision subtract

efdsub **r**D,**r**A,**rB**



$rD_{0:63} \leftarrow rA_{0:63}$ -$_{dp}$ $rB_{0:63}$

rB is subtracted from rA and the result is stored in **r**D. If **r**A is NaN or infinity, the result is either *pmax* (a_{sign}==0), or *nmax* (a_{sign}==1). Otherwise, If **rB** is NaN or infinity, the result is either *nmax* (b_{sign}==0), or *pmax* (b_{sign}==1). Otherwise, if an overflow occurs, *pmax* or *nmax* (as appropriate) is stored in **r**D. If an underflow occurs, +0 (for rounding modes RN, RZ, RP) or -0 (for rounding mode RM) is stored in **r**D.

Exceptions:

If the contents of rA or **rB** are Infinity, Denorm, or NaN, SPEFSCR[FINV] is set. If SPEFSCR[FINVE] is set, an interrupt is taken, and the destination register is not updated. Otherwise, if an overflow occurs, SPEFSCR[FOVF] is set, or if an underflow occurs, SPEFSCR[FUNF] is set. If either underflow or overflow exceptions are enabled and the corresponding bit is set, an interrupt is taken. If any of these interrupts are taken, the destination register is not updated.

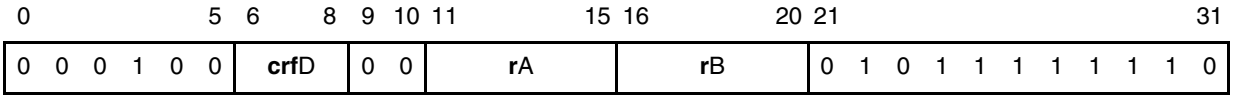
If the result of this instruction is inexact or if an overflow occurs but overflow exceptions are disabled, and no other interrupt is taken, SPEFSCR[FINXS] is set. If the floating-point inexact exception is enabled, an interrupt is taken using the floating-point round interrupt vector. In this case, the destination register is updated with the truncated result, the FG and FX bits are properly updated to allow rounding to be performed in the interrupt handler.

FG and FX are cleared if an overflow, underflow, or invalid operation/input error is signaled, regardless of enabled exceptions.

efdtsteq | Scalar DPFP APU | User | **efdtsteq**

Floating-point double-precision test equal

efdtsteq **crf**D,**r**A,**rB**



al ← $rA_{0:63}$
bl ← $rB_{0:63}$
if (al = bl) then cl ← 1
else cl ← 0
$CR_{4*crD:4*crD+3}$ ← undefined || cl || undefined || undefined

rA is compared against **rB**. If **r**A is equal to **rB**, the bit in the **crf**D is set, otherwise it is cleared. 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.

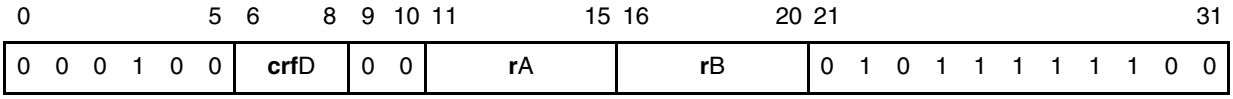
No exceptions are generated during the execution of **efdtsteq** If strict IEEE 754 compliance is required, the program should use **efdcmpeq**.

Implementation note: In an implementation, the execution of **efdtsteq** is likely to be faster than the execution of **efdcmpeq**.

efdtstgt | Scalar DPFP APU | User | **efdtstgt**

Floating-point double-precision test greater than

efdtstgt **crf**D,**r**A,**rB**



al ← $rA_{0:63}$
bl ← $rB_{0:63}$
if (al > bl) then cl ← 1
else cl ← 0
$CR_{4*crD:4*crD+3}$ ← undefined || cl || undefined || undefined

rA is compared against **r**B. If **r**A is greater than **r**B, the bit in the **crf**D is set, otherwise it is cleared. 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.

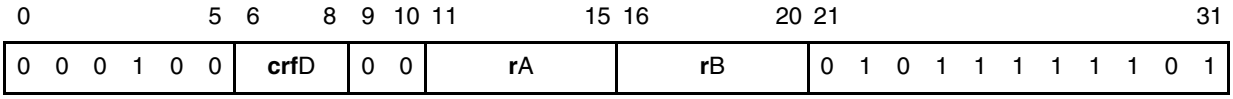
No exceptions are generated during the execution of **efdtstgt**. If strict IEEE 754 compliance is required, the program should use **efdcmpgt**.

*Note: Implementation note: In an implementation, the execution of **efdtstgt** is likely to be faster than the execution of **efdcmpgt**.*

efdtstlt | Scalar DPFP APU | User | **efdtstlt**

Floating-point double-precision test less than

efdtstlt **crf**D,**r**A,**rB**



al ← $rA_{0:63}$
bl ← $rB_{0:63}$
if (al < bl) then cl ← 1
else cl ← 0
$CR_{4*crD:4*crD+3}$ ← undefined || cl || undefined || undefined

rA is compared against **r**B. If **r**A is less than **r**B, the bit in the **crf**D is set, otherwise it is cleared. 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.

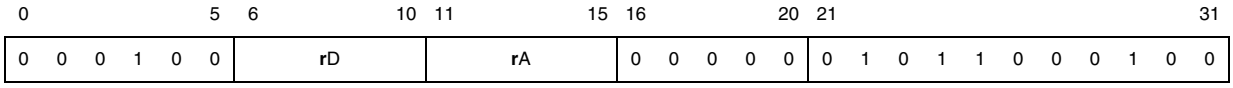
No exceptions are generated during the execution of **efdtstlt**. If strict IEEE 754 compliance is required, the program should use **efdcmplt**.

Implementation note: In an implementation, the execution of **efdtstlt** is likely to be faster than the execution of **efdcmplt**.

efsabs | Scalar SPFP APU | User | **efsabs**

Floating-Point Absolute Value

efsabs **r**D,**r**A



$rD_{32:63} \leftarrow 0b0 \parallel rA_{33:63}$

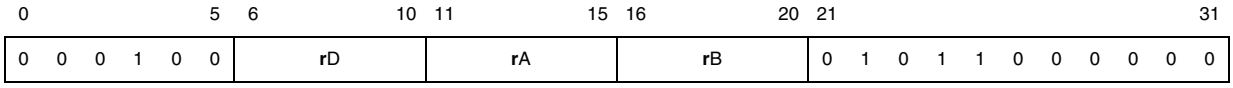
The sign bit of **r**A is cleared and the result is placed into **r**D.

It is implementation dependent if invalid values for rA (NaN, Denorm, Infinity) are detected and exceptions are taken.

efsadd | Scalar SPFP APU | User | **efsadd**

Floating-Point Add

efsadd **r**D,**r**A,**r**B



$$rD_{32:63} \leftarrow rA_{32:63} +_{sp} rB_{32:63}$$

The single-precision floating-point value of **r**A is added to **r**B and the result is stored in **r**D.

If an overflow condition is detected or the contents of **r**A or **r**B are NaN or Infinity, the result is an appropriately signed maximum floating-point value.

If an underflow condition is detected, the result is an appropriately signed floating-point 0.

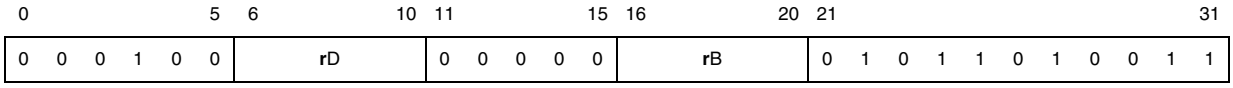
The following status bits are set in the SPEFSCR:

- FINV if the contents of **r**A or **r**B are +infinity, –infinity, denorm, or NaN
- FOFV if an overflow occurs
- FUNF if an underflow occurs
- FINXS, FG, FX if the result is inexact or overflow occurred and overflow exceptions are disabled

efscfsf | Scalar SPFP APU | User | **efscfsf**

Convert Floating-Point from Signed Fraction

efscfsf **r**D,**r**B



$rD_{32:63} \leftarrow$ CnvtI32ToFP32Sat($rB_{32:63}$, SIGN, LOWER, F)

The signed fractional value in **r**B is converted to the nearest single-precision floating-point value using the current rounding mode and placed into **r**D.

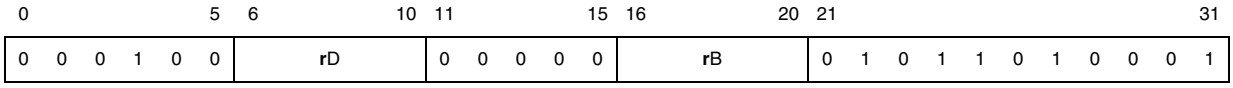
The following status bits are set in the SPEFSCR:

- FINXS, FG, FX if the result is inexact

efscfsi | Scalar SPFP APU | User | **efscfsi**

Convert Floating-Point from Signed Integer

efscfsi **r**D,**r**B



$rD_{32:63} \leftarrow$ CnvtSI32ToFP32Sat($rB_{32:63}$, SIGN, LOWER, I)

The signed integer value in **r**B is converted to the nearest single-precision floating-point value using the current rounding mode and placed into **r**D.

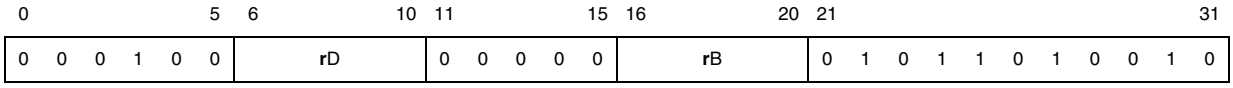
The following status bits are set in the SPEFSCR:

- FINXS, FG, FX if the result is inexact

efscfuf | Scalar SPFP APU | User | **efscfuf**

Convert Floating-Point from Unsigned Fraction

efscfuf **r**D,**r**B



$rD_{32:63} \leftarrow$ Cnvtl32ToFP32Sat($rB_{32:63}$, UNSIGN, LOWER, F)

The unsigned fractional value in **r**B is converted to the nearest single-precision floating-point value using the current rounding mode and placed into **r**D.

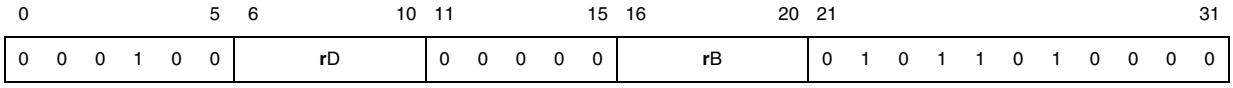
The following status bits are set in the SPEFSCR:

- FINXS, FG, FX if the result is inexact

efscfui | Scalar SPFP APU | User | **efscfui**

Convert Floating-Point from Unsigned Integer

efscfui **r**D,**r**B



$rD_{32:63} \leftarrow$ Cnvtl32ToFP32Sat($rB_{32:63}$, UNSIGN, LOWER, I)

The unsigned integer value in **r**B is converted to the nearest single-precision floating-point value using the current rounding mode and placed into **r**D.

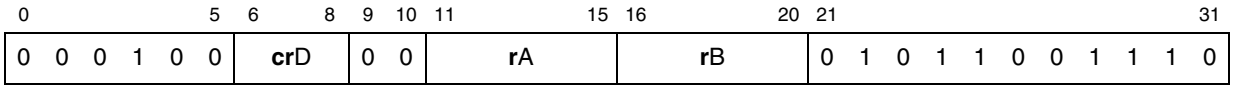
The following status bits are set in the SPEFSCR:

- FINXS, FG, FX if the result is inexact

efscmpeq | Scalar SPFP APU | User | **efscmpeq**

Floating-Point Compare Equal

efscmpeq **cr**D,**r**A,**r**B



al ← $rA_{32:63}$
bl ← $rB_{32:63}$
if (al = bl) then cl ← 1
else cl ← 0
$CR_{4*crD:4*crD+3}$ ← undefined || cl || undefined || undefined

The value in **r**A is compared against **r**B. If **r**A equals **r**B, the **cr**D bit is set, otherwise it is cleared. Comparison ignores the sign of 0 (+0 = –0).

If either operand contains a NaN, infinity, or a denorm and floating-point invalid exceptions are enabled in the SPEFSCR, the exception is taken. If the exception is not enabled, the comparison treats NaNs, infinities, and denorms as normalized numbers.

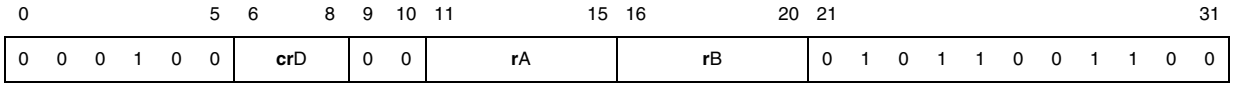
The following status bits are set in SPEFSCR:

- FINV if the contents of **r**A or **r**B are +infinity, –infinity, denorm or NaN

efscmpgt | Scalar SPFP APU | User | **efscmpgt**

Floating-Point Compare Greater Than

efscmpgt **cr**D,**r**A,**r**B



$a \leftarrow rA_{32:63}$
$b \leftarrow rB_{32:63}$
if (a > b) then $c \leftarrow 1$
else $c \leftarrow 0$
$CR_{4*crD:4*crD+3} \leftarrow$ undefined || c || undefined || undefined

The value in **r**A is compared against **r**B. If **r**A is greater than **r**B, the bit in the **cr**D is set, otherwise it is cleared. Comparison ignores the sign of 0 (+0 = –0).

If either operand contains a NaN, infinity, or a denorm and floating-point invalid exceptions are enabled in the SPEFSCR, the exception is taken. If the exception is not enabled, the comparison treats NaNs, infinities, and denorms as normalized numbers.

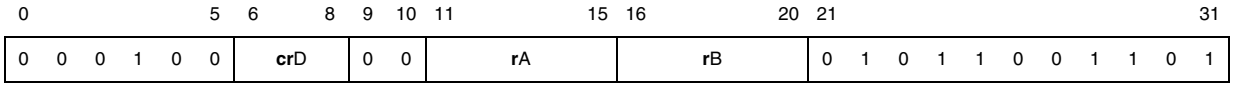
The following status bits are set in SPEFSCR:

- FINV if the contents of **r**A or **r**B are +infinity, –infinity, denorm or NaN

efscmplt | Scalar SPFP APU | User | **efscmplt**

Floating-Point Compare Less Than

efscmplt **cr**D,**r**A,**r**B



al ← $rA_{32:63}$
bl ← $rB_{32:63}$
if (al < bl) then cl ← 1
else cl ← 0
$CR_{4*crD:4*crD+3}$ ← undefined || cl || undefined || undefined

The value in **r**A is compared against **r**B. If **r**A is less than **r**B, the bit in the **cr**D is set, otherwise it is cleared. Comparison ignores the sign of 0 (+0 = –0).

If either operand contains a NaN, infinity, or a denorm and floating-point invalid exceptions are enabled in the SPEFSCR, the exception is taken. If the exception is not enabled, the comparison treats NaNs, infinities, and denorms as normalized numbers.

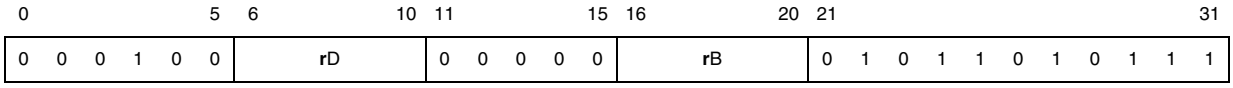
The following status bits are set in SPEFSCR:

- FINV if the contents of **r**A or **r**B are +infinity, –infinity, denorm or NaN

efsctsf | Scalar SPFP APU | User | **efsctsf**

Convert Floating-Point to Signed Fraction

efsctsf **r**D,**r**B



$rD_{32:63} \leftarrow$ CnvtFP32ToISat($rB_{32:63}$, SIGN, LOWER, ROUND, F)

The single-precision floating-point value in **r**B is converted to a signed fraction using the current rounding mode. The result saturates if it cannot be represented in a 32-bit fraction. NaNs are converted to 0.

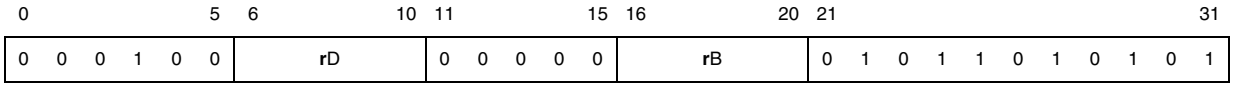
The following status bits are set in the SPEFSCR:

- FINV if the contents of **r**B are +infinity., –infinity, denorm, or NaN, or **r**B cannot be represented in the target format
- FINXS, FG, FX if the result is inexact

efsctsi | Scalar SPFP APU | User | **efsctsi**

Convert Floating-Point to Signed Integer

efsctsi **r**D,**r**B



$rD_{32:63} \leftarrow$ CnvtFP32ToISat($rB_{32:63}$, SIGN, LOWER, ROUND, I)

The single-precision floating-point value in **r**B is converted to a signed integer using the current rounding mode. The result saturates if it cannot be represented in a 32-bit integer. NaNs are converted to 0.

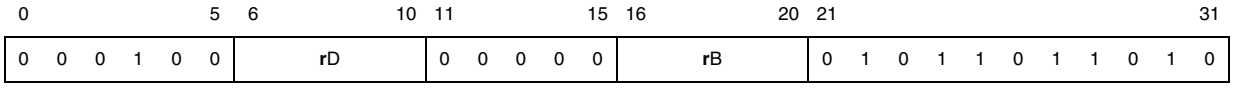
The following status bits are set in the SPEFSCR:

- FINV if the contents of **r**B are +infinity, –infinity, denorm, or NaN, or **r**B cannot be represented in the target format
- FINXS, FG, FX if the result is inexact

efsctsiz | Scalar SPFP APU | User | **efsctsiz**

Convert Floating-Point to Signed Integer with Round toward Zero

efsctsiz **r**D,**r**B



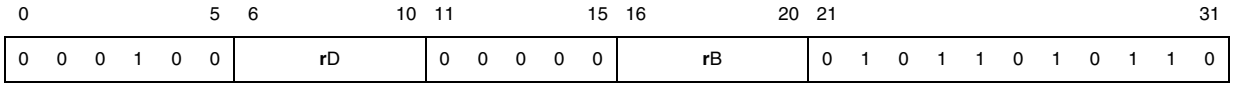
$rD_{32–63} \leftarrow$ CnvtFP32ToISat($rB_{32:63}$, SIGN, LOWER, TRUNC, I)

The single-precision floating-point value in **r**B is converted to a signed integer using the rounding mode Round towards Zero. The result saturates if it cannot be represented in a 32-bit integer. NaNs are converted to 0.

efsctuf | Scalar SPFP APU | User | **efsctuf**

Convert Floating-Point to Unsigned Fraction

efsctuf **r**D,**r**B



$rD_{32:63} \leftarrow$ CnvtFP32ToISat($rB_{32:63}$, UNSIGN, LOWER, ROUND, F)

The single-precision floating-point value in **r**B is converted to an unsigned fraction using the current rounding mode. The result saturates if it cannot be represented in a 32-bit unsigned fraction. NaNs are converted to 0.

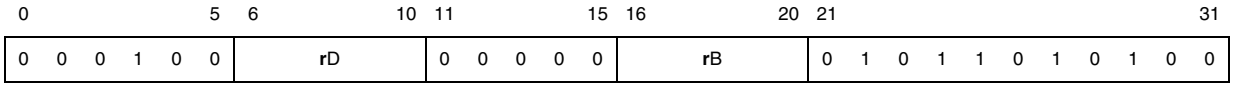
The following status bits are set in the SPEFSCR:

- FINV if the contents of **r**B are +infinity, –infinity, denorm, or NaN, or **r**B cannot be represented in the target format
- FINXS, FG, FX if the result is inexact

efsctui | Scalar SPFP APU | User | **efsctui**

Convert Floating-Point to Unsigned Integer

efsctui **r**D,**r**B



$rD_{32:63} \leftarrow$ CnvtFP32ToISat($rB_{32:63}$, UNSIGN, LOWER, ROUND, I)

The single-precision floating-point value in **r**B is converted to an unsigned integer using the current rounding mode. The result saturates if it cannot be represented in a 32-bit unsigned integer. NaNs are converted to 0.

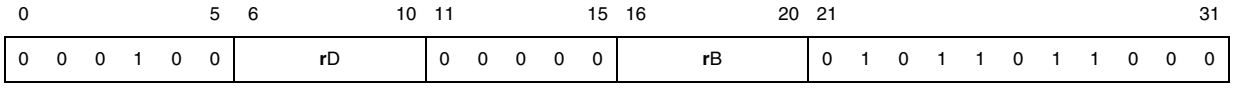
The following status bits are set in the SPEFSCR:

- FINV if the contents of **r**B are +infinity, –infinity, denorm, or NaN, or **r**B cannot be represented in the target format
- FINXS, FG, FX if the result is inexact

efsctuiz | Scalar SPFP APU | User | **efsctuiz**

Convert Floating-Point to Unsigned Integer with Round toward Zero

efsctuiz **r**D,**r**B



$rD_{32:63} \leftarrow$ CnvtFP32ToISat($rB_{32:63}$, UNSIGN, LOWER, TRUNC, I)

The single-precision floating-point value in **r**B is converted to an unsigned integer using the rounding mode Round toward Zero. The result saturates if it cannot be represented in a 32-bit unsigned integer. NaNs are converted to 0.

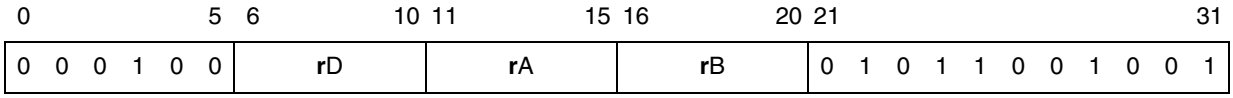
The following status bits are set in the SPEFSCR:

- FINV if the contents of **r**B are +infinity, –infinity, denorm, or NaN, or **r**B cannot be represented in the target format
- FINXS, FG, FX if the result is inexact

efsdiv | Scalar SPFP APU | User | **efsdiv**

Floating-Point Divide

efsdiv **r**D,**r**A,**r**B



$rD_{32:63} \leftarrow rA_{32:63} \div_{sp} rB_{32:63}$

The single-precision floating-point value in **r**A is divided by **r**B and the result is stored in **r**D.

If an overflow is detected, or **r**B is a denorm (or 0 value), or **r**A is a NaN or Infinity and **r**B is a normalized number, the result is an appropriately signed maximum floating-point value.

If an underflow is detected or rB is a NaN or Infinity, the result is an appropriately signed floating-point 0.

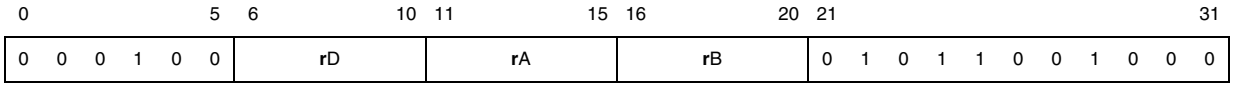
The following status bits are set in the SPEFSCR:

- FINV if the contents of **r**A or **r**B are +infinity, –infinity, denorm, or NaN
- FOFV if an overflow occurs
- FUNV if an underflow occurs
- FDBZS, FDBZ if a divide by zero occurs
- FINXS, FG, FX if the result is inexact or overflow occurred and overflow exceptions are disabled

efsmul | Scalar SPFP APU | User | **efsmul**

Floating-Point Multiply

efsmul **r**D,**r**A,**r**B



$$rD_{32:63} \leftarrow rA_{32:63} \times_{sp} rB_{32:63}$$

The single-precision floating-point value in **r**A is multiplied by **r**B and the result is stored in **r**D.

If an overflow is detected the result is an appropriately signed maximum floating-point value.

If one of **r**A or **r**B is a NaN or an Infinity and the other is not a denorm or zero, the result is an appropriately signed maximum floating-point value.

If an underflow is detected, or **r**A or **r**B is a denorm, the result is an appropriately signed floating-point 0.

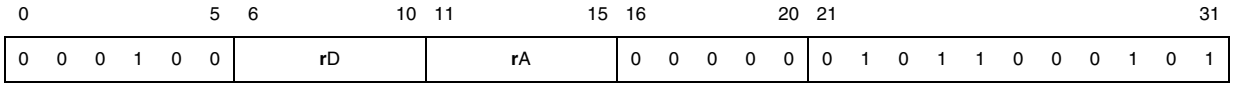
The following status bits are set in the SPEFSCR:

- FINV if the contents of **r**A or **r**B are +infinity, –infinity, denorm, or NaN
- FOFV if an overflow occurs
- FUNV if an underflow occurs
- FINXS, FG, FX if the result is inexact or overflow occurred and overflow exceptions are disabled

efsnabs | Scalar SPFP APU | User | **efsnabs**

Floating-Point Negative Absolute Value

efsnabs **r**D,**r**A



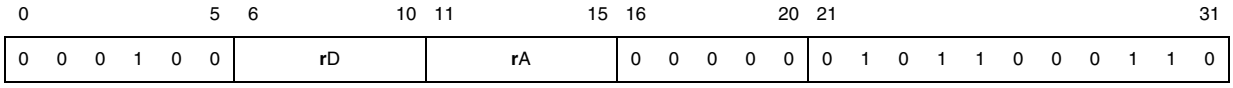
$rD_{32:63} \leftarrow 0b1 \parallel rA_{33:63}$

The sign bit of **r**A is set and the result is stored in **r**D. It is implementation dependent if invalid values for **r**A (NaN, Denorm, Infinity) are detected and exceptions are taken.

efsneg | Scalar SPFP APU | User | **efsneg**

Floating-Point Negate

efsneg **r**D,**r**A



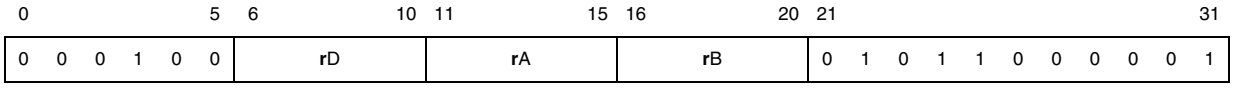
$rD_{32:63} \leftarrow \neg rA_{32} \,||\, rA_{33:63}$

The sign bit of **r**A is complemented and the result is stored in **r**D. It is implementation dependent if invalid values for **r**A (NaN, Denorm, Infinity) are detected and exceptions are taken.

efssub | Scalar SPFP APU | User | **efssub**

Floating-Point Subtract

efssub **r**D,**r**A,**r**B



$rD_{32:63} \leftarrow rA_{32:63}$ -$_{sp}$ $rB_{32:63}$

The single-precision floating-point value in **r**B is subtracted from that in **r**A and the result is stored in **r**D.

If an overflow condition is detected or the contents of **r**A or **r**B are NaN or Infinity, the result is an appropriately signed maximum floating-point value.

If an underflow condition is detected, the result is an appropriately signed floating-point 0.

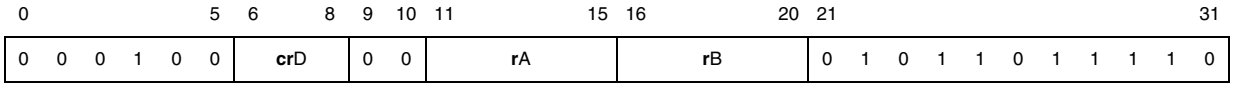
The following status bits are set in the SPEFSCR:

- FINV if the contents of **r**A or **r**B are +infinity, –infinity, denorm, or NaN
- FOFV if an overflow occurs
- FUNF if an underflow occurs
- FINXS, FG, FX if the result is inexact or overflow occurred and overflow exceptions are disabled

efststeq | Scalar SPFP APU | User | **efststeq**

Floating-Point Test Equal

efststeq **cr**D,**r**A,**r**B



al ← $rA_{32:63}$
bl ← $rB_{32:63}$
if (al = bl) then cl ← 1
else cl ← 0
$CR_{4*crD:4*crD+3}$ ← undefined || cl || undefined || undefined

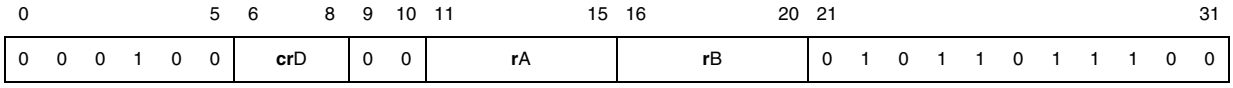
The value in **r**A is compared against **r**B. If **r**A equals **r**B, the bit in **cr**D is set, otherwise it is cleared. Comparison ignores the sign of 0 (+0 = –0). The comparison treats NaNs, infinities, and denorms as normalized numbers.

No exceptions are taken during execution of **efststeq**. If strict IEEE 754 compliance is required, the program should use **efscmpeq**.

efststgt | Scalar SPFP APU | User | **efststgt**

Floating-Point Test Greater Than

efststgt **cr**D,**r**A,**r**B



al ← $rA_{32:63}$
bl ← $rB_{32:63}$
if (al > bl) then cl ← 1
else cl ← 0
$CR_{4*crD:4*crD+3}$ ← undefined || cl || undefined || undefined

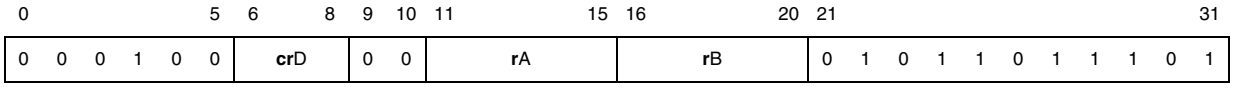
If **r**A is greater than **r**B, the bit in **cr**D is set, otherwise it is cleared. Comparison ignores the sign of 0 (+0 = –0). The comparison treats NaNs, infinities, and denorms as normalized numbers.

No exceptions are taken during the execution of **efststgt**. If strict IEEE 754 compliance is required, the program should use **efscmpgt**.

efststlt | Scalar SPFP APU | User | **efststlt**

Floating-Point Test Less Than

efststlt **cr**D,**r**A,**r**B



al ← $rA_{32:63}$
bl ← $rB_{32:63}$
if (al < bl) then cl ← 1
else cl ← 0
$CR_{4*crD:4*crD+3}$ ← undefined || cl || undefined || undefined

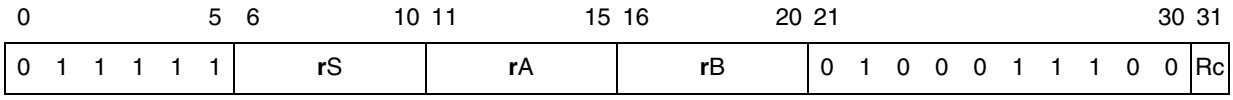
If **r**A is less than **r**B, the bit in the **cr**D is set, otherwise it is cleared. Comparison ignores the sign of 0 (+0 = –0). The comparison treats NaNs, infinities, and denorms as normalized numbers.

No exceptions are taken during the execution of **efststlt**. If strict IEEE 754 compliance is required, the program should use **efscmplt**.

eqv | Book E | User | **eqv**

Equivalent

eqv **r**A,**r**S,**r**B (Rc=0)
eqv. **r**A,**r**S,**r**B (Rc=1)



result$_{0:63}$ ← rS ≡ rB
if Rc=1 then do
LT ← result$_{32:63}$ < 0
GT ← result$_{32:63}$ > 0
EQ ← result$_{32:63}$ = 0
CR0 ← LT || GT || EQ || SO
rA ← result

The contents of **r**S are XORed with the contents of **r**B and the one's complement of the result is placed into **r**A.

Other registers altered:

- CR0 (if Rc=1)

evabs | SPE APU | User | **evabs**

Vector absolute value

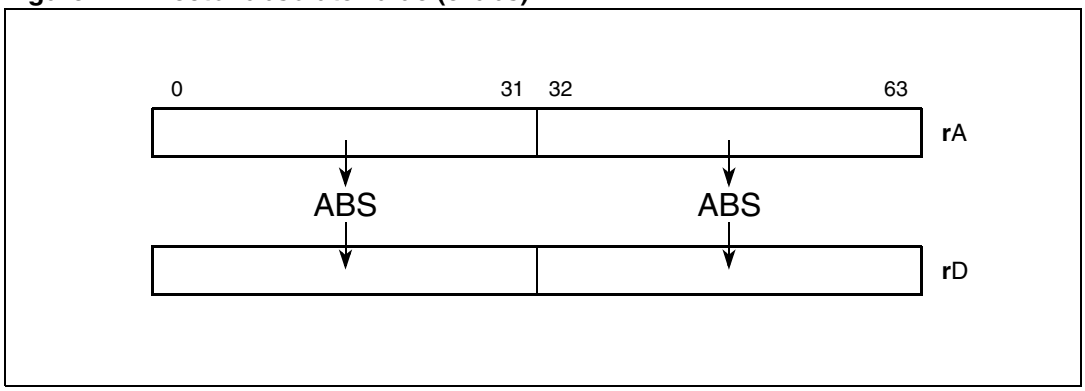
evabs **r**D,**r**A

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	0	1	0	0	**r**D					**r**A					0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0

$rD_{0:31} \leftarrow ABS(rA_{0:31})$
$rD_{32:63} \leftarrow ABS(rA_{32:63})$

The absolute value of each element of **r**A is placed in the corresponding elements of **r**D. An absolute value of 0x8000_0000 (most negative number) returns 0x8000_0000. No overflow is detected.

Figure 24. Vector absolute value (evabs)



evaddiw | SPE APU | User | **evaddiw**

Vector add immediate word

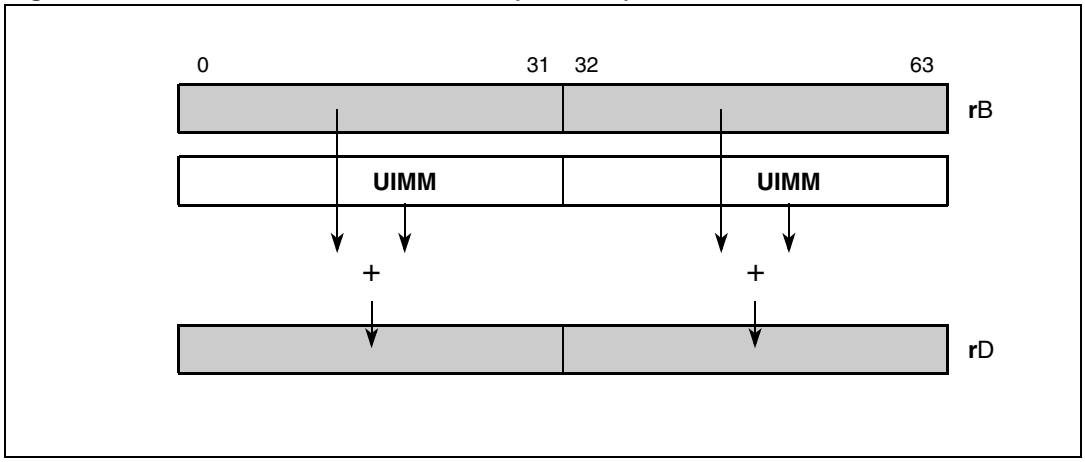
evaddiw **r**D,**r**B,UIMM

0	5	6	10	11	15	16	20	21	31
0 0 0 1 0 0		**r**D		UIMM		**r**B		0 1 0 0 0 0 0 0 0 1 0	

$rD_{0:31} \leftarrow rB_{0:31} + EXTZ(UIMM$) // Modulo sum
$rD_{32:63} \leftarrow rB_{32:63} + EXTZ(UIMM)$ // Modulo sum

UIMM is zero-extended and added to both the high and low elements of **r**B and the results are placed in **r**D. Note that the same value is added to both elements of the register. UIMM is 5 bits.

Figure 25. Vector add immediate word (evaddiw)



evaddsmiaaw | SPE APU | User | **evaddsmiaaw**

Vector add signed, modulo, integer to accumulator word

evaddsmiaaw **r**D,**r**A

0	5	6	10	11	15	16	20	21	31
0 0 0 1 0 0		**r**D		**r**A		0 0 0 0 0		1 0 0 1 1 0 0 1 0 0 1	

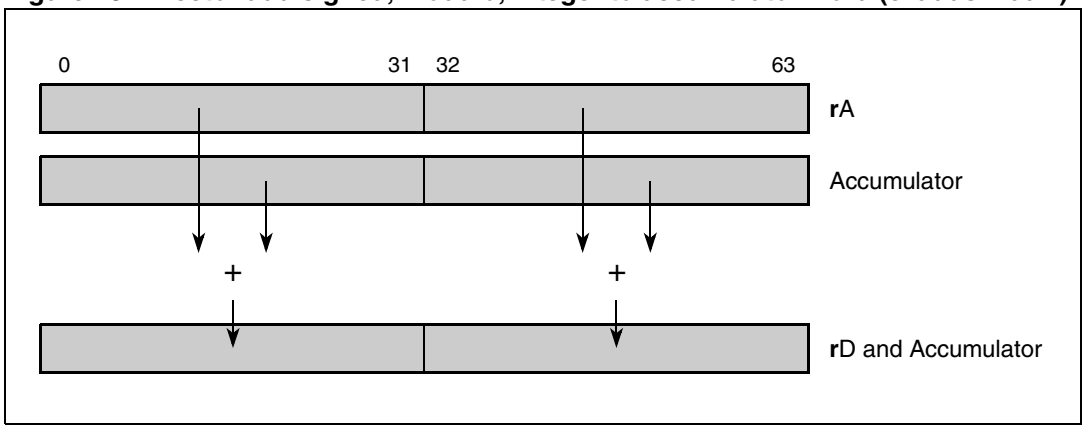
$rD_{0:31} \leftarrow ACC_{0:31} + rA_{0:31}$
$rD_{32:63} \leftarrow ACC_{32:63} + rA_{32:63}$

$ACC_{0:63} \leftarrow rD_{0:63}$

Each word element in **r**A is added to the corresponding element in the accumulator and the results are placed in **r**D and into the accumulator.

Other registers altered: ACC

Figure 26. Vector add signed, modulo, integer to accumulator word (evaddsmiaaw)



evaddssiaaw | SPE APU | User | **evaddssiaaw**

Vector add signed, saturate, integer to accumulator word

evaddssiaaw **r**D,**r**A

0	5	6	10	11	15	16	20	21	31
0 0 0 1 0 0		**r**D		**r**A		0 0 0 0 0		1 0 0 1 1 0 0 0 0 0 1	

// high
$temp_{0:63} \leftarrow EXTS(ACC_{0:31}) + EXTS(rA_{0:31})$
$ovh \leftarrow temp_{31} \oplus temp_{32}$
$rD_{0:31} \leftarrow SATURATE(ovh, temp_{31}, 0x80000000, 0x7fffffff, temp_{32:63})$

// low
$temp_{0:63} \leftarrow EXTS(ACC_{32:63}) + EXTS(rA_{32:63})$
$ovl \leftarrow temp_{31} \oplus temp_{32}$
$rD_{32:63} \leftarrow SATURATE(ovl, temp_{31}, 0x80000000, 0x7fffffff, temp_{32:63})$

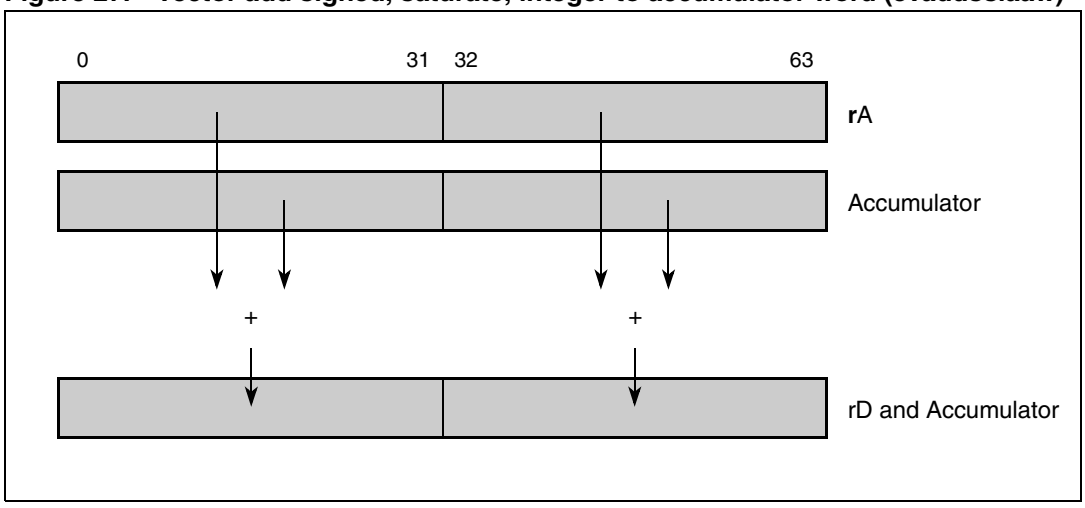
$ACC_{0:63} \leftarrow rD_{0:63}$

$SPEFSCR_{OVH} \leftarrow ovh$
$SPEFSCR_{OV} \leftarrow ovl$
$SPEFSCR_{SOVH} \leftarrow SPEFSCR_{SOVH} \mid ovh$
$SPEFSCR_{SOV} \leftarrow SPEFSCR_{SOV} \mid ovl$

Each signed integer word element in **r**A is sign-extended and added to the corresponding sign-extended element in the accumulator, saturating if overflow or underflow occurs, and the results are placed in **r**D and the accumulator. Any overflow or underflow is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR ACC

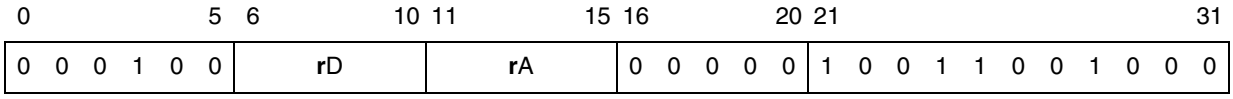
Figure 27. Vector add signed, saturate, integer to accumulator word (evaddssiaaw)



evaddumiaaw | SPE APU | User | **evaddumiaaw**

Vector add unsigned, modulo, integer to accumulator word

evaddumiaaw **r**D,**r**A



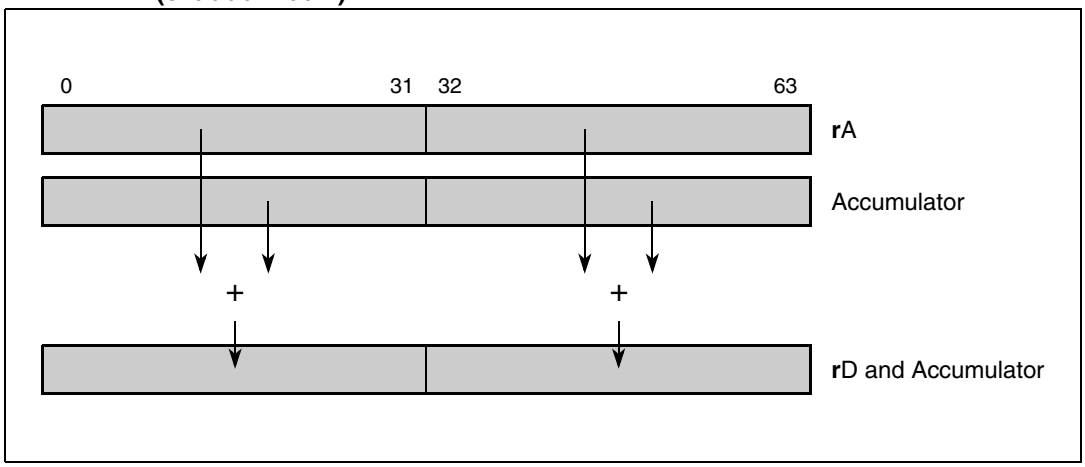
$rD_{0:31} \leftarrow ACC_{0:31} + rA_{0:31}$
$rD_{32:63} \leftarrow ACC_{32:63} + rA_{32:63}$

$ACC_{0:63} \leftarrow rD_{0:63}$

Each unsigned integer word element in **r**A is added to the corresponding element in the accumulator and the results are placed in **r**D and the accumulator.

Other registers altered: ACC

Figure 28. Vector add unsigned, modulo, integer to accumulator word (evaddumiaaw)



evaddusiaaw | SPE APU | User | **evaddusiaaw**

Vector add unsigned, saturate, integer to accumulator word

evaddusiaaw **r**D,**r**A

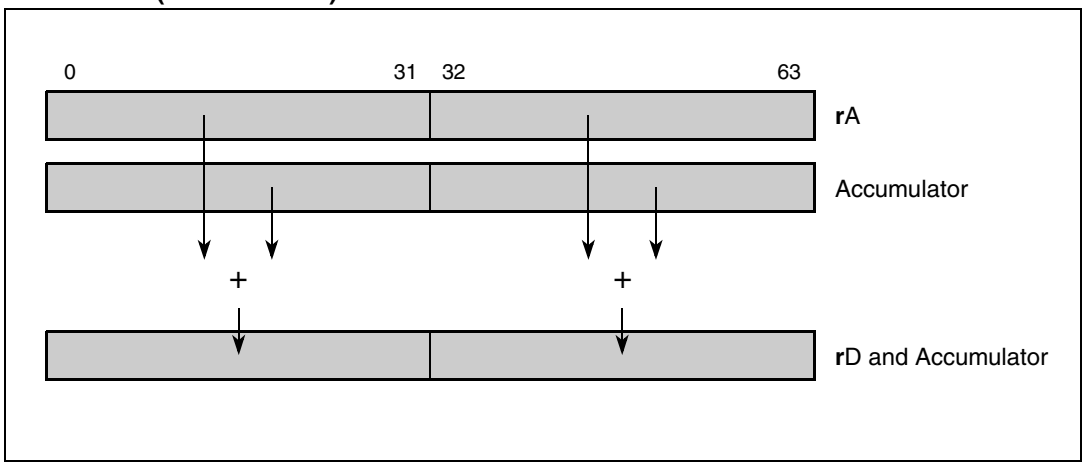
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	0	1	0	0	**r**D					**r**A					0	0	0	0	0	1	0	0	1	1	0	0	0	0	0	0

```
// high
temp0:63 ← EXTZ(ACC0:31) + EXTZ(rA0:31)
ovh ← temp31
rD0:31 ← SATURATE(ovh, temp31, 0xffffffff, 0xffffffff, temp32:63)

// low
temp0:63 ← EXTZ(ACC32:63) + EXTZ(rA32:63)
ovl ← temp31
rD32:63 ← SATURATE(ovl, temp31, 0xffffffff, 0xffffffff, temp32:63)

ACC0:63 ← rD0:63

SPEFSCROVH ← ovh
SPEFSCROV ← ovl
SPEFSCRSOVH ← SPEFSCRSOVH | ovh
SPEFSCRSOV ← SPEFSCRSOV | ovl
```

Each unsigned integer word element in **r**A is zero-extended and added to the corresponding zero-extended element in the accumulator, saturating if overflow occurs, and the results are placed in **r**D and the accumulator. Any overflow is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR ACC

Figure 29. Vector add unsigned, saturate, integer to accumulator word (evaddusiaaw)



evaddw | SPE APU | User | **evaddw**

Vector add word

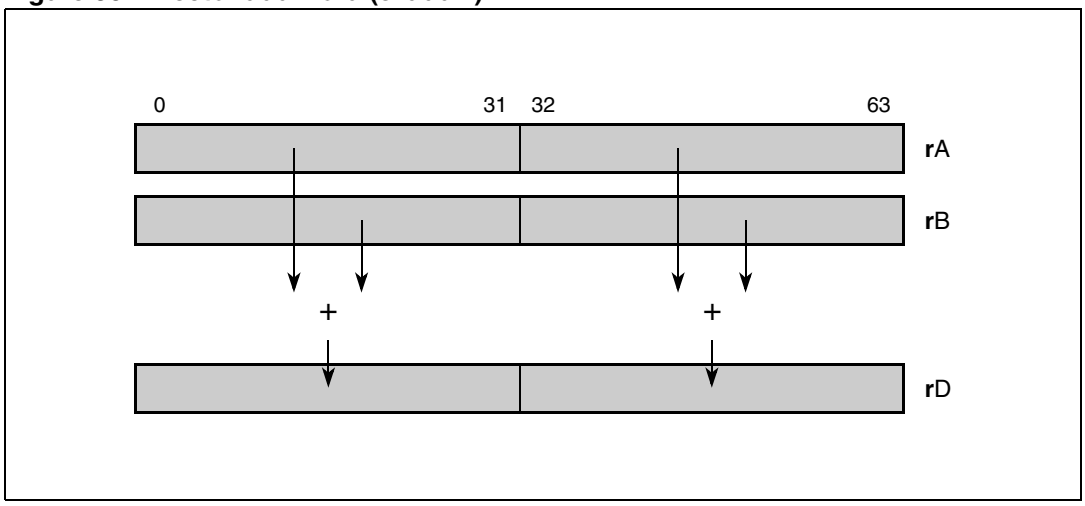
evaddw **r**D,**r**A,**r**B

0	5	6	10	11	15	16	20	21	31
0 0 0 1 0 0		**r**D		**r**A		**r**B		0 1 0 0 0 0 0 0 0 0 0	

$rD_{0:31} \leftarrow rA_{0:31} + rB_{0:31}$ // Modulo sum
$rD_{32:63} \leftarrow rA_{32:63} + rB_{32:63}$ // Modulo sum

The corresponding elements of **r**A and **r**B are added and the results are placed in **r**D. The sum is a modulo sum.

Figure 30. Vector add word (evaddw)



evand | SPE APU | User | **evand**

Vector AND

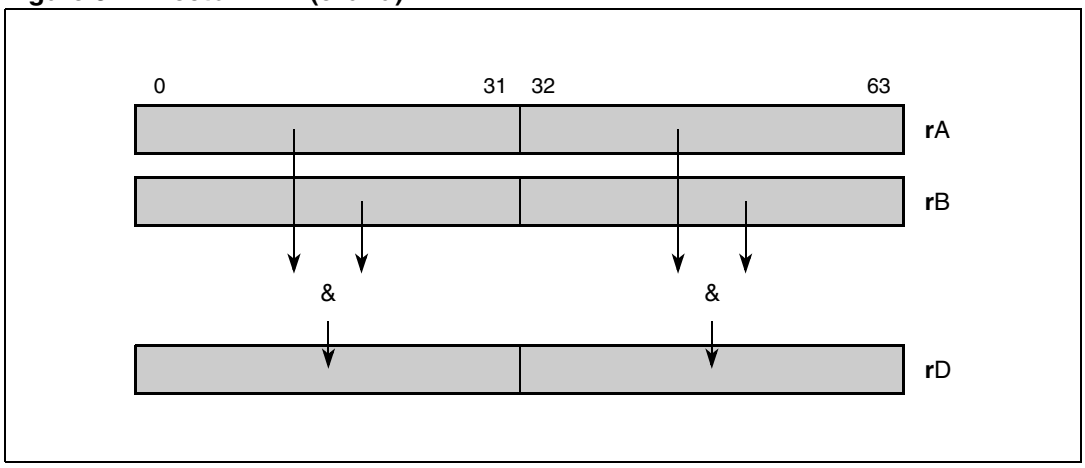
evand **r**D,**r**A,**r**B

0	5	6	10	11	15	16	20	21	31
0 0 0 1 0 0		**r**D		**r**A		**r**B		0 1 0 0 0 0 1 0 0 0 1	

$rD_{0:31} \leftarrow rA_{0:31}$ & $rB_{0:31}$ // Bitwise AND
$rD_{32:63} \leftarrow rA_{32:63}$ & $rB_{32:63}$ // Bitwise AND

The corresponding elements of **r**A and **r**B are ANDed bitwise and the results are placed in the corresponding element of **r**D.

Figure 31. Vector AND (evand)



evandc | SPE APU | User | **evandc**

Vector AND with complement

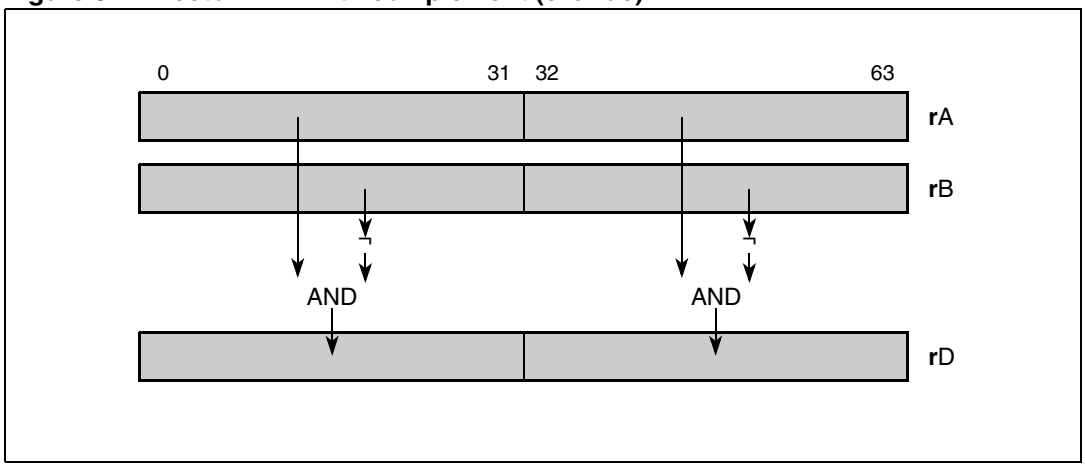
evandc **r**D,**r**A,**r**B

0	5	6	10	11	15	16	20	21	31
0 0 0 1 0 0		**r**D		**r**A		**r**B		0 1 0 0 0 0 1 0 0 1 0	

$rD_{0:31} \leftarrow rA_{0:31}$ & $(\neg rB_{0:31})$ // Bitwise ANDC
$rD_{32:63} \leftarrow rA_{32:63}$ & $(\neg rB_{32:63})$ // Bitwise ANDC

The word elements of **r**A and are ANDed bitwise with the complement of the corresponding elements of **r**B. The results are placed in the corresponding element of **r**D.

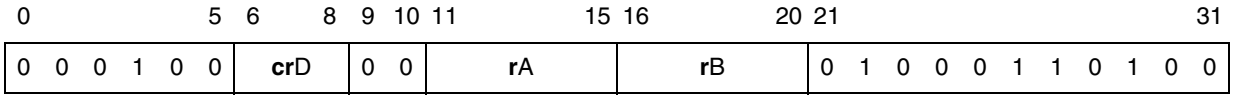
Figure 32. Vector AND with complement (evandc)



evcmpeq | SPE APU | User | **evcmpeq**

Vector compare equal

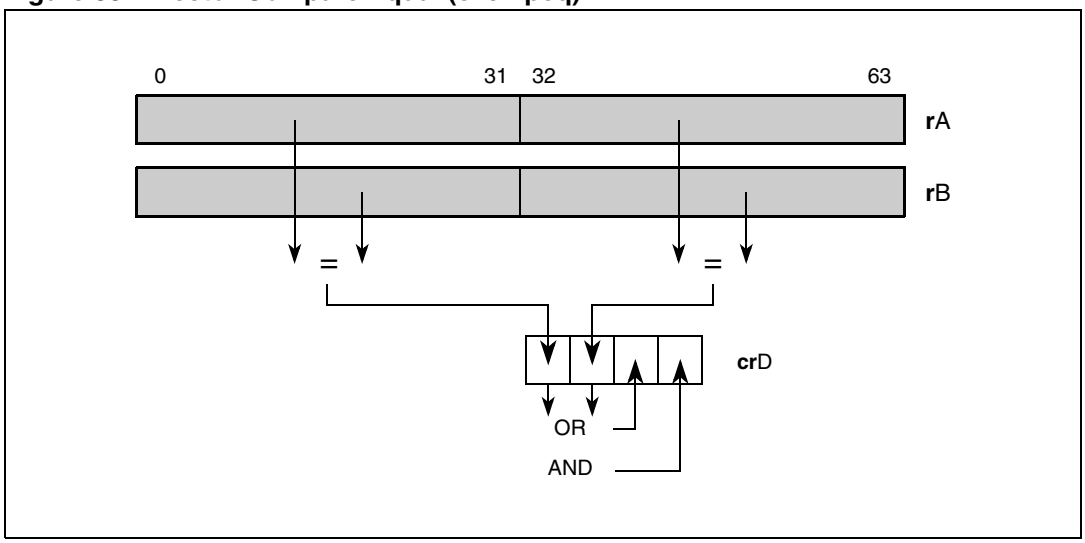
evcmpeq **cr**D,**r**A,**r**B



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
$CR_{4*crD:4*crD+3}$ ← ch || cl || (ch | cl) || (ch & cl)

The most significant bit in **cr**D is set if the high-order element of **r**A is equal to the high-order element of **r**B; it is cleared otherwise. The next bit in **cr**D is set if the low-order element of **r**A is equal to the low-order element of **r**B and cleared otherwise. The last two bits of **cr**D are set to the OR and AND of the result of the compare of the high and low elements.

Figure 33. Vector Compare Equal (evcmpeq)



evcmpgts | SPE APU | User | **evcmpgts**

Vector compare greater than signed

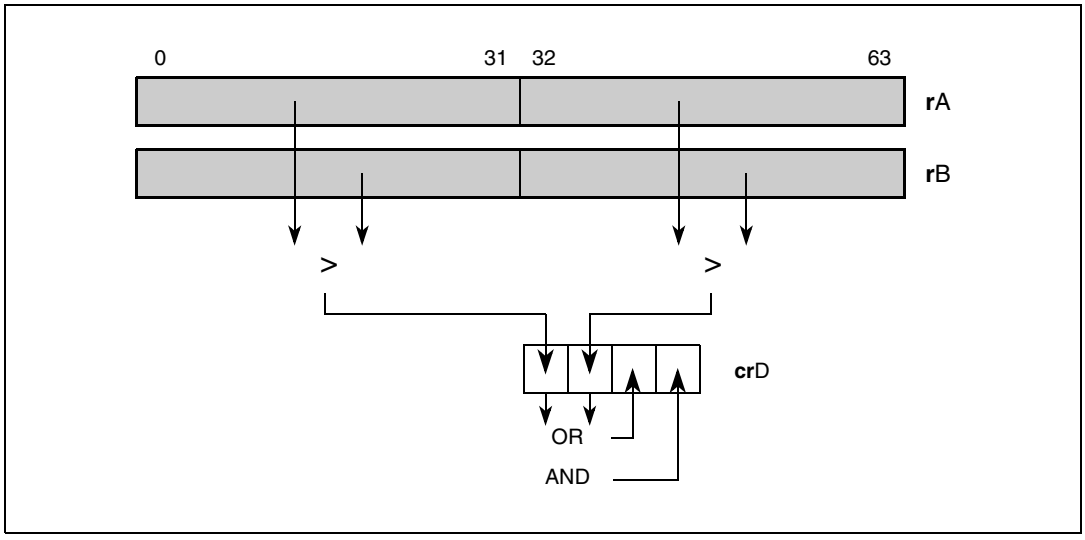
evcmpgts **cr**D,**r**A,**r**B

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	0	1	0	0	**cr**D			0	0	**r**A					**r**B					0	1	0	0	0	1	1	0	0	0	1

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
$CR_{4*crD:4*crD+3}$ ← ch || cl || (ch | cl) || (ch & cl)

The most significant bit in **cr**D is set if the high-order element of **r**A is greater than the high-order element of **r**B; it is cleared otherwise. The next bit in **cr**D is set if the low-order element of **r**A is greater than the low-order element of **r**B and cleared otherwise. The last two bits of **cr**D are set to the OR and AND of the result of the compare of the high and low elements.

Figure 34. Vector compare greater than signed (evcmpgts)



evcmpgtu | SPE APU | User | **evcmpgtu**

Vector compare greater than unsigned

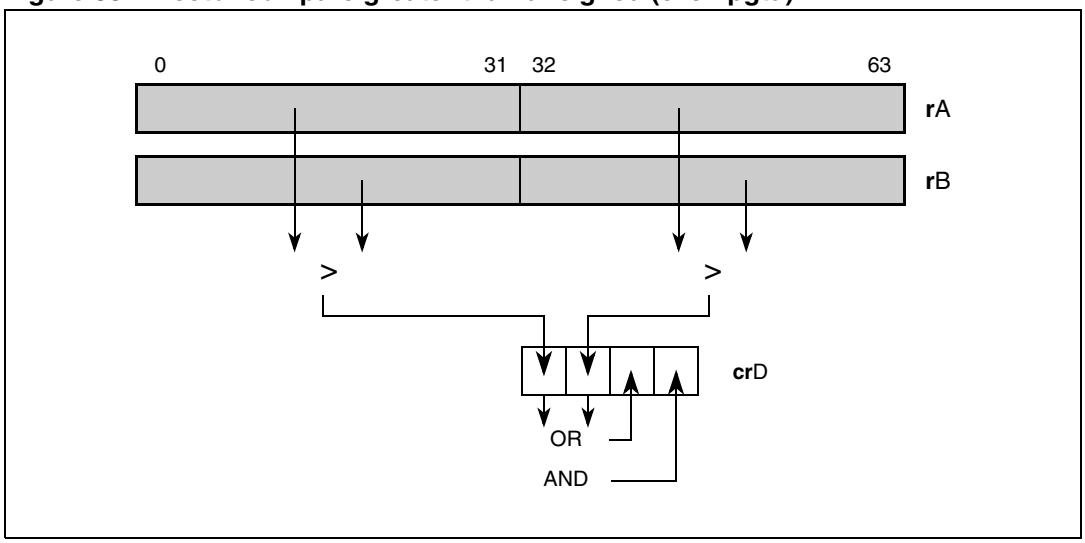
evcmpgtu **cr**D,**r**A,**r**B

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	0	1	0	0	**cr**D			0	0	**r**A					**r**B					0	1	0	0	0	1	1	0	0	0	0

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
$CR_{4*crD:4*crD+3}$ ← ch || cl || (ch | cl) || (ch & cl)

The most significant bit in **cr**D is set if the high-order element of **r**A is greater than the high-order element of **r**B; it is cleared otherwise. The next bit in **cr**D is set if the low-order element of **r**A is greater than the low-order element of **r**B and cleared otherwise. The last two bits of **cr**D are set to the OR and AND of the result of the compare of the high and low elements.

Figure 35. Vector compare greater than unsigned (evcmpgtu)



evcmplts | SPE APU | User | **evcmplts**

Vector compare less than signed

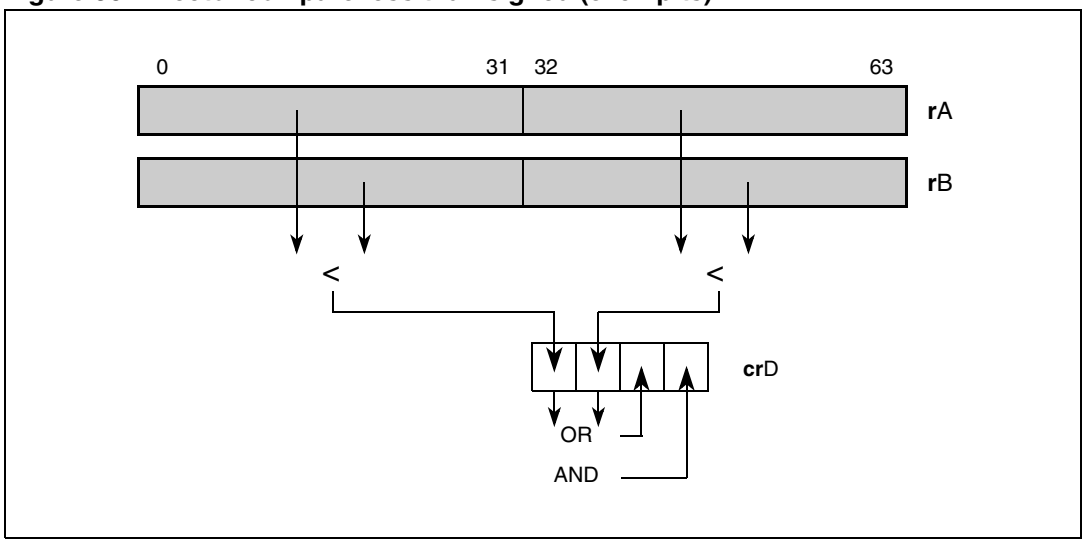
evcmplts **cr**D,**r**A,**r**B

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	0	1	0	0	**cr**D			0	0	**r**A					**r**B					0	1	0	0	0	1	1	0	0	1	1

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
$CR_{4*crD:4*crD+3}$ ← ch || cl || (ch | cl) || (ch & cl)

The most significant bit in **cr**D is set if the high-order element of **r**A is less than the high-order element of **r**B; it is cleared otherwise. The next bit in **cr**D is set if the low-order element of **r**A is less than the low-order element of **r**B and cleared otherwise. The last two bits of **cr**D are set to the OR and AND of the result of the compare of the high and low elements.

Figure 36. Vector compare less than signed (evcmplts)



evcmpltu | SPE APU | User | **evcmpltu**

Vector compare less than unsigned

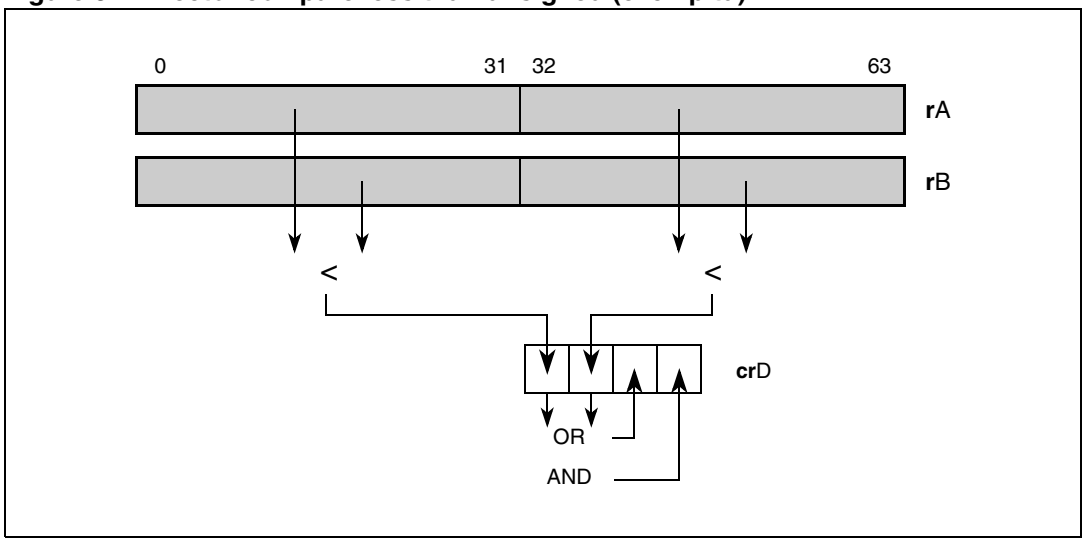
evcmpltu **crD,rA,rB**

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	0	1	0	0	**cr**D			0	0	**r**A					**r**B					0	1	0	0	0	1	1	0	0	1	0

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
$CR_{4*crD:4*crD+3}$ ← ch || cl || (ch | cl) || (ch & cl)

The most significant bit in **cr**D is set if the high-order element of **r**A is less than the high-order element of **r**B; it is cleared otherwise. The next bit in **cr**D is set if the low-order element of **r**A is less than the low-order element of **r**B and cleared otherwise. The last two bits of **cr**D are set to the OR and AND of the result of the compare of the high and low elements.

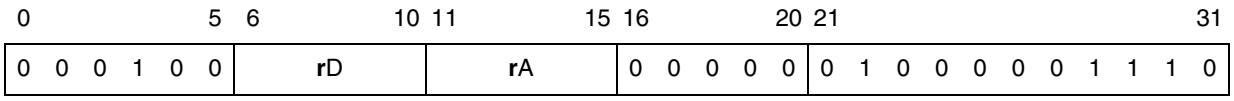
Figure 37. Vector compare less than unsigned (evcmpltu)



evcntlsw | SPE APU | User | **evcntlsw**

Vector count leading signed bits word

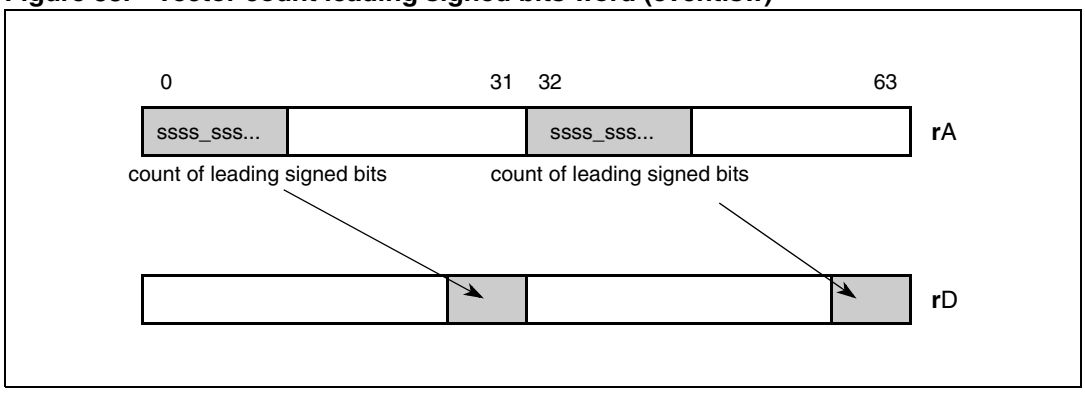
evcntlsw **r**D,**r**A



The leading sign bits in each element of **r**A are counted, and the respective count is placed into each element of **r**D.

evcntlzw is used for unsigned operands; **evcntlsw** is used for signed operands.

Figure 38. Vector count leading signed bits word (evcntlsw)



evcntlzw | SPE APU | User | **evcntlzw**

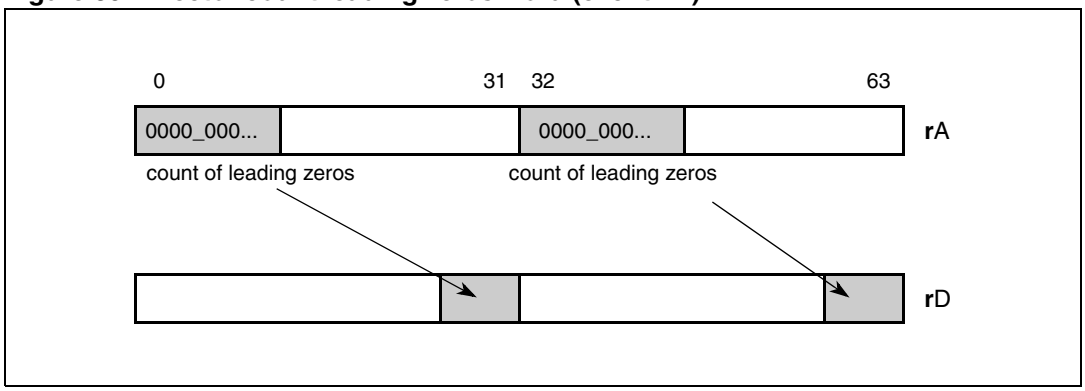
Vector count leading zeros word

evcntlzw **r**D,**r**A

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	0	1	0	0	**r**D					**r**A					0	0	0	0	0	0	1	0	0	0	0	0	1	1	0	1

The leading zero bits in each element of **r**A are counted, and the respective count is placed into each element of **r**D.

Figure 39. Vector count leading zeros word (evcntlzw)



evdivws | SPE APU | User | **evdivws**

Vector divide word signed

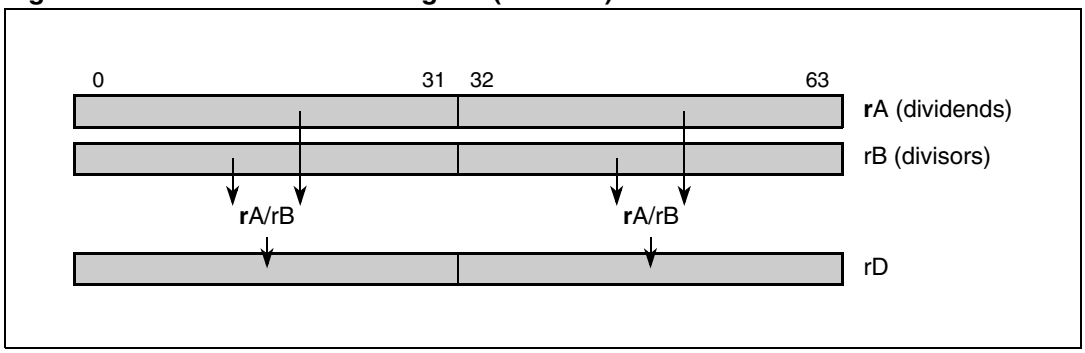
evdivws **r**D,**r**A,**r**B

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	0	1	0	0	**r**D					**r**A					**r**B					1	0	0	1	1	0	0	0	1	1	0

dividendh ← $rA_{0:31}$
dividendl ← $rA_{32:63}$
divisorh ← $rB_{0:31}$
divisorl ← $rB_{32:63}$
$rD_{0:31}$ ← dividendh ÷ divisorh
$rD_{32:63}$ ← dividendl ÷ divisorl
ovh ← 0
ovl ← 0
if ((dividendh < 0) & (divisorh = 0)) then
    $rD_{0:31}$ ← 0x80000000
    ovh ← 1
else if ((dividendh >= 0) & (divisorh = 0)) then
    $rD_{0:31}$ ← 0x7FFFFFFF
    ovh ← 1
else if ((dividendh = 0x80000000) & (divisorh = 0xFFFF_FFFF)) then
    $rD_{0:31}$ ← 0x7FFFFFFF
    ovh ← 1
if ((dividendl < 0) & (divisorl = 0)) then
    $rD_{32:63}$ ← 0x80000000
    ovl ← 1
else if ((dividendl >= 0) & (divisorl = 0)) then
    $rD_{32:63}$ ← 0x7FFFFFFF
    ovl ← 1
else if ((dividendl = 0x80000000) & (divisorl = 0xFFFF_FFFF)) then
    $rD_{32:63}$ ← 0x7FFFFFFF
    ovl ← 1
$SPEFSCR_{OVH}$ ← ovh
$SPEFSCR_{OV}$ ← ovl
$SPEFSCR_{SOVH}$ ← $SPEFSCR_{SOVH}$ | ovh
$SPEFSCR_{SOV}$ ← $SPEFSCR_{SOV}$ | ovl

The two dividends are the two elements of the contents of **r**A. The two divisors are the two elements of the contents of **r**B. The resulting two 32-bit quotients on each element are placed into **r**D. The remainders are not supplied. The operands and quotients are interpreted as signed integers. If overflow, underflow, or divide by zero occurs, the overflow and summary overflow SPEFSCR bits are set. 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.

Figure 40. Vector divide word signed (evdivws)



evdivwu | SPE APU | User | **evdivwu**

Vector divide word unsigned

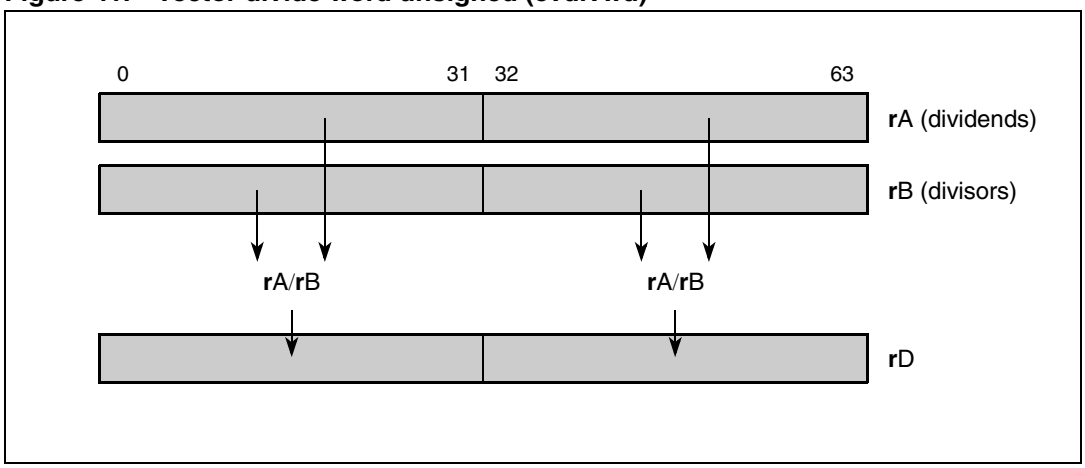
evdivwu **r**D,**r**A,**r**B

0	5	6	10	11	15	16	20	21	31
0 0 0 1 0 0		**r**D		**r**A		**r**B		1 0 0 1 1 0 0 0 1 1 1	

dividendh ← $rA_{0:31}$
dividendl ← $rA_{32:63}$
divisorh ← $rB_{0:31}$
divisorl ← $rB_{32:63}$
$rD_{0:31}$ ← dividendh ÷ divisorh
$rD_{32:63}$ ← dividendl ÷ divisorl
ovh ← 0
ovl ← 0
if (divisorh = 0) then
  $rD_{0:31}$ = 0xFFFFFFFF
  ovh ← 1
if (divisorl = 0) then
  $rD_{32:63}$ ← 0xFFFFFFFF
  ovl ← 1
$SPEFSCR_{OVH}$ ← ovh
$SPEFSCR_{OV}$ ← ovl
$SPEFSCR_{SOVH}$ ← $SPEFSCR_{SOVH}$ | ovh
$SPEFSCR_{SOV}$ ← $SPEFSCR_{SOV}$ | ovl

The two dividends are the two elements of the contents of **r**A. The two divisors are the two elements of the contents of **r**B. 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 **r**D. Remainders are not supplied. Operands and quotients are interpreted as unsigned integers. If a divide by zero occurs, the overflow and summary overflow SPEFSCR bits are set. 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.

Figure 41. Vector divide word unsigned (evdivwu)



eveqv | SPE APU | User | **eveqv**

Vector equivalent

eveqv **r**D,**r**A,**r**B

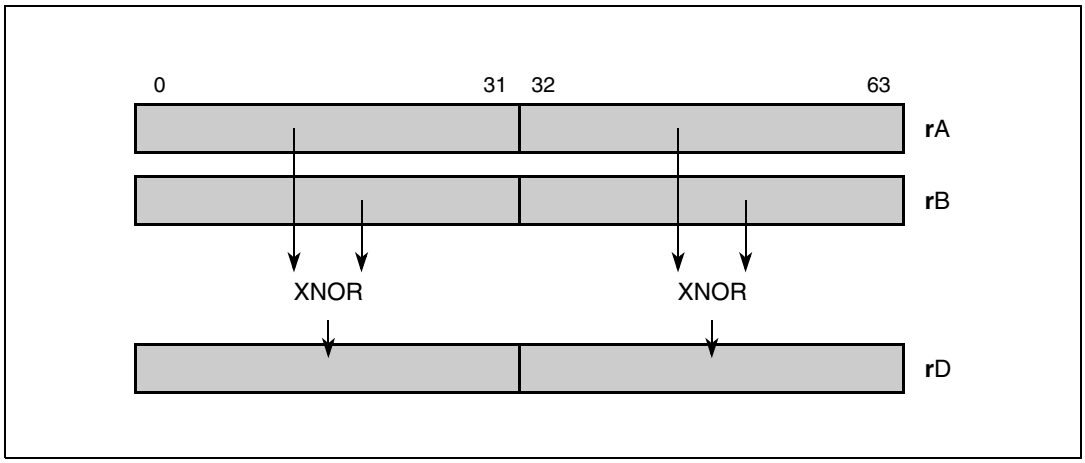
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	0	1	0	0	**r**D					**r**A					**r**B					0	1	0	0	0	0	1	1	0	0	1

$rD_{0:31} \leftarrow rA_{0:31} \equiv rB_{0:31}$ // Bitwise XNOR

$rD_{32:63} \leftarrow rA_{32:63} \equiv rB_{32:63}$ // Bitwise XNOR

The corresponding elements of **r**A & **r**B are XNORed bitwise, & the results are placed in **r**D.

Figure 42. Vector equivalent (eveqv)



evextsb | SPE APU | User | **evextsb**

Vector extend sign byte

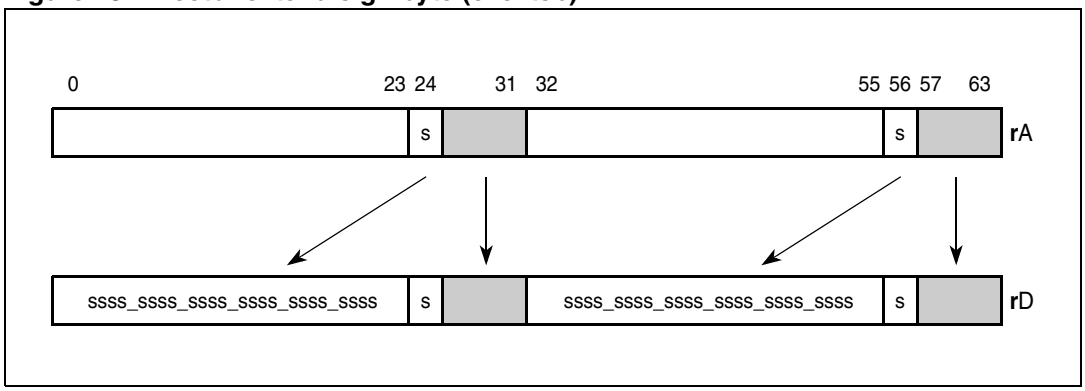
evextsb **r**D,**r**A

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	0	1	0	0	**r**D					**r**A					0	0	0	0	0	0	1	0	0	0	0	0	1	0	1	0

$rD_{0:31} \leftarrow EXTS(rA_{24:31})$
$rD_{32:63} \leftarrow EXTS(rA_{56:63})$

The signs of the byte in each of the elements in **r**A are extended, and the results are placed in **r**D.

Figure 43. Vector extend sign byte (evextsb)



evextsh | SPE APU | User | **evextsh**

Vector extend sign half word

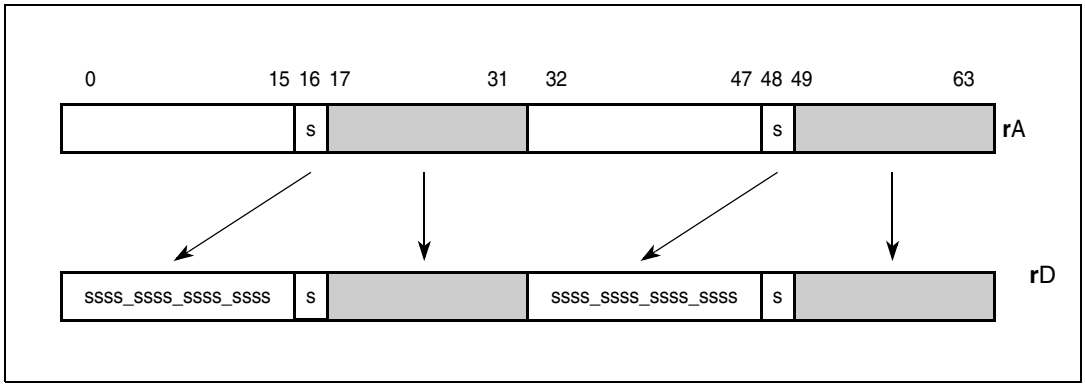
evextsh **r**D,**r**A

0	5	6	10	11	15	16	20	21	31
0 0 0 1 0 0		**r**D		**r**A		0 0 0 0 0		0 1 0 0 0 0 0 1 0 1 1	

$rD_{0:31} \leftarrow EXTS(rA_{16:31})$
$rD_{32:63} \leftarrow EXTS(rA_{48:63})$

The signs of the half words in each of the elements in **r**A are extended, and the results are placed in **r**D.

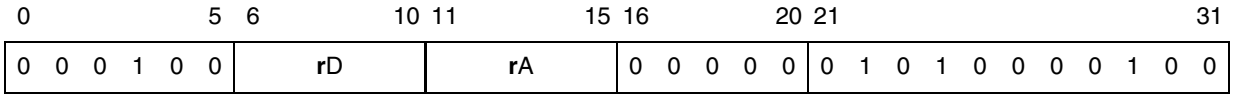
Figure 44. Vector extend sign half word (evextsh)



evfsabs | Vector SPFP APU | User | **evfsabs**

Vector floating-point single-precision absolute value

evfsabs **r**D,**r**A



$rD_{0:31} \leftarrow 0b0 \parallel rA_{1:31}$
$rD_{32:63} \leftarrow 0b0 \parallel rA_{33:63}$

The sign bit of each element in **r**A is set to 0 and the results are placed into **r**D.

Exceptions:

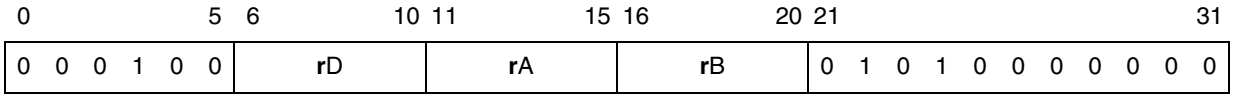
Exception detection for embedded floating-point absolute value operations is implementation dependent. An implementation may choose to not detect exceptions and carry out the computation. If the implementation does not detect exceptions, or if exception detection is disabled, the computation can be carried out in one of two ways, as a sign bit operation ignoring the rest of the contents of the source register, or by examining the input and appropriately saturating the input prior to performing the operation.

If an implementation chooses to handle exceptions, the exception is handled as follows: if the contents of either element of **r**A are Infinity, Denorm, or NaN, SPEFSCR[FINV,FINVH] are set appropriately, and SPEFSCR[FGH,FXH,FG,FX] are cleared appropriately. If floating-point invalid input exceptions are enabled, an interrupt is taken and the destination register is not updated.

evfsadd | Vector SPFP APU | User | **evfsadd**

Vector floating-point single-precision add

evfsadd **r**D,**r**A,**r**B



$rD_{0:31} \leftarrow rA_{0:31} +_{sp} rB_{0:31}$
$rD_{32:63} \leftarrow rA_{32:63} +_{sp} rB_{32:63}$

Each single-precision floating-point element of **r**A is added to the corresponding element of **r**B and the results are stored in **r**D. If an element of **r**A is NaN or infinity, the corresponding result is either *pmax* (a_{sign}==0), or *nmax* (a_{sign}==1). Otherwise, if an element of **r**B is NaN or infinity, the corresponding result is either *pmax* (b_{sign}==0), or *nmax* (b_{sign}==1). Otherwise, if an overflow occurs, *pmax* or *nmax* (as appropriate) is stored in the corresponding element of **r**D. If an underflow occurs, +0 (for rounding modes RN, RZ, RP) or –0 (for rounding mode RM) is stored in the corresponding element of **r**D.

Exceptions:

If the contents of either element of **r**A or **r**B are Infinity, Denorm, or NaN, SPEFSCR[FINV,FINVH] are set appropriately, and SPEFSCR[FGH,FXH,FG,FX] are cleared appropriately. If SPEFSCR[FINVE] is set, an interrupt is taken and the destination register is not updated. Otherwise, if an overflow occurs, SPEFSCR[FOVF,FOVFH] are set appropriately, or if an underflow occurs, SPEFSCR[FUNF,FUNFH] are set appropriately. If either underflow or overflow exceptions are enabled and a corresponding status bit is set, an interrupt is taken. If any of these interrupts are taken, the destination register is not updated.

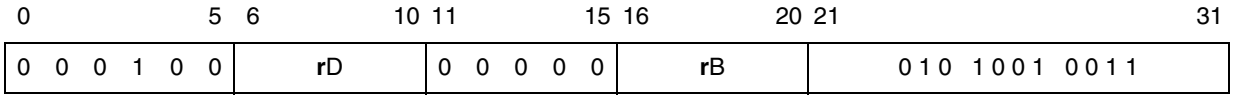
If either result element of this instruction is inexact, or overflows but overflow exceptions are disabled, and no other interrupt is taken, or underflows but underflow exceptions are disabled, and no other interrupt is taken, SPEFSCR[FINXS,FINXSH] is set. If the floating-point inexact exception is enabled, an interrupt is taken using the floating-point round interrupt vector. In this case, the destination register is updated with the truncated result(s). The FG and FX bits are properly updated to allow rounding to be performed in the interrupt handler.

FG and FX (FGH and FXH) are cleared if an overflow or underflow interrupt is taken, or if an invalid operation/input error is signaled for the low (high) element (regardless of FINVE).

evfscfsf | Vector SPFP APU | User | **evfscfsf**

Vector convert floating-point single-precision from signed fraction

evfscfsf **rD,rB**



$rD_{0:31} \leftarrow$ CnvtI32ToFP32Sat($rB_{0:31}$, SIGN, UPPER, F)

$rD_{32:63} \leftarrow$ CnvtI32ToFP32Sat($rB_{32:63}$, SIGN, LOWER, F)

Each signed fractional element of **r**B is converted to a single-precision floating-point value using the current rounding mode and the results are placed into the corresponding elements of **r**D.

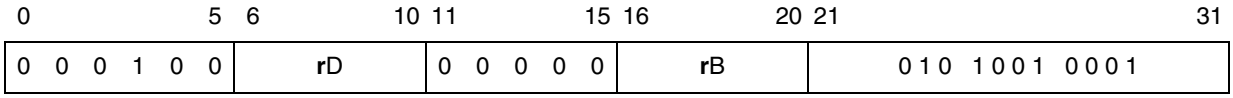
Exceptions:

This instruction can signal an inexact status and set SPEFSCR[FINXS] if the conversions are not exact. If the floating-point inexact exception is enabled, an interrupt is taken using the floating-point round interrupt vector. In this case, the destination register is updated with the truncated result(s). The FGH, FXH, FG and FX bits are properly updated to allow rounding to be performed in the interrupt handler.

evfscfsi | Vector SPFP APU | User | **evfscfsi**

Vector convert floating-point single-precision from signed integer

evfscfsi **r**D,**r**B



$rD_{0:31} \leftarrow$ CnvtSI32ToFP32Sat($rB_{0:31}$, SIGN, UPPER, I)
$rD_{32:63} \leftarrow$ CnvtSI32ToFP32Sat($rB_{32:63}$, SIGN, LOWER, I)

Each signed integer element of **r**B 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 **r**D.

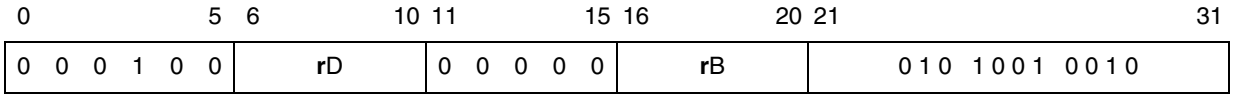
Exceptions:

This instruction can signal an inexact status and set SPEFSCR[FINXS] if the conversions are not exact. If the floating-point inexact exception is enabled, an interrupt is taken using the floating-point round interrupt vector. In this case, the destination register is updated with the truncated result(s). The FGH, FXH, FG and FX bits are properly updated to allow rounding to be performed in the interrupt handler.

evfscfuf | Vector SPFP APU | User | **evfscfuf**

Vector convert floating-point single-precision from unsigned fraction

evfscfuf **r**D,**r**B



$rD_{0:31} \leftarrow$ CnvtI32ToFP32Sat($rB_{0:31}$, UNSIGN, UPPER, F)
$rD_{32:63} \leftarrow$ CnvtI32ToFP32Sat($rB_{32:63}$, UNSIGN, LOWER, F)

Each unsigned fractional element of **r**B is converted to a single-precision floating-point value using the current rounding mode and the results are placed into the corresponding elements of **r**D.

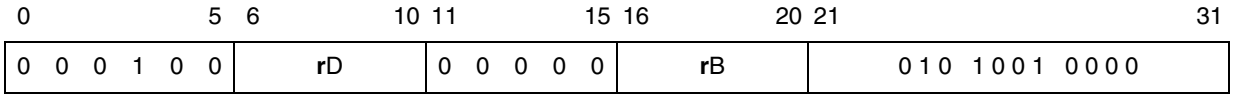
Exceptions:

This instruction can signal an inexact status and set SPEFSCR[FINXS] if the conversions are not exact. If the floating-point inexact exception is enabled, an interrupt is taken using the floating-point round interrupt vector. In this case, the destination register is updated with the truncated result(s). The FGH, FXH, FG and FX bits are properly updated to allow rounding to be performed in the interrupt handler.

evfscfui | Vector SPFP APU | User | **evfscfui**

Vector convert floating-point single-precision from unsigned integer

evfscfui rD,rB



$rD_{0:31} \leftarrow$ CnvtI32ToFP32Sat(rB_{031}, UNSIGN, UPPER, I)
$rD_{32:63} \leftarrow$ CnvtI32ToFP32Sat($rB_{32:63}$, UNSIGN, LOWER, I)

Each unsigned integer element of **r**B 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 **r**D.

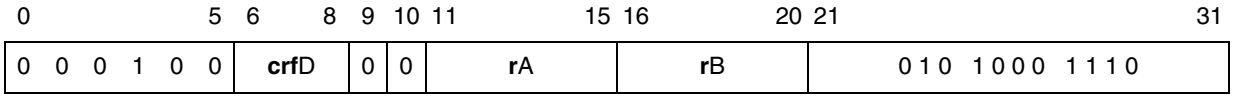
Exceptions:

This instruction can signal an inexact status and set SPEFSCR[FINXS] if the conversions are not exact. If the floating-point inexact exception is enabled, an interrupt is taken using the floating-point round interrupt vector. In this case, the destination register is updated with the truncated result(s). The FGH, FXH, FG and FX bits are properly updated to allow rounding to be performed in the interrupt handler.

evfscmpeq | Vector SPFP APU | User | **evfscmpeq**

Vector floating-point single-precision compare equal

evfscmpeq **crf**D,**r**A,**r**B



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
$CR_{4*crD:4*crD+3}$ ← ch || cl || (ch | cl) || (ch & cl)

Each element of **r**A is compared against the corresponding element of **r**B. If **r**A equals **r**B, the **crf**D bit is set, otherwise it is cleared. Comparison ignores the sign of 0 (+0 = –0).

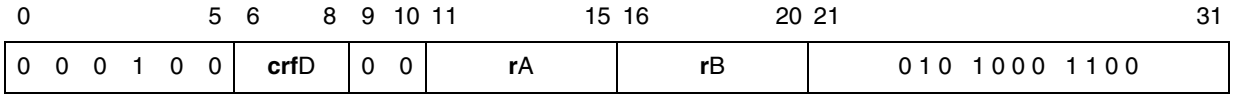
Exceptions:

If the contents of either element of **r**A or **r**B are Infinity, Denorm, or NaN, SPEFSCR[FINV,FINVH] are set appropriately, and SPEFSCR[FGH,FXH,FG,FX] are cleared appropriately. If floating-point invalid input exceptions are enabled, an interrupt is taken, and the condition register is not updated. Otherwise, the comparison proceeds after treating NaNs, Infinities, and Denorms as normalized numbers, using their values of '*e*' and '*f*' directly.

evfscmpgt | Vector SPFP APU | User | **evfscmpgt**

Vector floating-point single-precision compare greater than

evfscmpgt **crf**D,**r**A,**r**B



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
$CR_{4*crD:4*crD+3}$ ← ch || cl || (ch | cl) || (ch & cl)

Each element of **r**A is compared against the corresponding element of **r**B. If **r**A is greater than **r**B, the bit in the **crf**D is set, otherwise it is cleared. Comparison ignores the sign of 0 (+0 = –0).

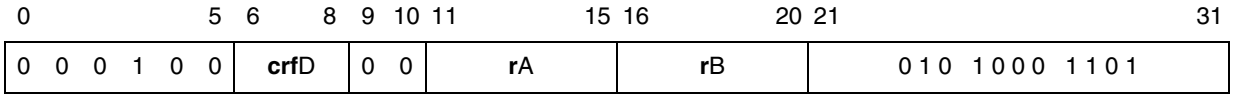
Exceptions:

If the contents of either element of **r**A or **r**B are Infinity, Denorm, or NaN, SPEFSCR[FINV,FINVH] are set appropriately, and SPEFSCR[FGH,FXH,FG,FX] are cleared appropriately. If floating-point invalid input exceptions are enabled then an interrupt is taken, and the condition register is not updated. Otherwise, the comparison proceeds after treating NaNs, Infinities, and Denorms as normalized numbers, using their values of '*e*' and '*f*' directly.

evfscmplt | Vector SPFP APU | User | **evfscmplt**

Vector floating-point single-precision compare less than

evfscmplt **crf**D,**r**A,**r**B



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
$CR_{4*crD:4*crD+3}$ ← ch || cl || (ch | cl) || (ch & cl)

Each element of **r**A is compared against the corresponding element of **r**B. If **r**A is less than **r**B, the bit in the **crf**D is set, otherwise it is cleared. Comparison ignores the sign of 0 (+0 = –0).

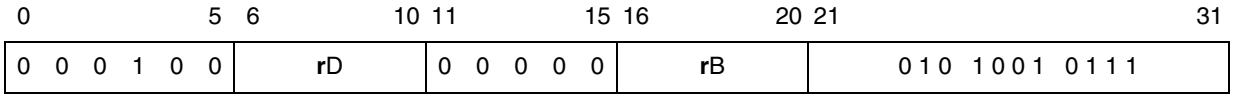
Exceptions:

If the contents of either element of **r**A or **r**B are Infinity, Denorm, or NaN, SPEFSCR[FINV,FINVH] are set appropriately, and SPEFSCR[FGH,FXH,FG,FX] are cleared appropriately. If floating-point invalid input exceptions are enabled then an interrupt is taken, and the condition register is not updated. Otherwise, the comparison proceeds after treating NaNs, Infinities, and Denorms as normalized numbers, using their values of '*e*' and '*f*' directly.

evfsctsf | Vector SPFP APU | User | **evfsctsf**

Vector convert floating-point single-precision to signed fraction

evfsctsf **r**D,**r**B



$rD_{0:31} \leftarrow$ CnvtFP32ToISat($rB_{0:31}$, SIGN, UPPER, ROUND, F)
$rD_{32:63} \leftarrow$ CnvtFP32ToISat($rB_{32:63}$, SIGN, LOWER, ROUND, F)

Each single-precision floating-point element in **r**B 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.

Exceptions:

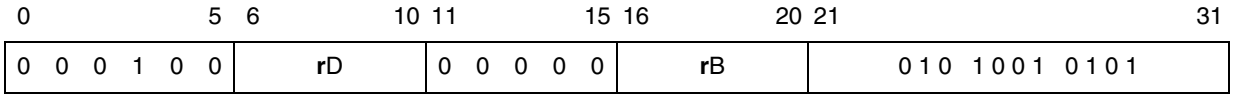
If either element of **r**B is Infinity, Denorm, or NaN, or if an overflow occurs, SPEFSCR[FINV,FINVH] are set appropriately and SPEFSCR[FGH,FXH,FG,FX] are cleared appropriately. If SPEFSCR[FINVE] is set, an interrupt is taken, the destination register is not updated, and no other status bits are set.

If either result element of this instruction is inexact and no other interrupt is taken, SPEFSCR[FINXS] is set. If the floating-point inexact exception is enabled, an interrupt is taken using the floating-point round interrupt vector. In this case, the destination register is updated with the truncated result. The FGH, FXH, FG and FX bits are properly updated to allow rounding to be performed in the interrupt handler.

evfsctsi | Vector SPFP APU | User | **evfsctsi**

Vector convert floating-point single-precision to signed integer

evfsctsi **r**D,**r**B



$rD_{0:31} \leftarrow$ CnvtFP32ToISat($rB_{0:31}$, SIGN, UPPER, ROUND, I)
$rD_{32:63} \leftarrow$ CnvtFP32ToISat($rB_{32:63}$, SIGN, LOWER, ROUND, I)

Each single-precision floating-point element in **r**B 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.

Exceptions:

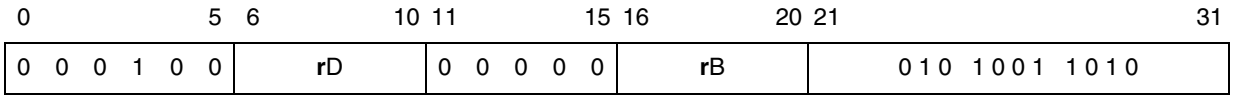
If the contents of either element of **r**B are Infinity, Denorm, or NaN, or if an overflow occurs on conversion, SPEFSCR[FINV,FINVH] are set appropriately, and SPEFSCR[FGH,FXH,FG,FX] are cleared appropriately. If SPEFSCR[FINVE] is set, an interrupt is taken, the destination register is not updated, and no other status bits are set.

If either result element of this instruction is inexact and no other interrupt is taken, SPEFSCR[FINXS] is set. If the floating-point inexact exception is enabled, an interrupt is taken using the floating-point round interrupt vector. In this case, the destination register is updated with the truncated result. The FGH, FXH, FG and FX bits are properly updated to allow rounding to be performed in the interrupt handler.

evfsctsiz | Vector SPFP APU | User | **evfsctsiz**

Vector convert floating-point single-precision to signed integer with round toward zero

evfsctsiz **r**D,**r**B



$rD_{0:31} \leftarrow$ CnvtFP32ToISat($rB_{0:31}$, SIGN, UPPER, TRUNC, I)
$rD_{32:63} \leftarrow$ CnvtFP32ToISat($rB_{32:63}$, SIGN, LOWER, TRUNC, I)

Each single-precision floating-point element in **r**B 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.

Exceptions:

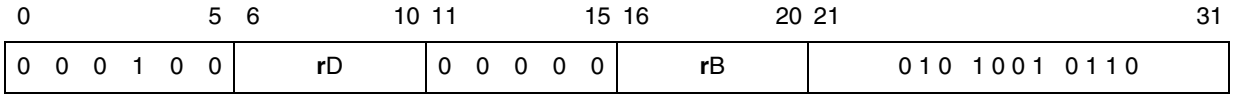
If either element of **r**B is Infinity, Denorm, or NaN, or if an overflow occurs, SPEFSCR[FINV,FINVH] are set appropriately, and SPEFSCR[FGH,FXH,FG,FX] are cleared appropriately. If SPEFSCR[FINVE] is set, an interrupt is taken, the destination register is not updated, and no other status bits are set.

If either result element of this instruction is inexact and no other interrupt is taken, SPEFSCR[FINXS] is set. If the floating-point inexact exception is enabled, an interrupt is taken using the floating-point round interrupt vector. In this case, the destination register is updated with the truncated result. The FGH, FXH, FG and FX bits are properly updated to allow rounding to be performed in the interrupt handler.

evfsctuf | Vector SPFP APU | User | **evfsctuf**

Vector convert floating-point single-precision to unsigned fraction

evfsctuf **r**D,**r**B



$rD_{0:31} \leftarrow$ CnvtFP32ToISat($rB_{0:31}$, UNSIGN, UPPER, ROUND, F)
$rD_{32:63} \leftarrow$ CnvtFP32ToISat($rB_{32:63}$, UNSIGN, LOWER, ROUND, F)

Each single-precision floating-point element in **r**B 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.

Exceptions:

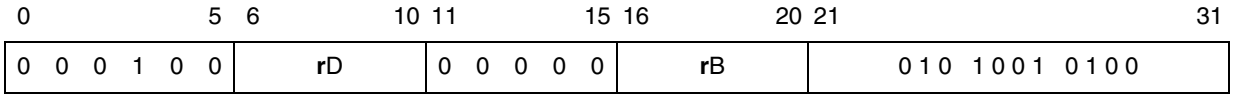
If either element of **r**B is Infinity, Denorm, or NaN, or if an overflow occurs, SPEFSCR[FINV,FINVH] are set appropriately, and SPEFSCR[FGH,FXH,FG,FX] are cleared appropriately. If SPEFSCR[FINVE] is set, an interrupt is taken, the destination register is not updated, and no other status bits are set.

If either result element of this instruction is inexact and no other interrupt is taken, SPEFSCR[FINXS] is set. If the floating-point inexact exception is enabled, an interrupt is taken using the floating-point round interrupt vector. In this case, the destination register is updated with the truncated result. The FGH, FXH, FG and FX bits are properly updated to allow rounding to be performed in the interrupt handler.

evfsctui | Vector SPFP APU | User | **evfsctui**

Vector convert floating-point single-precision to unsigned integer

evfsctui **r**D,**r**B



$rD_{0:31} \leftarrow$ CnvtFP32ToISat($rB_{0:31}$, UNSIGN, UPPER, ROUND, I)
$rD_{32:63} \leftarrow$ CnvtFP32ToISat($rB_{32:63}$, UNSIGN, LOWER, ROUND, I)

Each single-precision floating-point element in **r**B 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.

Exceptions:

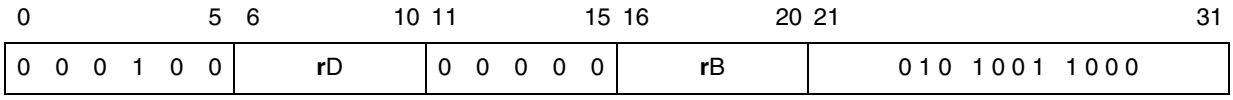
If either element of **r**B is Infinity, Denorm, or NaN, or if an overflow occurs, SPEFSCR[FINV,FINVH] are set appropriately, and SPEFSCR[FGH,FXH,FG,FX] are cleared appropriately. If SPEFSCR[FINVE] is set, an interrupt is taken, the destination register is not updated, and no other status bits are set.

If either result element of this instruction is inexact and no other interrupt is taken, SPEFSCR[FINXS] is set. If the floating-point inexact exception is enabled, an interrupt is taken using the floating-point round interrupt vector. In this case, the destination register is updated with the truncated result. The FGH, FXH, FG and FX bits are properly updated to allow rounding to be performed in the interrupt handler.

evfsctuiz | Vector SPFP APU | User | **evfsctuiz**

Vector convert floating-point single-precision to unsigned integer with round toward zero

evfsctuiz **r**D,**r**B



$rD_{0:31} \leftarrow$ CnvtFP32ToISat($rB_{0:31}$, UNSIGN, UPPER, TRUNC, I)
$rD_{32:63} \leftarrow$ CnvtFP32ToISat($rB_{32:63}$, UNSIGN, LOWER, TRUNC, I)

Each single-precision floating-point element in **r**B 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.

Exceptions:

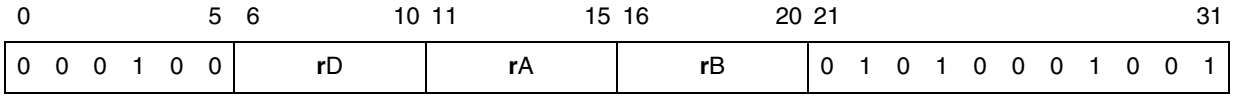
If either element of **r**B is Infinity, Denorm, or NaN, or if an overflow occurs, SPEFSCR[FINV,FINVH] are set appropriately, and SPEFSCR[FGH,FXH,FG,FX] are cleared appropriately. If SPEFSCR[FINVE] is set, an interrupt is taken, the destination register is not updated, and no other status bits are set.

If either result element of this instruction is inexact and no other interrupt is taken, SPEFSCR[FINXS] is set. If the floating-point inexact exception is enabled, an interrupt is taken using the floating-point round interrupt vector. In this case, the destination register is updated with the truncated result. The FGH, FXH, FG and FX bits are properly updated to allow rounding to be performed in the interrupt handler.

evfsdiv | Vector SPFP APU | User | **evfsdiv**

Vector floating-point single-precision divide

evfsdiv **r**D,**r**A,**r**B



$rD_{0:31} \leftarrow rA_{0:31} \div_{sp} rB_{0:31}$
$rD_{32:63} \leftarrow rA_{32:63} \div_{sp} rB_{32:63}$

Each single-precision floating-point element of **r**A is divided by the corresponding element of **r**B and the result is stored in **r**D. If an element of **r**B is a NaN or infinity, the corresponding result is a properly signed zero. Otherwise, if an element of **r**B is a zero (or a denormalized number optionally transformed to zero by the implementation), or if an element of **r**A is either NaN or infinity, the corresponding result is either *pmax* ($a_{sign}==b_{sign}$), or *nmax* ($a_{sign}!=b_{sign}$). Otherwise, if an overflow occurs, *pmax* or *nmax* (as appropriate) is stored in the corresponding element of **r**D. If an underflow occurs, +0 or –0 (as appropriate) is stored in the corresponding element of **r**D.

Exceptions:

If the contents of **r**A or **r**B are Infinity, Denorm, or NaN, or if both **r**A and **r**B are ±0, SPEFSCR[FINV,FINVH] are set appropriately, and SPEFSCR[FGH,FXH,FG,FX] are cleared appropriately. If SPEFSCR[FINVE] is set, an interrupt is taken and the destination register is not updated. Otherwise, if the content of **r**B is ±0 and the content of **r**A is a finite normalized non-zero number, SPEFSCR[FDBZ,FDBZH] are set appropriately. If floating-point divide-by-zero exceptions are enabled, an interrupt is then taken. Otherwise, if an overflow occurs, SPEFSCR[FOVF,FOVFH] are set appropriately, or if an underflow occurs, SPEFSCR[FUNF,FUNFH] are set appropriately. If either underflow or overflow exceptions are enabled and a corresponding bit is set, an interrupt is taken. If any of these interrupts are taken, the destination register is not updated.

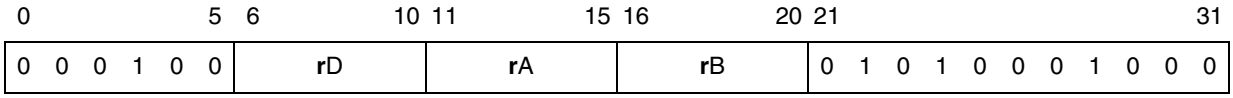
If either result element of this instruction is inexact, or overflows but overflow exceptions are disabled, and no other interrupt is taken, or underflows but underflow exceptions are disabled, and no other interrupt is taken, SPEFSCR[FINXS] is set. If the floating-point inexact exception is enabled, an interrupt is taken using the floating-point round interrupt vector. In this case, the destination register is updated with the truncated result(s). The FG and FX bits are properly updated to allow rounding to be performed in the interrupt handler.

FG and FX (FGH and FXH) are cleared if an overflow or underflow interrupt is taken, or if an invalid operation/input error is signaled for the low (high) element (regardless of FINVE).

evfsmul | Vector SPFP APU | User | **evfsmul**

Vector floating-point single-precision multiply

evfsmul **r**D,**r**A,**r**B



$rD_{0:31} \leftarrow rA_{0:31} \times_{sp} rB_{0:31}$
$rD_{32:63} \leftarrow rA_{32:63} \times_{sp} rB_{32:63}$

Each single-precision floating-point element of **r**A is multiplied with the corresponding element of **r**B and the result is stored in **r**D. If an element of **r**A or **r**B are either zero (or a denormalized number optionally transformed to zero by the implementation), the corresponding result is a properly signed zero. Otherwise, if an element of **r**A or **r**B are either NaN or infinity, the corresponding result is either *pmax* ($a_{sign}==b_{sign}$), or *nmax* ($a_{sign}!=b_{sign}$). Otherwise, if an overflow occurs, *pmax* or *nmax* (as appropriate) is stored in the corresponding element of **r**D. If an underflow occurs, +0 or –0 (as appropriate) is stored in the corresponding element of **r**D.

Exceptions:

If the contents of either element of **r**A or **r**B are Infinity, Denorm, or NaN, SPEFSCR[FINV,FINVH] are set appropriately, and SPEFSCR[FGH,FXH,FG,FX] are cleared appropriately. If SPEFSCR[FINVE] is set, an interrupt is taken and the destination register is not updated. Otherwise, if an overflow occurs, SPEFSCR[FOVF,FOVFH] are set appropriately, or if an underflow occurs, SPEFSCR[FUNF,FUNFH] are set appropriately. If either underflow or overflow exceptions are enabled and a corresponding status bit is set, an interrupt is taken. If any of these interrupts are taken, the destination register is not updated.

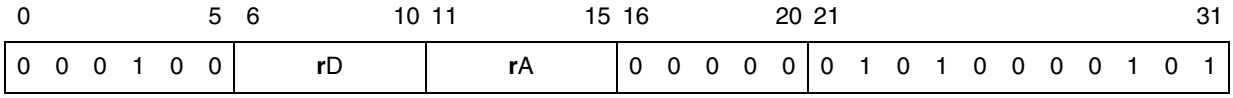
If either result element of this instruction is inexact, or overflows but overflow exceptions are disabled, and no other interrupt is taken, or underflows but underflow exceptions are disabled, and no other interrupt is taken, SPEFSCR[FINXS] is set. If the floating-point inexact exception is enabled, an interrupt is taken using the floating-point round interrupt vector. In this case, the destination register is updated with the truncated result(s). The FG and FX bits are properly updated to allow rounding to be performed in the interrupt handler.

FG and FX (FGH and FXH) are cleared if an overflow or underflow exception is taken, or if an invalid operation/input error is signaled for the low (high) element (regardless of FINVE).

evfsnabs | Vector SPFP APU | User | **evfsnabs**

Vector floating-point single-precision negative absolute value

evfsnabs **r**D,**r**A



$rD_{0:31} \leftarrow 0b1 \parallel rA_{1:31}$
$rD_{32:63} \leftarrow 0b1 \parallel rA_{33:63}$

The sign bit of each element in **r**A is set to 1 and the results are placed into **r**D.

Exceptions:

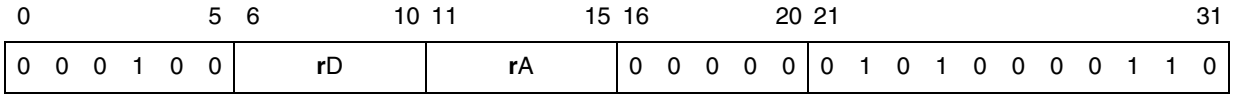
Exception detection for embedded floating-point absolute value operations is implementation dependent. An implementation may choose to not detect exceptions and carry out the sign bit operation. If the implementation does not detect exceptions, or if exception detection is disabled, the computation can be carried out in one of two ways, as a sign bit operation ignoring the rest of the contents of the source register, or by examining the input and appropriately saturating the input prior to performing the operation.

If an implementation chooses to handle exceptions, the exception is handled as follows: if the contents of either element of **r**A are Infinity, Denorm, or NaN, SPEFSCR[FINV,FINVH] are set appropriately, and SPEFSCR[FGH,FXH,FG,FX] are cleared appropriately. If floating-point invalid input exceptions are enabled then an interrupt is taken, and the destination register is not updated.

evfsneg | Vector SPFP APU | User | **evfsneg**

Vector floating-point single-precision negate

evfsneg **r**D,**r**A



$rD_{0:31} \leftarrow \neg rA_0 \,||\, rA_{1:31}$
$rD_{32:63} \leftarrow \neg rA_{32} \,||\, rA_{33:63}$

The sign bit of each element in **r**A is complemented and the results are placed into **r**D.

Exceptions:

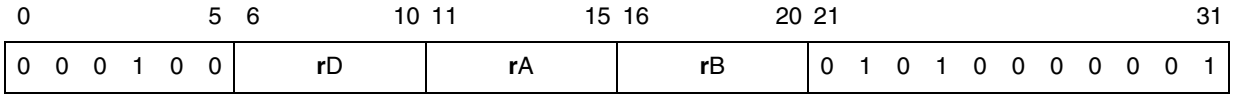
Exception detection for embedded floating-point absolute value operations is implementation dependent. An implementation may choose to not detect exceptions and carry out the sign bit operation. If the implementation does not detect exceptions, or if exception detection is disabled, the computation can be carried out in one of two ways, as a sign bit operation ignoring the rest of the contents of the source register, or by examining the input and appropriately saturating the input prior to performing the operation.

If an implementation chooses to handle exceptions, the exception is handled as follows: if the contents of either element of **r**A are Infinity, Denorm, or NaN, SPEFSCR[FINV,FINVH] are set appropriately, and SPEFSCR[FGH,FXH,FG,FX] are cleared appropriately. If floating-point invalid input exceptions are enabled then an interrupt is taken, and the destination register is not updated.

evfssub | Vector SPFP APU | User | **evfssub**

Vector floating-point single-precision subtract

evfssub **r**D,**r**A,**r**B



$rD_{0:31} \leftarrow rA_{0:31}$ -$_{sp}$ $rB_{0:31}$
$rD_{32:63} \leftarrow rA_{32:63}$ -$_{sp}$ $rB_{32:63}$

Each single-precision floating-point element of **r**B is subtracted from the corresponding element of **r**A and the results are stored in **r**D. If an element of **r**A is NaN or infinity, the corresponding result is either *pmax* (a_{sign}==0), or *nmax* (a_{sign}==1). Otherwise, if an element of **r**B is NaN or infinity, the corresponding result is either *nmax* (b_{sign}==0), or *pmax* (b_{sign}==1). Otherwise, if an overflow occurs, *pmax* or *nmax* (as appropriate) is stored in the corresponding element of **r**D. If an underflow occurs, +0 (for rounding modes RN, RZ, RP) or –0 (for rounding mode RM) is stored in the corresponding element of **r**D.

Exceptions:

If the contents of either element of **r**A or **r**B are Infinity, Denorm, or NaN, SPEFSCR[FINV,FINVH] are set appropriately, and SPEFSCR[FGH,FXH,FG,FX] are cleared appropriately. If SPEFSCR[FINVE] is set, an interrupt is taken and the destination register is not updated. Otherwise, if an overflow occurs, SPEFSCR[FOVF,FOVFH] are set appropriately, or if an underflow occurs, SPEFSCR[FUNF,FUNFH] are set appropriately. If either underflow or overflow exceptions are enabled and a corresponding status bit is set, an interrupt is taken. If any of these interrupts are taken, the destination register is not updated.

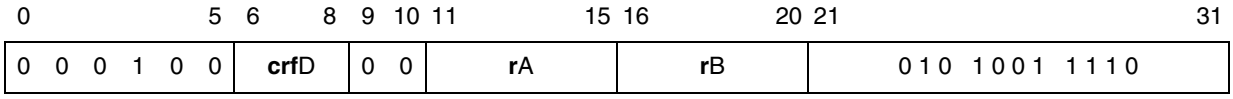
If either result element of this instruction is inexact, or overflows but overflow exceptions are disabled, and no other interrupt is taken, or underflows but underflow exceptions are disabled, and no other interrupt is taken, SPEFSCR[FINXS] is set. If the floating-point inexact exception is enabled, an interrupt is taken using the floating-point round interrupt vector. In this case, the destination register is updated with the truncated result(s). The FG and FX bits are properly updated to allow rounding to be performed in the interrupt handler.

FG and FX (FGH and FXH) are cleared if an overflow or underflow interrupt is taken, or if an invalid operation/input error is signaled for the low (high) element (regardless of FINVE).

evfststeq | Vector SPFP APU | User | **evfststeq**

Vector floating-point single-precision test equal

evfststeq **crf**D,**r**A,**r**B



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
$CR_{4*crD:4*crD+3}$ ← ch || cl || (ch | cl) || (ch & cl)

Each element of **r**A is compared against the corresponding element of **r**B. If **r**A equals **r**B, the bit in **crf**D is set, otherwise it is cleared. 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.

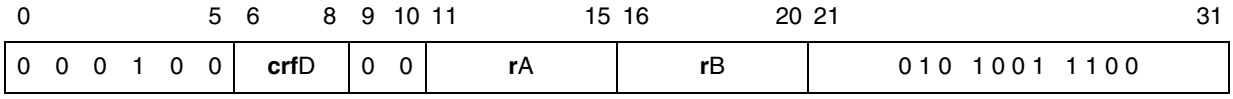
No exceptions are taken during the execution of **evfststeq**. If strict IEEE 754 compliance is required, the program should use **evfscmpeq**.

Implementation note: In an implementation, the execution of **evfststeq** is likely to be faster than the execution of **evfscmpeq**.

evfststgt | Vector SPFP APU | User | **evfststgt**

Vector floating-point single-precision test greater than

evfststgt **crf**D,**r**A,**r**B



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
$CR_{4*crD:4*crD+3}$ ← ch || cl || (ch | cl) || (ch & cl)

Each element of **r**A is compared against the corresponding element of **r**B. If **r**A is greater than **r**B, the bit in **crf**D is set, otherwise it is cleared. 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.

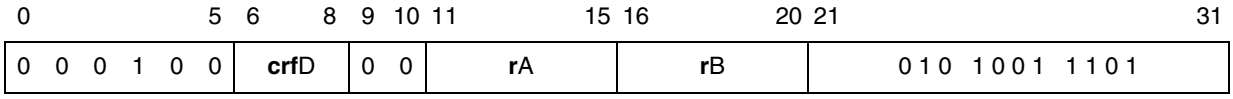
No exceptions are taken during the execution of **evfststgt**. If strict IEEE 754 compliance is required, the program should use **evfscmpgt**.

Implementation note: In an implementation, the execution of **evfststgt** is likely to be faster than the execution of **evfscmpgt**.

evfststlt | Vector SPFP APU | User | **evfststlt**

Vector floating-point single-precision test less than

evfststlt **crf**D,**r**A,**r**B



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
$CR_{4*crD:4*crD+3}$ ← ch || cl || (ch | cl) || (ch & cl)

Each element of **r**A is compared with the corresponding element of **r**B. If **r**A is less than **r**B, the bit in the **crf**D is set, otherwise it is cleared. 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**. If strict IEEE 754 compliance is required, the program should use **evfscmplt**.

Implementation note: In an implementation, the execution of **evfststlt** is likely to be faster than the execution of **evfscmplt**.

evldd | SPE, Vector SPFP, Scalar DPFP APUs | User | **evldd**

Vector load double word into double word

evldd **r**D,**d**(**r**A)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	0	1	0	0	**r**D					**r**A					UIMM(1)					0	1	1	0	0	0	0	0	0	0	0	1

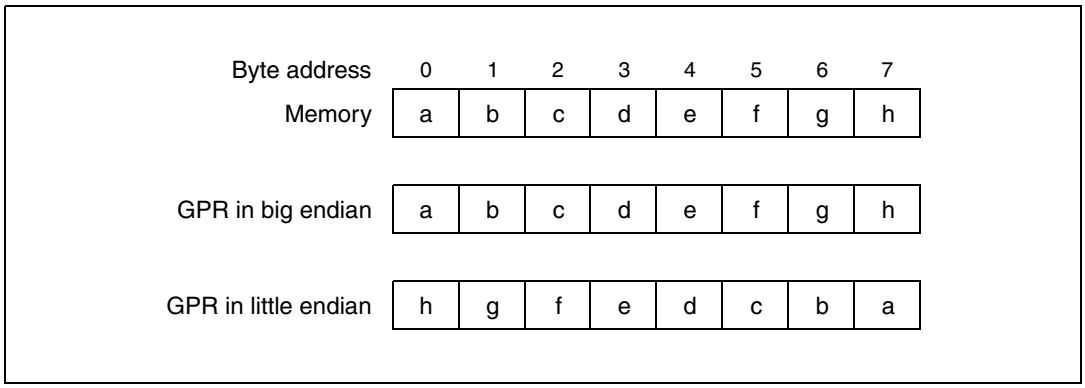
1. **d** = UIMM * 8

```
if (rA = 0) then b ← 0
else b ← (rA)
EA ← b + EXTZ(UIMM*8)
rD ← MEM(EA, 8)
```

The double word addressed by EA is loaded from memory and placed in **r**D.

Figure 45 shows how bytes are loaded into **r**D as determined by the endian mode.

Figure 45. evldd results in big- and little-endian modes



Implementation note: If the EA is not double-word aligned, an alignment exception occurs.

evlddx | SPE, Vector SPFP, Scalar DPFP APUs | User | **evlddx**

Vector load double word into double word indexed

evlddx **r**D,**r**A,**r**B

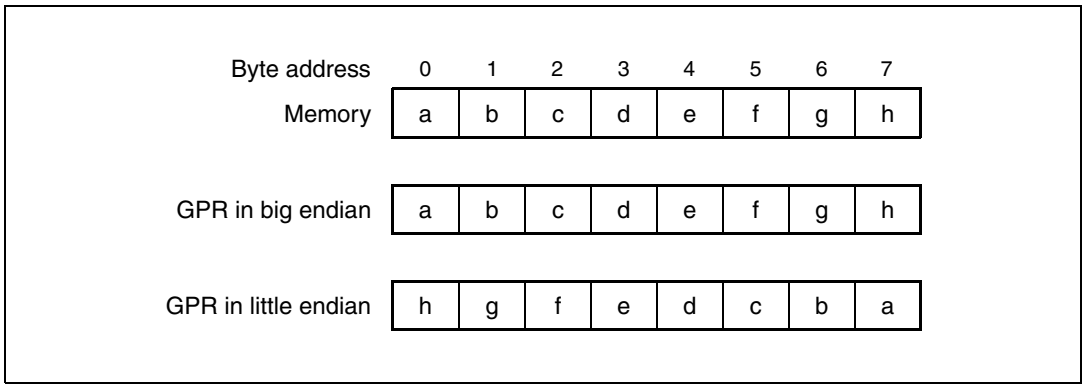
0	5	6	10	11	15	16	20	21	31
0 0 0 1 0 0		**r**D		**r**A		**r**B		0 1 1 0 0 0 0 0 0 0 0	

if (rA = 0) then b ← 0
else b ← (rA)
EA ← b + (rB)
rD ← MEM(EA, 8)

The double word addressed by EA is loaded from memory and placed in **r**D.

Figure 46 shows how bytes are loaded into **r**D as determined by the endian mode.

Figure 46. evlddx results in big- and little-endian modes



Implementation note: If the EA is not double-word aligned, an alignment exception occurs.

evldh | SPE APU | User | **evldh**

Vector load double into four half words

evldh **r**D,**d**(**r**A)

0	5	6	10	11	15	16	20	21	31
0 0 0 1 0 0		**r**D		**r**A		UIMM(1)		0 1 1 0 0 0 0 0 1 0 1	

1. **d** = UIMM * 8

if (rA = 0) then b ← 0
else b ← (rA)
EA ← b + EXTZ(UIMM*8)
$rD_{0:15}$ ← MEM(EA, 2)
$rD_{16:31}$ ← MEM(EA+2,2)
$rD_{32:47}$ ← MEM(EA+4,2)
$rD_{48:63}$ ← MEM(EA+6,2)

The double word addressed by EA is loaded from memory and placed in **r**D.

The figure below shows how bytes are loaded into **r**D as determined by the endian mode.

evldh Results in Big- and Little-Endian Modes

Byte address	0	1	2	3	4	5	6	7
Memory	a	b	c	d	e	f	g	h
GPR in big endian	a	b	c	d	e	f	g	h
GPR in little endian	b	a	d	c	f	e	h	g

Implementation note: If the EA is not double-word aligned, an alignment exception occurs.

evldhx | SPE APU | User | **evldhx**

Vector Load Double into Four Half Words Indexed

evldhx **r**D,**r**A,**r**B

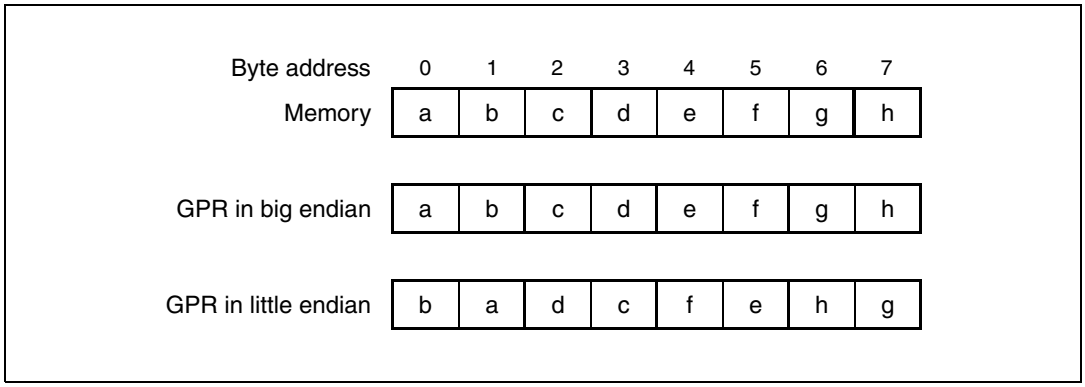
0	5	6	10	11	15	16	20	21	31
0 0 0 1 0 0		**r**D		**r**A		**r**B		0 1 1 0 0 0 0 0 1 0 0	

if (rA = 0) then b ← 0
else b ← (rA)
EA ← b + (rB)
$rD_{0:15}$ ← MEM(EA, 2)
$rD_{16:31}$ ← MEM(EA+2,2)
$rD_{32:47}$ ← MEM(EA+4,2)
$rD_{48:63}$ ← MEM(EA+6,2)

The double word addressed by EA is loaded from memory and placed in **r**D.

Figure 47 shows how bytes are loaded into **r**D as determined by the endian mode.

Figure 47. evldhx results in big- and little-endian modes



Implementation note: If the EA is not double-word aligned, an alignment exception occurs.

evldw | SPE APU | User | **evldw**

Vector load double into two words

evldw **rD,d(rA)**

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	0	1	0	0	**r**D					**r**A					UIMM(1)					0	1	1	0	0	0	0	0	0	1	1

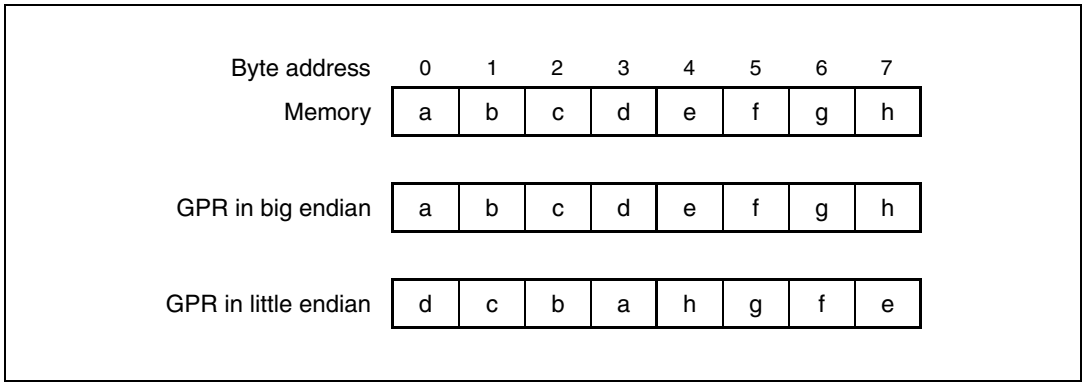
1. **d** = UIMM * 8

```
if (rA = 0) then b ← 0
else b ← (rA)
EA ← b + EXTZ(UIMM*8)
rD0:31 ← MEM(EA, 4)
rD32:63 ← MEM(EA+4, 4)
```

The double word addressed by EA is loaded from memory and placed in **r**D.

Figure 48 shows how bytes are loaded into **r**D as determined by the endian mode.

Figure 48. evldw results in big- and little-endian modes

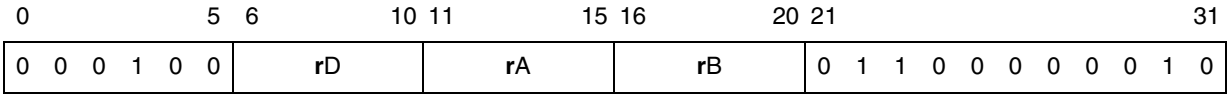


Implementation note: If the EA is not double-word aligned, an alignment exception occurs.

evldwx | SPE APU | User | **evldwx**

Vector load double into two words indexed

evldwx **r**D,**r**A,**r**B

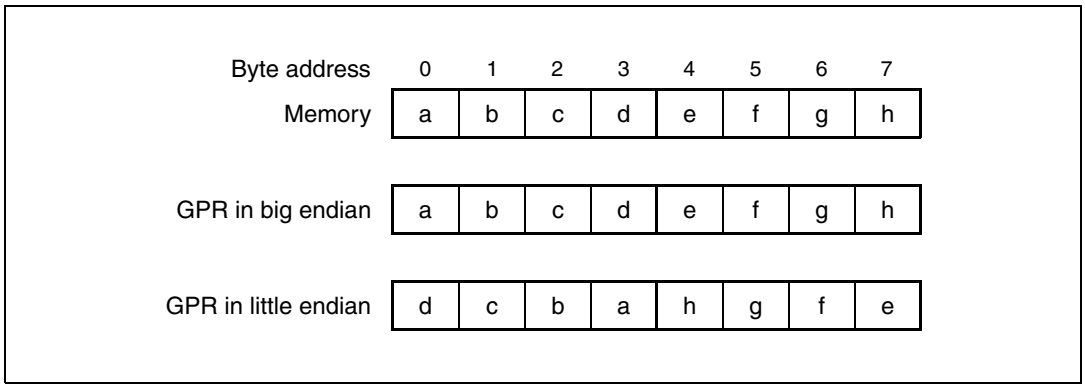


if (rA = 0) then b ← 0
else b ← (rA)
EA ← b + (rB)
$rD_{0:31}$ ← MEM(EA, 4)
$rD_{32:63}$ ← MEM(EA+4, 4)

The double word addressed by EA is loaded from memory and placed in **r**D.

Figure 49 shows how bytes are loaded into **r**D as determined by the endian mode.

Figure 49. evldwx results in big- and little-endian modes



Implementation note: If the EA is not double-word aligned, an alignment exception occurs.

evlhhesplat | SPE APU | User | **evlhhesplat**

Vector load half word into half words even and splat

evlhhesplat **r**D,**d**(**r**A)

0	5	6	10	11	15	16	20	21	31
0 0 0 1 0 0		**r**D		**r**A		UIMM(1)		0 1 1 0 0 0 0 1 0 0 1	

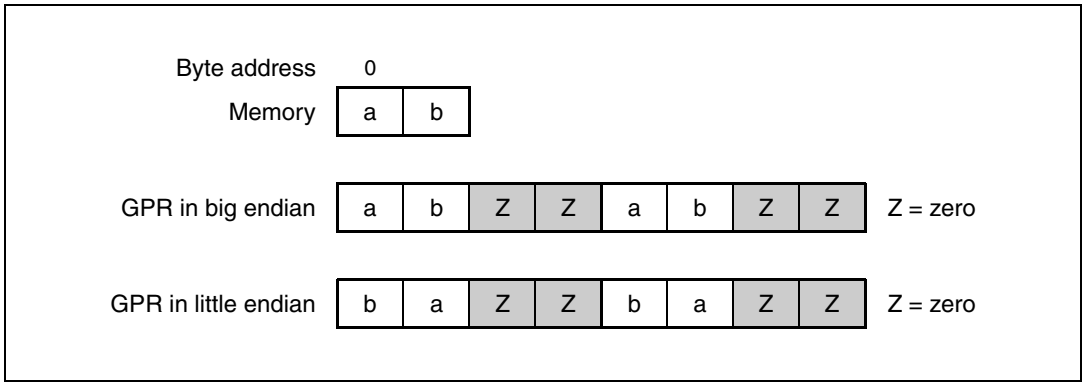
1. **d** = UIMM * 2

if (rA = 0) then b ← 0
else b ← (rA)
EA ← b + EXTZ(UIMM*2)
$rD_{0:15}$ ← MEM(EA,2)
$rD_{16:31}$ ← 0x0000
$rD_{32:47}$ ← MEM(EA,2)
$rD_{48:63}$ ← 0x0000

The half word addressed by EA is loaded from memory and placed in the even half words of each element of **r**D.

Figure 50 shows how bytes are loaded into **r**D as determined by the endian mode.

Figure 50. evlhhesplat results in big- and little-endian modes

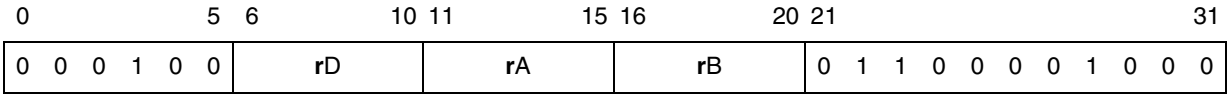


Implementation note: If the EA is not half-word aligned, an alignment exception occurs.

evlhhesplatx | SPE APU | User | **evlhhesplatx**

Vector load half word into half words even and splat indexed

evlhhesplatx **r**D,**r**A,**r**B

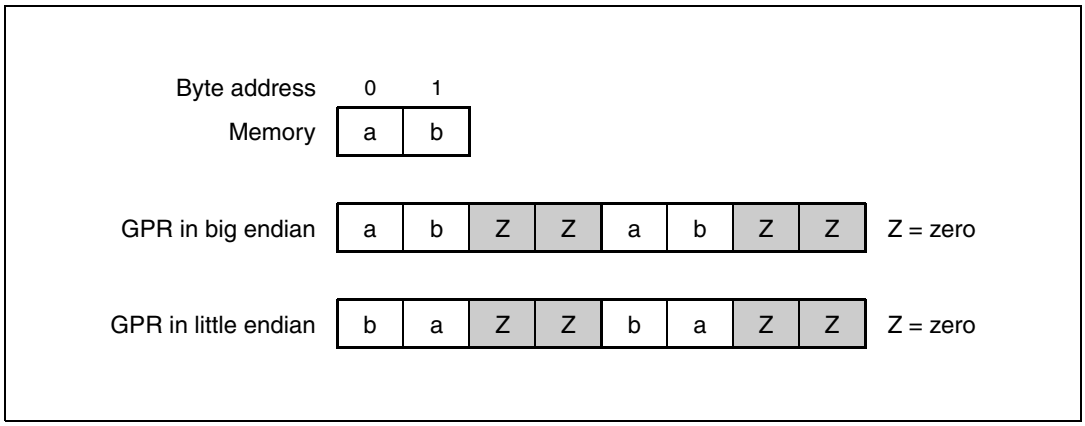


if (rA = 0) then b ← 0
else b ← (rA)
EA ← b + (rB)
$rD_{0:15}$ ← MEM(EA,2)
$rD_{16:31}$ ← 0x0000
$rD_{32:47}$ ← MEM(EA,2)
$rD_{48:63}$ ← 0x0000

The half word addressed by EA is loaded from memory and placed in the even half words of each element of **r**D.

Figure 51 shows how bytes are loaded into **r**D as determined by the endian mode.

Figure 51. evlhhesplatx results in big- and little-endian modes



Implementation note: If the EA is not half-word aligned, an alignment exception occurs.

evlhhossplat | SPE APU | User | **evlhhossplat**

Vector load half word into half word odd signed and splat

evlhhossplat **r**D,**d**(**r**A)

0	5	6	10	11	15	16	20	21	31
0 0 0 1 0 0		**r**D		**r**A		UIMM(1)		0 1 1 0 0 0 0 1 1 1 1	

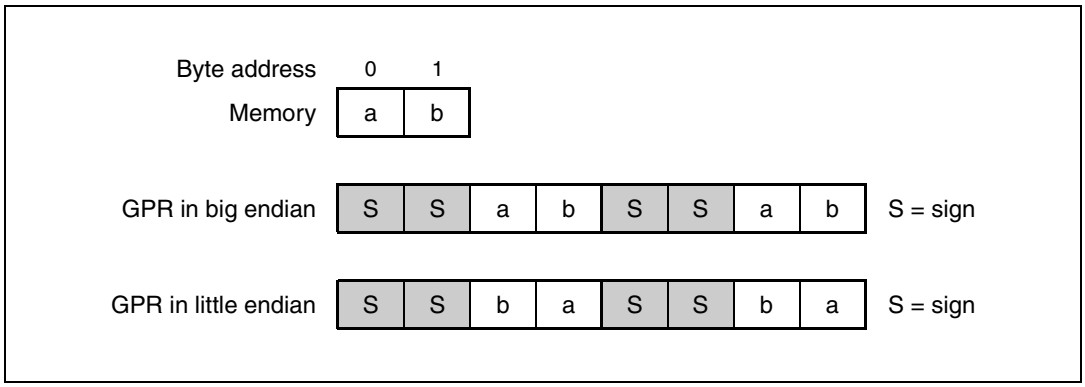
1. **d** = UIMM * 2

if (rA = 0) then b ← 0
else b ← (rA)
EA ← b + EXTZ(UIMM*2)
$rD_{0:31}$ ← EXTS(MEM(EA,2))
$rD_{32:63}$ ← EXTS(MEM(EA,2))

The half word addressed by EA is loaded from memory and placed in the odd half words sign extended in each element of **r**D.

Figure 52 shows how bytes are loaded into **r**D as determined by the endian mode.

Figure 52. evlhhossplat results in big- and little-endian modes



In big-endian memory, the msb of a is sign extended. In little-endian memory, the msb of b is sign extended.

Implementation note: If the EA is not half-word aligned, an alignment exception occurs.

evlhhossplatx | SPE APU | User | **evlhhossplatx**

Vector load half word into half word odd signed and splat indexed

evlhhossplatx **r**D,**r**A,**r**B

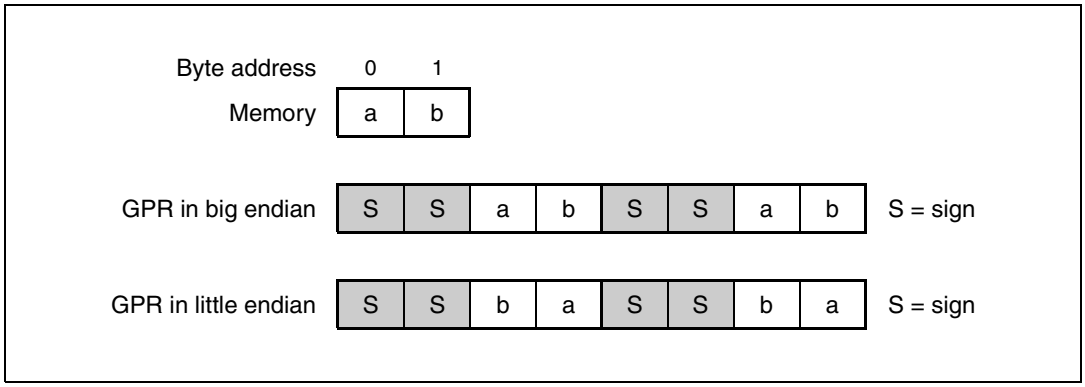
0	5	6	10	11	15	16	20	21	31
0 0 0 1 0 0		**r**D		**r**A		**r**B		0 1 1 0 0 0 0 1 1 1 0	

if (rA = 0) then b ← 0
else b ← (rA)
EA ← b + (rB)
$rD_{0:31}$ ← EXTS(MEM(EA,2))
$rD_{32:63}$ ← EXTS(MEM(EA,2))

The half word addressed by EA is loaded from memory and placed in the odd half words sign extended in each element of **r**D.

Figure 53 shows how bytes are loaded into **r**D as determined by the endian mode.

Figure 53. evlhhossplatx results in big- and little-endian modes



In big-endian memory, the msb of a is sign extended. In little-endian memory, the msb of b is sign extended.

Implementation note: If the EA is not half-word aligned, an alignment exception occurs.

evlhhousplat | SPE APU | User | **evlhhousplat**

Vector load half word into half word odd unsigned and splat

evlhhousplat **r**D,**d**(**r**A)

0	1	2	3	4	5	6–10	11–15	16–20	21	22	23	24	25	26	27	28	29	30	31
0	0	0	1	0	0	**r**D	**r**A	UIMM(1)	0	1	1	0	0	0	0	1	1	0	1

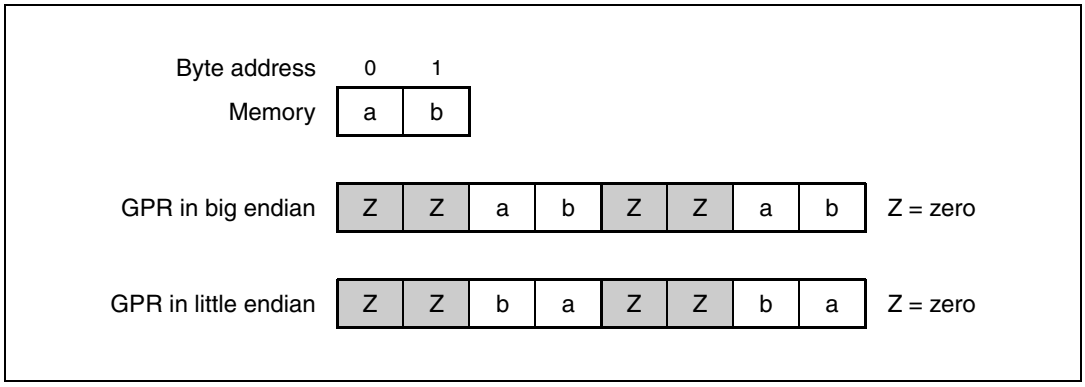
1. **d** = UIMM * 2

if (rA = 0) then b ← 0
else b ← (rA)
EA ← b + EXTZ(UIMM*2)
$rD_{0:15}$ ← 0x0000
$rD_{16:31}$ ← MEM(EA,2)
$rD_{32:47}$ ← 0x0000
$rD_{48:63}$ ← MEM(EA,2)

The half word addressed by EA is loaded from memory and placed in the odd half words zero extended in each element of **r**D.

Figure 54 shows how bytes are loaded into **r**D as determined by the endian mode.

Figure 54. evlhhousplat results in big- and little-endian modes

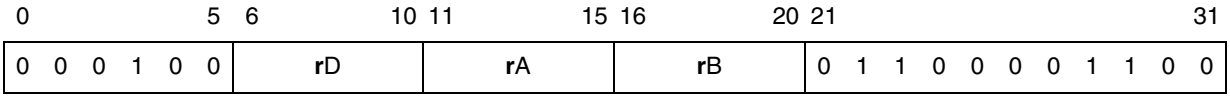


Implementation note: If the EA is not half-word aligned, an alignment exception occurs.

evlhhousplatx | SPE APU | User | **evlhhousplatx**

Vector load half word into half word odd unsigned and splat indexed

evlhhousplatx **r**D,**r**A,**r**B

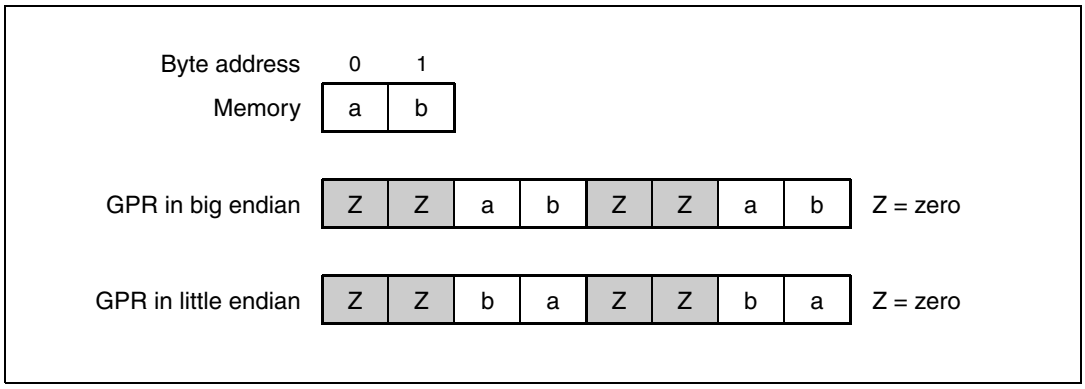


if (rA = 0) then b ← 0
else b ← (rA)
EA ← b + (rB)
$rD_{0:15}$ ← 0x0000
$rD_{16:31}$ ← MEM(EA,2)
$rD_{32:47}$ ← 0x0000
$rD_{48:63}$ ← MEM(EA,2)

The half word addressed by EA is loaded from memory and placed in the odd half words zero extended in each element of **r**D.

Figure 55 shows how bytes are loaded into **r**D as determined by the endian mode.

Figure 55. evlhhousplatx results in big- and little-endian modes



Implementation note: If the EA is not half-word aligned, an alignment exception occurs.

evlwhe | SPE APU | User | **evlwhe**

Vector load word into two half words even

evlwhe **r**D,**d**(**r**A)

0	5	6	10	11	15	16	20	21	31
0 0 0 1 0 0		**r**D		**r**A		UIMM(1)		0 1 1 0 0 0 1 0 0 0 1	

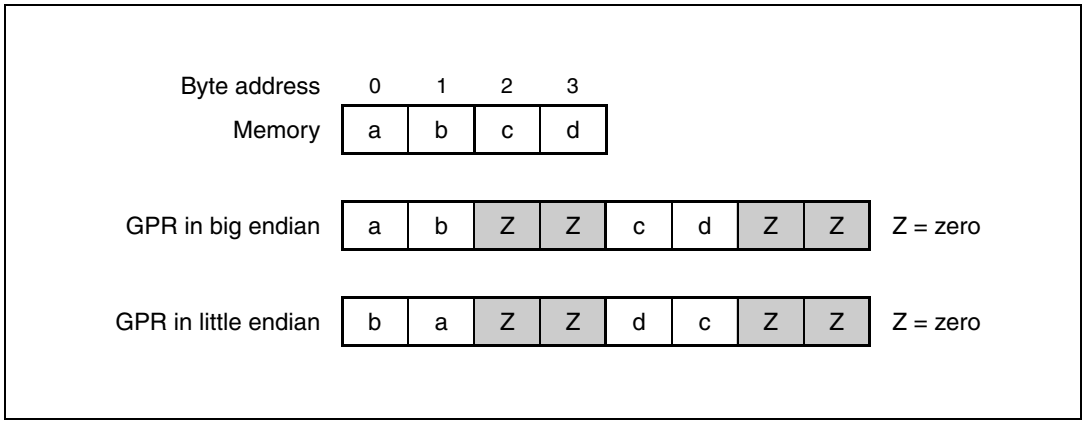
1. **d** = UIMM * 4

if (rA = 0) then b ← 0
else b ← (rA)
EA ← b + EXTZ(UIMM*4)
$rD_{0:15}$ ← MEM(EA,2)
$rD_{16:31}$ ← 0x0000
$rD_{32:47}$ ← MEM(EA+2,2)
$rD_{48:63}$ ← 0x0000

The word addressed by EA is loaded from memory and placed in the even half words in each element of **r**D.

Figure 56 shows how bytes are loaded into **r**D as determined by the endian mode.

Figure 56. evlwhe results in big- and little-endian modes

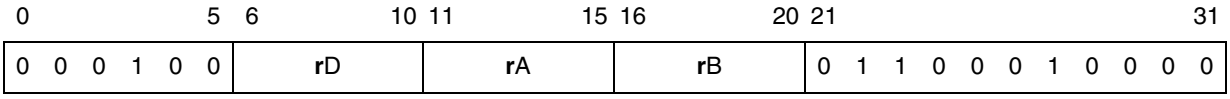


Implementation note: If the EA is not word aligned, an alignment exception occurs.

evlwhex | SPE APU | User | **evlwhex**

Vector load word into two half words even indexed

evlwhex **r**D,**r**A,**r**B

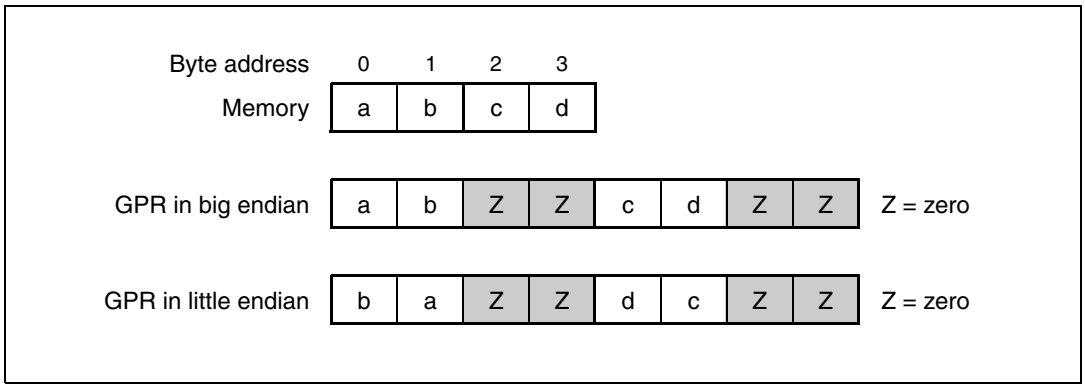


if (rA = 0) then b ← 0
else b ← (rA)
EA ← b + (rB)
$rD_{0:15}$ ← MEM(EA,2)
$rD_{16:31}$ ← 0x0000
$rD_{32:47}$ ← MEM(EA+2,2)
$rD_{48:63}$ ← 0x0000

The word addressed by EA is loaded from memory and placed in the even half words in each element of **r**D.

Figure 57 shows how bytes are loaded into **r**D as determined by the endian mode.

Figure 57. evlwhex results in big- and little-endian modes



Implementation note: If the EA is not word aligned, an alignment exception occurs.

evlwhos | SPE APU | User | **evlwhos**

Vector load word into two half words odd signed (with sign extension)

evlwhos **rD,d(rA)**

0	5	6	10	11	15	16	20	21	31
0 0 0 1 0 0		**r**D		**r**A		UIMM(1)		0 1 1 0 0 0 1 0 1 1 1	

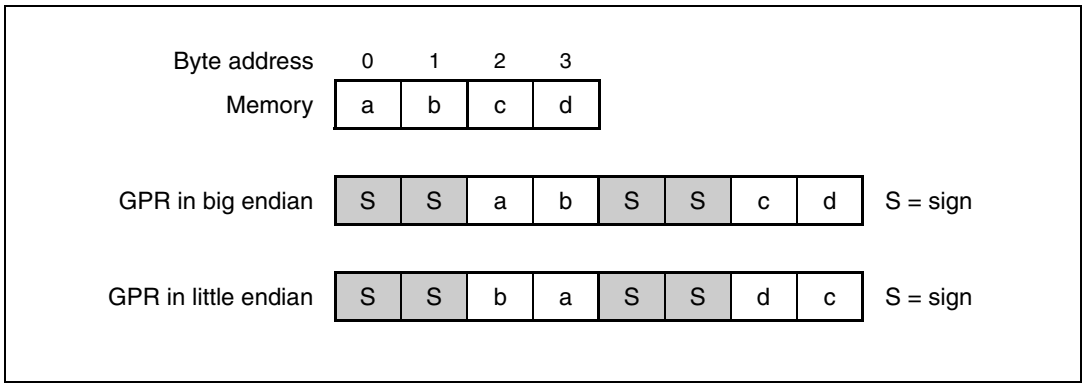
1. **d** = UIMM * 4

if (rA = 0) then b ← 0
else b ← (rA)
EA ← b + EXTZ(UIMM*4)
$rD_{0:31}$ ← EXTS(MEM(EA,2))
$rD_{32:63}$ ← EXTS(MEM(EA+2,2))

The word addressed by EA is loaded from memory and placed in the odd half words sign extended in each element of **r**D.

Figure 58 shows how bytes are loaded into **r**D as determined by the endian mode.

Figure 58. evlwhos results in big- and little-endian modes



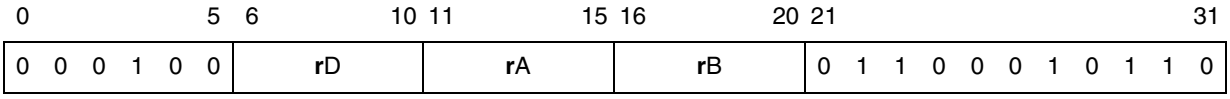
In big-endian memory, the most significant bits of a and c are sign extended. In little-endian memory, the most significant bits of b and d are sign extended.

Implementation note: If the EA is not word aligned, an alignment exception occurs.

evlwhosx | SPE APU | User | **evlwhosx**

Vector load word into two half words odd signed indexed (with sign extension)

evlwhosx **r**D,**r**A,**r**B

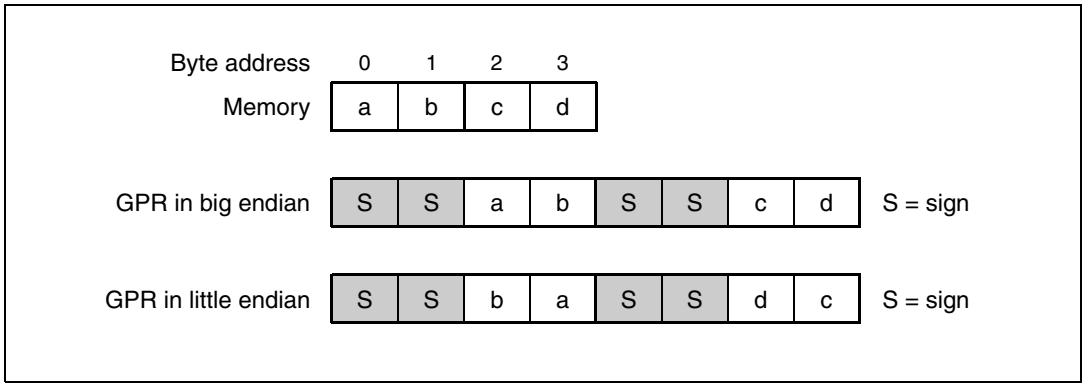


if (rA = 0) then b ← 0
else b ← (rA)
EA ← b + (rB)
$rD_{0:31}$ ← EXTS(MEM(EA,2))
$rD_{32:63}$ ← EXTS(MEM(EA+2,2))

The word addressed by EA is loaded from memory and placed in the odd half words sign extended in each element of **r**D.

Figure 59 shows how bytes are loaded into **r**D as determined by the endian mode.

Figure 59. evlwhosx results in big- and little-endian modes



In big-endian memory, the most significant bits of a and c are sign extended. In little-endian memory, the most significant bits of b and d are sign extended.

Implementation note: If the EA is not word aligned, an alignment exception occurs.

evlwhou | SPE APU | User | **evlwhou**

Vector load word into two half words odd unsigned (zero-extended)

evlwhou **r**D,**d**(**r**A)

0	5	6	10	11	15	16	20	21	31
0 0 0 1 0 0		**r**D		**r**A		UIMM(1)		0 1 1 0 0 0 1 0 1 0 1	

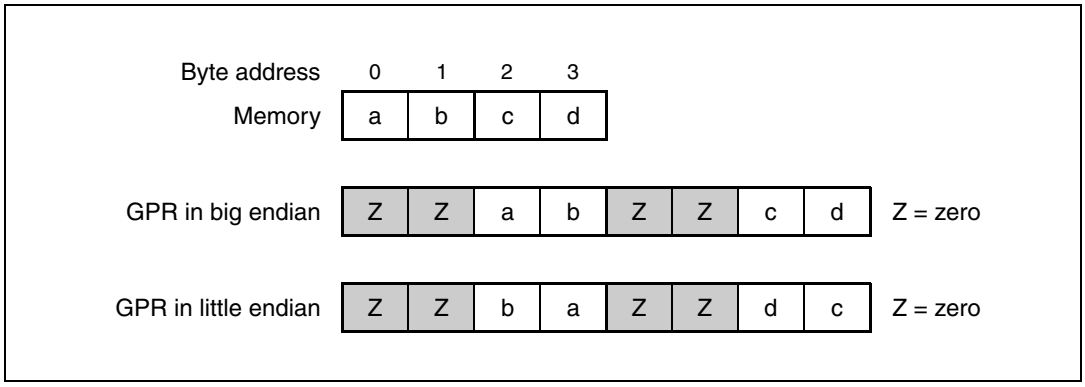
1. **d** = UIMM * 4

```
if (rA = 0) then b ← 0
else b ← (rA)
EA ← b + EXTZ(UIMM*4)
rD0:15 ← 0x0000
rD16:31 ← MEM(EA,2)
rD32:47 ← 0x0000
rD48:63 ← MEM(EA+2,2)
```

The word addressed by EA is loaded from memory and placed in the odd half words zero extended in each element of **r**D.

Figure 60 shows how bytes are loaded into **r**D as determined by the endian mode.

Figure 60. evlwhou results in big- and little-endian modes



Implementation note: If the EA is not word aligned, an alignment exception occurs.

evlwhoux | SPE APU | User | **evlwhoux**

Vector load word into two half words odd unsigned indexed (zero-extended)

evlwhoux **r**D,**r**A,**r**B

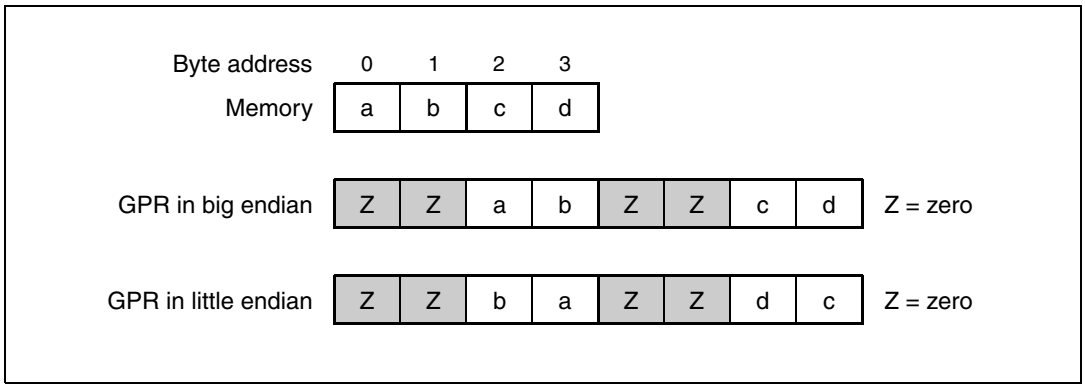
0	5	6	10	11	15	16	20	21	31
0 0 0 1 0 0		**r**D		**r**A		**r**B		0 1 1 0 0 0 1 0 1 0 0	

if (rA = 0) then b ← 0
else b ← (rA)
EA ← b + (rB)
$rD_{0:15}$ ← 0x0000
$rD_{16:31}$ ← MEM(EA,2)
$rD_{32:47}$ ← 0x0000
$rD_{48:63}$ ← MEM(EA+2,2)

The word addressed by EA is loaded from memory and placed in the odd half words zero extended in each element of **r**D.

Figure 61 shows how bytes are loaded into **r**D as determined by the endian mode.

Figure 61. evlwhoux results in big- and little-endian modes



Implementation note: If the EA is not word aligned, an alignment exception occurs.

evlwhsplat | SPE APU | User | **evlwhsplat**

Vector load word into two half words and splat

evlwhsplat **rD,d(rA)**

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	0	1	0	0	**r**D					**r**A					UIMM(1)					0	1	1	0	0	0	1	1	1	0	1

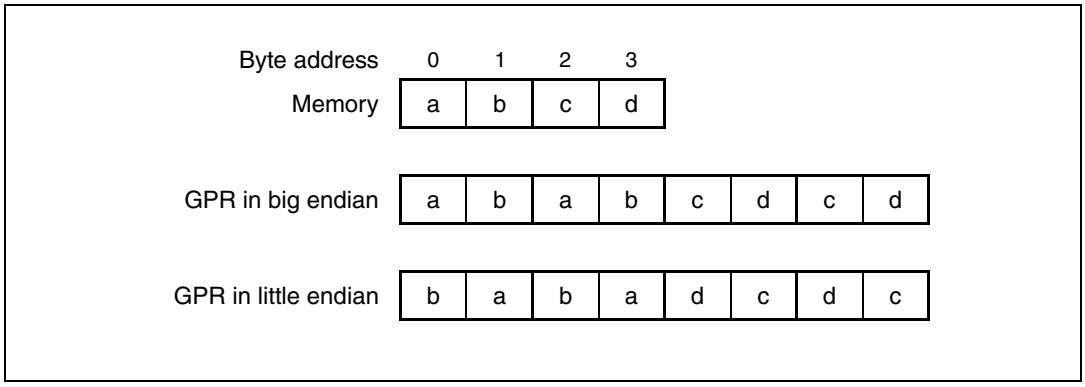
1. **d** = UIMM * 4

```
if (rA = 0) then b ← 0
else b ← (rA)
EA ← b + EXTZ(UIMM*4)
rD0:15 ← MEM(EA,2)
rD16:31 ← MEM(EA,2)
rD32:47 ← MEM(EA+2,2)
rD48:63 ← MEM(EA+2,2)
```

The word addressed by EA is loaded from memory and placed in both the even and odd half words in each element of **r**D.

Figure 62 shows how bytes are loaded into **r**D as determined by the endian mode.

Figure 62. evlwhsplat results in big- and little-endian modes



Implementation note: If the EA is not word aligned, an alignment exception occurs.

evlwhsplatx | SPE APU | User | **evlwhsplatx**

Vector load word into two half words and splat indexed

evlwhsplatx **r**D,**r**A,**r**B

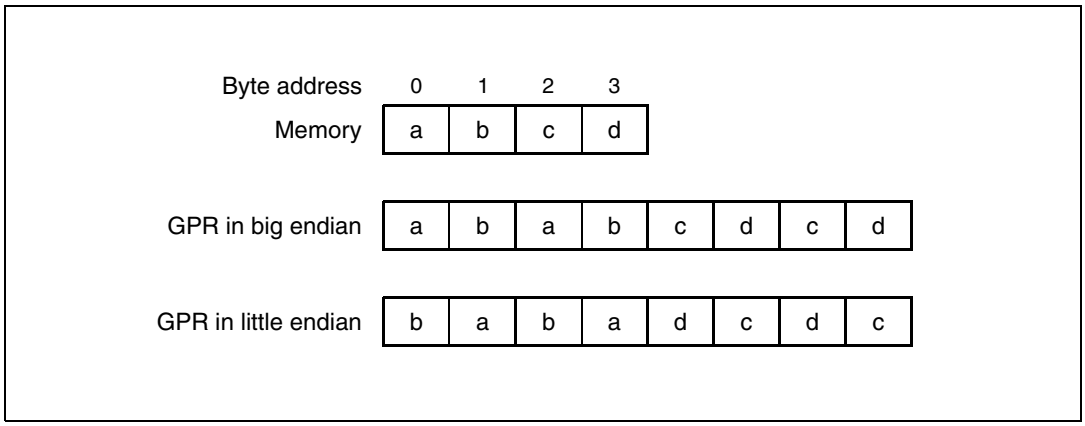
0	5	6	10	11	15	16	20	21	31
0 0 0 1 0 0		**r**D		**r**A		**r**B		0 1 1 0 0 0 1 1 1 0 0	

if (rA = 0) then b ← 0
else b ← (rA)
EA ← b + (rB)
$rD_{0:15}$ ← MEM(EA,2)
$rD_{16:31}$ ← MEM(EA,2)
$rD_{32:47}$ ← MEM(EA+2,2)
$rD_{48:63}$ ← MEM(EA+2,2)

The word addressed by EA is loaded from memory and placed in both the even and odd half words in each element of **r**D.

Figure 63 shows how bytes are loaded into **r**D as determined by the endian mode.

Figure 63. evlwhsplatx results in big- and little-endian modes



Implementation note: If the EA is not word aligned, an alignment exception occurs.

evlwwsplat | SPE APU | User | **evlwwsplat**

Vector load word into word and splat

evlwwsplat **r**D,**d**(**r**A)

0	5	6	10	11	15	16	20	21	31
0 0 0 1 0 0		**r**D		**r**A		UIMM(1)		0 1 1 0 0 0 1 1 0 0 1	

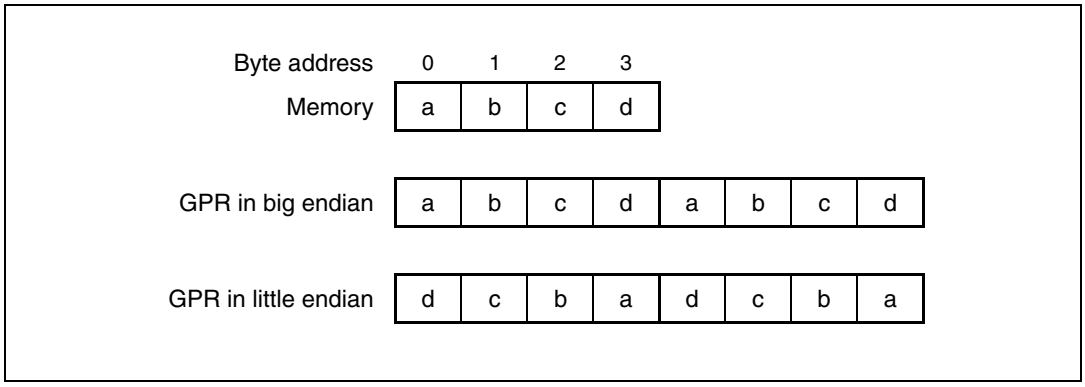
1. **d** = UIMM * 4

if (rA = 0) then b ← 0
else b ← (rA)
EA ← b + EXTZ(UIMM*4)
$rD_{0:31}$ ← MEM(EA,4)
$rD_{32:63}$ ← MEM(EA,4)

The word addressed by EA is loaded from memory and placed in both elements of **r**D.

Figure 64 shows how bytes are loaded into **r**D as determined by the endian mode.

Figure 64. evlwwsplat results in big- and little-endian modes



Implementation note: If the EA is not word aligned, an alignment exception occurs.

evlwwsplatx | SPE APU | User | **evlwwsplatx**

Vector load word into word and splat indexed

evlwwsplatx **r**D,**r**A,**r**B

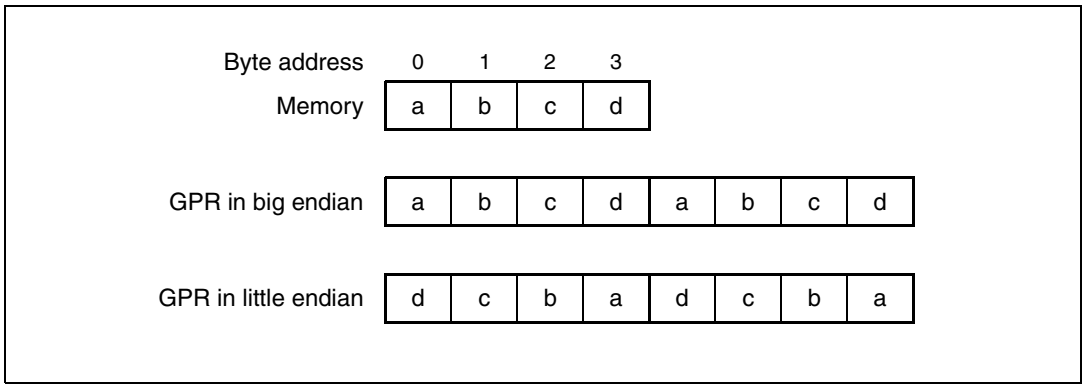
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	0	1	0	0	**r**D					**r**A					**r**B					0	1	1	0	0	0	1	1	0	0	0

if (rA = 0) then b ← 0
else b ← (rA)
EA ← b + (rB)
$rD_{0:31}$ ← MEM(EA,4)
$rD_{32:63}$ ← MEM(EA,4)

The word addressed by EA is loaded from memory and placed in both elements of **r**D.

Figure 65 shows how bytes are loaded into **r**D as determined by the endian mode.

Figure 65. evlwwsplatx results in big- and little-endian modes

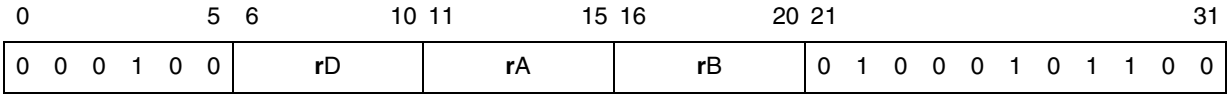


Implementation note: If the EA is not word aligned, an alignment exception occurs.

evmergehi | SPE, Ve;ctor SPFP, scalar DPFP APUs | User | **evmergehi**

Vector merge high

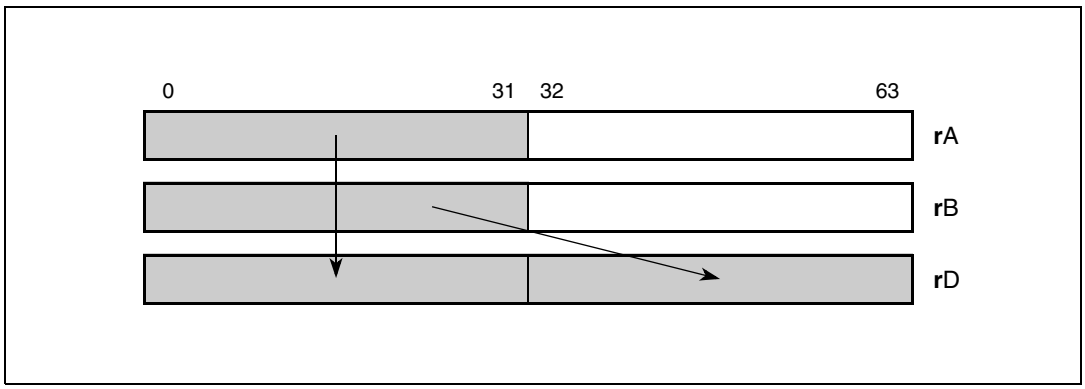
evmergehi **r**D,**r**A,**r**B



$rD_{0:31} \leftarrow rA_{0:31}$
$rD_{32:63} \leftarrow rB_{0:31}$

The high-order elements of **r**A and **r**B are merged and placed into **r**D, as shown in *Figure 66*.

Figure 66. High order element merging (evmergehi)



*Note: A vector splat high can be performed by specifying the same register in **r**A and **r**B.*

evmergehilo | SPE, Vector SPFP, Scalar DPFP APUs | User | **evmergehilo**

Vector merge high/low

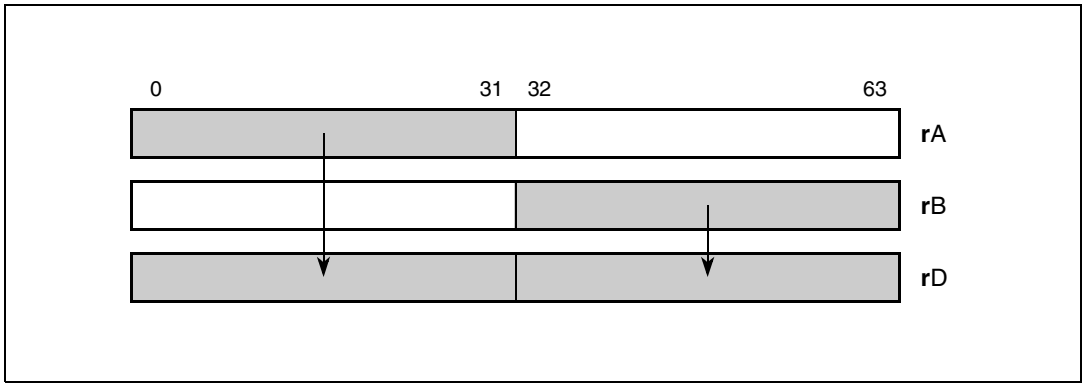
evmergehilo **r**D,**r**A,**r**B

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	0	1	0	0	**r**D					**r**A					**r**B					0	1	0	0	0	1	0	1	1	1	0

$rD_{0:31} \leftarrow rA_{0:31}$
$rD_{32:63} \leftarrow rB_{32:63}$

The high-order element of **r**A and the low-order element of **r**B are merged and placed into **r**D, as shown in *Figure 67*.

Figure 67. High order element merging (evmergehilo)

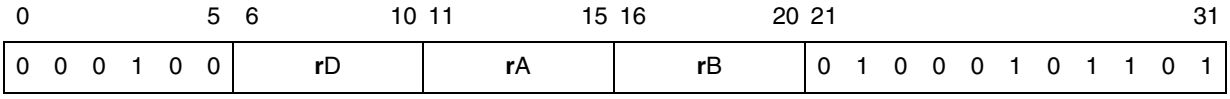


Application note: With appropriate specification of **r**A and **r**B, **evmergehi**, **evmergelo**, **evmergehilo**, and **evmergelohi** provide a full 32-bit permute of two source operands.

evmergelo | SPE APU | User | **evmergelo**

Vector merge low

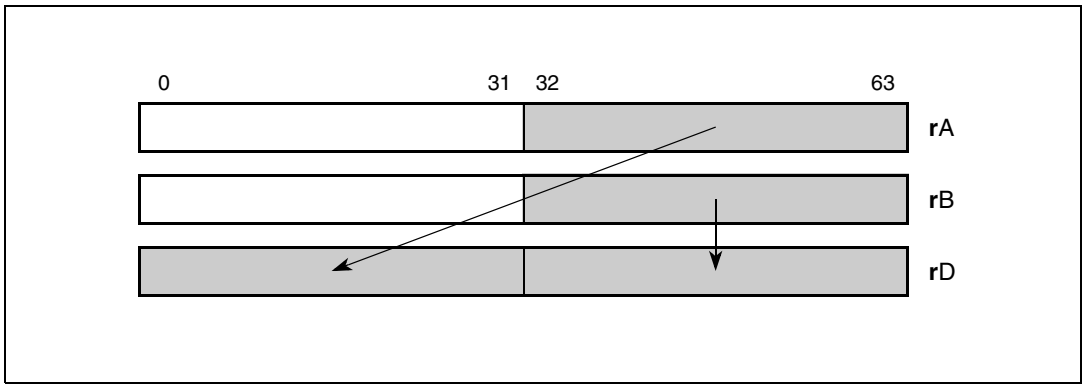
evmergelo **r**D,**r**A,**r**B



$rD_{0:31} \leftarrow rA_{32:63}$
$rD_{32:63} \leftarrow rB_{32:63}$

The low-order elements of **r**A and **r**B are merged and placed in **r**D, as shown in *Figure 68*.

Figure 68. Low order element merging (evmergelo)

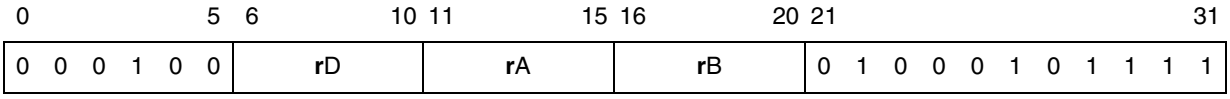


*Note: A vector splat low can be performed by specifying the same register in **r**A and **r**B.*

evmergelohi | SPE APU | User | **evmergelohi**

Vector merge low/high

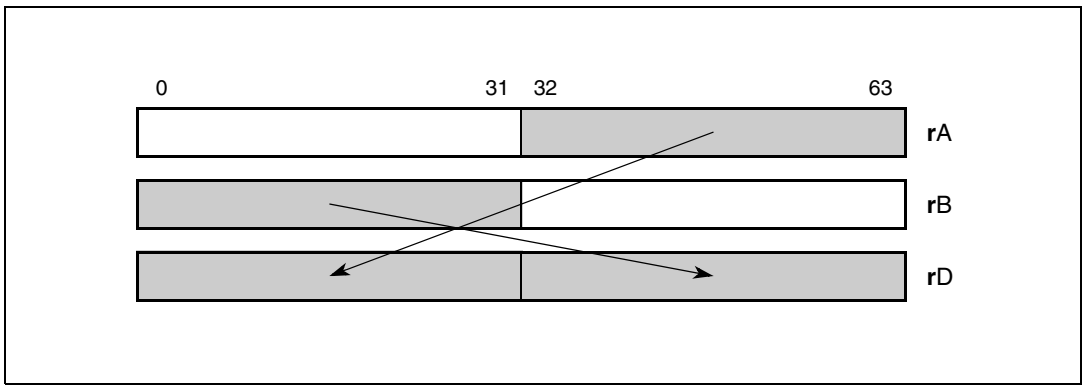
evmergelohi **r**D,**r**A,**r**B



$rD_{0:31} \leftarrow rA_{32:63}$
$rD_{32:63} \leftarrow rB_{0:31}$

The low-order element of **r**A and the high-order element of **r**B are merged and placed into **r**D, as shown in *Figure 69*.

Figure 69. Low order element merging (evmergelohi)

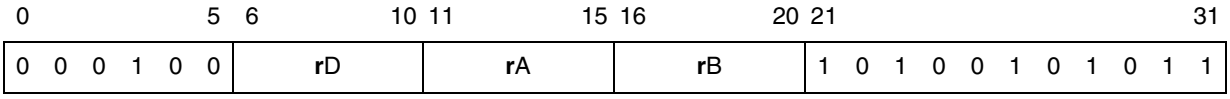


*Note: A vector swap can be performed by specifying the same register in **r**A and **r**B.*

evmhegsmfaa | SPE APU | User | **evmhegsmfaa**

Vector multiply half words, even, guarded, signed, modulo, fractional and accumulate

evmhegsmfaa **r**D,**r**A,**r**B



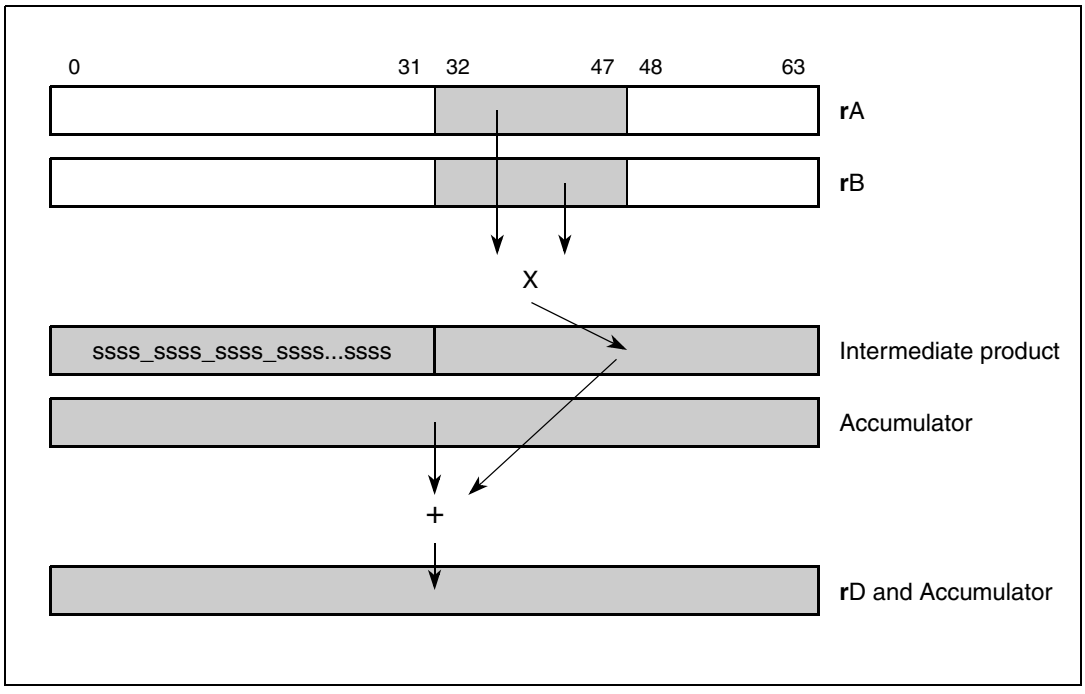
$temp_{0:31} \leftarrow rA_{32:47} \times_{sf} rB_{32:47}$
$temp_{0:63} \leftarrow EXTS(temp_{0:31})$
$rD_{0:63} \leftarrow ACC_{0:63} + temp_{0:63}$

// update accumulator
$ACC_{0:63} \leftarrow rD_{0:63}$

The corresponding low even-numbered, half-word signed fractional elements in **r**A and **r**B are multiplied. The product is added to the contents of the 64-bit accumulator and the result is placed into **r**D and the accumulator.

Note: This is a modulo sum. There is no overflow check and no saturation is performed. Any overflow of the 64-bit sum is not recorded into the SPEFSCR.

Figure 70. evmhegsmfaa (even form)



evmhegsmfan | SPE APU | User | **evmhegsmfan**

Vector multiply half words, even, guarded, signed, modulo, fractional and accumulate negative

evmhegsmfan **r**D,**r**A,**r**B

0	5	6	10	11	15	16	20	21	31
0 0 0 1 0 0		**r**D		**r**A		**r**B		1 0 1 1 0 1 0 1 0 1 1	

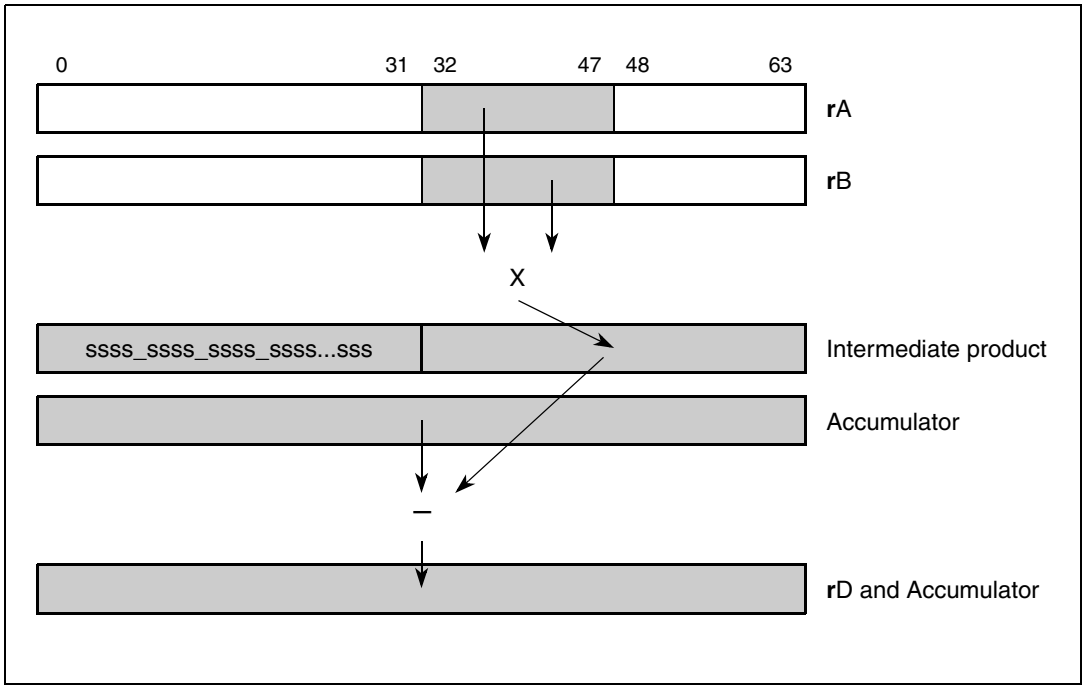
$temp_{0:31} \leftarrow rA_{32:47} \times_{sf} rB_{32:47}$
$temp_{0:63} \leftarrow EXTS(temp_{0:31})$
$rD_{0:63} \leftarrow ACC_{0:63} - temp_{0:63}$

// update accumulator
$ACC_{0:63} \leftarrow rD_{0:63}$

The corresponding low even-numbered, half-word signed fractional elements in **r**A and **r**B are multiplied. The product is subtracted from the contents of the 64-bit accumulator and the result is placed into **r**D and the accumulator.

Note: This is a modulo difference. There is no overflow check and no saturation is performed. Any overflow of the 64-bit difference is not recorded into the SPEFSCR.

Figure 71. evmhegsmfan (even form)



evmhegsmiaa | SPE APU | User | **evmhegsmiaa**

Vector multiply half words, even, guarded, signed, modulo, integer and accumulate

evmhegsmiaa **r**D,**r**A,**r**B

0	5	6	10	11	15	16	20	21	31
0 0 0 1 0 0		**r**D		**r**A		**r**B		1 0 1 0 0 1 0 1 0 0 1	

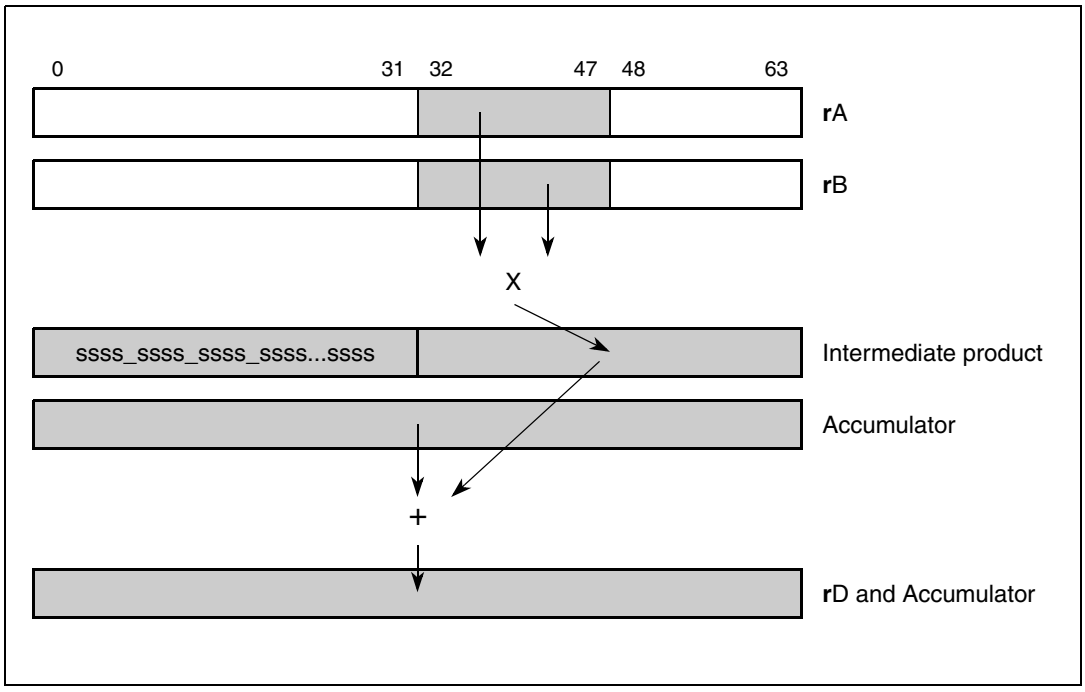
$temp_{0:31} \leftarrow rA_{32:47} \times_{si} rB_{32:47}$
$temp_{0:63} \leftarrow EXTS(temp_{0:31})$
$rD_{0:63} \leftarrow ACC_{0:63} + temp_{0:63}$

// update accumulator
$ACC_{0:63} \leftarrow rD_{0:63}$

The corresponding low even-numbered half-word signed integer elements in **r**A and **r**B are multiplied. The intermediate product is sign-extended and added to the contents of the 64-bit accumulator, and the resulting sum is placed into **r**D and into the accumulator.

Note: This is a modulo sum. There is no overflow check and no saturation is performed. Any overflow of the 64-bit sum is not recorded into the SPEFSCR.

Figure 72. evmhegsmiaa (even form)



evmhegsmian | SPE APU | User | **evmhegsmian**

Vector multiply half words, even, guarded, signed, modulo, integer and accumulate negative

evmhegsmian **r**D,**r**A,**r**B

0	5	6	10	11	15	16	20	21	31
0 0 0 1 0 0		**r**D		**r**A		**r**B		1 0 1 1 0 1 0 1 0 0 1	

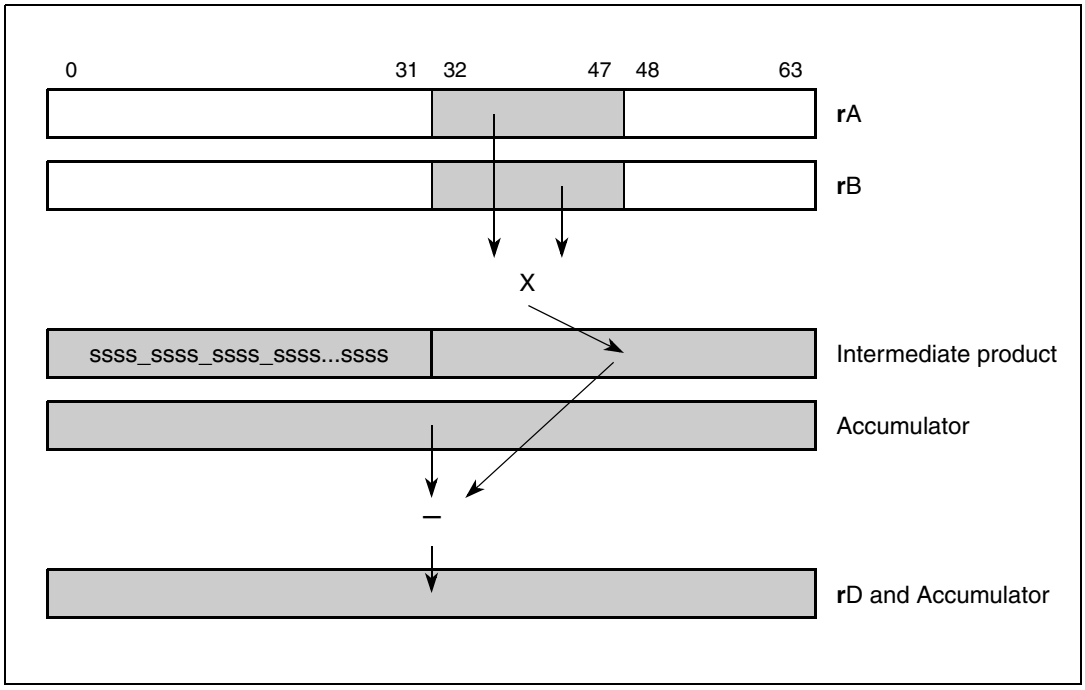
$temp_{0:31} \leftarrow rA_{32:47} \times_{si} rB_{32:47}$
$temp_{0:63} \leftarrow EXTS(temp_{0:31})$
$rD_{0:63} \leftarrow ACC_{0:63} - temp_{0:63}$

// update accumulator
$ACC_{0:63} \leftarrow rD_{0:63}$

The corresponding low even-numbered half-word signed integer elements in **r**A and **r**B are multiplied. The intermediate product is sign-extended and subtracted from the contents of the 64-bit accumulator, and the result is placed into **r**D and into the accumulator.

Note: This is a modulo difference. There is no check for overflow and no saturation is performed. Any overflow of the 64-bit difference is not recorded into the SPEFSCR.

Figure 73. evmhegsmian (even form)



evmhegumiaa | SPE APU | User | **evmhegumiaa**

Vector multiply half words, even, guarded, unsigned, modulo, integer and accumulate

evmhegumiaa **r**D,**r**A,**r**B

0	5	6	10	11	15	16	20	21	31
0 0 0 1 0 0		**r**D		**r**A		**r**B		1 0 1 0 0 1 0 1 0 0 0	

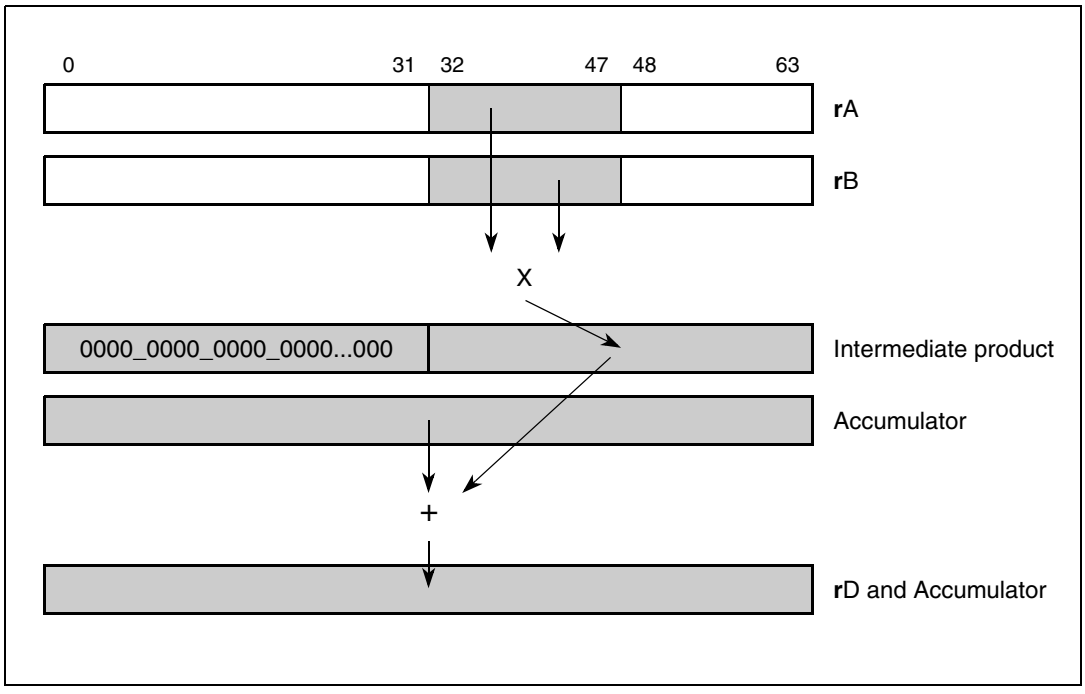
$temp_{0:31} \leftarrow rA_{32:47} \times_{ui} rB_{32:47}$
$temp_{0:63} \leftarrow EXTZ(temp_{0:31})$
$rD_{0:63} \leftarrow ACC_{0:63} + temp_{0:63}$

// update accumulator
$ACC_{0:63} \leftarrow rD_{0:63}$

The corresponding low even-numbered half-word unsigned integer elements in **r**A and **r**B are multiplied. The intermediate product is zero-extended and added to the contents of the 64-bit accumulator. The resulting sum is placed into **r**D and into the accumulator.

Note: This is a modulo sum. There is no overflow check and no saturation is performed. Any overflow of the 64-bit sum is not recorded into the SPEFSCR.

Figure 74. evmhegumiaa (even form)



evmhegumian | SPE APU | User | **evmhegumian**

Vector multiply half words, even, guarded, unsigned, modulo, integer and accumulate negative

evmhegumian **r**D,**r**A,**r**B

0	5	6	10	11	15	16	20	21	31
0 0 0 1 0 0		**r**D		**r**A		**r**B		1 0 1 1 0 1 0 1 0 0 0	

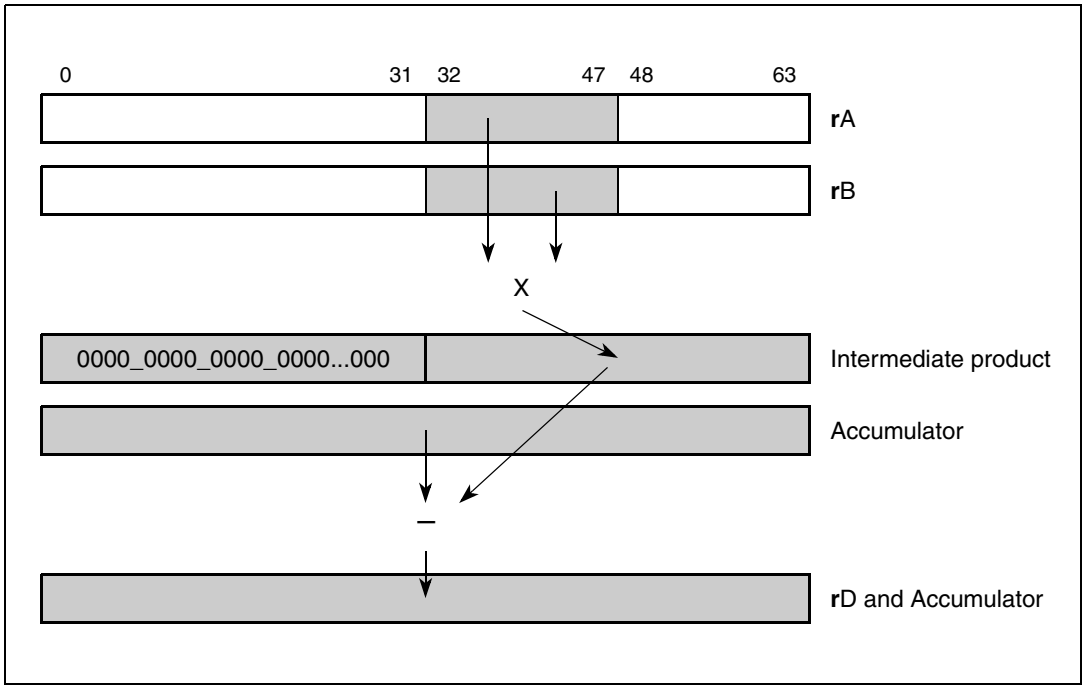
$temp_{0:31} \leftarrow rA_{32:47} \times_{ui} rB_{32:47}$
$temp_{0:63} \leftarrow EXTZ(temp_{0:31})$
$rD_{0:63} \leftarrow ACC_{0:63} - temp_{0:63}$

// update accumulator
$ACC_{0:63} \leftarrow rD_{0:63}$

The corresponding low even-numbered unsigned integer elements in **r**A and **r**B are multiplied. The intermediate product is zero-extended and subtracted from the contents of the 64-bit accumulator. The result is placed into **r**D and into the accumulator.

Note: This is a modulo difference. There is no check for overflow and no saturation is performed. Any overflow of the 64-bit difference is not recorded into the SPEFSCR.

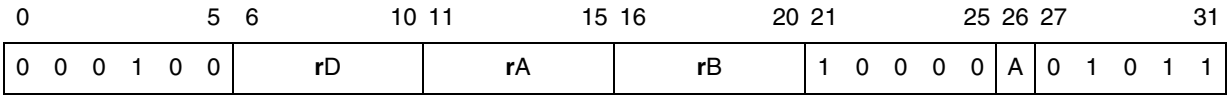
Figure 75. evmhegumian (even form)



evmhesmf | SPE APU | User | **evmhesmf**

Vector multiply half words, even, signed, modulo, fractional (to accumulator)

evmhesmf	**r**D,**r**A,**r**B	**(A = 0)**
evmhesmfa	**r**D,**r**A,**r**B	**(A = 1)**



// high
$rD_{0:31} \leftarrow (rA_{0:15} \times_{sf} rB_{0:15})$

// low
$rD_{32:63} \leftarrow (rA_{32:47} \times_{sf} rB_{32:47})$

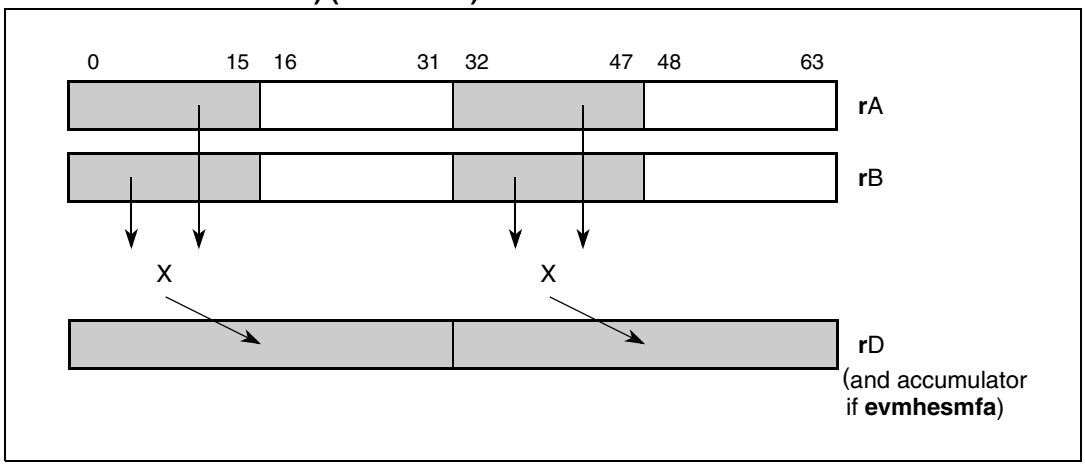
// update accumulator
if A = 1 then $ACC_{0:63} \leftarrow rD_{0:63}$

The corresponding even-numbered half-word signed fractional elements in **r**A and **r**B are multiplied then placed into the corresponding words of **r**D.

If A = 1, the result in **r**D is also placed into the accumulator.

Other registers altered: ACC (If A = 1)

Figure 76. Even multiply of two signed modulo fractional elements (to accumulator) (evmhesmf)



evmhesmfaaw | SPE APU | User | **evmhesmfaaw**

Vector multiply half words, even, signed, modulo, fractional and accumulate into words

evmhesmfaaw **r**D,**r**A,**r**B

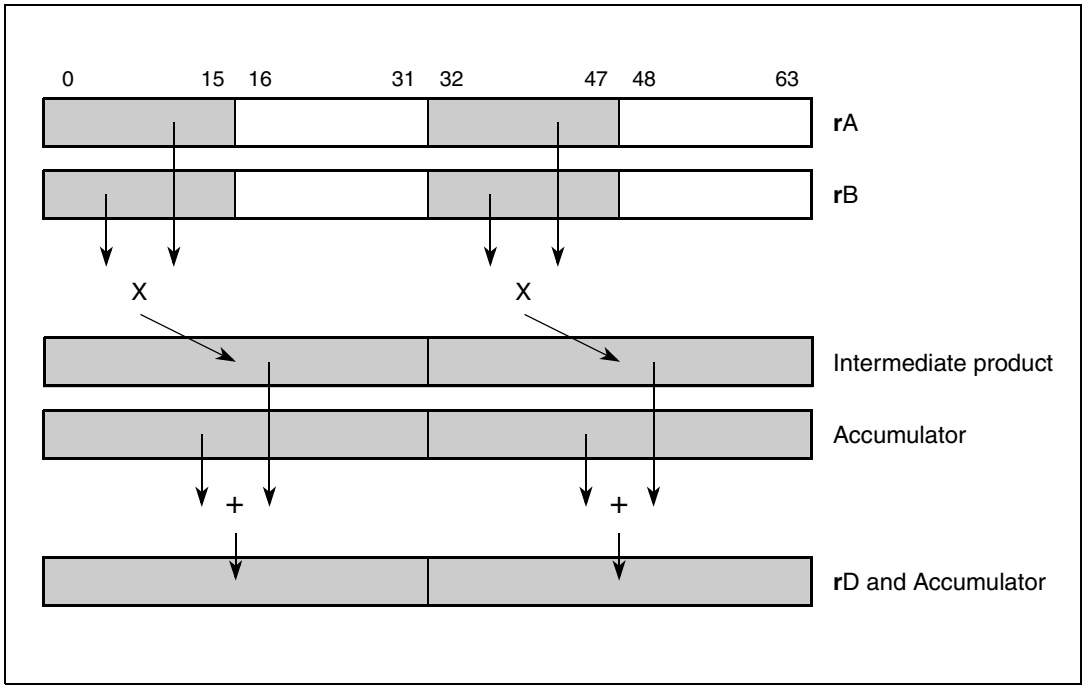
0	5	6	10	11	15	16	20	21	31
0 0 0 1 0 0		**r**D		**r**A		**r**B		1 0 1 0 0 0 0 1 0 1 1	

```
// high
temp0:31 ← (rA0:15 ×sf rB0:15)
rD0:31 ← ACC0:31 + temp0:31

// low
temp0:31 ← (rA32:47 ×sf rB32:47)
rD32:63 ← ACC32:63 + temp0:31

// update accumulator
ACC0:63 ← rD0:63
```

For each word element in the accumulator, the corresponding even-numbered half-word signed fractional elements in **r**A and **r**B 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 **r**D words and into the accumulator.

Other registers altered: ACC

Figure 77. Even form of vector half-word multiply (evmhesmfaaw)



evmhesmfanw | SPE APU | User | **evmhesmfanw**

Vector multiply half words, even, signed, modulo, fractional and accumulate negative into words

evmhesmfanw **r**D,**r**A,**r**B

0	5	6	10	11	15	16	20	21	31
0 0 0 1 0 0		**r**D		**r**A		**r**B		1 0 1 1 0 0 0 1 0 1 1	

// high
$temp_{0:31} \leftarrow rA_{0:15} \times_{sf} rB_{0:15}$
$rD_{0:31} \leftarrow ACC_{0:31} - temp_{0:31}$

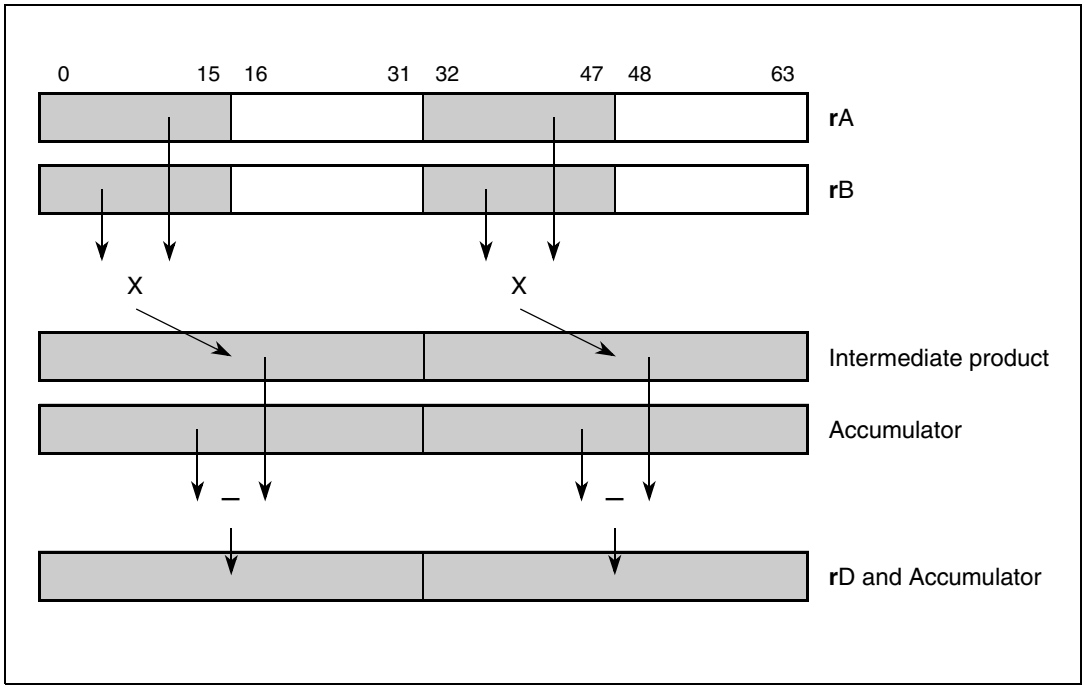
// low
$temp_{0:31} \leftarrow rA_{32:47} \times_{sf} rB_{32:47}$
$rD_{32:63} \leftarrow ACC_{32:63} - temp_{0:31}$

// update accumulator
$ACC_{0:63} \leftarrow rD_{0:63}$

For each word element in the accumulator, the corresponding even-numbered half-word signed fractional elements in **r**A and **r**B 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 **r**D words and into the accumulator.

Other registers altered: ACC

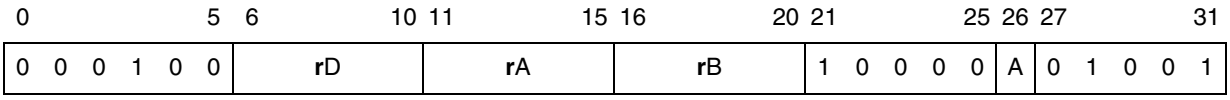
Figure 78. Even form of vector half-word multiply (evmhesmfanw)



evmhesmi | SPE APU | User | **evmhesmi**

Vector multiply half words, even, signed, modulo, integer (to accumulator)

evmhesmi **r**D,**r**A,**r**B **(A = 0)**
evmhesmia **r**D,**r**A,**r**B **(A = 1)**



// high
$rD_{0:31} \leftarrow rA_{0:15} \times_{si} rB_{0:15}$

// low
$rD_{32:63} \leftarrow rA_{32:47} \times_{si} rB_{32:47}$

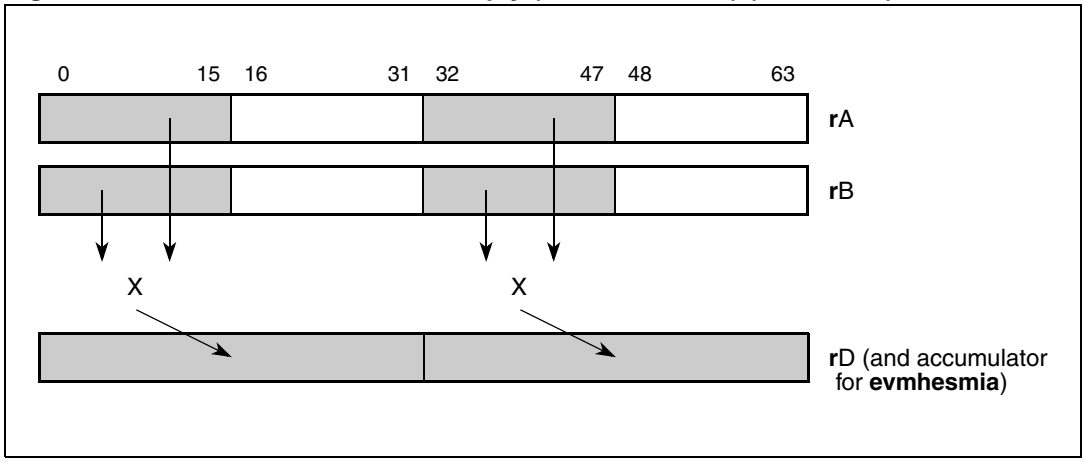
// update accumulator
if A = 1, then $ACC_{0:63} \leftarrow rD_{0:63}$

The corresponding even-numbered half-word signed integer elements in **r**A and **r**B are multiplied. The two 32-bit products are placed into the corresponding words of **r**D.

If A = 1, the result in **r**D is also placed into the accumulator.

Other registers altered: ACC (If A = 1)

Figure 79. Even form for vector multiply (to accumulator) (evmhesmi)



evmhesmiaaw | SPE APU | User | **evmhesmiaaw**

Vector multiply half words, even, signed, modulo, integer and accumulate into words

evmhesmiaaw **r**D,**r**A,**r**B

0	5	6	10	11	15	16	20	21	31
0 0 0 1 0 0		**r**D		**r**A		**r**B		1 0 1 0 0 0 0 1 0 0 1	

// high
$temp_{0:31} \leftarrow rA_{0:15} \times_{si} rB_{0:15}$
$rD_{0:31} \leftarrow ACC_{0:31} + temp_{0:31}$

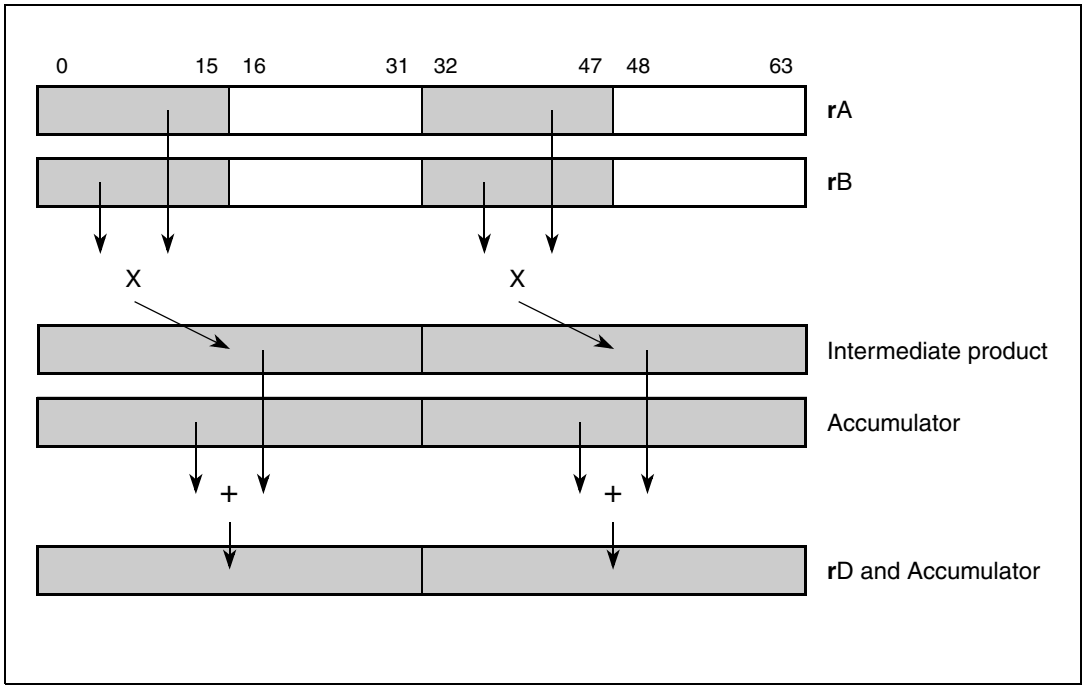
// low
$temp_{0:31} \leftarrow rA_{32:47} \times_{si} rB_{32:47}$
$rD_{32:63} \leftarrow ACC_{32:63} + temp_{0:31}$

// update accumulator
$ACC_{0:63} \leftarrow rD_{0:63}$

For each word element in the accumulator, the corresponding even-numbered half-word signed integer elements in **r**A and **r**B 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 **r**D words and into the accumulator.

Other registers altered: ACC

Figure 80. Even form of vector half-word multiply (evmhesmiaaw)



evmhesmianw | SPE APU | User | **evmhesmianw**

Vector multiply half words, even, signed, modulo, integer and accumulate negative into words

evmhesmianw **r**D,**r**A,**r**B

0	5	6	10	11	15	16	20	21	31
0 0 0 1 0 0		**r**D		**r**A		**r**B		1 0 1 1 0 0 0 1 0 0 1	

// high
$temp0_{0:31} \leftarrow rA_{0:15} \times_{si} rB_{0:15}$
$rD_{0:31} \leftarrow ACC_{0:31} - temp0_{0:31}$

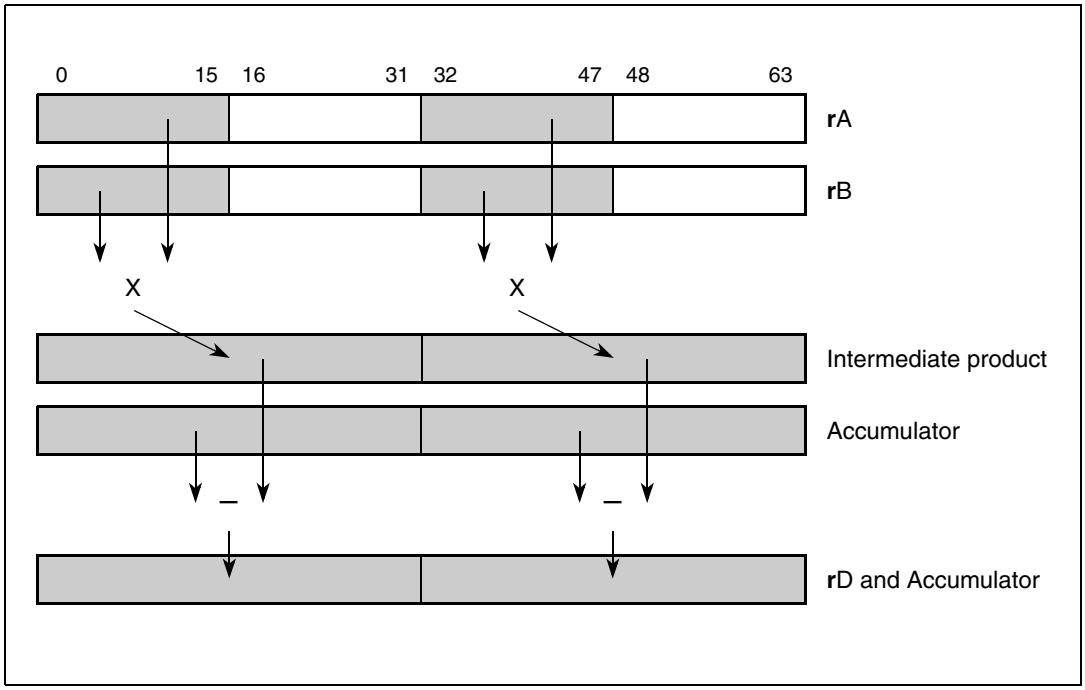
// low
$temp1_{0:31} \leftarrow rA_{32:47} \times_{si} rB_{32:47}$
$rD_{32:63} \leftarrow ACC_{32:63} - temp1_{0:31}$

// update accumulator
$ACC_{0:63} \leftarrow rD_{0:63}$

For each word element in the accumulator, the corresponding even-numbered half-word signed integer elements in **r**A and **r**B 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 **r**D words and into the accumulator.

Other registers altered: ACC

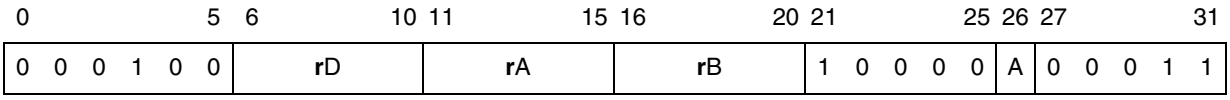
Figure 81. Even form of vector half-word multiply (evmhesmianw)



evmhessf | SPE APU | User | **evmhessf**

Vector multiply half words, even, signed, saturate, fractional (to accumulator)

evmhessf **r**D,**r**A,**r**B **(A = 0)**
evmhessfa **r**D,**r**A,**r**B **(A = 1)**



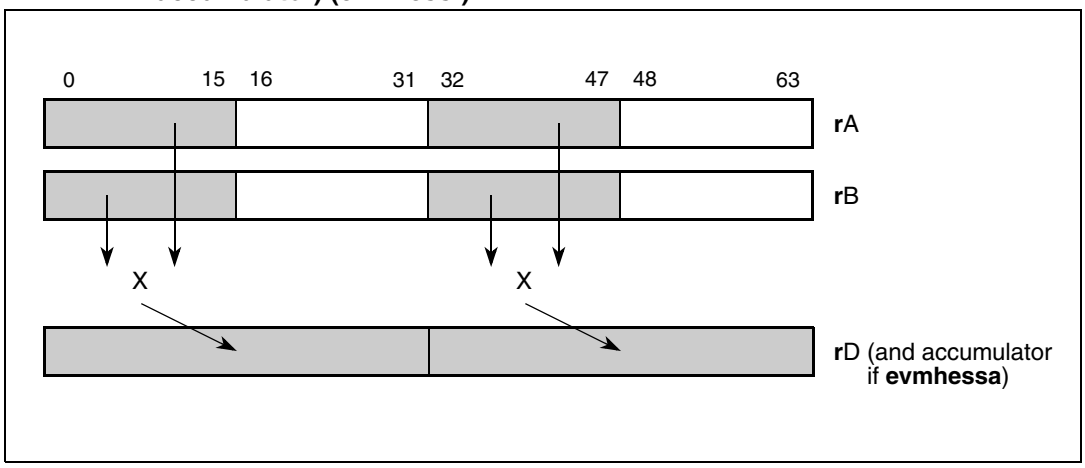

```
// high
temp0:31 ← rA0:15 ×sf rB0:15
if (rA0:15 = 0x8000) & (rB0:15 = 0x8000) then
                rD0:31 ← 0x7FFF_FFFF //saturate
                movh ← 1
else
                rD0:31 ← temp0:31
                movh ← 0

// low
temp0:31 ← rA32:47 ×sf rB32:47
if (rA32:47 = 0x8000) & (rB32:47 = 0x8000) then
                rD32:63 ← 0x7FFF_FFFF //saturate
                movl ← 1
else
                rD32:63 ← temp0:31
                movl ← 0

// update accumulator
if A = 1 then ACC0:63 ← rD0:63

// update SPEFSCR
SPEFSCROVH ← movh
SPEFSCROV ← movl
SPEFSCRSOVH ← SPEFSCRSOVH | movh
SPEFSCRSOV ← SPEFSCRSOV | movl
```

The corresponding even-numbered half-word signed fractional elements in **r**A and **r**B are multiplied. The 32 bits of each product are placed into the corresponding words of **r**D. If both inputs are –1.0, the result saturates to the largest positive signed fraction and the overflow and summary overflow bits are recorded in the SPEFSCR.

If A = 1, the result in **r**D is also placed into the accumulator.

Other registers altered: SPEFSCR
ACC (If A = 1)

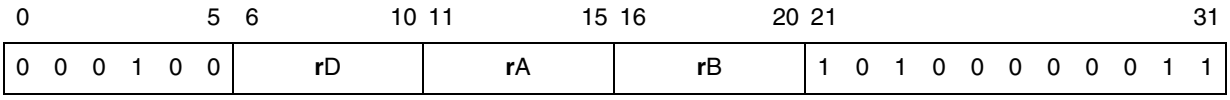
Figure 82. Even multiply of two signed saturate fractional elements (to accumulator) (evmhessf)



evmhessfaaw | SPE APU | User | **evmhessfaaw**

Vector multiply half words, even, signed, saturate, fractional and accumulate into words

evmhessfaaw **r**D,**r**A,**r**B



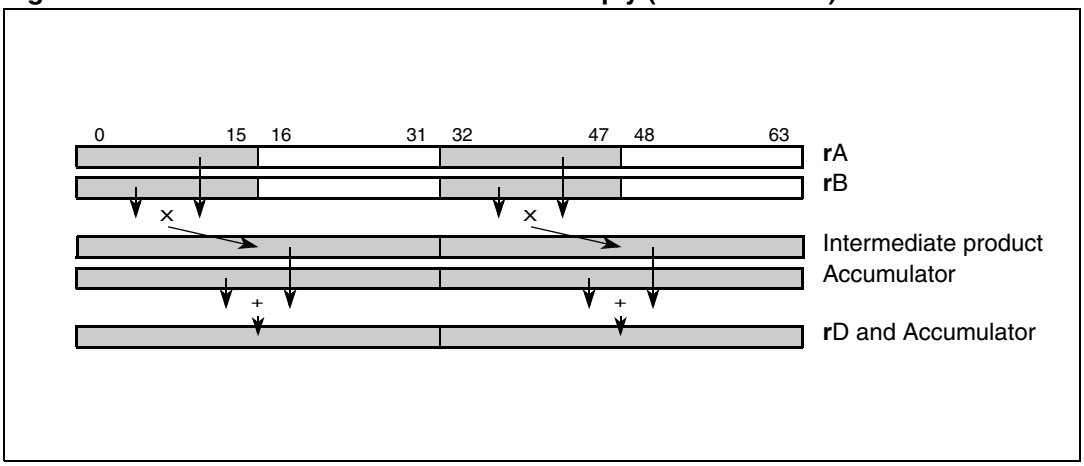

```
// high
temp0:31 ← rA0:15 ×sf rB0:15
if (rA0:15 = 0x8000) & (rB0:15 = 0x8000) then
                temp0:31 ← 0x7FFF_FFFF //saturate
                movh ← 1
else
                movh ← 0
temp0:63 ← EXTS(ACC0:31) + EXTS(temp0:31)
ovh ← (temp31 ⊕ temp32)
rD0:31 ← SATURATE(ovh, temp31, 0x8000_0000, 0x7FFF_FFFF, temp32:63)

// low
temp0:31 ← rA32:47 ×sf rB32:47
if (rA32:47 = 0x8000) & (rB32:47 = 0x8000) then
                temp0:31 ← 0x7FFF_FFFF //saturate
                movl ← 1
else
                movl ← 0
temp0:63 ← EXTS(ACC32:63) + EXTS(temp0:31)
ovl ← (temp31 ⊕ temp32)
rD32:63 ← SATURATE(ovl, temp31, 0x8000_0000, 0x7FFF_FFFF, temp32:63)

// update accumulator
ACC0:63 ← rD0:63

// update SPEFSCR
SPEFSCROVH ← movh
SPEFSCROV ← movl
SPEFSCRSOVH ← SPEFSCRSOVH | ovh | movh
SPEFSCRSOV ← SPEFSCRSOV | ovl| movl
```

The corresponding even-numbered half-word signed fractional elements in **r**A and **r**B 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 or underflow occurs, and the result is placed in **r**D and the accumulator.

If there is an overflow or underflow from either the multiply or the addition, the overflow and summary overflow bits are recorded in the SPEFSCR.

Other registers altered: SPEFSCR ACC

Figure 83. Even form of vector half-word multiply (evmhessfaaw)



evmhessfanw | SPE APU | User | **evmhessfanw**

Vector multiply half words, even, signed, saturate, fractional and accumulate negative into words

evmhessfanw **r**D,**r**A,**r**B

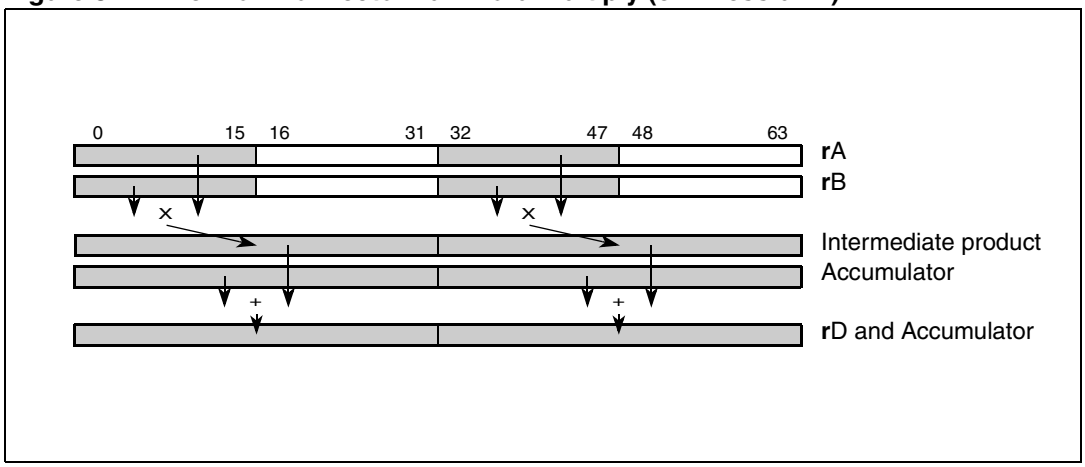
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	0	1	0	0	**r**D					**r**A					**r**B					1	0	1	1	0	0	0	0	0	1	1

++The corresponding even-numbered half-word signed fractional elements in **r**A and **r**B 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 or underflow occurs, and the result is placed in **r**D and the accumulator.

If there is an overflow or underflow from either the multiply or the addition, the overflow and summary overflow bits are recorded in the SPEFSCR.

Other registers altered: SPEFSCR ACC

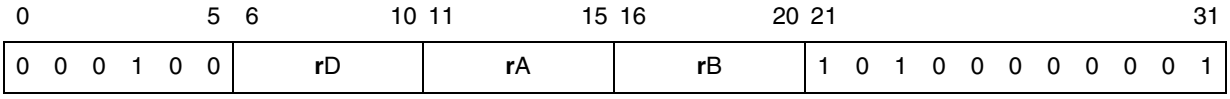
Figure 84. Even form of vector half-word multiply (evmhessfanw)



evmhessiaaw | SPE APU | User | **evmhessiaaw**

Vector multiply half words, even, signed, saturate, integer and accumulate into words

evmhessiaaw **r**D,**r**A,**r**B



// high
$temp_{0:31} \leftarrow rA_{0:15} \times_{si} rB_{0:15}$
$temp_{0:63} \leftarrow EXTS(ACC_{0:31}) + EXTS(temp_{0:31})$
$ovh \leftarrow (temp_{31} \oplus temp_{32})$
$rD_{0:31} \leftarrow SATURATE(ovh, temp_{31, 0x8000\_0000, 0x7FFF\_FFFF, temp32:63})$

// low
$temp_{0:31} \leftarrow rA_{32:47} \times_{si} rB_{32:47}$
$temp_{0:63} \leftarrow EXTS(ACC_{32:63}) + EXTS(temp_{0:31})$
$ovl \leftarrow (temp_{31} \oplus temp_{32})$
$rD_{32:63} \leftarrow SATURATE(ovl, temp_{31, 0x8000\_0000, 0x7FFF\_FFFF, temp32:63})$

// update accumulator
$ACC_{0:63} \leftarrow rD_{0:63}$

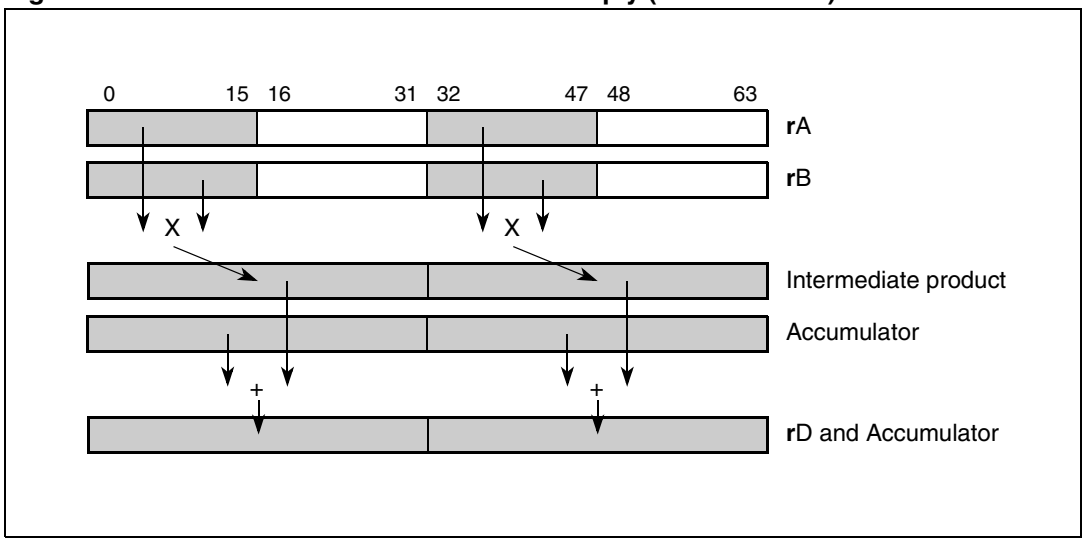
// update SPEFSCR
$SPEFSCR_{OVH} \leftarrow ovh$
$SPEFSCR_{OV} \leftarrow ovl$
$SPEFSCR_{SOVH} \leftarrow SPEFSCR_{SOVH} \mid ovh$
$SPEFSCR_{SOV} \leftarrow SPEFSCR_{SOV} \mid ovl$

The corresponding even-numbered half-word signed integer elements in **r**A and **r**B 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 **r**D and the accumulator.

If there is an overflow or underflow from either the multiply or the addition, the overflow and summary overflow bits are recorded in the SPEFSCR.

Other registers altered: SPEFSCR ACC

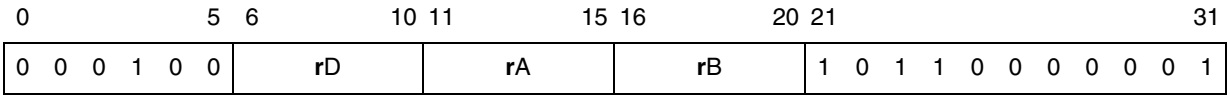
Figure 85. Even form of vector half-word multiply (evmhessiaaw)



evmhessianw | SPE APU | User | **evmhessianw**

Vector multiply half words, even, signed, saturate, integer and accumulate negative into words

evmhessianw **r**D,**r**A,**r**B



// high
$temp_{0:31} \leftarrow rA_{0:15} \times_{si} rB_{0:15}$
$temp_{0:63} \leftarrow EXTS(ACC_{0:31}) - EXTS(temp_{0:31})$
$ovh \leftarrow (temp_{31} \oplus temp_{32})$
$rD_{0:31} \leftarrow SATURATE(ovh, temp_{31, 0x8000\_0000, 0x7FFF\_FFFF, temp32:63})$

// low
$temp_{0:31} \leftarrow rA_{32:47} \times_{si} rB_{32:47}$
$temp_{0:63} \leftarrow EXTS(ACC_{32:63}) - EXTS(temp_{0:31})$
$ovl \leftarrow (temp_{31} \oplus temp_{32})$
$rD_{32:63} \leftarrow SATURATE(ovl, temp_{31, 0x8000\_0000, 0x7FFF\_FFFF, temp32:63})$

// update accumulator
$ACC_{0:63} \leftarrow rD_{0:63}$

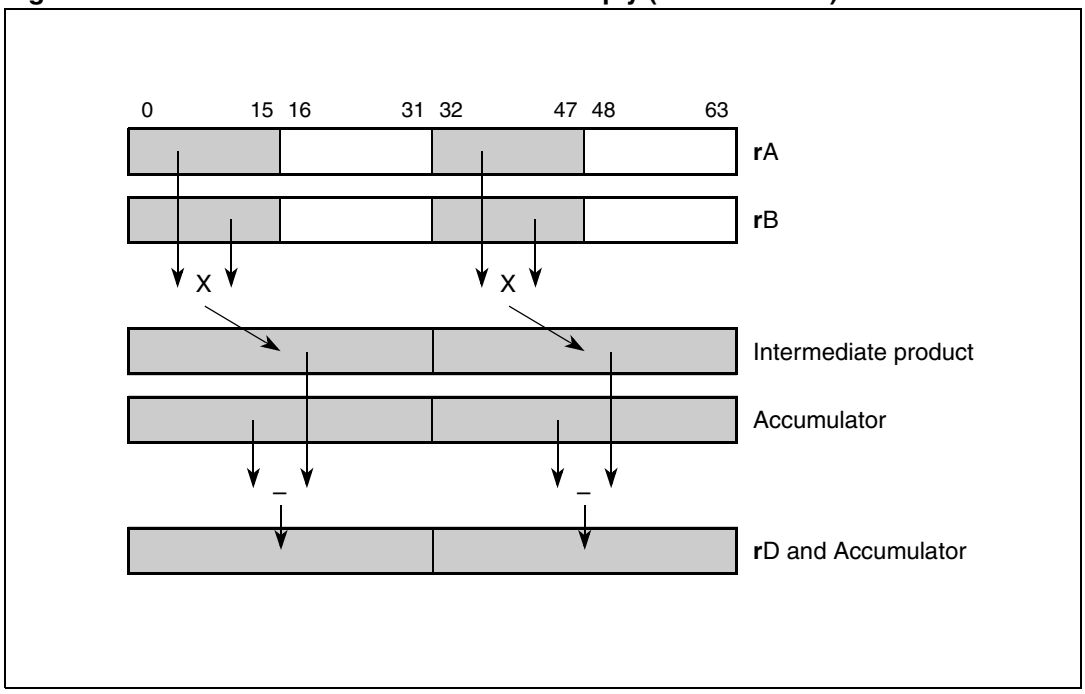
// update SPEFSCR
$SPEFSCR_{OVH} \leftarrow ovh$
$SPEFSCR_{OV} \leftarrow ovl$
$SPEFSCR_{SOVH} \leftarrow SPEFSCR_{SOVH} \mid ovh$
$SPEFSCR_{SOV} \leftarrow SPEFSCR_{SOV} \mid ovl$

The corresponding even-numbered half-word signed integer elements in **r**A and **r**B 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 **r**D and the accumulator.

If there is an overflow or underflow from either the multiply or the addition, the overflow and summary overflow bits are recorded in the SPEFSCR.

Other registers altered: SPEFSCR ACC

Figure 86. Even form of vector half-word multiply (evmhessianw)



evmheumi | SPE APU | User | **evmheumi**

Vector multiply half words, even, unsigned, modulo, integer (to accumulator)

evmheumi **r**D,**r**A,**r**B **(A = 0)**
evmheumia **r**D,**r**A,**r**B **(A = 1)**

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	0	1	0	0	**r**D					**r**A					**r**B					1	0	0	0	0	A	0	1	0	0	0

// high
$rD_{0:31} \leftarrow rA_{0:15} \times_{ui} rB_{0:15}$

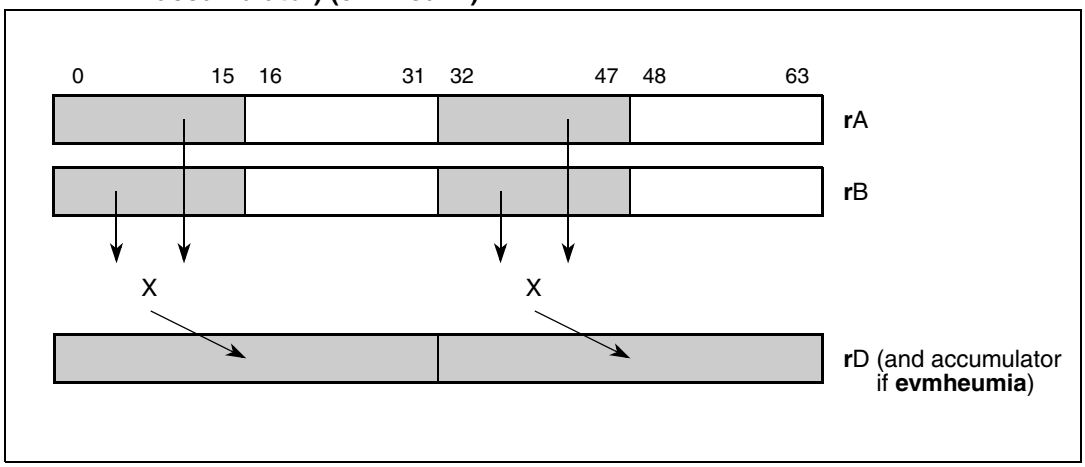
// low
$rD_{32:63} \leftarrow rA_{32:47} \times_{ui} rB_{32:47}$

// update accumulator
if A = 1 then $ACC_{0:63} \leftarrow rD_{0:63}$

The corresponding even-numbered half-word unsigned integer elements in **r**A and **r**B are multiplied. The two 32-bit products are placed into the corresponding words of **r**D.

If A = 1, the result in **r**D is also placed into the accumulator.

Figure 87. Vector multiply half words, even, unsigned, modulo, integer (to accumulator) (evmheumi)



evmheumiaaw | SPE APU | User | **evmheumiaaw**

Vector multiply half words, even, unsigned, modulo, integer and accumulate into words

evmheumiaaw **r**D,**r**A,**r**B

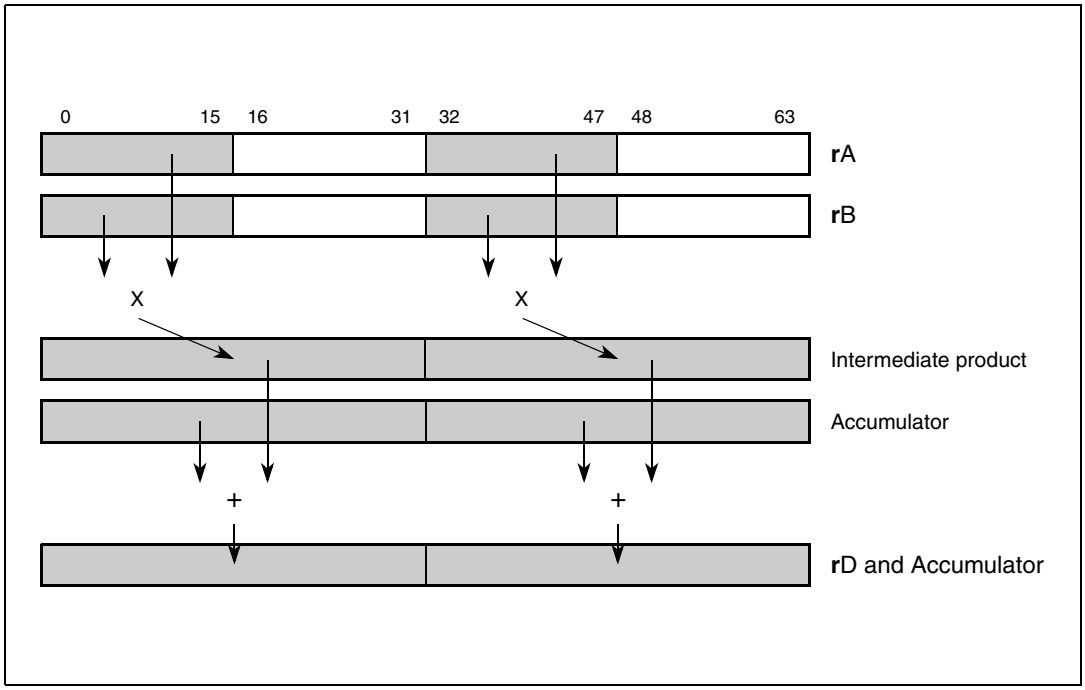
0	5	6	10	11	15	16	20	21	31
0 0 0 1 0 0		**r**D		**r**A		**r**B		1 0 1 0 0 0 0 1 0 0 0	

```
// high
temp0:31 ← rA0:15 ×ui rB0:15
rD0:31 ← ACC0:31 + temp0:31

// low
temp0:31 ← rA32:47 ×ui rB32:47
rD32:63 ← ACC32:63 + temp0:31

// update accumulator
ACC0:63 ← rD0:63
```

For each word element in the accumulator, the corresponding even-numbered half-word unsigned integer elements in **r**A and **r**B are multiplied. Each intermediate product is added to the contents of the corresponding accumulator words and the sums are placed into the corresponding **r**D and accumulator words.

Other registers altered: ACC

Figure 88. Even form of vector half-word multiply (evmheumiaaw)



evmheumianw | SPE APU | User | **evmheumianw**

Vector multiply half words, even, unsigned, modulo, integer and accumulate negative into words

evmheumianw **r**D,**r**A,**r**B

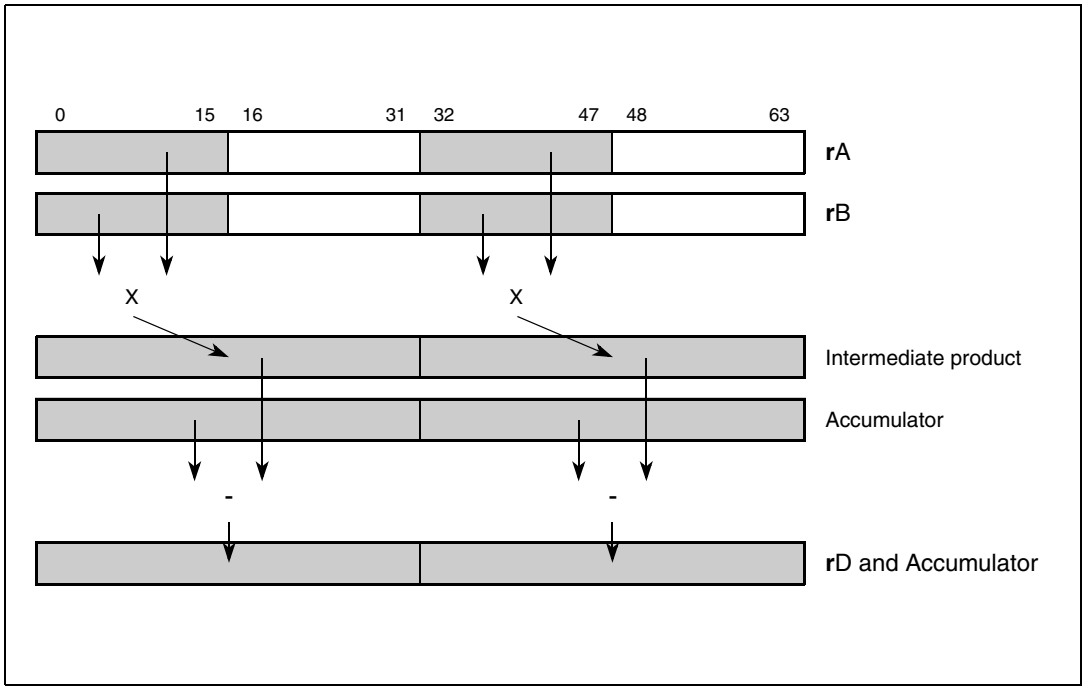
0	5	6	10	11	15	16	20	21	31
0 0 0 1 0 0		**r**D		**r**A		**r**B		1 0 1 1 0 0 0 1 0 0 0	

```
// high
temp0:31 ← rA0:15 ×ui rB0:15
rD0:31 ← ACC0:31 - temp0:31

// low
temp0:31 ← rA32:47 ×ui rB32:47
rD32:63 ← ACC32:63 - temp0:31

// update accumulator
ACC0:63 ← rD0:63
```

For each word element in the accumulator, the corresponding even-numbered half-word unsigned integer elements in **r**A and **r**B are multiplied. Each intermediate product is subtracted from the contents of the corresponding accumulator words. The differences are placed into the corresponding **r**D and accumulator words.

Other registers altered: ACC

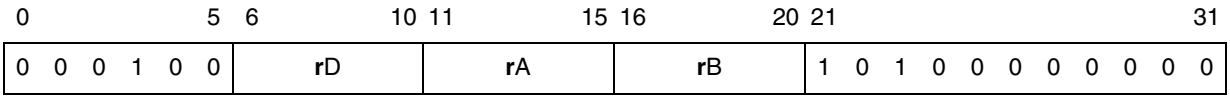
Figure 89. Even form of vector half-word multiply (evmheumianw)



evmheusiaaw | SPE APU | User | **evmheusiaaw**

Vector multiply half words, even, unsigned, saturate, integer and accumulate into words

evmheusiaaw **r**D,**r**A,**r**B



// high
$temp_{0:31} \leftarrow rA_{0:15} \times_{ui} rB_{0:15}$
$temp_{0:63} \leftarrow EXTZ(ACC_{0:31}) + EXTZ(temp_{0:31})$
$ovh \leftarrow temp_{31}$
$rD_{0:31} \leftarrow SATURATE(ovh, 0, 0xFFFF\_FFFF, 0xFFFF\_FFFF, temp_{32:63})$

//low
$temp_{0:31} \leftarrow rA_{32:47} \times_{ui} rB_{32:47}$
$temp_{0:63} \leftarrow EXTZ(ACC_{32:63}) + EXTZ(temp_{0:31})$
$ovl \leftarrow temp_{31}$
$rD_{32:63} \leftarrow SATURATE(ovl, 0, 0xFFFF\_FFFF, 0xFFFF\_FFFF, temp_{32:63})$

// update accumulator
$ACC_{0:63} \leftarrow rD_{0:63}$

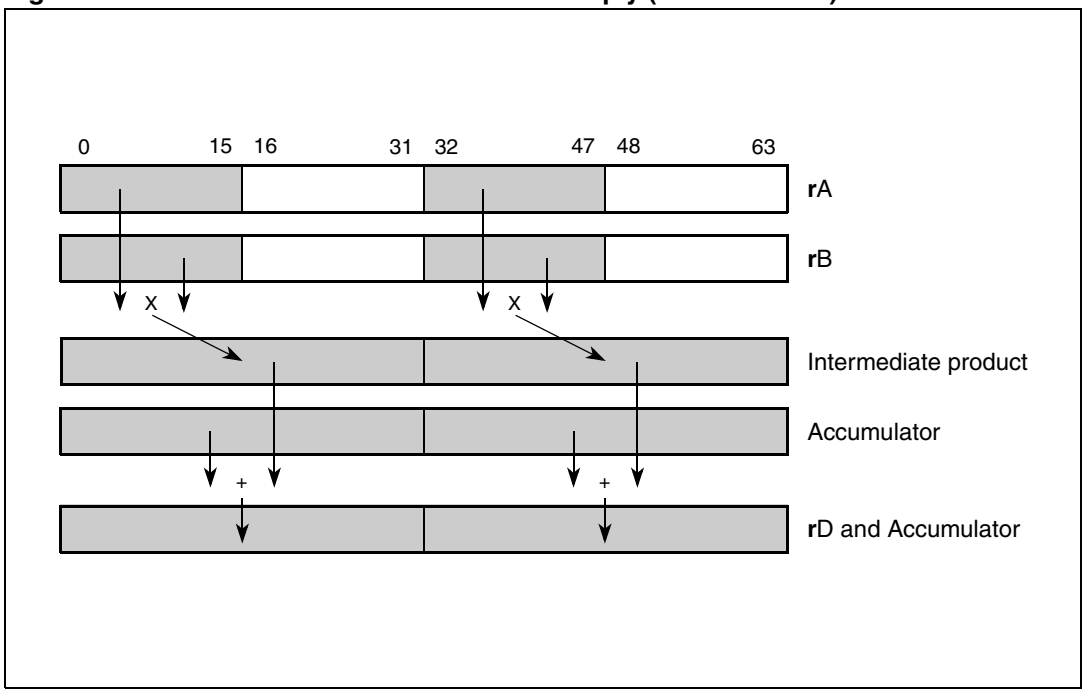
// update SPEFSCR
$SPEFSCR_{OVH} \leftarrow ovh$
$SPEFSCR_{OV} \leftarrow ovl$
$SPEFSCR_{SOVH} \leftarrow SPEFSCR_{SOVH} \mid ovh$
$SPEFSCR_{SOV} \leftarrow SPEFSCR_{SOV} \mid ovl$

For each word element in the accumulator, corresponding even-numbered half-word unsigned integer elements in **r**A and **r**B 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 **r**D and the accumulator.

If the addition causes overflow, the overflow and summary overflow bits are recorded in the SPEFSCR.

Other registers altered: SPEFSCR ACC

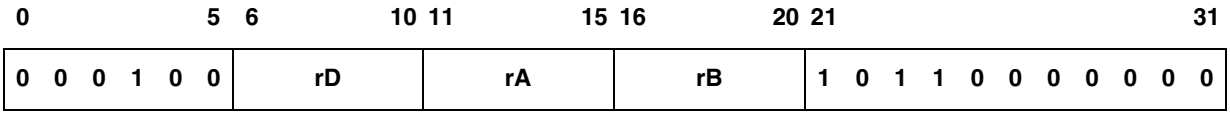
Figure 90. Even form of vector half-word multiply (evmheusiaaw)



evmheusianw | SPE APU | User | **evmheusianw**

Vector multiply half words, even, unsigned, saturate, integer and accumulate negative into words

evmheusianw **r**D,**r**A,**r**B



// high
$temp_{0:31} \leftarrow rA_{0:15} \times_{ui} rB_{0:15}$
$temp_{0:63} \leftarrow EXTZ(ACC_{0:31}) - EXTZ(temp_{0:31})$
$ovh \leftarrow temp_{31}$
$rD_{0:31} \leftarrow SATURATE(ovh, 0, 0x0000\_0000, 0x0000\_0000, temp_{32:63})$

//low
$temp_{0:31} \leftarrow rA_{32:47} \times_{ui} rB_{32:47}$
$temp_{0:63} \leftarrow EXTZ(ACC_{32:63}) - EXTZ(temp_{0:31})$
$ovl \leftarrow temp_{31}$
$rD_{32:63} \leftarrow SATURATE(ovl, 0, 0x0000\_0000, 0x0000\_0000, temp_{32:63})$

// update accumulator
$ACC_{0:63} \leftarrow rD_{0:63}$

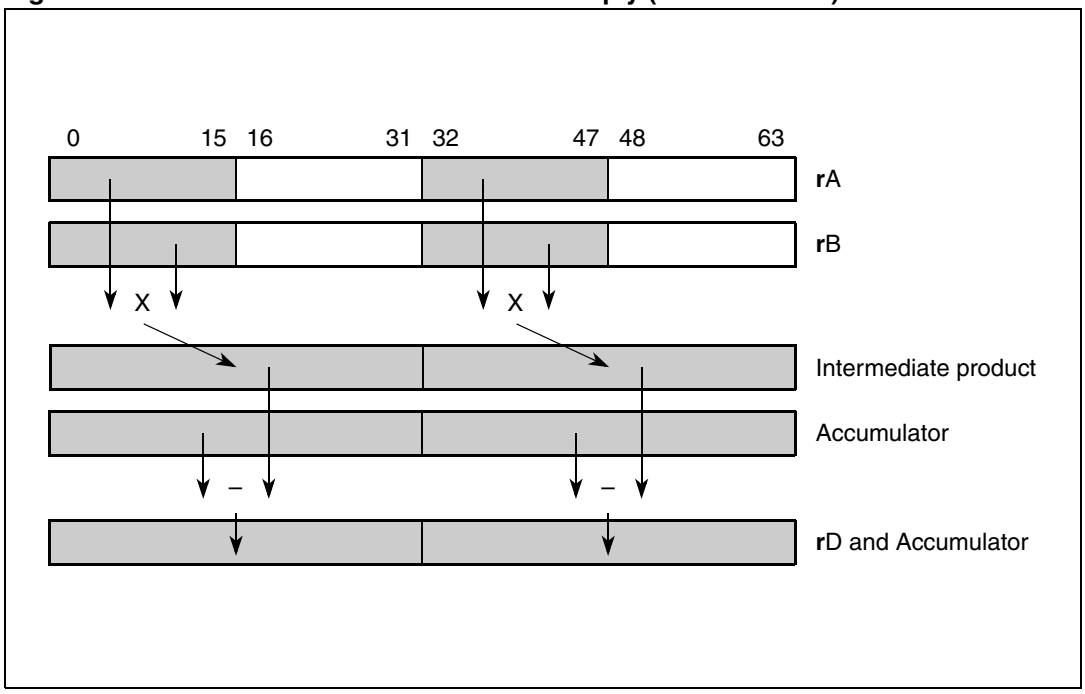
// update SPEFSCR
$SPEFSCR_{OVH} \leftarrow ovh$
$SPEFSCR_{OV} \leftarrow ovl$
$SPEFSCR_{SOVH} \leftarrow SPEFSCR_{SOVH} \mid ovh$
$SPEFSCR_{SOV} \leftarrow SPEFSCR_{SOV} \mid ovl$

For each word element in the accumulator, corresponding even-numbered half-word unsigned integer elements in **r**A and **r**B are multiplied producing a 32-bit product. Each 32-bit product is then subtracted from the corresponding word in the accumulator, saturating if underflow occurs, and the result is placed in **r**D and the accumulator.

If there is an underflow from the subtraction, the overflow and summary overflow bits are recorded in the SPEFSCR.

Other registers altered: SPEFSCR ACC

Figure 91. Even form of vector half-word multiply (evmheusianw)



evmhogsmfaa | SPE APU | User | **evmhogsmfaa**

Vector multiply half words, odd, guarded, signed, modulo, fractional and accumulate

evmhogsmfaa **r**D,**r**A,**r**B

0	5	6	10	11	15	16	20	21	31
0 0 0 1 0 0		rD		rA		rB		1 0 1 0 0 1 0 1 1 1 1	

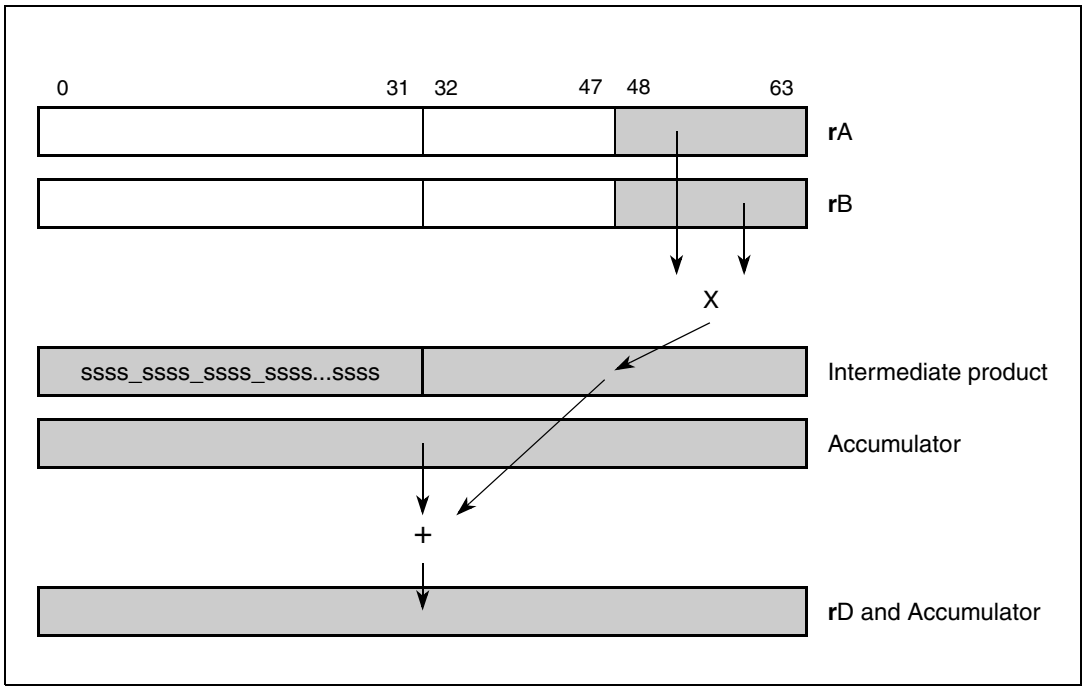
$temp_{0:31} \leftarrow rA_{48:63} \times_{sf} rB_{48:63}$
$temp_{0:63} \leftarrow EXTS(temp_{0:31})$
$rD_{0:63} \leftarrow ACC_{0:63} + temp_{0:63}$

// update accumulator
$ACC_{0:63} \leftarrow rD_{0:63}$

The corresponding low odd-numbered half-word signed fractional elements in **r**A and **r**B 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 into **r**D and into the accumulator.

Note: This is a modulo sum. There is no check for overflow and no saturation is performed. An overflow from the 64-bit sum, if one occurs, is not recorded into the SPEFSCR.

Figure 92. evmhogsmfaa (odd form)



evmhogsmfan | SPE APU | User | **evmhogsmfan**

Vector multiply half words, odd, guarded, signed, modulo, fractional and accumulate negative

evmhogsmfan **r**D,**r**A,**r**B

0	5	6	10	11	15	16	20	21	31
0 0 0 1 0 0		**r**D		**r**A		**r**B		1 0 1 1 0 1 0 1 1 1 1	

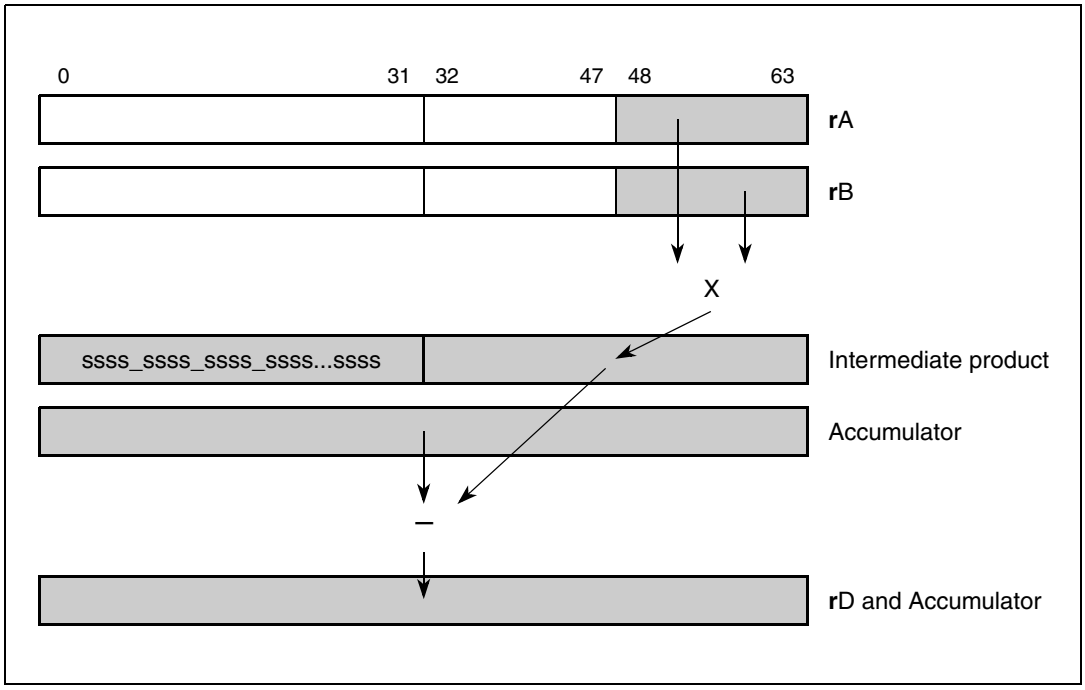
$temp_{0:31} \leftarrow rA_{48:63} \times_{sf} rB_{48:63}$
$temp_{0:63} \leftarrow EXTS(temp_{0:31})$
$rD_{0:63} \leftarrow ACC_{0:63} - temp_{0:63}$

// update accumulator
$ACC_{0:63} \leftarrow rD_{0:63}$

The corresponding low odd-numbered half-word signed fractional elements in **r**A and **r**B 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 into **r**D and into the accumulator.

Note: This is a modulo difference. There is no check for overflow and no saturation is performed. Any overflow of the 64-bit difference is not recorded into the SPEFSCR.

Figure 93. evmhogsmfan (odd form)



evmhogsmiaa | SPE APU | User | **evmhogsmiaa**

Vector multiply half words, odd, guarded, signed, modulo, integer, and accumulate

evmhogsmiaa **r**D,**r**A,**r**B

0	5	6	10	11	15	16	20	21	31
0 0 0 1 0 0		**r**D		**r**A		**r**B		1 0 1 0 0 1 0 1 1 0 1	

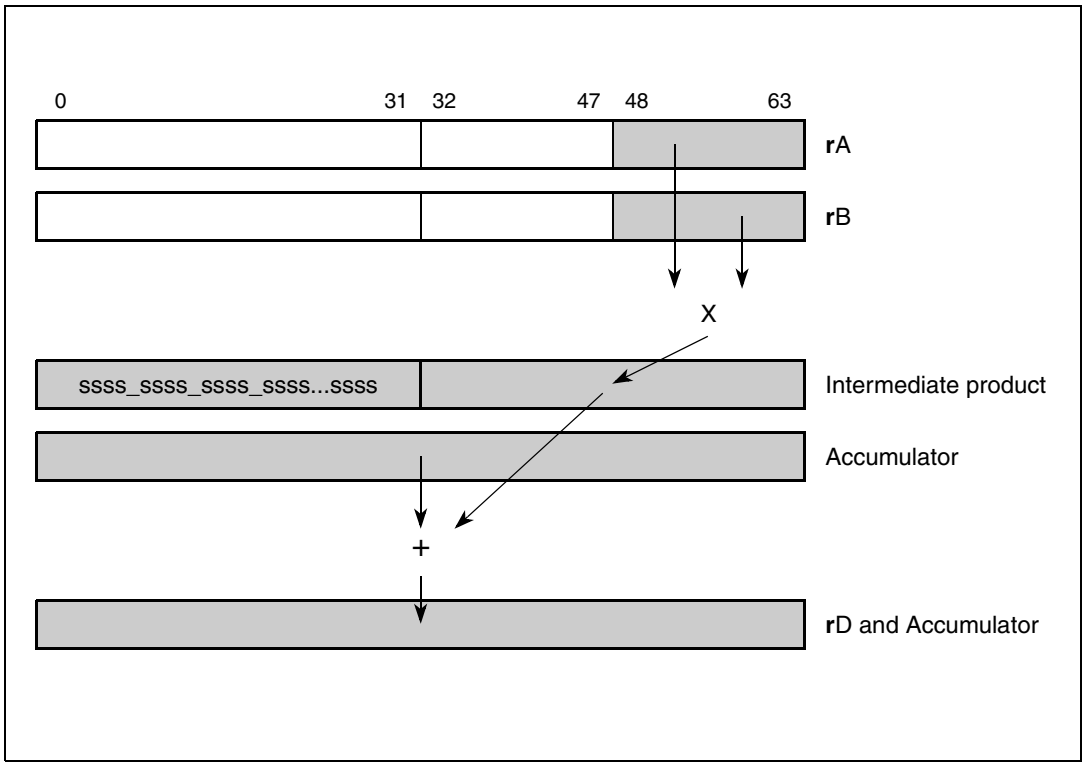
$temp_{0:31} \leftarrow rA_{48:63} \times_{si} rB_{48:63}$
$temp_{0:63} \leftarrow EXTS(temp_{0:31})$
$rD_{0:63} \leftarrow ACC_{0:63} + temp_{0:63}$

// update accumulator
$ACC_{0:63} \leftarrow rD_{0:63}$

The corresponding low odd-numbered half-word signed integer elements in **r**A and **r**B 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 into **r**D and into the accumulator.

Note: This is a modulo sum. There is no check for overflow and no saturation is performed. An overflow from the 64-bit sum, if one occurs, is not recorded into the SPEFSCR.

Figure 94. evmhogsmiaa (odd form)



evmhogsmian | SPE APU | User | **evmhogsmian**

Vector multiply half words, odd, guarded, signed, modulo, integer and accumulate negative

evmhogsmian **r**D,**r**A,**r**B

0	5	6	10	11	15	16	20	21	31
0 0 0 1 0 0		**rD**		**rA**		**rB**		1 0 1 1 0 1 0 1 1 0 1	

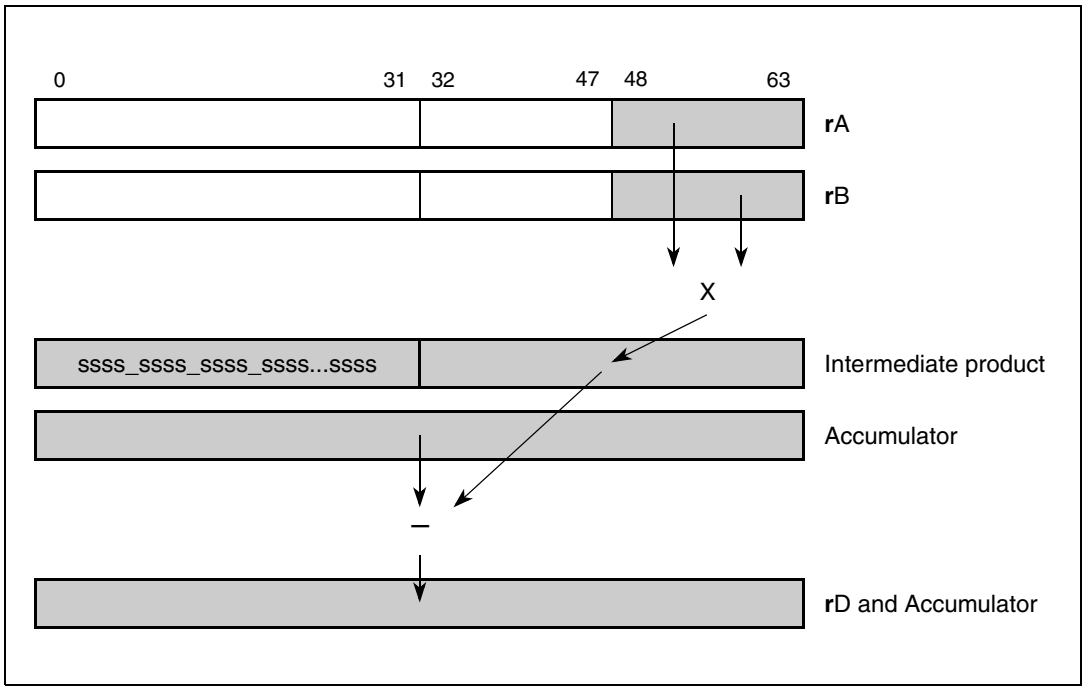
$temp_{0:31} \leftarrow rA_{48:63} \times_{si} rB_{48:63}$
$temp_{0:63} \leftarrow EXTS(temp_{0:31})$
$rD_{0:63} \leftarrow ACC_{0:63} - temp_{0:63}$

// update accumulator
$ACC_{0:63} \leftarrow rD_{0:63}$

The corresponding low odd-numbered half-word signed integer elements in **r**A and **r**B 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 into **r**D and into the accumulator.

Note: This is a modulo difference. There is no check for overflow and no saturation is performed. Any overflow of the 64-bit difference is not recorded into the SPEFSCR.

Figure 95. evmhogsmian (odd form)



evmhogumiaa | SPE APU | User | **evmhogumiaa**

Vector multiply half words, odd, guarded, unsigned, modulo, integer and accumulate

evmhogumiaa **r**D,**r**A,**r**B

0	5	6	10	11	15	16	20	21	31
0 0 0 1 0 0		rD		rA		rB		1 0 1 0 0 1 0 1 1 0 0	

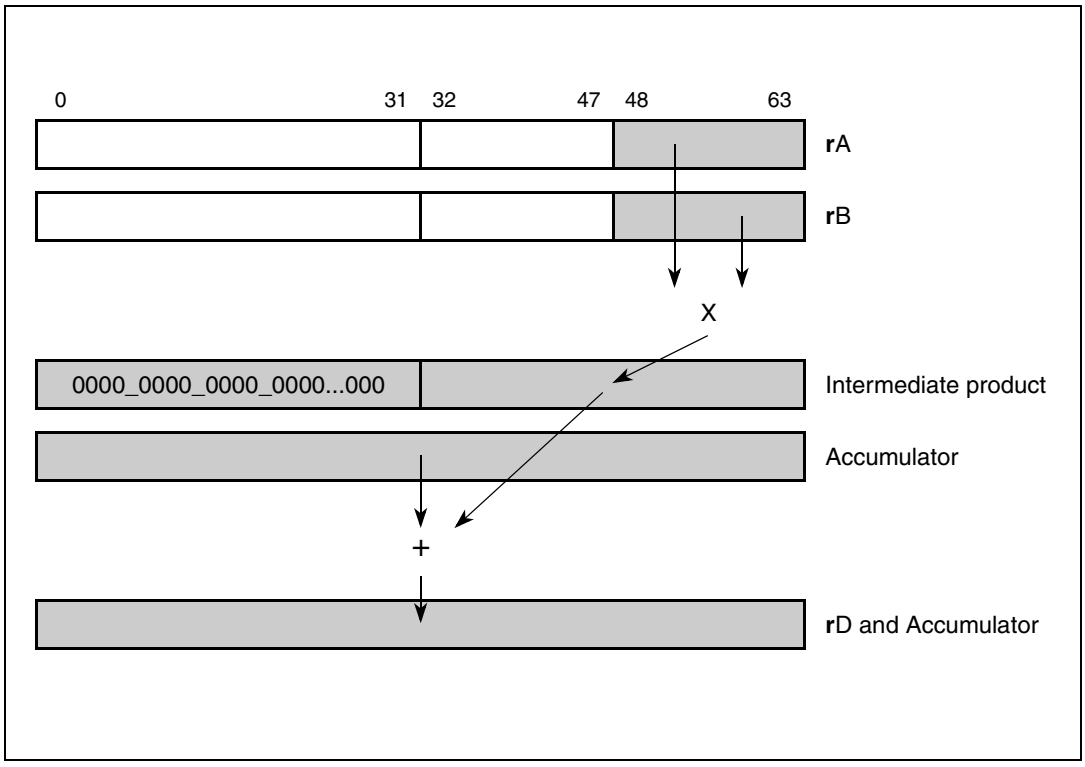
$temp_{0:31} \leftarrow rA_{48:63} \times_{ui} rB_{48:63}$
$temp_{0:63} \leftarrow EXTZ(temp_{0:31})$
$rD_{0:63} \leftarrow ACC_{0:63} + temp_{0:63}$

// update accumulator
$ACC_{0:63} \leftarrow rD_{0:63}$

The corresponding low odd-numbered half-word unsigned integer elements in **r**A and **r**B 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 into **r**D and into the accumulator.

Note: This is a modulo sum. There is no check for overflow and no saturation is performed. An overflow from the 64-bit sum, if one occurs, is not recorded into the SPEFSCR.

Figure 96. evmhogumiaa (odd form)



evmhogumian | SPE APU | User | **evmhogumian**

Vector multiply half words, odd, guarded, unsigned, modulo, integer and accumulate negative

evmhogumian **r**D,**r**A,**r**B

0	5	6	10	11	15	16	20	21	31
0 0 0 1 0 0		rD		rA		rB		1 0 1 1 0 1 0 1 1 0 0	

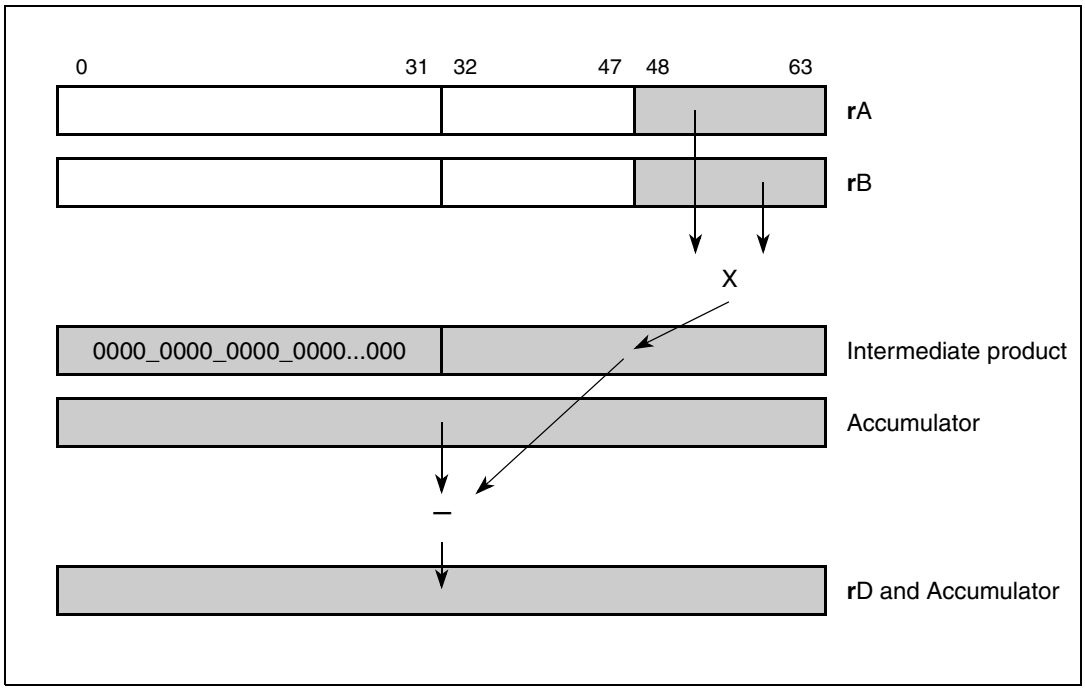
$temp_{0:31} \leftarrow rA_{48:63} \times_{ui} rB_{48:63}$
$temp_{0:63} \leftarrow EXTZ(temp_{0:31})$
$rD_{0:63} \leftarrow ACC_{0:63} - temp_{0:63}$

// update accumulator
$ACC_{0:63} \leftarrow rD_{0:63}$

The corresponding low odd-numbered half-word unsigned integer elements in **r**A and **r**B 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 into **r**D and into the accumulator.

Note: This is a modulo difference. There is no check for overflow and no saturation is performed. Any overflow of the 64-bit difference is not recorded into the SPEFSCR.

Figure 97. evmhogumian (odd form)



evmhosmf | SPE APU | User | **evmhosmf**

Vector multiply half words, odd, signed, modulo, fractional (to accumulator)

evmhosmf **r**D,**r**A,**r**B **(A = 0)**
evmhosmfa **r**D,**r**A,**r**B **(A = 1)**

0	5	6	10	11	15	16	20	21	25	26	27	31
0 0 0 1 0 0		**r**D		**r**A		**r**B		1 0 0 0 0		A	0 1 1 1 1	

// high
$rD_{0:31} \leftarrow (rA_{16:31} \times_{sf} rB_{16:31})$

// low
$rD_{32:63} \leftarrow (rA_{48:63} \times_{sf} rB_{48:63})$

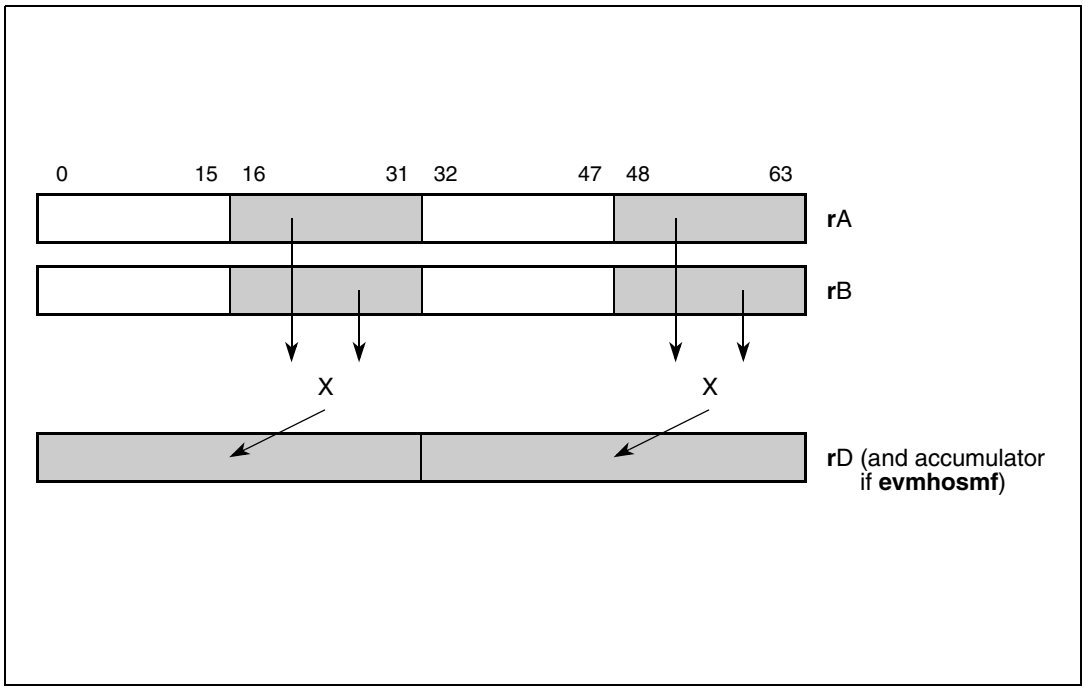
// update accumulator
if A = 1 then $ACC_{0:63} \leftarrow rD_{0:63}$

The corresponding odd-numbered, half-word signed fractional elements in **r**A and **r**B are multiplied. Each product is placed into the corresponding words of **r**D.

If A = 1, the result in **r**D is also placed into the accumulator.

Other registers altered: ACC (If A = 1)

Figure 98. Vector multiply half words, odd, signed, modulo, fractional (to accumulator) (evmhosmf)



evmhosmfaaw | SPE APU | User | **evmhosmfaaw**

Vector multiply half words, odd, signed, modulo, fractional and accumulate into words

evmhosmfaaw **r**D,**r**A,**r**B

0	5	6	10	11	15	16	20	21	31
0 0 0 1 0 0		**r**D		**r**A		**r**B		1 0 1 0 0 0 0 1 1 1 1	

// high
$temp_{0:31} \leftarrow rA_{16:31} \times_{sf} rB_{16:31}$
$rD_{0:31} \leftarrow ACC_{0:31} + temp_{0:31}$

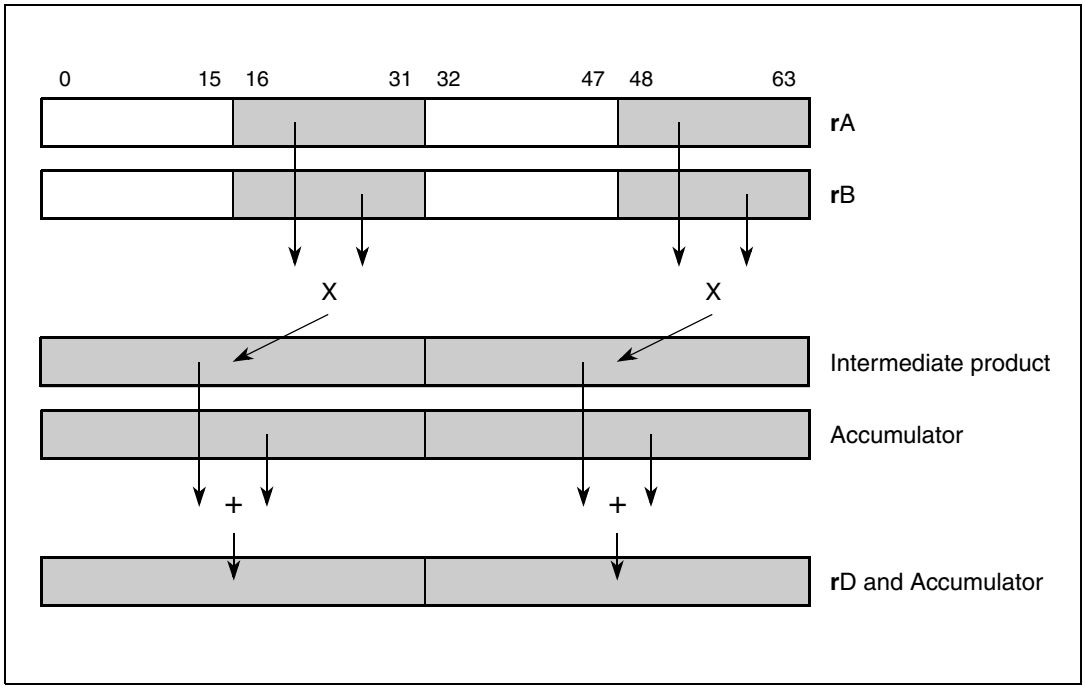
// low
$temp_{0:31} \leftarrow rA_{48:63} \times_{sf} rB_{48:63}$
$rD_{32:63} \leftarrow ACC_{32:63} + temp_{0:31}$

// update accumulator
$ACC_{0:63} \leftarrow rD_{0:63}$

For each word element in the accumulator, the corresponding odd-numbered half-word signed fractional elements in **r**A and **r**B are multiplied. The 32 bits of each intermediate product is added to the contents of the corresponding accumulator word and the results are placed into the corresponding **r**D words and into the accumulator

Other registers altered: ACC

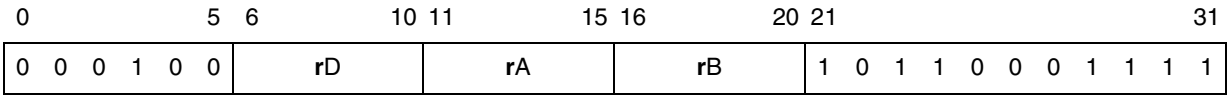
Figure 99. Odd form of vector half-word multiply (evmhosmfaaw)



evmhosmfanw | SPE APU | User | **evmhosmfanw**

Vector multiply half words, odd, signed, modulo, fractional and accumulate negative into words

evmhosmfanw **r**D,**r**A,**r**B



// high
$temp_{0:31} \leftarrow rA_{16:31} \times_{sf} rB_{16:31}$
$rD_{0:31} \leftarrow ACC_{0:31} - temp_{0:31}$

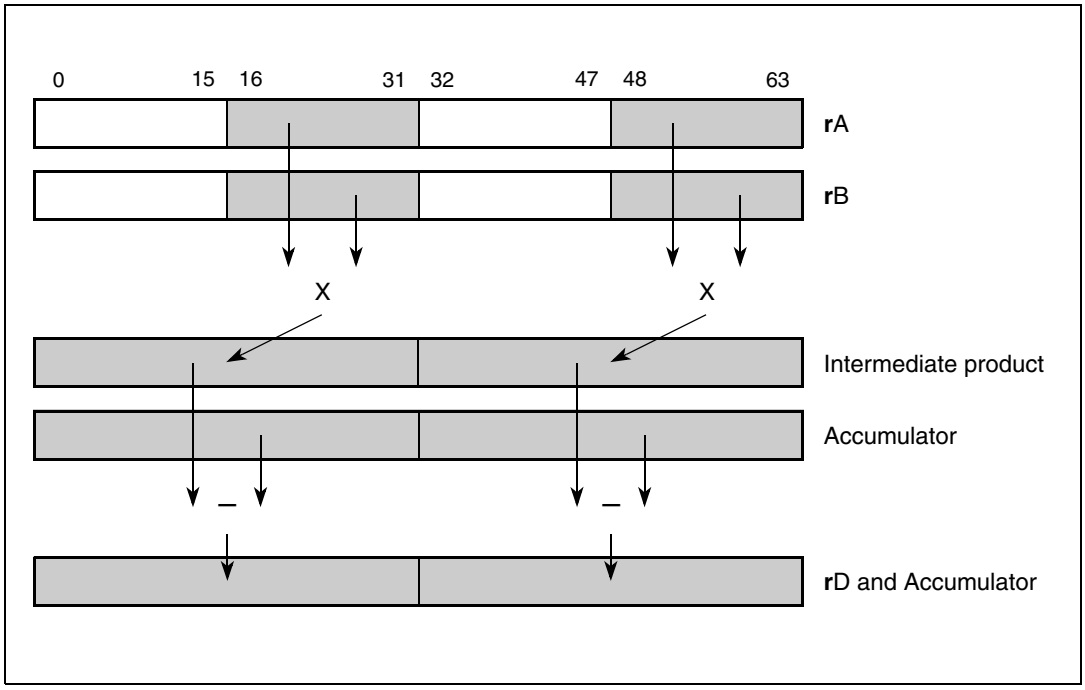
// low
$temp_{0:31} \leftarrow rA_{48:63} \times_{sf} rB_{48:63}$
$rD_{32:63} \leftarrow ACC_{32:63} - temp_{0:31}$

// update accumulator
$ACC_{0:63} \leftarrow rD_{0:63}$

For each word element in the accumulator, the corresponding odd-numbered half-word signed fractional elements in **r**A and **r**B are multiplied. The 32 bits of each intermediate product is subtracted from the contents of the corresponding accumulator word and the results are placed into the corresponding **r**D words and into the accumulator.

Other registers altered: ACC

Figure 100. Odd form of vector half-word multiply (evmhosmfanw)



evmhosmi | SPE APU | User | **evmhosmi**

Vector multiply half words, odd, signed, modulo, integer (to accumulator)

evmhosmi	**r**D,**r**A,**r**B	(A = 0)
evmhosmia	**r**D,**r**A,**r**B	(A = 1)

0	5	6	10	11	15	16	20	21	25	26	27	31
0 0 0 1 0 0		**r**D		**r**A		**r**B		1 0 0 0 0		A	0 1 1 0 1	

// high
$rD_{0:31} \leftarrow rA_{16:31} \times_{si} rB_{16:31}$

// low
$rD_{32:63} \leftarrow rA_{48:63} \times_{si} rB_{48:63}$

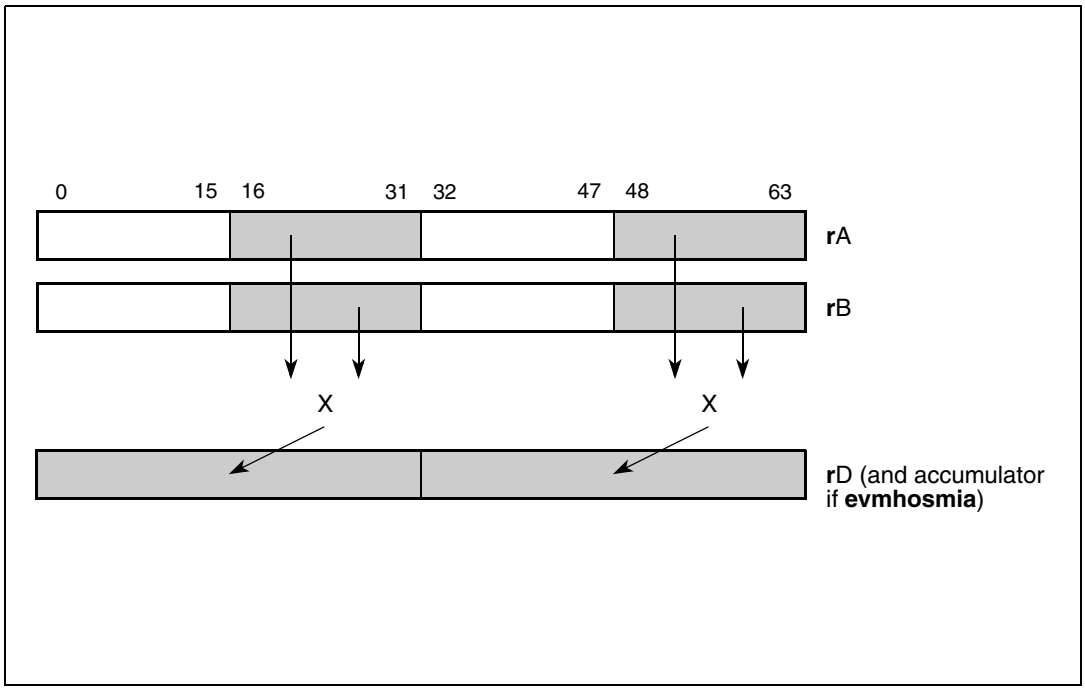
// update accumulator
if A = 1 then $ACC_{0:63} \leftarrow rD_{0:63}$

The corresponding odd-numbered half-word signed integer elements in **r**A and **r**B are multiplied. The two 32-bit products are placed into the corresponding words of **r**D.

If A = 1, the result in **r**D is also placed into the accumulator.

Other registers altered: ACC (If A = 1)

Figure 101. Vector multiply half words, odd, signed, modulo, integer (to accumulator) (evmhosmi)



evmhosmiaaw | SPE APU | User | **evmhosmiaaw**

Vector multiply half words, odd, signed, modulo, integer and accumulate into words

evmhosmiaaw **r**D,**r**A,**r**B

0	5	6	10	11	15	16	20	21	31
0 0 0 1 0 0		**r**D		**r**A		**r**B		1 0 1 0 0 0 0 1 1 0 1	

// high
$temp_{0:31} \leftarrow rA_{16:31} \times_{si} rB_{16:31}$
$rD_{0:31} \leftarrow ACC_{0:31} + temp_{0:31}$

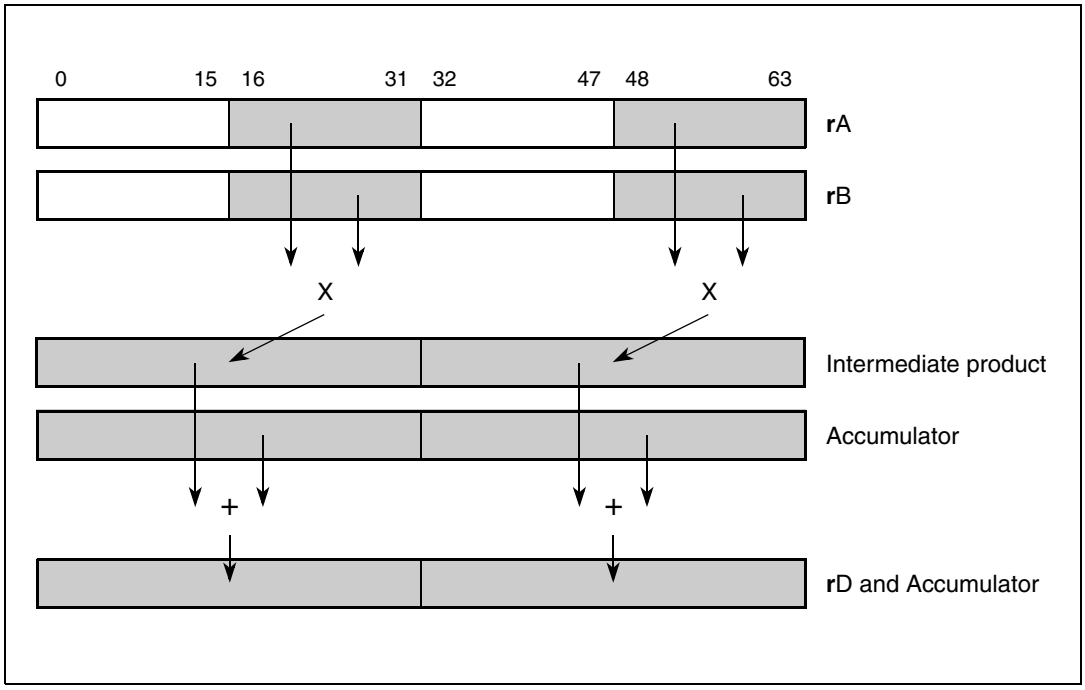
// low
$temp_{0:31} \leftarrow rA_{48:63} \times_{si} rB_{48:63}$
$rD_{32:63} \leftarrow ACC_{32:63} + temp_{0:31}$

// update accumulator
$ACC_{0:63} \leftarrow rD_{0:63}$

For each word element in the accumulator, the corresponding odd-numbered half-word signed integer elements in **r**A and **r**B 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 **r**D words and into the accumulator.

Other registers altered: ACC

Figure 102. Odd form of vector half-word multiply (evmhosmiaaw)



evmhosmianw | SPE APU | User | **evmhosmianw**

Vector multiply half words, odd, signed, modulo, integer and accumulate negative into words

evmhosmianw **r**D,**r**A,**r**B

0	5	6	10	11	15	16	20	21	31
0 0 0 1 0 0		**r**D		**r**A		**r**B		1 0 1 1 0 0 0 1 1 0 1	

// high
$temp_{0:31} \leftarrow rA_{16:31} \times_{si} rB_{16:31}$
$rD_{0:31} \leftarrow ACC_{0:31} - temp_{0:31}$

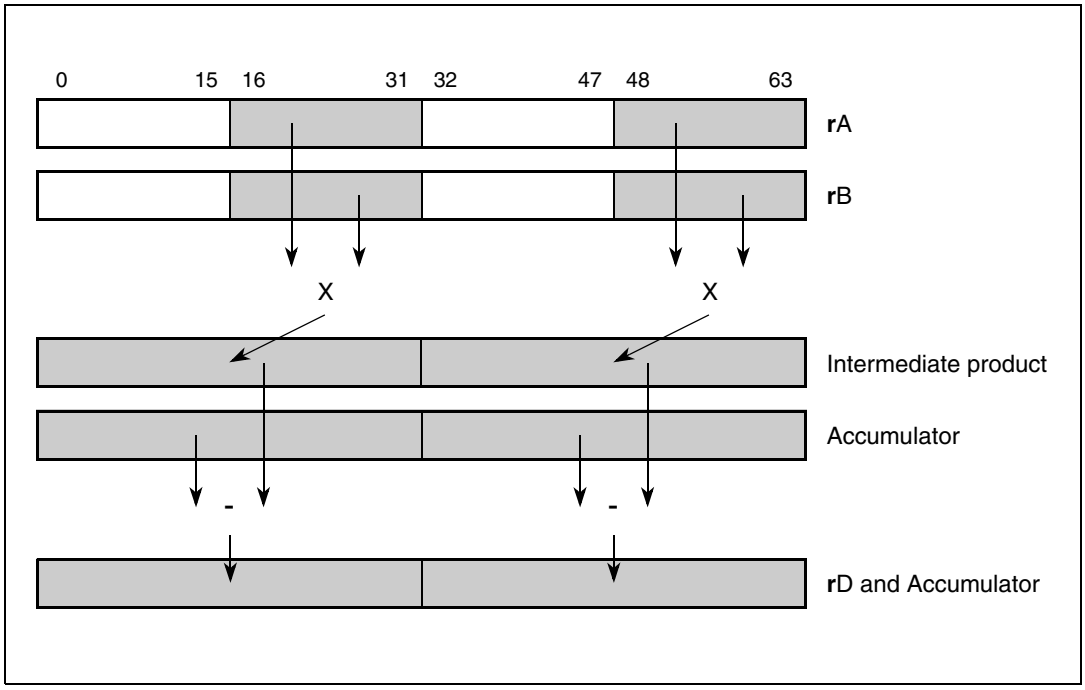
// low
$temp_{0:31} \leftarrow rA_{48:63} \times_{si} rB_{48:63}$
$rD_{32:63} \leftarrow ACC_{32:63} - temp_{0:31}$

// update accumulator
$ACC_{0:63} \leftarrow rD_{0:63}$

For each word element in the accumulator, the corresponding odd-numbered half-word signed integer elements in **r**A and **r**B 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 **r**D words and into the accumulator.

Other registers altered: ACC

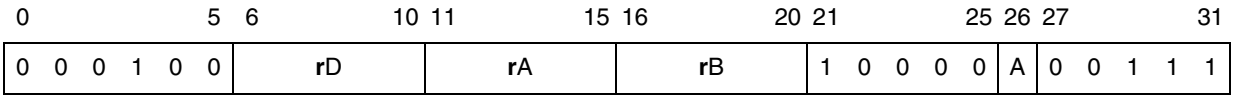
Figure 103. Odd form of vector half-word multiply (evmhosmianw)



evmhossf | SPE APU | User | **evmhossf**

Vector multiply half words, odd, signed, saturate, fractional (to accumulator)

evmhossf **r**D,**r**A,**r**B **(A = 0)**
evmhossfa **r**D,**r**A,**r**B **(A = 1)**



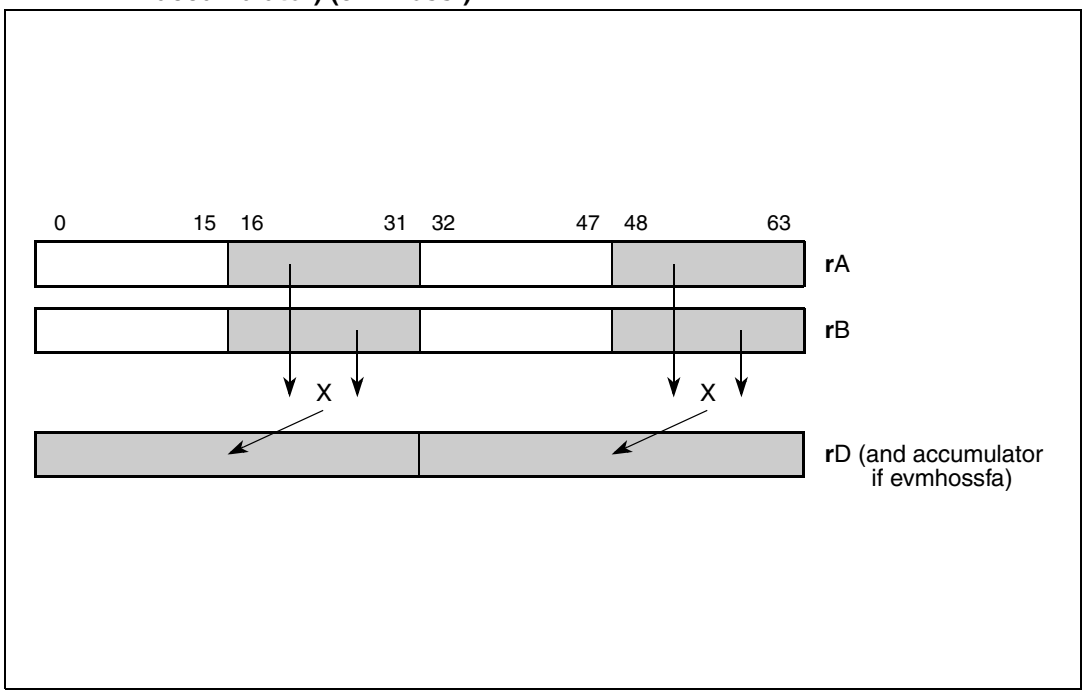

```
// high
temp0:31 ← rA16:31 ×sf rB16:31
if (rA16:31 = 0x8000) & (rB16:31 = 0x8000) then
                rD0:31 ← 0x7FFF_FFFF //saturate
                movh ← 1
else
                rD0:31 ← temp0:31
                movh ← 0

// low
temp0:31 ← rA48:63 ×sf rB48:63
if (rA48:63 = 0x8000) & (rB48:63 = 0x8000) then
                rD32:63 ← 0x7FFF_FFFF //saturate
                movl ← 1
else
                rD32:63 ← temp0:31
                movl ← 0

// update accumulator
if A = 1 then ACC0:63 ← rD0:63

// update SPEFSCR
SPEFSCROVH ← movh
SPEFSCROV ← movl
SPEFSCRSOVH ← SPEFSCRSOVH | movh
SPEFSCRSOV ← SPEFSCRSOV | movl
```

The corresponding odd-numbered half-word signed fractional elements in **r**A and **r**B are multiplied. The 32 bits of each product are placed into the corresponding words of **r**D. If both inputs are –1.0, the result saturates to the largest positive signed fraction and the overflow and summary overflow bits are recorded in the SPEFSCR.

If A = 1, the result in **r**D is also placed into the accumulator.

Other registers altered: SPEFSCR
ACC (If A = 1)

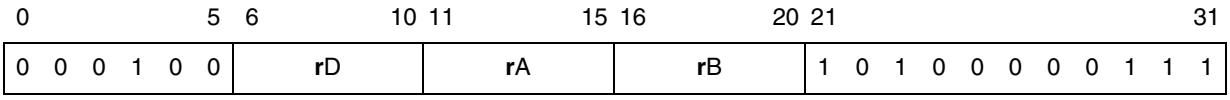
Figure 104. Vector multiply half words, odd, signed, saturate, fractional (to accumulator) (evmhossf)



evmhossfaaw | SPE APU | User | **evmhossfaaw**

Vector multiply half words, odd, signed, saturate, fractional and accumulate into words

evmhossfaaw **r**D,**r**A,**r**B



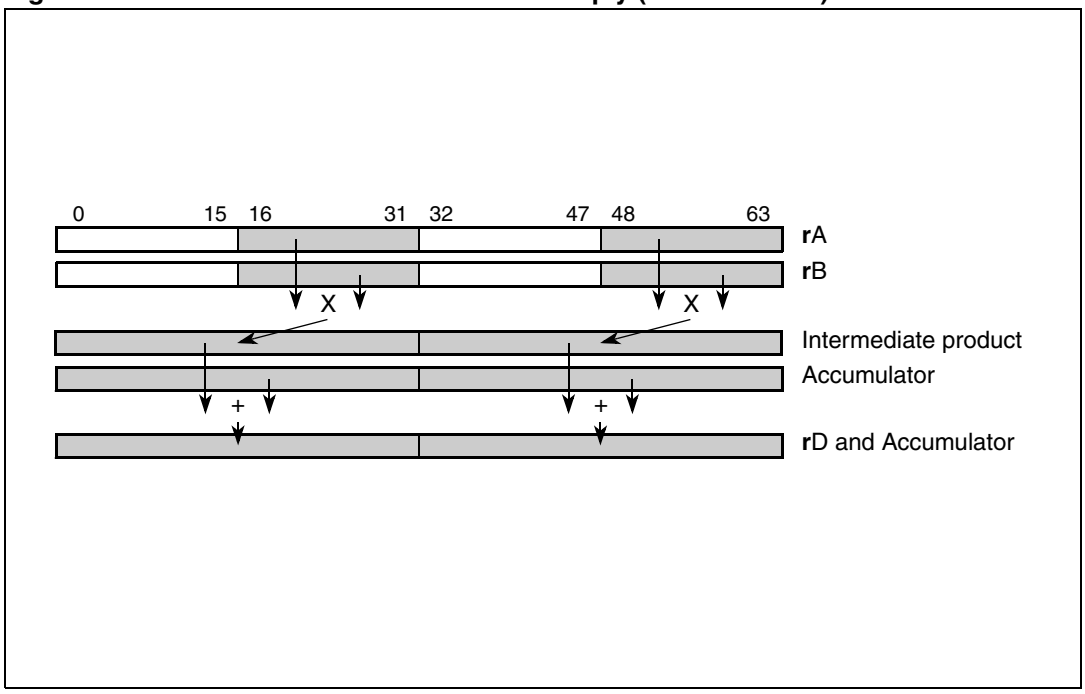

```
// high
temp0:31 ← rA16:31 ×sf rB16:31
if (rA16:31 = 0x8000) & (rB16:31 = 0x8000) then
                temp0:31 ← 0x7FFF_FFFF //saturate
                movh ← 1
else
                movh ← 0
temp0:63 ← EXTS(ACC0:31) + EXTS(temp0:31)
ovh ← (temp31 ⊕ temp32)
rD0:31 ← SATURATE(ovh, temp31, 0x8000_0000, 0x7FFF_FFFF, temp32:63)

// low
temp0:31 ← rA48:63 ×sf rB48:63
if (rA48:63 = 0x8000) & (rB48:63 = 0x8000) then
                temp0:31 ← 0x7FFF_FFFF //saturate
                movl ← 1
else
                movl ← 0
temp0:63 ← EXTS(ACC32:63) + EXTS(temp0:31)
ovl ← (temp31 ⊕ temp32)
rD32:63 ← SATURATE(ovl, temp31, 0x8000_0000, 0x7FFF_FFFF, temp32:63)

// update accumulator
ACC0:63 ← rD0:63

// update SPEFSCR
SPEFSCROVH ← movh
SPEFSCROV ← movl
SPEFSCRSOVH ← SPEFSCRSOVH | ovh | movh
SPEFSCRSOV ← SPEFSCRSOV | ovl| movl
```

The corresponding odd-numbered half-word signed fractional elements in **r**A and **r**B 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 or underflow occurs, and the result is placed in **r**D and the accumulator.

If there is an overflow or underflow from either the multiply or the addition, the overflow and summary overflow bits are recorded in the SPEFSCR.

Other registers altered: SPEFSCR ACC

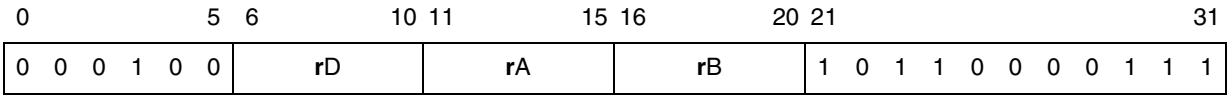
Figure 105. Odd form of vector half-word multiply (evmhossfaaw)



evmhossfanw | SPE APU | User | **evmhossfanw**

Vector multiply half words, odd, signed, saturate, fractional and accumulate negative into words

evmhossfanw **r**D,**r**A,**r**B



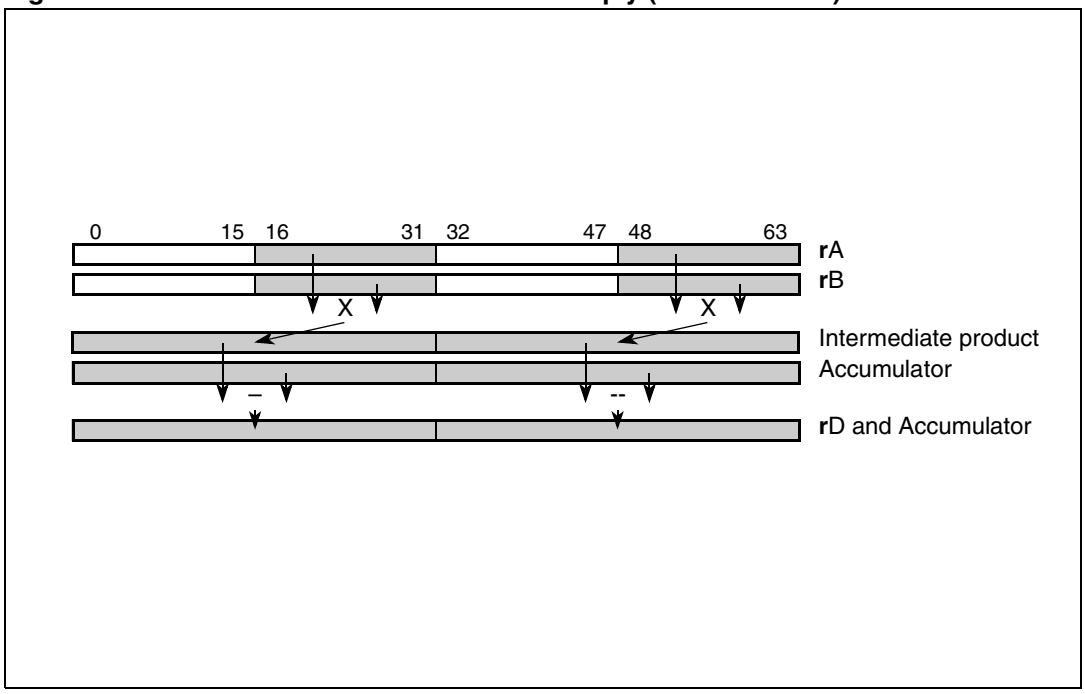

```
// high
temp0:31 ← rA16:31 ×sf rB16:31
if (rA16:31 = 0x8000) & (rB16:31 = 0x8000) then
                temp0:31 ← 0x7FFF_FFFF //saturate
                movh ← 1
else
                movh ← 0
temp0:63 ← EXTS(ACC0:31) - EXTS(temp0:31)
ovh ← (temp31 ⊕ temp32)
rD0:31 ← SATURATE(ovh, temp31, 0x8000_0000, 0x7FFF_FFFF, temp32:63)

// low
temp0:31 ← rA48:63 ×sf rB48:63
if (rA48:63 = 0x8000) & (rB48:63 = 0x8000) then
                temp0:31 ← 0x7FFF_FFFF //saturate
                movl ← 1
else
                movl ← 0
temp0:63 ← EXTS(ACC32:63) - EXTS(temp0:31)
ovl ← (temp31 ⊕ temp32)
rD32:63 ← SATURATE(ovl, temp31, 0x8000_0000, 0x7FFF_FFFF, temp32:63)

// update accumulator
ACC0:63 ← rD0:63

// update SPEFSCR
SPEFSCROVH ← movh
SPEFSCROV ← movl
SPEFSCRSOVH ← SPEFSCRSOVH | ovh | movh
SPEFSCRSOV ← SPEFSCRSOV | ovl| movl
```

The corresponding odd-numbered half-word signed fractional elements in **r**A and **r**B 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 or underflow occurs, and the result is placed in **r**D and the accumulator.

If there is an overflow or underflow from either the multiply or the subtraction, the overflow and summary overflow bits are recorded in the SPEFSCR.

Other registers altered: SPEFSCR ACC

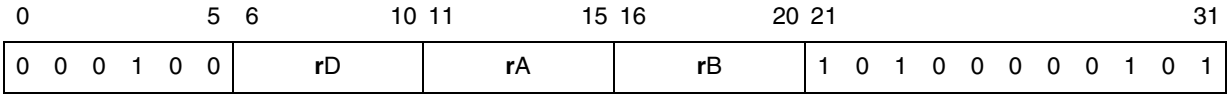
Figure 106. odd Form of Vector Half-Word Multiply (evmhossfanw)



evmhossiaaw | SPE APU | User | **evmhossiaaw**

Vector multiply half words, odd, signed, saturate, integer and accumulate into words

evmhossiaaw **r**D,**r**A,**r**B



// high
$temp_{0:31} \leftarrow rA_{16:31} \times_{si} rB_{16:31}$
$temp_{0:63} \leftarrow EXTS(ACC_{0:31}) + EXTS(temp_{0:31})$
$ovh \leftarrow (temp_{31} \oplus temp_{32})$
$rD_{0:31} \leftarrow SATURATE(ovh, temp_{31, 0x8000\_0000, 0x7FFF\_FFFF, temp32:63})$

// low
$temp_{0:31} \leftarrow rA_{48:63} \times_{si} rB_{48:63}$
$temp_{0:63} \leftarrow EXTS(ACC_{32:63}) + EXTS(temp_{0:31})$
$ovl \leftarrow (temp_{31} \oplus temp_{32})$
$rD_{32:63} \leftarrow SATURATE(ovl, temp_{31, 0x8000\_0000, 0x7FFF\_FFFF, temp32:63})$

// update accumulator
$ACC_{0:63} \leftarrow rD_{0:63}$

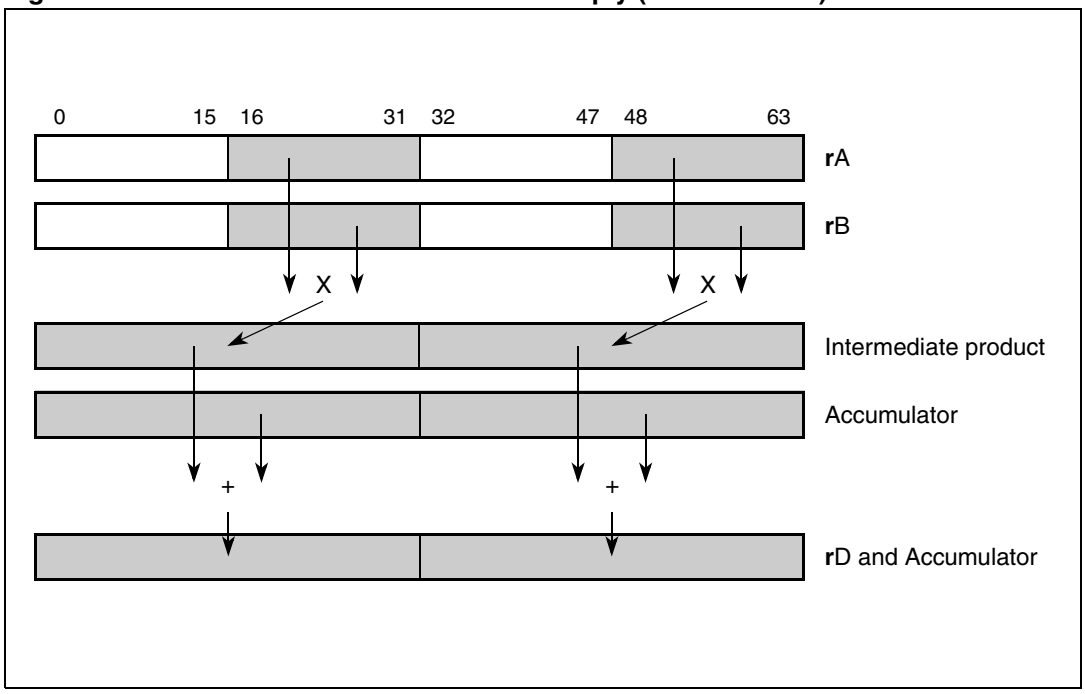
// update SPEFSCR
$SPEFSCR_{OVH} \leftarrow ovh$
$SPEFSCR_{OV} \leftarrow ovl$
$SPEFSCR_{SOVH} \leftarrow SPEFSCR_{SOVH} \mid ovh$
$SPEFSCR_{SOV} \leftarrow SPEFSCR_{SOV} \mid ovl$

The corresponding odd-numbered half-word signed integer elements in **r**A and **r**B 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 **r**D and the accumulator.

If there is an overflow or underflow from the addition, the overflow and summary overflow bits are recorded in the SPEFSCR.

Other registers altered: SPEFSCR ACC

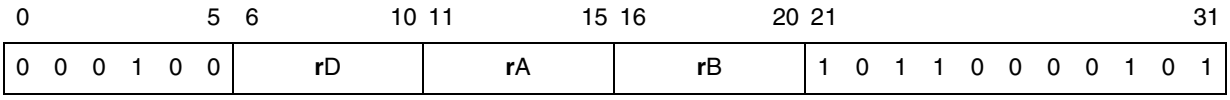
Figure 107. Odd form of vector half-word multiply (evmhossiaaw)



evmhossianw | SPE APU | User | **evmhossianw**

Vector multiply half words, odd, signed, saturate, integer and accumulate negative into words

evmhossianw **r**D,**r**A,**r**B



// high
$temp_{0:31} \leftarrow rA_{16:31} \times_{si} rB_{16:31}$
$temp_{0:63} \leftarrow EXTS(ACC_{0:31})$ - $EXTS(temp_{0:31})$
$ovh \leftarrow (temp_{31} \oplus temp_{32})$
$rD_{0:31} \leftarrow SATURATE(ovh, temp_{31, 0x8000\_0000, 0x7FFF\_FFFF, temp32:63})$

// low
$temp_{0:31} \leftarrow rA_{48:63} \times_{si} rB_{48:63}$
$temp_{0:63} \leftarrow EXTS(ACC_{32:63})$ - $EXTS(temp_{0:31})$
$ovl \leftarrow (temp_{31} \oplus temp_{32})$
$rD_{32:63} \leftarrow SATURATE(ovl, temp_{31, 0x8000\_0000, 0x7FFF\_FFFF, temp32:63})$

// update accumulator
$ACC_{0:63} \leftarrow rD_{0:63}$

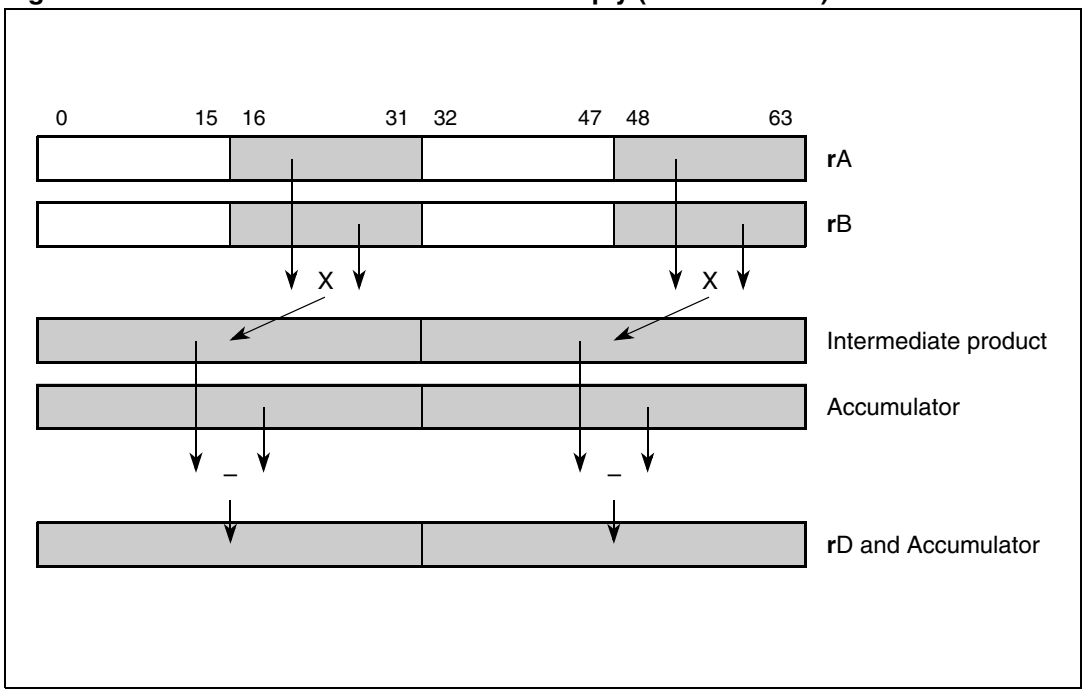
// update SPEFSCR
$SPEFSCR_{OVH} \leftarrow ovh$
$SPEFSCR_{OV} \leftarrow ovl$
$SPEFSCR_{SOVH} \leftarrow SPEFSCR_{SOVH} \mid ovh$
$SPEFSCR_{SOV} \leftarrow SPEFSCR_{SOV} \mid ovl$

The corresponding odd-numbered half-word signed integer elements in **r**A and **r**B 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 **r**D and the accumulator.

If there is an overflow or underflow from the subtraction, the overflow and summary overflow bits are recorded in the SPEFSCR.

Other registers altered: SPEFSCR ACC

Figure 108. Odd form of vector half-word multiply (evmhossianw)



evmhoumi | SPE APU | User | **evmhoumi**

Vector multiply half words, odd, unsigned, modulo, integer (to accumulator)

evmhoumi **r**D,**r**A,**r**B **(A = 0)**
evmhoumia **r**D,**r**A,**r**B **(A = 1)**

0	5	6	10	11	15	16	20	21	25	26	27	31
0 0 0 1 0 0		**r**D		**r**A		**r**B		1 0 0 0 0		A	0 1 1 0 0	

// high
$rD_{0:31} \leftarrow rA_{16:31} \times_{ui} rB_{16:31}$

// low
$rD_{32:63} \leftarrow rA_{48:63} \times_{ui} rB_{48:63}$

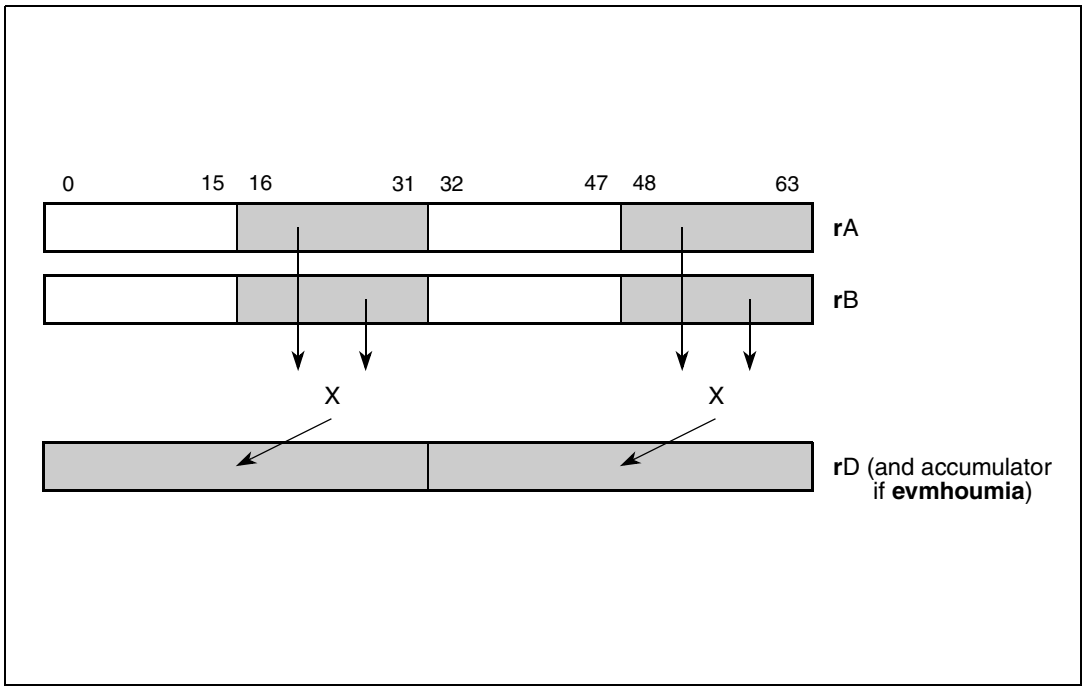
// update accumulator
if A = 1 then $ACC_{0:63} \leftarrow rD_{0:63}$

The corresponding odd-numbered half-word unsigned integer elements in **r**A and **r**B are multiplied. The two 32-bit products are placed into the corresponding words of **r**D.

If A = 1, the result in **r**D is also placed into the accumulator.

Other registers altered: ACC (If A = 1)

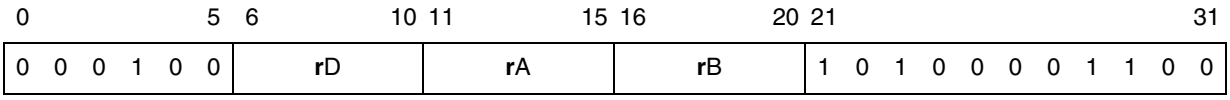
Figure 109. Vector multiply half words, odd, unsigned, modulo, integer (to accumulator) (evmhoumi)



evmhoumiaaw | SPE APU | User | **evmhoumiaaw**

Vector multiply half words, odd, unsigned, modulo, integer and accumulate into words

evmhoumiaaw **r**D,**r**A,**r**B



// high
$temp_{0:31} \leftarrow rA_{16:31} \times_{ui} rB_{16:31}$
$rD_{0:31} \leftarrow ACC_{0:31} + temp_{0:31}$

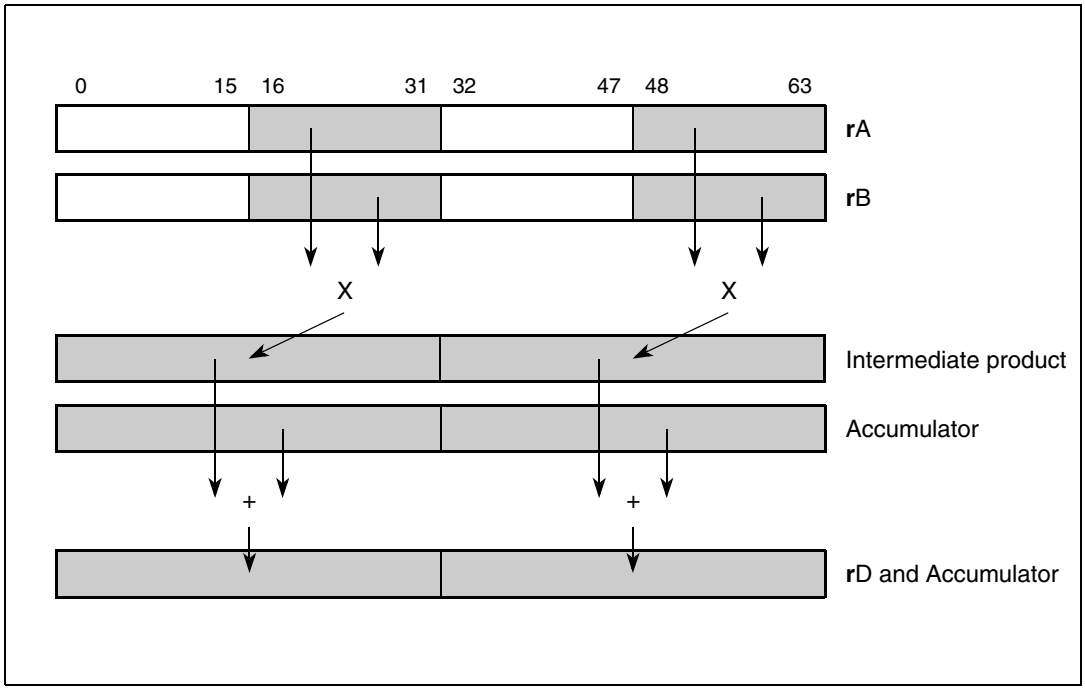
// low
$temp_{0:31} \leftarrow rA_{48:63} \times_{ui} rB_{48:63}$
$rD_{32:63} \leftarrow ACC_{32:63} + temp_{0:31}$

// update accumulator
$ACC_{0:63} \leftarrow rD_{0:63}$

For each word element in the accumulator, the corresponding odd-numbered half-word unsigned integer elements in **r**A and **r**B are multiplied. Each intermediate product is added to the contents of the corresponding accumulator word. The sums are placed into the corresponding **r**D and accumulator words.

Other registers altered: ACC

Figure 110. Odd form of vector half-word multiply (evmhoumiaaw)



evmhoumianw | SPE APU | User | **evmhoumianw**

Vector multiply half words, odd, unsigned, modulo, integer and accumulate negative into words

evmhoumianw **r**D,**r**A,**r**B

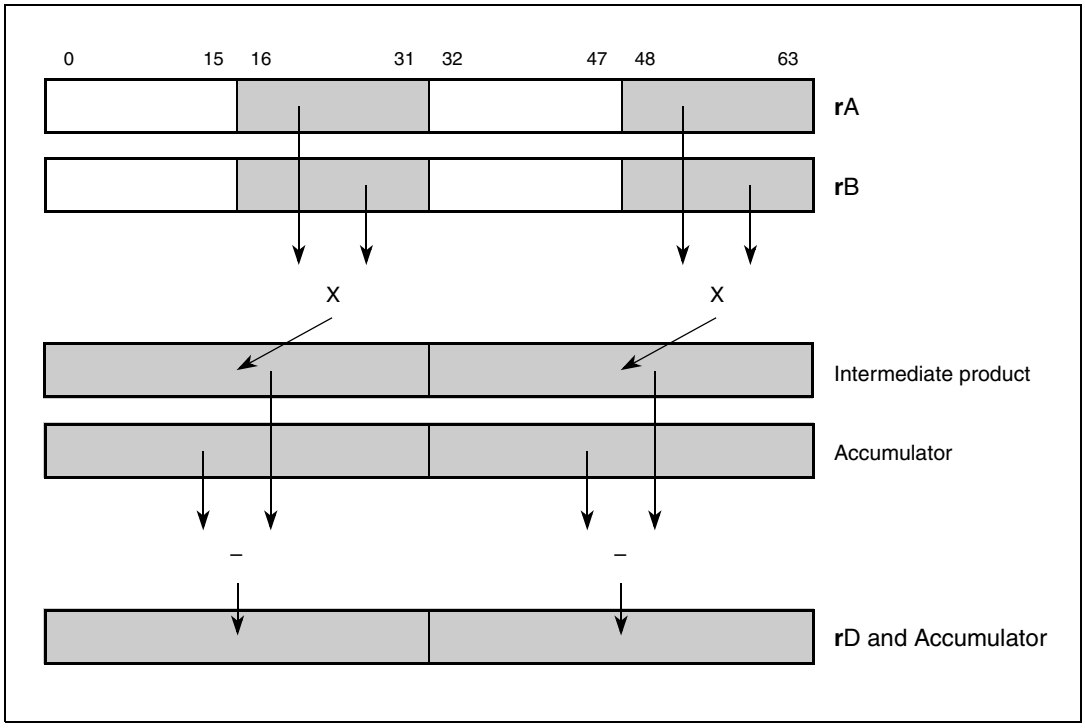
0	5	6	10	11	15	16	20	21	31
0 0 0 1 0 0		**r**D		**r**A		**r**B		1 0 1 1 0 0 0 1 1 0 0	

```
// high
temp0:31 ← rA0:15 ×ui rB0:15
rD0:31 ← ACC0:31 - temp0:31
/
/ low
temp0:31 ← rA32:47 ×ui rB32:47
rD32:63 ← ACC32:63 - temp0:31

// update accumulator
ACC0:63 ← rD0:63
```

For each word element in the accumulator, the corresponding odd-numbered half-word unsigned integer elements in **r**A and **r**B are multiplied. Each intermediate product is subtracted from the contents of the corresponding accumulator word. The results are placed into the corresponding **r**D and accumulator words.

Other registers altered: ACC

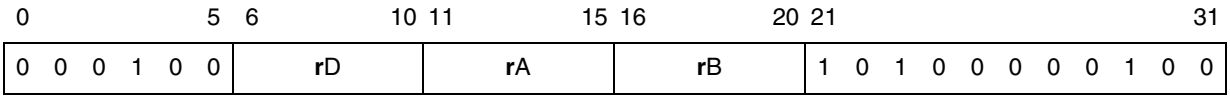
Figure 111. Odd form of vector half-word multiply (evmhoumianw)



evmhousiaaw | SPE APU | User | **evmhousiaaw**

Vector multiply half words, odd, unsigned, saturate, integer and accumulate into words

evmhousiaaw **r**D,**r**A,**r**B



// high
$temp_{0:31} \leftarrow rA_{16:31} \times_{ui} rB_{16:31}$
$temp_{0:63} \leftarrow EXTZ(ACC_{0:31}) + EXTZ(temp_{0:31})$
$ovh \leftarrow temp_{31}$
$rD_{0:31} \leftarrow SATURATE(ovh, 0, 0xFFFF\_FFFF, 0xFFFF\_FFFF, temp_{32:63})$

//low
$temp_{0:31} \leftarrow rA_{48:63} \times_{ui} rB_{48:63}$
$temp_{0:63} \leftarrow EXTZ(ACC_{32:63}) + EXTZ(temp_{0:31})$
$ovl \leftarrow temp_{31}$
$rD_{32:63} \leftarrow SATURATE(ovl, 0, 0xFFFF\_FFFF, 0xFFFF\_FFFF, temp_{32:63})$

// update accumulator
$ACC_{0:63} \leftarrow rD_{0:63}$

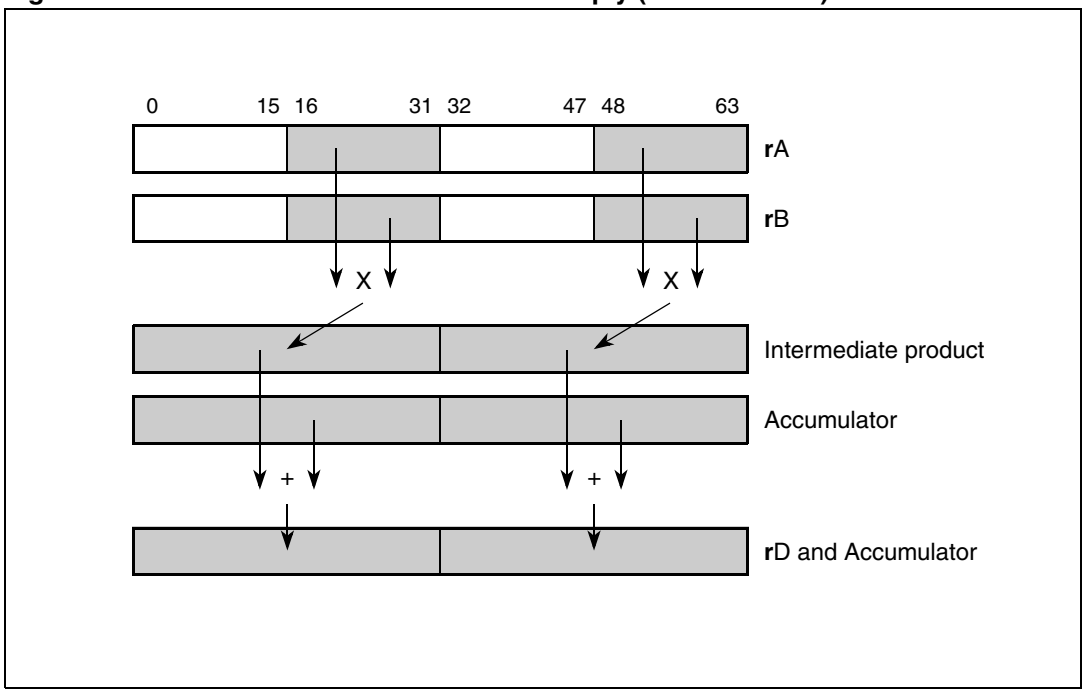
// update SPEFSCR
$SPEFSCR_{OVH} \leftarrow ovh$
$SPEFSCR_{OV} \leftarrow ovl$
$SPEFSCR_{SOVH} \leftarrow SPEFSCR_{SOVH} \mid ovh$
$SPEFSCR_{SOV} \leftarrow SPEFSCR_{SOV} \mid ovl$

For each word element in the accumulator, corresponding odd-numbered half-word unsigned integer elements in **r**A and **r**B 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 **r**D and the accumulator.

If the addition causes overflow, the overflow and summary overflow bits are recorded in the SPEFSCR.

Other registers altered: SPEFSCR ACC

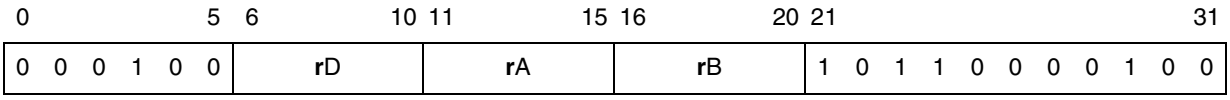
Figure 112. Odd form of vector half-word multiply (evmhousiaaw)



evmhousianw | SPE APU | User | **evmhousianw**

Vector multiply half words, odd, unsigned, saturate, integer and accumulate negative into words

evmhousianw **r**D,**r**A,**r**B



// high
$temp_{0:31} \leftarrow rA_{16:31} \times_{ui} rB_{16:31}$
$temp_{0:63} \leftarrow EXTZ(ACC_{0:31})$ - $EXTZ(temp_{0:31})$
$ovh \leftarrow temp_{31}$
$rD_{0:31} \leftarrow SATURATE(ovh, 0, 0xFFFF\_FFFF, 0xFFFF\_FFFF, temp_{32:63})$

//low
$temp_{0:31} \leftarrow rA_{48:63} \times_{ui} rB_{48:63}$
$temp_{0:63} \leftarrow EXTZ(ACC_{32:63})$ - $EXTZ(temp_{0:31})$
$ovl \leftarrow temp_{31}$
$rD_{32:63} \leftarrow SATURATE(ovl, 0, 0xFFFF\_FFFF, 0xFFFF\_FFFF, temp_{32:63})$

// update accumulator
$ACC_{0:63} \leftarrow rD_{0:63}$

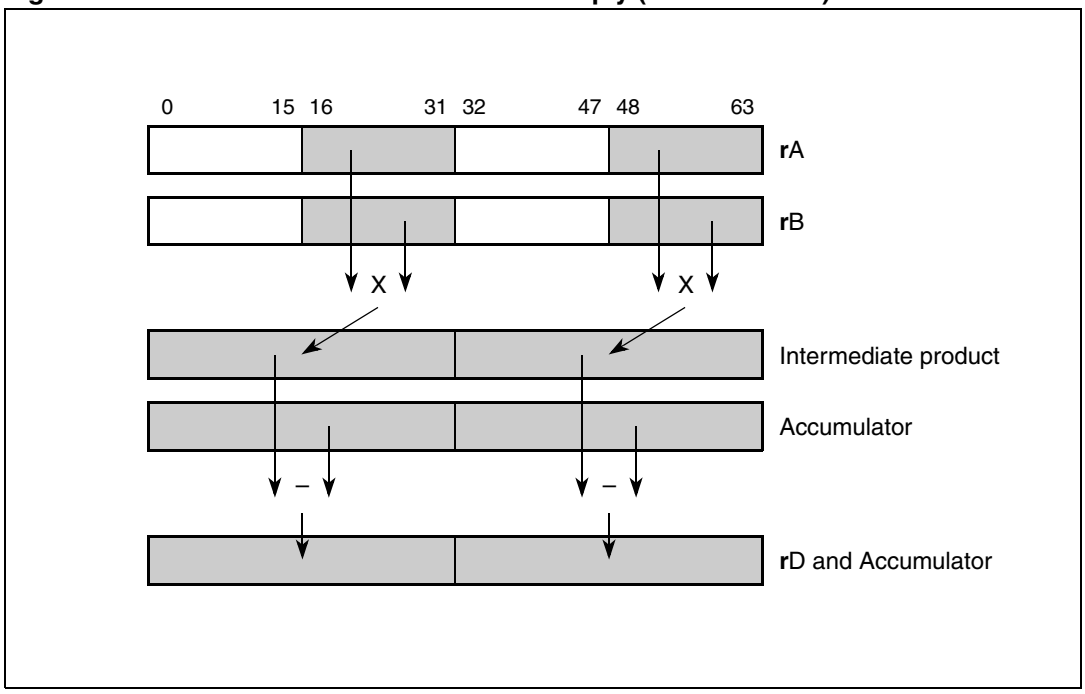
// update SPEFSCR
$SPEFSCR_{OVH} \leftarrow ovh$
$SPEFSCR_{OV} \leftarrow ovl$
$SPEFSCR_{SOVH} \leftarrow SPEFSCR_{SOVH} \mid ovh$
$SPEFSCR_{SOV} \leftarrow SPEFSCR_{SOV} \mid ovl$

For each word element in the accumulator, corresponding odd-numbered half-word unsigned integer elements in **r**A and **r**B 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 **r**D and the accumulator.

If subtraction causes overflow, the overflow and summary overflow bits are recorded in the SPEFSCR.

Other registers altered: SPEFSCR ACC

Figure 113. Odd form of vector half-word multiply (evmhousianw)



evmra | SPE APU | User | **evmra**

Initialize accumulator

evmra **r**D,**r**A

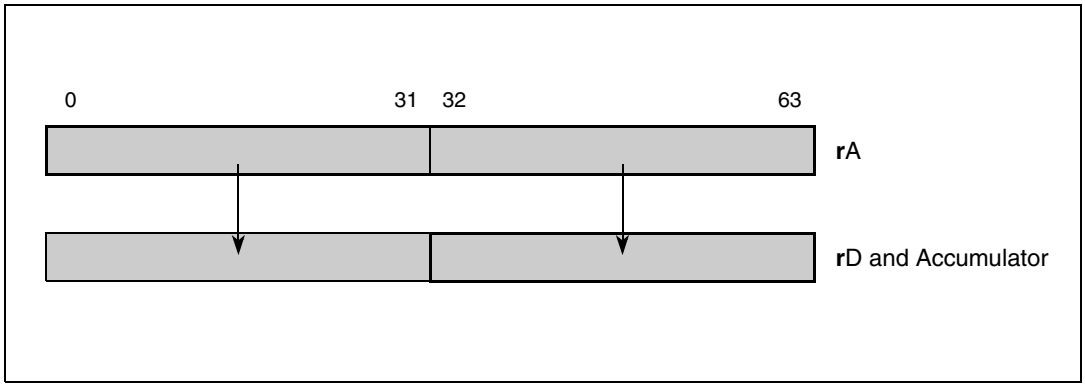
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	0	1	0	0	**r**D					**r**A					0	0	0	0	0	1	0	0	1	1	0	0	0	1	0	0

$ACC_{0:63} \leftarrow rA_{0:63}$
$rD_{0:63} \leftarrow rA_{0:63}$

The contents of **r**A are written into the accumulator and copied into **r**D. This is the method for initializing the accumulator.

Other registers altered: ACC

Figure 114. Initialize accumulator (evmra)



evmwhsmf | SPE APU | User | **evmwhsmf**

Vector multiply word high signed, modulo, fractional (to accumulator)

evmwhsmf **r**D,**r**A,**r**B **(A = 0)**
evmwhsmfa **r**D,**r**A,**r**B **(A = 1)**

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	0	1	0	0	**r**D					**r**A					**r**B					1	0	0	0	1	A	0	1	1	1	1

// high
$temp_{0:63} \leftarrow rA_{0:31} \times_{sf} rB_{0:31}$
$rD_{0:31} \leftarrow temp_{0:31}$

// low
$temp_{0:63} \leftarrow rA_{32:63} \times_{sf} rB_{32:63}$
$rD_{32:63} \leftarrow temp_{0:31}$

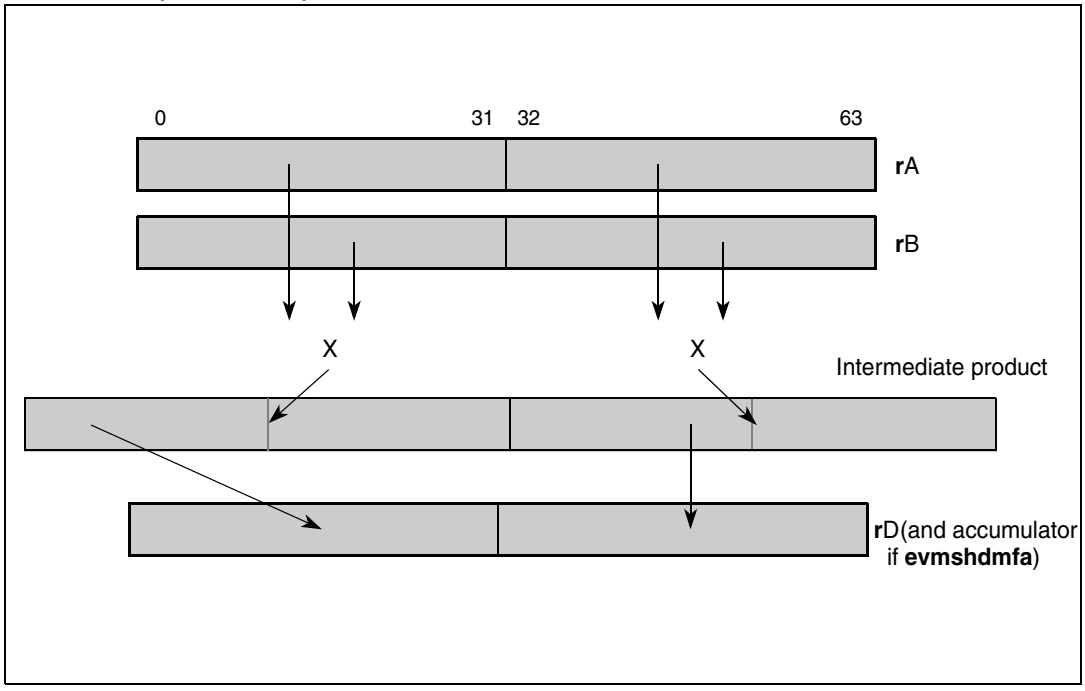
// update accumulator
if A = 1 then $ACC_{0:63} \leftarrow rD_{0:63}$

The corresponding word signed fractional elements in **r**A and **r**B are multiplied and bits 0–31 of the two products are placed into the two corresponding words of **r**D.

If A = 1, the result in **r**D is also placed into the accumulator.

Other registers altered: ACC (if A =1)

Figure 115. Vector multiply word high signed, modulo, fractional (to accumulator) (evmwhsmf)



evmwhsmi | SPE APU | User | **evmwhsmi**

Vector multiply word high signed, modulo, integer (to accumulator)

evmwhsmi **r**D,**r**A,**r**B **(A = 0)**
evmwhsmia **r**D,**r**A,**r**B **(A = 1)**

0	5	6	10	11	15	16	20	21	25	26	27	31
0 0 0 1 0 0		**r**D		**r**A		**r**B		1 0 0 0 1		A	0 1 1 0 1	

// high
$temp_{0:63} \leftarrow rA_{0:31} \times_{si} rB_{0:31}$
$rD_{0:31} \leftarrow temp_{0:31}$

// low
$temp_{0:63} \leftarrow rA_{32:63} \times_{si} rB_{32:63}$
$rD_{32:63} \leftarrow temp_{0:31}$

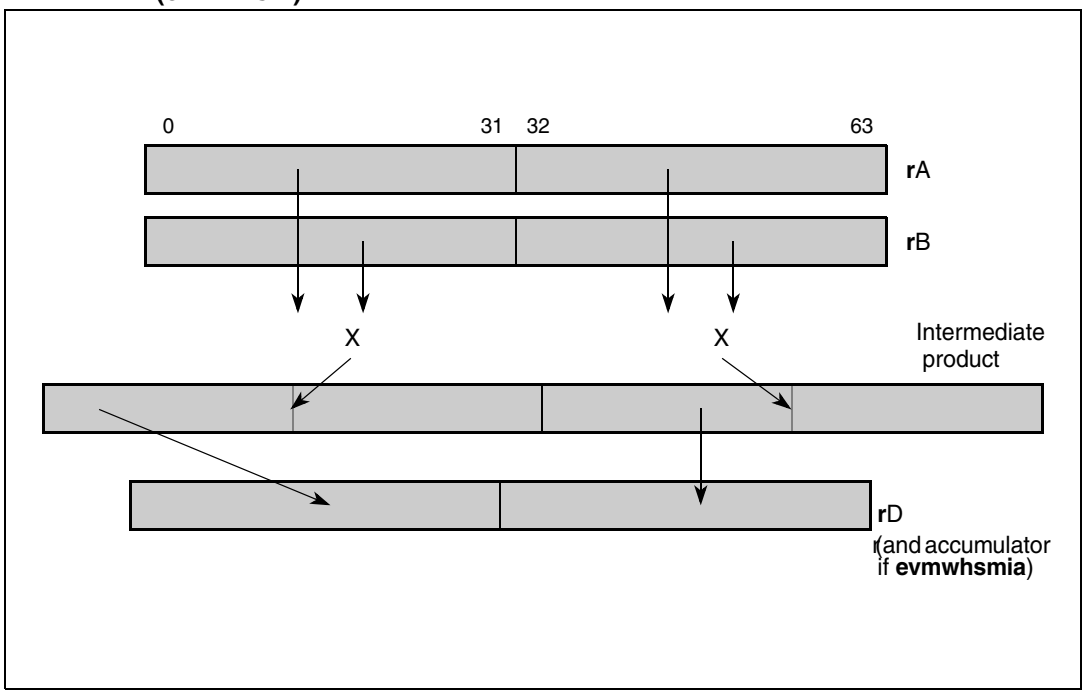
// update accumulator
if A = 1 then $ACC_{0:63} \leftarrow rD_{0:63}$

The corresponding word signed integer elements in **r**A and **r**B are multiplied. Bits 0–31 of the two 64-bit products are placed into the two corresponding words of **r**D.

If A = 1,The result in **r**D is also placed into the accumulator.

Other registers altered: ACC (If A = 1)

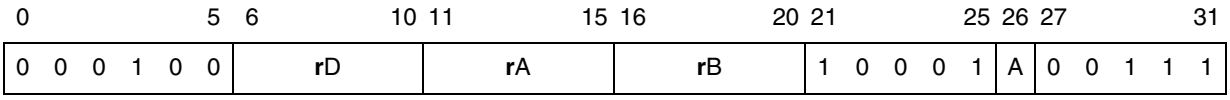
Figure 116. Vector multiply word high signed, modulo, integer (to accumulator) (evmwhsm)



evmwhssf | SPE APU | User | **evmwhssf**

Vector multiply word high signed, saturate, fractional (to accumulator)

evmwhssf **r**D,**r**A,**r**B **(A = 0)**
evmwhssfa **r**D,**r**A,**r**B **(A = 1)**

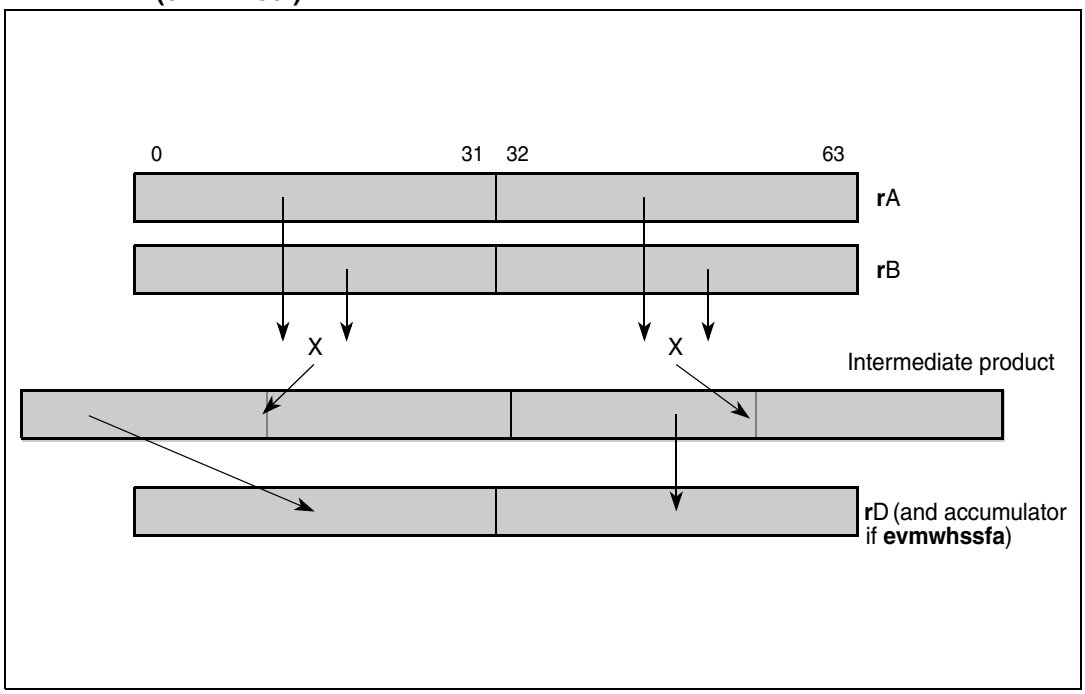



```
// high
temp0:63 ← rA0:31 ×sf rB0:31
if (rA0:31 = 0x8000_0000) & (rB0:31 = 0x8000_0000) then
        rD0:31 ← 0x7FFF_FFFF //saturate
        movh ← 1
else
        rD0:31 ← temp0:31
        movh ← 0

// low
temp0:63 ← rA32:63 ×sf rB32:63
if (rA32:63 = 0x8000_0000) & (rB32:63 = 0x8000_0000) then
        rD32:63 ← 0x7FFF_FFFF //saturate
        movl ← 1
else
        rD32:63 ← temp0:31
        movl ← 0

// update accumulator
if A = 1 then ACC0:63 ← rD0:63

// update SPEFSCR
SPEFSCROVH ← movh
SPEFSCROV ← movl
SPEFSCRSOVH ← SPEFSCRSOVH | movh
SPEFSCRSOV ← SPEFSCRSOV | movl
```

The corresponding word signed fractional elements in **r**A and **r**B are multiplied. Bits 0–31 of each product are placed into the corresponding words of **r**D. If both inputs are –1.0, the result saturates to the largest positive signed fraction and the overflow and summary overflow bits are recorded in the SPEFSCR.

Other registers altered: SPEFSCR ACC (If A = 1)

Figure 117. Vector multiply word high signed, saturate, fractional (to accumulator) (evmwhssf)



evmwhumi | SPE APU | User | **evmwhumi**

Vector multiply word high unsigned, modulo, integer (to accumulator)

evmwhumi **r**D,**r**A,**r**B **(A = 0)**
evmwhumia **r**D,**r**A,**r**B **(A = 1)**

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	0	1	0	0	**r**D					**r**A					**r**B					1	0	0	0	1	A	0	1	1	0	0

// high
$temp_{0:63} \leftarrow rA_{0:31} \times_{ui} rB_{0:31}$
$rD_{0:31} \leftarrow temp_{0:31}$

// low
$temp_{0:63} \leftarrow rA_{32:63} \times_{ui} rB_{32:63}$
$rD_{32:63} \leftarrow temp_{0:31}$

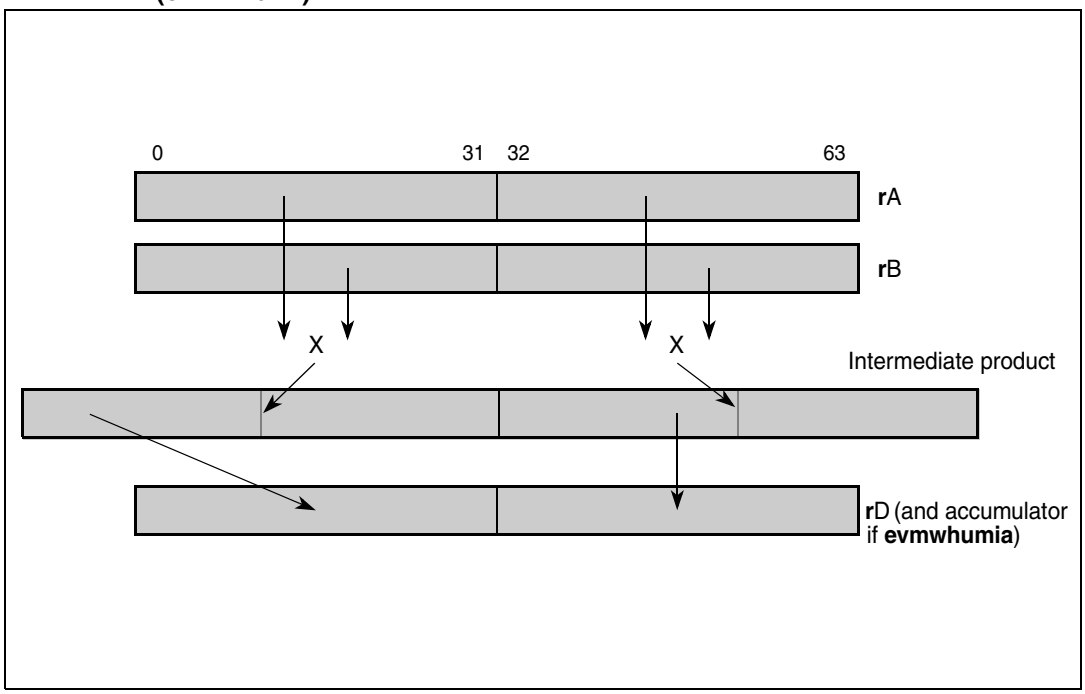
// update accumulator
if A = 1, $ACC_{0:63} \leftarrow rD_{0:63}$

The corresponding word unsigned integer elements in **r**A and **r**B are multiplied. Bits 0–31 of the two products are placed into the two corresponding words of **r**D.

If A = 1, the result in **r**D is also placed into the accumulator.

Other registers altered: ACC (If A = 1)

Figure 118. Vector multiply word high unsigned, modulo, integer (to accumulator) (evmwhumi)



evmwlsmiaaw | SPE APU | User | **evmwlsmiaaw**

Vector multiply word low signed, modulo, integer and accumulate in words

evmwlsmiaaw **r**D,**r**A,**r**B

0	5	6	10	11	15	16	20	21	31
0 0 0 1 0 0		**r**D		**r**A		**r**B		1 0 1 0 1 0 0 1 0 0 1	

// high
$temp_{0:63} \leftarrow rA_{0:31} \times_{si} rB_{0:31}$
$rD_{0:31} \leftarrow ACC_{0:31} + temp_{32:63}$

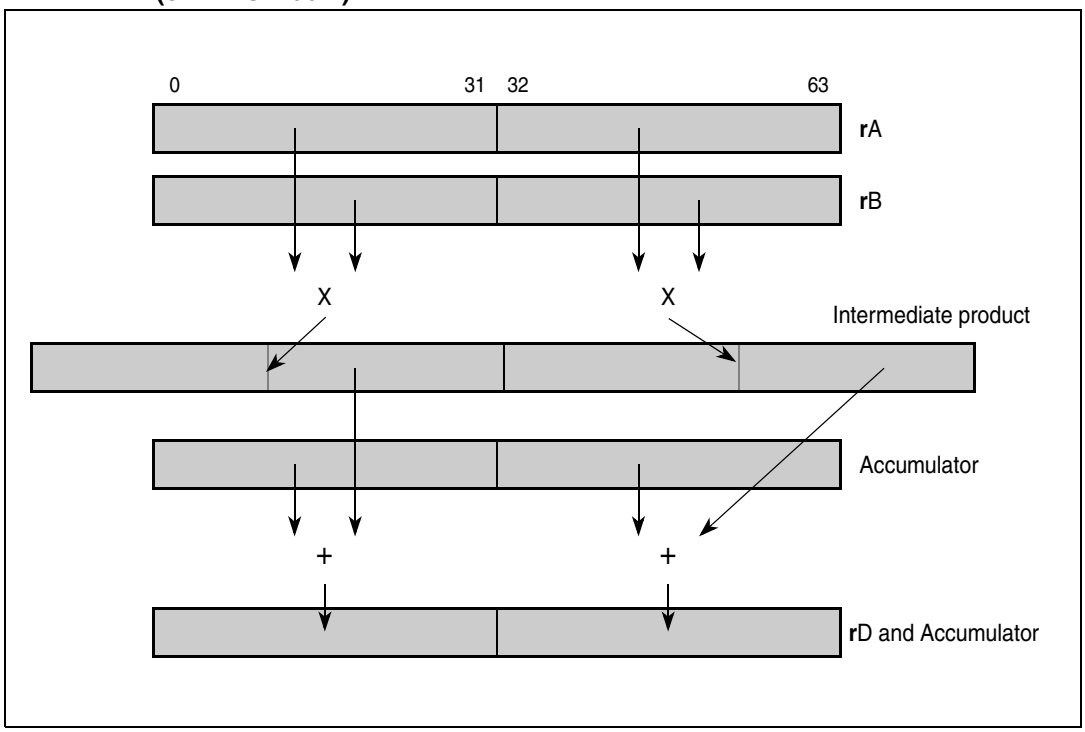
// low
$temp_{0:63} \leftarrow rA_{32:63} \times_{si} rB_{32:63}$
$rD_{32:63} \leftarrow ACC_{32:63} + temp_{32:63}$

// update accumulator
$ACC_{0:63} \leftarrow rD_{0:63}$

For each word element in the accumulator, the corresponding word signed integer elements in **r**A and **r**B are multiplied. The least significant 32 bits of each intermediate product is added to the contents of the corresponding accumulator words, and the result is placed into **r**D and the accumulator.

Other registers altered: ACC

Figure 119. Vector multiply word low signed, modulo, integer & accumulate in words (evmwlsmiaaw)



evmwlsmianw | SPE APU | User | **evmwlsmianw**

Vector multiply word low signed, modulo, integer and accumulate negative in words

evmwlsmianw **r**D,**r**A,**r**B

0	5	6	10	11	15	16	20	21	31
0 0 0 1 0 0		**r**D		**r**A		**r**B		1 0 1 1 1 0 0 1 0 0 1	

// high
$temp_{0:63} \leftarrow rA_{0:31} \times_{si} rB_{0:31}$
$rD_{0:31} \leftarrow ACC_{0:31} - temp_{32:63}$

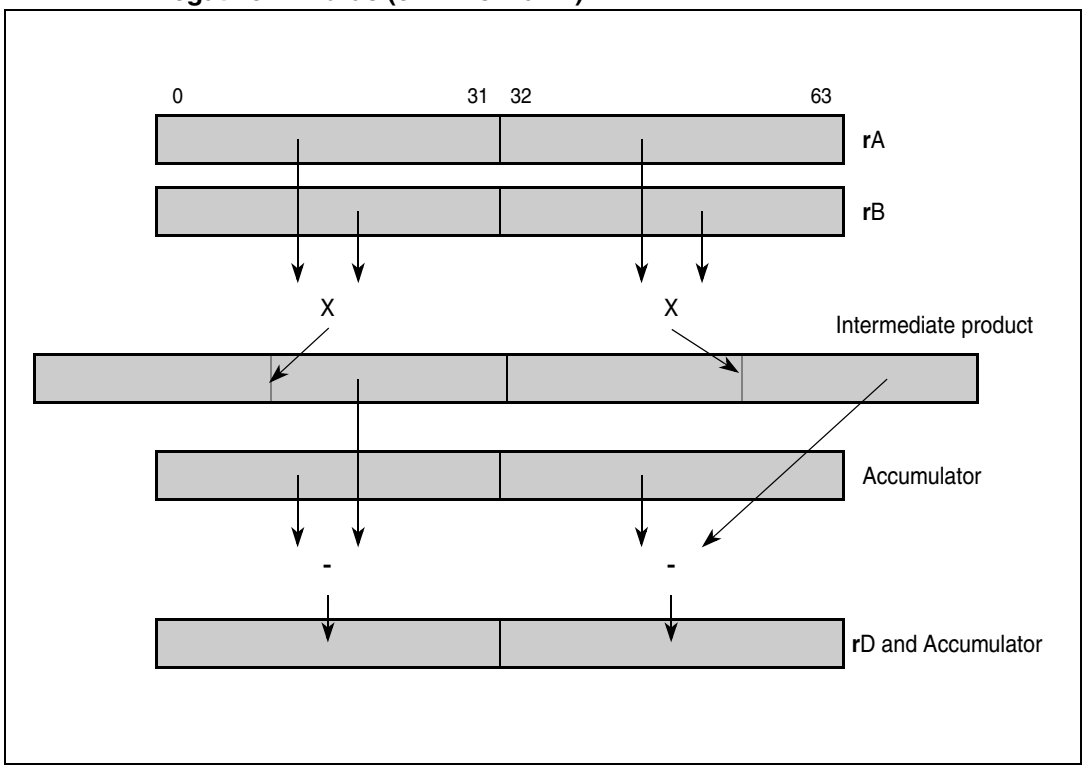
// low
$temp_{0:63} \leftarrow rA_{32:63} \times_{si} rB_{32:63}$
$rD_{32:63} \leftarrow ACC_{32:63} - temp_{32:63}$

// update accumulator
$ACC_{0:63} \leftarrow rD_{0:63}$

For each word element in the accumulator, the corresponding word elements in **r**A and **r**B are multiplied. The least significant 32 bits of each intermediate product is subtracted from the contents of the corresponding accumulator words and the result is placed in **r**D and the accumulator.

Other registers altered: ACC

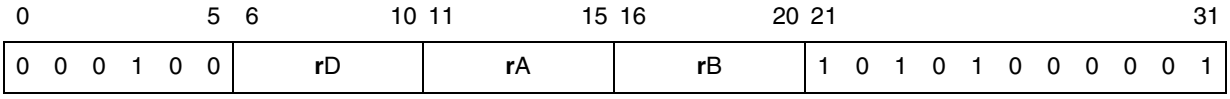
Figure 120. Vector multiply word low signed, modulo, integer and accumulate negative in words (evmwlsmianw)



evmwlssiaaw | SPE APU | User | **evmwlssiaaw**

Vector multiply word low signed, saturate, integer and accumulate in words

evmwlssiaaw **r**D,**r**A,**r**B



// high
$temp_{0:63} \leftarrow rA_{0:31} \times_{si} rB_{0:31}$
$temp_{0:63} \leftarrow EXTS(ACC_{0:31}) + EXTS(temp_{32:63})$
$ovh \leftarrow (temp_{31} \oplus temp_{32})$
$rD_{0:31} \leftarrow SATURATE(ovh, temp_{31, 0x8000\_0000, 0x7FFF\_FFFF, temp32:63})$

// low
$temp_{0:63} \leftarrow rA_{32:63} \times_{si} rB_{32:63}$
$temp_{0:63} \leftarrow EXTS(ACC_{32:63}) + EXTS(temp_{32:63})$
$ovl \leftarrow (temp_{31} \oplus temp_{32})$
$rD_{32:63} \leftarrow SATURATE(ovl, temp_{31, 0x8000\_0000, 0x7FFF\_FFFF, temp32:63})$

// update accumulator
$ACC_{0:63} \leftarrow rD_{0:63}$

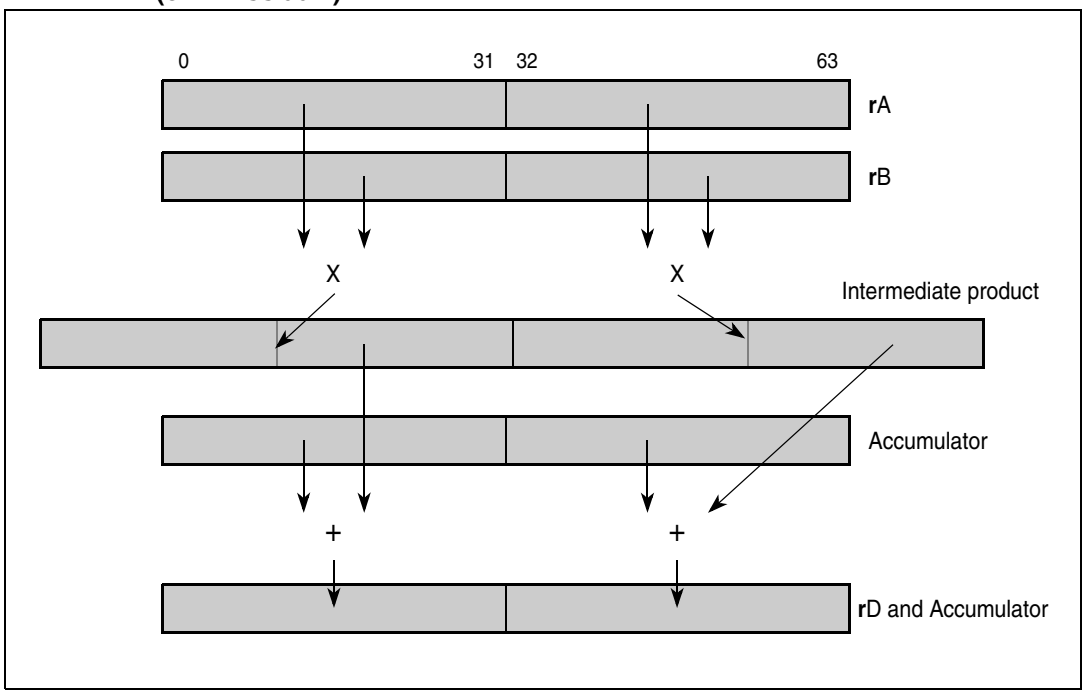
// update SPEFSCR
$SPEFSCR_{OVH} \leftarrow ovh$
$SPEFSCR_{OV} \leftarrow ovl$
$SPEFSCR_{SOVH} \leftarrow SPEFSCR_{SOVH} \mid ovh$
$SPEFSCR_{SOV} \leftarrow SPEFSCR_{SOV} \mid ovl$

The corresponding word signed integer elements in **r**A and **r**B are multiplied producing a 64-bit product. The 32 lsbs of each product is added to the corresponding word in the ACC, saturating if overflow or underflow occurs; the result is placed in **r**D and the accumulator.

If there is overflow or underflow from the addition, overflow and summary overflow bits are recorded in the SPEFSCR.

Other registers altered: SPEFSCR ACC

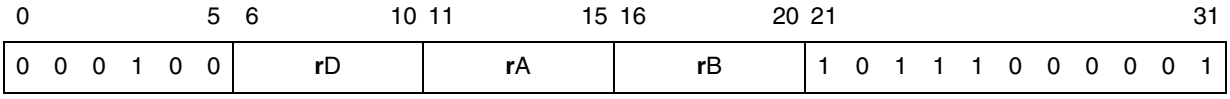
Figure 121. Vector multiply word low signed, saturate, integer & accumulate in words (evmwlssiaaw)



evmwlssianw | SPE APU | User | **evmwlssianw**

Vector multiply word low signed, saturate, integer and accumulate negative in words

evmwlssianw **r**D,**r**A,**r**B



// high
$temp_{0:63} \leftarrow rA_{0:31} \times_{si} rB_{0:31}$
$temp_{0:63} \leftarrow EXTS(ACC_{0:31}) - EXTS(temp_{32:63})$
$ovh \leftarrow (temp_{31} \oplus temp_{32})$
$rD_{0:31} \leftarrow SATURATE(ovh, temp_{31, 0x8000\_0000, 0x7FFF\_FFFF, temp32:63})$

// low
$temp_{0:63} \leftarrow rA_{32:63} \times_{si} rB_{32:63}$
$temp_{0:63} \leftarrow EXTS(ACC_{32:63}) - EXTS(temp_{32:63})$
$ovl \leftarrow (temp_{31} \oplus temp_{32})$
$rD_{32:63} \leftarrow SATURATE(ovl, temp_{31, 0x8000\_0000, 0x7FFF\_FFFF, temp32:63})$

// update accumulator
$ACC_{0:63} \leftarrow rD_{0:63}$

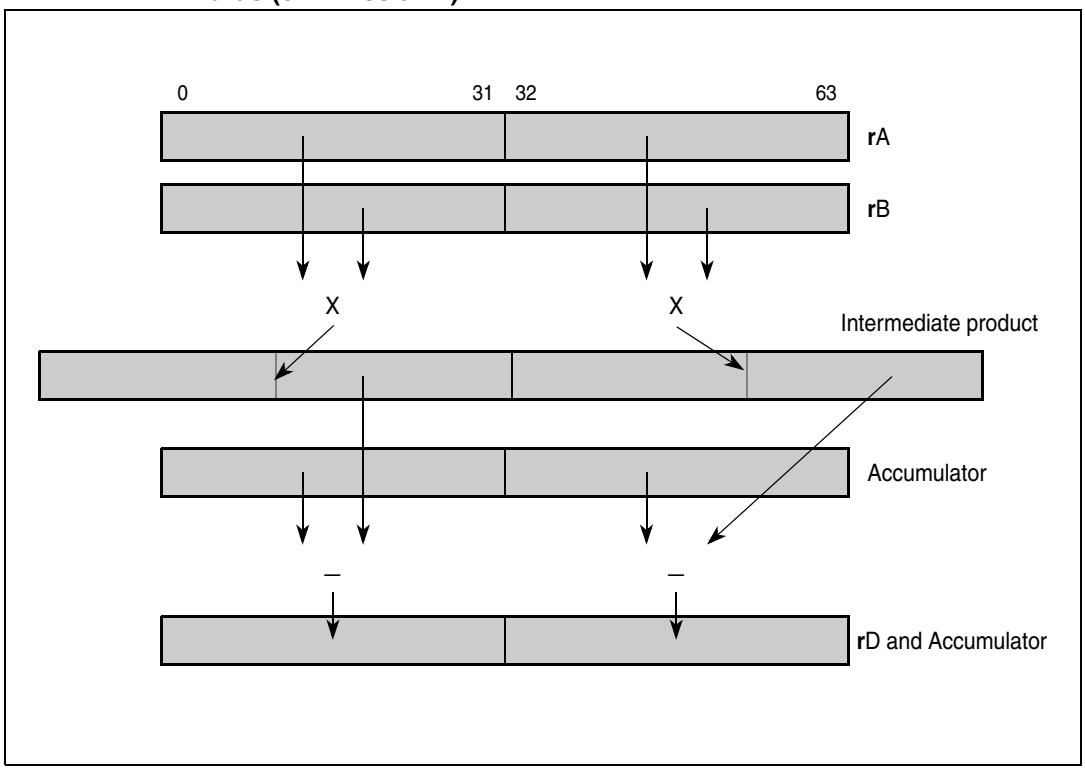
// update SPEFSCR
$SPEFSCR_{OVH} \leftarrow ovh$
$SPEFSCR_{OV} \leftarrow ovl$
$SPEFSCR_{SOVH} \leftarrow SPEFSCR_{SOVH} \mid ovh$
$SPEFSCR_{SOV} \leftarrow SPEFSCR_{SOV} \mid ovl$

The corresponding word signed integer elements in **r**A and **r**B are multiplied producing a 64-bit product. The 32 lsbs of each product are subtracted from the corresponding ACC word, saturating if overflow or underflow occurs, and the result is placed in **r**D and the ACC.

If addition causes overflow or underflow, overflow and summary overflow SPEFSCR bits are recorded.

Other registers altered: SPEFSCR ACC

Figure 122. Vector multiply word low signed, saturate, integer & accumulate negative in words (evmwlssianw)



evmwlumi | SPE APU | User | **evmwlumi**

Vector multiply word low unsigned, modulo, integer

evmwlumi **rD,rA,rB** (A = 0)

evmwlumia **rD,rA,rB** (A = 1)

0	5	6	10	11	15	16	20	21	25	26	27	31
0 0 0 1 0 0		**r**d		**r**a		**r**b		1 0 0 0 1		a	0 1 0 0 0	

```
// high
temp0:63 ← rA0:31 ×ui rB0:31
rD0:31 ← temp32:63

// low
temp0:63 ← rA32:63 ×ui rB32:63
rD32:63 ← temp32:63

// update accumulator
If A = 1 then ACC0:63 ← rD0:63
```

The corresponding word unsigned integer elements in **r**A and **r**B are multiplied. The least significant 32 bits of each product are placed into the two corresponding words of **r**D.

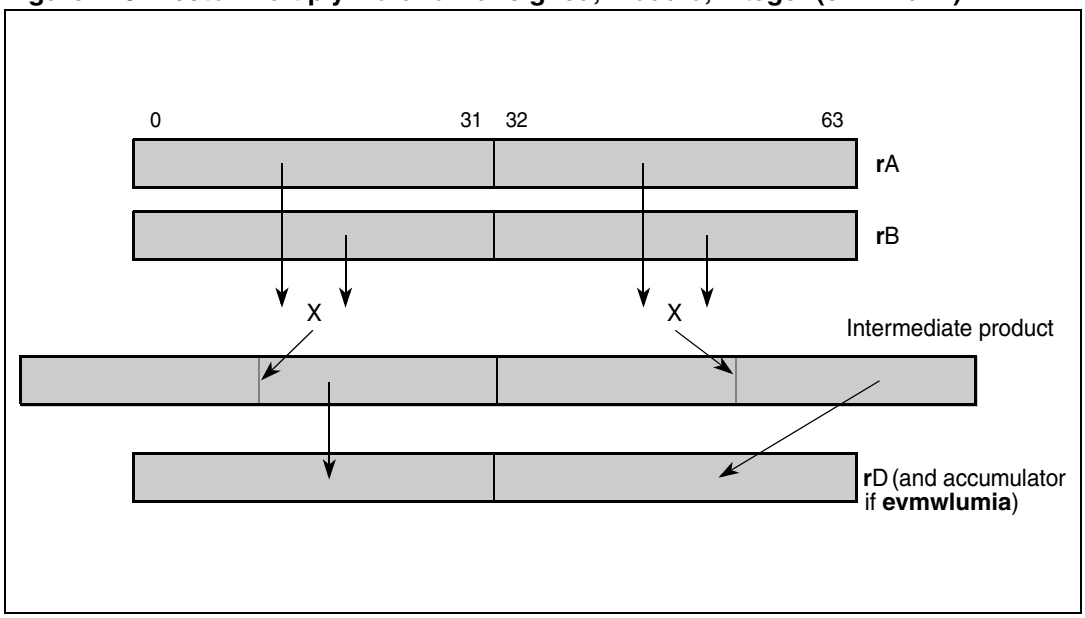
Note: The least significant 32 bits of the product are independent of whether the word elements in **r**A and **r**B are treated as signed or unsigned 32-bit integers.

If A = 1, the result in **r**D is also placed into the accumulator.

Other registers altered: ACC (If A = 1)

Note that **evmwlumi** and **evmwlumia** can be used for signed or unsigned integers.

Figure 123. Vector multiply word low unsigned, modulo, integer (evmwlumi)



evmwlumiaaw | SPE APU | User | **evmwlumiaaw**

Vector multiply word low unsigned, modulo, integer and accumulate in words

evmwlumiaaw **r**D,**r**A,**r**B

0	5	6	10	11	15	16	20	21	31
0 0 0 1 0 0		**r**D		**r**A		**r**B		1 0 1 0 1 0 0 1 0 0 0	

// high
$temp_{0:63} \leftarrow rA_{0:31} \times_{ui} rB_{0:31}$
$rD_{0:31} \leftarrow ACC_{0:31} + temp_{32:63}$

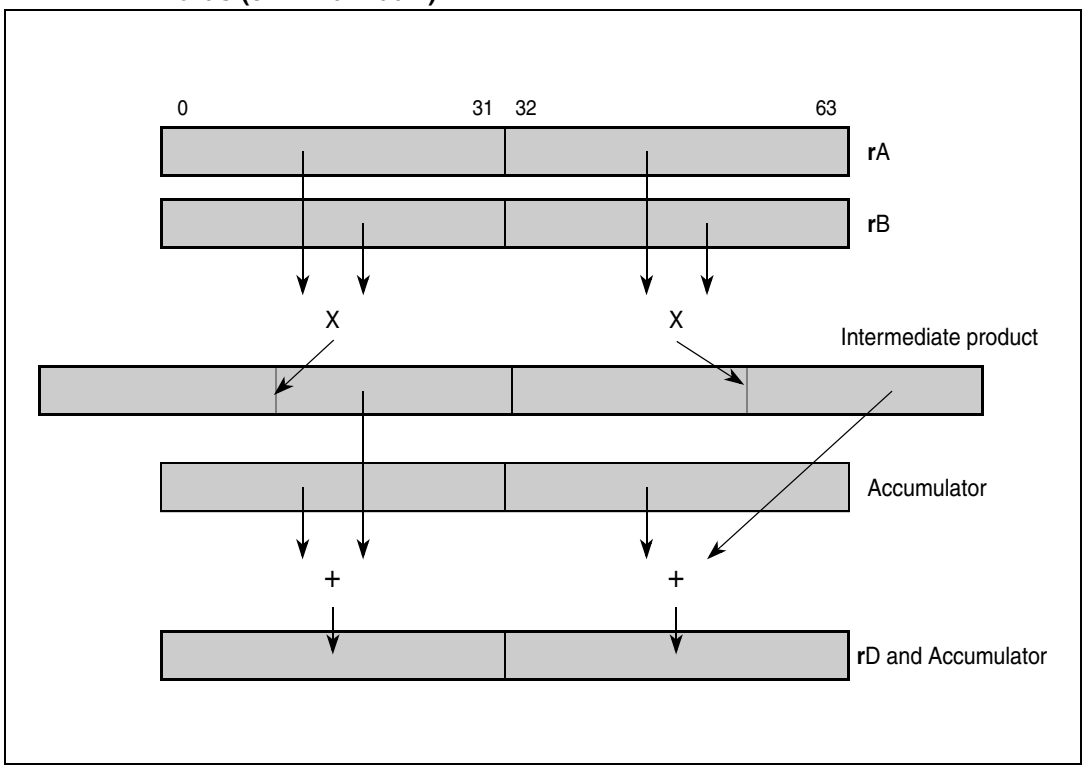
// low
$temp_{0:63} \leftarrow rA_{32:63} \times_{ui} rB_{32:63}$
$rD_{32:63} \leftarrow ACC_{32:63} + temp_{32:63}$

// update accumulator
$ACC_{0:63} \leftarrow rD_{0:63}$

For each word element in the accumulator, the corresponding word unsigned integer elements in **r**A and **r**B 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 into **r**D and the accumulator.

Other registers altered: ACC

Figure 124. Vector multiply word low unsigned, modulo, integer & accumulate in words (evmwlumiaaw)



evmwlumianw | SPE APU | User | **evmwlumianw**

Vector multiply word low unsigned, modulo, integer and accumulate negative in words

evmwlumianw **r**D,**r**A,**r**B

0	5	6	10	11	15	16	20	21	31
0 0 0 1 0 0		**r**D		**r**A		**r**B		1 0 1 1 1 0 0 1 0 0 0	

// high
$temp_{0:63} \leftarrow rA_{0:31} \times_{ui} rB_{0:31}$
$rD_{0:31} \leftarrow ACC_{0:31} - temp_{32:63}$

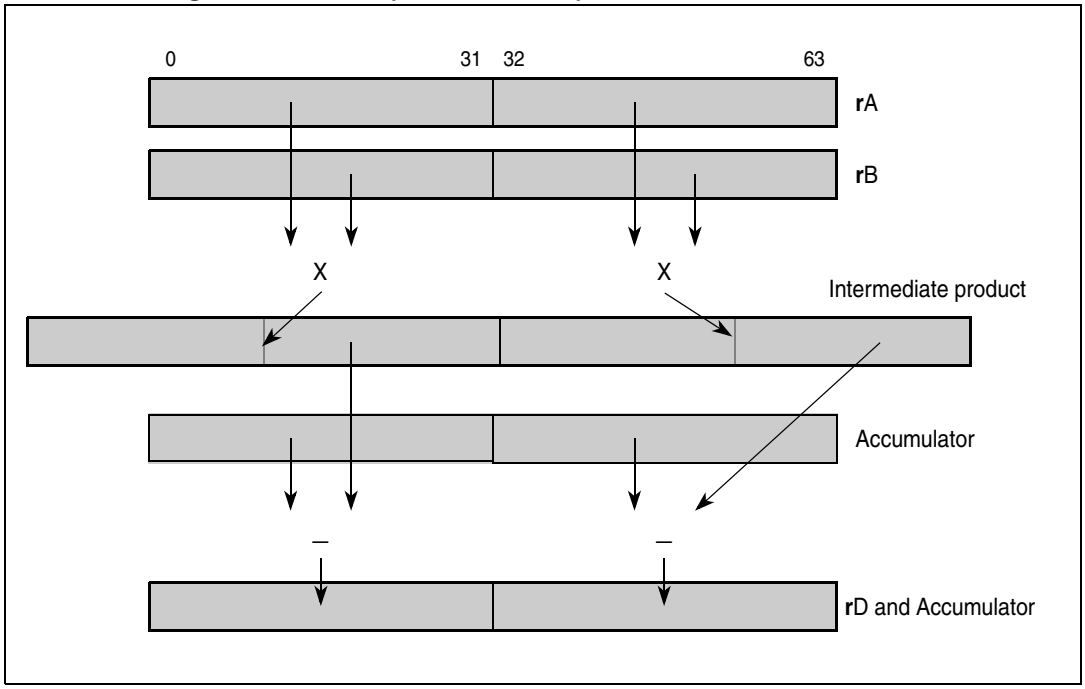
// low
$temp_{0:63} \leftarrow rA_{32:63} \times_{ui} rB_{32:63}$
$rD_{32:63} \leftarrow ACC_{32:63} - temp_{32:63}$

// update accumulator
$ACC_{0:63} \leftarrow rD_{0:63}$

For each word element in the accumulator, the corresponding word unsigned integer elements in **r**A and **r**B 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 into **r**D and the accumulator.

Other registers altered: ACC

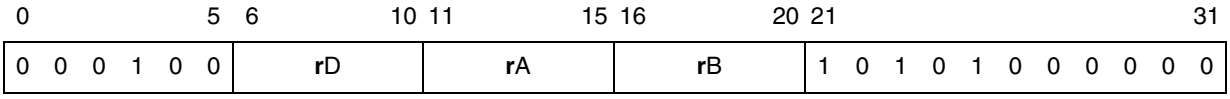
Figure 125. Vector multiply word low unsigned, modulo, integer & accumulate negative in words (evmwlumianw)



evmwlusiaaw | SPE APU | User | **evmwlusiaaw**

Vector multiply word low unsigned, saturate, integer and accumulate in words

evmwlusiaaw **r**D,**r**A,**r**B



// high
$temp_{0:63} \leftarrow rA_{0:31} \times_{ui} rB_{0:31}$
$temp_{0:63} \leftarrow EXTZ(ACC_{0:31}) + EXTZ(temp_{32:63})$
$ovh \leftarrow temp_{31}$
$rD_{0:31} \leftarrow SATURATE(ovh, 0, 0xFFFF\_FFFF, 0xFFFF\_FFFF, temp_{32:63})$

//low
$temp_{0:63} \leftarrow rA_{32:63} \times_{ui} rB_{32:63}$
$temp_{0:63} \leftarrow EXTZ(ACC_{32:63}) + EXTZ(temp_{32:63})$
$ovl \leftarrow temp_{31}$
$rD_{32:63} \leftarrow SATURATE(ovl, 0, 0xFFFF\_FFFF, 0xFFFF\_FFFF, temp_{32:63})$

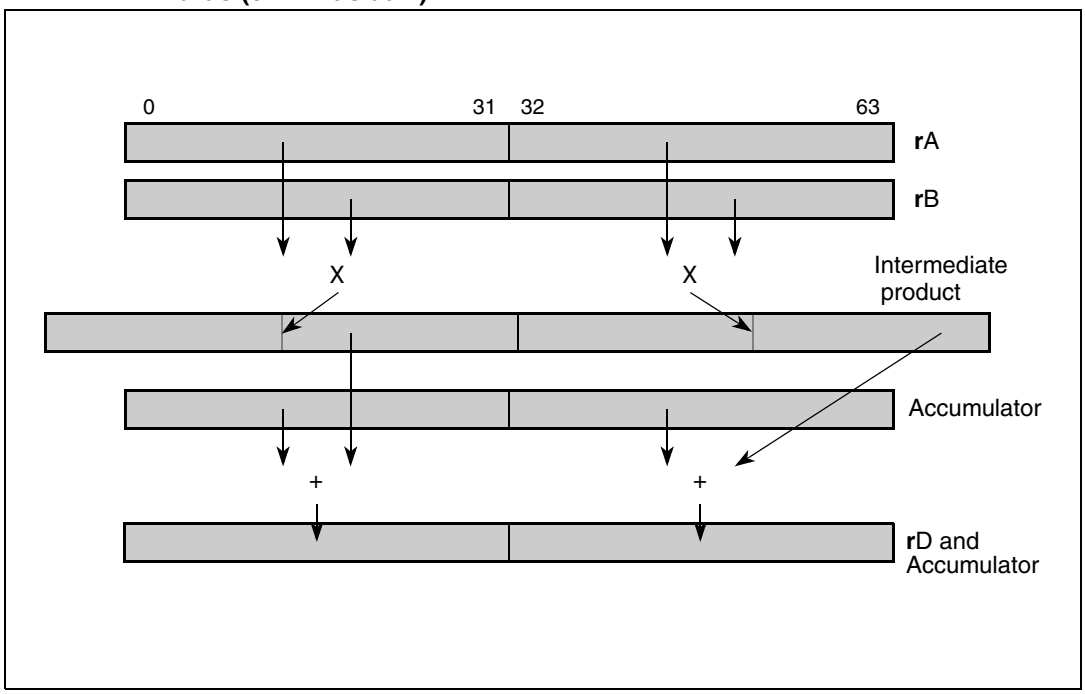
// update accumulator
$ACC_{0:63} \leftarrow rD_{0:63}$
// update SPEFSCR
$SPEFSCR_{OVH} \leftarrow ovh$
$SPEFSCR_{OV} \leftarrow ovl$
$SPEFSCR_{SOVH} \leftarrow SPEFSCR_{SOVH} \mid ovh$
$SPEFSCR_{SOV} \leftarrow SPEFSCR_{SOV} \mid ovl$

For each word element in the ACC, corresponding word unsigned integer elements in **r**A and **r**B are multiplied, producing a 64-bit product. The 32 lsbs of each product are added to the corresponding ACC word, saturating if overflow occurs; the result is placed in **r**D and the ACC.

If the addition causes overflow, the overflow and summary overflow bits are recorded in the SPEFSCR.

Other registers altered: SPEFSCR ACC

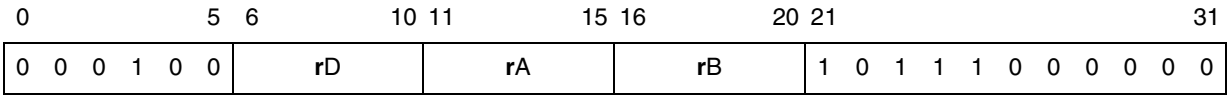
Figure 126. Vector multiply word low unsigned, saturate, integer & accumulate in words (evmwlusiaaw)



evmwlusianw | SPE APU | User | **evmwlusianw**

Vector multiply word low unsigned, saturate, integer and accumulate negative in words

evmwlusianw **r**D,**r**A,**r**B



// high
$temp_{0:63} \leftarrow rA_{0:31} \times_{ui} rB_{0:31}$
$temp_{0:63} \leftarrow EXTZ(ACC_{0:31}) - EXTZ(temp_{32:63})$
$ovh \leftarrow temp_{31}$
$rD_{0:31} \leftarrow SATURATE(ovh, 0, 0x0000\_0000, 0x0000\_0000, temp_{32:63})$

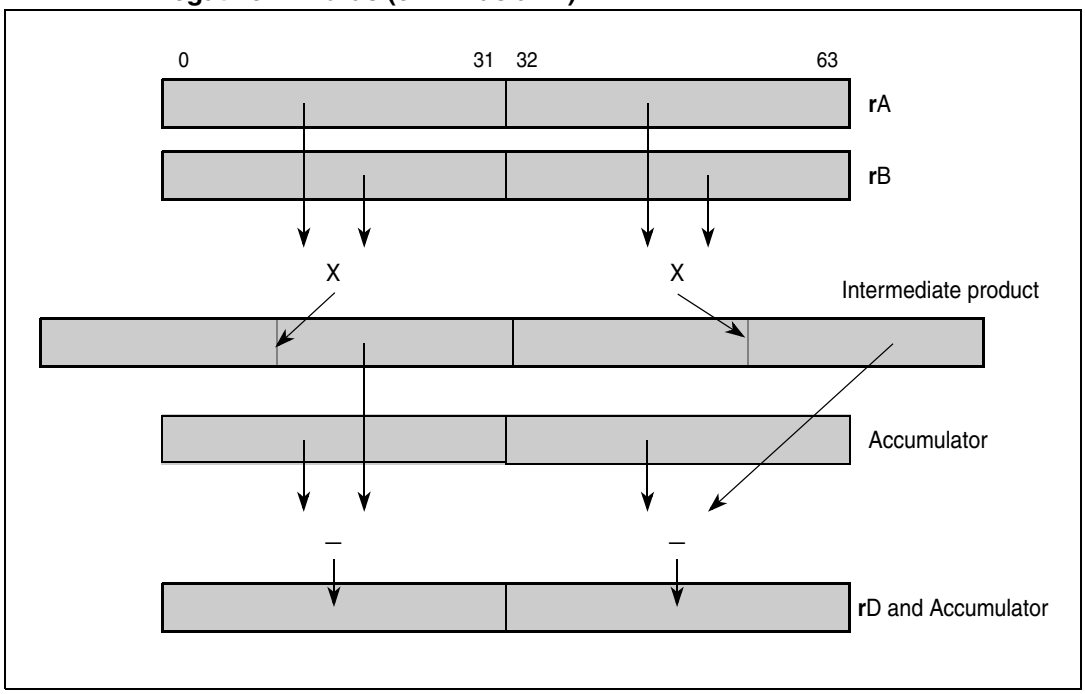
//low
$temp_{0:63} \leftarrow rA_{32:63} \times_{ui} rB_{32:63}$
$temp_{0:63} \leftarrow EXTZ(ACC_{32:63}) - EXTZ(temp_{32:63})$
$ovl \leftarrow temp_{31}$
$rD_{32:63} \leftarrow SATURATE(ovl, 0, 0x0000\_0000, 0x0000\_0000, temp_{32:63})$
// update accumulator
$ACC_{0:63} \leftarrow rD_{0:63}$
// update SPEFSCR
$SPEFSCR_{OVH} \leftarrow ovh$
$SPEFSCR_{OV} \leftarrow ovl$
$SPEFSCR_{SOVH} \leftarrow SPEFSCR_{SOVH} \mid ovh$
$SPEFSCR_{SOV} \leftarrow SPEFSCR_{SOV} \mid ovl$

For each ACC word element, corresponding word elements in **r**A and **r**B are multiplied producing a 64-bit product. The 32 lsbs of each product are subtracted from corresponding ACC words, saturating if underflow occurs; the result is placed in **r**D and the ACC.

If there is an underflow from the subtraction, the overflow and summary overflow bits are recorded in the SPEFSCR.

Other registers altered: SPEFSCR ACC

Figure 127. Vector multiply word low unsigned, saturate, integer & accumulate negative in words (evmwlusianw)



evmwsmf | SPE APU | User | **evmwsmf**

Vector multiply word signed, modulo, fractional (to accumulator)

evmwsmf **r**D,**r**A,**r**B **(A = 0)**

evmwsmfa **r**D,**r**A,**r**B **(A = 1)**

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	0	1	0	0	**r**D					**r**A					**r**B					1	0	0	0	1	A	1	1	0	1	1

$rD_{0:63} \leftarrow rA_{32:63} \times_{sf} rB_{32:63}$

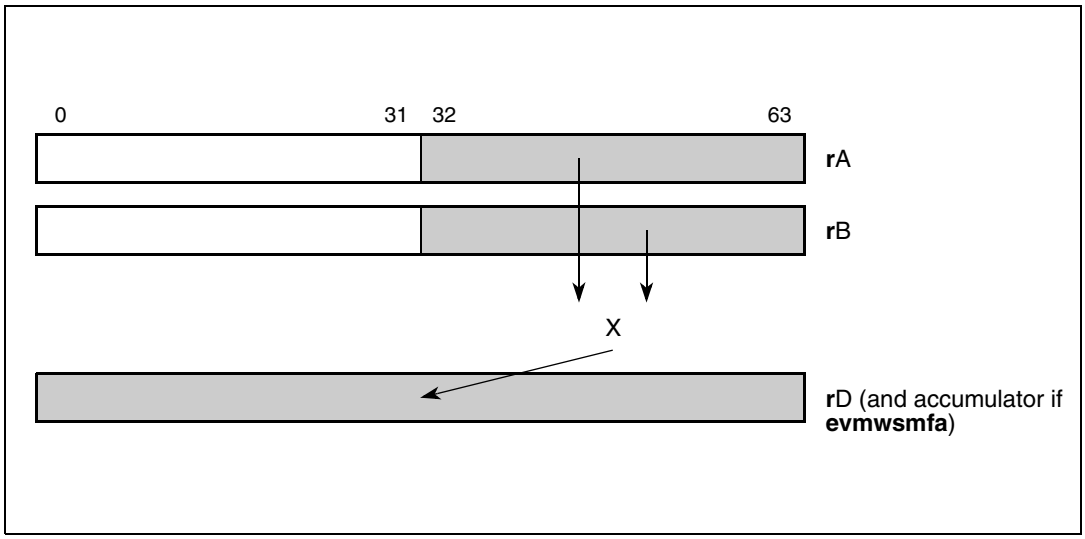
// update accumulator
if A = 1 then $ACC_{0:63} \leftarrow rD_{0:63}$

The corresponding low word signed fractional elements in **r**A and **r**B are multiplied. The product is placed into **r**D.

If A = 1, the result in **r**D is also placed into the accumulator.

Other registers altered: ACC (If A = 1)

Figure 128. Vector multiply word signed, modulo, fractional (to accumulator) (evmwsmf)



evmwsmfaa | SPE APU | User | **evmwsmfaa**

Vector multiply word signed, modulo, fractional and accumulate

evmwsmfaa **r**D,**r**A,**r**B

0	5	6	10	11	15	16	20	21	31
0 0 0 1 0 0		**r**D		**r**A		**r**B		1 0 1 0 1 0 1 1 0 1 1	

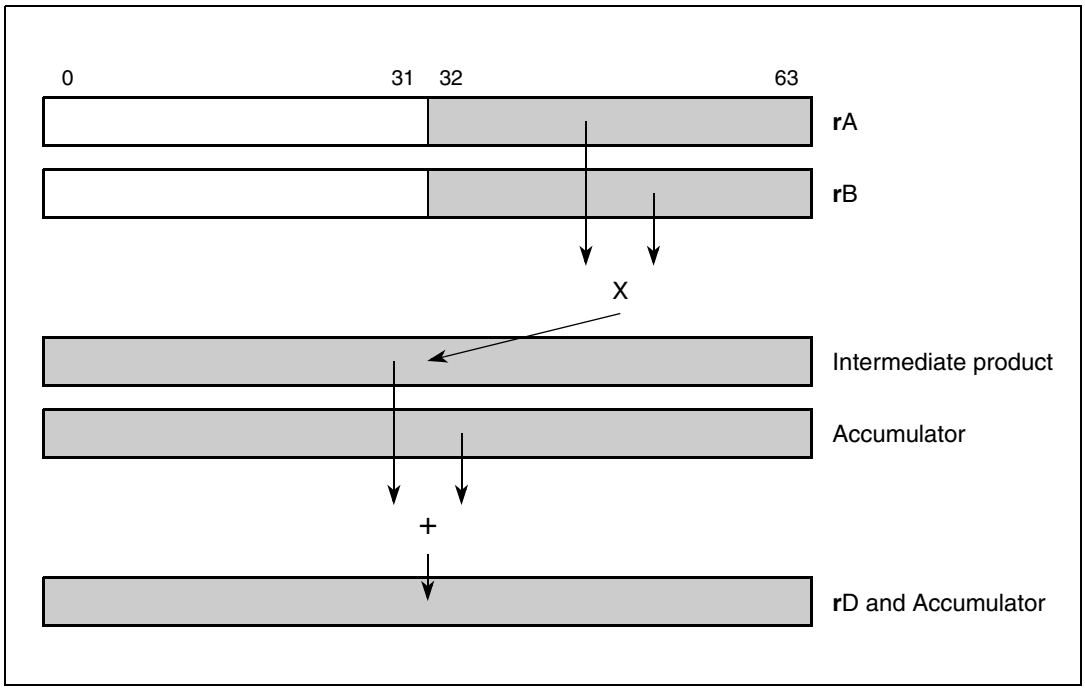
$temp_{0:63} \leftarrow rA_{32:63} \times_{sf} rB_{32:63}$
$rD_{0:63} \leftarrow ACC_{0:63} + temp_{0:63}$

// update accumulator
$ACC_{0:63} \leftarrow rD_{0:63}$

The corresponding low word signed fractional elements in **r**A and **r**B are multiplied. The intermediate product is added to the contents of the 64-bit accumulator and the result is placed in **r**D and the accumulator.

Other registers altered: ACC

Figure 129. Vector multiply word signed, modulo, fractional & accumulate (evmwsmfaa)



evmwsmfan | SPE APU | User | **evmwsmfan**

Vector multiply word signed, modulo, fractional and accumulate negative

evmwsmfan **r**D,**r**A,**r**B

0	5	6	10	11	15	16	20	21	31
0 0 0 1 0 0		**r**D		**r**A		**r**B		1 0 1 1 1 0 1 1 0 1 1	

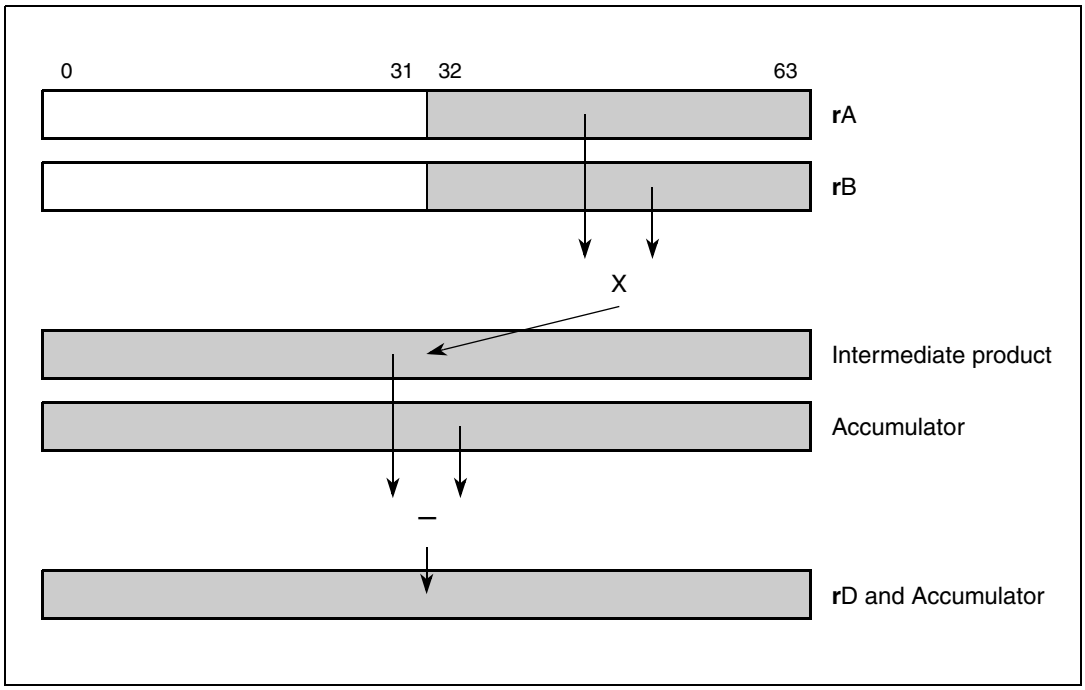
$temp_{0:63} \leftarrow rA_{32:63} \times_{sf} rB_{32:63}$
$rD_{0:63} \leftarrow ACC_{0:63} - temp_{0:63}$

// update accumulator
$ACC_{0:63} \leftarrow rD_{0:63}$

The corresponding low word signed fractional elements in **r**A and **r**B are multiplied. The intermediate product is subtracted from the contents of the accumulator and the result is placed in **r**D and the accumulator.

Other registers altered: ACC

Figure 130. Vector multiply word signed, modulo, fractional & accumulate negative (evmwsmfan)



evmwsmi | SPE APU | User | **evmwsmi**

Vector multiply word signed, modulo, integer (to accumulator)

evmwsmi **r**D,**r**A,**r**B **(A = 0)**

evmwsmia **r**D,**r**A,**r**B **(A = 1)**

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	0	1	0	0	**r**D					**r**A					**r**B					1	0	0	0	1	A	1	1	0	0	1

$rD_{0:63} \leftarrow rA_{32:63} \times_{si} rB_{32:63}$

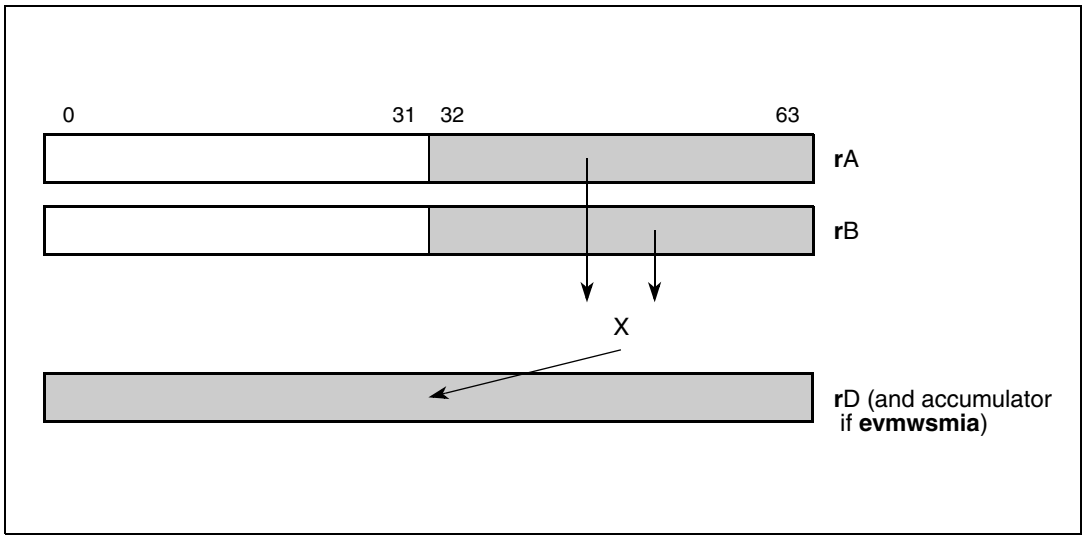
// update accumulator
if A = 1 then $ACC_{0:63} \leftarrow rD_{0:63}$

The low word signed integer elements in **r**A and **r**B are multiplied. The product is placed into **r**D.

If A = 1, the result in **r**D is also placed into the accumulator.

Other registers altered: ACC (If A = 1)

Figure 131. Vector multiply word signed, modulo, integer (to accumulator) (evmwsmi)



evmwsmiaa | SPE APU | User | **evmwsmiaa**

Vector multiply word signed, modulo, integer and accumulate

evmwsmiaa **r**D,**r**A,**r**B

0	5	6	10	11	15	16	20	21	31
0 0 0 1 0 0		**r**D		**r**A		**r**B		1 0 1 0 1 0 1 1 0 0 1	

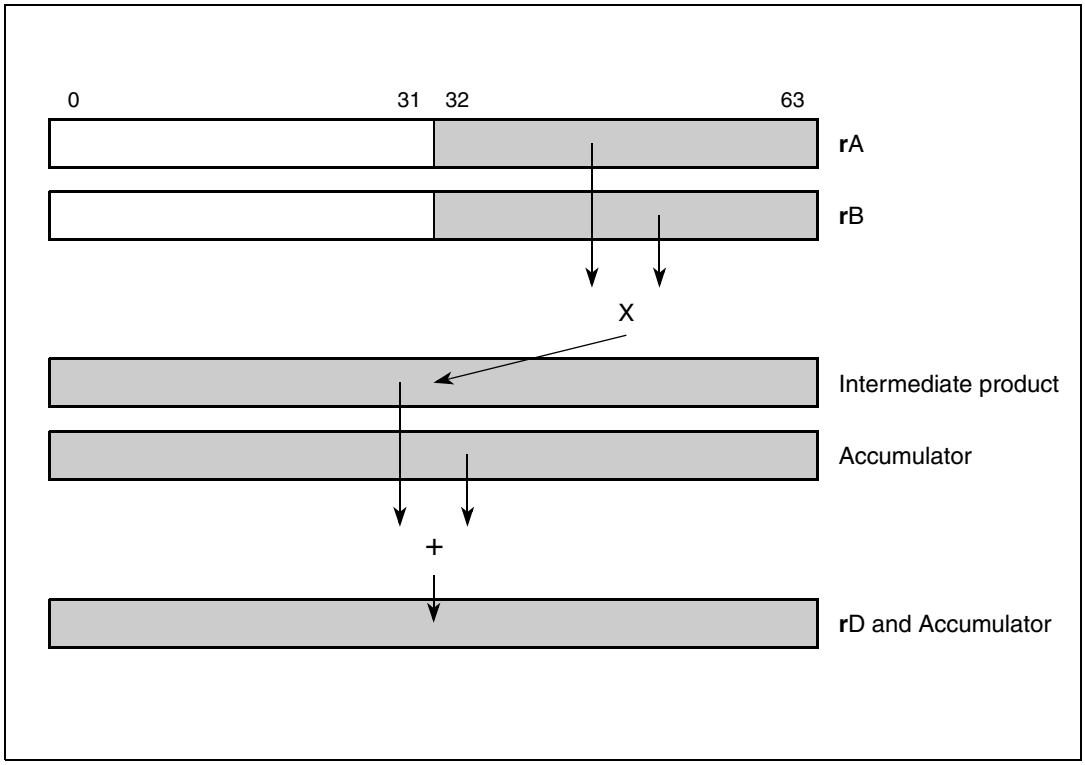
$temp_{0:63} \leftarrow rA_{32:63} \times_{si} rB_{32:63}$
$rD_{0:63} \leftarrow ACC_{0:63} + temp_{0:63}$

// update accumulator
$ACC_{0:63} \leftarrow rD_{0:63}$

The low word signed integer elements in **r**A and **r**B are multiplied. The intermediate product is added to the contents of the 64-bit accumulator and the result is placed into **r**D and the accumulator.

Other registers altered: ACC

Figure 132. Vector multiply word signed, modulo, integer & accumulate (evmwsmiaa)



evmwsmian | SPE APU | User | **evmwsmian**

Vector multiply word signed, modulo, integer and accumulate negative

evmwsmian **r**D,**r**A,**r**B

0	5	6	10	11	15	16	20	21	31
0 0 0 1 0 0		**r**D		**r**A		**r**B		1 0 1 1 1 0 1 1 0 0 1	

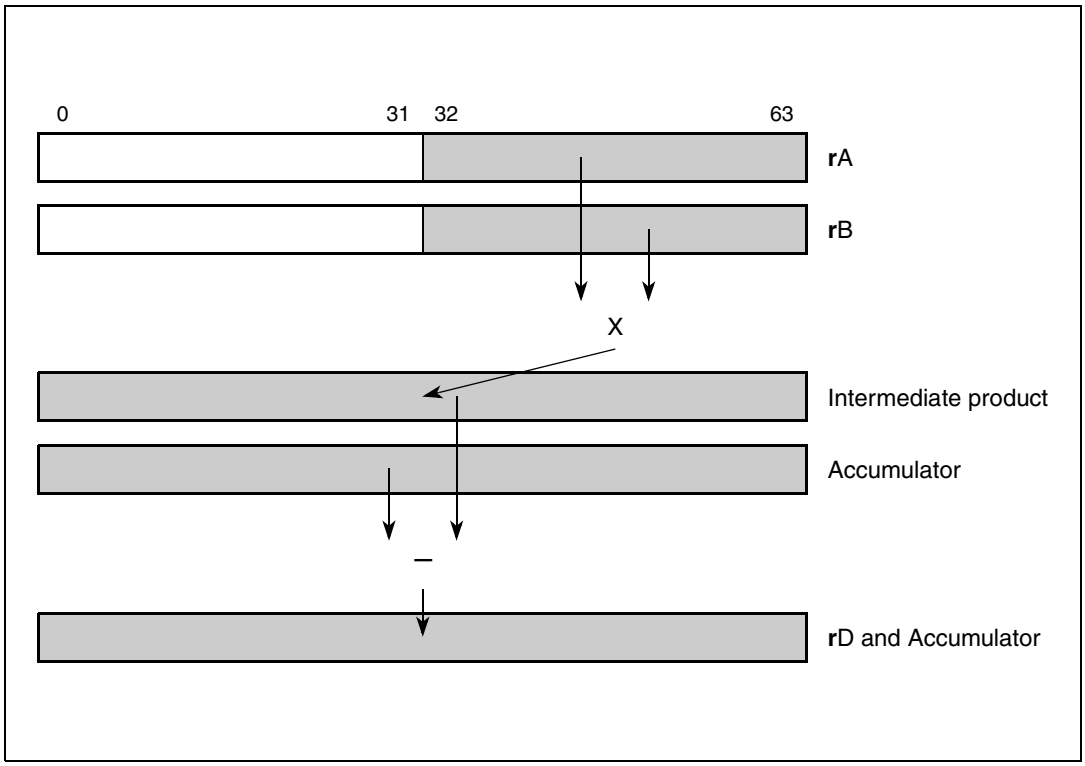
$temp_{0:63} \leftarrow rA_{32:63} \times_{si} rB_{32:63}$
$rD_{0:63} \leftarrow ACC_{0:63} - temp_{0:63}$

// update accumulator
$ACC_{0:63} \leftarrow rD_{0:63}$

The low word signed integer elements in **r**A and **r**B are multiplied. The intermediate product is subtracted from the contents of the 64-bit accumulator and the result is placed into **r**D and the accumulator.

Other registers altered: ACC

Figure 133. Vector multiply word signed, modulo, integer & accumulate negative (evmwsmian)



evmwssf | SPE APU | User | **evmwssf**

Vector multiply word signed, saturate, fractional (to accumulator)

evmwssf **r**D,**r**A,**r**B (A = 0)

evmwssfa **r**D,**r**A,**r**B (A = 1)

0	5	6	10	11	15	16	20	21	25	26	27	31
0 0 0 1 0 0		**r**D		**r**A		**r**B		1 0 0 0 1		A	1 0 0 1 1	

$temp_{0:63} \leftarrow rA_{32:63} \times_{sf} rB_{32:63}$
if ($rA_{32:63}$ = 0x8000_0000) & ($rB_{32:63}$ = 0x8000_0000) then
$rD_{0:63} \leftarrow$ 0x7FFF_FFFF_FFFF_FFFF //saturate
mov $\leftarrow$ 1
else
$rD_{0:63} \leftarrow temp_{0:63}$
mov $\leftarrow$ 0

// update accumulator
if A = 1 then $ACC_{0:63} \leftarrow rD_{0:63}$

// update SPEFSCR
$SPEFSCR_{OVH} \leftarrow 0$
$SPEFSCR_{OV} \leftarrow$ mov
$SPEFSCR_{SOV} \leftarrow SPEFSCR_{SOV}$ | mov

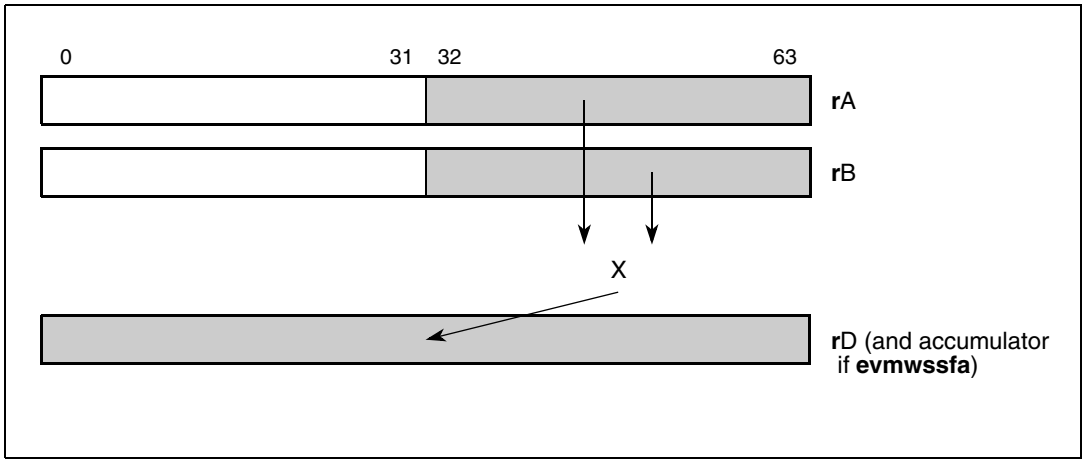
The low word signed fractional elements in **r**A and **r**B are multiplied. The 64 bit product is placed into **r**D. If both inputs are –1.0, the result saturates to the largest positive signed fraction and the overflow and summary overflow bits are recorded in the SPEFSCR.

The architecture specifies that if the final result cannot be represented in 64 bits, SPEFSCR[OV] should be set (along with the SOV bit, if it is not already set).

If A = 1, the result in **r**D is also placed into the accumulator.

Other registers altered: SPEFSCR ACC (If A = 1)

Figure 134. Vector multiply word signed, saturate, fractional (to accumulator) (evmwssf)



evmwssfaa | SPE APU | User | **evmwssfaa**

Vector multiply word signed, saturate, fractional and accumulate

evmwssfaa **r**D,**r**A,**r**B

0	5	6	10	11	15	16	20	21	31
0 0 0 1 0 0		**r**D		**r**A		**r**B		1 0 1 0 1 0 1 0 0 1 1	

$temp_{0:63} \leftarrow rA_{32:63} \times_{sf} rB_{32:63}$
if ($rA_{32:63}$ = 0x8000_0000) & ($rB_{32:63}$ = 0x8000_0000) then
$temp_{0:63} \leftarrow$ 0x7FFF_FFFF_FFFF_FFFF //saturate
mov $\leftarrow$ 1
else
mov $\leftarrow$ 0
$temp_{0:64} \leftarrow EXTS(ACC_{0:63}) + EXTS(temp_{0:63})$
ov $\leftarrow (temp_0 \oplus temp_1)$
$rD_{0:63} \leftarrow temp_{1:64}$)
// update accumulator
$ACC_{0:63} \leftarrow rD_{0:63}$
// update SPEFSCR
$SPEFSCR_{OVH} \leftarrow 0$
$SPEFSCR_{OV} \leftarrow$ mov
$SPEFSCR_{SOV} \leftarrow SPEFSCR_{SOV}$ | ov | mov

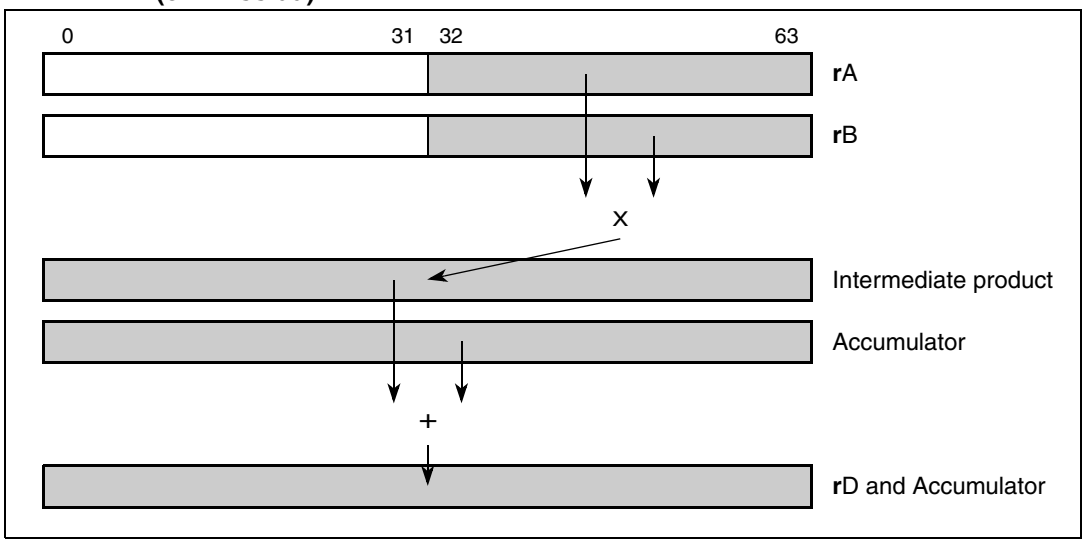
The low word signed fractional elements in **r**A and **r**B are multiplied producing a 64-bit product. If both inputs are –1.0, the product saturates to the largest positive signed fraction. The 64-bit product is added to the ACC and the result is placed in **r**D and the ACC.

If there is an overflow from either the multiply or the addition, the SPEFSCR overflow and summary overflow bits are recorded.

Note: There is no saturation on the addition with the accumulator.

Other registers altered: SPEFSCR ACC

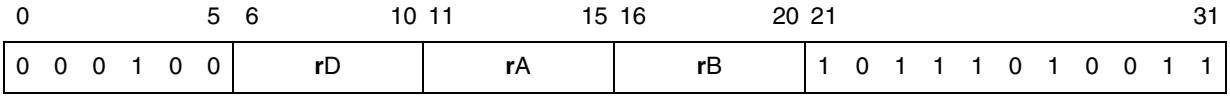
Figure 135. Vector multiply word signed, saturate, fractional, & accumulate (evmwssfaa)



evmwssfan | SPE APU | User | **evmwssfan**

Vector multiply word signed, saturate, fractional and accumulate negative

evmwssfan **r**D,**r**A,**r**B



$temp_{0:63} \leftarrow rA_{32:63} \times_{sf} rB_{32:63}$
if $(rA_{32:63} = 0x8000\_0000)$ & $(rB_{32:63} = 0x8000\_0000)$ then
$temp_{0:63} \leftarrow 0x7FFF\_FFFF\_FFFF\_FFFF$ //saturate
$mov \leftarrow 1$
else
$mov \leftarrow 0$
$temp_{0:64} \leftarrow EXTS(ACC_{0:63}) - EXTS(temp_{0:63})$
$ov \leftarrow (temp_0 \oplus temp_1)$
$rD_{0:63} \leftarrow temp_{1:64}$)
// update accumulator
$ACC_{0:63} \leftarrow rD_{0:63}$
// update SPEFSCR
$SPEFSCR_{OVH} \leftarrow 0$
$SPEFSCR_{OV} \leftarrow mov$
$SPEFSCR_{SOV} \leftarrow SPEFSCR_{SOV} \mid ov \mid mov$

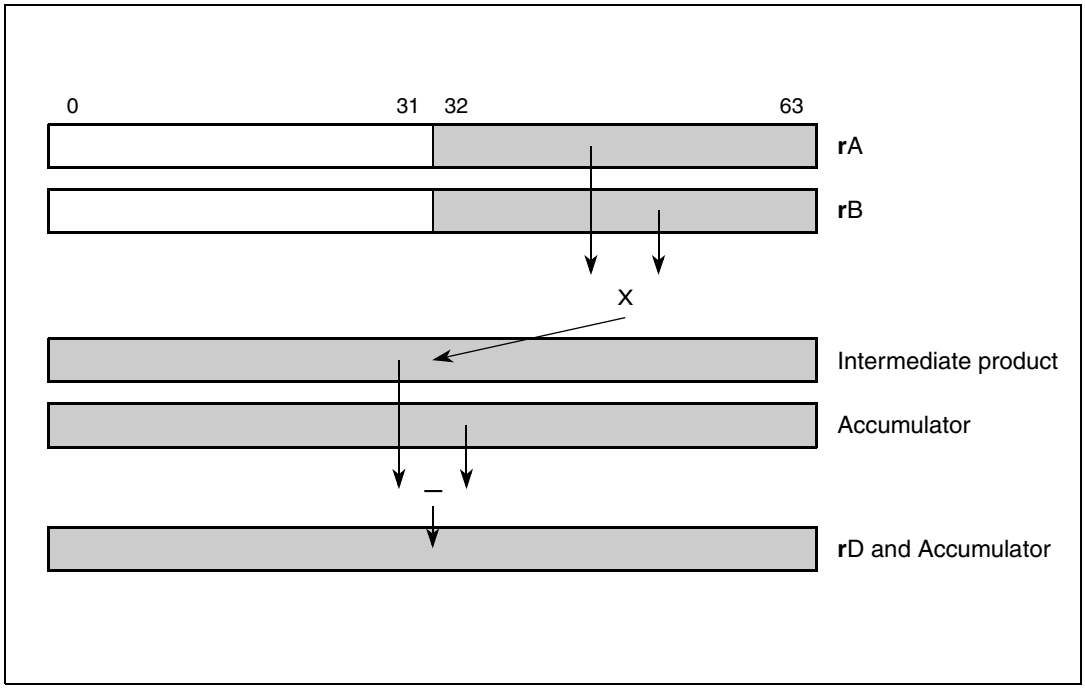
The low word signed fractional elements in **r**A and **r**B are multiplied producing a 64-bit product. If both inputs are –1.0, the product saturates to the largest positive signed fraction. The 64-bit product is subtracted from the ACC and the result is placed in **r**D and the ACC.

If there is an overflow from either the multiply or the addition, the SPEFSCR overflow and summary overflow bits are recorded.

Note: There is no saturation on the subtraction with the accumulator.

Other registers altered: SPEFSCR ACC

Figure 136. Vector multiply word signed, saturate, fractional & accumulate negative (evmwssfan)



evmwumi | SPE APU | User | **evmwumi**

Vector multiply word unsigned, modulo, integer (to accumulator)

evmwumi **r**D,**r**A,**r**B **(A = 0)**
evmwumia **r**D,**r**A,**r**B **(A = 1)**

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	0	1	0	0	**r**D					**r**A					**r**B					1	0	0	0	1	A	1	1	0	0	0

$rD_{0:63} \leftarrow rA_{32:63} \times_{ui} rB_{32:63}$

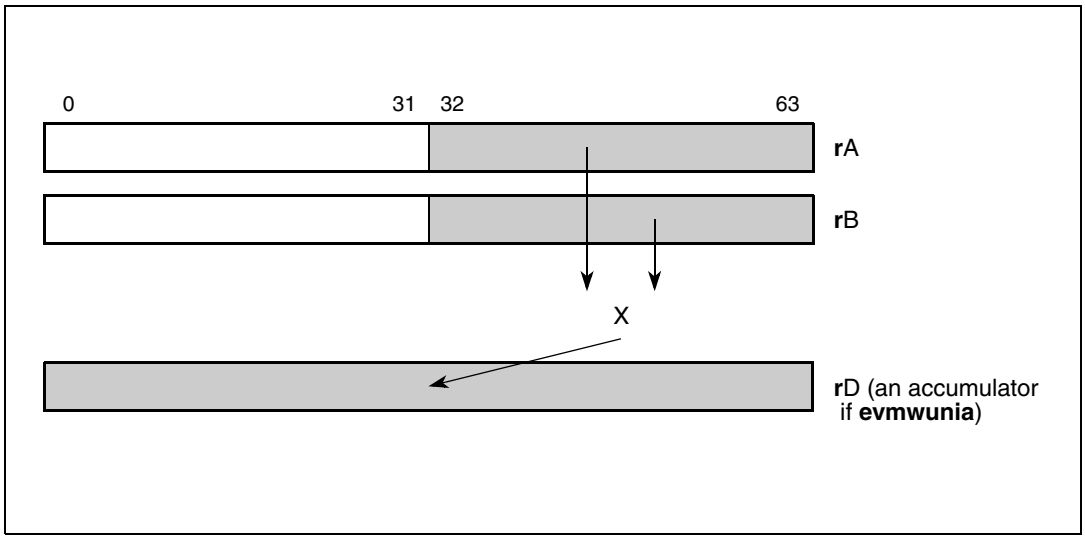
// update accumulator
if A = 1 then $ACC_{0:63} \leftarrow rD_{0:63}$

The low word unsigned integer elements in **r**A and **r**B are multiplied to form a 64-bit product that is placed into **r**D.

If A = 1, the result in **r**D is also placed into the accumulator.

Other registers altered: ACC (If A = 1)

Figure 137. Vector multiply word unsigned, modulo, integer (to accumulator) (evmwumi)



evmwumiaa | SPE APU | User | **evmwumiaa**

Vector multiply word unsigned, modulo, integer and accumulate

evmwumiaa **r**D,**r**A,**r**B

0	5	6	10	11	15	16	20	21	31
0 0 0 1 0 0		**r**D		**r**A		**r**B		1 0 1 0 1 0 1 1 0 0 0	

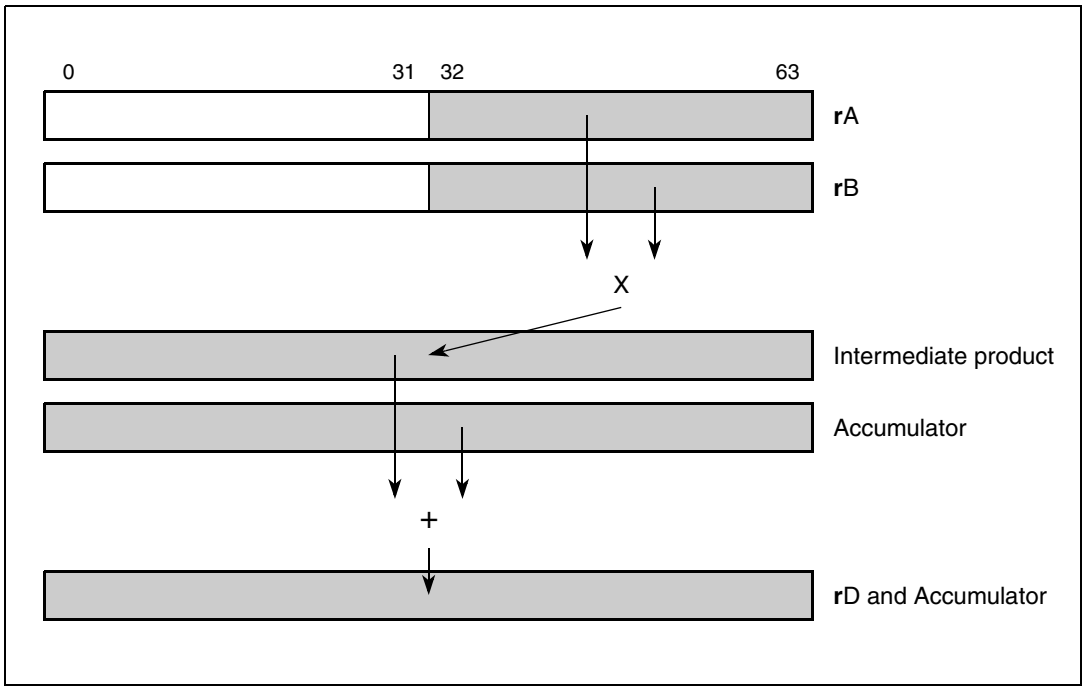
$temp_{0:63} \leftarrow rA_{32:63} \times_{ui} rB_{32:63}$
$rD_{0:63} \leftarrow ACC_{0:63} + temp_{0:63}$

// update accumulator
$ACC_{0:63} \leftarrow rD_{0:63}$

The low word unsigned integer elements in **r**A and **r**B 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 into **r**D.

Other registers altered: ACC

Figure 138. Vector multiply word unsigned, modulo, integer & accumulate (evmwumiaa)



evmwumian | SPE APU | User | **evmwumian**

Vector multiply word unsigned, modulo, integer and accumulate negative

evmwumian **r**D,**r**A,**r**B

0	5	6	10	11	15	16	20	21	31
0 0 0 1 0 0		**r**D		**r**A		**r**B		1 0 1 1 1 0 1 1 0 0 0	

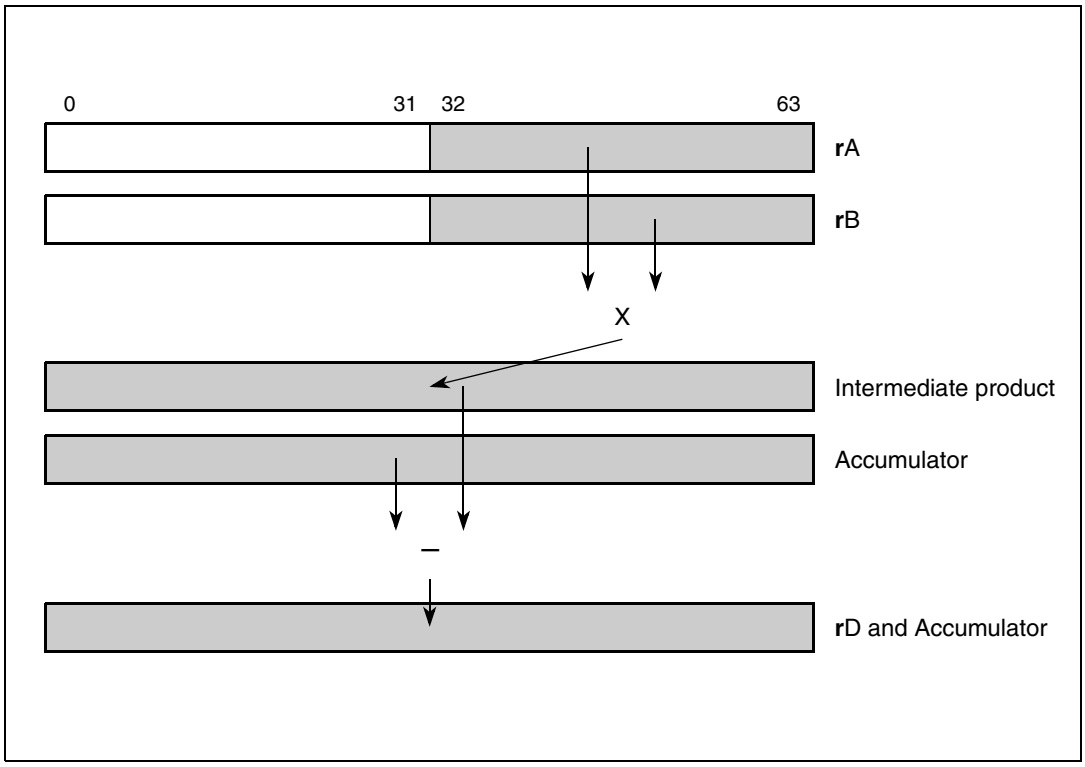
$temp_{0:63} \leftarrow rA_{32:63} \times_{ui} rB_{32:63}$
$rD_{0:63} \leftarrow ACC_{0:63} - temp_{0:63}$

// update accumulator
$ACC_{0:63} \leftarrow rD_{0:63}$

The low word unsigned integer elements in **r**A and **r**B 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 into **r**D.

Other registers altered: ACC

Figure 139. Vector multiply word unsigned, modulo, integer & accumulate negative (evmwumian)



evnand | SPE APU | User | **evnand**

Vector NAND

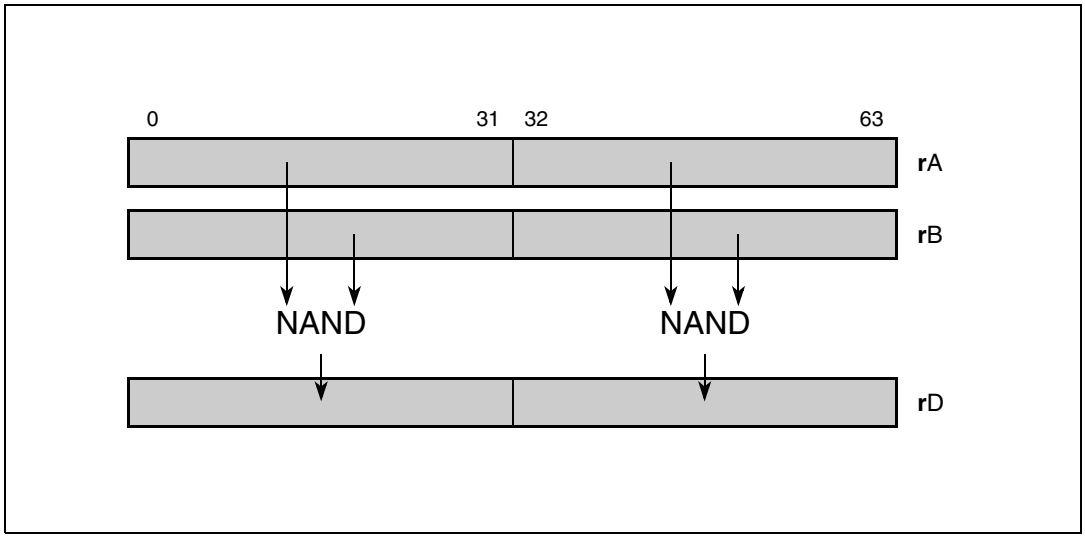
evnand **r**D,**r**A,**r**B

0	5	6	10	11	15	16	20	21	31
0 0 0 1 0 0		**r**D		**r**A		**r**B		0 1 0 0 0 0 1 1 1 1 0	

$rD_{0:31} \leftarrow \neg(rA_{0:31}$ & $rB_{0:31})$ // Bitwise NAND
$rD_{32:63} \leftarrow \neg(rA_{32:63}$ & $rB_{32:63})$ // Bitwise NAND

Corresponding word elements of **r**A and **r**B are bitwise NANDed. The result is placed in the corresponding element of **r**D.

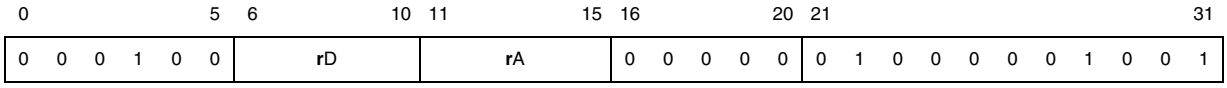
Figure 140. Vector NAND (evnand)



evneg | SPE APU | User | **evneg**

Vector negate

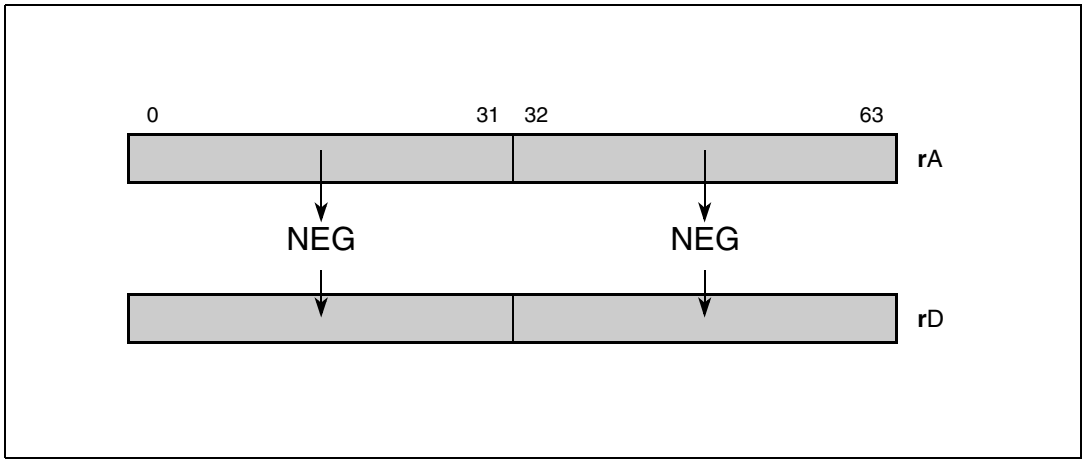
evneg **r**D,**r**A



$rD_{0:31} \leftarrow NEG(rA_{0:31})$
$rD_{32:63} \leftarrow NEG(rA_{32:63})$

The negative of each element of **r**A is placed in **r**D. The negative of 0x8000_0000 (most negative number) returns 0x8000_0000. No overflow is detected.

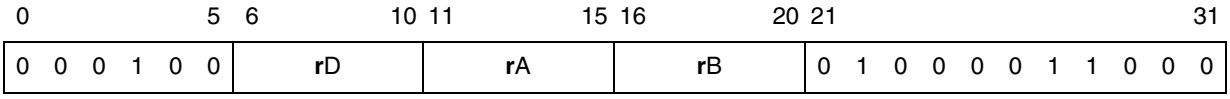
Figure 141. Vector negate (evneg)



evnor | SPE APU | User | **evnor**

Vector NOR

evnor **r**D,**r**A,**r**B

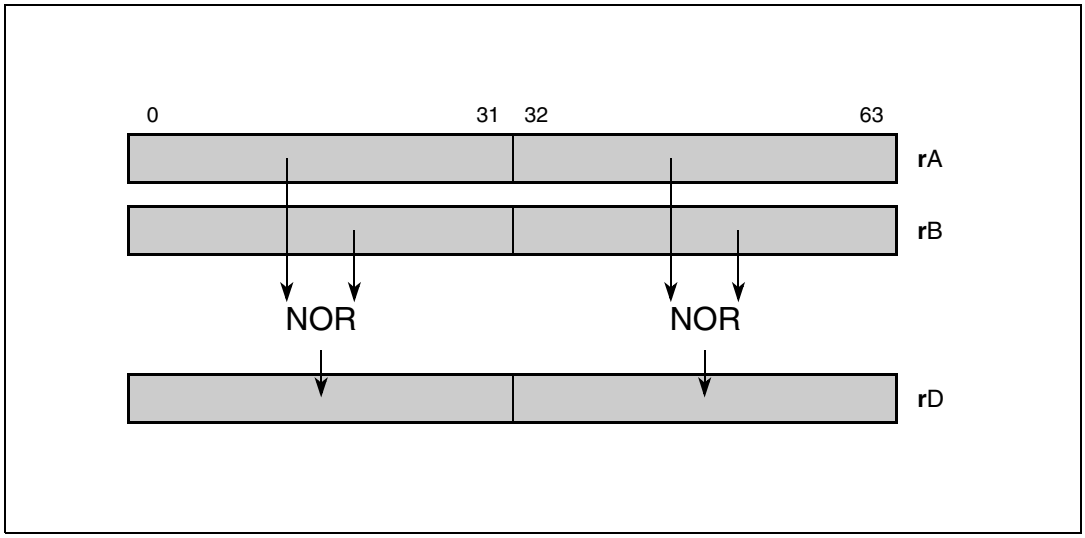


$rD_{0:31} \leftarrow \neg(rA_{0:31} \mid rB_{0:31})$ // Bitwise NOR
$rD_{32:63} \leftarrow \neg(rA_{32:63} \mid rB_{32:63})$ // Bitwise NOR

Each element of **r**A and **r**B is bitwise NORed. The result is placed in the corresponding element of **r**D.

*Note: Use **evnand** or **evnor** for **evnot**.*

Figure 142. Vector NOR (evnor)



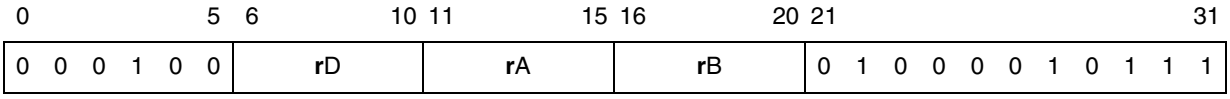
Simplified mnemonic: **evnot** **r**D,**r**A performs a complement register

evnot **r**D,**r**A equivalent to**evnor** **r**D,**r**A,**r**A

evor | SPE APU | User | **evor**

Vector OR

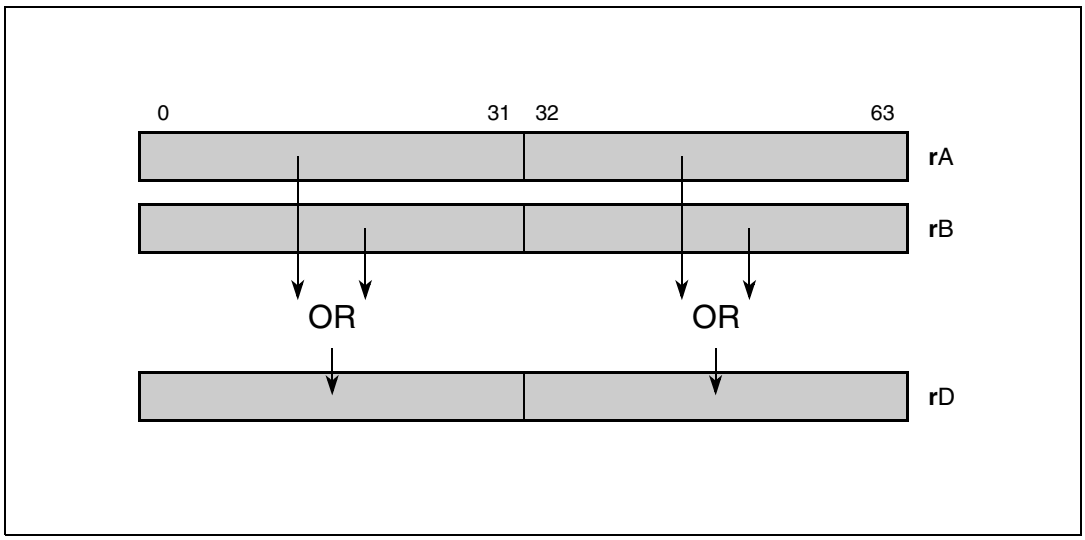
evor **r**D,**r**A,**r**B



$rD_{0:31} \leftarrow rA_{0:31} \mid rB_{0:31}$ //Bitwise OR
$rD_{32:63} \leftarrow rA_{32:63} \mid rB_{32:63}$ // Bitwise OR

Each element of **r**A and **r**B is bitwise ORed. The result is placed in the corresponding element of **r**D.

Figure 143. Vector OR (evor)



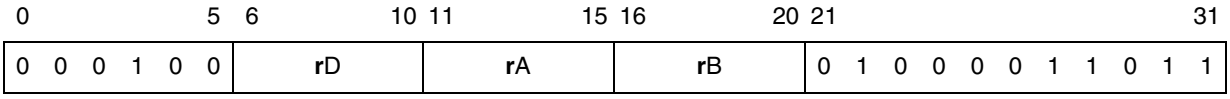
Simplified mnemonic: **evmr** **r**D,**r**A handles moving of the full 64-bit SPE register.

evmr **r**D,**r**A equivalent to **evor** **r**D,**r**A,**r**A

evorc | SPE APU | User | **evorc**

Vector OR with complement

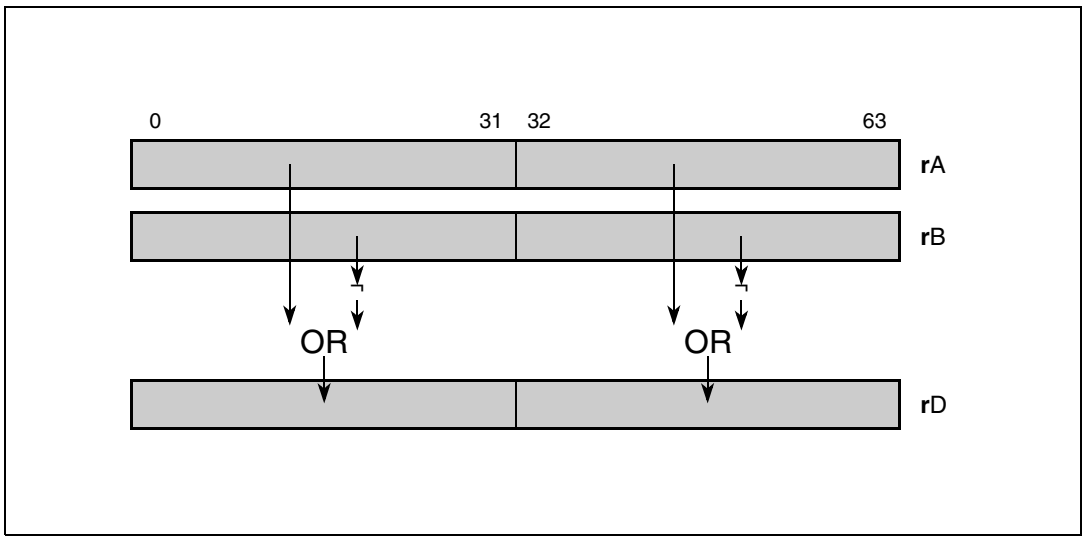
evorc **r**D,**r**A,**r**B



$rD_{0:31} \leftarrow rA_{0:31} \mid (\neg rB_{0:31})$ // Bitwise ORC
$rD_{32:63} \leftarrow rA_{32:63} \mid (\neg rB_{32:63})$ // Bitwise ORC

Each element of **r**A is bitwise ORed with the complement of **r**B. The result is placed in the corresponding element of **r**D.

Figure 144. Vector OR with complement (evorc)



evrlw | SPE APU | User | **evrlw**

Vector rotate left word

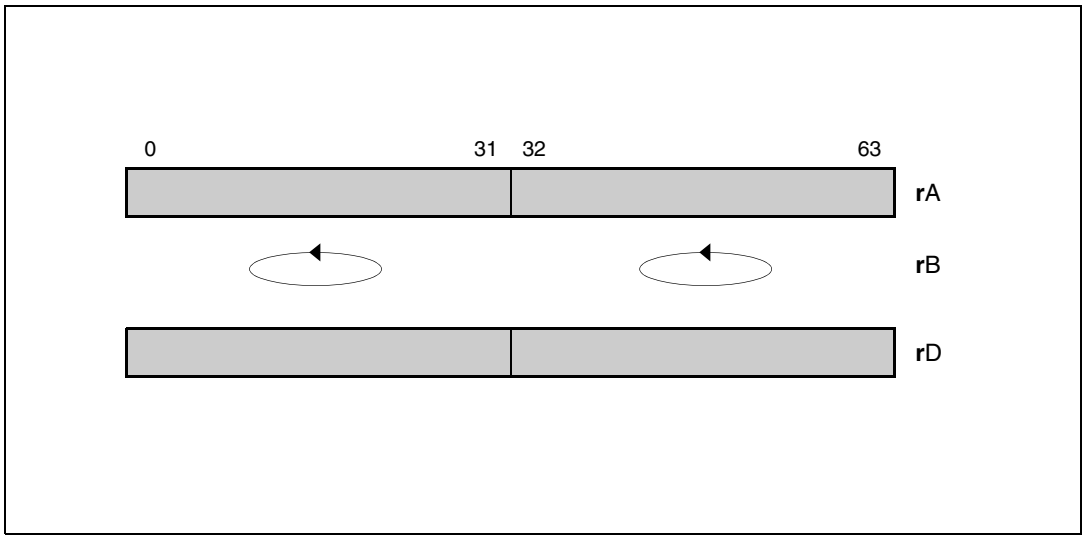
evrlw **r**D,**r**A,**r**B

0	1	2	3	4	5	6	10	11	15	16	20	21	31
0	0	0	1	0	0	**r**D		**r**A		**r**B		0 1 0 0 0 1 0 1 0 0 0	

nh ← $rB_{27:31}$
nl ← $rB_{59:63}$
$rD_{0:31}$ ← ROTL($rA_{0:31}$, nh)
$rD_{32:63}$ ← ROTL($rA_{32:63}$, nl)

Each of the high and low elements of **r**A is rotated left by an amount specified in **r**B. The result is placed into **r**D. Rotate values for each element of **r**A are found in bit positions **r**B[27–31] and **r**B[59–63].

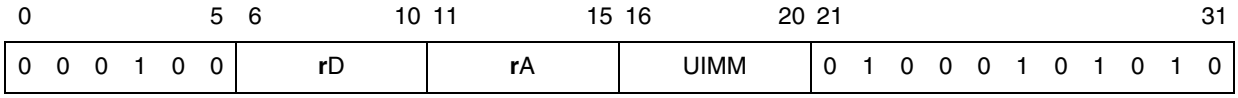
Figure 145. Vector rotate left word (evrlw)



evrlwi | SPE APU | User | **evrlwi**

Vector rotate left word immediate

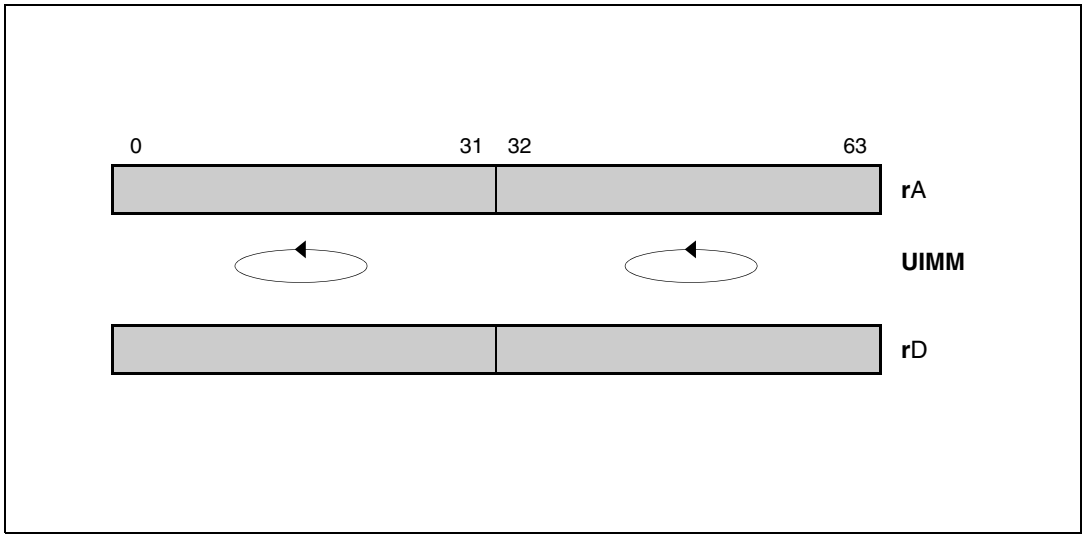
evrlwi **r**D,**r**A,UIMM



n ← UIMM
$rD_{0:31}$ ← ROTL($rA_{0:31}$, n)
$rD_{32:63}$ ← ROTL($rA_{32:63}$, n)

Both the high and low elements of **r**A are rotated left by an amount specified by a 5-bit immediate value.

Figure 146. Vector rotate left word immediate (evrlwi)



evrndw | SPE APU | User | **evrndw**

Vector round word

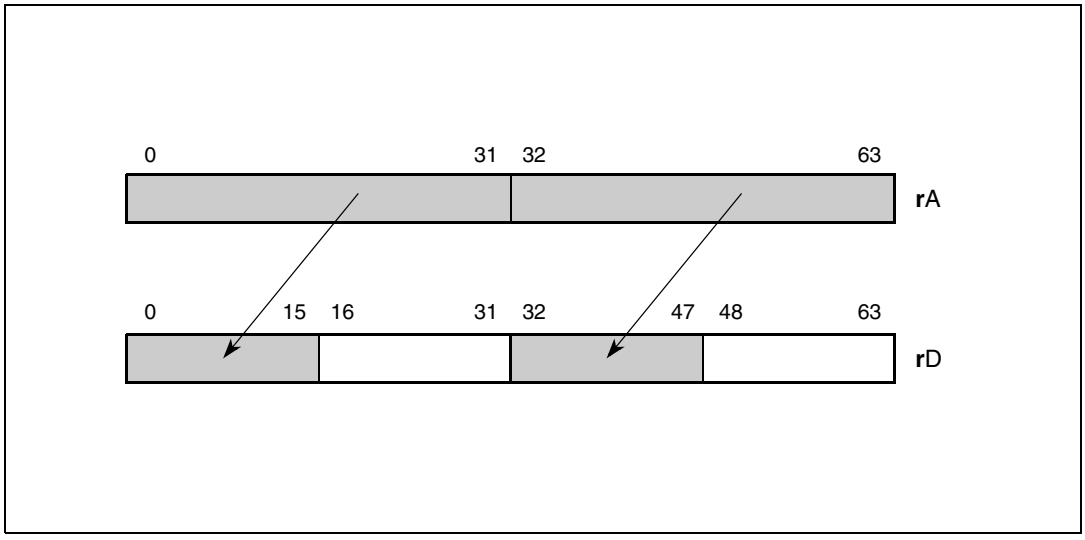
evrndw **r**D,**r**A

0	5	6	10	11	15	16	20	21	31
0 0 0 1 0 0		**r**D		**r**A		0 0 0 0 0		0 1 0 0 0 0 0 1 1 0 0	

$rD_{0:31} \leftarrow (rA_{0:31}+0x00008000)$ & 0xFFFF0000 // Modulo sum
$rD_{32:63} \leftarrow (rA_{32:63}+0x00008000)$ & 0xFFFF0000 // Modulo sum

The 32-bit elements of **r**A are rounded into 16 bits. The result is placed into **r**D. The resulting 16 bits are placed in the most significant 16 bits of each element of **r**D, zeroing out the low order 16 bits of each element.

Figure 147. Vector round word (evrndw)



evsel | SPE APU | User | **evsel**

Vector select

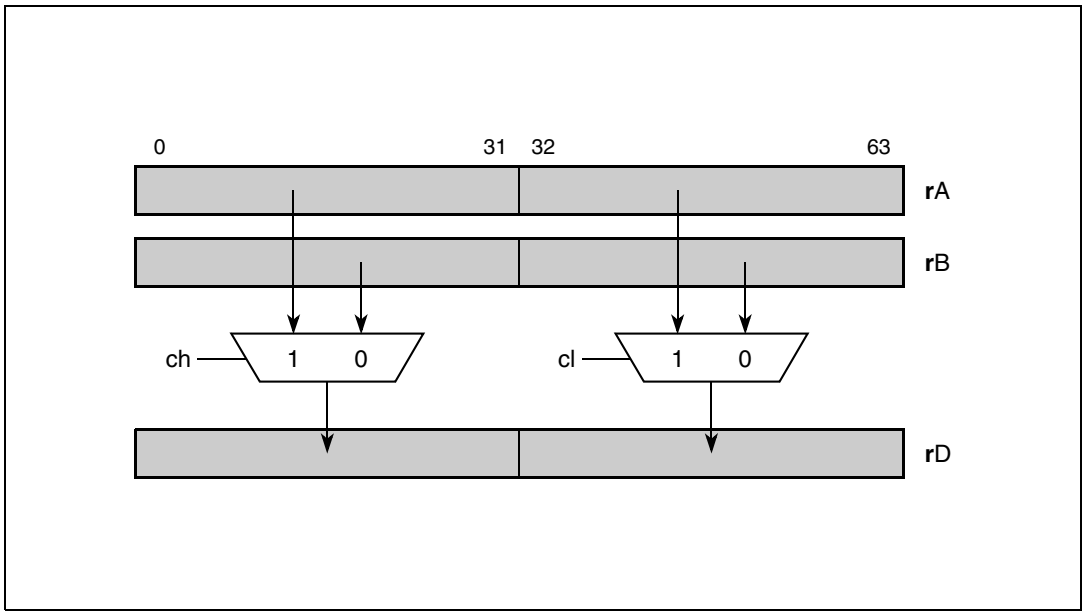
evsel **r**D,**r**A,**r**B,**cr**S

0	1	2	3	4	5	6–10	11–15	16–20	21	22	23	24	25	26	27	28	29–31
0	0	0	1	0	0	**r**D	**r**A	**r**B	0	1	0	0	1	1	1	1	**cr**S

ch ← CR_{crS*4}
cl ← $CR_{crS*4+1}$
if (ch = 1) then $rD_{0:31} \leftarrow rA_{0:31}$
else $rD_{0:31} \leftarrow rB_{0:31}$
if (cl = 1) then $rD_{32:63} \leftarrow rA_{32:63}$
else $rD_{32:63} \leftarrow rB_{32:63}$

If the most significant bit in the **cr**S field of CR is set, the high-order element of **r**A is placed in the high-order element of **r**D; otherwise, the high-order element of **r**B is placed into the high-order element of **r**D. If the next most significant bit in the **cr**S field of CR is set, the low-order element of **r**A is placed in the low-order element of **r**D, otherwise, the low-order element of **r**B is placed into the low-order element of **r**D. This is shown in *Figure 148*.

Figure 148. Vector select (evsel)



evslw | SPE APU | User | **evslw**

Vector shift left word

evslw **r**D,**r**A,**r**B

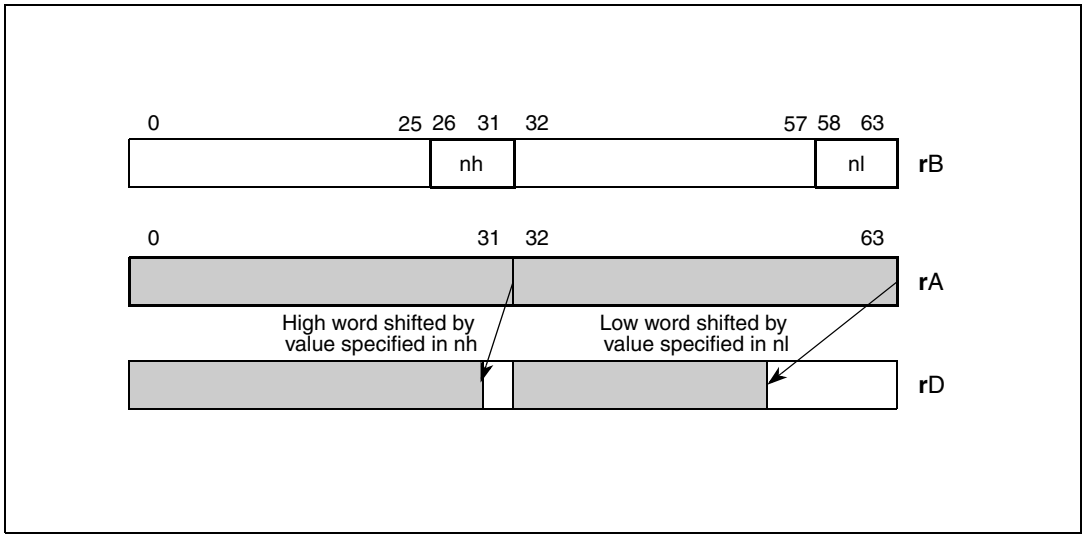
0	5	6	10	11	15	16	20	21	31
0 0 0 1 0 0		**r**D		**r**A		**r**B		0 1 0 0 0 1 0 0 1 0 0	

$nh \leftarrow rB_{26:31}$
$nl \leftarrow rB_{58:63}$
$rD_{0:31} \leftarrow SL(rA_{0:31}, nh)$
$rD_{32:63} \leftarrow SL(rA_{32:63}, nl)$

Each of the high and low elements of **r**A are shifted left by an amount specified in **r**B. The result is placed into **r**D. The separate shift amounts for each element are specified by 6 bits in **r**B that lie in bit positions 26–31 and 58–63.

Shift amounts from 32 to 63 give a zero result.

Figure 149. Vector shift left word (evslw)



evslwi | SPE APU | User | **evslwi**

Vector shift left word immediate

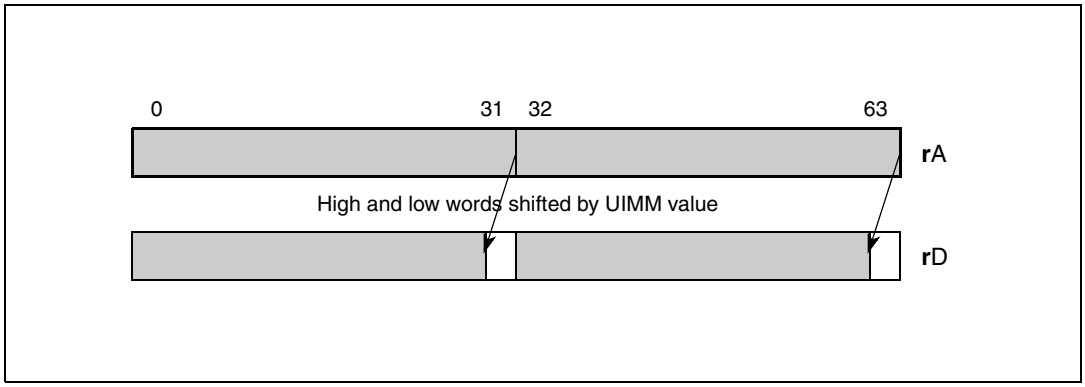
evslwi **r**D,**r**A,UIMM

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	0	1	0	0	**r**D					**r**A					UIMM					0	1	0	0	0	1	0	0	1	1	0

n ← UIMM
$rD_{0:31} \leftarrow SL(rA_{0:31}, n)$
$rD_{32:63} \leftarrow SL(rA_{32:63}, n)$

Both high and low elements of **r**A are shifted left by the 5-bit UIMM value and the results are placed in **r**D.

Figure 150. Vector shift left word immediate (evslwi)



evsplatfi | SPE APU | User | **evsplatfi**

Vector splat fractional immediate

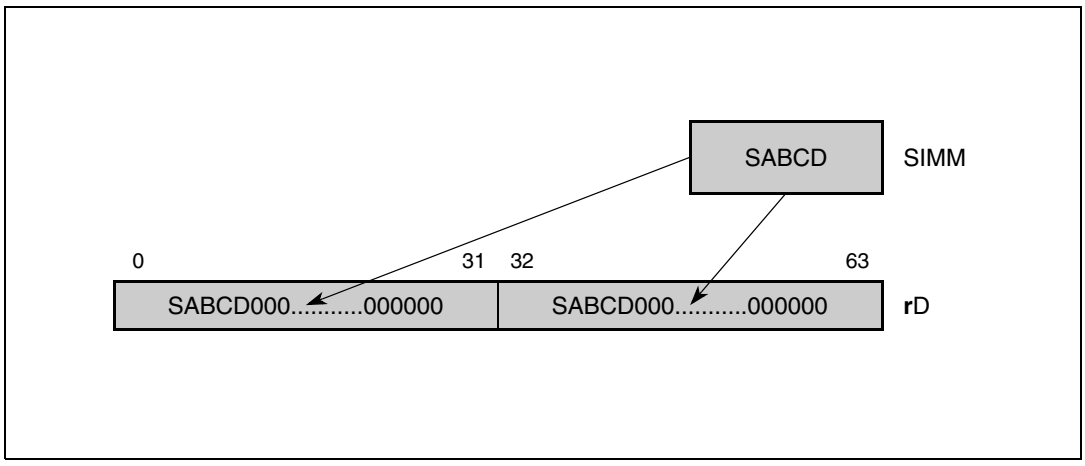
evsplatfi **r**D,SIMM

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	0	1	0	0	**r**D					SIMM					0	0	0	0	0	0	1	0	0	0	1	0	1	0	1	1

$rD_{0:31} \leftarrow SIMM \,||\, {}^{27}0$
$rD_{32:63} \leftarrow SIMM \,||\, {}^{27}0$

The 5-bit immediate value is padded with trailing zeros and placed in both elements of **r**D, as shown in *Figure 151*. The SIMM ends up in bit positions **r**D[0–4] and **r**D[32–36].

Figure 151. Vector splat fractional immediate (evsplatfi)



evsplati | SPE APU | User | **evsplati**

Vector splat immediate

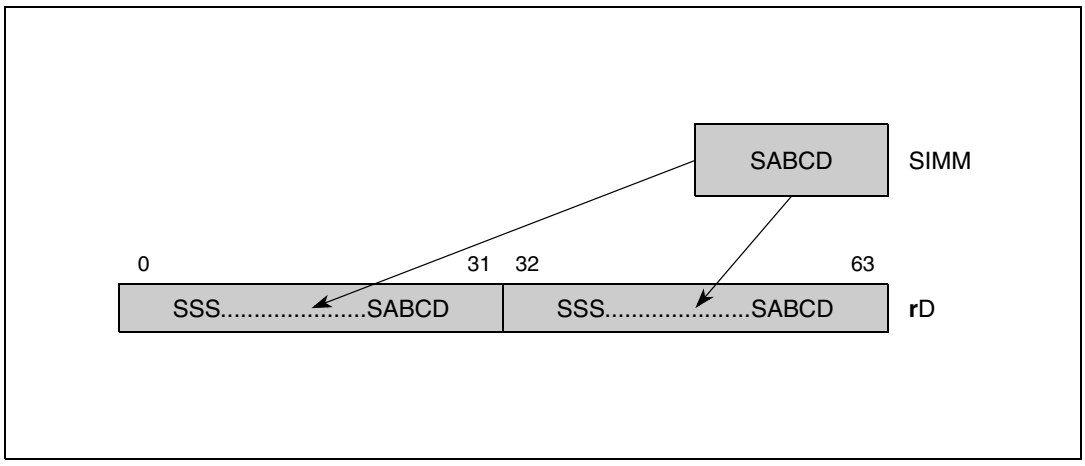
evsplati **r**D,SIMM

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	0	1	0	0	**r**D					SIMM					0	0	0	0	0	0	1	0	0	0	1	0	1	0	0	1

$rD_{0:31} \leftarrow EXTS(SIMM)$
$rD_{32:63} \leftarrow EXTS(SIMM)$

The 5-bit immediate value is sign extended and placed in both elements of **r**D, as shown in *Figure 152*.

Figure 152. evsplati sign extend



evsrwis | SPE APU | User | **evsrwis**

Vector shift right word immediate signed

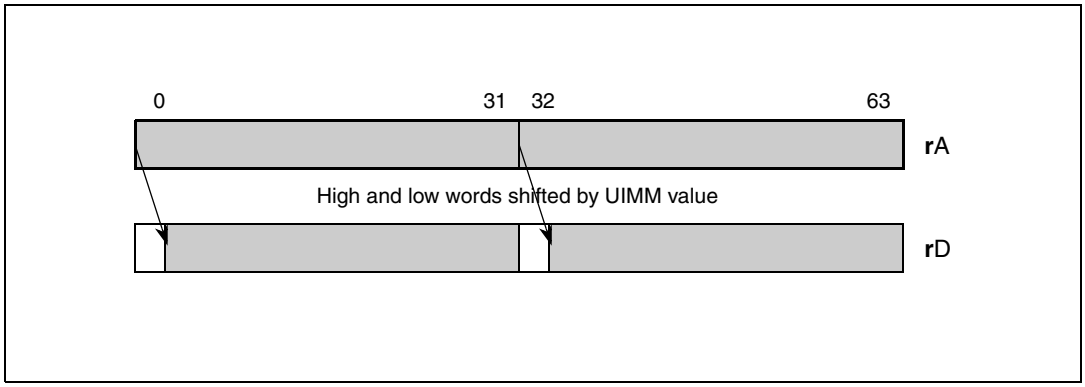
evsrwis **r**D,**r**A,UIMM

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	0	1	0	0	**r**D										**r**A															
UIMM																0	1	0	0	0	1	0	0	0	1	1					

n ← UIMM
$rD_{0:31} \leftarrow EXTS(rA_{0:31-n})$
$rD_{32:63} \leftarrow EXTS(rA_{32:63-n})$

Both high and low elements of **r**A are shifted right by the 5-bit UIMM value. Bits in the most significant positions vacated by the shift are filled with a copy of the sign bit.

Figure 153. Vector shift right word immediate signed (evsrwis)



evsrwiu | SPE APU | User | **evsrwiu**

Vector shift right word immediate unsigned

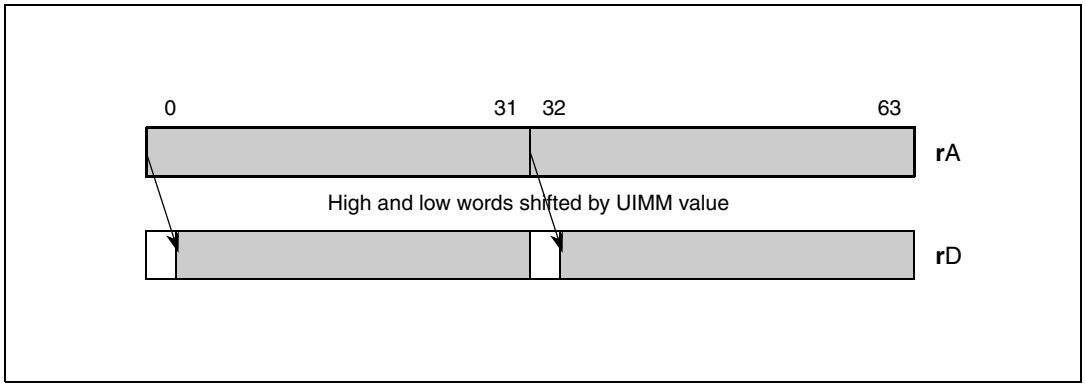
evsrwiu **r**D,**r**A,UIMM

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	0	1	0	0	**r**D					**r**A					UIMM				0	1	0	0	0	1	0	0	0	1	0	

n ← UIMM
$rD_{0:31} \leftarrow EXTZ(rA_{0:31-n})$
$rD_{32:63} \leftarrow EXTZ(rA_{32:63-n})$

Both high and low elements of **r**A are shifted right by the 5-bit UIMM value; 0 bits are shifted in to the most significant position. Bits in the most significant positions vacated by the shift are filled with a zero bit.

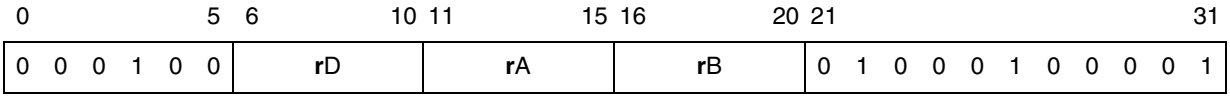
Figure 154. Vector shift right word immediate unsigned (evsrwiu)



evsrws | SPE APU | User | **evsrws**

Vector shift right word signed

evsrws **r**D,**r**A,**r**B

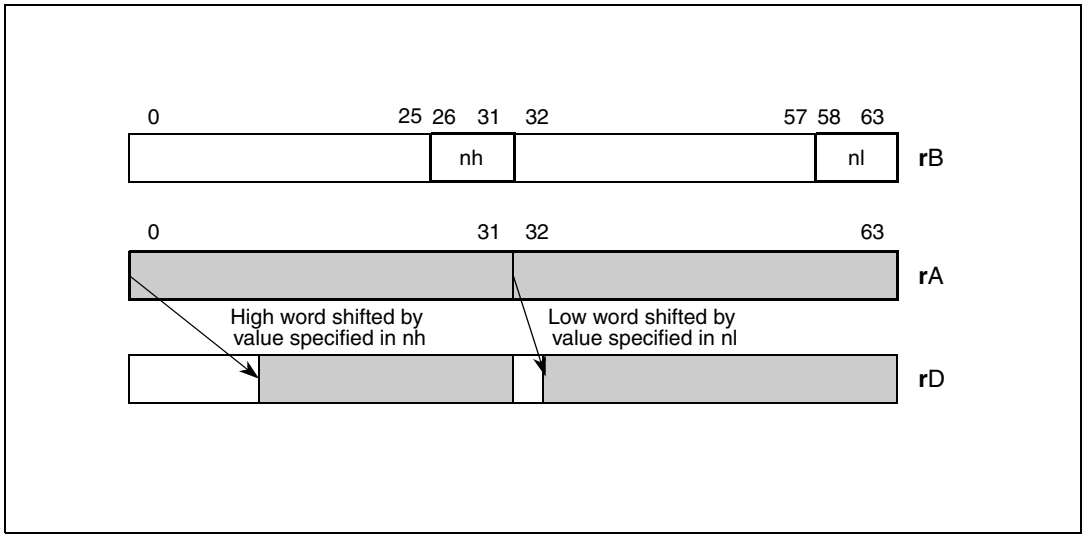


$nh \leftarrow rB_{26:31}$
$nl \leftarrow rB_{58:63}$
$rD_{0:31} \leftarrow EXTS(rA_{0:31-nh})$
$rD_{32:63} \leftarrow EXTS(rA_{32:63-nl})$

Both the high and low elements of **r**A are shifted right by an amount specified in **r**B. The result is placed into **r**D. The separate shift amounts for each element are specified by 6 bits in **r**B that lie in bit positions 26–31 and 58–63. The sign bits are shifted in to the most significant position.

Shift amounts from 32 to 63 give a result of 32 sign bits.

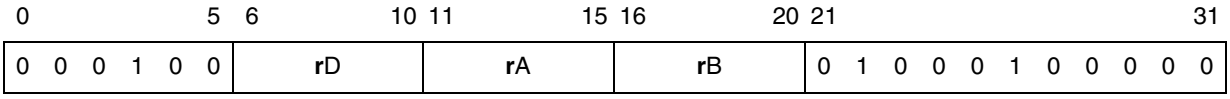
Figure 155. Vector shift right word signed (evsrws)



evsrwu | SPE APU | User | **evsrwu**

Vector shift right word unsigned

evsrwu **r**D,**r**A,**r**B

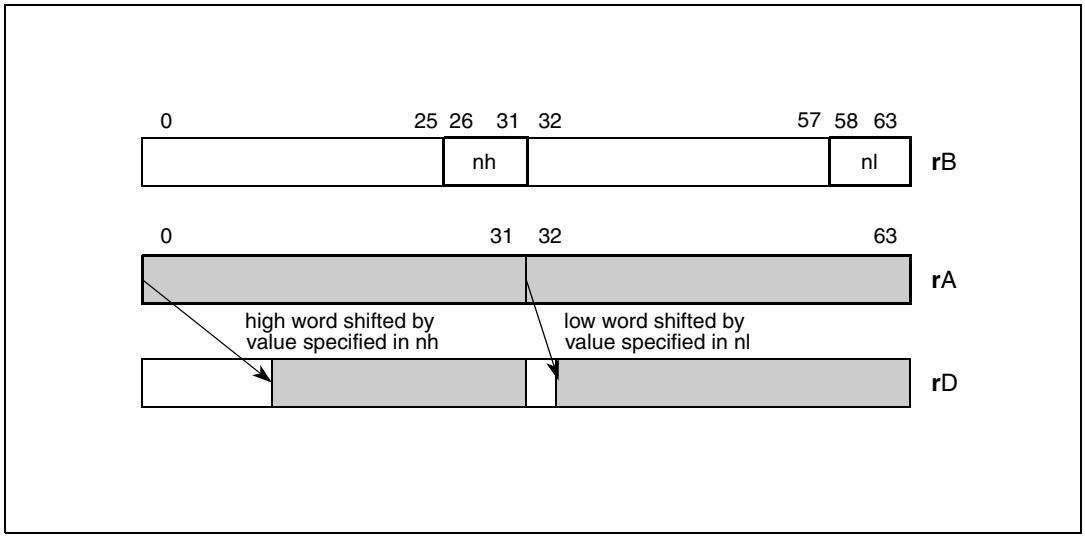


$nh \leftarrow rB_{26:31}$
$nl \leftarrow rB_{58:63}$
$rD_{0:31} \leftarrow EXTZ(rA_{0:31\text{-}nh})$
$rD_{32:63} \leftarrow EXTZ(rA_{32:63\text{-}nl})$

Both the high and low elements of **r**A are shifted right by an amount specified in **r**B. The result is placed into **r**D. The separate shift amounts for each element are specified by 6 bits in **r**B that lie in bit positions 26–31 and 58–63. Zero bits are shifted in to the most significant position.

Shift amounts from 32 to 63 give a zero result.

Figure 156. Vector shift right word unsigned (evsrwu)



evstdd | SPE, Vector SPFP, Scalar DPFP APUs | User | **evstdd**

Vector store double of double

evstdd **r**S,**d**(**r**A)

0	5	6	10	11	15	16	20	21	31
0 0 0 1 0 0		**r**S		**r**A		UIMM(1)		0 1 1 0 0 1 0 0 0 0 1	

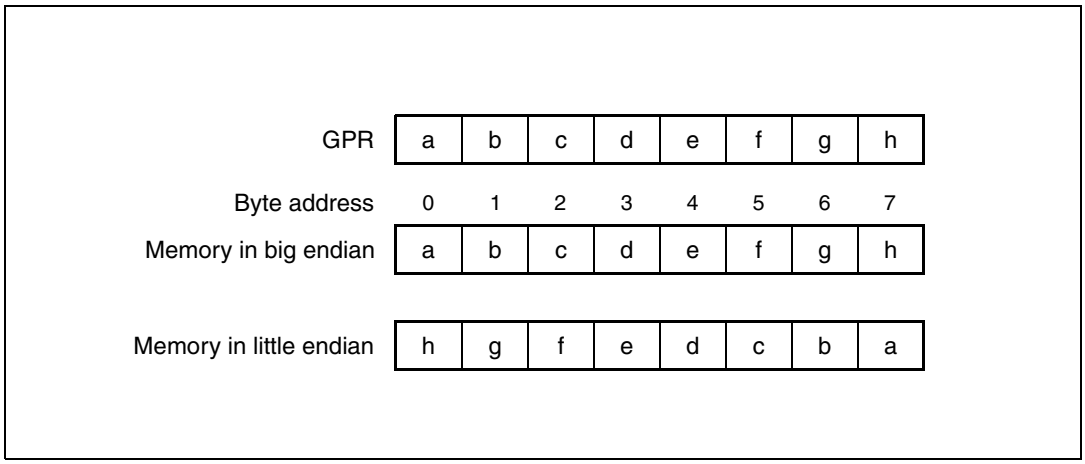
1. **d** = UIMM * 8

```
if (rA = 0) then b ← 0
else b ← (rA)
EA ← b + EXTZ(UIMM*8)
MEM(EA,8) ← RS0:63
```

The contents of **r**S are stored as a double word in storage addressed by EA.

Figure 157 shows how bytes are stored in memory as determined by the endian mode.

Figure 157. evstdd results in big- and little-endian modes



Implementation note: If the EA is not double-word aligned, an alignment exception occurs.

evstddx | SPE, Vector SPFP, Scalar DPFP APUs | User | **evstddx**

Vector store double of double indexed

evstddx **r**S,**r**A,**r**B

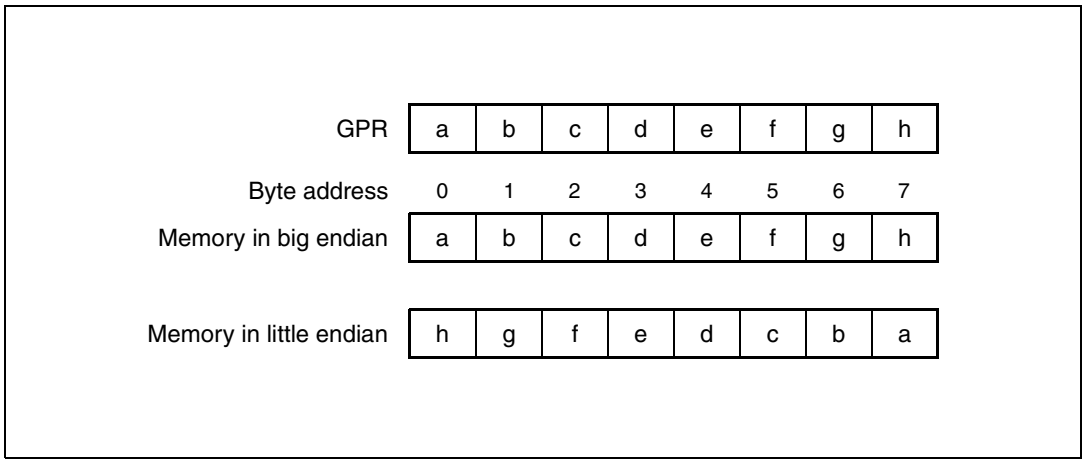
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	0	1	0	0	**r**S					**r**A					**r**B					0	1	1	0	0	1	0	0	0	0	0

if (rA = 0) then b ← 0
else b ← (rA)
EA ← b + (rB)
MEM(EA,8) ← $RS_{0:63}$

The contents of **r**S are stored as a double word in storage addressed by EA.

Figure 158 shows how bytes are stored in memory as determined by the endian mode.

Figure 158. evstddx Results in big- and little-endian modes



Note: Implementation:
If the EA is not double-word aligned, an alignment exception occurs.

evstdh | SPE APU | User | **evstdh**

Vector store double of four half words

evstdh **r**S,**d**(**r**A)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	0	1	0	0	**r**S					**r**A					UIMM(1)					0	1	1	0	0	1	0	0	1	0	1

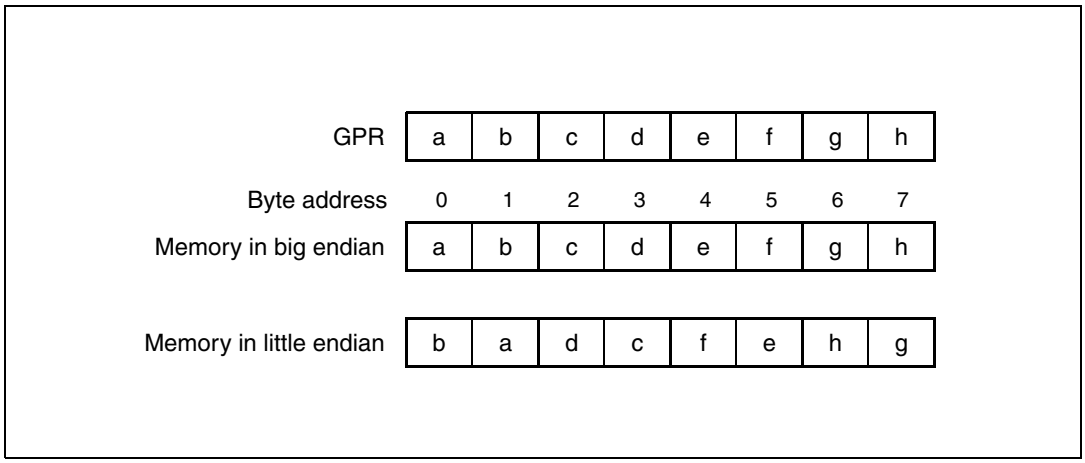
1. **d** = UIMM * 8

```
if (rA = 0) then b ← 0
else b ← (rA)
EA ← b + EXTZ(UIMM*8)
MEM(EA,2) ← RS0:15
MEM(EA+2,2) ← RS16:31
MEM(EA+4,2) ← RS32:47
MEM(EA+6,2) ← RS48:63
```

The contents of **r**S are stored as four half words in storage addressed by EA.

Figure 159 shows how bytes are stored in memory as determined by the endian mode.

Figure 159. evstdh Results in big- and little-endian modes

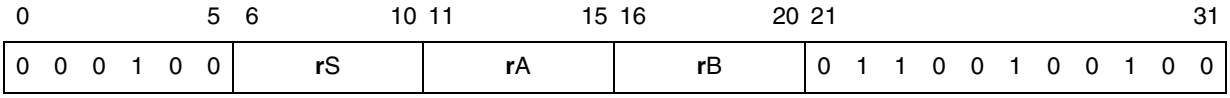


Note: Implementation note:
If the EA is not double-word aligned, an alignment exception occurs.

evstdhx | SPE APU | User | **evstdhx**

Vector store double of four half words indexed

evstdhx **r**S,**r**A,**r**B



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 **r**S are stored as four half words in storage addressed by EA.

Figure 160 shows how bytes are stored in memory as determined by the endian mode.

Figure 160. evstdhx Results in big- and little-endian modes

Note: Implementation:
If the EA is not double-word aligned, an alignment exception occurs.

evstdw | SPE APU | User | **evstdw**

Vector store double of two words

evstdw **rS,d(rA)**

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	0	1	0	0	**r**S					**r**A					UIMM(1)					0	1	1	0	0	1	0	0	0	1	1

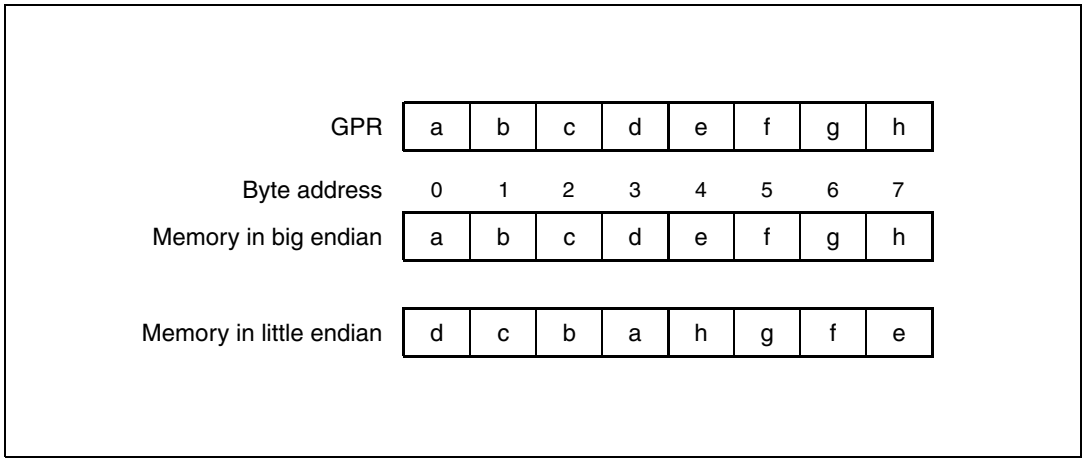
1. **d** = UIMM * 8

```
if (rA = 0) then b ← 0
else b ← (rA)
EA ← b + EXTZ(UIMM*8)
MEM(EA,4) ← RS0:31
MEM(EA+4,4) ← RS32:63
```

The contents of **r**S are stored as two words in storage addressed by EA.

Figure 161 shows how bytes are stored in memory as determined by the endian mode.

Figure 161. evstdw results in big- and little-endian modes

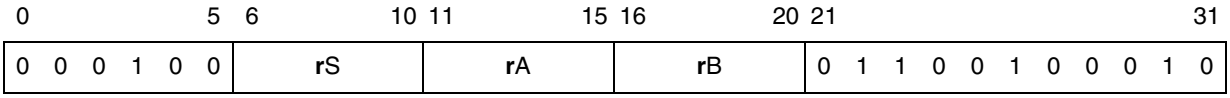


Note: Implementation:
If the EA is not double-word aligned, an alignment exception occurs.

evstdwx | SPE APU | User | **evstdwx**

Vector store double of two words indexed

evstdwx **r**S,**r**A,**r**B

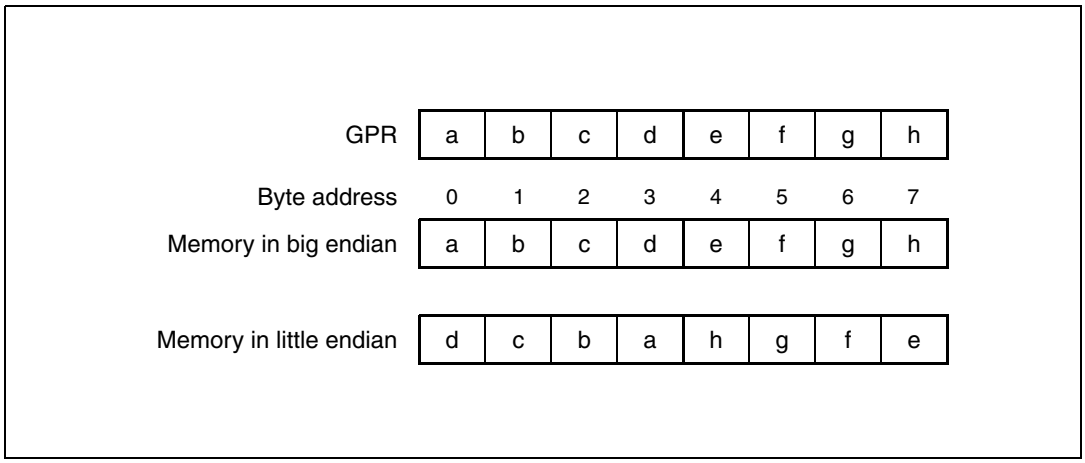


if (rA = 0) then b ← 0
else b ← (rA)
EA ← b + (rB)
MEM(EA,4) ← $RS_{0:31}$
MEM(EA+4,4) ← $RS_{32:63}$

The contents of **r**S are stored as two words in storage addressed by EA.

Figure 162 shows how bytes are stored in memory as determined by the endian mode.

Figure 162. evstdwx Results in big- and little-endian modes



Note: Implementation:
If the EA is not double-word aligned, an alignment exception occurs.

evstwhe | SPE APU | User | **evstwhe**

Vector store word of two half words from even

evstwhe **rS,d(rA)**

0	5	6	10	11	15	16	20	21	31
0 0 0 1 0 0		**r**S		**r**A		UIMM(1)		0 1 1 0 0 1 1 0 0 0 1	

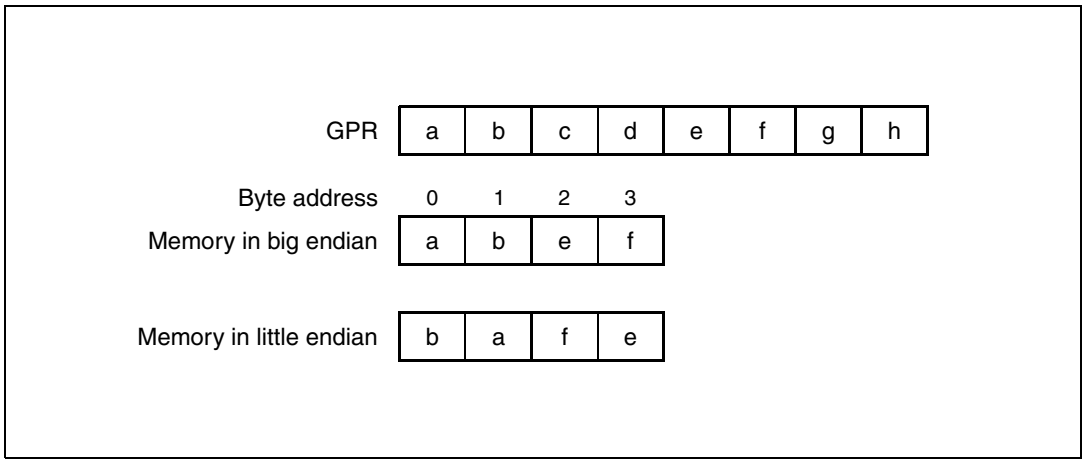
1. **d** = UIMM * 4

```
if (rA = 0) then b ← 0
else b ← (rA)
EA ← b + EXTZ(UIMM*4)
MEM(EA,2) ← RS0:15
MEM(EA+2,2) ← RS32:47
```

The even half words from each element of **r**S are stored as two half words in storage addressed by EA.

Figure 163 shows how bytes are stored in memory as determined by the endian mode.

Figure 163. evstwhe Results in big- and little-endian modes

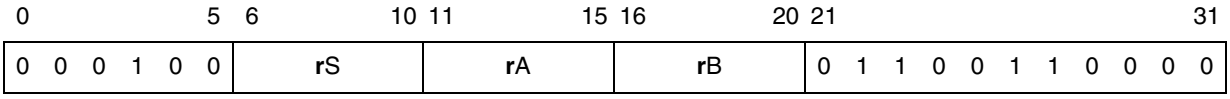


Note: Implementation:
If the EA is not word aligned, an alignment exception occurs.

evstwhex | SPE APU | User | **evstwhex**

Vector store word of two half words from even indexed

evstwhex **r**S,**r**A,**r**B

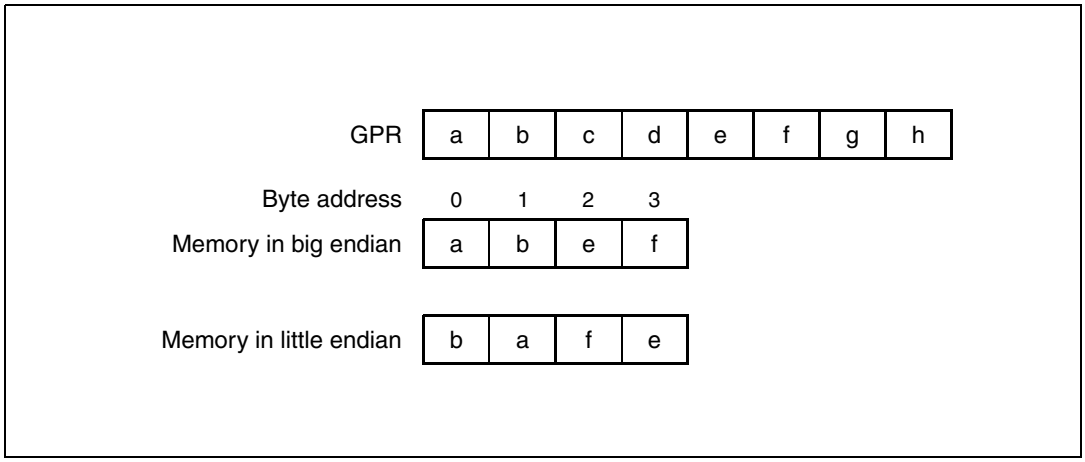


if (rA = 0) then b ← 0
else b ← (rA)
EA ← b + (rB)
MEM(EA,2) ← $RS_{0:15}$
MEM(EA+2,2) ← $RS_{32:47}$

The even half words from each element of **r**S are stored as two half words in storage addressed by EA.

Figure 164 shows how bytes are stored in memory as determined by the endian mode.

Figure 164. evstwhex Results in big- and little-endian modes



Note: Implementation:
If the EA is not word aligned, an alignment exception occurs.

evstwho | SPE APU | User | **evstwho**

Vector store word of two half words from odd

evstwho **rS,d(rA)**

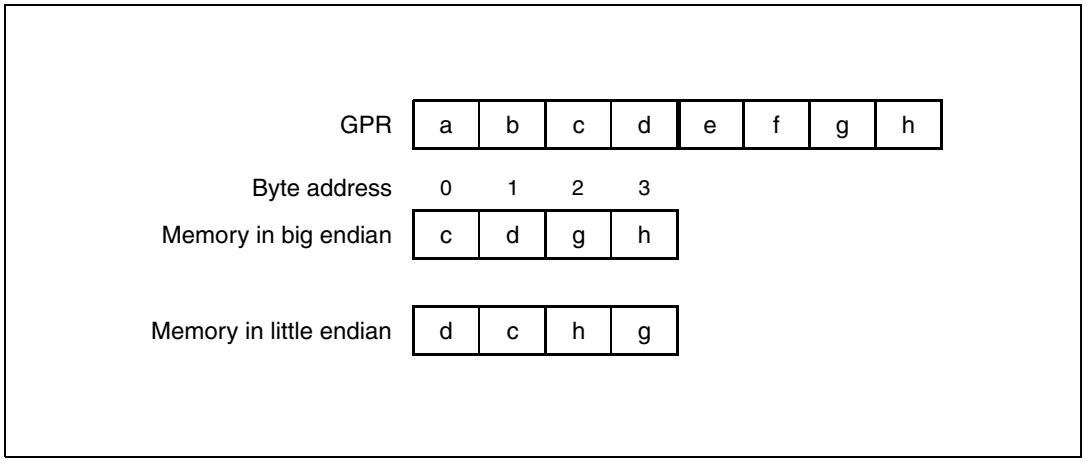
0	1	2	3	4	5	6	10	11	15	16	20	21	22	23	24	25	26	27	28	29	30	31
0	0	0	1	0	0	**r**S		**r**A		UIMM(1)		0	1	1	0	0	1	1	0	1	0	1

1. **d** = UIMM * 4

```
if (rA = 0) then b ← 0
else b ← (rA)
EA ← b + EXTZ(UIMM*4)
MEM(EA,2) ← RS16:31
MEM(EA+2,2) ← RS48:63
```

The odd half words from each element of **r**S are stored as two half words in storage addressed by EA.

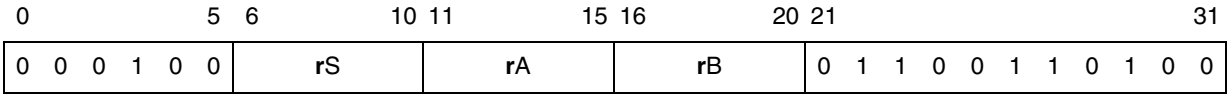
Figure 165. evstwho Results in big- and little-endian modes



evstwhox | SPE APU | User | **evstwhox**

Vector store word of two half words from odd indexed

evstwhox **r**S,**r**A,**r**B

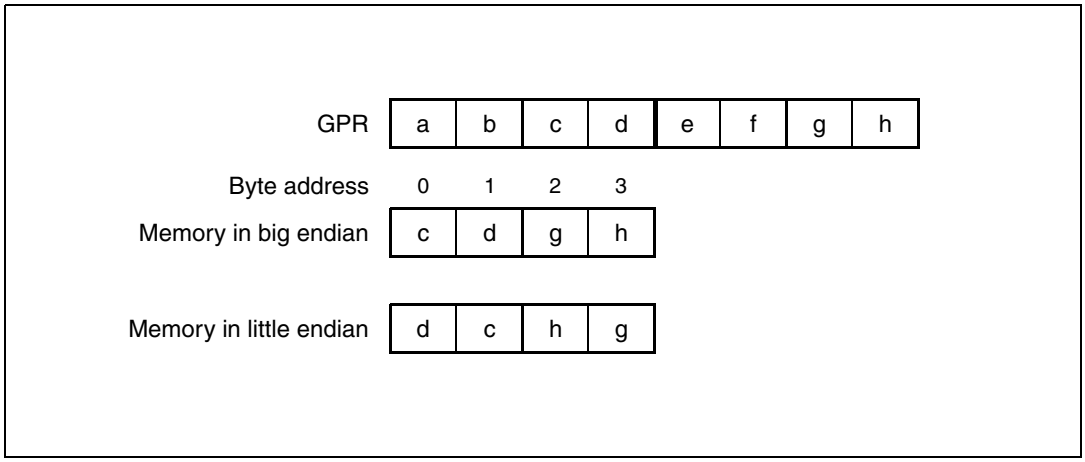


if (rA = 0) then b ← 0
else b ← (rA)
EA ← b + (rB)
MEM(EA,2) ← $RS_{16:31}$
MEM(EA+2,2) ← $RS_{48:63}$

The odd half words from each element of **r**S are stored as two half words in storage addressed by EA.

Figure 166 shows how bytes are stored in memory as determined by the endian mode.

Figure 166. evstwhox Results in big- and little-endian modes



Note: Implementation:
If the EA is not word aligned, an alignment exception occurs.

evstwwe | SPE APU | User | **evstwwe**

Vector store word of word from even

evstwwe **rS,d(rA)**

0	1	2	3	4	5	6	10	11	15	16	20	21	22	23	24	25	26	27	28	29	30	31
0	0	0	1	0	0	**r**S			**r**A		UIMM(1)		0	1	1	0	0	1	1	1	0	0	1

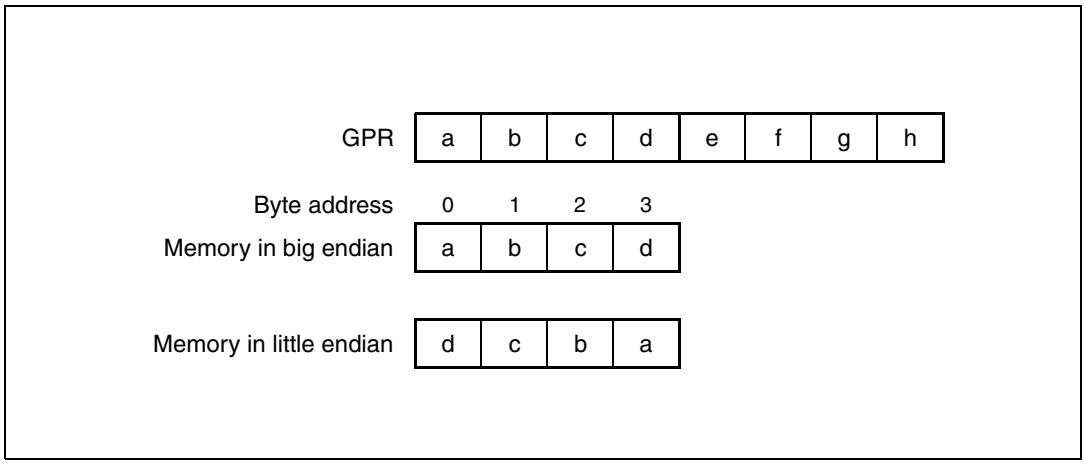
1. **d** = UIMM * 4

```
if (rA = 0) then b ← 0
else b ← (rA)
EA ← b + EXTZ(UIMM*4)
MEM(EA,4) ← RS0:31
```

The even word of **r**S is stored in storage addressed by EA.

Figure 167 shows how bytes are stored in memory as determined by the endian mode.

Figure 167. evstwwe Results in big- and little-endian modes

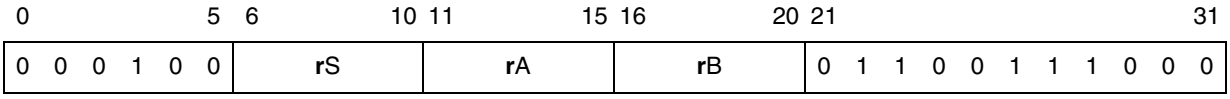


Note: Implementation note:
If the EA is not word aligned, an alignment exception occurs.

evstwwex | SPE APU | User | **evstwwex**

Vector store word of word from even indexed

evstwwex **r**S,**r**A,**r**B

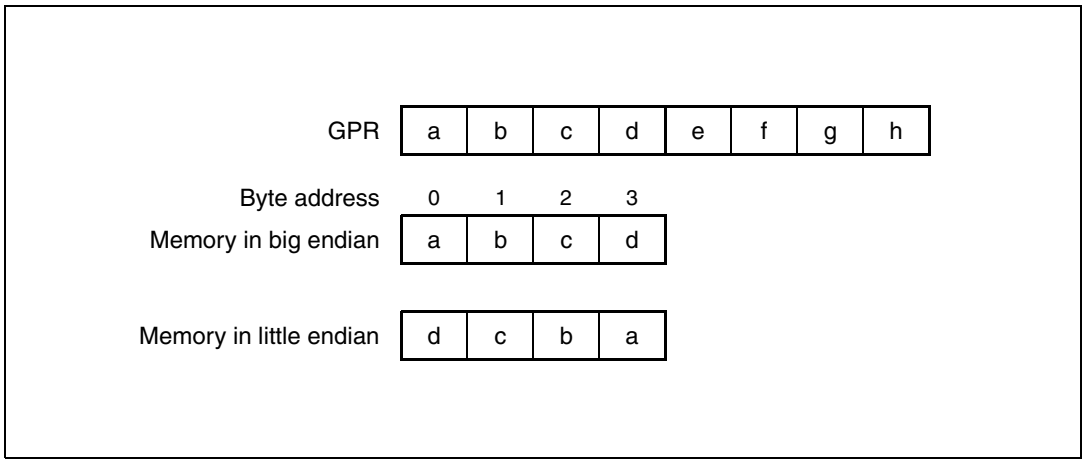


if (rA = 0) then b ← 0
else b ← (rA)
EA ← b + (rB)
MEM(EA,4) ← $RS_{0:31}$

The even word of **r**S is stored in storage addressed by EA.

Figure 168 shows how bytes are stored in memory as determined by the endian mode.

Figure 168. evstwwex Results in big- and little-endian modes



Note: Implementation:
If the EA is not word aligned, an alignment exception occurs.

evstwwo | SPE APU | User | **evstwwo**

Vector store word of word from odd

evstwwo **rS,d(rA)**

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	0	1	0	0	**r**S										**r**A					UIMM(1)											
0	1	1	0	0	1	1	1	1	0	1																					

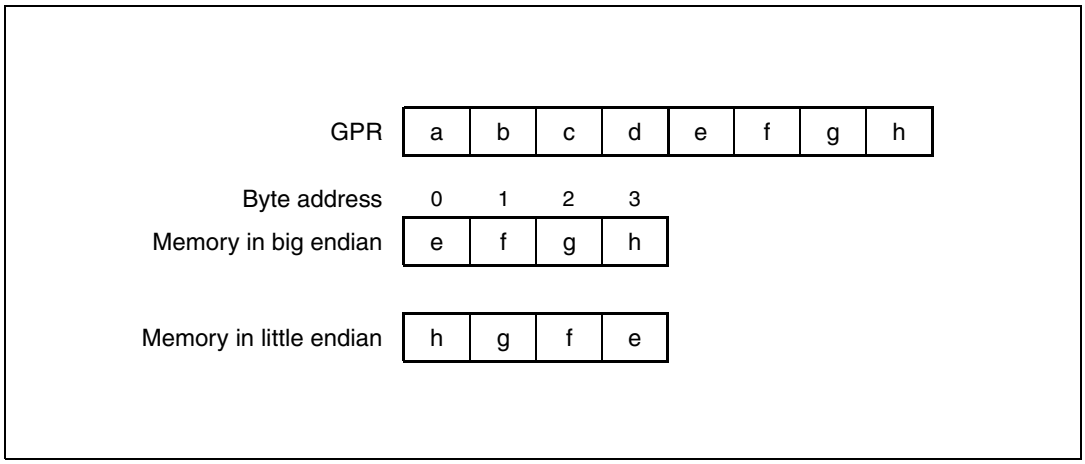
1. **d** = UIMM * 4

```
if (rA = 0) then b ← 0
else b ← (rA)
EA ← b + EXTZ(UIMM*4)
MEM(EA,4) ← rS32:63
```

The odd word of **r**S is stored in storage addressed by EA.

Figure 169 shows how bytes are stored in memory as determined by the endian mode.

Figure 169. evstwwo Results in big- and little-endian modes

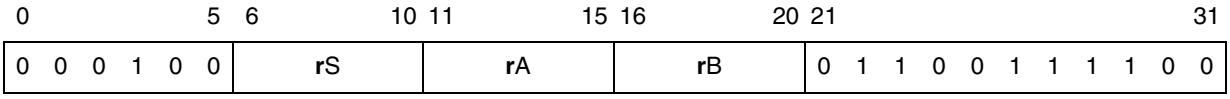


Note: Implementation note:
If the EA is not word aligned, an alignment exception occurs.

evstwwox | SPE APU | User | **evstwwox**

Vector store word of word from odd indexed

evstwwox **r**S,**r**A,**r**B

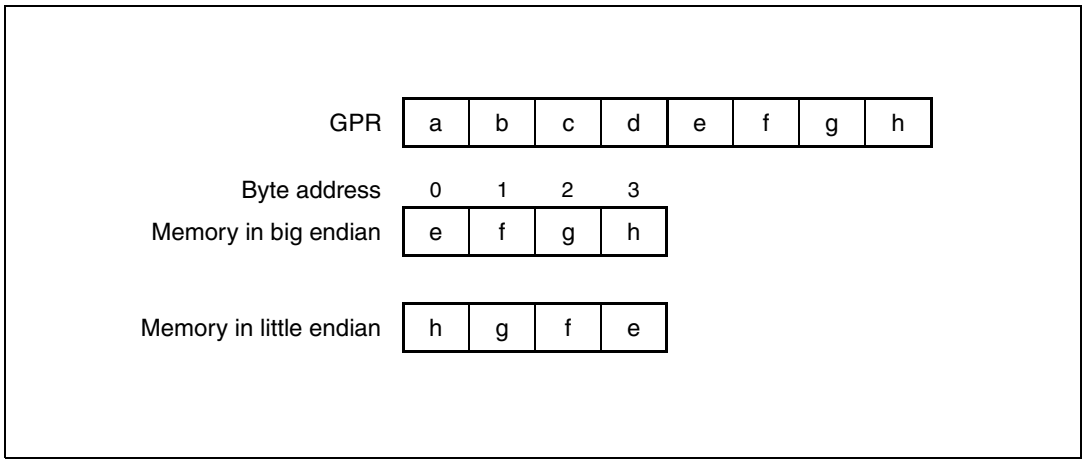


if (rA = 0) then b ← 0
else b ← (rA)
EA ← b + (rB)
MEM(EA,4) ← $rS_{32:63}$

The odd word of **r**S is stored in storage addressed by EA.

Figure 170 shows how bytes are stored in memory as determined by the endian mode.

Figure 170. evstwwox Results in big- and little-endian modes



Note: Implementation note:
If the EA is not word aligned, an alignment exception occurs.

evsubfsmiaaw | SPE APU | User | **evsubfsmiaaw**

Vector subtract signed, modulo, integer to accumulator word

evsubfsmiaaw **r**D,**r**A

0	5	6	10	11	15	16	20	21	31
0 0 0 1 0 0		**r**D		**r**A		0 0 0 0 0		1 0 0 1 1 0 0 1 0 1 1	

// high
$rD_{0:31} \leftarrow ACC_{0:31} - rA_{0:31}$

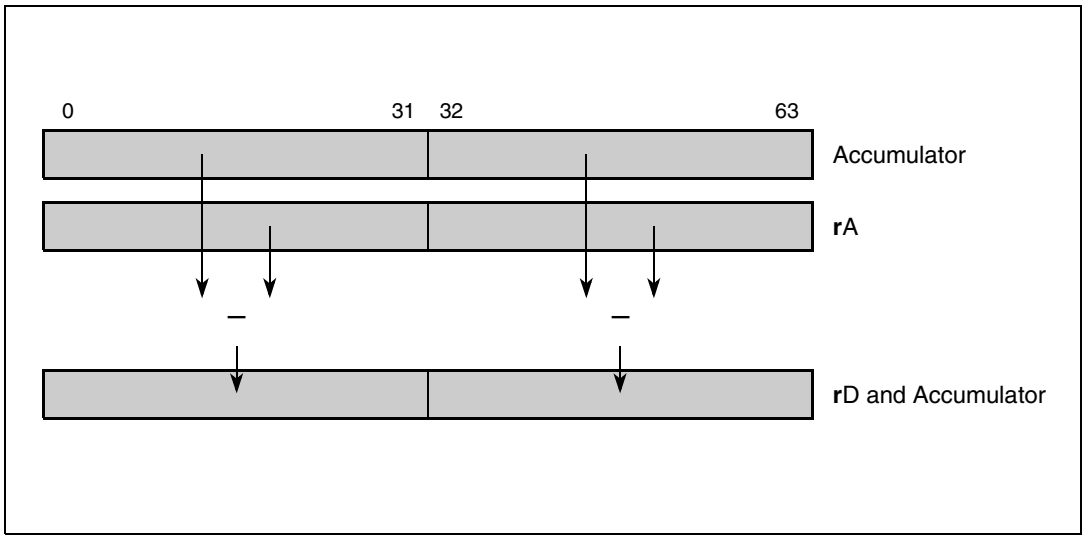
// low
$rD_{32:63} \leftarrow ACC_{32:63} - rA_{32:63}$

// update accumulator
$ACC_{0:63} \leftarrow rD_{0:63}$

Each word element in **r**A is subtracted from the corresponding element in the accumulator and the difference is placed into the corresponding **r**D word and into the accumulator.

Other registers altered: ACC

Figure 171. Vector subtract signed, modulo, integer to accumulator word (evsubfsmiaaw)



evsubfssiaaw | SPE APU | User | **evsubfssiaaw**

Vector subtract signed, saturate, integer to accumulator word

evsubfssiaaw **r**D,**r**A

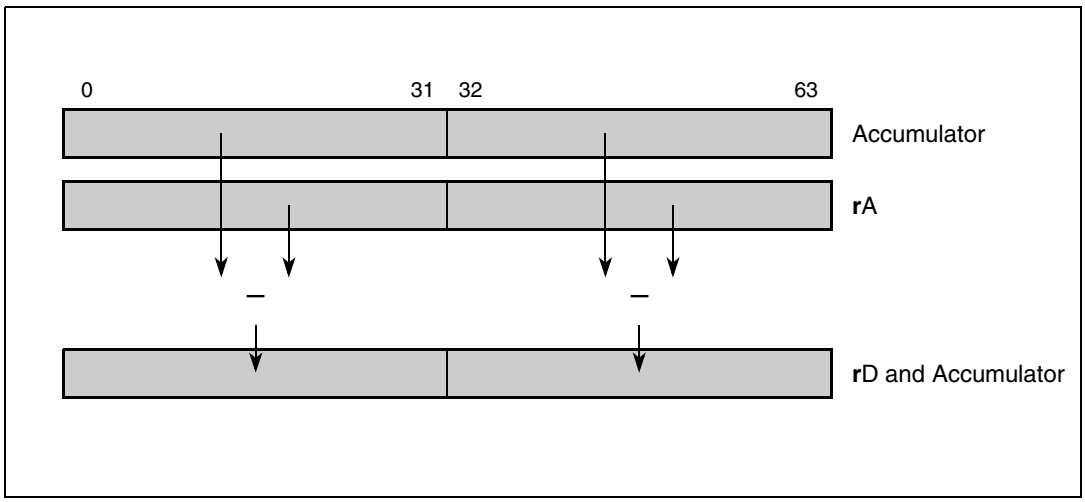
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	0	1	0	0	**r**D					**r**A					0	0	0	0	0	1	0	0	1	1	0	0	0	0	1	1

```
// high
temp0:63 ← EXTS(ACC0:31) - EXTS(rA0:31)
ovh ← temp31 ⊕ temp32
rD0:31 ← SATURATE(ovh, temp31, 0x80000000, 0x7fffffff, temp32:63)

// low
temp0:63 ← EXTS(ACC32:63) - EXTS(rA32:63)
ovl ← temp31 ⊕ temp32
rD32:63 ← SATURATE(ovl, temp31, 0x80000000, 0x7fffffff, temp32:63)

// update accumulator
ACC0:63 ← rD0:63

SPEFSCROVH ← ovh
SPEFSCROV ← ovl
SPEFSCRSOVH ← SPEFSCRSOVH | ovh
SPEFSCRSOV ← SPEFSCRSOV | ovl
```

Each signed integer word element in **r**A is sign-extended and subtracted from the corresponding sign-extended element in the accumulator, saturating if overflow occurs, and the results are placed in **r**D and the accumulator. Any overflow is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR ACC

Figure 172. Vector subtract signed, saturate, integer to accumulator word (evsubfssiaaw)



evsubfumiaaw | SPE APU | User | **evsubfumiaaw**

Vector subtract unsigned, modulo, integer to accumulator word

evsubfumiaaw **r**D,**r**A

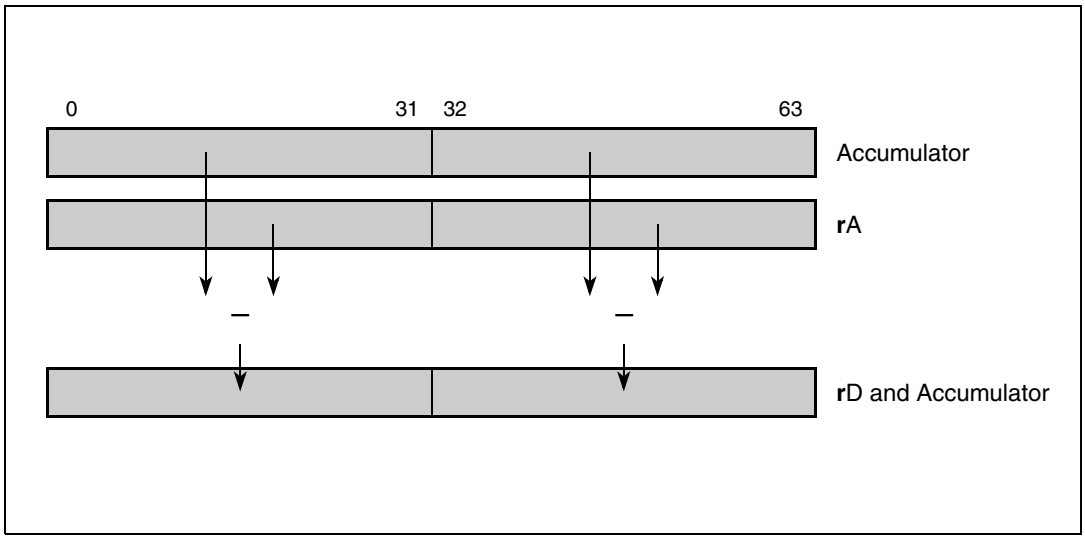
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	0	1	0	0	**r**D					**r**A					0	0	0	0	0	1	0	0	1	1	0	0	1	0	1	0

```
// high
rD0:31 ← ACC0:31 - rA0:31

// low
rD32:63 ← ACC32:63 - rA32:63

// update accumulator
ACC0:63 ← rD0:63
```

Each unsigned integer word element in **r**A is subtracted from the corresponding element in the accumulator and the results are placed in **r**D and into the accumulator.

Other registers altered: ACC

Figure 173. Vector subtract unsigned, modulo, integer to accumulator word (evsubfumiaaw)



evsubfusiaaw | SPE APU | User | **evsubfusiaaw**

Vector subtract unsigned, saturate, integer to accumulator word

evsubfusiaaw **r**D,**r**A

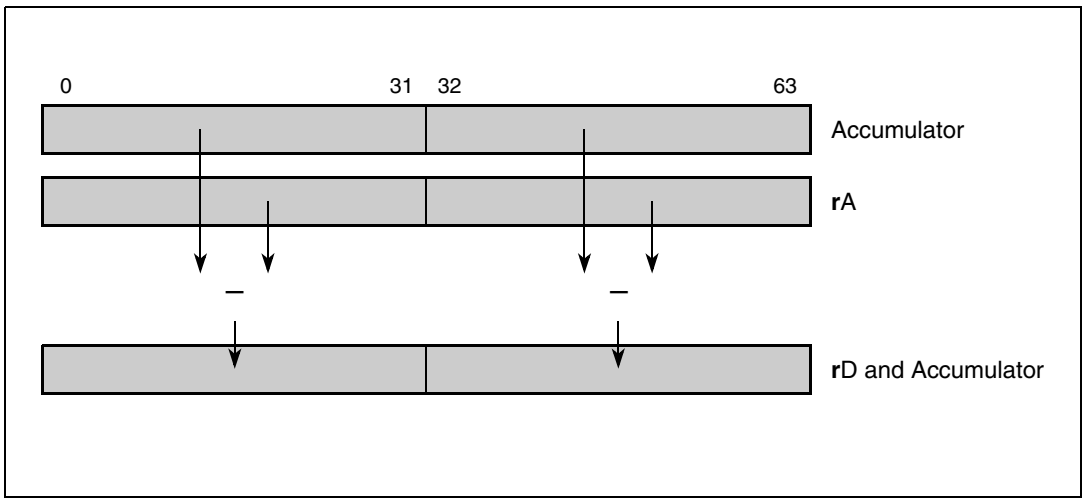
0	5	6	10	11	15	16	20	21	31
0 0 0 1 0 0		**r**D		**r**A		0 0 0 0 0		1 0 0 1 1 0 0 0 0 1 0	

```
// high
temp0:63 ← EXTZ(ACC0:31) - EXTZ(rA0:31)
ovh ← temp31
rD0:31 ← SATURATE(ovh, temp31, 0x00000000, 0x00000000, temp32:63)

// low
temp0:63 ← EXTS(ACC32:63) - EXTS(rA32:63)
ovl ← temp31
rD32:63 ← SATURATE(ovl, temp31, 0x00000000, 0x00000000, temp32:63)

// update accumulator
ACC0:63 ← rD0:63

SPEFSCROVH ← ovh
SPEFSCROV ← ovl
SPEFSCRSOVH ← SPEFSCRSOVH | ovh
SPEFSCRSOV ← SPEFSCRSOV | ovl
```

Each unsigned integer word element in **r**A is zero-extended and subtracted from the corresponding zero-extended element in the accumulator, saturating if underflow occurs, and the results are placed in **r**D and the accumulator. Any underflow is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR ACC

Figure 174. Vector subtract unsigned, saturate, integer to accumulator word (evsubfusiaaw)



evsubfw | SPE APU | User | **evsubfw**

Vector subtract from word

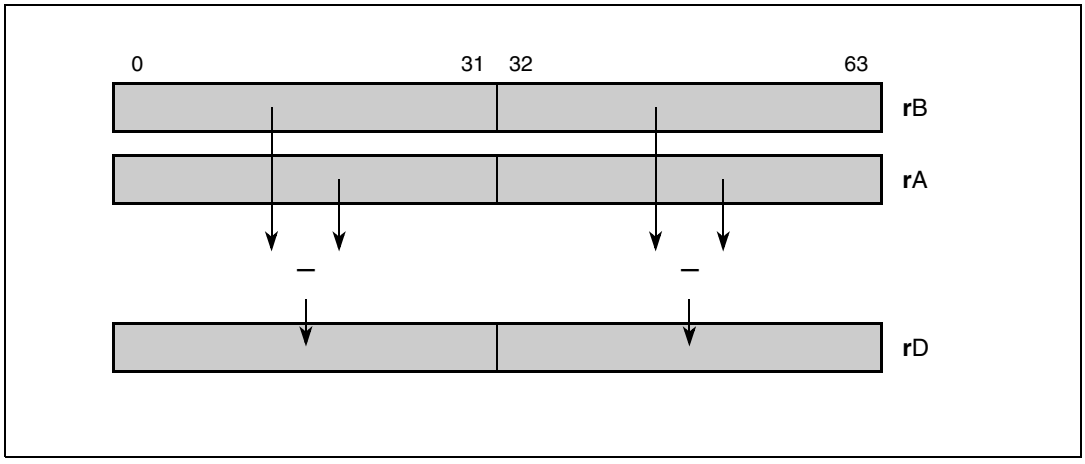
evsubfw **r**D,**r**A,**r**B

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	0	1	0	0	**r**D					**r**A					**r**B					0	1	0	0	0	0	0	0	1	0	0

$rD_{0:31} \leftarrow rB_{0:31} - rA_{0:31}$ // Modulo difference
$rD_{32:63} \leftarrow rB_{32:63} - rA_{32:63}$ // Modulo difference

Each signed integer element of **r**A is subtracted from the corresponding element of **r**B and the results are placed into **r**D.

Figure 175. Vector subtract from word (evsubfw)



evsubifw | SPE APU | User | **evsubifw**

Vector subtract immediate from word

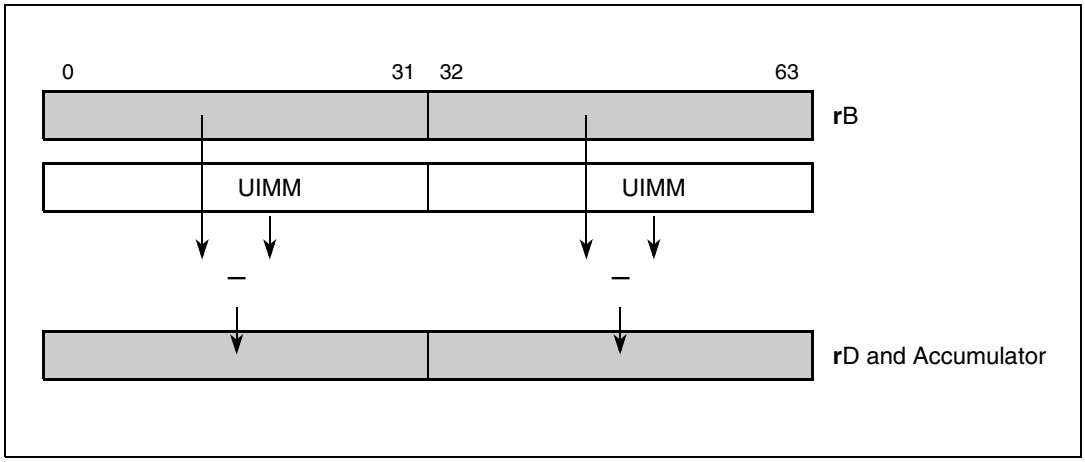
evsubifw **r**D,UIMM,**r**B

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	0	1	0	0	**r**D					UIMM					**r**B					0	1	0	0	0	0	0	0	1	1	0

$rD_{0:31} \leftarrow rB_{0:31}$ - EXTZ(UIMM) // Modulo difference
$rD_{32:63} \leftarrow rB_{32:63}$ - EXTZ(UIMM) // Modulo difference

UIMM is zero-extended and subtracted from both the high and low elements of **r**B. Note that the same value is subtracted from both elements of the register. UIMM is 5 bits.

Figure 176. Vector subtract immediate from word (evsubifw)



evxor | SPE APU | User | **evxor**

Vector XOR

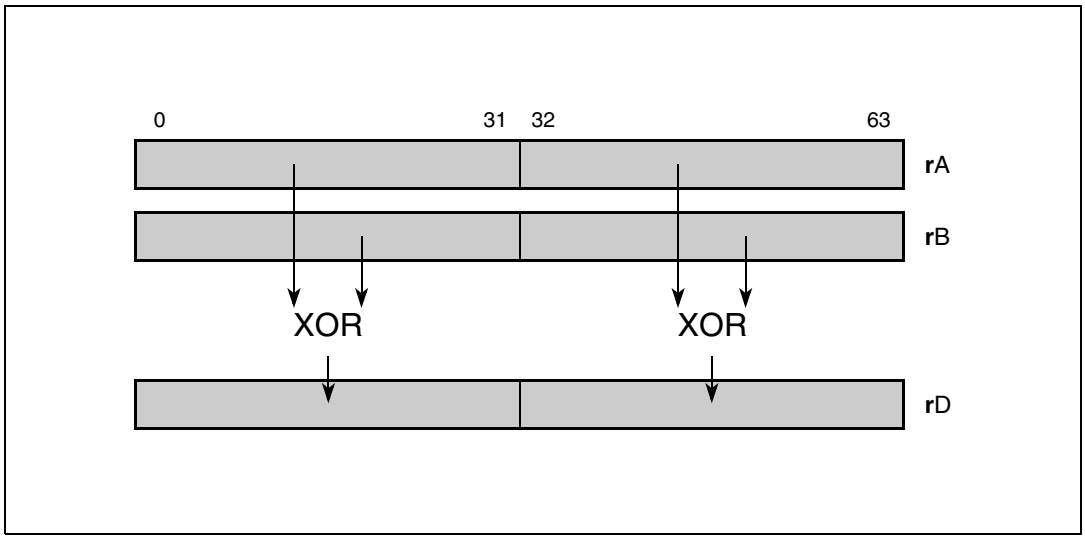
evxor **r**D,**r**A,**r**B

0	5	6	10	11	15	16	20	21	31
0 0 0 1 0 0		**r**D		**r**A		**r**B		0 1 0 0 0 0 1 0 1 1 0	

$rD_{0:31} \leftarrow rA_{0:31} \oplus rB_{0:31}$ // Bitwise XOR
$rD_{32:63} \leftarrow rA_{32:63} \oplus rB_{32:63}$ // Bitwise XOR

Each element of **r**A and **r**B is exclusive-ORed. The results are placed in **r**D.

Figure 177. Vector XOR (evxor)



extsb | Book E | User | **extsb**

Extend sign (byte | half word)

extsb	**r**A,**r**S	(SZ=0b01, Rc=0)
extsb.	**r**A,**r**S	(SZ=0b01, Rc=1)
extsh	**r**A,**r**S	(SZ=0b00, Rc=0)
extsh.	**r**A,**r**S	(SZ=0b00, Rc=1)

0	5	6	10	11	15	16	20	21	23	24	25	26	30	31
0 1 1 1 1 1		**r**S		**r**A		///		1 1 1		S	Z	1 1 0 1 0		Rc

if ‘extsb[.]’ then n ← 56
if ‘extsh[.]’ then n ← 48
if ‘extsw’ then n ← 32
if Rc=1 then do

$LT \leftarrow rS_{n:63} < 0$
$GT \leftarrow rS_{n:63} > 0$
$EQ \leftarrow rS_{n:63} = 0$
$CR0 \leftarrow LT \,||\, GT \,||\, EQ \,||\, SO$

$s \leftarrow rS_n$
$rA \leftarrow {}^{n}s \,||\, rS_{n:63}$

For **extsb**[**.**], the contents of **r**S[56–63] are placed into **r**A[56–63]. Bit **r**S[56] is copied into bits 0–55 of **r**A. If Rc=1, CR field 0 is set to reflect the result.

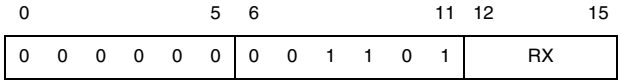
For **extsh**[**.**], the contents of **r**S[48–63] are placed into **r**A[48–63]. **r**S[48] is copied into **r**A[0–47]. If Rc=1, CR field 0 is set to reflect the result.

- Other registers altered:
  CR0 (if Rc=1)

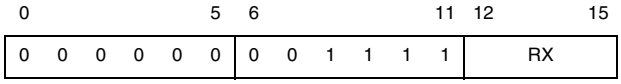
_extsb*x* | VLE | User | **_extsb*x***

Extend Sign (Byte | Halfword)

se_extsb **r**X



se_extsh **r**X



if se_extsb then n ← 56
if se_extsh then n ← 48
if 'extsw' then n ← 32
if Rc=1 then do
LT ← $GPR(RS)_{n:63} < 0$
GT ← $GPR(RS)_{n:63} > 0$
EQ ← $GPR(RS)_{n:63} = 0$
CR0 ← LT || GT || EQ || SO
s ← $GPR(RS \text{ or } RX)_{n}$
$GPR(RA \text{ or } RX) \leftarrow {}^{n-32}s \,||\, GPR(RS \text{ or } RX)_{n:63}$

For **se_extsb**, the contents of bits 56–63 of GPR(**r**X) are placed into bits 56–63 of GPR(**r**X). Bit 56 of the contents of GPR(**r**X) is copied into bits 32–55 of GPR(**r**X).

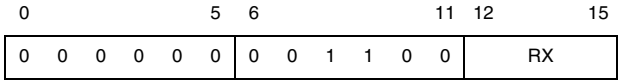
For **se_extsh**, the contents of bits 48–63 of GPR(**r**X) are placed into bits 48–63 of GPR(**r**X). Bit 48 of the contents of GPR(**r**X) is copied into bits 32–47 of GPR(**r**X).

Special Registers Altered: CR0 (if Rc=1)

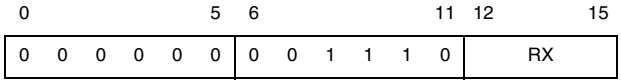
_extz*x* | VLE | User | **_extz*x***

Extend Zero (Byte | Halfword)

se_extzb **r**X



se_extzh **r**X



if ‘se_extzb’ then n ← 56
if ‘se_extzh’ then n ← 48
GPR(RX) ← $^{n-32}0$ || GPR(RX)$_{n:63}$

For **se_extzb**, the contents of bits 56–63 of GPR(**r**X) are placed into bits 56–63 of GPR(**r**X). Bits 32–55 of GPR(**r**X) are cleared.

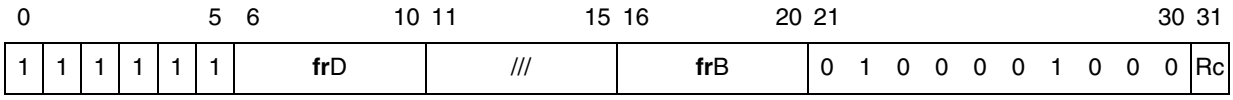
For **se_extzh**, the contents of bits 48–63 of GPR(**r**X) are placed into bits 48–63 of GPR(**r**X). Bits 32–47 of GPR(**r**X) are cleared.

Special Registers Altered: None

fabs | Book E | User | **fabs**

Floating absolute value

fabs **fr**D,**fr**B (Rc=0)
fabs. **fr**D,**fr**B (Rc=1)



frD) ← 0b0||frB$_{1:63}$

The contents of **fr**B with bit 0 cleared are placed into **fr**D.

If MSR[FP]=0, an attempt to execute **fabs**[**.**] causes a floating-point unavailable interrupt.

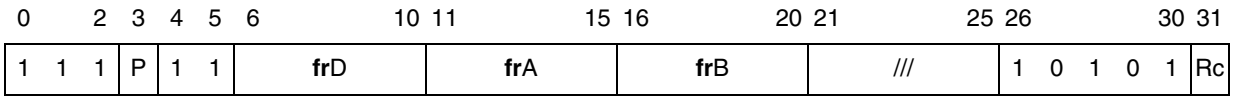
Other registers altered:

- CR1 ← FX || FEX || VX || OX (if Rc=1)

fadd | Book E | User | **fadd**

Floating add [single]

fadd	**fr**D,**fr**A,**fr**B	(P=1, Rc=0)
fadd.	**fr**D,**fr**A,**fr**B	(P=1, Rc=1)
fadds	**fr**D,**fr**A,**fr**B	(P=0, Rc=0)
fadds.	**fr**D,**fr**A,**fr**B	(P=0, Rc=1)



if P=1 then frD ← frA $+_{dp}$ frB
else frD ← frA $+_{sp}$ frB

The floating-point operand in **fr**A is added to the floating-point operand in **fr**B.

If the msb 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, FPSCR[RN], and placed into **fr**D.

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.

If MSR[FP]=0, an attempt to execute **fadd**[**s**][**.**] causes a floating-point unavailable interrupt.

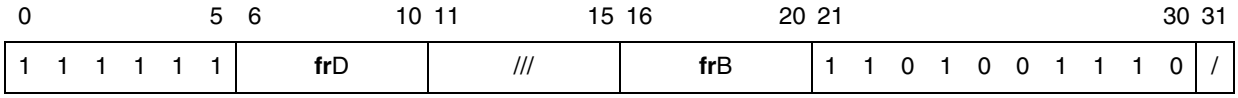
Other registers altered:

- FPRF FR FI FX OX UX XX VXSNAN VXISI
  CR1 ← FX || FEX || VX || OX (if Rc=1)

fcfid | Book E | User | **fcfid**

Floating convert from integer doubleword

fcfid **fr**D,**fr**B




```
                sign ← frB0
                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 loop 0 times if frB = max negative integer */
                        frac0:63 ← frac1:63 || 0b0
                        exp ← exp : 1
                End
                Round Float( sign, exp, frac0:63, FPSCR[RN] )
                If sign = 0 then FPSCR[FPRF] ← '+normal number'
                If sign = 1 then FPSCR[FPRF] ← ':normal number'
                frD0 ← sign
                frD[1-11] ← exp + 1023   /* exp + bias */
                frD[12-63] ← frac1:52
                Done
Zero Operand:
                FPSCR[FR,FI] ← 0b00
                FPSCR[FPRF] ← '+zero'
                frD ← 0x0000_0000_0000_0000
                Done
Round Float( sign, exp, frac0:63, round_mode ):
                inc ← 0
                lsb ← frac52
                gbit ← frac53
                rbit ← frac54
                xbit ← frac55:63 > 0
                If round_mode = 0b00 then
                        Do    /* comparison ignores 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
                        Do    /* comparison ignores 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
                        Do    /* comparison ignores 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
                FPSCR[FR] ← inc
                FPSCR[FI] ← gbit | rbit | xbit
                FPSCR[XX] ← FPSCR[XX] | FPSCR[FI]
                Return
```

The 64-bit signed operand in **fr**B is converted to an infinitely precise floating-point integer. The result of the conversion is rounded to double-precision, as specified by FPSCR[RN], and placed into **fr**D.

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.

If MSR[FP]=0, an attempt to execute **fcfid** causes a floating-point unavailable interrupt.

Other registers altered: FPRF FR FI FX XX

fcmpu | Book E | User | **fcmpu**

Floating compare

fcmpu **cr**D,**fr**A,**fr**B (U=0)
fcmpo **cr**D,**fr**A,**fr**B (U=1)

0	1	2	3	4	5	6	10	11	15	16	20	21	22	23	24	25	26	27	28	29	30	31
1	1	1	1	1	1	**cr**D	//		**fr**A		**fr**B	0	0	0	0	U	0	0	0	0	0	/

```
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×crD:4×crD+3 ← c
if 'fcmpu' & (frA is a SNaN or frB is a SNaN) then VXSNAN ← 1
if 'fcmpo' then do
  if frA is a SNaN or frB is a SNaN then do
    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 **fr**A is compared to the floating-point operand in **fr**B. The result of the compare is placed into CR field **cr**D and the FPCC.

If either of the operands is a NaN, either quiet or signaling, the CR field **cr**D and the FPCC are set to reflect unordered.

If **fcmpu**, then if either of the operands is a signaling NaN, VXSNAN is set.

If **fcmpo**, then do the following:

If either of the operands is a signaling NaN and invalid operation is disabled (VE=0), VXVC is set. If neither operand is a signaling NaN but at least one operand is a quiet NaN, then VXVC is set.

If MSR[FP]=0, an attempt to execute **fcmpo** or **fcmpu** causes a floating-point unavailable interrupt.

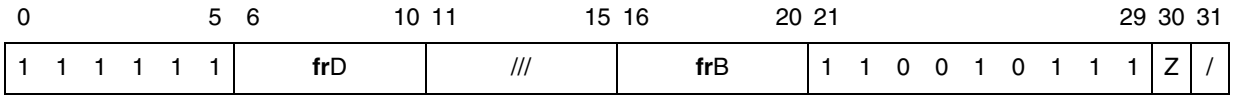
Other registers altered:

- CR field **cr**D
  FPCC FX VXSNAN
  VXVC(if **fcmpo**)

fctid | Book E | User | **fctid**

Floating convert to integer doubleword

fctid **fr**D,**fr**B (Z=0)
fctidz **fr**D,**fr**B (Z=1)



if 'fctid[.]' then round_mode ← FPSCR[RN]
if 'fctidz[.]' then round_mode ← 0b01
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
/* normal; need leading 0 for later complement */
If frB[1:11] > 0 then $frac_{0:64}$ ← 0b01 || frB[12:63] || $^{11}0$
/* denormal */
If frB[1:11] = 0 then $frac_{0:64}$ ← 0b00 || frB[12:63] || $^{11}0$
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)
/* needed leading 0 for $:2^{64} < frB < :2^{63}$ */
If sign=1 then $frac_{0:64}$ ← ¬$frac_{0:64}$ + 1
If $frac_{0:64} > 2^{63}$:1 then goto Large Operand
If $frac_{0:64} < :2^{63}$ then goto Large Operand
FPSCR[XX] ← FPSCR[XX] | FPSCR[FI]
FPSCR[FPRF] ← undefined
frD ← $frac_{1:64}$
Done
Round Integer(sign,frac0:64, gbit, rbit, xbit, round_mode):
inc ← 0
If round_mode = 0b00 then /* comparison ignores u bits */
Do
If sign || $frac_{64}$ || gbit || rbit || xbit = 0bu11uu then inc ← 1
If sign || $frac_{64}$ || gbit || rbit || xbit = 0bu011u then inc ← 1
If sign || $frac_{64}$ || gbit || rbit || xbit = 0bu01u1 then inc ← 1
End
If round_mode = 0b10 then /* comparison ignores u bits */
Do
If sign || $frac_{64}$ || 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    /* comparison ignores u bits */
                        Do
                                        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 sign = 0 then frD ← 0x7FFF_FFFF_FFFF_FFFF
                        If sign = 1 then frD ← 0x8000_0000_0000_0000
                        FPSCR[FPRF] ← undefined
                End
                Done
        SNaN Operand:
                FPSCR[FR,FI,VXSNAN,VXCVI] ← 0b0011
                If FPSCR[VE] = 0 then    Do
                        frD ← 0x8000_0000_0000_0000
                        FPSCR[FPRF] ← undefined
                End
                Done
        QNaN Operand:
                FPSCR[FR,FI,VXCVI] ← 0b001
                If FPSCR[VE] = 0 then    Do
                        frD ← 0x8000_0000_0000_0000
                        FPSCR[FPRF] ← undefined
                End
                Done
        Large Operand:
                FPSCR[FR,FI,VXCVI] ← 0b001
                If FPSCR[VE] = 0 then Do
                        If sign = 0 then frD ← 0x7FFF_FFFF_FFFF_FFFF
                        If sign = 1 then frD ← 0x8000_0000_0000_0000
                        FPSCR[FPRF] ← undefined
                End
                Done
```

For **fctid** or **fctid.**, the rounding mode is specified by FPSCR[RN].

For **fctidz** or **fctidz.**, the rounding mode used is round toward zero.

The floating-point operand in **fr**B is converted to a 64-bit signed integer, using the rounding mode specified by the instruction, and placed into **fr**D.

If the floating-point operand in **fr**B is greater than 2^{63}–1, then 0x7FFF_FFFF_FFFF_FFFF is placed into **fr**D. If the floating-point operand in **fr**B is less than -2^{63}, 0x8000_0000_0000_0000 is placed into **fr**D.

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.

If MSR[FP]=0, an attempt to execute **fctid**[**z**] causes a floating-point unavailable interrupt.

Other registers altered:

- FPRF (undefined) FR FI FX XX VXSNAN VXCVI

fctiw | Book E | User | **fctiw**

Floating convert to integer word

fctiw	**fr**D,**fr**B	(Z=0, Rc=0)
fctiw.	**fr**D,**fr**B	(Z=0, Rc=1)
fctiwz	**fr**D,**fr**B	(Z=1, Rc=0)
fctiwz.	**fr**D,**fr**B	(Z=1, Rc=1)

0 5	6 10	11 15	16 20	21 29	30	31
1 1 1 1 1 1	**fr**D	///	**fr**B	0 0 0 0 0 0 1 1 1	Z	Rc

if ‘fctiw[.]’ then round_mode ← FPSCR[RN]
if ‘fctiwz[.]’ then round_mode ← 0b01
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
/* normal; need leading 0 for later complement */
If frB[1:11] > 0 then $frac_{0:64}$ ← 0b01 || frB[12:63] || $^{11}0$
/* denormal */
If frB[1:11] = 0 then $frac_{0:64}$ ← 0b00 || frB[12:63] || $^{11}0$
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)
/* needed leading 0 for $:2^{64}$ < frB < $:2^{63}$ */
If sign=1 then $frac_{0:64}$ ← ¬$frac_{0:64}$ + 1
If $frac_{0:64}$ > 2^{31}:1 then goto Large Operand
If $frac_{0:64}$ < $:2^{31}$ then goto Large Operand
FPSCR[XX] ← FPSCR[XX] | FPSCR[FI]
frD ← 0xuuuu_uuuu || $frac_{33:64}$ /* u is undefined hex digit */
FPSCR[FPRF] ← undefined
Done
Round Integer(sign, frac0:64, gbit, rbit, xbit, round_mode):
inc ← 0
If round_mode = 0b00 then /* comparison ignores u bits */
Do
If sign || $frac_{64}$ || gbit || rbit || xbit = 0bu11uu then inc ← 1
If sign || $frac_{64}$ || gbit || rbit || xbit = 0bu011u then inc ← 1
If sign || $frac_{64}$ || gbit || rbit || xbit = 0bu01u1 then inc ← 1
End
If round_mode = 0b10 then /* comparison ignores u bits */
Do

```
                                        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    /* comparison ignores u bits */
                        Do
                                        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    /* u is undefined hex digit */
                        If sign = 0 then frD ← 0xuuuu_uuuu_7FFF_FFFF
                        If sign = 1 then frD ← 0xuuuu_uuuu_8000_0000
                        FPSCR[FPRF] ← undefined
                End
                Done
SNaN Operand:
                FPSCR[FR,FI,VXSNAN,VXCVI] ← 0b0011
                If FPSCR[VE] = 0 then    Do    /* u is undefined hex digit */
                        frD ← 0xuuuu_uuuu_8000_0000
                        FPSCR[FPRF] ← undefined
                End
                Done
QNaN Operand:
                FPSCR[FR,FI,VXCVI] ← 0b001
                If FPSCR[VE] = 0 then    Do    /* u is undefined hex digit */
                        frD ← 0xuuuu_uuuu_8000_0000
                        FPSCR[FPRF] ← undefined
                End
                Done
Large Operand:
                FPSCR[FR,FI,VXCVI] ← 0b001
                If FPSCR[VE] = 0 then Do    /* u is undefined hex digit */
                        If sign = 0 then frD ← 0xuuuu_uuuu_7FFF_FFFF
                        If sign = 1 then frD ← 0xuuuu_uuuu_8000_0000
                        FPSCR[FPRF] ← undefined
                End
                Done
```

For **fctiw** or **fctiw.**, the rounding mode is specified by FPSCR[RN].

For **fctiwz** or **fctiwz.**, the rounding mode used is round toward zero.

The floating-point operand in **fr**B is converted to a 32-bit signed integer, using the rounding mode specified by the instruction, and placed into **fr**D[32–63]; **fr**D[0–31] are undefined.

If the operand in **fr**B is greater than $2^{31}-1$, then **fr**D[32–63] are set to 0x7FFF_FFFF. If the operand in **fr**B is less than -2^{31}, then **fr**D[32–63] are set to 0x8000_0000.

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.

If MSR[FP]=0, an attempt to execute **fctiw**[**z**][**.**] causes a floating-point unavailable interrupt.

Other registers altered:

- FPRF (undefined) FR FI FX XX VXSNAN VXCVI
  CR1 ← FX || FEX || VX || OX (if Rc=1)

fdiv | Book E | User | **fdiv**

Floating divide [single]

fdiv	**fr**D,**fr**A,**fr**B	(P=1, Rc=0)
fdiv.	**fr**D,**fr**A,**fr**B	(P=1, Rc=1)
fdivs	**fr**D,**fr**A,**fr**B	(P=0, Rc=0)
fdivs.	**fr**D,**fr**A,**fr**B	(P=0, Rc=1)

0	2	3	4	5	6	10	11	15	16	20	21	25	26	30	31
1 1 1		P	1	1	**fr**D		**fr**A		**fr**B		///		1 0 0 1 0		Rc

if P=1 then frD ← frA $\div_{dp}$ frB
else frD ← frA $\div_{sp}$ frB

The floating-point operand in **fr**A is divided by the floating-point operand in **fr**B. The remainder is not supplied as a result.

If the msb 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, FPSCR[RN], and placed into **fr**D.

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.

If MSR[FP]=0, an attempt to execute **fdiv**[**s**][**.**] causes a floating-point unavailable interrupt.

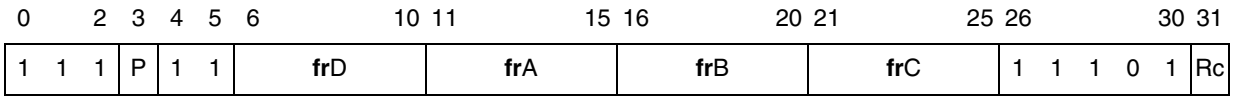
Other registers altered:

- FPRF FR FI FX OX UX ZX XX VXSNAN VXIDI VXZDZ
  CR1 ← FX || FEX || VX || OX (if Rc=1)

fmadd | Book E | User | **fmadd**

Floating multiply-add [single]

fmadd	**fr**D,**fr**A,**fr**C,**fr**B	(P=1, Rc=0)
fmadd.	**fr**D,**fr**A,**fr**C,**fr**B	(P=1, Rc=1)
fmadds	**fr**D,**fr**A,**fr**C,**fr**B	(P=0, Rc=0)
fmadds.	**fr**D,**fr**A,**fr**C,**fr**B	(P=0, Rc=1)



if P=1 then frD ← [frA $\times_{fp}$ frC] $+_{dp}$ frB
else frD ← [frA $\times_{fp}$ frC] $+_{sp}$ frB

The floating-point operand in **fr**A is multiplied by the floating-point operand in **fr**C. The floating-point operand in **fr**B is added to this intermediate result.

If the msb 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, FPSCR[RN], and placed into **fr**D.

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE]=1.

If MSR[FP]=0, an attempt to execute **fmadd**[**s**][**.**] causes a floating-point unavailable interrupt.

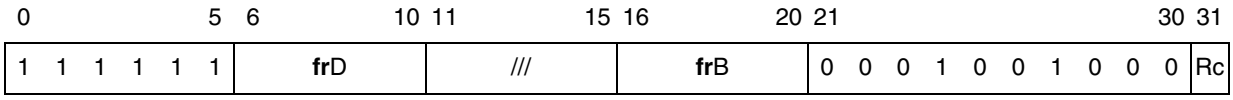
Other registers altered:

- FPRF FR FI FX OX UX XX VXSNAN VXISI VXIMZ
  CR1 ← FX || FEX || VX || OX (if Rc=1)

fmr | Book E | User | **fmr**

Floating move register

fmr **fr**D,**fr**B (Rc=0)
fmr. **fr**D,**fr**B (Rc=1)



frD ← frB

The contents of **fr**B are placed into **fr**D.

If MSR[FP]=0, an attempt to execute **fmr**[**.**] causes a floating-point unavailable interrupt.

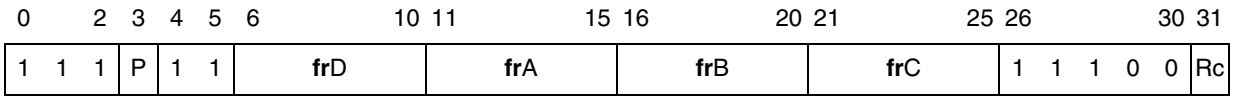
Other registers altered:

- CR1 ← FX || FEX || VX || OX (if Rc=1)

fmsub | Book E | User | **fmsub**

Floating multiply-subtract [single]

fmsub	**fr**D,**fr**A,**fr**C,**fr**B	(P=1, Rc=0)
fmsub.	**fr**D,**fr**A,**fr**C,**fr**B	(P=1, Rc=1)
fmsubs	**fr**D,**fr**A,**fr**C,**fr**B	(P=0, Rc=0**)**
fmsubs.	**fr**D,**fr**A,**fr**C,**fr**B	(P=0, Rc=1)



if P=1 then frD ← [frA $\times_{fp}$ frC] $-_{dp}$ frB
else frD ← [frA $\times_{fp}$ frC] $-_{sp}$ frB

The floating-point operand in **fr**A is multiplied by the floating-point operand in **fr**C. The floating-point operand in **fr**B is subtracted from this intermediate result.

If the msb 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, FPSCR[RN], and placed into **fr**D.

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE]=1.

If MSR[FP]=0, an attempt to execute **fmsub**[**s**][**.**] causes a floating-point unavailable interrupt.

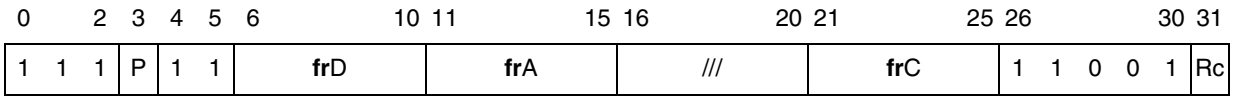
Other registers altered:

- FPRF FR FI FX OX UX XX VXSNAN VXISI VXIMZ
  CR1 ← FX || FEX || VX || OX (if Rc=1)

fmul | Book E | User | **fmul**

Floating multiply [single]

fmul	**fr**D,**fr**A,**fr**C	(P=1, Rc=0)
fmul.	**fr**D,**fr**A,**fr**C	(P=1, Rc=1)
fmuls	**fr**D,**fr**A,**fr**C	(P=0, Rc=0)
fmuls.	**fr**D,**fr**A,**fr**C	(P=0, Rc=1)



if P=1 then frD ← frA $\times_{dp}$ frC
else frD ← frA $\times_{sp}$ frC

The floating-point operand in **fr**A is multiplied by the floating-point operand in **fr**C.

If the msb 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, FPSCR[RN], and placed into **fr**D.

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.

If MSR[FP]=0, an attempt to execute **fmul**[**s**][**.**] causes a floating-point unavailable interrupt.

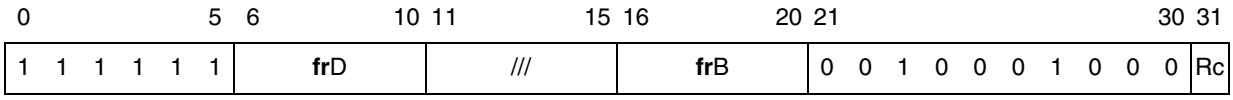
Other registers altered:

- FPRF FR FI FX OX UX XX VXSNAN VXIMZ
  CR1 ← FX || FEX || VX || OX (if Rc=1)

fnabs | Book E | User | **fnabs**

Floating negative absolute value

fnabs **fr**D,**fr**B (Rc=0)
fnabs. **fr**D,**fr**B (Rc=1)



$frD \leftarrow 0b1 || frB_{1:63}$

The contents of **fr**B with bit 0 set are placed into **fr**D.

If MSR[FP]=0, an attempt to execute **fnabs**[**.**] causes a floating-point unavailable interrupt.

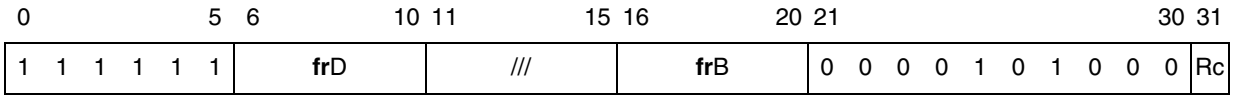
Other registers altered:

- CR1 ← FX || FEX || VX || OX (if Rc=1)

fneg | Book E | User | **fneg**

Floating negate

fneg **fr**D,**fr**B (Rc=0)
fneg. **fr**D,**fr**B (Rc=1)



$frD \leftarrow \neg frB0 || frB_{1:63}$

The contents of **fr**B with bit 0 inverted are placed into **fr**D.

If MSR[FP]=0, an attempt to execute **fneg[.]** causes a floating-point unavailable interrupt.

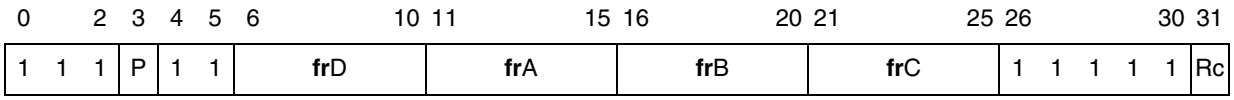
Other registers altered:

- CR1 ← FX || FEX || VX || OX (if Rc=1)

fnmadd | Book E | User | **fnmadd**

Floating negative multiply-add [single]

fnmadd	**fr**D,**fr**A,**fr**C,**fr**B	(P=1, Rc=0)
fnmadd.	**fr**D,**fr**A,**fr**C,**fr**B	(P=1, Rc=1)
fnmadds	**fr**D,**fr**A,**fr**C,**fr**B	(P=0, Rc=0)
fnmadds.	**fr**D,**fr**A,**fr**C,**fr**B	(P=0, Rc=1)



if P=1 then frD ← -([frA $\times_{fp}$ frC] $+_{dp}$ frB)
else frD ← -([frA $\times_{fp}$ frC] $+_{sp}$ frB)

The floating-point operand in **fr**A is multiplied by the floating-point operand in **fr**C. The floating-point operand in **fr**B is added to this intermediate result.

If the msb 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, FPSCR[RN], then negated and placed into **fr**D.

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.

An attempt to execute **fnmadd**[**s**][**.**] causes a floating-point unavailable interrupt.

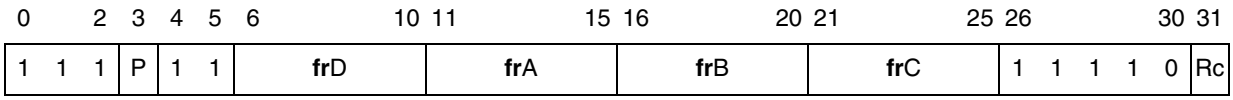
Other registers altered:

- FPRF FR FI FX OX UX XX VXSNAN VXISI VXIMZ
  CR1 ← FX || FEX || VX || OX (if Rc=1)

fnmsub | Book E | User | **fnmsub**

Floating negative multiply-subtract [single]

fnmsub	**fr**D,**fr**A,**fr**C,**fr**B	(P=1, Rc=0)
fnmsub.	**fr**D,**fr**A,**fr**C,**fr**B	(P=1, Rc=1)
fnmsubs	**fr**D,**fr**A,**fr**C,**fr**B	(P=0, Rc=0)
fnmsubs.	**fr**D,**fr**A,**fr**C,**fr**B	(P=0, Rc=1)



if P=1 then frD ← -([frA $\times_{fp}$ frC] :$_{dp}$ frB)
else frD ← -([frA $\times_{fp}$ frC] :$_{sp}$ frB)

The floating-point operand in **fr**A is multiplied by the floating-point operand in **fr**C. The floating-point operand in **fr**B is subtracted from this intermediate result.

If the msb 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, FPSCR[RN], then negated and placed into **fr**D.

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.

An attempt to execute **fnmsub**[**s**][**.**] causes a floating-point unavailable interrupt.

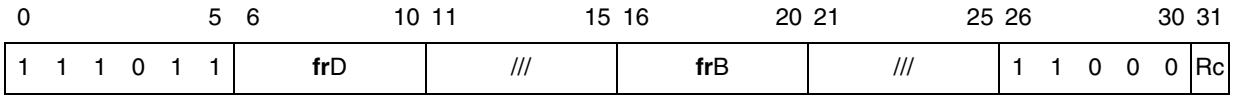
Other registers altered:

- FPRF FR FI FX OX UX XX VXSNAN VXISI VXIMZ
  CR1 ← FX || FEX || VX || OX (if Rc=1)

fres | Book E | User | **fres**

Floating reciprocal estimate single

fres **frD,frB** (Rc=0)
fres. **frD,frB** (Rc=1)



frD ← FPReciprocalEstimate(frB)

A single-precision estimate of the reciprocal of the floating-point operand in **fr**B is placed into **fr**D. The estimate placed into **fr**D is correct to a precision of one part in 256 of the reciprocal of (**fr**B), that is,

$$\left| \frac{\text{estimate} - \frac{1}{x}}{\frac{1}{x}} \right| \leq \frac{1}{256}$$

In this example, x is the initial value in **fr**B. Note that the value placed into **fr**D may vary between implementations, and between different executions on the same implementation.

Operation with various special values of the operand is summarized in *Table 204*.

Table 204. Operations with special values

Operand	Result	Exception
–∞	–0	None
–0	–∞ (No result if FPSCR[ZE] = 1)	ZX
+0	+∞ (No result if FPSCR[ZE] = 1)	ZX
+∞	+0	None
SNaN	QNaN (No result if FPSCR[VE] = 1.)	VXSNAN
QNaN	QNaN	None

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.

If MSR[FP]=0, an attempt to execute **fres**[**.**] causes a floating-point unavailable interrupt.

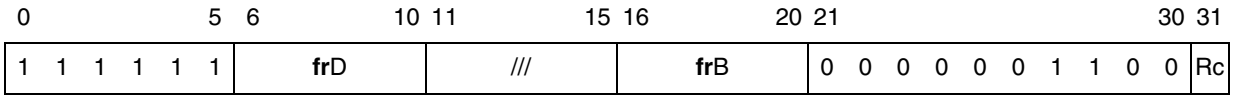
Other registers altered:

- FPRF FR (undefined) FI (undefined)
  FX OX UX ZX VXSNAN
  CR1 ← FX || FEX || VX || OX (if Rc=1)

frsp | Book E | User | **frsp**

Floating round to single-precision

frsp **frD,frB** (Rc=0)
frsp. **frD,frB** (Rc=1)



If frB[1:11] < 897 and $frB_{1:63}$ > 0 then Do
If FPSCR[UE] = 0 then goto Disabled Exponent Underflow
If FPSCR[UE] = 1 then goto Enabled Exponent Underflow
If frB[1:11] > 1150 and frB[1:11] < 2047 then Do
If FPSCR[OE] = 0 then goto Disabled Exponent Overflow
If FPSCR[OE] = 1 then goto Enabled Exponent Overflow
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
Disabled Exponent Underflow:
sign ← frB_0
If frB[1:11] = 0 then Do
exp ← :1022
$frac_{0:52}$ ← 0b0 || frB[12:63]
If frB[1:11] > 0 then Do
exp ← frB[1:11] : 1023
fr ← 0b1 || frB[12:63]
Denormalize operand:
G || R || X ← 0b000
Do while exp < :126
exp ← exp + 1
$frac_{0:52}$ || G || R || X ← 0b0 || $frac_{0:52}$ || G || (R |
X)
FPSCR[UX] ← ($frac_{24:52}$ || G || R || X) > 0
Round Single(sign,exp,$frac_{0:52}$,G,R,X)
FPSCR[XX] ← FPSCR[XX] | FPSCR[FI]
If $frac_{0:52}$ = 0 then Do
frD_0 ← sign
$frD_{1:63}$ ← 0
If sign = 0 then FPSCR[FPRF] ← '+zero'
If sign = 1 then FPSCR[FPRF] ← ':zero'
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'
If $frac_0$ = 0 then Do
If sign = 0 then FPSCR[FPRF] ←

```
‘+denormalized number’
                                        If sign = 1 then FPSCR[FPRF] ←
‘:denormalized number’
                        Normalize operand:
                                        Do while frac0 = 0
                                                exp ← exp:1
                                                frac0:52 ← frac1:52 || 0b0
                        frD0 ← sign
                        frD[1-11] ← exp + 1023
                        frD[12-63] ← frac1:52
                Done
                Enabled exponent underflow:
                        FPSCR[UX] ← 1
                        sign ← frB0
                        If frB[1:11] = 0 then Do
                                        exp ← :1022
                                        frac0:52 ← 0b0 || frB[12:63]
                        If frB[1:11] > 0 then Do
                                        exp ← frB[1:11] : 1023
                                        frac0:52 ← 0b1 || frB[12:63]
                        Normalize operand:
                                        Do while frac0 = 0
                                                exp ← exp : 1
                                                frac0:52 ← frac1:52 || 0b0
                        Round Single(sign,exp,frac0:52,0,0,0)
                        FPSCR[XX] ← FPSCR[XX] | FPSCR[FI]
                        exp ← exp + 192
                        frD0 ← sign
                        frD[1-11] ← exp + 1023
                        frD[12-63] ← frac1:52
                        If sign = 0 then FPSCR[FPRF] ← ‘+normal number’
                        If sign = 1 then FPSCR[FPRF] ← ‘:normal number’
                        Done
                Disabled exponent overflow
                        FPSCR[OX] ← 1
                        If FPSCR[RN] = 0b00 then Do             /* Round to Nearest
*/
                                        If frB0 = 0 then frD ←
0x7FF0_0000_0000_0000
                                        If frB0 = 1 then frD ←
0xFFF0_0000_0000_0000
                                        If frB0 = 0 then FPSCR[FPRF] ← ‘+infinity’
                                        If frB0 = 1 then FPSCR[FPRF] ← ‘:infinity’
                        If FPSCR[RN] = 0b01 then Do             /* Round toward
Zero */
                                        If frB0 = 0 then frD ←
0x47EF_FFFF_E000_0000
                                        If frB0 = 1 then frD ←
0xC7EF_FFFF_E000_0000
                                        If frB0 = 0 then FPSCR[FPRF] ← ‘+normal
number’
                                        If frB0 = 1 then FPSCR[FPRF] ← ‘:normal
```

```
number'
                    If FPSCR[RN] = 0b10 then Do          /* Round toward
+Infinity */
                              If frB0 = 0 then frD ←
0x7FF0_0000_0000_0000
                              If frB0 = 1 then frD ←
0xC7EF_FFFF_E000_0000
                              If frB0 = 0 then FPSCR[FPRF] ← '+infinity'
                              If frB0 = 1 then FPSCR[FPRF] ← ':normal
number'
                    If FPSCR[RN] = 0b11 then Do          /* Round toward
:Infinity */
                              If frB0 = 0 then frD ←
0x47EF_FFFF_E000_0000
                              If frB0 = 1 then frD ←
0xFFF0_0000_0000_0000
                              If frB0 = 0 then FPSCR[FPRF] ← '+normal
number'
                              If frB0 = 1 then FPSCR[FPRF] ← ':infinity'
                    FPSCR[FR] ← undefined
                    FPSCR[FI] ← 1
                    FPSCR[XX] ← 1
                    Done
          Enabled Exponent Overflow:
                    sign ← frB0
                    exp ← frB[1:11] : 1023
                    frac0:52 ← 0b1 || frB[12:63]
                    Round Single(sign,exp,frac0:52,0,0,0)
                    FPSCR[XX] ← FPSCR[XX] | FPSCR[FI]
          Enabled Overflow:
                    FPSCR[OX] ← 1
                    exp ← exp : 192
                    frD0 ← sign
                    frD[1-11] ← exp + 1023
                    frD[12-63] ← frac1:52
                    If sign = 0 then FPSCR[FPRF] ← '+normal number'
                    If sign = 1 then FPSCR[FPRF] ← ':normal number'
                    Done
          Zero Operand:
                    frD ← frB
                    If frB0 = 0 then FPSCR[FPRF] ← '+zero'
                    If frB0 = 1 then FPSCR[FPRF] ← ':zero'
                    FPSCR[FR,FI] ← 0b00
                    Done
          Infinity Operand:
                    frD ← frB
                    If frB0 = 0 then FPSCR[FPRF] ← '+infinity'
                    If frB0 = 1 then FPSCR[FPRF] ← ':infinity'
                    FPSCR[FR,FI] ← 0b00
                    Done
          QNaN Operand:
                    frD ← frB0:34 || 29 0
```

```
                                FPSCR[FPRF] ← 'QNaN'
                                FPSCR[FR,FI] ← 0b00
                                Done
                        SNaN Operand:
                                FPSCR[VXSNAN] ← 1
                                If FPSCR[VE] = 0 then Do
                                        frD[0:11] ← frB[0:11]
                                        frD12 ← 1
                                        frD[13:63] ← frB[13:34] || 29 0
                                        FPSCR[FPRF] ← 'QNaN'
                                FPSCR[FR,FI] ← 0b00
                                Done
                        Normal Operand:
                                sign ← frB0
                                exp ← frB[1:11] : 1023
                                frac0:52 ← 0b1 || frB[12:63]
                                Round Single(sign,exp,frac0:52,0,0,0)
                                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
                                frD0 ← sign
                                frD[1-11] ← exp + 1023
                                frD[12-63] ← frac1:52
                                If sign = 0 then FPSCR[FPRF] ← '+normal number'
                                If sign = 1 then FPSCR[FPRF] ← ':normal number'
                                Done
                        Round Single(sign,exp,frac0:52,G,R,X):
                                inc ← 0
                                lsb ← frac23
                                gbit ← frac24
                                rbit ← frac25
                                xbit ← (frac26:52||G||R||X)≠0
                                If FPSCR[RN] = 0b00 then Do    /* comparison ignores 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
                                If FPSCR[RN] = 0b10 then Do    /* comparison ignores 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
                                If FPSCR[RN] = 0b11 then Do    /* comparison ignores 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
                        frac0:23 ← frac0:23 + inc
                        If carry_out = 1 then Do
                                        frac0:23 ← 0b1 || frac0:22
                                        exp ← exp + 1
                        frac24:52 ← 29 0
                        FPSCR[FR] ← inc
                        FPSCR[FI] ← gbit | rbit | xbit
                        Return
```

The floating-point operand in **fr**B is rounded to single-precision, using the rounding mode specified by FPSCR[RN], and placed into **fr**D.

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE]=1.

If MSR[FP]=0, an attempt to execute **frsp**[**.**] causes a floating-point unavailable interrupt.

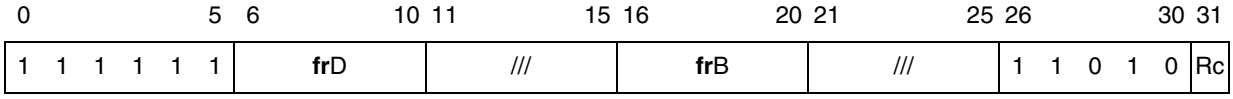
Other registers altered:

- FPRF FR FI FX OX UX XX VXSNAN
  CR1 ← FX || FEX || VX || OX (if Rc=1)

frsqrte | Book E | User | **frsqrte**

Floating reciprocal square root estimate

frsqrte **fr**D,**fr**B (Rc=0)
frsqrte. **fr**D,**fr**B (Rc=1)



frD ← FPReciprocalSquareRootEstimate(frB)

A double-precision estimate of the reciprocal of the square root of the floating-point operand in **fr**B is placed into **fr**D. The estimate is correct to a precision of one part in 32 of the reciprocal of the square root of (**fr**B), that is,

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

Here, x is the initial value in **fr**B. Note that the value placed into **fr**D may vary between implementations, and between different executions on the same implementation.

Operation with various special values of the operand is summarized in *Table 205*

Table 205. Operations with special values

Operand	Result	Exception
–∞	QNaN (No result if FPSCR[VE] = 1.)	VXSQRT
< 0	QNaN (No result if FPSCR[VE] = 1.)	VXSQRT
–0	–∞ (No result if FPSCR[ZE] = 1.)	ZX
+0	+∞ (No result if FPSCR[ZE] = 1.)	ZX
+∞	+0	None
SNaN	QNaN (No result if FPSCR[VE] = 1.)	VXSNAN
QNaN	QNaN	None

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.

If MSR[FP]=0, attempting to execute **frsqrte[.]** causes a floating-point unavailable interrupt.

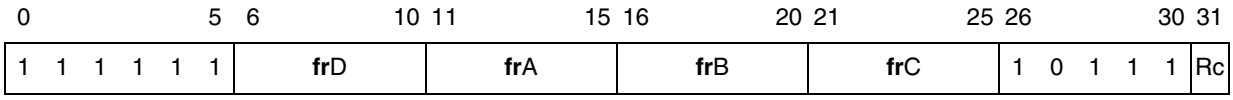
Other registers altered:

- FPRF FR (undefined) FI (undefined)
  FX ZX VXSNAN VXSQRT
  CR1 ← FX || FEX || VX || OX (if Rc=1)

fsel | Book E | User | **fsel**

Floating select

fsel **fr**D,**fr**A,**fr**C,**fr**B (Rc=0)
fsel. **fr**D,**fr**A,**fr**C,**fr**B (Rc=1)



if frA ≥ 0.0 then frD ← frC
else frD ← frB

The floating-point operand in **fr**A is compared to the value zero. If the operand is greater than or equal to zero, **fr**D is set to the contents of **fr**C. If the operand is less than zero or is a NaN, **fr**D is set to the contents of **fr**B. The comparison ignores the sign of zero (that is, +0 and –0 are regarded as equal).

If MSR[FP]=0, an attempt to execute **fsel[.]** causes a floating-point unavailable interrupt.

Other registers altered:

- CR1 ← FX || FEX || VX || OX (if Rc=1)

Note: Programming: Examples of uses of this instruction can be found in the appendix

Warning: Care must be taken in using fsel if IEEE compatibility is required, or if the values being tested can be NaNs or infinities

fsqrt | Book E | User | **fsqrt**

Floating square root [single]

fsqrt	**fr**D,**fr**B	(P=1, Rc=0)
fsqrt.	**fr**D,**fr**B	(P=1, Rc=1)
fsqrts	**fr**D,**fr**B	(P=0, Rc=0)
fsqrts.	**fr**D,**fr**B	(P=0, Rc=1)

0	2	3	4	5	6	10	11	15	16	20	21	25	26	30	31
1 1 1		P	1	1	**fr**D		///		**fr**B		///		1 0 1 1 0		Rc

```
if P=1 then frD ← FPSquareRootDouble( frB )
else        frD ← FPSquareRootSingle( frB )
```

The square root of the floating-point operand in **fr**B is placed into **fr**D.

If the msb 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, FPSCR[RN], and placed into **fr**D.

Operation with various special values of the operand is summarized in *Table 206*

Table 206. Operations with special values

Operand	Result	Exception
–∞	QNaN (No result if FPSCR[VE] = 1)	VXSQRT
< 0	QNaN (No result if FPSCR[VE] = 1)	VXSQRT
–0	–0	None
+∞	+∞	None
SNaN	QNaN(No result if FPSCR[VE] = 1)\	VXSNAN
QNaN	QNaN	None

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE]=1.

If MSR[FP]=0, an attempt to execute **fsqrt**[**s**][**.**] causes a floating-point unavailable interrupt.

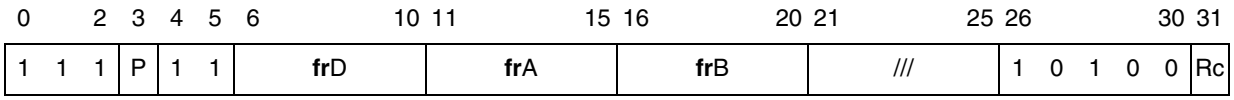
Other registers altered:

- FPRF FR FI FX XX VXSNAN VXSQRT
  CR1 ← FX || FEX || VX || OX (if Rc=1)

fsub | Book E | User | **fsub**

Floating subtract [single]

fsub	**fr**D,**fr**A,**fr**B	(P=1, Rc=0)
fsub.	**fr**D,**fr**A,**fr**B	(P=1, Rc=1)
fsubs	**fr**D,**fr**A,**fr**B	(P=0, Rc=0)
fsubs.	**fr**D,**fr**A,**fr**B	(P=0, Rc=1)



if P=1 then frD ← frA $-_{dp}$ frB
else frD ← frA $-_{sp}$ frB

The floating-point operand in **fr**B is subtracted from the floating-point operand in **fr**A.

If the msb 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, FPSCR[RN]. and placed into **fr**D.

The execution of the Floating Subtract instruction is identical to that of Floating Add, except that the contents of **fr**B 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.

If MSR[FP]=0, an attempt to execute **fsub**[**s**][**.**] causes a floating-point unavailable interrupt.

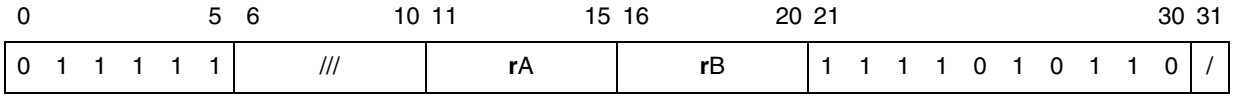
Other registers altered:

- FPRF FR FI FX OX UX XX VXSNAN VXISI
  CR1 ← FX || FEX || VX || OX (if Rc=1)

icbi | Book E | User | **icbi**

Instruction cache block invalidate

icbi **r**A,**r**B



if rA=0 then a ← $^{64}0$ else a ← rA
EA ← $^{32}0$ || $(a + rB)_{32:63}$
InvalidateInstructionCacheBlock(EA)

EA calculation: Addressing Mode EA for **r**A=0 EA for **r**A≠0
$^{32}0$ || $\mathbf{r}B_{32:63}$ $^{32}0$ || $(\mathbf{r}A+\mathbf{r}B)_{32:63}$

If the block containing the byte addressed by EA is in memory 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, so that subsequent references cause the block to be fetched from main memory.

If the block containing the byte addressed by EA is in memory 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, so that subsequent references cause the block to be fetched from main memory.

The function of this instruction is independent of whether the block containing the byte addressed by EA is in memory that is write-through required or caching-inhibited.

This instruction is treated as a load.

icbi may cause a cache-locking exception on some implementations. See the implementation documentation.

On some implementations, HID1[ABE] must be set to allow management of external L2 caches (for implementations with L2 caches) as well as other L1 caches in the system.

Other registers altered: None

icblc | Cache locking APU | User | **icblc**

Instruction cache block lock clear

icblc CT,**r**A,**r**B Form: X

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	1	1	1	1	1	CT					**r**A					**r**B					0	0	1	1	1	0	0	1	1	0	/

if **r**A = 0 then a ← $^{64}0$ else a ← GPR(**r**A)
if Mode32 then EA ← $^{32}0$ || (a + GPR(**r**B))$_{32:63}$
if Mode64 then EA ← a + GPR(**r**B)
InstructionCacheBlockClearLock(CT, EA)

EA calculation: EA for **r**A=0 EA for **r**A≠0

$^{32}0$ || GPR(rB)$_{32:63}$ $^{32}0$ || (GPR(rA)+GPR(rB))$_{32:63}$

The instruction cache specified by CT has the cache line corresponding to EA unlocked allowing the line to participate in the normal replacement policy.

Cache lock clear instructions remove locks previously set by cache lock set instructions.

User-level cache instructions on page 180, lists supported CT values. An implementation may use other CT values to enable software to target specific, implementation-dependent portions of its cache hierarchy or structure.

The **icbtlc** instruction requires read (R) or execute (X) permissions with respect to translation and memory protection and can cause DSI and DTLB error interrupts accordingly.

An unable-to-unlock condition is said to occur any of the following conditions exist:

- The target address is marked cache-inhibited, or the storage attributes of the address uses 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.
- The target address is not in the cache or is present in the cache but is not locked.

If an unable-to-unlock condition occurs, no cache operation is performed.

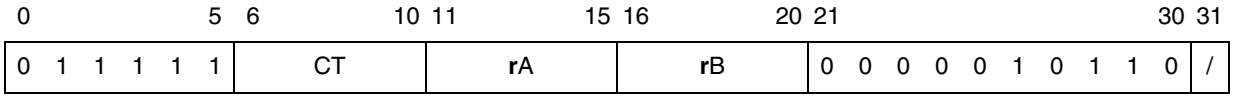
EIS specifics

Setting L1CSR1[ICLFI] allows system software to clear all L1 instruction cache locking bits without knowing the addresses of the lines locked.

icbt | Book E | User | **icbt**

Instruction cache block touch

icbt CT,**r**A,**r**B



if rA=0 then a ← $^{64}0$ else a ← rA
EA ← $^{32}0$ || $(a + rB)_{32:63}$
PrefetchInstructionCacheBlock(CT, EA)

EA calculation:

Addressing Mode	EA for **r**A=0	EA for **r**A≠0
	$^{32}0$ || $\mathbf{r}B_{32:63}$	$^{32}0$ || $(\mathbf{r}A+\mathbf{r}B)_{32:63}$

This instruction is a hint that performance would likely be improved if the block containing the byte addressed by EA is fetched into the instruction cache, because the program will probably soon execute code from the addressed location.

User-level cache instructions on page 180," lists supported CT values. An implementation may use other CT values to enable software to target specific, implementation-dependent portions of its cache hierarchy or structure.

Implementations should perform no operation when CT specifies a value not supported by the implementation.

The hint is ignored if the block is caching-inhibited.

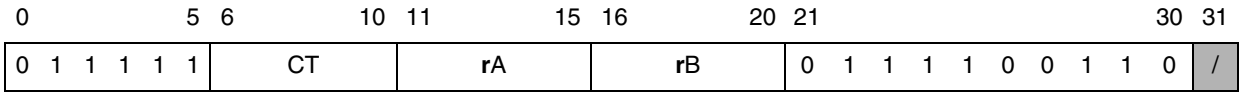
This instruction treated as a load (see the discussion of cache and MMU operation in the user's manual), except that an interrupt is not taken for a translation or protection violation.

Other registers altered: None

icbtls | Cache locking APU | User | **icbtls**

Instruction cache block touch and lock set

icbtls CT,**r**A,**r**B Form: X



if **r**A = 0 then a ← $^{64}0$ else a ← GPR(**r**A)
if Mode32 then EA ← $^{32}0$ || (a + GPR(**r**B))$_{32:63}$
if Mode64 then EA ← a + GPR(**r**B)
PrefetchInstructionCacheBlockLockSet(CT, EA)
EA calculation: EA for **r**A=0 EA for **r**A≠0
$^{32}0$ || GPR(rB)$_{32:63}$ $^{32}0$ || (GPR(rA)+GPR(rB))$_{32:63}$

The instruction cache specified by CT has the line corresponding to EA loaded and locked. If the line exists in the cache, it is locked without refetching from memory.

Cache touch and lock set instructions allow software to lock lines into the cache to shorten latency for critical cache accesses and more deterministic behavior. Lines locked in the cache do not participate in the normal replacement policy when a line must be victimized for replacement.

User-level cache instructions on page 180," lists supported CT values. An implementation may use other CT values to enable software to target specific, implementation-dependent portions of its cache hierarchy or structure.

The **icbtls** requires read (R) or execute (X) permissions for translation and memory protection and can cause DSI and DTLB error interrupts accordingly.

For unable-to-lock conditions, described in *Unable-to-lock conditions on page 849*," no cache operation is performed and LICSR0[ICUL] is set.

An overlocking condition is said to exist is all the available ways for a given cache index are already locked. If an overlocking condition occurs for a **icbtls** instruction and if the lock was targeted for the primary cache or secondary cache (CT = 0 or CT = 2), the requested line is not locked into the cache. When an overlock condition occurs, L1CSR1[ICLO] (L2CSR[L2CLO] for CT = 2) is set. If L1CSR1[ICLOA] is set (or L2CSR[L2CLOA] for CT = 2), the requested line is locked into the cache and implementation dependent line currently locked in the cache is evicted.

Results of overlocking and unable-to-lock conditions for caches other than the primary and secondary cache are defined as part of the architecture for the cache hierarchy designated by CT.

If a unified primary cache is implemented and L1CSR1 is not implemented, L1CSR0[DCUL] and L1CSR0[DCLO] are updated instead of the corresponding L1CSR1 bits.

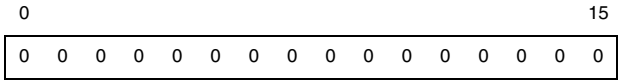
Other registers altered:

- L1CSR1[ICUL] if unable to lock occurs
- L1CSRI[ICLO] (L2CSR[L2CLO]) if lock overflow occurs

_illegal | VLE | User | **_illegal**

Illegal

se_illegal



SRR1 ← MSR

SRR0 ← CIA

NIA ← $IVPR_{32:47}$ || $IVOR6_{48:59}$ || 0b0000

$MSR_{WE,EE,PR,IS,DS,FP,FE0,FE1}$ ← 0b0000_0000

se_illegal is used to request an illegal instruction exception. A program interrupt is generated. The contents of the MSR are copied into SRR1 and the address of the **se_illegal** instruction is placed into SRR0.

MSR[WE,EE,PR,IS,DS,FP,FE0,FE1] are cleared.

The interrupt causes the next instruction to be fetched from address IVPR[32–47]||IVOR6[48–59]||0b0000

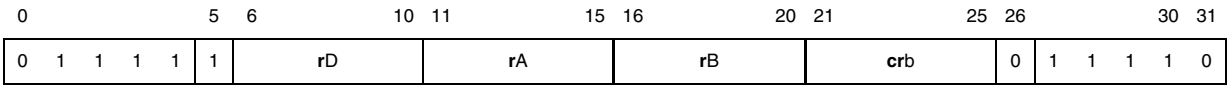
This instruction is context synchronizing.

Special Registers Altered: SRR0 SRR1 MSR[WE,EE,PR,IS,DS,FP,FE0,FE1]

isel | Integer Select APU | User | **isel**

Integer Select

isel **r**D, **r**A, **r**B, **cr**b



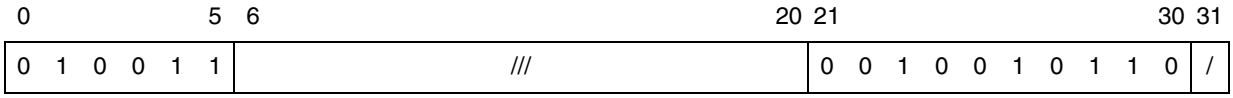
if (**r**A = 0) then a ← ^{64}0 else a ← GPR(**r**A)
c ← $cr_{\mathbf{cr}b + 32}$
if c then **r**D ← a
else **r**D ← GPR(**r**B)

If CR[**cr**b + 32] is set, the contents of **r**A|0 are copied into **r**D. If CR[**cr**b + 32] is clear, the contents of **r**B are copied into **r**D.

isync | Book E | User | **isync**

Instruction synchronize

isync



isync provides an ordering function for the effects of all instructions executed by the processor executing the **isync** instruction. Executing an **isync** ensures that all instructions preceding the **isync** have completed before **isync** completes, and that no subsequent instructions are initiated until after **isync** completes. It also causes any prefetched instructions to be discarded, with the effect that subsequent instructions are fetched and executed in the context established by the instructions preceding **isync**.

isync may complete before memory accesses associated with instructions preceding **isync** have been performed.

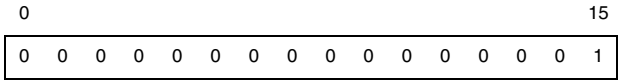
isync is context synchronizing. See *Context synchronization on page 144*."

Other registers altered: None

_isync | VLE | User | **_isync**

Instruction Synchronize

se_isync



The **se_isync** instruction provides an ordering function for the effects of all instructions executed by the processor executing the **se_isync** instruction. 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 causes any prefetched instructions to be discarded, with the effect that subsequent instructions are fetched and executed in the context established by the instructions preceding the **se_isync** instruction.

The **se_isync** instruction may complete before memory accesses associated with instructions preceding the **se_isync** instruction have been performed.

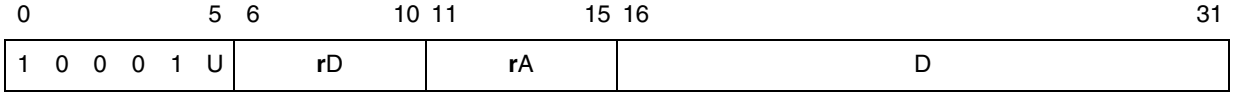
This instruction is context synchronizing (see Book E). It has identical semantics to Book E **isync**, just a different encoding.

Special Registers Altered: None

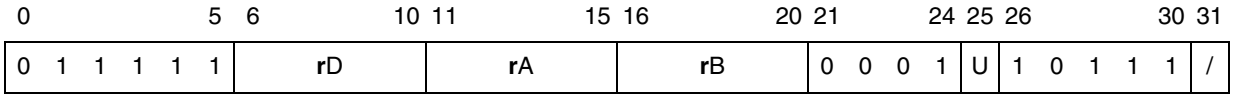
lbz | Book E | User | **lbz**

Load byte and zero [with update] [indexed]

lbz **r**D,D(**r**A) (D-mode, U=0)
lbzu **r**D,D(**r**A) (D-mode, U=1)



lbzx **r**D,**r**A,**r**B (X-mode, U=0)
lbzux **r**D,**r**A,**r**B (X-mode, U=1)



if rA=0 then a ← $^{64}0$ else a ← rA
if D-mode then EA ← $^{32}0$ || (a + EXTS(D))$_{32:63}$
if X-mode then EA ← $^{32}0$ || (a + rB)$_{32:63}$
rD ← $^{56}0$ || MEM(EA,1)
if U=1 then rA ← EA

The EA is calculated as follows:

- For **lbz** and **lbzu,** EA is bits 32–63 of the sum of the contents of **r**A, or 64 zeros if **r**A=0, and the sign-extended value of the D field.
- For **lbzx** and **lbzux**, EA is bits 32–63 of the sum of the contents of **r**A, or 64 zeros if **r**A=0, and the contents of **r**B.

The byte in memory addressed by EA is loaded into **r**D[56–63]; **r**D[0–55] are cleared.

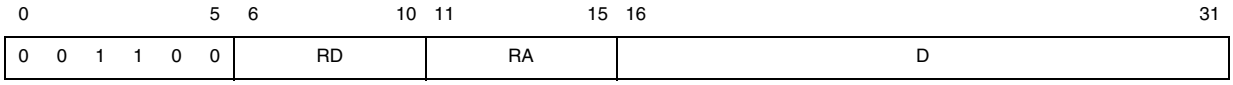
If U=1 (with update), EA is placed into **r**A. If U=1 (with update), and **r**A=0 or **r**A=**r**D, the instruction form is invalid.

Other registers altered: None

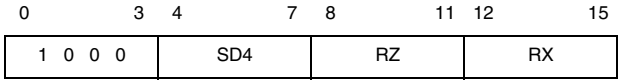
_lbz*x* | VLE | User | **_lbz*x***

Load Byte and Zero [with Update] [Indexed]

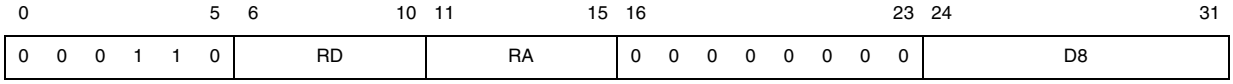
e_lbz **r**D,D(**r**A) (D-mode)



se_lbz **r**Z,SD4(**r**X) (SD4-mode)



e_lbzu **r**D,D8(**r**A) (D8-mode)



if (RA=0 & !se_lbz) then a ← $^{32}0$ else a ← GPR(RA or RX)

if D-mode then EA ← $(a + EXTS(D))_{32:63}$

if D8-mode then EA ← $(a + EXTS(D8))_{32:63}$

if SD4-mode then EA ← $(a + (^{28}0 \,||\, SD4))_{32:63}$

GPR(RD or RZ) ← $^{24}0$ || MEM(EA,1)

if e_lbzu then GPR(RA) ← EA

Let the EA be calculated as follows:

- For **e_lbz** and **e_lbzu**, let EA be the sum of the contents of GPR(**r**A), or 32 0s if **r**A = 0, and the sign-extended value of the D or D8 instruction field.
- For **se_lbz**, let EA be the sum of the contents of GPR(**r**X) and the zero-extended value of the SD4 instruction field.

The byte in memory addressed by EA is loaded into bits 56–63 of GPR(**r**D or **r**Z). Bits 32–55 of GPR(**r**D or **r**Z) are cleared.

If **e_lbzu**, EA is placed into GPR(**r**A).

If **e_lbzu** and **r**A = 0 or **r**A = **r**D, the instruction form is invalid.

Special Registers Altered: None

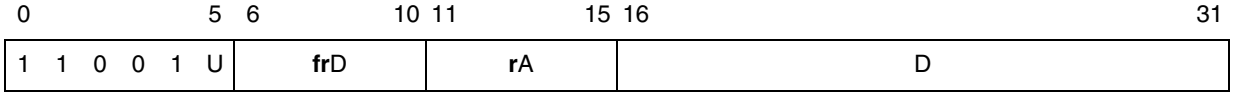
lfd

Book E	User

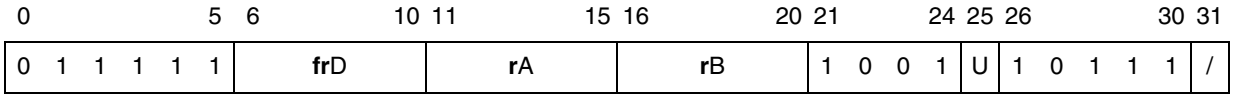
lfd

Load floating-point double

lfd **fr**D,D(**r**A) (D-mode, U=0)
lfdu **fr**D,D(**r**A) (D-mode, U=1)



lfdx **fr**D,**r**A,**r**B (X-mode, U=0)
lfdux **fr**D,**r**A,**r**B (X-mode, U=1)



if rA=0 then a ← $^{64}0$ else a ← rA
if D-mode then EA ← $^{32}0$ || $(a + EXTS(D))_{32:63}$
if X-mode then EA ← $^{32}0$ || $(a + rB)_{32:63}$
frD ← MEM(EA,8)
if U=1 then rA ← EA

The EA is calculated as follows:

- For **lfd** and **lfdu**, EA is 32 zeros concatenated with bits 32–63 of the sum of the contents of **r**A, or 64 zeros if **r**A=0, and the sign-extended value of the D field.
- For **lfdx** and **lfdux**, EA is 32 zeros concatenated with bits 32–63 of the sum of the contents of **r**A, or 64 zeros if **r**A=0, and the contents of **r**B.

The double word addressed by EA is placed into **fr**D.

If U=1 (with update), EA is placed into register **r**A.
If U=1 (with update) and **r**A=0, the instruction form is invalid.

If MSR[FP]=0, an attempt to execute **lfd**[**u**][**x**] causes a floating-point unavailable interrupt.

Other registers altered: None

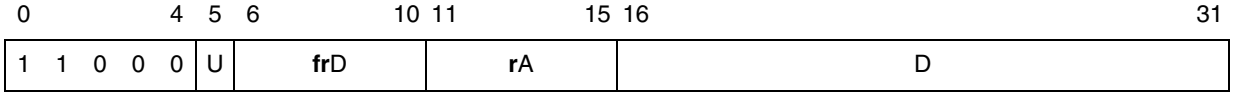
lfs

Book E	User

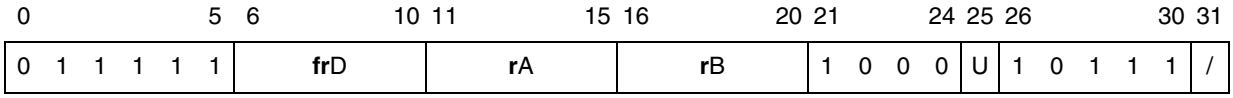
lfs

Load floating-point single

lfs **fr**D,D(**r**A) (D-mode, U=0)
lfsu **fr**D,D(**r**A) (D-mode, U=1)



lfsx **fr**D,**r**A,**r**B (X-mode, U=0)
lfsux **fr**D,**r**A,**r**B (X-mode, U=1)



if rA=0 then a ← $^{64}0$ else a ← rA
if D-mode then EA ← $^{32}0$ || $(a + EXTS(D))_{32:63}$
if X-mode then EA ← $^{32}0$ || $(a + rB)_{32:63}$
frD ← DOUBLE(MEM(EA,4))
if U=1 then rA ← EA

The EA is calculated as follows:

- For **lfs** and **lfsu**, EA is 32 zeros concatenated with bits 32–63 of the sum of the contents of **r**A, or 64 zeros if **r**A=0, and the sign-extended value of the D field.
- For **lfsx** and **lfsux**, EA is 32 zeros concatenated with bits 32–63 of the sum of the contents of **r**A, or 64 zeros if **r**A=0, and the contents of **r**B.

The word addressed by EA is interpreted as a single-precision operand, converted to floating-point double format, and placed into **fr**D.

If U=1 (with update), EA is placed into register **r**A.
If U=1 (with update) and **r**A=0, the instruction form is invalid.

If MSR[FP]=0, an attempt to execute **lfs**[**u**][**x**] causes a floating-point unavailable interrupt.

Other registers altered: None

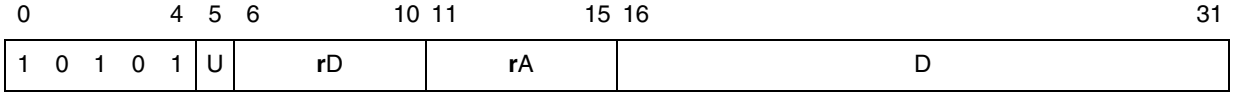
lha

Book E	User

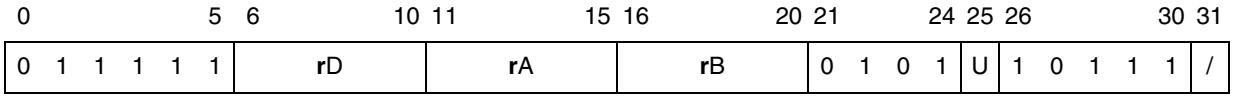
lha

Load half word algebraic [with update] [indexed]

lha **r**D,D(**r**A) (D-mode, U=0)
lhau **r**D,D(**r**A) (D-mode, U=1)



lhax **r**D,**r**A,**r**B (X-mode, U=0)
lhaux **r**D,**r**A,**r**B (X-mode, U=1)



if rA=0 then a ← $^{64}0$ else a ← rA
if D-mode then EA ← $^{32}0$ || $(a + EXTS(D))_{32:63}$
if X-mode then EA ← $^{32}0$ || $(a + rB)_{32:63}$
rD ← $^{32}0$ || $EXTS(MEM(EA,2))_{32:63}$
if U=1 then rA ← EA

The EA is calculated as follows:

- For **lha** and **lhau**, EA is bits 32–63 of the sum of the contents of **r**A, or 64 zeros if **r**A=0, and the sign-extended value of the D field.
- For **lhax** and **lhaux**, EA is bits 32–63 of the sum of the contents of **r**A, or 64 zeros if **r**A=0, and the contents of **r**B.

The half word addressed by EA is loaded into **r**D[48–63]. **r**D[32–47] are filled with a copy of bit 0 of the loaded half word. Bits **r**D[0–31] are cleared.

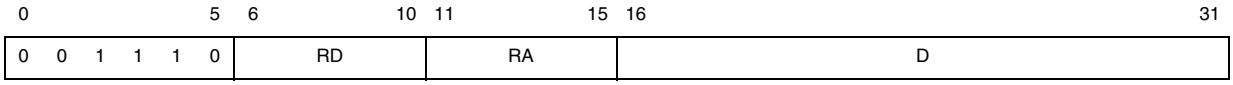
If U=1 (with update), EA is placed into **r**A.
If U=1 (with update), and **r**A=0 or **r**A=**r**D, the instruction form is invalid.

Other registers altered: None

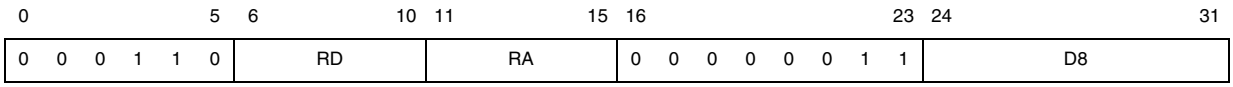
_lha*x* | VLE | User | **_lha*x***

Load Halfword Algebraic [with Update] [Indexed]

e_lha **r**D,D(**r**A) (D-mode)



e_lhau **r**D,D8(**r**A) (D8-mode)



if RA=0 then a ← $^{32}0$ else a ← GPR(RA)

if D-mode then EA ← $(a + EXTS(D))_{32:63}$

if D8-mode then EA ← $(a + EXTS(D8))_{32:63}$

$GPR(RD) \leftarrow EXTS(MEM(EA,2))_{32:63}$

if e_lhau then GPR(RA) ← EA

Let the EA be calculated as follows:

- For **e_lha** and **e_lhau**, let EA be the sum of the contents of GPR(**r**A), or 32 0s if **r**A = 0, and the sign-extended value of the D or D8 instruction field.

The half word in memory addressed by EA is loaded into bits 48–63 of GPR(**r**D). Bits 32–47 of GPR(**r**D) are filled with a copy of bit 0 of the loaded half word.

If **e_lhau**, EA is placed into GPR(**r**A).

If **e_lhau** and **r**A = 0 or **r**A = **r**D, the instruction form is invalid.

Special Registers Altered: None

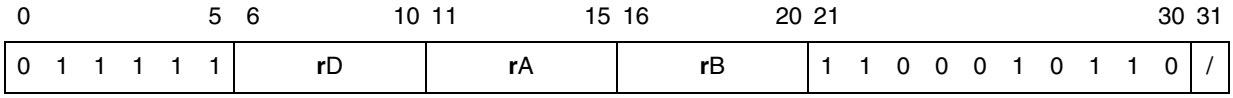
lhbrx

Book E	User

lhbrx

Load half word byte-reverse indexed

lhbrx **r**D,**r**A,**r**B



if rA=0 then a ← $^{64}0$ else a ← rA
EA ← $^{32}0$ || $(a + rB)_{32:63}$
$data_{0:15}$ ← MEM(EA,2)
rD ← $^{48}0$ || $data_{8:15}$ || $data_{0:7}$

The EA is calculated as follows:

- For **lhbrx**, EA is bits 32–63 of the sum of the contents of **r**A, or 64 zeros if **r**A=0, and the contents of **r**B.

Bits 0–7 of the half word addressed by EA are loaded into **r**D[56–63]. Bits 8–15 of the half word addressed by EA are loaded into **r**D[48–55]; **r**D[0–47] are cleared.

Other registers altered: None

Programming notes:

- When EA references big-endian memory, these instructions have the effect of loading data in little-endian byte order. Likewise, when EA references little-endian memory, these instructions have the effect of loading data in big-endian byte order.
- In some implementations, the Load Half Word Byte-Reverse Indexed instructions may have greater latency than other load instructions.

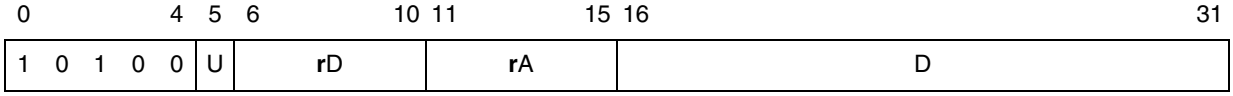
lhz

Book E	User

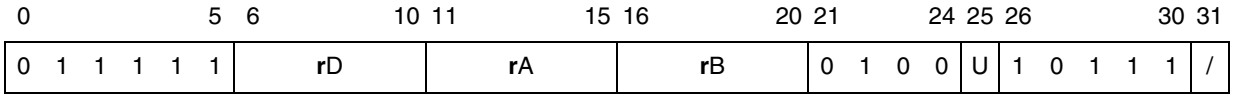
lhz

Load half word and zero [with update] [indexed]

lhz **r**D,D(**r**A) (D-mode, U=0)
lhzu **r**D,D(**r**A) (D-mode, U=1)



lhzx **r**D,**r**A,**r**B (X-mode, U=0)
lhzux **r**D,**r**A,**r**B (X-mode, U=1)



if rA=0 then a ← $^{64}0$ else a ← rA
if D-mode then EA ← $^{32}0$ || (a + EXTS(D))$_{32:63}$
if X-mode then EA ← $^{32}0$ || (a + rB)$_{32:63}$
rD ← $^{48}0$ || MEM(EA,2)
if U=1 then **r**A ← EA

The EA is calculated as follows:

- For **lhz** and **lhzu**, EA is bits 32–63 of the sum of the contents of **r**A, or 64 zeros if **r**A=0, and the sign-extended value of the D field.
- For **lhzx** and **lhzux**, EA is bits 32–63 of the sum of the contents of **r**A, or 64 zeros if **r**A=0, and the contents of **r**B.

The half word addressed by EA is loaded into **r**D[48–63]; **r**D[0–47] are cleared.

If U=1 (with update), EA is placed into **r**A.

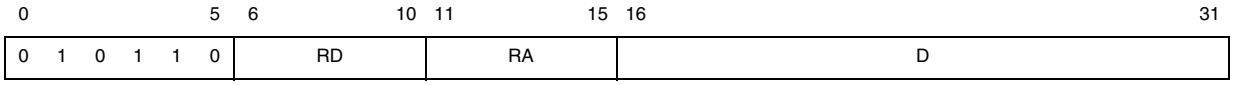
If U=1 (with update), and **r**A=0 or **r**A=**r**D, the instruction form is invalid.

Other registers altered: None

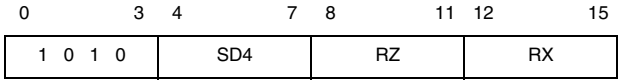
_lhz*x* | VLE | User | **_lhz*x***

Load Halfword and Zero [with Update] [Indexed]

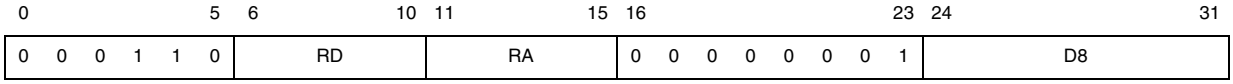
e_lhz **r**D,D(**r**A) (D-mode)



se_lhz **r**Z,SD4(**r**X) (SD4-mode)



e_lhzu **r**D,D8(**r**A) (D8-mode)



if (RA=0 & !se_lhz) then a ← $^{32}0$ else a ← GPR(RA or RX)

if D-mode then EA ← $(a + EXTS(D))_{32:63}$

if D8-mode then EA ← $(a + EXTS(D8))_{32:63}$

if SD4-mode then EA ← $(a + (^{27}0 \,||\, SD4 \,||\, 0))_{32:63}$

GPR(RD or RZ) ← $^{16}0$ || MEM(EA,2)

if e_lhzu then GPR(RA) ← EA

Let the EA be calculated as follows:

- For **e_lhz** and **e_lhzu**, let EA be the sum of the contents of GPR(**r**A), or 32 0s if **r**A = 0, and the sign-extended value of the D or D8 instruction field.
- For **se_lhz** let EA be the sum of the contents of GPR(**r**X) and the zero-extended value of the SD4 instruction field shifted left by 1 bit.

The half word in memory addressed by EA is loaded into bits 48–63 of GPR(**r**D). Bits 32–47 of GPR(**r**D) are cleared.

If **e_lhzu**, EA is placed into GPR(**r**A).

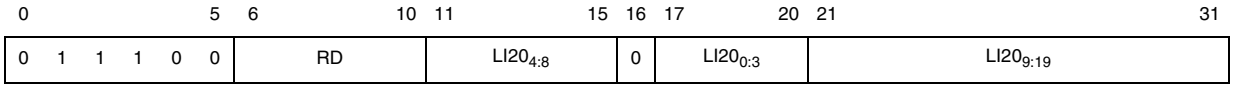
If **e_lhzu** and **r**A = 0 or **r**A = **r**D, the instruction form is invalid.

Special Registers Altered: None

_li*x* | VLE | User | **_li*x***

Load Immediate [Shifted]

e_li **r**D,LI20 (LI20-mode)

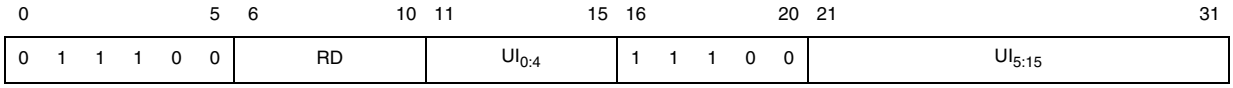


$LI20 \leftarrow LI20_{0:3} \,||\, LI20_{4:8} \,||\, LI20_{9:19}$
$GPR(RD) \leftarrow EXTS(LI20)$

For **e_li**, the sign-extended LI20 field is placed into GPR(**r**D).

Special Registers Altered: None

e_lis **r**D,UI

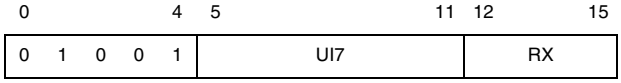


$UI \leftarrow UI_{0:4} \,||\, UI_{5:15}$
$GPR(RD) \leftarrow UI \,||\, {}^{16}0$

For **e_lis**, the UI field is concatenated on the right with 16 0's and placed into GPR(**r**D).

Special Registers Altered: None

se_li **r**X,UI7



$GPR(RX) \leftarrow {}^{25}0 \,||\, UI7$

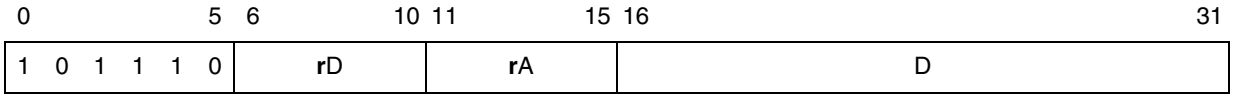
For **se_li**, the zero-extended UI7 field is placed into GPR(**r**X).

Special Registers Altered: None

lmw | Book E | User | **lmw**

Load multiple word

lmw **r**D,D(**r**A)



if rA=0 then EA ← $^{32}0$ || EXTS(D)$_{32:63}$
else EA ← $^{32}0$ || (rA+EXTS(D))$_{32:63}$
r ← rD
do while r ≤ 31
GPR(r) ← $^{32}0$ || MEM(EA,4)
r ← r + 1
EA ← $^{32}0$ || (EA+4)$_{32:63}$

The EA is bits 32–63 of the sum of the contents of **r**A, or 64 zeros if **r**A=0, and the sign-extended value of the D instruction field.

Here n=(32–**r**D). n consecutive words starting at EA are loaded into bits 32–63 of registers **r**D through GPR31. Bits 0–31 of these GPRs are cleared.

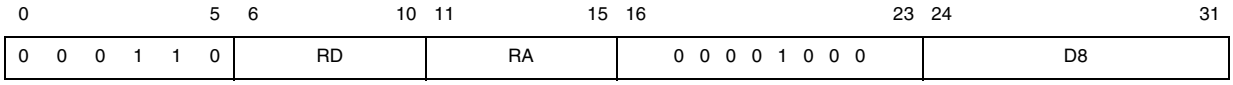
EA must be a multiple of 4. If it is not, either an alignment interrupt is invoked or the results are boundedly undefined. If **r**A is in the range of registers to be loaded, including the case in which **r**A=0, the instruction form is invalid.

Other registers altered: None

_lmw | VLE | User | **_lmw**

Load Multiple Word

e_lmw **r**D,D8(**r**A)



if RA=0 then EA ← $EXTS(D8)_{32:63}$

else EA ← $(GPR(RA)+EXTS(D8))_{32:63}$

r ← RD

do while r ≤ 31

GPR(r) ← MEM(EA,4)

r ← r + 1

EA ← $(EA+4)_{32:63}$

Let the EA be the sum of the contents of GPR(**r**A), or 32 0s if **r**A = 0, and the sign-extended value of the D8 instruction field.

Let n = (32-rD). n consecutive words starting at EA are loaded into bits 32–63 of registers GPR(**r**D) through GPR(31).

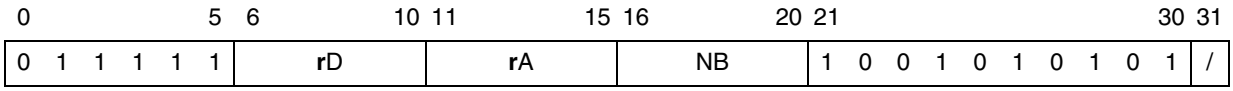
EA must be a multiple of 4. If it is not, either an alignment interrupt is invoked or the results are boundedly undefined. If **r**A is in the range of registers to be loaded, including the case in which **r**A = 0, the instruction form is invalid.

Special Registers Altered: None

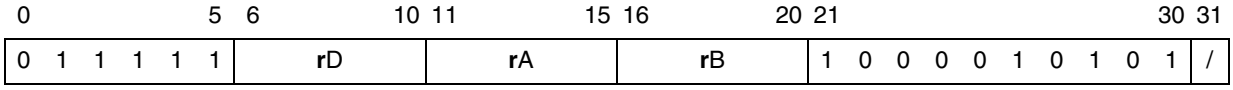
lswi | Book E | User | **lswi**

Load string word (immediate | indexed)

lswi **r**D,**r**A,NB



lswx **r**D,**r**A,**r**B



if rA=0 then a ← $^{64}0$ else a ← rA
if 'lswi' then EA ← $^{32}0$ || $a_{32:63}$
if 'lswx' then EA ← $^{32}0$ || $(a + rB)_{32:63}$
if 'lswi' & NB=0 then n ← 32
if 'lswi' & NB≠0 then n ← NB
if 'lswx' then n ← $XER_{57:63}$
r ← rD : 1
i ← 32
rD ← 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 ← $^{32}0$ || $(EA+1)_{32:63}$
    n ← n : 1

The EA is calculated as follows:

- For **lswi**, EA is 32 zeros concatenated with **r**A[32–63], or 32 zeros if **r**A=0.
- For **lwsx**, EA is 32 zeros concatenated with bits 32–63 of the sum of the contents of **r**A, or 64 zeros if **r**A=0, and the contents of **r**B.

If **lswi**, n = NB if NB ≠ 0, n = 32 if NB=0. If **lswx**, n=XER[57–63]. n is the number of bytes to load. Here 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 registers **r**D through (**r**D+nr–1). Data is loaded into the low-order 4 bytes of each GPR; the high-order 4 bytes are cleared.

Bytes are loaded left to right in each GPR. The sequence wraps to GPR0 if required. If the 4 LSBs of GPR(**r**D+nr–1) are partially filled, the unfilled LSBs of that GPR are cleared.

If **lswx** and n=0, the contents of **r**D are undefined.

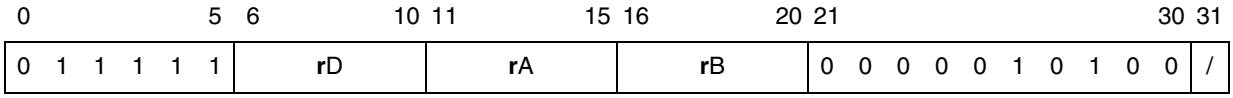
If **r**A, or **r**B for **lswx**, is in the range of registers to be loaded, including where **r**A=0, an illegal instruction type program interrupt is invoked or results are boundedly undefined. If **r**D=**r**A, or **r**D=**r**B for **lswx**, the instruction form is invalid. Other registers altered: None

Note: Programming: String instructions move data without concern for alignment. They can perform short moves between arbitrary locations or long moves between misaligned memory fields.

lwarx | Book E | User | **lwarx**

Load word and reserve indexed

lwarx **r**D,**r**A,**r**B



if rA=0 then a ← $^{64}0$ else a ← rA
EA ← $^{32}0$ || $(a + rB)_{32:63}$
RESERVE ← 1
RESERVE_ADDR ← real_addr(EA)
rD ← $^{32}0$ || MEM(EA,4)

EA is bits 32–63 of the sum of the contents of **r**A (32 zeros if **r**A=0), and the contents of **r**B.

The word addressed by EA is loaded into **r**D[32–63]; **r**D[0–31] are cleared.

lwarx creates a reservation for use by a **stwcx.** instruction. An address computed from the EA is associated with the reservation and replaces any previously associated address. See *Atomic update primitives using lwarx and stwcx. on page 176*.”

If EA is not a multiple of 4, an alignment interrupt occurs or results are boundedly undefined.

Other registers altered: None

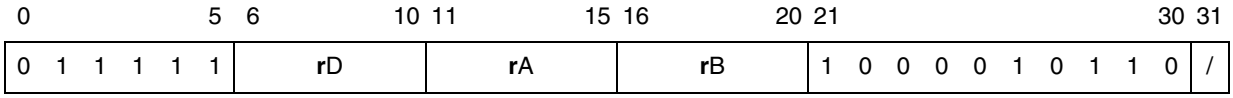
Programming notes:

- **lwarx**, and **stwcx.** permit programmers to write an instruction sequence that appears to perform an atomic update operation on a memory location. This operation depends on a single reservation resource in each processor. At most one reservation exists on any given processor.
- Because **lwarx** instructions have implementation dependencies (such as the granularity at which reservations are managed), they must be used with care. System library programs should use these instructions to implement high-level synchronization functions (such as test and set, compare and swap) needed by application programs. Application programs should use these library programs, rather than use **lwarx** directly The granularity with which reservations are managed is implementation-dependent. Therefore the location to be accessed by **lwarx** should be allocated by a system library program. See *Atomic update primitives using lwarx and stwcx. on page 176*.”

lwbrx | Book E | User | **lwbrx**

Load word byte-reverse indexed

lwbrx **r**D,**r**A,**r**B



if rA=0 then a ← $^{64}0$ else a ← rA
EA ← $^{32}0$ || $(a + rB)_{32:63}$
$data_{0:31}$ ← MEM(EA,4)
rD ← $^{32}0$ || $data_{24:31}$ || $data_{16:23}$ || $data_{8:15}$ || $data_{0:7}$

The EA is calculated as follows:

- For **lwbrx**, EA is bits 32–63 of the sum of the contents of **r**A, or 64 zeros if **r**A=0, and the contents of **r**B.

Bits 0–7 of the word addressed by EA are loaded into **r**D[56–63]. Bits 8–15 of the word addressed by EA are loaded into **r**D[48–55]. Bits 16–23 of the word addressed by EA are loaded into **r**D[40–47]. Bits 24–31 of the word addressed by EA are loaded into **r**D[32–39]. Bits **r**D[0–31] are cleared.

Other registers altered: None

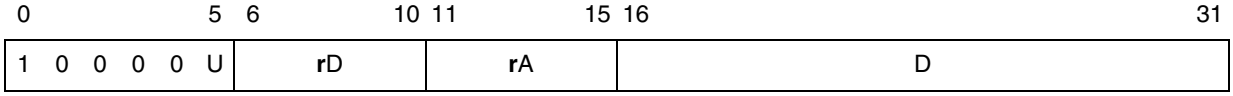
Programming notes:

- When EA references big-endian memory, these instructions have the effect of loading data in little-endian byte order. Likewise, when EA references little-endian memory, these instructions have the effect of loading data in big-endian byte order.
- In some implementations, the load word byte-reverse instructions may have greater latency than other load instructions.

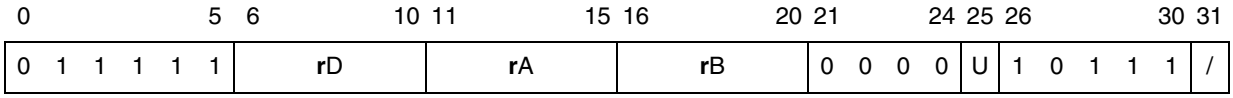
lwz | Book E | User | **lwz**

Load word and zero [with update] [indexed]

lwz **r**D,D(**r**A) (D-mode, U=0)
lwzu **r**D,D(**r**A) (D-mode, U=1)



lwzx **r**D,**r**A,**r**B (X-mode, U=0)
lwzux **r**D,**r**A,**r**B (X-mode, U=1)



if rA=0 then a ← $^{64}0$ else a ← rA
if D-mode then EA ← $^{32}0$ || (a + EXTS(D))$_{32:63}$
if X-mode then EA ← $^{32}0$ || (a + rB)$_{32:63}$
rD ← $^{32}0$ || MEM(EA,4)
if U=1 then rA ← EA

The EA is calculated as follows:

- For **lwz** and **lwzu**, EA is bits 32–63 of the sum of the contents of **r**A, or 64 zeros if **r**A=0, and the sign-extended value of the D field.
- For **lwzx** and **lwzux**, EA is bits 32–63 of the sum of the contents of **r**A, or 64 zeros if **r**A=0, and the contents of **r**B.

The word addressed by the EA is loaded into **r**D[32–63]; **r**D[0–31] are cleared.

If U=1 (with update), EA is placed into **r**A.

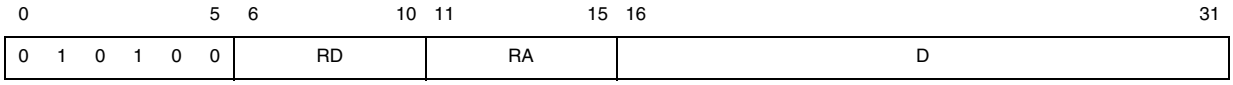
If U=1 (with update), and **r**A=0 or **r**A=**r**D, the instruction form is invalid.

Other registers altered: None

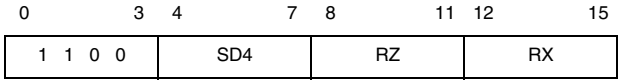
_lwz | VLE | User | **_lwz**

Load Word and Zero [with Update] [Indexed]

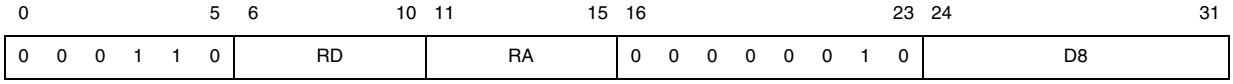
e_lwz **r**D,D(**r**A) (D-mode)



se_lwz **r**Z,SD4(**r**X) (SD4-mode)



e_lwzu **r**D,D8(**r**A) (D8-mode)



if (RA=0 & !se_lwz) then a ← $^{32}0$ else a ← GPR(RA or RX)

if D-mode then EA ← $(a + EXTS(D))_{32:63}$

if D8-mode then EA ← $(a + EXTS(D8))_{32:63}$

if SD4-mode then EA ← $(a + (^{26}0 \,||\, SD4 \,||\, ^{2}0))_{32:63}$

GPR(RD or RZ) ← MEM(EA,4)

if e_lwzu then GPR(RA) ← EA

Let the EA be calculated as follows:

- For **e_lwz** and **e_lwzu**, let EA be the sum of the contents of GPR(**r**A), or 32 0s if **r**A = 0, and the sign-extended value of the D or D8 instruction field.
- For **se_lwz** let EA be the sum of the contents of GPR(**r**X) and the zero-extended value of the SD4 instruction field shifted left by 2 bits.

The word in memory addressed by the EA is loaded into bits 32–63 of GPR(**r**D).

If **e_lwzu**, EA is placed into GPR(**r**A).

If **e_lwzu** and **r**A = 0 or **r**A = **r**D, the instruction form is invalid.

Special Registers Altered: None

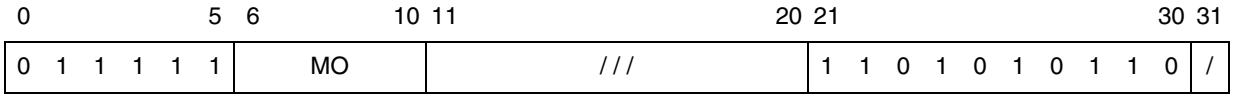
mbar

Book E	User

mbar

Memory barrier

mbar MO



When MO=0, **mbar** provides a memory ordering function for all memory access instructions executed by the processor executing the **mbar** instruction. Executing an **mbar** instruction ensures that all data memory accesses caused by instructions preceding the **mbar** have completed before any data memory accesses caused by any instructions after the **mbar**. This order is seen by all mechanisms.

When **mbar** (MO = 1), as defined by the EIS, **mbar** functions like **eieio** as it is defined by the Classic PowerPC architecture. It provides ordering for the effects of load and store instructions. These instructions consist of two sets, which are ordered separately. Memory accesses caused by a dcbz or a dcba are ordered like a store. The two sets follow:

- Caching-inhibited, guarded loads and stores to memory and write-through-required stores to memory. **mbar** (MO=1) controls the order in which accesses are performed in main memory. It ensures that all applicable memory accesses caused by instructions preceding the **mbar** have completed with respect to main memory before any applicable memory accesses caused by instructions following **mbar** access main memory. It acts like a barrier that flows through the memory queues and to main memory, preventing the reordering of memory accesses across the barrier. No ordering is performed for **dcbz** if the instruction causes the system alignment error handler to be invoked.
  All accesses in this set are ordered as one set; there is not one order for guarded, caching-inhibited loads and stores and another for write-through-required stores.
- Stores to memory that are caching-allowed, write-through not required, and memory-coherency required. **mbar** (MO=1) controls the order in which accesses are performed with respect to coherent memory. It ensures that, with respect to coherent memory, applicable stores caused by instructions before the **mbar** complete before any applicable stores caused by instructions after it.

Except for **dcbz** and **dcba**, **mbar** (MO=1) does not affect the order of cache operations (whether caused explicitly by a cache management instruction or implicitly by the cache coherency mechanism). Also. **mbar** does not affect the order of accesses in one set with respect to accesses in the other.

mbar (MO=1) may complete before memory accesses caused by instructions preceding it have been performed with respect to main memory or coherent memory as appropriate. **mbar** (MO=1) is intended for use in managing shared data structures, in accessing memory-mapped I/O, and in preventing load/store combining operations in main memory. For the first use, the shared data structure and the lock that protects it must be altered only by stores that are in the same set (for both cases described above). For the second use, **mbar** (MO=1) can be thought of as placing a barrier into the stream of memory accesses issued by a core, such that any given memory access appears to be on the same side of the barrier to both the core and the I/O device.

Because the core performs store operations in order to memory that is designated as both caching-inhibited and guarded, **mbar** (MO=1) is needed for such memory only when loads must be ordered with respect to stores or with respect to other loads.

Note that **mbar** (MO=1) does not connect hardware considerations to it such as multiprocessor implementations that send an **mbar** (MO=1) address-only broadcast (useful in some designs). For example, if a design has an external buffer that re-orders loads and stores for better bus efficiency, **mbar** (MO=1) broadcasts signals to that buffer that previous loads/stores (marked caching-inhibited, guarded, or write-through required) must complete before any following loads/stores (marked caching-inhibited, guarded, or write-through required).

If MO is not 0 or 1, an implementation may support the **mbar** instruction ordering a particular subset of memory accesses. An implementation may also support multiple, non-zero values of MO that each specify a different subset of memory accesses that are ordered by the **mbar** instruction. Which subsets of memory accesses are ordered and which values of MO specify these subsets is implementation-dependent. See the user's manual for the implementation.

On some implementations, HID1[ABE] must be set to allow management of external L2 caches (for implementations with L2 caches) as well as other L1 caches in the system.

Other registers altered: None

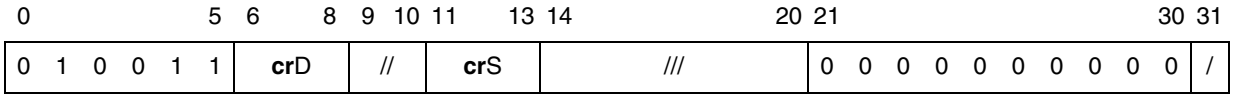
Programming note: **mbar** is provided to implement a pipelined memory barrier. The following sequence shows one use of **mbar** in supporting shared data, ensuring the action is completed before releasing the lock.

P1	**P2**
lock	. . .
read & write	. . .
mbar	. . .
free lock	. . .
. . .	lock
. . .	read & write
. . .	mbar
. . .	free lock

mcrf | Book E | User | **mcrf**

Move condition register field

mcrf **cr**D,**cr**S



$CR_{4xBF+32:4xBF+35} \leftarrow CR_{4x\mathbf{cr}S+32:4x\mathbf{cr}S+35}$

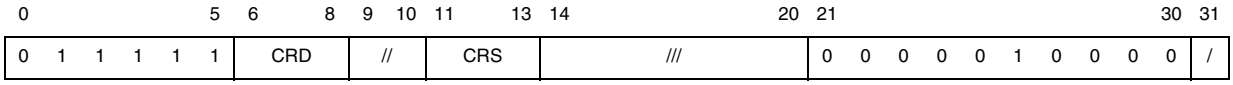
The contents of field **cr**S (bits 4×**cr**S+32–4×**cr**S+35) of CR are copied to field **cr**D (bits 4×**cr**D+32–4×**cr**D+35) of CR.

Other registers altered: CR

_mcrf | VLE | User | **_mcrf**

Move CR Field

e_mcrf **cr**D,**cr**S



$CR_{4xCRD+32:4xCRD+35} \leftarrow CR_{4xCRS+32:4xCRS+35}$

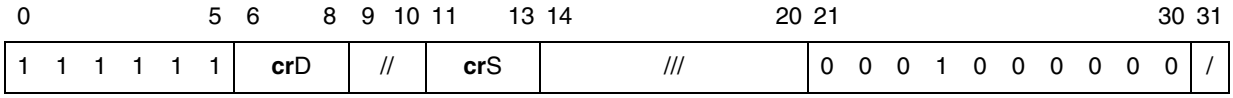
The contents of field **cr**S (bits 4×CRS+32 through 4×CRS+35) of the CR are copied to field **cr**D (bits 4×CRD+32 through 4×CRD+35) of the CR.

Special Registers Altered: CR

mcrfs | Book E | User | **mcrfs**

Move to condition register from FPSCR

mcrfs **cr**D,**cr**S



$CR_{BF \times 4:crD \times 4+3} \leftarrow FPSCR_{crS \times 4:crS \times 4+3}$
$FPSCR_{crS \times 4:crS \times 4+3} \leftarrow 0b0000$

The contents of FPSCR[**cr**S] are copied to CR field **cr**D. All exception bits copied are cleared in the FPSCR. If the FX bit is copied, it is cleared in the FPSCR.

If MSR[FP]=0, an attempt to execute **mcrfs** causes a floating-point unavailable interrupt.

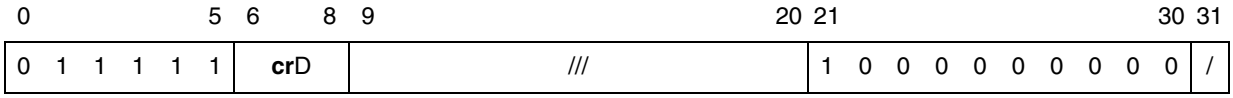
Other registers altered:

- CR field **cr**D
  FX OX(if **cr**S=0)
  UX ZX XX VXSNAN(if **cr**S=1)
  VXISI VXIDI VXZDZ VXIMZ(if **cr**S=2)
  VXVC(if **cr**S=3)
  VXSOFT VXSQRT VXCVI(if **cr**S=5)

mcrxr | Book E | User | **mcrxr**

Move to condition register from integer exception register

mcrxr **cr**D



$CR_{4\times crD+32:4\times crD+35} \leftarrow XER_{32:35}$
$XER_{32:35} \leftarrow 0b0000$

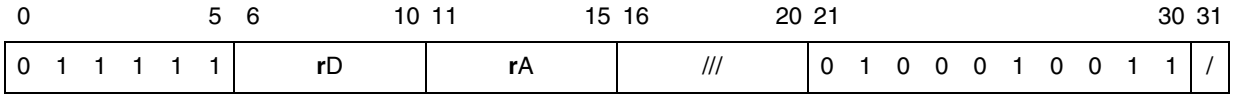
The contents of XER[32–35] are copied to CR field **cr**D. XER[32–35] are cleared.

Other registers altered: CR XER[32–35]

mfapidi | Book E | User | **mfapidi**

Move from APID Indirect

mfapidi **r**D,**r**A



rD ¨ implementation-dependent value based on rA

The contents of **r**A are provided to any auxiliary processing extensions that may be present. A value, that is implementation-dependent and extension-dependent, is placed in **r**D.

Other 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 processing extensions. See user's manual for the implementation for details on the behavior of this instruction.

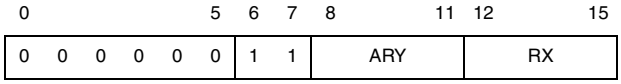
_mfar

VLE	User

_mfar

Move from Alternate Register

se_mfar **r**X,**ar**Y



GPR(RX) ← GPR(ARY)

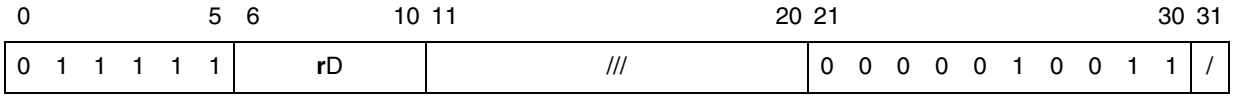
For **se_mfar**, the contents of GPR(**ar**Y) are placed into GPR(**r**X). **ar**Y specifies a GPR in the range R8–R23. The encoding 0000 specifies R8, 0001 specifies R9,…, 1111 specifies R23.

Special Registers Altered: None

mfcr | Book E | User | **mfcr**

Move from condition register

mfcr **r**D



rD ← ^{32}0 || CR

The contents of the CR are placed into **r**D[32–63]. Bits **r**D[0–31] are cleared.

Other registers altered: None

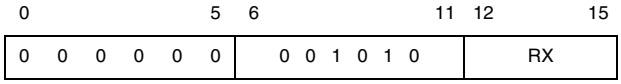
_mfctr

VLE	User

_mfctr

Move From Count Register

se_mfctr **rX**



GPR(RX) ← CTR

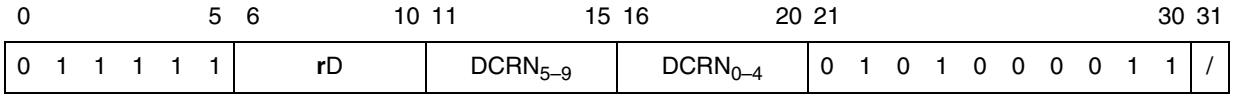
The CTR contents are placed into bits 32–63 of GPR(**r**X).

Special Registers Altered: None

mfdcr | Book E | User | **mfdcr**

Move from device control register

mfdcr **r**D,DCRN



rD ← DCREG(DCRN)

DCRN identifies the DCR (see the user's manual for a list of DCRs supported by the implementation).

The contents of the designated DCR are placed into **r**D. For 32-bit DCRs, the contents of the DCR are placed into **r**D[32–63]. Bits **r**D[0–31] are cleared.

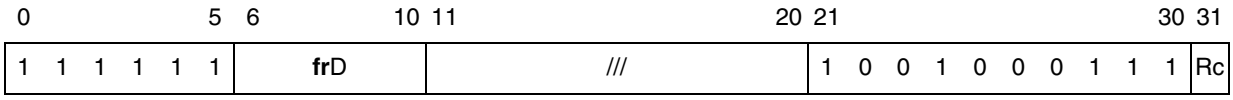
Execution of this instruction is restricted to supervisor mode.

Other registers altered: None

mffs | Book E | User | **mffs**

Move from FPSCR

mffs **fr**D (Rc=0)
mffs. **fr**D (Rc=1)



frD ← FPSCR

The contents of the FPSCR are placed into **fr**D[32–63]; **fr**D[0–31] are undefined.

If MSR[FP]=0, an attempt to execute **mffs[.]** causes a floating-point unavailable interrupt.

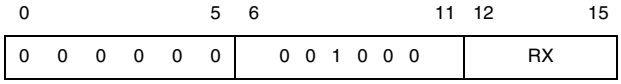
Other registers altered:

- CR1 ← FX || FEX || VX || OX (if Rc=1)

_mflr | VLE | User | **_mflr**

Move From Link Register

se_mflr **r**X



GPR(RX) ← LR

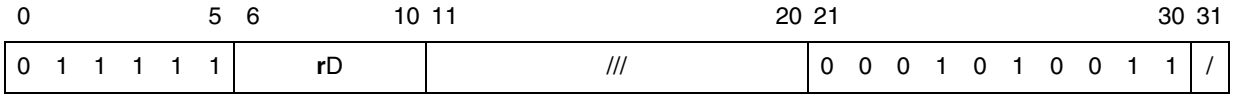
The LR contents are placed into bits 32–63 of GPR(**r**X).

Special Registers Altered: None

mfmsr | Book E | User | **mfmsr**

Move from machine state register

mfmsr **r**D



rD ← ^{32}0 || MSR

The contents of the MSR are placed into **r**D[32–63]. Bits **r**D[0–31] are cleared.

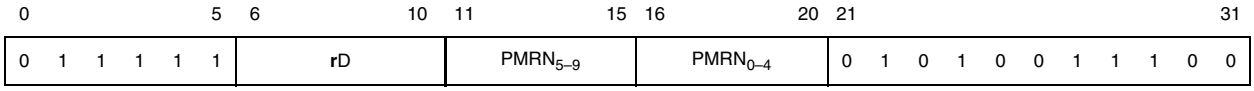
Execution of this instruction is restricted to supervisor mode.

Other registers altered: None

mfpmr | Book E | User | **mfpmr**

Move from Performance Monitor Register

mfpmr **r**D,PMRN



GPR(**r**D) ← PMREG(PMRN)

PMRN denotes a performance monitor register. *Section 2.16: Performance monitor registers (PMRs)*," lists supported performance monitor registers.

The contents of the designated performance monitor register are placed into GPR[**r**D].

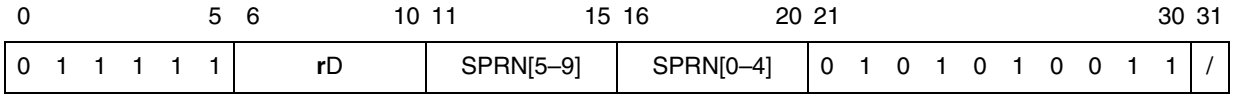
When MSR[PR] = 1, specifying a performance monitor register that is not implemented and is not privileged (PMRN[5] = 0) results in an illegal instruction exception-type program interrupt. When MSR[PR] = 1, specifying a performance monitor register that is privileged (PMRN[5] = 1) results in a privileged instruction exception-type program interrupt. When MSR[PR] = 0, specifying an unimplemented performance monitor register is boundedly undefined.

Other registers altered: None

mfspr | Book E | User | **mfspr**

Move from special purpose register

mfspr **r**D,SPRN



rD ← SPREG(SPRN)

SPRN denotes an SPR (see *Chapter 2.18: Book E SPR model on page 130*").

The contents of the designated SPR are placed into **r**D. For 32-bit SPRs, the contents of the SPR are placed into **r**D[32–63]. Bits **r**D[0–31] are cleared.

Table 207. Effect of SPRN[5] and MSR[PR]

SPRN[5]	MSR[PR]	SPRN class	Result
0	1	Defined	If not implemented, illegal instruction exception If implemented, as defined in Book E
0	1	Allocated	If not implemented, illegal instruction exception If implemented, as defined in user's manual
0	1	Preserved	If not implemented, illegal instruction exception If implemented, as defined in PowerPC Architecture
0	1	Reserved	Illegal instruction exception
1	1	—	Privileged exception
—	0	Defined	If not implemented, boundedly undefined If implemented, as defined in Book E
—	0	Allocated	If not implemented, boundedly undefined If implemented, as defined in user's manual
—	0	Preserved	If not implemented, boundedly undefined If implemented, as defined in PowerPC Architecture
—	0	Reserved	Boundedly undefined

Execution of this instruction specifying a defined and privileged SPR (SPRN[5]=1) when MSR[PR]=1 results in a privileged instruction exception-type program interrupt.

Other registers altered: None

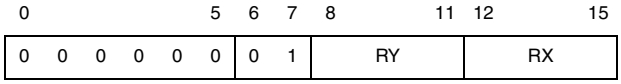
_mr

VLE	User

_mr

Move Register

se_mr **rX,rY**



GPR(RX) ← GPR(RY)

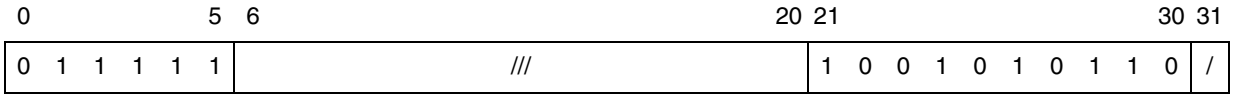
For **se_mr**, the contents of GPR(**r**Y) are placed into GPR(**r**X).

Special Registers Altered: None

msync | Book E | User | **msync**

Memory synchronize

msync



The **msync** instruction provides an ordering function for the effects of all instructions executed by the processor executing the **msync**. Executing **msync** ensures that all instructions preceding the **msync** have completed before **msync** completes and that no subsequent instructions are initiated until after the **msync** completes. It also creates a memory barrier (see *Atomic update primitives using lwarx and stwcx. on page 176*"), which orders the memory accesses associated with these instructions.

The **msync** may not complete before memory accesses associated with instructions preceding **msync** have been performed.

On some implementations, HID1[ABE] must be set to allow management of external L2 caches (for implementations with L2 caches) as well as other L1 caches in the system.

msync is execution synchronizing. (See *Execution synchronization on page 145*.")

Other registers altered: None

Programming notes:

- **msync** can be used to ensure that all stores into a data structure, caused by store instructions executed in a critical section of a program, are performed with respect to another processor before the store that releases the lock is performed with respect to that processor.
  The functions performed by the **msync** may take a significant amount of time to complete, so indiscriminate use of this instruction may adversely affect performance. The Memory Barrier (**mbar**) instruction may be more appropriate than **msync** for many cases.
- **msync** replaces the **sync** instruction; it uses the same opcode as **sync** such that PowerPC applications calling for **sync** invoke the **msync** when executed on an Book E implementation. The functionality of **msync** is identical to **sync** except that **msync** also does not complete until all previous memory accesses complete. **mbar** is provided in the Book E for those occasions when only ordering of memory accesses is required without execution synchronization.

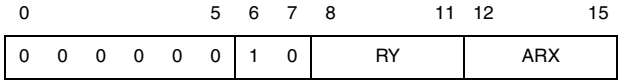
_mtar

VLE	User

_mtar

Move to Alternate Register

se_mtar **ar**X,**r**Y



GPR(ARX) ← GPR(RY)

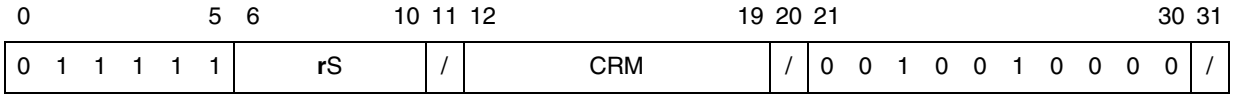
For **se_mtar**, the contents of GPR(**r**Y) are placed into GPR(**ar**X). **ar**X specifies a GPR in the range R8–R23. The encoding 0000 specifies R8, 0001 specifies R9,…, 1111 specifies R23.

Special Registers Altered: None

mtcrf | Book E | User | **mtcrf**

Move to condition register fields

mtcrf CRM,**r**S



i ← 0
do while i < 8
if CRM_i=1 then $CR_{4\times i+32:4\times i+35} \leftarrow rS_{4\times i+32:4\times i+35}$
i ← i+1

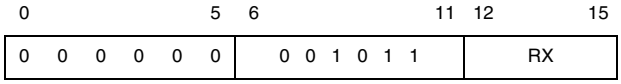
The contents of **r**S[32–63] are placed into the CR under control of the field mask specified by CRM. The field mask identifies the 4-bit fields affected. Let i be an integer in the range 0–7. If CRM_i = 1, CR field i (CR bits 4×i+32 through 4×i+35) is set to the contents of the corresponding field of **r**S[32–63].

Other registers altered: CR fields selected by mask

_mtctr | VLE | User | **_mtctr**

Move To Count Register

se_mtctr **rX**



CTR ← GPR(RX)

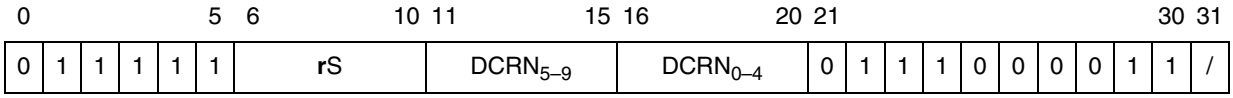
The contents of bits 32–63 of GPR(**rX**) are placed into the CTR.

Special Registers Altered: CTR

mtdcr | Book E | User | **mtdcr**

Move to device control register

mtdcr DCRN,**r**S



DCREG(DCRN) ← rS

DCRN identifies the DCR (see user's manual for a list of DCRs supported by the implementation).

The contents of **r**S are placed into the designated DCR. For 32-bit DCRs, **r**S[32–63] are placed into the DCR.

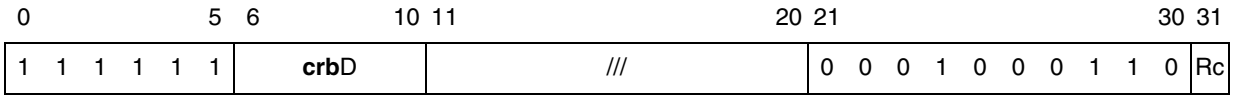
Execution of this instruction is restricted to supervisor mode.

Other registers altered: See the user's manual for the implementation

mtfsb0 | Book E | User | **mtfsb0**

Move to FPSCR Bit 0

mtfsb0 **crb**D (Rc=0)
mtfsb0. **crb**D (Rc=1)



FPSCR[BT]← 0b0

FPSCR[BT] is cleared.

If MSR[FP]=0, an attempt to execute **mtfsb0**[**.**] causes a floating-point unavailable interrupt.

Other registers altered:

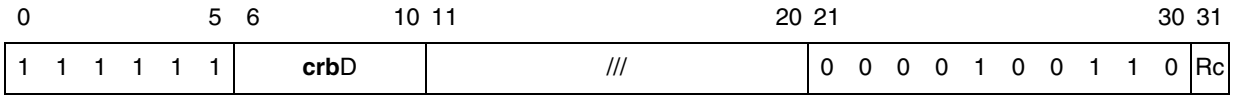
- FPSCR[BT]
  CR1 ← FX || FEX || VX || OX (if Rc=1)

Programming note: Bits 1 and 2 (FEX and VX) cannot be explicitly reset.

mtfsb1 | Book E | User | **mtfsb1**

Move to FPSCR Bit 1

mtfsb1 **crb**D (Rc=0)
mtfsb1. **crb**D (Rc=1)



FPSCR[BT] ← 0b1

FPSCR[BT] is set.

If MSR[FP]=0, an attempt to execute **mtfsb1**[**.**] causes a floating-point unavailable interrupt.

Other registers altered:

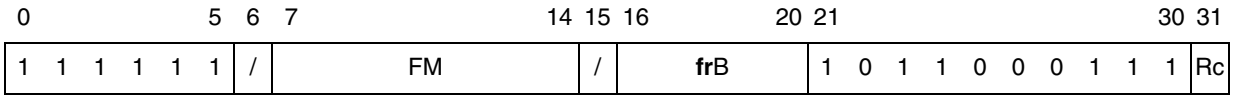
- FPSCIR[BT,FX]
  CR1 ← FX || FEX || VX || OX (if Rc=1)

Programming note: Bits 1 and 2 (FEX and VX) cannot be explicitly set.

mtfsf | Book E | User | **mtfsf**

Move to FPSCR fields

mtfsf FM,**fr**B (Rc=0)
mtfsf. FM,**fr**B (Rc=1)



i ← 0
do while i<8
if FM_i=1 then $FPSCR_{4\times i:4\times i+3} \leftarrow frB_{4\times i:4\times i+3}$
i ← i+1

The contents of **fr**B[32–63] are placed into the FPSCR under control of the field mask specified by FM. The field mask identifies the 4-bit fields affected. Let i be an integer in the range 0–7. If FM_i=1, FPSCR field i (FPSCR bits 4×i through 4×i+3) is set to the contents of the corresponding field of the low-order 32 bits of **fr**B.

FPSCR[FX] is altered only if FM_0 = 1.

If MSR[FP]=0, an attempt to execute **mtfsf**[**.**] causes a floating-point unavailable interrupt.

Other registers altered:

- FPSCR fields selected by mask
  CR1 ← FX || FEX || VX || OX (if Rc=1)

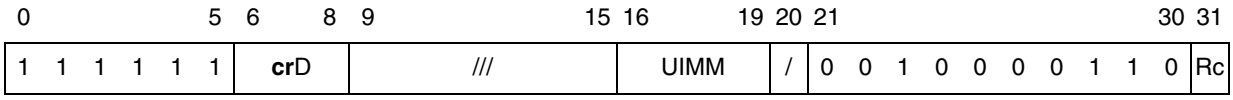
Programming notes:

- Updating fewer than all eight fields of the FPSCR may have substantially poorer performance on some implementations than updating all the fields.
- When FPSCR[0–3] is specified, bits 0 (FX) and 3 (OX) are set to the values of $(\mathbf{fr}B)_{32}$ and $(\mathbf{fr}B)_{35}$ (that is, even if this instruction causes OX to change from 0 to 1, FX is set from $(\mathbf{fr}B)_{32}$ and not by the usual rule that FX is set when an exception bit changes from 0 to 1). Bits 1 and 2 (FEX and VX) are set according to the usual rule (see *Table 10: FPSCR field descriptions on page 59*) and not from $(\mathbf{fr}B)_{33–34}$.

mtfsfi | Book E | User | **mtfsfi**

Move to FPSCR field immediate

mtfsfi **cr**D,UIMM (Rc=0)
mtfsfi. **cr**D,UIMM (Rc=1)



$FPSCR_{BF^{\times}4:crD^{\times}4+3} \leftarrow UIMM$

The value of the UIMM field is placed into FPSCR[**cr**D].

FPSCR[FX] is altered only if **cr**D = 0.

If MSR[FP]=0, an attempt to execute **mtfsfi**[**.**] causes a floating-point unavailable interrupt.

Other registers altered:

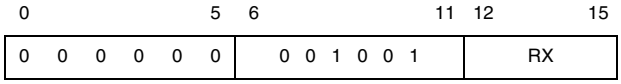
- FPSCR[**cr**D]
  CR1 ← FX || FEX || VX || OX (if Rc=1)

Programming note: When FPSCR[0–3] is specified, bits 0 (FX) and 3 (OX) are set to the values of U0 and U3 (that is, even if this instruction causes OX to change from 0 to 1, FX is set from U0 and not by the usual rule that FX is set when an exception bit changes from 0 to 1). Bits 1 and 2 (FEX and VX) are set according to the usual rule (see *Table 10: FPSCR field descriptions on page 59*), and not from U1–2.

_mtlr | VLE | User | **_mtlr**

Move To Link Register

se_mtlr **rX**



LR ← GPR(RX)

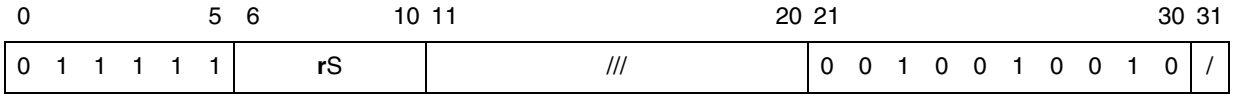
The contents of bits 32–63 of GPR(**rX**) are placed into the LR.

Special Registers Altered: LR

mtmsr | Book E | Supervisor | **mtmsr**

Move to machine state register

mtmsr **r**S



MSR $\leftarrow$ $rS_{32:63}$

The contents of **r**S[32–63] are placed into the MSR.

Execution of this instruction is restricted to supervisor mode.

Execution of this instruction is execution synchronizing. See *Execution synchronization on page 145*."

In addition, changes 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] causes 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] causes the critical input interrupt to be taken before the next instruction is executed if no higher priority exception exists.

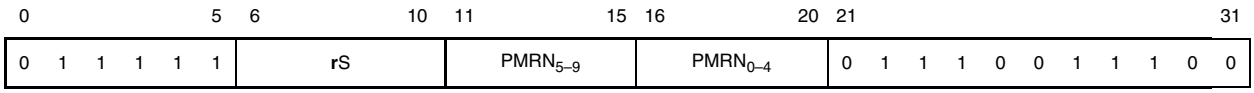
Other registers altered: MSR

Programming note: For a discussion of software synchronization requirements when altering certain MSR bits, refer to *Chapter 2.18.2: Synchronization requirements for SPRs on page 130*."

mtpmr | Performance Monitor APU | User/Supervisor | **mtpmr**

Move To Performance Monitor Register

mtpmr PMRN,**r**S



PMREG(PMRN) ← GPR(RS)

PMRN denotes a performance monitor register. *Section 2.16: Performance monitor registers (PMRs)*," lists supported performance monitor registers).

The contents of GPR[**r**S] are placed into the designated performance monitor register.

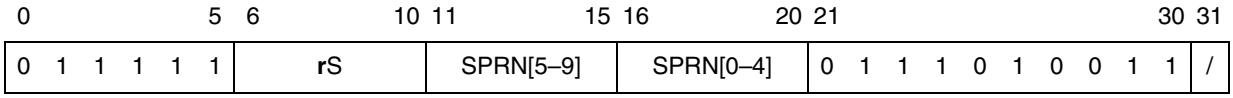
When MSR[PR] = 1, specifying a performance monitor register that is not implemented and is not privileged (PMRN[5] = 0) results in an illegal instruction exception-type program interrupt. When MSR[PR] = 1, specifying a performance monitor register that is privileged (PMRN[5] = 1) results in a privileged instruction exception-type program interrupt. When MSR[PR] = 0, specifying a unimplemented performance monitor register is boundedly undefined.

Other registers altered: None

mtspr | Book E | User/Supervisor | **mtspr**

Move to special purpose register

mtspr SPRN,**r**S



SPREG(SPRN) ← rS

SPRN denotes an SPR (see *Chapter 2.18: Book E SPR model on page 130*," and the user's manual of the implementation for a list of all SPRs that are implemented).

The contents of **r**S are placed into the designated SPR. For 32-bit SPRs, the contents of **r**S[32–63] are placed into the SPR.

When MSR[PR]=1, specifying an SPR that is not implemented and is not privileged (SPRN[5]=0) results in an illegal instruction exception-type program interrupt. When MSR[PR]=1, specifying an SPR that is privileged (SPRN[5]=1) results in a privileged instruction exception-type program interrupt. When MSR[PR]=0, specifying an SPR that is not implemented is boundedly undefined.

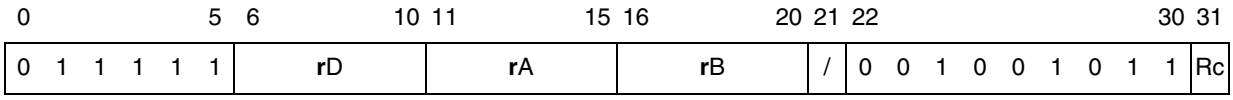
Other registers altered: See *Chapter 2.18: Book E SPR model on page 130*," or the user's manual for the implementation.

Programming note: For a discussion of software synchronization requirements when altering certain SPRs, please refer to *Chapter 2.18.2: Synchronization requirements for SPRs on page 130*."

mulhw | Book E | User | **mulhw**

Multiply high word

mulhw **r**D,**r**A,**r**B (Rc=0)
mulhw. **r**D,**r**A,**r**B (Rc=1)



$prod_{0:63} \leftarrow rA_{32:63} \times rB_{32:63}$
if Rc=1 then do
$LT \leftarrow prod_{0:31} < 0$
$GT \leftarrow prod_{0:31} > 0$
$EQ \leftarrow prod_{0:31} = 0$
$CR0 \leftarrow LT \,||\, GT \,||\, EQ \,||\, SO$
$rD_{32:63} \leftarrow prod_{0:31}$
$rD_{0:31} \leftarrow$ undefined

Bits 0–31 of the 64-bit product of the contents of **r**A[32–63] and the contents of **r**B[32–63] are placed into **r**D[32–63]. Bits **r**D[0–31] are undefined.

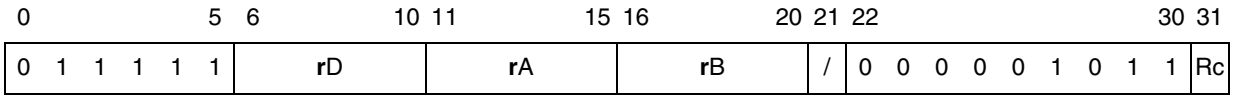
Both operands and the product are interpreted as signed integers.

Other registers altered: CR0 (if Rc=1)

mulhwu | Book E | User | **mulhwu**

Multiply high word unsigned

mulhwu **r**D,**r**A,**r**B (Rc=0)
mulhwu. **r**D,**r**A,**r**B (Rc=1)



$prod_{0:63} \leftarrow rA_{32:63} \times rB_{32:63}$
if Rc=1 then do
$LT \leftarrow prod_{0:31} < 0$
$GT \leftarrow prod_{0:31} > 0$
$EQ \leftarrow prod_{0:31} = 0$
$CR0 \leftarrow LT \,||\, GT \,||\, EQ \,||\, SO$
$rD_{32:63} \leftarrow prod_{0:31}$
$rD_{0:31} \leftarrow$ undefined

Bits 0–31 of the 64-bit product the contents of **r**A[32–63] and the contents of **r**B[32–63] are placed into **r**D[32–63]. Bits **r**D[0–31] are undefined.

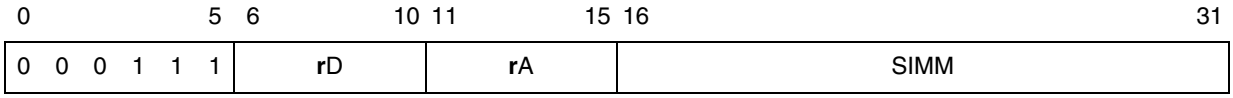
Both operands and the product are interpreted as unsigned integers, except that if Rc=1 the first three bits of CR field 0 are set by signed comparison of the result to zero.

Other registers altered: CR0 (if Rc=1)

mulli | Book E | User | **mulli**

Multiply low immediate

mulli **r**D,**r**A,SIMM



$prod_{0:127} \leftarrow rA \times EXTS(SIMM)$
$rD \leftarrow prod_{64:127}$

Bits 64–127 of the 128-bit product of the contents of **r**A and the sign-extended value of the SIMM field are placed into **r**D.

Both operands and the product are interpreted as signed integers.

Other registers altered: None

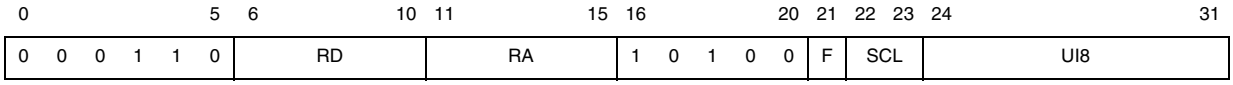
Programming notes:

- For **mulli**, 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**, bits 32–63 of the product are independent of whether the operands are regarded as signed or unsigned 32-bit integers.

_mulli*x* | VLE | User | **_mulli*x***

Multiply Low [2 operand] Immediate

e_mulli **r**D,**r**A,SCI8



imm ← SCI8(F,SCL,UI8)
$prod_{0:63}$ ← GPR(RA) × imm

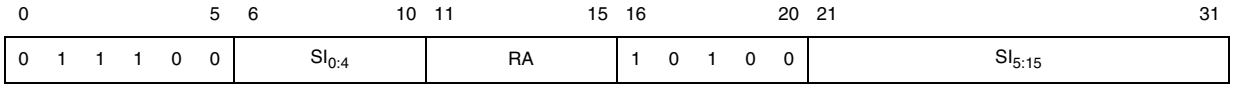
GPR(RD) ← $prod_{32:63}$

Bits 32–63 of the 64-bit product of the contents of GPR(**r**A) and the value of SCI8 are placed into GPR(**r**D).

Both operands and the product are interpreted as signed integers.

Special Registers Altered: None

e_mull2i **r**A,SI



$prod_{0:63}$ ← GPR(RA) × EXTS($SI_{0:4}$ || $SI_{5:15}$)

GPR(RA) ← $prod_{32:63}$

Bits 32–63 of the 64-bit product of the contents of GPR(**r**A) and the sign-extended value of the SI field are placed into GPR(**r**A).

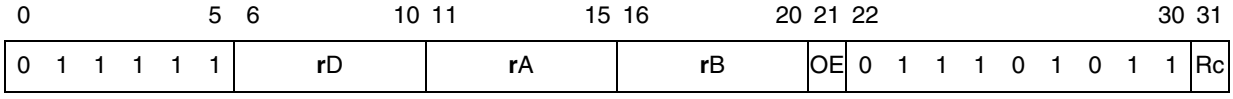
Both operands and the product are interpreted as signed integers.

Special Registers Altered: None

mullw | Book E | User | **mullw**

Multiply low word

mullw	**r**D,**r**A,**r**B	(OE=0, Rc=0)
mullw.	**r**D,**r**A,**r**B	(OE=0, Rc=1)
mullwo	**r**D,**r**A,**r**B	(OE=1, Rc=0)
mullwo.	**r**D,**r**A,**r**B	(OE=1, Rc=1)



$prod_{0:63} \leftarrow rA_{32:63} \times rB_{32:63}$
if OE=1 then do
  $OV \leftarrow (prod_{0:31} \neq {}^{32}0)$ & $(prod_{0:31} \neq {}^{32}1)$
  $SO \leftarrow SO \mid OV$
if Rc=1 then do
  $LT \leftarrow prod_{32:63} < 0$
  $GT \leftarrow prod_{32:63} > 0$
  $EQ \leftarrow prod_{32:63} = 0$
  $CR0 \leftarrow LT \mid\mid GT \mid\mid EQ \mid\mid SO$
$rD \leftarrow prod_{0:63}$

The 64-bit product of the contents of **r**A[32–63] and the contents of **r**B[32–63] is placed into **r**D.

If OE=1, OV is set if the product cannot be represented in 32 bits.

Both operands and the product are interpreted as signed integers.

Other registers altered:

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

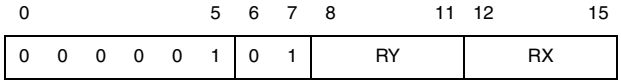
Programming notes:

- Bits 32–63 of the product are independent of whether the operands are regarded as signed or unsigned 32-bit integers.

_mullw*x* | VLE | User | **_mullw*x***

Multiply Low Word

se_mullw **rX,rY**



$prod_{0:63} \leftarrow GPR(RX)_{32:63} \times GPR(RY)_{32:63}$

$GPR(RX) \leftarrow prod_{32:63}$

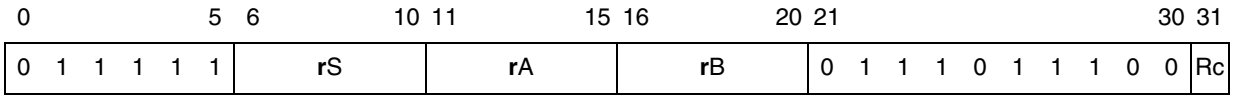
Bits 32–63 of the 64-bit product of the contents of bits 32–63 of GPR(**r**X) and the contents of bits 32–63 of GPR(**r**Y) is placed into GPR(**r**X).

Special Registers Altered: None

nand | Book E | User | **nand**

NAND

nand **r**A,**r**S,**r**B (Rc=0)
nand. **r**A,**r**S,**r**B (Rc=1)



$result_{0:63} \leftarrow \neg(rS \,\&\, rB)$
if Rc=1 then do
$LT \leftarrow result_{32:63} < 0$
$GT \leftarrow result_{32:63} > 0$
$EQ \leftarrow result_{32:63} = 0$
$CR0 \leftarrow LT \,||\, GT \,||\, EQ \,||\, SO$
$rA \leftarrow result$

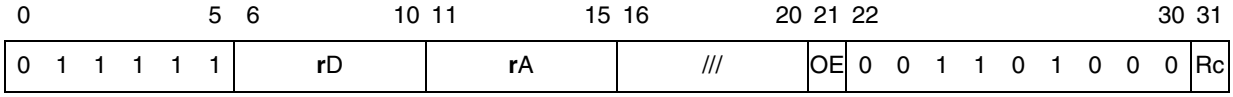
The contents of **r**S are ANDed with the contents of **r**B and the one's complement of the result is placed into **r**A.

Other registers altered: CR0 (if Rc=1)

neg | Book E | User | **neg**

Negate

neg	**r**D,**r**A	(OE=0, Rc=0)
neg.	**r**D,**r**A	(OE=0, Rc=1)
nego	**r**D,**r**A	(OE=1, Rc=0)
nego.	**r**D,**r**A	(OE=1, Rc=1)



$carry_{0:63} \leftarrow Carry(\neg rA + 1)$
$sum_{0:63} \leftarrow \neg rA + 1$
if OE=1 then do
$OV \leftarrow carry_{32} \oplus carry_{33}$
$SO \leftarrow SO \mid (carry_{32} \oplus carry_{33})$
if Rc=1 then do
$LT \leftarrow sum_{32:63} < 0$
$GT \leftarrow sum_{32:63} > 0$
$EQ \leftarrow sum_{32:63} = 0$
$CR0 \leftarrow LT \mid\mid GT \mid\mid EQ \mid\mid SO$
$rD \leftarrow sum$

The sum of the one's complement of the contents of **r**A and 1 is placed into **r**D.

If **r**A contains the most negative 64-bit number (0x8000_0000_0000_0000), the result is the most negative number. Similarly, if **r**A[32–63] contain the most negative 32-bit number (0x8000_0000), bits 32–63 of the result contain the most negative 32-bit number and, if OE=1, OV is set.

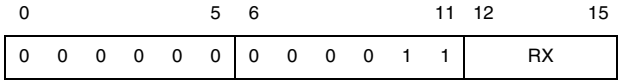
Other registers altered:

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

_neg*x* | VLE | User | **_neg*x***

Negate

se_neg **r**X



$result_{32:63} \leftarrow \neg GPR(RX) + 1$

$GPR(RX) \leftarrow result_{32:63}$

The sum of the one's complement of the contents of GPR(**r**X) and 1 is placed into GPR(**r**X).

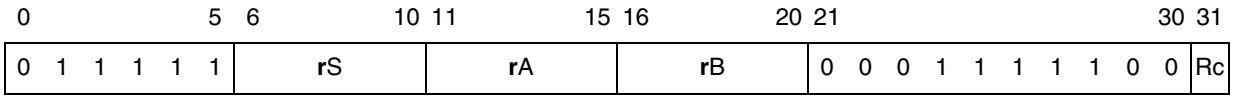
If bits 32–63 of GPR(**r**X) contain the most negative 32-bit number (0x8000_0000), bits 32–63 of the result contain the most negative 32-bit number

Special Registers Altered: None

nor | Book E | User | **nor**

NOR

nor **r**A,**r**S,**r**B (Rc=0)
nor. **r**A,**r**S,**r**B (Rc=1)



$result_{0:63} \leftarrow \neg(rS \mid rB)$
if Rc=1 then do
$LT \leftarrow result_{32:63} < 0$
$GT \leftarrow result_{32:63} > 0$
$EQ \leftarrow result_{32:63} = 0$
$CR0 \leftarrow LT \mid\mid GT \mid\mid EQ \mid\mid SO$
$rA \leftarrow result$

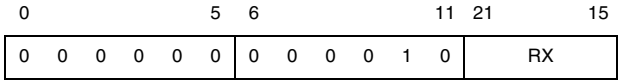
The contents of **r**S are ORed with the contents of **r**B and the one's complement of the result is placed into **r**A.

Other registers altered: CR0 (if Rc=1)

_not*x* | VLE | User | **_not*x***

NOT

se_not **r**X



$result_{32:63} \leftarrow \neg GPR(RX)$

$GPR(RX) \leftarrow result_{32:63}$

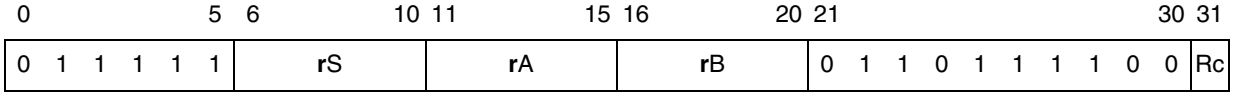
The contents of GPR(**r**X) are inverted.

Special Registers Altered: None

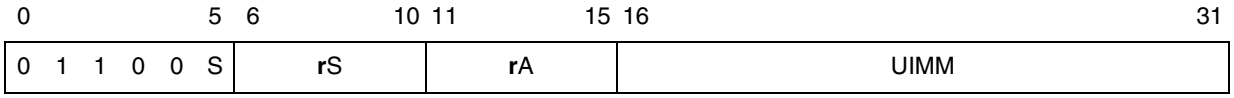
or | Book E | User | **or**

OR [Immediate [shifted] | with complement]

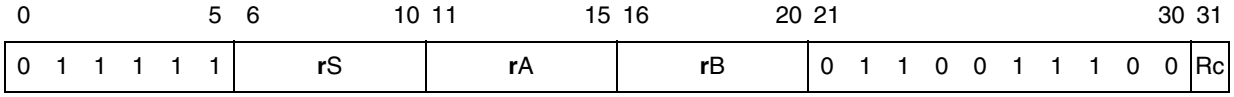
or **r**A,**r**S,**r**B (Rc=0)
or. **r**A,**r**S,**r**B (Rc=1)



ori **r**A,**r**S,UIMM (S=0, Rc=0)
oris **r**A,**r**S,UIMM (S=1, Rc=0)



orc **r**A,**r**S,**r**B (Rc=0)
orc. **r**A,**r**S,**r**B (Rc=1)



if ‘ori’ then b ← $^{48}0$ || UIMM
if ‘oris’ then b ← $^{32}0$ || UIMM || $^{16}0$
if ‘or[.]’ then b ← rB
if ‘orc[.]’ then b ← ¬rB
$result_{0:63}$ ← rS | b
if Rc=1 then do
$LT \leftarrow result_{32:63} < 0$
$GT \leftarrow result_{32:63} > 0$
$EQ \leftarrow result_{32:63} = 0$
CR0 ← LT || GT || EQ || SO
rA ← result

For **ori**, the contents of **r**S are ORed with $^{48}0$ || UIMM.

For **oris**, the contents of **r**S are ORed with $^{32}0$ || UIMM || $^{16}0$.

For **or**[**.**], the contents of **r**S are ORed with the contents of **r**B.

For **orc**[**.**], the contents of **r**S are ORed with the one’s complement of the contents of **r**B.

The result is placed into **r**A.

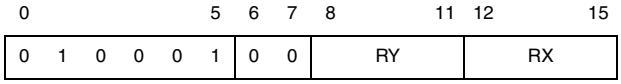
The preferred no-op is **ori 0,0,0**

Other registers altered: CR0 (if Rc=1)

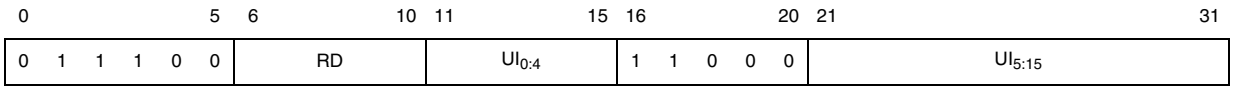
_or*x* | VLE | User | **_or*x***

OR [2 operand] [Immediate | with Complement] [Shifted][and Record]

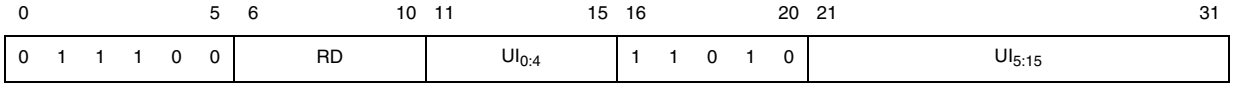
se_or **r**X,**r**Y



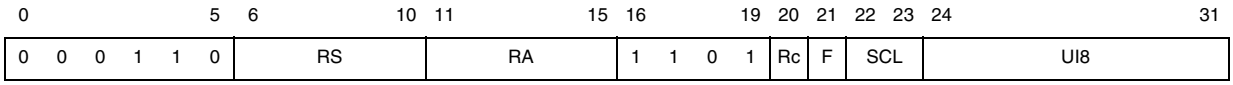
e_or2i **r**D,UI



e_or2is **r**D,UI



e_ori **r**A,**r**S,SCI8 (Rc = 0)
e_ori. **r**A,**r**S,SCI8 (Rc = 1)



if ‘e_ori[.]’ then b ← SCI8(F,SCL,UI8)

if ‘e_or2i’ then b ← $^{16}0$ || $UI_{0:4}$ || $UI_{5:15}$
if ‘e_or2is’ then b ← $UI_{0:4}$ || $UI_{5:15}$ || $^{16}0$
if ‘se_or’ then b ← GPR(RB)

$result_{0:63}$ ← GPR(RS or RD or RX) | b

if Rc=1 then do

LT ← $result_{32:63} < 0$

GT ← $result_{32:63} > 0$

EQ ← $result_{32:63} = 0$

CR0 ← LT || GT || EQ || SO

GPR(RA or RD or RX) ← result

For **e_ori[.]**, the contents of GPR(**r**S) are ORed with the value of SCI8.

For **e_or2i,** the contents of GPR(**r**D) are ORed with $^{16}0$ || UI.

For **e_or2is**, the contents of GPR(**r**D) are ORed with UI || $^{16}0$.

For **se_or**, the contents of GPR(**r**X) are ORed with the contents of GPR(**r**Y).

The result is placed into GPR(**r**A or **r**X).

The preferred ‘no-op’ (an instruction that does nothing) is:

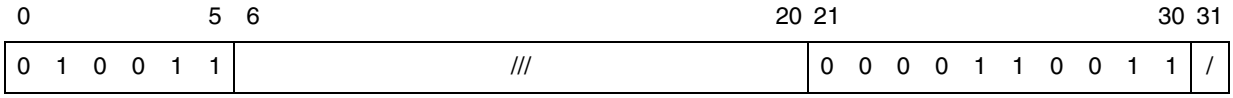
e_ori 0,0,0

Special Registers Altered: CR0 (if Rc = 1)

rfci | Book E | Supervisor | **rfci**

Return from critical interrupt

rfci



MSR ← CSRR1
NIA ← 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. 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 is the address 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 **rfci**).

Execution of this instruction is restricted to supervisor mode.

Execution of this instruction is context synchronizing. See *Context synchronization on page 144*.”

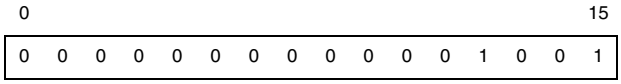
Other registers altered: MSR

Programming note: In addition to Branch to LR (**bclr**[**l**]) and Branch to CTR (**bcctr**[**l**]) instructions, **rfi** and **rfci** allow software to branch to any valid 64-bit address by using the respective 64-bit SRR0 and CSRR0.

_rfci | VLE | Supervisor | **_rfci**

Return From Critical Interrupt

se_rfci



MSR ← CSRR1

NIA ← $CSRR0_{0:62}$ || 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[32–62]||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 SRR0 or CSRR0 by the interrupt processing mechanism (see Book E) is the address 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**).

Execution of this instruction is privileged and restricted to supervisor mode.

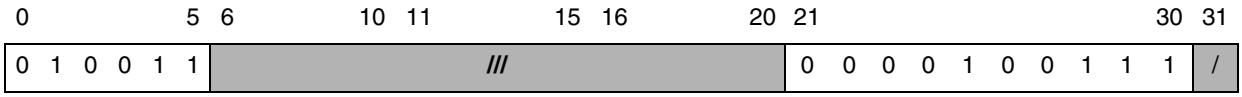
Execution of this instruction is context synchronizing.

Special Registers Altered: MSR

rfdi | Debug APU | Supervisor | **rfdi**

Return from debug interrupt

rfdi



if Mode32 then m ← 32
if Mode64 then m ← 0
MSR ← DSRR1
NIA ← $^{m}0$ || $DSRR0_{m:61}$ || 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. 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 (that is, the address in DSRR0 at the time of the execution of the **rfdi**).

Execution of this instruction is privileged and restricted to supervisor mode.

Execution of this instruction is context synchronizing.

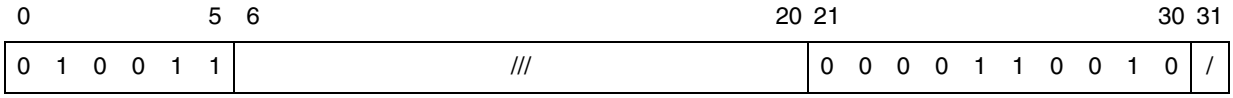
Other registers altered:

- MSR set as described above.

rfi | Book E | Supervisor | **rfi**

Return from interrupt

rfi



MSR ← SRR1
NIA ← SRR0[0:61] || 0b00

The **rfi** instruction is used to return from a non-critical 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. 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 is the address 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 **rfi**).

Execution of this instruction is restricted to supervisor mode.

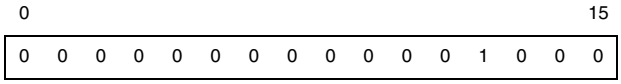
Execution of this instruction is context synchronizing. See *Context synchronization on page 144*."

Other registers altered: MSR

_rfi | VLE | Supervisor | **_rfi**

Return From Interrupt

se_rfi



MSR ← SRR1

NIA ← $SRR0_{0:62}$ || 0b0

The **se_rfi** instruction is used to return from a non-critical 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[32–62]||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 SRR0 or CSRR0 by the interrupt processing mechanism (see Book E) is the address 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**).

Execution of this instruction is privileged and restricted to supervisor mode.

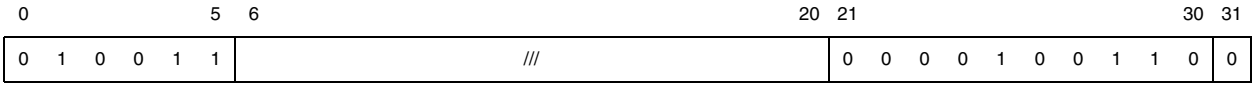
Execution of this instruction is context synchronizing.

Special Registers Altered: MSR

rfmci | Machine Check APU | Supervisor | **rfmci**

Return from Machine Check Interrupt

rfmci



MSR ← MCSRR1
NIA ← $MCSRR0_{0:61}$ || 0b00

The **rfmci** instruction is used to return from a machine check interrupt, or as a means of simultaneously establishing a new context and synchronizing on that new context.

The contents of machine check save/restore register 1 (MCSRR1) are placed into the MSR. If the new MSR value does not enable any pending exceptions, the next instruction is fetched, under control of the new MSR value from the address MCSRR0[32-61]|| 0b00. 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 is the address 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 **rfi** or **rfci**).

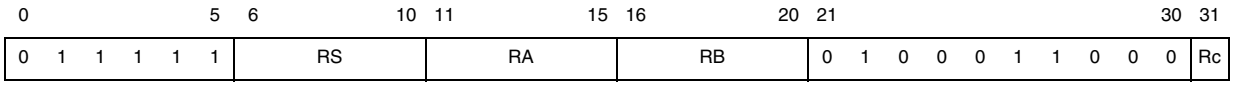
Execution of this instruction is privileged and context synchronizing.

Special registers altered: MSR

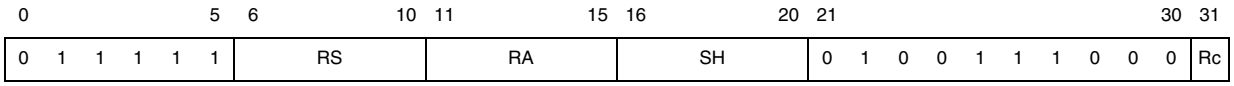
_rlw | VLE | User | **_rlw**

Rotate Left Word [Immediate]

e_rlw **r**A,**r**S,**r**B (Rc = 0)
e_rlw. **r**A,**r**S,**r**B (Rc = 1)



e_rlwi **r**A,**r**S,SH (Rc = 0)
e_rlwi. **r**A,**r**S,SH (Rc = 1)



if ‘e_rlw[.]’ then n ← $GPR(RB)_{59:63}$

else n ← SH

$result_{32:63}$ ← $ROTL_{32}(GPR(RS)_{32:63},n)$

if Rc=1 then do

LT ← $result_{32:63}$ < 0

GT ← $result_{32:63}$ > 0

EQ ← $result_{32:63}$ = 0

CR0 ← LT || GT || EQ || SO

GPR(RA) ← $result_{32:63}$

If **e_rlw[.]**, let the shift count *n* be the contents of bits 59–63 of GPR(**r**B).

If **e_rlwi[.]**, let the shift count *n* be SH.

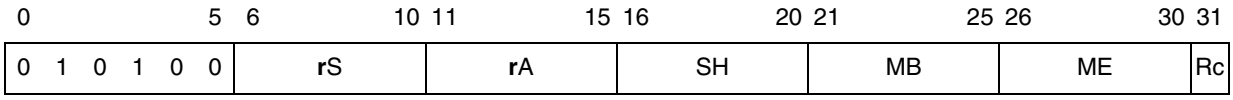
The contents of GPR(**r**S) are $rotated_{32}$ left *n* bits. The rotated data is placed into GPR(**r**A).

Special Registers Altered: CR0 (if Rc = 1)

rlwimi | Book E | User | **rlwimi**

Rotate left word immediate then mask insert

rlwimi **r**A,**r**S,SH,MB,ME (Rc=0)
rlwimi. **r**A,**r**S,SH,MB,ME (Rc=1)



n ← SH
b ← MB+32
e ← ME+32
r ← ($rS_{32:63}$,n)
m ← MASK(b,e)
$result_{0:63}$ ← r&m | rA&¬m
if Rc=1 then do
LT ← $result_{32:63}$ < 0
GT ← $result_{32:63}$ > 0
EQ ← $result_{32:63}$ = 0
CR0 ← LT || GT || EQ || SO
rA ← $result_{0:63}$

The shift count n is the value SH.

The contents of **r**S are $rotated_{32}$ left n 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 **r**A under control of the generated 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.)

Other registers altered: CR0 (if Rc=1)

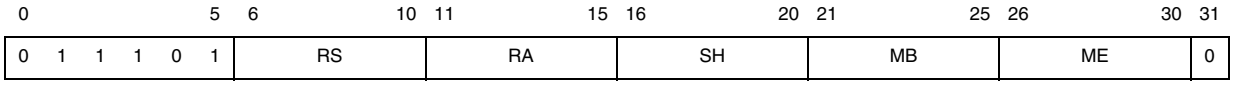
Programming note: Uses for **rlwimi**[.]:

- To insert a k-bit field that is left-justified in **r**S[32–63], into **r**A[32–63] starting at bit position j, by setting SH=64-j, MB=j-32, and ME=(j+k)-33.
- To insert an k-bit field that is right-justified in **r**S[32–63], into **r**A[32–63] starting at bit position j, by setting SH=64-(j+k), MB=j-32, and ME=(j+k)-33.

_rlwimi | VLE | User | **_rlwimi**

Rotate Left Word Immediate then Mask Insert

e_rlwimi **r**A,**r**S,SH,MB,ME



n ← SH

b ← MB+32

e ← ME+32

r ← $ROTL_{32}$(GPR(RS)$_{32:63}$,n)

m ← MASK(b,e)

result$_{32:63}$ ← r&m | GPR(RA)&¬m

GPR(RA) ← result$_{32:63}$

Let the shift count *n* be the value SH.

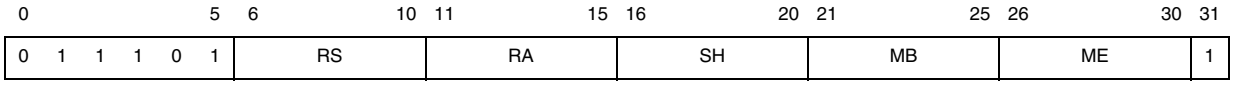
The contents of GPR(**r**S) are rotated$_{32}$ left *n* bits. A mask is generated having 1 bits from bit MB+32 through bit ME+32 and 0 bits elsewhere. The rotated data are inserted into GPR(**r**A) under control of the generated mask (if a mask bit is 1 the associated bit of the rotated data is placed into the target register, and if the mask bit is 0 the associated bit in the target register remains unchanged).

Special Registers Altered: None

_rlwinm | VLE | User | **_rlwinm**

Rotate Left Word Immediate then AND with Mask

e_rlwinm **r**A,**r**S,SH,MB,ME



n ← SH

b ← MB+32

e ← ME+32

r ← $ROTL_{32}$(GPR(RS)$_{32:63}$,n)

m ← MASK(b,e)

$result_{32:63}$ ← r & m

GPR(RA) ← $result_{32:63}$

Let the shift count *n* be SH.

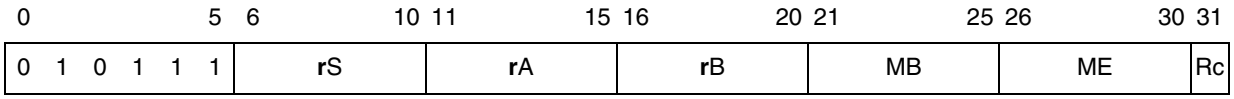
The contents of GPR(**r**S) are $rotated_{32}$ left *n* bits. A mask is generated having 1 bits from bit MB+32 through bit ME+32 and 0 bits elsewhere. The rotated data are ANDed with the generated mask and the result is placed into GPR(**r**A).

Special Registers Altered: None

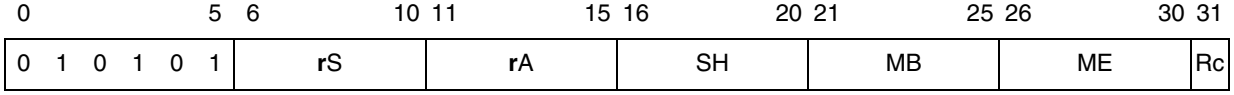
rlwnm | Book E | User | **rlwnm**

Rotate left word [immediate] then AND with mask

rlwnm **r**A,**r**S,**r**B,MB,ME (Rc=0)
rlwnm. **r**A,**r**S,**r**B,MB,ME (Rc=1)



rlwinm **r**A,**r**S,SH,MB,ME (Rc=0)
rlwinm. **r**A,**r**S,SH,MB,ME (Rc=1)



if 'rlwnm[.]' then n ← $rB_{59:63}$
else n ← SH
b ← MB+32
e ← ME+32
r ← ($rS_{32–63}$,n)
m ← MASK(b,e)
$result_{0:63}$ ← r & m
if Rc=1 then do
LT ← $result_{32:63}$ < 0
GT ← $result_{32:63}$ > 0
EQ ← $result_{32:63}$ = 0
CR0 ← LT || GT || EQ || SO
rA ← $result_{0:63}$

If **rlwnm**[**.**], the shift count, n, is the contents of **r**B[59–63]. If **rlwinm**[**.**], n is SH. The **r**S contents are $rotated_{32}$ left n bits. The mask has 1s from bit MB+32 through bit ME+32 and 0s elsewhere. The rotated data is ANDed with the mask and the result is placed into **r**A.

Other registers altered: CR0 (if Rc=1)

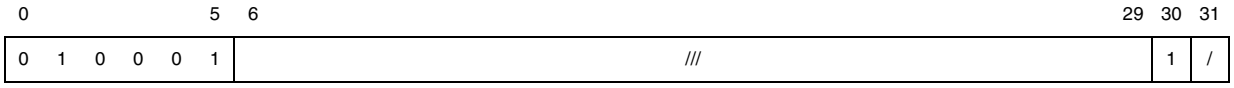
Uses for *rlwnm*[.]	Uses for *rlwinm*[.]
To extract a *k*-bit field starting at bit position *j* in **r**S[32–63], right-justified into **r**A[32–63] (clearing the remaining 32–*k* bits of **r**A[32–63])…	
…by setting **r**B[59–63]=*j*+*k*-*32*, MB=32–*k*, and ME=31.	…by setting SH=*j*+*k*-*32*, MB=32–*k*, and ME=31.
To extract a *k*-bit field that starts at bit position *j* in **r**S[32–63], left-justified into **r**A[32–63] (clearing the remaining 32–*k* bits of **r**A[32–63])…	
…by setting **r**B[59–63]=*j*-32, MB=0, and ME=*k*–1.	…by setting SH=*j*-32, MB=0, and ME=*k*–1.
To rotate the contents of bits 32–63 of a register left by *k* bits…	
…setting **r**B[59–63]=*k*, MB=0, and ME=31.	…setting SH=*k*, MB=0, and ME=31.
To rotate the contents of bits 32–63 of a register right by *k* bits…	
…by setting **r**B[59–63] =32–*k*, MB=0, and ME=31.	…by setting SH=32–*k*, MB=0, and ME=31.

Uses for *rlwnm*[.]	Uses for *rlwinm*[.]
	To shift the contents of bits 32–63 of a register right by *k* bits, by setting SH=32–*k*, MB=*k*, and ME=31. To clear the high-order *j* bits of the contents of bits 32–63 of a register and then shift the result left by *k* bits, by setting SH=*k*, MB=*j*–*k* and ME=31–*k*. To clear the low-order *k* bits of bits 32–63 of a register, by setting SH=0, MB=0, and ME=31–*k*.
For the uses given above, bits **r**A[0–31] are cleared.	

sc | Book E | Supervisor | **sc**

System call

sc



SRR1 ← MSR
SRR0 ← CIA+4
NIA ← EVPR[0:47] || IVOR8[48-59] || 0b0000
MSR[WE,EE,PR,IS,DS,FP,FE0,FE1] ← 0b0000_0000

sc is used to request a system service. A system call interrupt is generated. The MSR contents are copied into SRR1 and the address of the instruction after the **sc** instruction is placed into SRR0.

MSR[WE,EE,PR,IS,DS,FP,FE0,FE1] are cleared.

The interrupt causes the next instruction to be fetched from the address IVPR[0–47]||IVOR8[48-59]||0b0000.

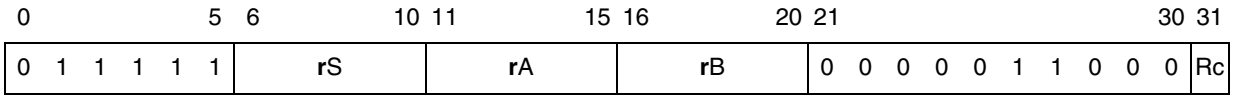
sc is context synchronizing. See *Context synchronization on page 144*."

Other registers altered: SRR0 SRR1 MSR[WE,EE,PR,IS,DS,FP,FE0,FE1]

slw | Book E | User | **slw**

Shift left word

slw **r**A,**r**S,**r**B (Rc=0)
slw. **r**A,**r**S,**r**B (Rc=1)



n ← $rB_{59:63}$
r ← ROTL32($rS_{32:63}$,n)
if rB_{58}=0 then m ← MASK(32,63-n)
else m ← $^{64}0$
$result_{0:63}$ ← r & m
if Rc=1 then do
LT ← $result_{32:63}$ < 0
GT ← $result_{32:63}$ > 0
EQ ← $result_{32:63}$ = 0
CR0 ← LT || GT || EQ || SO
rA ← $result_{0:63}$

The shift count n is the value specified by the contents of **r**B[58–63].

The contents of **r**S[32–63] are shifted left n 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 **r**A[32–63]. Bits **r**A[0–31] are cleared.

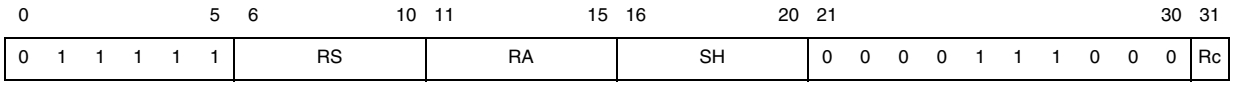
Shift amounts from 32 to 63 give a zero result.

Other registers altered: CR0 (if Rc=1)

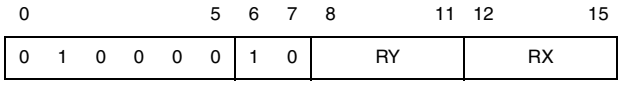
_slw*x* | VLE | User | **_slw*x***

Shift Left Word [Immediate] [and Record]

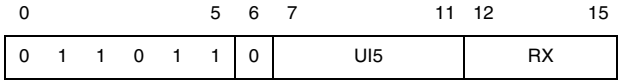
e_slwi **r**A,**r**S,SH (Rc = 0)
e_slwi. **r**A,**r**S,SH (Rc = 1)



se_slw **r**X,**r**Y



se_slwi **r**X,UI5



if ‘e_slwi[.]’ then n ← SH

if se_slw then n ← $GPR(RY)_{58:63}$
if se_slwi then n ← UI5

r ← $ROTL_{32}(GPR(RS \text{ or } RX)_{32:63}, n)$

if n<32 then m ← MASK(32,63-n)

else m ← $^{32}0$

$result_{32:63}$ ← r & m

if Rc=1 then do

LT ← $result_{32:63} < 0$

GT ← $result_{32:63} > 0$

EQ ← $result_{32:63} = 0$

CR0 ← LT || GT || EQ || SO

GPR(RA or RX) ← $result_{32:63}$

Let the shift count *n* be the value specified by the contents of bits 58–63 of GPR(**r**B or **r**Y), or by the value of the SH or UI5 field.

The contents of bits 32–63 of GPR(**r**S or **r**X) are shifted left *n* 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 bits 32–63 of GPR(**r**A or **r**X).

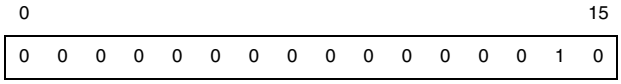
Shift amounts from 32 to 63 give a zero result.

Special Registers Altered: CR0 (if Rc = 1)

_sc | VLE | User | **_sc**

System Call

se_sc



SRR1 ← MSR

SRR0 ← CIA+2

NIA ← $IVPR_{32:47}$ || $IVOR8_{48:59}$ || 0b0000

$MSR_{WE,EE,PR,IS,DS,FP,FE0,FE1}$ ← 0b0000_0000

se_sc is used to request a system service. A system call interrupt is generated. The contents of the MSR are copied into SRR1 and the address of the instruction after the **se_sc** instruction is placed into SRR0.

MSR[WE,EE,PR,IS,DS,FP,FE0,FE1] are cleared.

The interrupt causes the next instruction to be fetched from the address

IVPR[32–47]||IVOR8[48–59]||0b0000

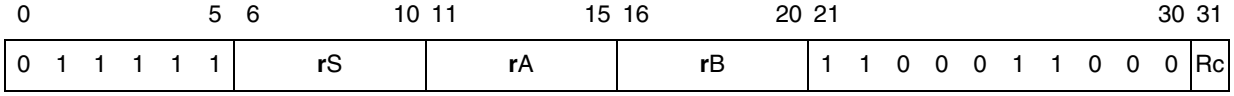
This instruction is context synchronizing.

Special Registers Altered: SRR0 SRR1 MSR[WE,EE,PR,IS,DS,FP,FE0,FE1]

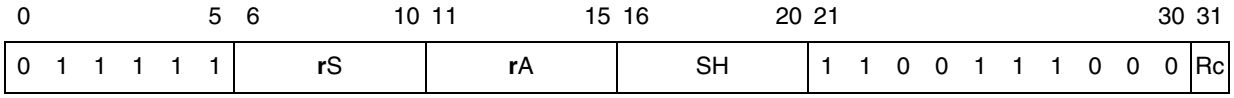
sraw | Book E | User | **sraw**

Shift right algebraic word [immediate]

sraw **r**A,**r**S,**r**B (Rc=0)
sraw. **r**A,**r**S,**r**B (Rc=1)



srawi **r**A,**r**S,SH (Rc=0)
srawi. **r**A,**r**S,SH (Rc=1)



if ‘sraw[.]’ then n ← $rB_{59:63}$
else n ← SH
r ← ROTL64(rS[32:63],64-n)
if ‘sraw[.]’ & rB_{58}=1 then m ← $^{64}0$
else m ← MASK(n+32,63)
s ← rS_{32}
$result_{0:63}$ ← r&m | (^{64}s)&¬m
if Rc=1 then do
LT ← $result_{32:63}$ < 0
GT ← $result_{32:63}$ > 0
EQ ← $result_{32:63}$ = 0
CR0 ← LT || GT || EQ || SO
rA ← $result_{0:63}$
CA ← s & $((r\&\neg m)_{32:63} \neq 0)$

If **sraw**[**.**], the shift count n is the contents of **r**B[58–63].

If **srawi**[**.**], the shift count n is the value of the SH field.

The contents of **r**S[32–63] are shifted right n bits. Bits shifted out of position 63 are lost. Bit 32 of **r**S is replicated to fill the vacated positions on the left. The 32-bit result is placed into **r**A[32–63]. **r**S[32] is replicated to fill bits **r**A[0–31].

CA is set if **r**S[32–63] contain a negative value and any 1 bits are shifted out of bit position 63; otherwise CA is cleared.

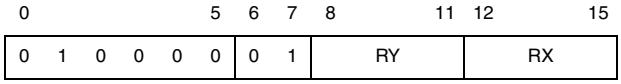
A shift amount of zero causes **r**A to receive EXTS(**r**S[32–63]), and CA to be cleared. For **sraw**[**.**] shift amounts from 32 to 63 give a result of 64 signed bits, and cause CA to receive **r**S[32] (that is, sign bit of **r**S[32–63]).

Other registers altered: CA CR0 (if Rc=1)

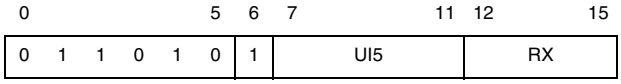
_sraw*x* | VLE | User | **_sraw*x***

Shift Right Algebraic Word [Immediate] [and Record]

se_sraw **r**X,**r**Y



se_srawi **r**X,UI5



if 'se_sraw' then n ← $GPR(RY)_{59:63}$
if 'se_srawi' then n ← UI5

r ← $ROTL_{32}(GPR(RS \text{ or } RX)_{32:63}, 32-n)$

if ((se_sraw & $GPR(RY)_{58}$=1) then m ← $^{32}0$

else m ← MASK(n+32,63)

s ← $GPR(RS \text{ or } RX)_{32}$

$result_{0:63}$ ← r&m | (^{32}s)&¬m

if Rc=1 then do

LT ← $result_{32:63} < 0$

GT ← $result_{32:63} > 0$

EQ ← $result_{32:63} = 0$

CR0 ← LT || GT || EQ || SO

GPR(RA or RX) ← $result_{32:63}$

CA ← s & $((r\&\neg m)_{32:63} \neq 0)$

If **se_sraw**, let the shift count *n* be the contents of bits 58–63 of GPR(**r**Y).

If **se_srawi**, let the shift count *n* be the value of the UI5 field.

The contents of bits 32–63 of GPR(**r**S or **r**X) are shifted right *n* bits. Bits shifted out of position 63 are lost. Bit 32 of **r**S or **r**X is replicated to fill vacated positions on the left. The 32-bit result is placed into bits 32–63 of GPR(**r**A or **r**X).

CA is set if bits 32–63 of GPR(**r**S or **r**X) contain a negative value and any 1 bits are shifted out of bit position 63; otherwise CA is cleared.

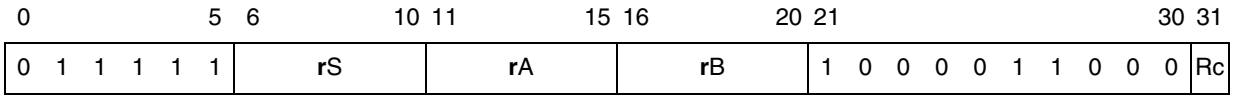
A shift amount of zero causes GPR(**r**A or **r**X) to receive EXTS($GPR(\textbf{r}S \text{ or } \textbf{r}X)_{32:63}$), and CA to be cleared. For **se_sraw,** shift amounts from 32 to 63 give a result of 64 sign bits, and cause CA to receive bit 32 of the contents of GPR(**r**S or **r**X) (that is, sign bit of $GPR(\textbf{r}S \text{ or } \textbf{r}X)_{32:63}$).

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

srw | Book E | User | **srw**

Shift right word

srw **r**A,**r**S,**r**B (Rc=0)
srw. **r**A,**r**S,**r**B (Rc=1)



n ← $rB_{59–63}$
r ← ROTL64($rS_{32–63}$,64-n)
if rB_{58}=0 then m ← MASK(n+32,63)
else m ← $^{64}0$
$result_{0:63}$ ← r & m
if Rc=1 then do
LT ← $result_{32:63}$ < 0
GT ← $result_{32:63}$ > 0
EQ ← $result_{32:63}$ = 0
CR0 ← LT || GT || EQ || SO
rA ← $result_{0:63}$

The shift count n is the value specified by the contents of **r**B[58–63].

The contents of **r**S[32–63] are shifted right n 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 **r**A[32–63]. Bits **r**A[0–31] are cleared.

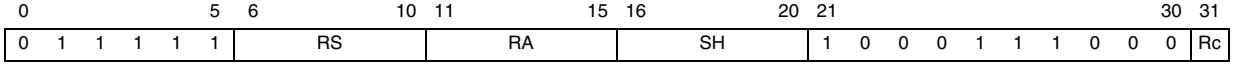
Shift amounts from 32 to 63 give a zero result.

Other registers altered: CR0 (if Rc=1)

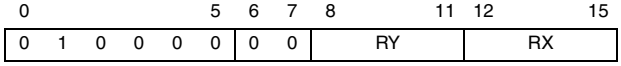
_srw*x* | VLE | User | **_srw*x***

Shift Right Word [Immediate] [and Record]

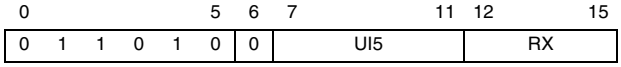
e_srwi **r**A,**r**S,SH (Rc = 0)
e_srwi. **r**A,**r**S,SH (Rc = 1)



se_srw **r**X,**r**Y



se_srwi **r**X,UI5



$n \leftarrow GPR(RB)_{59:63}$

if ‘e_srwi[.]’ then $n \leftarrow SH$
if ‘se_srw’ then $n \leftarrow GPR(RY)_{59:63}$
if ‘se_srwi’ then $n \leftarrow UI5$
$r \leftarrow ROTL_{32}(GPR(RS \text{ or } RX)_{32:63}, 32-n)$

if $((se\_srw \ \& \ GPR(RY)_{58}=1)$ then $m \leftarrow {}^{32}0$

else $m \leftarrow MASK(n+32,63)$

$result_{32:63} \leftarrow r \ \& \ m$

if Rc=1 then do

$LT \leftarrow result_{32:63} < 0$

$GT \leftarrow result_{32:63} > 0$

$EQ \leftarrow result_{32:63} = 0$

$CR0 \leftarrow LT \ || \ GT \ || \ EQ \ || \ SO$

$GPR(RA \text{ or } RX) \leftarrow result_{32:63}$

If **e_srwi**, let the shift count *n* be the value of the SH field.

If **se_srw**, let the shift count *n* be the contents of bits 58–63 of GPR(**r**Y).

If **se_srwi**, let the shift count *n* be the value of the UI5 field.

The contents of bits 32–63 of GPR(**r**S or **r**X) are shifted right *n* 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 bits 32–63 of GPR(**r**A or **r**X).

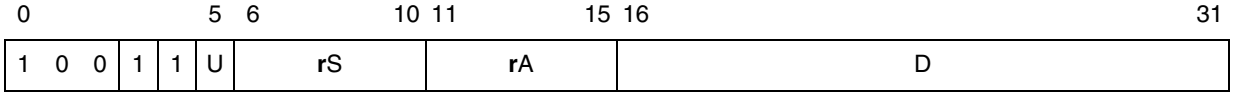
Shift amounts from 32 to 63 give a zero result.

Special Registers Altered: CR0 (if Rc = 1)

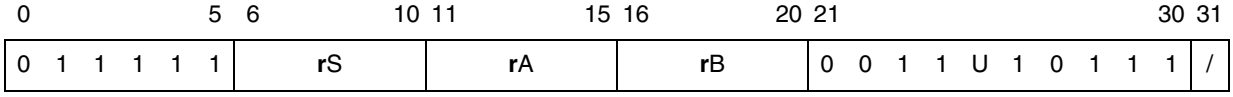
stb | Book E | User | **stb**

Store byte [with update] [indexed]

stb **r**S,D(**r**A) (D-mode, I=0)
stbu **r**S,D(**r**A) (D-mode, I=1)



stbx **r**S,**r**A,**r**B (X-mode, U=0)
stbux **r**S,**r**A,**r**B (X-mode, U=1)



if rA=0 then a ← $^{64}0$ else a ← rA
if D-mode then EA ← $^{32}0$ || (a + EXTS(D))$_{32:63}$
if X-mode then EA ← $^{32}0$ || (a + rB)$_{32:63}$
MEM(EA,1) ← rS$_{56:63}$
if U=1 then rA ← EA

The EA is calculated as follows:

- For **stb** and **stbu**, EA is bits 32–63 of the sum of the contents of **r**A, or 64 zeros if **r**A=0, and the sign-extended value of the D field.
- For **stbx** and **stbux**, EA is bits 32–63 of the sum of the contents of **r**A, or 64 zeros if **r**A=0, and the contents of **r**B.

The contents of **r**S[56–63] are stored into the byte addressed by EA.

If U=1 (with update), EA is placed into **r**A.

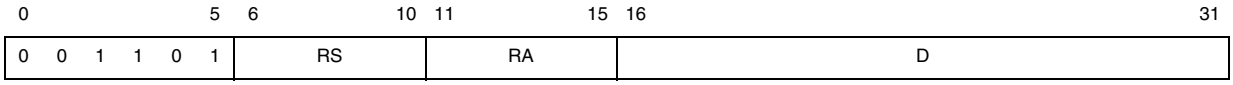
If U=1 (with update) and **r**A=0, the instruction form is invalid.

Other registers altered: None

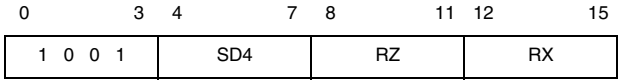
_stb*x* | VLE | User | **_stb*x***

Store Byte [with Update] [Indexed]

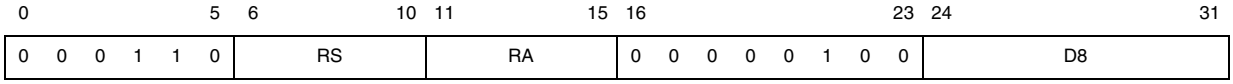
e_stb **r**S,D(**r**A) (D-mode)



se_stb **r**Z,SD4(**r**X) (SD4-mode)



e_stbu **r**S,D8(**r**A) (D8-mode)



if (RA=0 & !se_stb) then a ← $^{32}0$ else a ← GPR(RA or RX)

if D-mode then EA ← $(a + EXTS(D))_{32:63}$

if D8-mode then EA ← $(a + EXTS(D8))_{32:63}$

if SD4-mode then EA ← $(a + (^{28}0 \,||\, SD4))_{32:63}$

$MEM(EA,1) \leftarrow GPR(RS \text{ or } RZ)_{56:63}$

if e_stbu then GPR(RA) ← EA

Let the EA be calculated as follows:

- For **e_stb** and **e_stbu**, let EA be the sum of the contents of GPR(**r**A), or 32 0s if **r**A = 0, and the sign-extended value of the D or D8 instruction field.
- For **se_stb**, let EA be the sum of the contents of GPR(**r**X) and the zero-extended value of the SD4 instruction field.

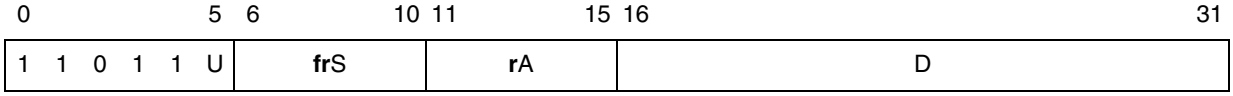
The contents of bits 56–63 of GPR(**r**S) are stored into the byte in memory addressed by EA.

- If **e_stbu**, EA is placed into GPR(**r**A).
- If **e_stbu** and **r**A = 0, the instruction form is invalid.
- None

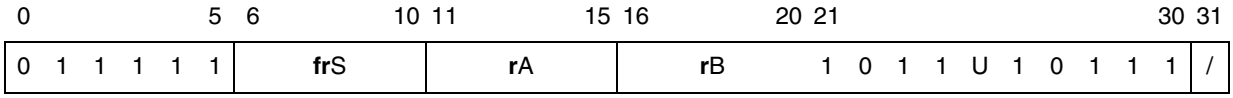
stfd | Book E | User | **stfd**

Store floating-point double [with update] [indexed]

stfd **fr**S,D(**r**A) (D-mode, U=0)
stfdu **fr**S,D(**r**A) (D-mode, U=1)



stfdx **fr**S,**r**A,**r**B (X-mode, U=0)
stfdux **fr**S,**r**A,**r**B (X-mode, U=1)



if rA=0 then a ← $^{64}0$ else a ← rA
if D-mode then EA ← $^{32}0$ || $(a + EXTS(D))_{32:63}$
if X-mode then EA ← $^{32}0$ || $(a + rB)_{32:63}$
MEM(EA,8) ← frS
if U=1 then rA ← EA

The EA is calculated as follows:

- For **stfd** and **stfdu**, EA is 32 zeros concatenated with bits 32–63 of the sum of the contents of **r**A, or 64 zeros if **r**A=0, and the sign-extended value of the D instruction field.
- For **stfdx** and **stfdux**, EA is 32 zeros concatenated with bits 32–63 of the sum of the contents of **r**A, or 64 zeros if **r**A=0, and the contents of **r**B.

The contents of **fr**S are stored into the double word addressed by EA.

If U=1 (with update), EA is placed into **r**A.

If U=1 (with update) and **r**A=0, the instruction form is invalid.

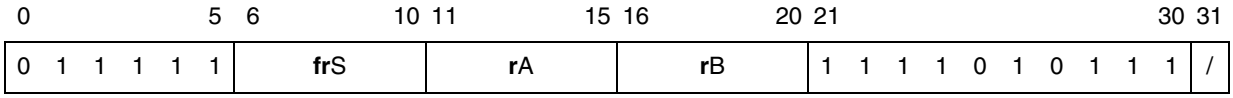
If MSR[FP]=0, **stfd**[**u**][**x**] causes a floating-point unavailable interrupt.

Other registers altered: None

stfiwx | Book E | User | **stfiwx**

Store floating-point as integer word indexed

stfiwx **fr**S,**r**A,**r**B



if rA=0 then a ← $^{64}0$ else a ← rA
EA ← $^{32}0$ || $(a + rB)_{32:63}$
MEM(EA,4) ← frS[32:63]

The EA is calculated as follows:

- For **stfiwx**, EA is 32 zeros concatenated with bits 32–63 of the sum of the contents of **r**A, or 64 zeros if **r**A=0, and the contents of **r**B.

The contents of **fr**S[32–63] are stored, without conversion, into the word addressed by EA.

If the contents of **fr**S were produced, either directly or indirectly, by a load floating-point single instruction, a single-precision arithmetic instruction, or **frsp**, the value stored is undefined. (The contents of **fr**S are produced directly by such an instruction if **fr**S is the target register for the instruction. The contents of **fr**S are produced indirectly by such an instruction if **fr**S 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.)

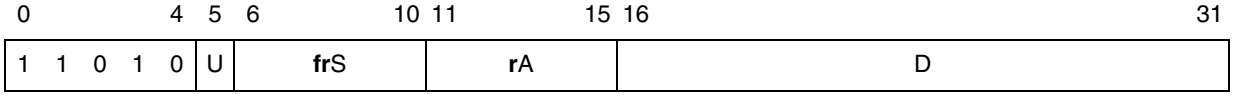
If MSR[FP]=0, an attempt to execute **stfiwx** causes a floating-point unavailable interrupt.

Other registers altered: None

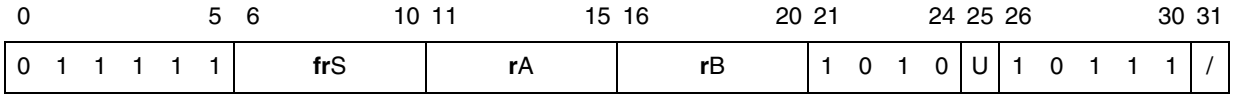
stfs | Book E | User | **stfs**

Store floating-point single [with update] [indexed]

stfs **fr**S,D(**r**A) (D-mode, U=0)
stfsu **fr**S,D(**r**A) (D-mode, U=1)



stfsx **fr**S,**r**A,**r**B (X-mode, U=0)
stfsux **fr**S,**r**A,**r**B (X-mode, U=1)



if rA=0 then a ← $^{64}0$ else a ← rA
if D-mode then EA ← $^{32}0$ || $(a + EXTS(D))_{32:63}$
if X-mode then EA ← $^{32}0$ || $(a + rB)_{32:63}$
MEM(EA,4) ← SINGLE(frS)
if U=1 then rA ← EA

The EA is calculated as follows:

- For **stfs** and **stfsu**, EA is 32 zeros concatenated with bits 32–63 of the sum of the contents of **r**A, or 64 zeros if **r**A=0, and the sign-extended value of the D field.
- For **stfsx** and **stfsux**, EA is 32 zeros concatenated with bits 32–63 of the sum of the contents of **r**A, or 64 zeros if **r**A=0, and the contents of **r**B.

The contents of **fr**S are converted to single format and stored into the word addressed by EA.

If U=1 (with update), EA is placed into **r**A.

If U=1 (with update) and **r**A=0, the instruction form is invalid.

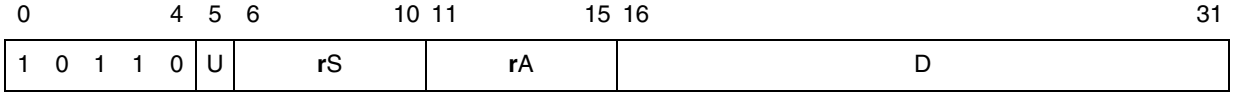
If MSR[FP]=0, **stfs**[**u**][**x**] causes a floating-point unavailable interrupt.

Other registers altered: None

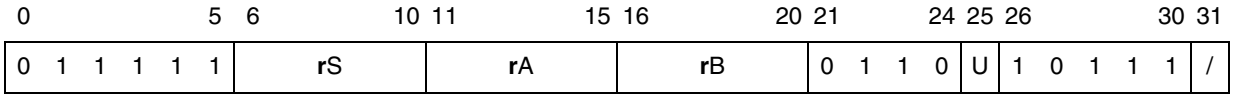
sth | Book E | User | **sth**

Store half word [with update] [indexed]

sth **r**S,D(**r**A) (D-mode, U=0)
sthu **r**S,D(**r**A) (D-mode, U=1)



sthx **r**S,**r**A,**r**B (X-mode, U=0)
sthux **r**S,**r**A,**r**B (X-mode, U=1)



if rA=0 then a ← $^{64}0$ else a ← rA
if D-mode then EA ← $^{32}0$ || (a + EXTS(D))$_{32:63}$
if X-mode then EA ← $^{32}0$ || (a + rB)$_{32:63}$
MEM(EA,2) ← rS$_{48:63}$
if U=1 then rA ← EA

The EA is calculated as follows:

- For **sth** and **sthu**, EA is bits 32–63 of the sum of the contents of **r**A, or 64 zeros if **r**A=0, and the sign-extended value of the D field.
- For **sthx** and **sthux**, EA is bits 32–63 of the sum of the contents of **r**A, or 64 zeros if **r**A=0, and the contents of **r**B.

The contents of **r**S[48–63] are stored into the half word addressed by EA.

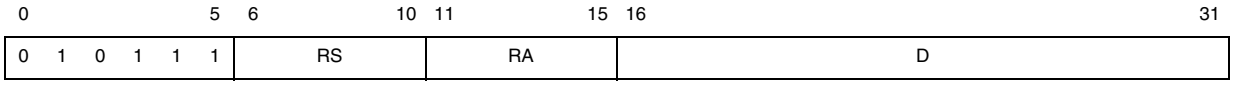
If U=1 (with update), EA is placed into **r**A.
If U=1 (with update) and **r**A=0, the instruction form is invalid.

Other registers altered: None

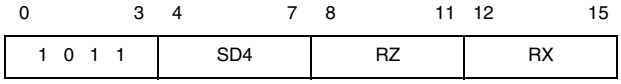
_sthx | VLE | User | **_sthx**

Store Halfword [with Update] [Indexed]

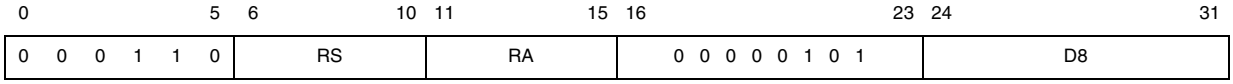
e_sth **r**S,D(**r**A) (D-mode)



se_sth **r**Z,SD4(**r**X) (SD4-mode)



e_sthu **r**S,D8(**r**A) (D8-mode)



if (RA=0 & !se_sth) then a ← $^{32}0$ else a ← GPR(RA or RX)

if D-mode then EA ← $(a + EXTS(D))_{32:63}$

if D8-mode then EA ← $(a + EXTS(D8))_{32:63}$

if SD4-mode then EA ← $(a + (^{27}0 \,||\, SD4 \,||\, 0))_{32:63}$

MEM(EA,2) ← $GPR(RS \text{ or } RZ)_{48:63}$

if e_sthu then GPR(RA) ← EA

Let the EA be calculated as follows:

- For **e_sth** and **e_sthu**, let EA be the sum of the contents of GPR(**r**A), or 32 0s if **r**A = 0, and the sign-extended value of the D or D8 instruction field.
- For **se_sth** let EA be the sum of the contents of GPR(**r**X) and the zero-extended value of the SD4 instruction field shifted left by 1 bit.

The contents of bits 48–63 of GPR(**r**S) are stored into the half word in memory addressed by EA.

If **e_sthu**, EA is placed into GPR(**r**A).

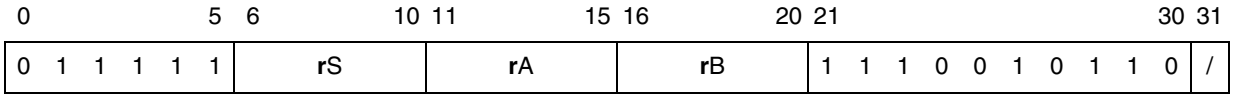
If **e_sthu** and **r**A = 0, the instruction form is invalid.

Special Registers Altered: None

sthbrx | Book E | User | **sthbrx**

Store half word byte-reverse

sthbrx **r**S,**r**A,**r**B



if rA=0 then a ← $^{64}0$ else a ← rA
EA ← $^{32}0$ || $(a + rB)_{32:63}$
MEM(EA,2) ← $rS_{56:63}$ || $rS_{48:55}$

The EA is calculated as follows:

- For **sthbrx**, EA is bits 32–63 of the sum of the contents of **r**A, or 64 zeros if **r**A=0, and the contents of **r**B.

rS[56–63] are stored into bits 0–7 of the half word addressed by EA. Bits 48–55 of **r**S are stored into bits 8–15 of the half word addressed by EA.

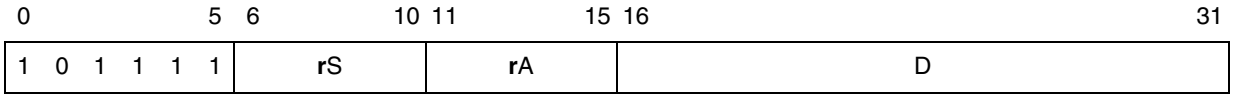
Other registers altered: None

Programming note: When EA references big-endian memory, these instructions have the effect of storing data in little-endian byte order. Likewise, when EA references little-endian memory, these instructions have the effect of storing data in big-endian byte order.

stmw | Book E | User | **stmw**

Store multiple word

stmw **r**S,D(**r**A)



if rA=0 then EA ← $^{32}0$ || EXTS(D)$_{32:63}$
else EA ← $^{32}0$ || (rA+EXTS(D))$_{32:63}$
r ← rS
do while r ≤ 31
MEM(EA,4) ← GPR(r)$_{32:63}$
r ← r + 1
EA ← $^{32}0$ || (EA+4)$_{32:63}$

The EA is bits 32–63 of the sum of the contents of **r**A, or 64 zeros if **r**A=0, and the sign-extended value of the D instruction field.

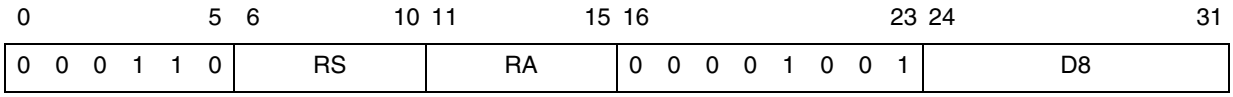
EA must be a multiple of 4. If it is not, either an alignment interrupt is invoked or the results are boundedly undefined.

Other registers altered: None

_stmw | VLE | User | **_stmw**

Store Multiple Word

e_stmw **r**S,D8**(r**A**)** (D8-mode)



if RA=0 then EA ← EXTS(D8)$_{32:63}$

else EA ← (GPR(RA)+EXTS(D8))$_{32:63}$

r ← RS

do while r ≤ 31

MEM(EA,4) ← GPR(r)$_{32:63}$

r ← r + 1

EA ← (EA+4)$_{32:63}$

Let the EA be the sum of the contents of GPR(**r**A), or 32 0s if **r**A = 0, and the sign-extended value of the D8 instruction field.

Let n = (32 - **r**S). Bits 32–63 of registers GPR(**r**S) through GPR(31) are stored in n consecutive words in memory starting at address EA.

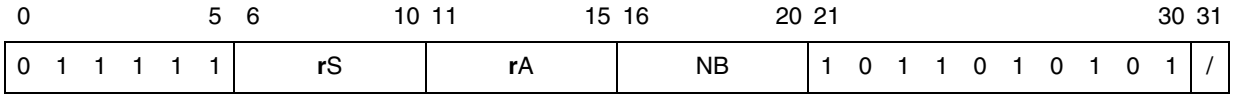
EA must be a multiple of 4. If it is not, either an alignment interrupt is invoked or the results are boundedly undefined.

Special Registers Altered: None

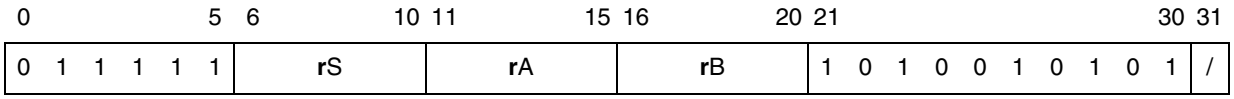
stswi | Book E | User | **stswi**

Store string word (immediate | indexed)

stswi **r**S,**r**A,NB



stswx **r**S,**r**A,**r**B



if rA=0 then a ← $^{64}0$ else a ← rA
if 'stswi' then EA ← $^{32}0$ || $a_{32:63}$
if 'stswx' then EA ← $^{32}0$ || $(a + rB)_{32:63}$
if 'stswi' & NB=0 then n ← 32
if 'stswi' & NB≠0 then n ← NB
if 'stswx' then n ← $XER_{57:63}$
r ← rS - 1
i ← 32
do while n > 0
if i=32 then r ← r + 1 (mod 32)
MEM(EA,1) ← $GPR(r)_{i:i+7}$
i ← i + 8
if i = 64 then i ← 32
EA ← $^{32}0$ || $(EA+1)_{32:63}$
n ← n - 1

The EA is calculated as follows:

- For **stswi**, EA is 32 zeros concatenated with bits 32–63 of the contents of **r**A, or 32 zeros if **r**A=0.
- For **stswx**, EA is 32 zeros concatenated with bits 32–63 of the sum of the contents of **r**A, or 64 zeros if **r**A=0, and the contents of **r**B.

If **stswi**, let n=NB if NB≠0, n=32 if NB=0. If **stswx**, 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.

n consecutive bytes starting at EA are stored from registers **r**S through GPR(**r**S+nr–1). Data is stored from the low-order 4 bytes of each GPR.

Bytes are stored left to right from each GPR. The register sequence can wrap to GPR0.

If **stswx** and n=0, no bytes are stored.

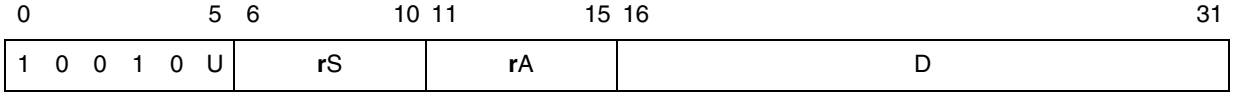
Other registers altered: None

Programming note: Store string word and load string word instructions allow movement of data between memory and registers without concern for alignment. They can be used for a short move between arbitrary locations or long moves between misaligned memory fields.

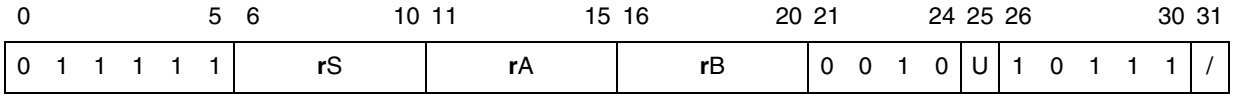
stw | Book E | User | **stw**

Store word [with update] [indexed]

stw **r**S,D(**r**A) (D-mode, U=0)
stwu **r**S,D(**r**A) (D-mode, U=1)



stwx **r**S,**r**A,**r**B (X-mode, U=0)
stwux **r**S,**r**A,**r**B (X-mode, U=1)



if rA=0 then a ← $^{64}0$ else a ← rA
if D-mode then EA ← $^{32}0$ || (a + EXTS(D))$_{32:63}$
if X-mode then EA ← $^{32}0$ || (a + rB)$_{32:63}$
MEM(EA,4) ← rS$_{[32:63]}$
if U=1 then rA ← EA

The EA is calculated as follows:

- For **stw** and **stwu**, EA is bits 32–63 of the sum of the contents of **r**A, or 64 zeros if **r**A=0, and the sign-extended value of the D field.
- For **stwx** and **stwux**, EA is bits 32–63 of the sum of the contents of **r**A, or 64 zeros if **r**A=0, and the contents of **r**B.

The contents of **r**S[32–63] are stored into the word addressed by EA.

If U=1 (with update), EA is placed into **r**A.

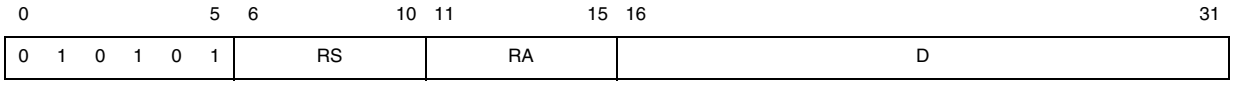
If U=1 (with update) and **r**A=0, the instruction form is invalid.

Other registers altered: None

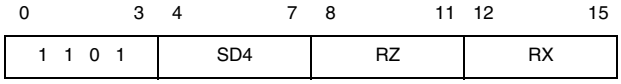
_stwx | VLE | User | **_stwx**

Store Word [with Update] [Indexed]

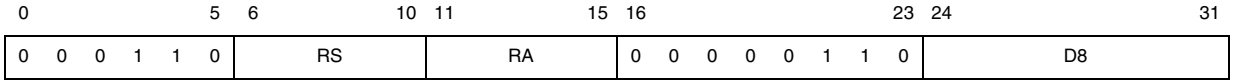
e_stw **r**S,D(**r**A) (D-mode)



se_stw **r**Z,SD4(**r**X) (SD4-mode)



e_stwu **r**S,D8(**r**A) (D8-mode)



if (RA=0 & !se_stw) then a ← $^{32}0$ else a ← GPR(RA or RX)

if D-mode then EA ← $(a + EXTS(D))_{32:63}$

if D8-mode then EA ← $(a + EXTS(D8))_{32:63}$

if SD4-mode then EA ← $(a + (^{26}0 \,||\, SD4 \,||\, ^{2}0))_{32:63}$

MEM(EA,4) ← $GPR(RS \text{ or } RZ)_{32:63}$

Let the EA be calculated as follows:

- For **e_stw** and **e_stwu**, let EA be the sum of the contents of GPR(**r**A), or 32 0s if **r**A = 0, and the sign-extended value of the D or D8 instruction field.
- For **se_stw**, let EA be the sum of the contents of GPR(**r**X) and the zero-extended value of the SD4 instruction field shifted left by 2 bits.

The contents of bits 32–63 of GPR(**r**S) are stored into the word in memory addressed by EA.

If **e_stwu**, EA is placed into GPR(**r**A).

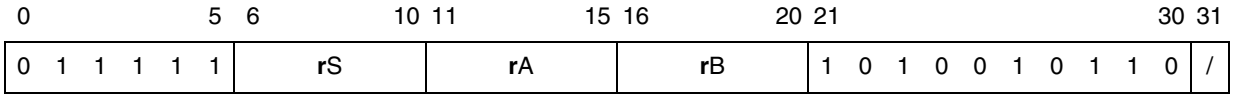
If **e_stwu** and **r**A = 0, the instruction form is invalid.

Special Registers Altered: None

stwbrx | Book E | User | **stwbrx**

Store word byte-reverse

stwbrx **r**S,**r**A,**r**B



if rA=0 then a ← $^{64}0$ else a ← rA
EA ← $^{32}0$ || $(a + rB)_{32:63}$
MEM(EA,4) ← $rS_{56:63}$ || $rS_{48:55}$ || $rS_{40:47}$ || $rS_{32:39}$

The EA is calculated as follows:

- For **stwbrx**, EA is bits 32–63 of the sum of the contents of **r**A, or 64 zeros if **r**A=0, and the contents of **r**B.

Bits 56–63 of **r**S are stored into bits 0–7 of the word addressed by EA. Bits 48–55 of **r**S are stored into bits 8–15 of the word addressed by EA. Bits 40–47 of **r**S are stored into bits 16–23 of the word addressed by EA. Bits 32–39 of **r**S are stored into bits 24–31 of the word addressed by EA.

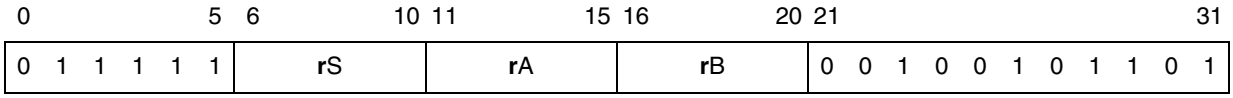
Other registers altered: None

Programming note: When EA references big-endian memory, these instructions have the effect of storing data in little-endian byte order. Likewise, when EA references little-endian memory, these instructions have the effect of storing data in big-endian byte order.

stwcx. | Book E | User | **stwcx.**

Store word conditional indexed

stwcx. **r**S,**r**A,**r**B



if rA=0 then a ← $^{64}0$ else a ← rA
EA ← $^{32}0$ || $(a + rB)_{32:63}$
if RESERVE then
  if RESERVE_ADDR = real_addr(EA) then
    MEM(EA,4) ← $rS_{[32:63]}$
    CR0 ← 0b00 || 0b1 || XER_{SO}
  else
    u ← undefined 1-bit value
    if u then MEM(EA,4) ← $rS_{[32:63]}$
    CR0 ← 0b00 || u || XER_{SO}
  RESERVE ← 0
else
  CR0 ← 0b00 || 0b0 || XER_{SO}

The EA is calculated as follows:

- For **stwcx.**, EA is bits 32–63 of the sum of the contents of **r**A, or 64 zeros if **r**A=0, and the contents of **r**B.

If a reservation exists and the address specified by the **stwcx.** is the same as that specified by the **lwarx** instruction that established the reservation, the contents of **r**S[32–63] are stored into the word addressed by EA and the reservation is cleared.

If a reservation exists but the address specified by **stwcx.** is not the same as that specified by the load and reserve instruction that established the reservation, the reservation is cleared, and it is undefined whether the instruction completes without altering memory.

If a reservation does not exist, the instruction completes without altering memory.

CR field 0 is set to reflect whether the store operation was performed, as follows:

CR0[LT,GT,EQ,SO] = 0b00 || store_performed || XER[SO]

EA must be a multiple of 4. If it is not, either an alignment interrupt is invoked or the results are boundedly undefined.

Other registers altered: CR0

Programming notes:

- **stwcx.**, in combination with **lwarx**, permits the programmer to write a sequence of instructions that appear to perform an atomic update operation on a memory location. This operation depends on a single reservation resource in each processor. At most one reservation exists on any given processor: there are not separate reservations for words and for double words.
- Because **stwcx.** instructions have implementation dependencies (such as 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 (such as, test and set, and compare

and swap) needed by application programs. Application programs should use these library programs, rather than use **stwcx.** directly.

- The granularity with which reservations are managed is implementation-dependent. Therefore, the memory to be accessed by **stwcx.** should be allocated by a system library program. Additional information can be found in *Atomic update primitives using lwarx and stwcx. on page 176*."
- When correctly used, the load and reserve and store conditional instructions can provide an atomic update function for a single aligned word (**lwarx** and **stwcx.**) of memory. In general, correct use requires that **lwarx** be paired with **stwcx.** with the same address specified by both instructions of the pair. The only exception is that an unpaired **stwcx.** to any (scratch) effective address can be used to clear any reservation held by the processor. Examples of correct uses of these instructions to emulate primitives such as fetch and add, test and set, and compare and swap can be found in *Appendix C: Programming examples on page 1143*.

A reservation is cleared if any of the following events occur:

- The processor holding the reservation executes another load and reserve instruction; this clears the first reservation and establishes a new one.
- The processor holding the reservation executes a store conditional instruction to any address.
- Another processor executes any store instruction to the address associated with the reservation.
- Any mechanism, other than the processor holding the reservation, stores to the address associated with the reservation.

See *Atomic update primitives using lwarx and stwcx. on page 176*," for additional information.

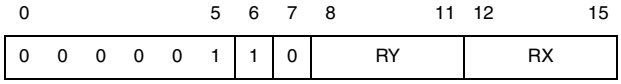
_sub

VLE	User

_sub

Subtract

se_sub **rX,rY**



$sum_{32:63} \leftarrow GPR(RX) + \neg GPR(RY) + 1$

$GPR(RX) \leftarrow sum_{32:63}$

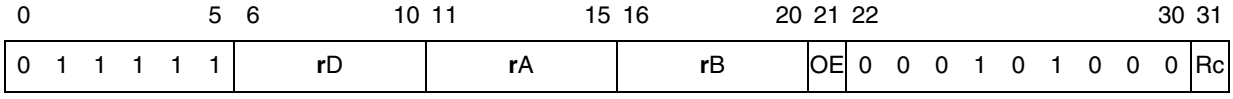
The sum of the contents of GPR(**r**X), the one's complement of contents of GPR(**r**Y), and 1 is placed into GPR(**r**X).

Special Registers Altered: None

subf | Book E | User | **subf**

Subtract from

subf	**r**D,**r**A,**r**B	(OE=0, Rc=0)
subf.	**r**D,**r**A,**r**B	(OE=0, Rc=1)
subfo	**r**D,**r**A,**r**B	(OE=1, Rc=0)
subfo.	**r**D,**r**A,**r**B	(OE=1, Rc=1)



$carry_{0:63} \leftarrow Carry(\neg rA + rB + 1)$
$sum_{0:63} \leftarrow \neg rA + rB + 1$
if OE=1 then do
$OV \leftarrow carry_{32} \oplus carry_{33}$
$SO \leftarrow SO \mid (carry_{32} \oplus carry_{33})$
if Rc=1 then do
$LT \leftarrow sum_{32:63} < 0$
$GT \leftarrow sum_{32:63} > 0$
$EQ \leftarrow sum_{32:63} = 0$
$CR0 \leftarrow LT \mid\mid GT \mid\mid EQ \mid\mid SO$
$rD \leftarrow sum$

The sum of the one's complement of the contents of **r**A, the contents of **r**B, and 1 is placed into **r**D.

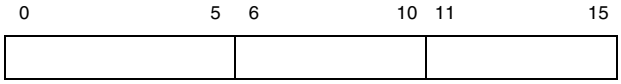
Other registers altered:

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

_subf*x* | VLE | User | **_subf*x***

Subtract From

se_subf **r**X,**r**Y



$sum_{32:63} \leftarrow \neg GPR(RX) + GPR(RY) + 1$

$GPR(RX) \leftarrow sum_{32:63}$

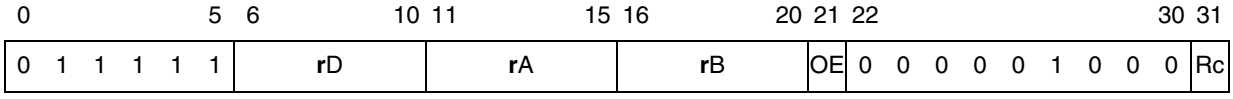
The sum of the one's complement of the contents of GPR(**r**X), the contents of GPR(**r**Y), and 1 is placed into GPR(**r**X).

Special Registers Altered: None

subfc | Book E | User | **subfc**

Subtract from carrying

subfc	**r**D,**r**A,**r**B	(OE=0, Rc=0)
subfc.	**r**D,**r**A,**r**B	(OE=0, Rc=1)
subfco	**r**D,**r**A,**r**B	(OE=1, Rc=0)
subfco.	**r**D,**r**A,**r**B	(OE=1, Rc=1)



$carry_{0:63} \leftarrow Carry(\neg rA + rB + 1)$
$sum_{0:63} \leftarrow \neg rA + rB + 1$
if OE=1 then do
$OV \leftarrow carry_{32} \oplus carry_{33}$
$SO \leftarrow SO \mid (carry_{32} \oplus carry_{33})$
if Rc=1 then do
$LT \leftarrow sum_{32:63} < 0$
$GT \leftarrow sum_{32:63} > 0$
$EQ \leftarrow sum_{32:63} = 0$
$CR0 \leftarrow LT \mid\mid GT \mid\mid EQ \mid\mid SO$
$rD \leftarrow sum$
$CA \leftarrow carry_{32}$

The sum of the one's complement of the contents of **r**A, the contents of **r**B, and 1 is placed into **r**D.

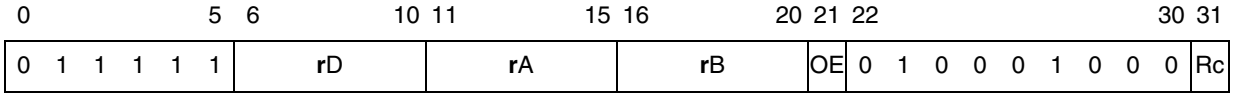
Other registers altered:

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

subfe | Book E | User | **subfe**

Subtract from extended

subfe	**r**D,**r**A,**r**B	(OE=0, Rc=0)
subfe.	**r**D,**r**A,**r**B	(OE=0, Rc=1)
subfeo	**r**D,**r**A,**r**B	(OE=1, Rc=0)
subfeo.	**r**D,**r**A,**r**B	(OE=1, Rc=1)



if E=0 then Cin ← CA
$carry_{0:63}$ ← Carry(¬rA + rB + Cin)
$sum_{0:63}$ ← ¬rA + rB + Cin
if OE=1 then do
OV ← $carry_{32} \oplus carry_{33}$
SO ← SO | ($carry_{32} \oplus carry_{33}$)
if Rc=1 then do
LT ← $sum_{32:63} < 0$
GT ← $sum_{32:63} > 0$
EQ ← $sum_{32:63} = 0$
CR0 ← LT || GT || EQ || SO
rD ← sum
CA ← $carry_{32}$

For **subfe**[**o**][**.**], the sum of the one's complement of the contents of **r**A, the contents of **r**B, and CA is placed into **r**D.

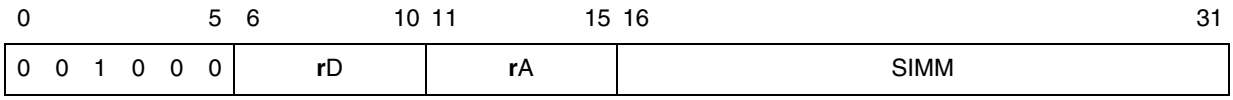
Other registers altered:

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

subfic | Book E | User | **subfic**

Subtract from immediate carrying

subfic **r**D,**r**A,SIMM



$carry_{0:63} \leftarrow Carry(\neg rA + EXTS(SIMM) + 1)$
$sum_{0:63} \leftarrow \neg rA + EXTS(SIMM) + 1$
$rD \leftarrow sum$
$CA \leftarrow carry_{32}$

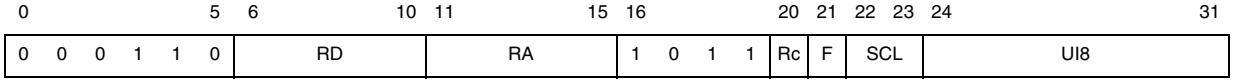
The sum of the one's complement of the contents of **r**A, the sign-extended value of the SIMM field, and 1 is placed into **r**D.

Other registers altered: CA

_subfic*x* | VLE | User | **_subfic*x***

Subtract From Immediate Carrying [and Record]

e_subfic **r**D,**r**A,SCI8 (Rc = 0)
e_subfic. **r**D,**r**A,SCI8 (Rc = 1)



imm ← SCI8(F,SCL,UI8)
$carry_{32:63}$ ← Carry(¬GPR(RA) + imm + 1)

$sum_{32:63}$ ← ¬GPR(RA) + imm + 1

if Rc=1 then do

LT ← $sum_{32:63}$ < 0

GT ← $sum_{32:63}$ > 0

EQ ← $sum_{32:63}$ = 0

CR0 ← LT || GT || EQ || SO

GPR(RD) ← $sum_{32:63}$
CA ← $carry_{32}$

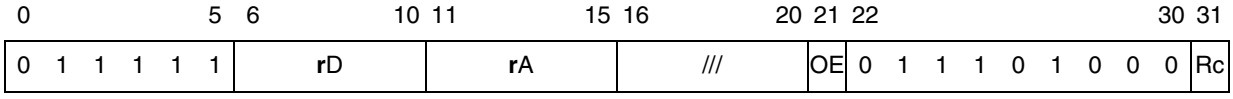
The sum of the one's complement of the contents of GPR(**r**A), the value of SCI8, and 1 is placed into GPR(**r**D).

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

subfme | Book E | User | **subfme**

Subtract from minus one extended

subfme **r**D,**r**A (OE=0, Rc=0)
subfme. **r**D,**r**A (OE=0, Rc=1)
subfmeo **r**D,**r**A (OE=1, Rc=0)
subfmeo. **r**D,**r**A (OE=1, Rc=1)



if E=0 then Cin ← CA
$carry_{0:63}$ ← Carry(¬rA + Cin + 0xFFFF_FFFF_FFFF_FFFF)
$sum_{0:63}$ ← ¬rA + Cin + 0xFFFF_FFFF_FFFF_FFFF
if OE=1 then do
OV ← $carry_{32} \oplus carry_{33}$
SO ← SO | ($carry_{32} \oplus carry_{33}$)
if Rc=1 then do
LT ← $sum_{32:63} < 0$
GT ← $sum_{32:63} > 0$
EQ ← $sum_{32:63} = 0$
CR0 ← LT || GT || EQ || SO
rD ← sum
CA ← $carry_{32}$

For **subfme**[**o**][**.**], the sum of CA, $^{64}1$, and the one's complement of the contents of **r**A is placed into **r**D.

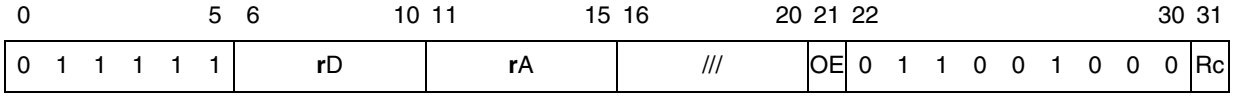
Other registers altered:

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

subfze | Book E | User | **subfze**

Subtract from zero extended

subfze **r**D,**r**A (OE=0, Rc=0)
subfze. **r**D,**r**A (OE=0, Rc=1)
subfzeo **r**D,**r**A (OE=1, Rc=0)
subfzeo. **r**D,**r**A (OE=1, Rc=1)



if E=0 then Cin ← CA
$carry_{0:63}$ ← Carry(¬rA + Cin)
$sum_{0:63}$ ← ¬rA + Cin
if OE=1 then do
OV ← $carry_{32} \oplus carry_{33}$
SO ← SO | ($carry_{32} \oplus carry_{33}$)
if Rc=1 then do
LT ← $sum_{32:63} < 0$
GT ← $sum_{32:63} > 0$
EQ ← $sum_{32:63} = 0$
CR0 ← LT || GT || EQ || SO
rD ← sum
CA ← $carry_{32}$

For **subfze**[**o**][**.**], the sum of the one's complement of the contents of **r**A and CA is placed into **r**D.

Other registers altered:

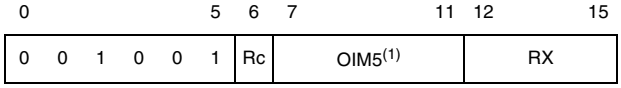
- CA
  CR0 (if Rc=1)
  SO OV (if OE=1)

_subi*x* | VLE | User | **_subi*x***

Subtract Immediate [and Record]

se_subi **r**X,OIMM (Rc = 0)

se_subi. **r**X,OIMM (Rc = 1)



1. OIMM = OIM5 +1

$sum_{32:63} \leftarrow GPR(RX) + \neg(^{27}0 \,||\, OFFSET(OIM5)) + 1$

if Rc=1 then do

$LT \leftarrow sum_{32:63} < 0$

$GT \leftarrow sum_{32:63} > 0$

$EQ \leftarrow sum_{32:63} = 0$

$CR0 \leftarrow LT \,||\, GT \,||\, EQ \,||\, SO$

$GPR(RX) \leftarrow sum_{32:63}$

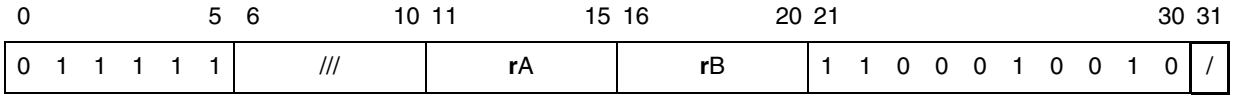
The sum of the contents of GPR(**r**X), the one's complement of the zero-extended value of the offseted OIM5 field (a final value in the range 1–32), and 1 is placed into GPR(**r**X).

Special Registers Altered: CR0 (if Rc = 1)

tlbivax | Book E | Supervisor | **tlbivax**

TLB Invalidate virtual address indexed

tlbivax **r**A,**r**B



if rA=0 then a ← $^{64}0$ else a ← rA
EA ← $^{32}0$ || $(a + rB)_{32:63}$
AS ← implementation-dependent value
ProcessID ← implementation-dependent value
VA ← AS || ProcessID || EA
InvalidateTLB(VA)

EIS note: Executing **tlbivax** invalidates any TLB entry that corresponds to a virtual address calculated by this instruction if IPROT is not set; this includes invalidating TLB entries on other devices as well as on the processor executing **tlbivax**. Thus an invalidate operation is broadcast throughout the coherent domain of the processor executing **tlbivax**. On some implementations, HID1[ABE] must be set to allow management of external L2 caches (for implementations with L2 caches) as well as other L1 caches in the system.

EA calculation:	Addressing Mode	EA for **r**A=0	EA for **r**A≠0
		$^{32}0$ || $\mathbf{r}B_{32:63}$	$^{32}0$ || $(\mathbf{r}A+\mathbf{r}B)_{32:63}$

Address space (AS) is defined as implementation-dependent (for example, it could be MSR[DS] or a bit from an implementation-dependent SPR).

ProcessID is implementation-dependent (for example, it could be from the PID or from an implementation-dependent SPR). The EIS implements the architected PID and additional implementation-specific PIDs. See *Section 2.12.1: Process ID registers (PID0–PIDn)*."

The virtual address (VA) is the value AS || ProcessID || EA.

A TLB entry corresponding to VA is made invalid (that is, removed from the TLB). This instruction causes the target TLB entry to be invalidated in all processors.

The operation performed by this instruction is ordered by **mbar** (or **msync**) with respect to a subsequent **tlbsync** executed by the processor executing **tlbivax**. Operations caused by **tlbivax** and **tlbsync** are ordered by **mbar** as a set of operations independent of the other sets that **mbar** orders.

Other registers altered: None

Programming notes:

- The effects of the invalidation are not guaranteed to be visible to the programming model until the completion of a context synchronizing operation. See *Context synchronization on page 144*."
- Care must be taken not to invalidate TLB entries that contain interrupt vector mappings.

tlbre | Book E | Supervisor | **tlbre**

TLB Read entry

tlbre

0	1	2	3	4	5	6 ... 20	21	22	23	24	25	26	27	28	29	30	31
0	1	1	1	1	1	///(1)	1	1	1	0	1	1	0	0	1	0	/1

1. This field is defined as allocated by the Book E architecture, for possible use in an implementation. These bits are not implemented by the EIS.

The RTL for the EIS definition of **tlbre** is as follows:

```
tlb_entry_id = MAS0(TLBSEL, ESEL | MAS2(EPN)
result = MMU(tlb_entry_id)
MAS0, MAS1, MAS2, MAS3, (and MAS7 if HID0[EN_MAS7_UPDATE] = 1) = result
```

Bits 6–20 of the encoding are allocated for implementation-dependent use and may be used to specify the source TLB entry, the source portion of the source TLB entry, and the target resource into which the result is placed. The EIS makes no use of these bits.

The implementation-defined TLB entry is read, and the implementation-defined portion of the TLB entry is extracted and placed into an implementation-defined target resource.

If the instruction specifies a TLB entry that does not exist, the results are undefined.

EIS implementation note: tlbre causes the contents of a single TLB entry to be extracted from the MMU and be placed in the corresponding fields of the MMU assist (MAS) registers. The entry extracted is specified by the TLBSEL, ESEL and EPN fields of MAS0 and MAS2. The contents extracted from the MMU are placed in MAS0–MAS3.

See the user's manual for the implementation.

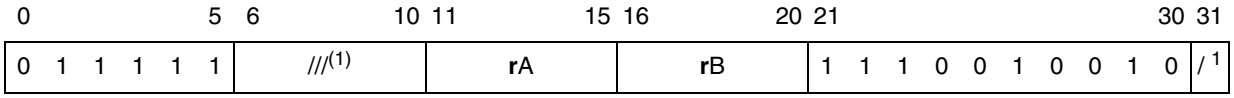
Execution of this instruction is restricted to supervisor mode.

Other registers altered: MAS0, MAS1, MAS2, and MAS3, as defined by the EIS

tlbsx | Book E | Supervisor | **tlbsx**

TLB Search indexed

tlbsx **r**A,**r**B



1. This field is defined as allocated by the Book E architecture, for possible use in an implementation. These bits are not implemented by the EIS.

if RA!=0 then generate exception
EA = $^{32}0$ || GPR(RB)$_{32:63}$
ProcessID = MAS6(SPID)
AS = MAS6(SAS)
VA0 = AS || $^{(MMUCFG[PIDSIZE] + 1)}0$ || EA
VA1 = AS || ProcessID || EA
if Valid_TLB_matching_entry_exists (VA0) or Valid_TLB_matching_entry_exists (VA1)
#

#

MAS0, MAS1, MAS2, MAS3 = result

EA calculation: Addressing ModeEA for **r**A=0EA for **r**A≠0
$^{32}0$ || **r**B$_{32:63}$ $^{32}0$ || (**r**A+**r**B)$_{32:63}$

Note that rA = 0 is a preferred form for **tlbsx** and that some ST implementations take an illegal instruction exception program interrupt if rA != 0.

Virtual address 0 (VA0) is the value AS || $^{(MMUCFG[PIDSIZE] + 1)}0$ || EA
Virtual address 1 (VA1) is the value AS || ProcessID || EA

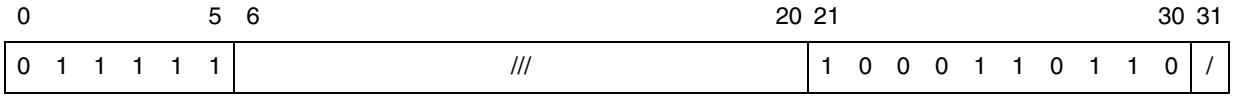
If the TLB contains an entry corresponding to VA, an implementation-dependent value is placed into an implementation-dependent-specified target. Otherwise the contents of the implementation-dependent-specified target are left undefined.

Other registers altered: implementation-dependent. See *Supervisor-level tlb management instructions on page 183.*

tlbsync | Book E | Supervisor | **tlbsync**

TLB Synchronize

tlbsync



tlbsync provides an ordering function for the effects of all **tlbivax** instructions executed by the processor executing **tlbsync**, with respect to the memory barrier created by a subsequent **msync** instruction executed by the same processor. Executing **tlbsync** ensures that all of the following occur.

- All TLB invalidations caused by **tlbivax** instructions preceding the **tlbsync** instruction will have completed on any other processor before any memory accesses associated with data accesses caused by instructions following the **msync** instruction are performed with respect to that processor.
- All memory 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 **msync** instruction, to the extent required by the associated memory-coherence required attributes, before the **mbar** or **msync** instruction's memory barrier is created.

The operation performed by this instruction is ordered by the **mbar** and **msync** instructions 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 that 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.

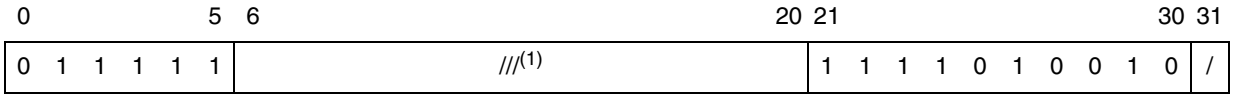
Execution of this instruction is restricted to supervisor mode.

Other registers altered: None

tlbwe | Book E | Supervisor | **tlbwe**

TLB Write entry

tlbwe



1. This field is defined as allocated by the Book E architecture, for possible use in an implementation. These bits are not implemented by the EIS.

Bits 6–20 of the instruction encoding are allocated for implementation-dependent use, 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 EIS does not make use of these bits.

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. See the user's manual for the implementation.

Execution of this instruction is restricted to supervisor mode.

Other registers altered: None

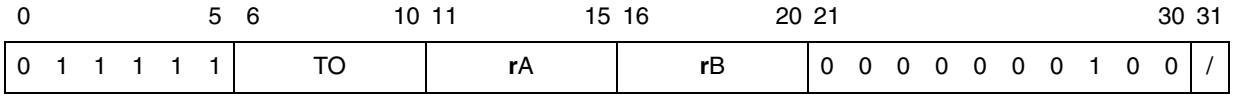
Programming notes:

- The effects of the update are not guaranteed to be visible to the programming model until the completion of a context synchronizing operation. See *Context synchronization on page 144*."
- Care must be taken not to invalidate any TLB entry that contains the mapping for any interrupt vector.

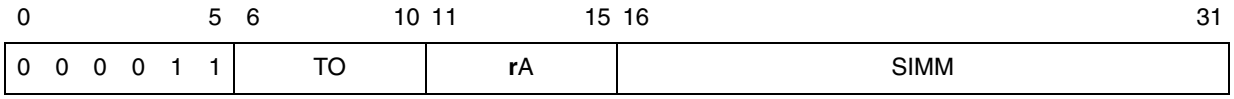
tw | Book E | User | **tw**

Trap word [immediate]

tw TO,**r**A,**r**B



twi TO,**r**A,SIMM



a ← EXTS($rA_{32:63}$)
if 'tw' then b ← EXTS($rB_{32:63}$)
if 'twi' then b ← EXTS(SIMM)
if (a < b) & TO_0 then TRAP
if (a > b) & TO_1 then TRAP
if (a = b) & TO_2 then TRAP
if (a $<_u$ b) & TO_3 then TRAP
if (a $>_u$ b) & TO_4 then TRAP

For **tw**, the contents of **r**A[32–63] are compared with the contents of **r**B[32–63].

For **twi**, the contents of **r**A[32–63] are compared with the sign-extended value of the SIMM field.

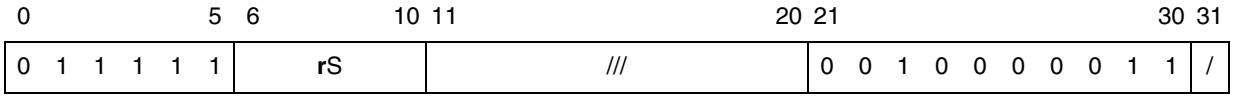
If any bit in the TO field is set and its corresponding condition is met by the result of the comparison, then the system trap handler is invoked.

Other registers altered: None

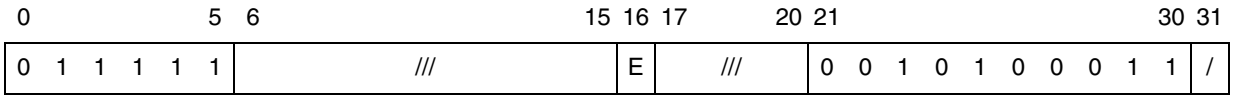
wrtee | Book E | Supervisor | **wrtee**

Write MSR external enable [immediate]

wrtee **r**S



wrteei E



if 'wrtee' then MSR[EE] ← rS_{48}
if 'wrteei' then MSR[EE] ← E

For **wrtee**, **r**S[48] is placed into MSR[EE].

For **wrteei**, the value specified in the E field is placed into MSR[EE].

Execution of this instruction is restricted to supervisor mode.

In addition, changes to MSR[EE] are effective as soon as the instruction completes. Thus if MSR[EE]=0 and an external interrupt is pending, executing a **wrtee** or **wrteei** that sets MSR[EE] causes the external interrupt to be taken before the next instruction is executed, if no higher priority exception exists.

Other registers altered: MSR

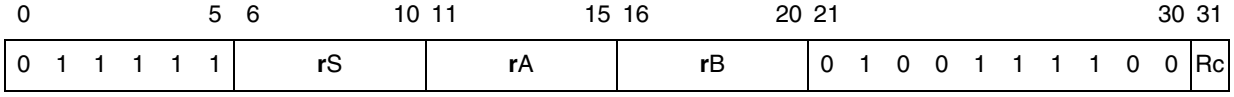
Programming note: **wrtee** and **wrteei** are used to update of MSR[EE] without affecting other MSR bits. Typical usage is as follows:

```
mfmsr          Rn      #save EE in GPR(Rn)48
wrteei         0       #turn off EE
:       :              :
:       :              #code with EE disabled
:       :              :
wrtee          Rn      #restore EE without altering other MSR bits
that may have changed
```

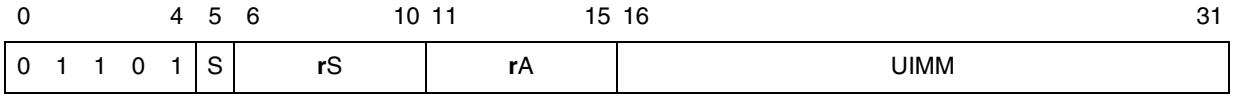
xor | Book E | User | **xor**

XOR [Immediate [shifted]]

xor **r**A,**r**S,**r**B (Rc=0)
xor. **r**A,**r**S,**r**B (Rc=1)



xori **r**A,**r**S,UIMM (S=0, Rc=0)
xoris **r**A,**r**S,UIMM (S=1, Rc=0)



if ‘xori’ then b ← $^{48}0$ || UIMM
if ‘xoris’ then b ← $^{32}0$ || UIMM || $^{16}0$
if ‘xor[.]’ then b ← rB
$result_{0:63}$ ← rS ⊕ b
if Rc=1 then do
LT ← $result_{32:63} < 0$
GT ← $result_{32:63} > 0$
EQ ← $result_{32:63} = 0$
CR0 ← LT || GT || EQ || SO
rA ← result

For **xori**, the contents of **r**S are XORed with $^{48}0$ || UIMM.

For **xoris**, the contents of **r**S are XORed with $^{32}0$ || UIMM || $^{16}0$.

For **xor**[**.**], the contents of **r**S are XORed with the contents of **r**B.

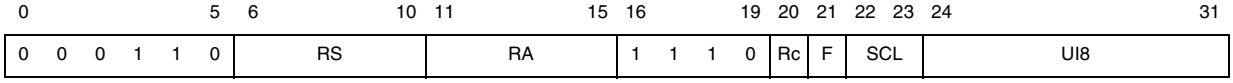
The result is placed into **r**A.

Other registers altered: CR0 (if Rc=1)

_xor*x* | VLE | User | **_xor*x***

XOR [Immediate] [and Record]

e_xori **r**A,**r**S,SCI8 (Rc = 0)
e_xori. **r**A,**r**S,SCI8 (Rc = 1)



if ‘e_xori[.]’ then b ← SCI8(F,SCL,UI8)

$result_{32:63}$ ← GPR(RS) ⊕ b

if Rc=1 then do

LT ← $result_{32:63}$ < 0

GT ← $result_{32:63}$ > 0

EQ ← $result_{32:63}$ = 0

CR0 ← LT || GT || EQ || SO

GPR(RA) ← result

For **e_xori[.]**, the contents of GPR(**r**S) are XORed with SCI8.

The result is placed into GPR(**r**A).

Special Registers Altered: CR0 (if Rc = 1)

Part II: EIS-defined extensions to the Book E architecture

This part describes the extensions defined by the Book E Implementation Standards (EIS). It consists of the following:

- *Chapter 7: Auxiliary processing units (APUs) on page 823*," describes APUs such as the **isel** instruction, performance monitor, signal processing engine (SPE), locking, and machine check APUs.
- *Chapter 8: Storage-related APUs on page 848*," describes the following APUs defined by the storage architecture:
  - *Chapter 8.1: Cache line locking APU on page 848*"
  - *Chapter 8.2: Direct cache flush APU on page 850*"
  - *Chapter 8.3: Cache way partitioning APU on page 851*"
- Subsequent chapters describe the VLE extension
  - *Chapter 9: VLE introduction on page 852*"
  - *Chapter 10: VLE storage addressing on page 759*
  - *Chapter 11: VLE compatibility with the EIS on page 856*"
  - *Chapter 12: VLE instruction classes on page 860*"
  - *Chapter 13: VLE instruction set on page 891*"
  - *Chapter 14: VLE instruction index on page 967*"

7 Auxiliary processing units (APUs)

This chapter describes the APUs defined by the EIS, which are as follows:

- *Chapter 7.1: Integer select APU*"
- *Chapter 7.2: Performance monitor APU*"
- *Chapter 7.3: Signal processing engine APU (SPE APU)*"
- *Chapter 7.4: Embedded vector and scalar single-precision floating-point APUs (SPFP APUs)*"
- *Chapter 7.5: Machine check APU*"
- *Chapter 7.6: Debug APU*"
- *Chapter 7.7: Alternate time base*"

Note that individual processors may implement APUs that are not defined by the EIS. Individual processors may either further extend these APUs or may implement a subset of the resources described here. See the documentation for the individual implementation.

7.1 Integer select APU

Control code, which is characterized by unpredictable short branches, is common in embedded applications. When mispredicted, these branches cause long pipeline delays. The integer select (**isel**) APU consists of a single instruction (**isel**), a conditional register move that helps eliminate some of these branches. The **isel** instruction works as follows:

```
if crB then
        rD = rA
else
        rD = rB
```

The **isel** instruction allows more efficient implementation of a condition sequence such as the one in the following generic example:

```
int16 global1,…, global37,...;
....
void procedure17(int16 parm) {
        if      (global1 == 27) {
                global37 = parm + 17;
        }
        else {
                global37 = parm - 17;
        }
}
```

7.1.1 Integer select APU programming model

The integer select APU includes only the **isel** instruction, described in *Chapter 6: Instruction set on page 330*." It accesses the GPRs and the CR and does not implement additional registers or interrupt resources.

7.1.2 Using isel to Improve conditional branch performance

The Integer Select instruction, **isel**, can be used to handle short conditional branch segments more efficiently. **isel** has two source registers and one destination register. Under the control of a specified condition code bit, it copies one or the other source operand to the destination.

Table 208 shows a coding example with and without the **isel** instruction.

Table 208. Recoding with isel

Code sequence without isel	Code sequence with isel

cmpi cr3, r17, 27;
bne cr3, NotEqual;
addi r15, r17, 17;
jmp Assign;
NotEqual:
addi r15, r17, -17;
Assign:
stw r15, (rGlobals + g37);
	cmpi cr3, r17, 27;
	addi r15, r17, 17;
	addi r16, r17, -17;
	isel r15, r15, r16, cr3.eq
	stw r15, (rGlobals + g37);

The sequence without **isel** turns conditional branches into a code sequence that sets a condition code according to the results of a computation. It uses a conditional branch to choose a target sequence, but needs an unconditional branch for the IF clause. The conditional branch is often hard to predict, the code sequences are generally small, and the resulting throughput is typically low.

The sequence using **isel** does the following:

- Sets a condition code according to the results of a comparison
- Has code that executes both the IF and the ELSE segments
- Has a final statement that copies the results of one of the segments to the desired destination register
- Works well for small code segments and for unpredictable branches
- Can reduce code size

7.2 Performance monitor APU

The EIS defines the performance monitor as an APU. Software communication with the performance monitor APU is achieved through performance monitor registers (PMRs) rather than SPRs. The PMRs are used for enabling conditions that can trigger an APU-defined performance monitor interrupt.

7.2.1 Performance monitor APU programming model

The performance monitor APU provides a set of PMRs for defining, enabling, and counting conditions that trigger the performance interrupt. The APU defines instructions for reading and writing the PMRs.

Performance monitor APU registers

The performance monitor APU defines IVOR35 (SPR 531) for indicating the address of the performance monitor interrupt vector. IVOR35 is described in *Interrupt vector offset registers (IVORs) on page 83*."

The APU also defines a set of PMRs that are separate from the SPR resources. However, like SPRs and as shown in *Table 209* and *Table 210*, bit 5 indicates whether a register is user- or supervisor-accessible. Supervisor-level PMRs in *Table 209* are accessed through the **mtpmr** and **mfpmr** instructions. Attempting to read or write supervisor-level registers while in user-mode causes a privilege exception.

Table 209. Performance monitor registers—supervisor level

Register name	Abbreviation	PMR number	pmr[0–4]	pmr[5–9]
Counter 0	PMC0	16	00000	10000
Counter 1	PMC1	17	00000	10001
Counter 2	PMC2	18	00000	10010
Counter 3	PMC3	19	00000	10011
Local control a0	PMLCa0	144	00100	10000
Local control a1	PMLCa1	145	00100	10001
Local control a2	PMLCa2	146	00100	10010
Local control a3	PMLCa3	147	00100	10011
Local control b0	PMLCb0	272	01000	10000
Local control b1	PMLCb1	273	01000	10001
Local control b2	PMLCb2	274	01000	10010
Local control b3	PMLCb3	275	01000	10011
Global control 0	PMGC0	400	01100	10000

The user-level PMRs in *Table 210* are read-only and are accessed with the **mfpmr** instruction. Attempting to write user-level registers in either supervisor or user mode causes an illegal instruction exception.

Table 210. Performance monitor registers—user level (read-only)

Register name	Abbreviation	PMR number	pmr[0–4]	pmr[5–9]
Counter 0	UPMC0	0	00000	00000
Counter 1	UPMC1	1	00000	00001
Counter 2	UPMC2	2	00000	00010
Counter 3	UPMC3	3	00000	00011
Local control a0	UPMLCa0	128	00100	00000
Local control a1	UPMLCa1	129	00100	00001
Local control a2	UPMLCa2	130	00100	00010
Local control a3	UPMLCa3	131	00100	00011
Local control b0	UPMLCb0	256	01000	00000

Table 210. Performance monitor registers—user level (read-only) (continued)

Register name	Abbreviation	PMR number	pmr[0–4]	pmr[5–9]
Local control b1	UPMLCb1	257	01000	00001
Local control b2	UPMLCb2	258	01000	00010
Local control b3	UPMLCb3	259	01000	00011
Global control 0	UPMGC0	384	01100	00000

PMRs are fully described in *Chapter 2.16 on page 124*."

Performance monitor apu instructions

The APU also defines the instructions in *Table 211* to move to and move from these PMRs. Full descriptions of these instructions can be found in *Chapter 6 on page 330*."

Table 211. Performance monitor apu instructions

Name	Mnemonic	Syntax
Move from Performance Monitor Register	**mfpmr**	**r**D,PMRN
Move to Performance Monitor Register	**mtpmr**	PMRN,**r**S

Performance monitor APU interrupt model

The performance monitor APU provides a performance monitor interrupt that is triggered by an enabled condition or event.

7.3 Signal processing engine APU (SPE APU)

This section describes the SPE APU programming model, exceptions, and functions.

7.3.1 Overview

This section describes the instruction set architecture of the signal processing engine (SPE) APU. The SPE APU is designed to accelerate signal processing applications normally suited to DSP operation. This is accomplished using short (two-element) vectors 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.

7.3.2 Nomenclature and conventions

Several conventions regarding nomenclature are used in this document:

- The signal processing engine APU is abbreviated as SPE.
- All register bit numbering is 64-bit, with bit 0 being the most significant bit. Registers that are only 32-bit define bit 32 as the most significant bit. For both 32- and 64-bit registers, bit 63 is the least significant bit.
- 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 half words. Bits 16 to 31 and bits 48 to 63 are referenced as odd half words.
- Mnemonics for SPE instructions generally begin with the letters ‘ev’ (embedded vector).

7.3.3 Programming model

This section describes SPE registers, instructions, and interrupts.

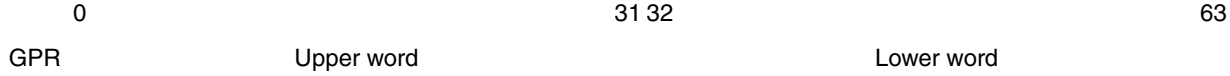
General operation

SPE instructions generally take elements from each 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 (that is, they 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. SPE compare instructions store the compare result into the condition register (CR). The meaning of the CR bits is now overloaded for SPE operations. 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.

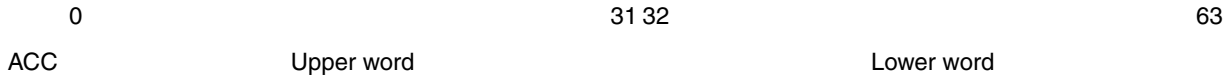
GPR registers

The SPE APU requires a GPR register file with thirty-two 64-bit registers. For 32-bit implementations, PowerPC Book E 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, that is, those instructions continue to operate on the 64-bit registers as they would if the SPE APU was not implemented. SPE APU 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.



Accumulator register

A partially visible accumulator register (ACC) is provided for the integer/fractional multiply accumulate (MAC) forms of instructions. The accumulator is a 64-bit register that holds the results of the multiply accumulate 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.



Signal processing embedded floating-point status and control register (SPEFSCR)

Status and control for SPE uses the SPEFSCR, described in *Chapter 2.14.1: Signal processing, embedded floating-point status, control register (SPEFSCR) on page 119*." The embedded floating-point APUs also use SPEFSCR. Status and control bits are shared for embedded floating-point operations and SPE vector operations. The SPEFSCR is implemented as SPR number 512 and is read and written by the **mfspr** and **mtspr** instructions in both user and supervisor mode.

SPE exception bit in ESR

ESR[SPE] is defined as the SPE exception bit. This bit is set whenever the processor takes an interrupt related to the execution of SPE instructions. (Note that the same bit is used for embedded floating-point APU exceptions. Thus, SPE and embedded floating-point exceptions are indistinguishable in the ESR.)

SPE available bit in MSR

MSR[SPE] is defined as the SPE available bit. If this bit is not set and software attempts to execute an SPE instruction, the SPE APU unavailable interrupt is taken.

Software note: This bit can be used by software to detect when a process uses the upper 32 bits of a 64-bit register on a 32-bit implementation and thus save them on context switch.

Data formats

The SPE APU provides two different data formats, integer and fractional. Both data formats can be treated as signed or unsigned quantities.

Integer format

Integer data format is the same as what is conventionally used in computing.

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 set OV or OVH in 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} set OV or OVH in SPEFSCR.

Fractional format

Fractional data format is the same that 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 decimal 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 set OV or OVH in SPEFSCR. SPE does not contain explicit instructions that manipulate unsigned fractional data. Unsigned integer forms produce the same bit results as unsigned fractional values would; therefore, unsigned fractional instruction forms are not defined for SPE.

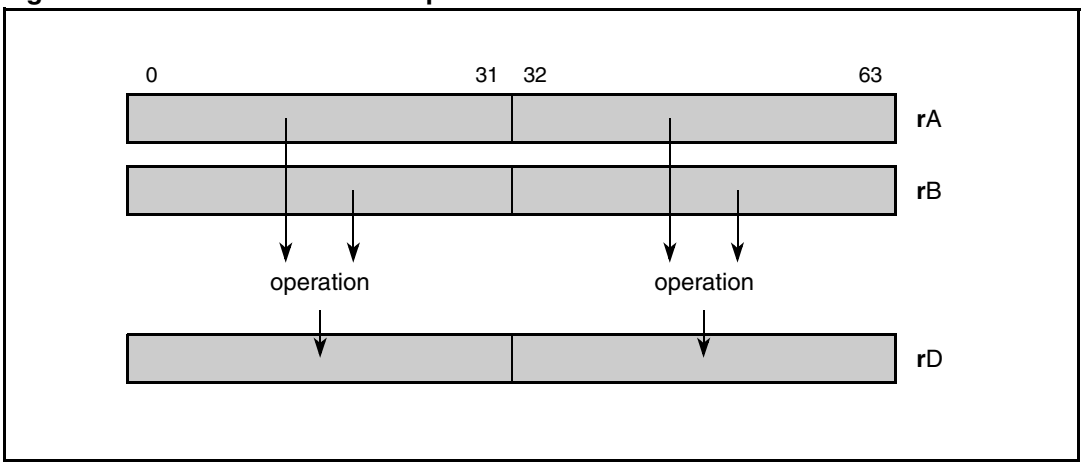
Signed fractions consist of 16-, 32-, or 64-bit binary fractional values in two's-complement form that range from –1 to less than 1. Signed fractions place the decimal point immediately to the right of the most significant bit. The largest representable value is $1 - 2^{-(n-1)}$, where n represents the 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 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 (lsb) of the shifted result.

Computational operations

SPE supports several different computational capabilities. These can be grouped as follows:

- 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. *Figure 178* shows how operations are typically performed in vector operations.

Figure 178. Two-element vector operations



- Multiply and accumulate instructions. These instructions perform multiply operations, add the result to the accumulator and place the result into the destination register and 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 212*.

Table 212. Mnemonic extensions for multiply accumulate instructions

Extension	Meaning	Comments
Multiply form		
he	Half word even	16 X 16 → 32
heg	Half word even guarded	16 X 16 → 32, 64-bit final accum result
ho	Half word odd	16 X 16 → 32
hog	Half word odd guarded	16 X 16 → 32, 64-bit final accum 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	$Accumulator_{0:31}$ + $result_{0:31}$ → $accumulator_{0:31}$ $Accumulator_{32:63}$ + $result_{32:63}$ → $accumulator_{32:63}$
an	Add negated to accumulator	Accumulator – result → accumulator
anw	Add negated to accumulator	$Accumulator_{0:31}$ – $result_{0:31}$ → $accumulator_{0:31}$ $Accumulator_{32:63}$ – $result_{32:63}$ → $accumulator_{32:63}$

- 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.

SPE exceptions and interrupts

The APU defines the following SPE exceptions:

- SPE/embedded floating-point unavailable exception (causes the SPE/embedded floating point unavailable interrupt)
- SPE vector alignment exception (causes the alignment interrupt)

Interrupt vector offset registers (IVORs) IVOR32 (SPE/embedded floating-point unavailable interrupt) and IVOR5 (alignment interrupt) are used by the interrupt model. The SPR number for IVOR32 is 528; IVOR5 is defined by Book E. These registers are privileged.

SPE/Embedded floating point unavailable exception

The SPE/embedded floating point unavailable exception occurs when execution of an SPE instruction (except **brinc**) is attempted and bit 38 (SPE available, MSR[SPE]) is not set. If the SPE/embedded floating point unavailable exception occurs, a SPE/embedded floating point unavailable exception interrupt is taken and the processor suppresses execution of the instruction causing the exception. SRR0, SRR1, MSR, and ESR are modified as follows:

- SRR0 is set to the EA of the instruction causing the interrupt.
- SRR1 is set to the contents of the MSR at the time of the interrupt.
- MSR bits CE, ME, and DE are unchanged. All other bits are cleared.
- ESR[36] bit is set. All other ESR bits are cleared.

Instruction execution resumes at address IVPR[0–47]||IVOR32[48–59]||0b0000.

Software note: This exception is also used by the embedded floating-point APUs in the same manner. It should be used by software to determine if the application is using the upper 32 bits of the GPRs and thus is required to save and restore them on a context switch.

SPE vector alignment exception

The SPE vector alignment exception is taken if the EA of any of the following instructions in not aligned to a 64-bit boundary: **evldd**, **evlddx**, **evldw**, **evldwx**, **evldh**, **evldhx**, **evstdd**, **evstddx**, **evstdw**, **evstdwx**, **evstdh**, or **evstdhx**. When an SPE vector alignment exception occurs, an alignment interrupt is taken and the processor suppresses execution of the instruction causing the exception. SRR0, SRR1, MSR, ESR, and DEAR are modified as follows:

- SRR0 is set to the EA of the instruction causing the interrupt.
- SRR1 is set to the contents of the MSR at the time of the interrupt.
- MSR bits CE, ME, and DE are unchanged. All other bits are cleared.
- ESR[56] bit is set. ESR[ST] is set if the instruction causing the interrupt is a store. All other ESR bits are cleared.
- DEAR is updated with the EA used in the load or the store.

Instruction execution resumes at address IVPR[0–47]||IVOR32[48–59]||0b0000.

Interrupt priorities

The following list shows the priority order in which SPE APU and SPFP APU interrupts are taken (see *Embedded floating-point interrupts on page 837*"):

1. SPE APU unavailable interrupt
2. SPE vector alignment interrupt
3. Embedded floating-point data interrupt
4. Embedded floating-point round interrupt

7.3.4 Instruction definitions

Chapter 6: Instruction set on page 330," gives complete descriptions of SPE and embedded floating-point instructions. *Chapter 6.3.1 on page 336*," provides pseudo RTL for saturation and bit reversal to more accurately describe those functions that are referenced in the instruction pseudo RTL.

7.4 Embedded vector and scalar single-precision floating-point APUs (SPFP APUs)

This section describes the instruction set architecture of the embedded floating-point APUs. The EIS defines the following APUs:

- Embedded vector single-precision floating-point APU
- Embedded scalar single-precision floating-point APU
- Embedded scalar double-precision floating-point APU

Each of these APUs may be implemented independently of the other. In addition, there is a strong relationship with the SPE APU in that each of the embedded floating-point APUs shares a common status register with the SPE.

7.4.1 Nomenclature and conventions

Several conventions regarding nomenclature are used in this document:

- The embedded vector single-precision floating-point APU operations are abbreviated as vector floating-point or vector SPFP.
- The embedded scalar single-precision floating-point APU operations are abbreviated as scalar SPFP.
- The embedded scalar double-precision floating-point APU operations are abbreviated as scalar DPFP.
- Bits 0 to 31 of a 64-bit register are referenced as field 0, upper half, upper word, or high-word element of the register. Bits 32–63 are referred to as field 1, lower half, or lower-word element of the register. Each half is an element of a 64-bit GPR.
- Mnemonics for vector floating-point instructions generally begin with the letters '**evf**' (embedded vector float).
- Mnemonics for single-precision floating-point instructions generally begin with the letters '**efs**' (embedded floating single).
- References to 'floating-point' or 'embedded SPFP' refer to both APUs.

7.4.2 Embedded floating-point APUs programming model

The embedded floating-point APUs use the GPRs as source and destination operands; however, double precision and vector instruction require 64-bit GPRs as described in *Embedded floating-point APUs GPR implementations on page 836*."

Embedded floating-point instructions

The following sections show opcodes for the three embedded floating-point APUs, as follows:

- *Opcodes for embedded vector floating-point instructions on page 833*"
- *Opcodes for embedded scalar single-precision floating-point instructions on page 833*"
- *Opcodes for embedded scalar double-precision floating-point instructions on page 834*"

Opcodes for embedded vector floating-point instructions

Table 213 lists the embedded vector floating-point opcodes.

Table 213. Embedded vector floating-point instruction opcodes

Instruction	Opcode bits					Comments
	0–5	6–10	11–15	16–20	21–31	
evfsabs	4	rD	rA	00000	010 1000 0100	
evfsadd	4	rD	rA	rB	010 1000 0000	
evfscfsf	4	rD	00000	rB	010 1001 0011	
evfscfsi	4	rD	00000	rB	010 1001 0001	
evfscfuf	4	rD	00000	rB	010 1001 0010	
evfscfui	4	rD	00000	rB	010 1001 0000	
evfscmpeq	4	crfD 00	rA	rB	010 1000 1110	
evfscmpgt	4	crfD 00	rA	rB	010 1000 1100	
evfscmplt	4	crfD 00	rA	rB	010 1000 1101	
evfsctsf	4	rD	00000	rB	010 1001 0111	
evfsctsi	4	rD	00000	rB	010 1001 0101	
evfsctsiz	4	rD	00000	rB	010 1001 1010	
evfsctuf	4	rD	00000	rB	010 1001 0110	
evfsctui	4	rD	00000	rB	010 1001 0100	
evfsctuiz	4	rD	00000	rB	010 1001 1000	
evfsdiv	4	rD	rA	rB	010 1000 1001	
evfsmul	4	rD	rA	rB	010 1000 1000	
evfsnabs	4	rD	rA	00000	010 1000 0101	
evfsneg	4	rD	rA	00000	010 1000 0110	
evfssub	4	rD	rA	rB	010 1000 0001	rA - rB
evfststeq	4	crfD 00	rA	rB	010 1001 1110	
evfststgt	4	crfD 00	rA	rB	010 1001 1100	
evfststlt	4	crfD 00	rA	rB	010 1001 1101	

Opcodes for embedded scalar single-precision floating-point instructions

Table 214 lists the embedded scalar single-precision floating-point opcodes.

Table 214. Embedded scalar single-precision floating-point instruction opcodes

Instruction	Opcode bits					Comments
	0–5	6–10	11–15	16–20	21–31	
efsabs	4	rD	rA	00000	010 1100 0100	
efsadd	4	rD	rA	rB	010 1100 0000	
efscfd	4	rD	00000	rB	010 1100 1111	
efscfsf	4	rD	00000	rB	010 1101 0011	
efscfsi	4	rD	00000	rB	010 1101 0001	
efscfuf	4	rD	00000	rB	010 1101 0010	
efscfui	4	rD	00000	rB	010 1101 0000	
efscmpeq	4	crfD 00	rA	rB	010 1100 1110	
efscmpgt	4	crfD 00	rA	rB	010 1100 1100	
efscmplt	4	crfD 00	rA	rB	010 1100 1101	
efsctsf	4	rD	00000	rB	010 1101 0111	
efsctsi	4	rD	00000	rB	010 1101 0101	
efsctsiz	4	rD	00000	rB	010 1101 1010	
efsctuf	4	rD	00000	rB	010 1101 0110	
efsctui	4	rD	00000	rB	010 1101 0100	
efsctuiz	4	rD	00000	rB	010 1101 1000	
efsdiv	4	rD	rA	rB	010 1100 1001	
efsmul	4	rD	rA	rB	010 1100 1000	
efsnabs	4	rD	rA	00000	010 1100 0101	
efsneg	4	rD	rA	00000	010 1100 0110	
efssub	4	rD	rA	rB	010 1100 0001	rA - rB
efststeq	4	crfD 00	rA	rB	010 1101 1110	
efststgt	4	crfD 00	rA	rB	010 1101 1100	
efststlt	4	crfD 00	rA	rB	010 1101 1101	

Opcodes for embedded scalar double-precision floating-point instructions

Table 215 lists the embedded scalar double-precision floating-point opcodes.

Table 215. Embedded scalar double-precision floating-point instruction opcodes

Instruction	Opcode bits					Comments
	0–5	6–10	11–15	16–20	21–31	
efdabs	4	rD	rA	00000	010 1110 0100	
efdadd	4	rD	rA	rB	010 1110 0000	
efdcfs	4	rD	00000	rB	010 1110 1111	

Table 215. Embedded scalar double-precision floating-point instruction opcodes

Instruction	Opcode bits					Comments
	0–5	6–10	11–15	16–20	21–31	
efdcfsf	4	rD	00000	rB	010 1111 0011	
efdcfsi	4	rD	00000	rB	010 1111 0001	
efdcfsid	4	rD	00000	rB	010 1110 0011	64-bit only
efdcfuf	4	rD	00000	rB	010 1111 0010	
efdcfui	4	rD	00000	rB	010 1111 0000	
efdcfuid	4	rD	00000	rB	010 1110 0010	64-bit only
efdcmpeq	4	crfD 00	rA	rB	010 1110 1110	
efdcmpgt	4	crfD 00	rA	rB	010 1110 1100	
efdcmplt	4	crfD 00	rA	rB	010 1110 1101	
efdctsf	4	rD	00000	rB	010 1111 0111	
efdctsi	4	rD	00000	rB	010 1111 0101	
efdctsidz	4	rD	00000	rB	010 1110 1011	64-bit only
efdctsiz	4	rD	00000	rB	010 1111 1010	
efdctuf	4	rD	00000	rB	010 1111 0110	
efdctui	4	rD	00000	rB	010 1111 0100	
efdctuidz	4	rD	00000	rB	010 1110 1010	64-bit only
efdctuiz	4	rD	00000	rB	010 1111 1000	
efddiv	4	rD	rA	rB	010 1110 1001	
efdmul	4	rD	rA	rB	010 1110 1000	
efdnabs	4	rD	rA	00000	010 1110 0101	
efdneg	4	rD	rA	00000	010 1110 0110	
efdsub	4	rD	rA	rB	010 1110 0001	rA - rB
efdtsteq	4	crfD 00	rA	rB	010 1111 1110	
efdtstgt	4	crfD 00	rA	rB	010 1111 1100	
efdtstlt	4	crfD 00	rA	rB	010 1111 1101	

Optional load/store instructions

All embedded floating-point APUs use GPRs to hold and operate on floating-point values. The APUs do not architect load and store instructions to move the data to and from memory, but instead rely on existing instructions in the architecture to perform this function. In the case where either the vector single-precision embedded floating-point APU or the scalar double-precision embedded floating-point APU is implemented on a 32-bit implementation, the GPRs are required to be 64-bits long. Because a 32-bit implementation contains no load or store instructions that operate on 64-bit data, new instructions are required to perform these actions. In this case (and for a 64-bit implementation), an implementation may implement the following load/store instructions from the SPE APU.

For scalar double-precision:

- **evldd**—Vector load doubleword into doubleword
- **evlddx**—Vector load doubleword into doubleword indexed
- **evstdd**—Vector store doubleword of doubleword
- **evstddx**—Vector store doubleword of doubleword
- **evmergehi**—Vector merge high
- **evmergelo**—Vector merge low

For vector single-precision, all of the vector load/store word and doubleword instructions, merge instructions, and word forms of splat instructions may be implemented. Because the vector single-precision embedded floating-point APU uses a significant set of the SPE vector load/store/merge instructions, it is strongly recommended that the SPE APU be present when implementing the vector single-precision embedded floating-point APU.

Floating-point conversion models

Each APU contains floating-point conversion to and from integer and fractional type instructions. The floating-point to and from non–floating-point conversion model pseudo RTL is provided in *Chapter 6.3.2: Embedded floating-point conversion models on page 337*," as a group of functions that is called from the individual instruction pseudo-RTL descriptions included in the instruction descriptions in *Chapter 6: Instruction set on page 330*."

Embedded floating-point registers

The embedded floating-point APUs share register resources with the SPE APU, as described in the following sections.

Embedded floating-point APUs GPR implementations

Embedded floating-point operations are performed in the GPRs of the processor.

The vector floating-point and double-precision floating-point require a GPR register file with thirty-two 64-bit registers. This is consistent with the SPE APU. Thus, these can coexist with the SPE APU.

Single-precision floating-point requires a GPR register file with thirty-two 32-bit or 64-bit registers. When implemented with a 64-bit register file on a 32-bit implementation, single-precision floating-point operations only use and modify bits 32–63 of the GPR. In this case, bits 0–31 of the GPR are left unchanged by a single-precision floating-point operation. For 64-bit implementations, bits 0–31 are undefined after a single-precision floating-point operation.

Floating-point double-precision instructions operate on the entire 64 bits of the GPRs where a floating-point data item consists of 64 bits.

Vector floating-point instructions 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 a single-precision floating-point value. 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. Floating-point compare instructions treat NaNs, Infinity and Denorm as normalized numbers for the comparison calculation when default results are provided.

Signal processing embedded floating-point status and control register (SPEFSCR)

The embedded floating-point APUs use the SPEFSCR, which is described in *Chapter 2.14.1: Signal processing, embedded floating-point status, control register (SPEFSCR) on page 119*." The SPE APU also uses SPEFSCR. Status and control bits are shared for vector floating-point operations, single-precision floating-point operations and SPE vector operations. The SPEFSCR is implemented as SPR number 512 and is read and written by **mfspr** and **mtspr** in both user and supervisor mode. Vector floating-point instructions affect both the high- and low-element floating-point status flags (bits 34–39 and 50–55). Scalar SPFP instructions affect only the low-element flags and leave the high element flags undefined.

Embedded floating-point exception bit—ESR[SPE]

ESR[SPE] is defined as the embedded floating-point exception bit. This bit is set whenever the processor takes an interrupt related to the execution of the embedded floating-point instructions. (Note that the same bit is used for SPE APU exceptions. Thus, SPE and embedded floating-point interrupts are indistinguishable in the ESR.)

Embedded floating-point interrupts

The following sections describe the embedded floating-point APU interrupts:

- *SPE/embedded floating-point unavailable interrupt on page 837*"
- *Embedded floating-point data interrupt on page 837*"
- *Embedded floating-point round interrupt on page 838*"

SPE/embedded floating-point unavailable interrupt

The SPE/embedded floating-point unavailable interrupt vector is used by the embedded scalar double-precision floating-point APU and the embedded vector single-precision floating-point APU. It is not used by the embedded scalar single-precision floating-point APU. The SPE/embedded floating-point unavailable interrupt occurs when an embedded vector floating-point or an embedded scalar double-precision floating-point instruction is executed and bit 38 of the MSR is not set. If the SPE/embedded floating-point unavailable interrupt occurs, the processor suppresses execution of the instruction causing the exception.

The SRR0, SRR1, MSR, and ESR registers are modified as follows:

- SRR0 is set to the EA of the instruction causing the interrupt.
- SRR1 is set to the contents of the MSR at the time of the interrupt.
- MSR bits CE, ME, and DE are unchanged. All other bits are cleared.
- ESR[24] is set. All other ESR bits are cleared.

Instruction execution resumes at address IVPR[0–47]||IVOR32[48–59]||0b0000.

This interrupt is also used by the SPE APU in the same manner. It should be used by software to determine if the application is using the upper 32 bits of the GPRs and thus is required to save and restore them on a context switch.

Embedded floating-point data interrupt

The embedded floating-point data interrupt vector is used for enabled floating-point invalid operation/input error, underflow, overflow, and divide-by-zero exceptions (collectively called floating-point data exceptions). When one of these enabled exceptions occurs, the

processor suppresses execution of the instruction causing the exception. The SRR0, SRR1, MSR, ESR, and SPEFSCR are modified as follows:

- SRR0 is set to the EA of the instruction causing the interrupt.
- SRR1 is set to the contents of the MSR at the time of the interrupt.
- MSR bits CE, ME and DE are unchanged. All other bits are cleared.
- ESR[SPE] is set. All other ESR bits are cleared.
- One or more SPEFSCR status bits are set to indicate the type of exception. The affected bits are FINVH, FINV, FDBZH, FDBZ, FOVFH, FOVF, FUNFH, and FUNF. SPEFSCR[FG,FGH, FX, FXH] are cleared.

Instruction execution resumes at address IVPR[0–47]||IVOR32[48–59]||0b0000.

Embedded floating-point round interrupt

The embedded floating-point round interrupt occurs if no other floating-point data interrupt is taken and one of the following conditions is met:

- SPEFSCR[FINXE] is set and the unrounded result of an operation is not exact
- SPEFSCR[FINXE] is set, an overflow occurs, and overflow exceptions are disabled (FOVF or FOVFH set with FOVFE cleared)
- An underflow occurs and underflow exceptions are disabled (FUNF set with FUNFE cleared)

The embedded floating-point round interrupt does not occur if an enabled embedded floating-point data interrupt occurs.

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 floating-point instruction for which rounding might occur regardless of the value of FINXE unless an embedded floating-point data interrupt also occurs and is taken.

When the 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 are both zero.

The FG and FX bits are provided so that an interrupt handler can round the result as it desires. FG (the guard bit) 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 (the sticky bit) is the value of the OR of all bits to the right of the guard bit (FG) of the destination format mantissa from the infinitely precise intermediate calculation before rounding.

The SRR0, SRR1, MSR, ESR, and SPEFSCR are modified as follows:

- SRR0 is set to the EA of the instruction following the instruction causing the interrupt.
- SRR1 is set to the contents of the MSR at the time of the interrupt.
- MSR bits CE, ME, and DE are unchanged. All other bits are cleared.
- ESR[SPE] is set. All other ESR bits are cleared.
- SPEFSCR FGH, FG, FXH, and FX are set appropriately. SPEFSCR[FINXS] is set.

Instruction execution resumes at address IVPR[0–47]||IVOR32[48–59]||0b0000.

Interrupt priorities

The following list shows the priority order in which SPE and embedded floating-point interrupts are taken (see *Interrupt priorities on page 831*"):

1. SPE/embedded floating-point unavailable interrupt
2. SPE vector alignment interrupt
3. Embedded floating-point data interrupt
4. Embedded floating-point round interrupt

An embedded floating-point data interrupt is taken if either element of a vector or scalar floating-point operation generates an embedded floating-point data exception. An embedded floating-point round interrupt is taken if either element of a vector floating-point operation or a scalar floating-point operation generates an embedded floating-point round exception and no operation (both element for vector floating-point) generates an embedded floating-point data exception.

7.4.3 Embedded floating-point APU operations

This section describes embedded floating-point APU operational modes, data formats, underflow and overflow handling, IEEE 754 compliance, and conversion models.

Operational modes

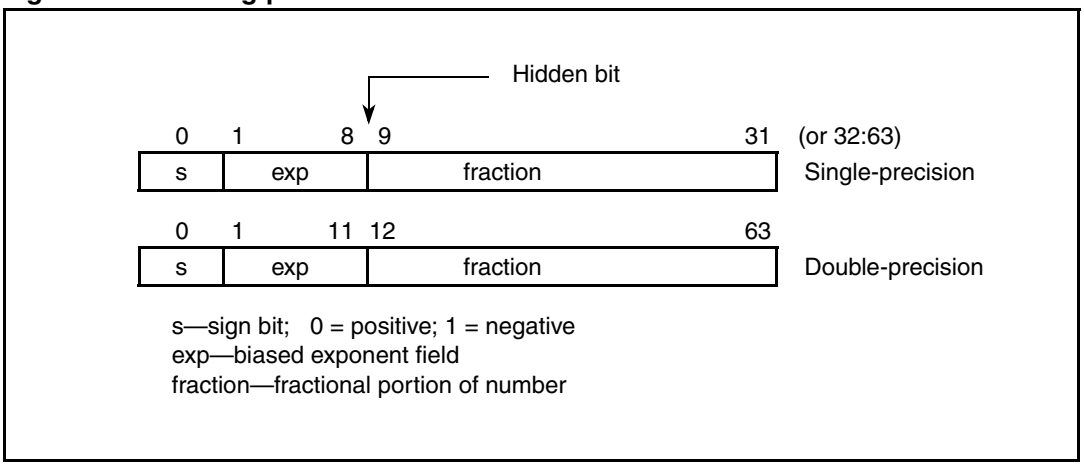
All embedded floating-point operations are governed by the setting of the mode bit in SPEFSCR. The mode bit defines how floating-point results are computed and how floating-point exceptions are handled. Mode 0 defines a real-time, default-results-oriented mode that saturates results. Other modes are currently not defined.

Floating-point data formats

Single-precision floating-point data elements are 32 bits wide with 1 sign bit (s), 8 bits of biased exponent (exp) and 23 bits of fraction.

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 179. Floating-point data formats



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. 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 0x7F7F_FFFF, which is approximately 3.4E+38 (2^{128}), and the minimum positive normalized value (pmin) is represented by the encoding 0x0080_0000, which is approximately 1.2E–38 (2^{-126}).

Two specific values of the biased exponent are reserved (0 and 255 for single-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 zero and a fraction that is zero.

Infinities of both positive and negative sign are represented by a maximum exponent field value (255 for single-precision) and a fraction that is zero.

Denormalized numbers of both positive and negative sign are represented by a biased exponent value of 0 and a non-zero fraction. For these numbers, the hidden bit is defined by the IEEE 754 standard to be zero. 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) and a fraction that is non-zero.

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 of an instruction is such that r > pmax or r < nmax. Additionally, an implementation may also signal overflow by comparing the exponents of the operands. In this case, the hardware examines both exponents ignoring the fractional values. If it is determined that the operation to be performed may overflow (ignoring the fractional values), an overflow may be said to occur. For addition and subtraction this can occur if the larger exponent of both operands is 254. For multiplication this can occur if the sum of the exponents of the operands less the bias is 254. Thus:

single-precision addition:
if A_{exp} >= 254 | B_{exp} >= 254 then overflow

double-precision addition:
if A_{exp} >= 2046 | B_{exp} >= 2046 then overflow

single-precision multiplication:
if $A_{exp} + B_{exp} - 127$ >= 254 then overflow

double-precision multiplication:
if $A_{exp} + B_{exp} - 1023$ >= 2046 then overflow

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. As with overflow detection, an implementation may also signal underflow by comparing the exponents of the operands. In this case, the hardware examines both exponents regardless of the fractional values. If it is determined that the operation to be performed may underflow (ignoring the fractional values), an underflow may be said to occur. For division this can occur if the difference of the exponent of the A operand less the exponent of the B operand less the bias is 1. Thus:

single-precision division:
if $A_{exp} - B_{exp} - 127 <= 1$ then underflow

double-precision multiplication:
if $A_{exp} - B_{exp} - 1023 <= 1$ then underflow

The embedded floating-point APUs will not produce +Inf, –Inf, NaN, or a Denormalized number. If the result of an instruction overflows and floating-point overflow exceptions are disabled (SPEFSCR[FOVFE] is cleared), *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 floating-point underflow exceptions are disabled (SPEFSCR[FUNFE] is cleared), +0 or -0 is generated as the result of that instruction based upon the sign of the result.

IEEE 754 compliance

The embedded floating-point APU implements a floating-point system as defined in 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 a floating-point instruction has data values that are +infinity, –infinity, denorm, or NaN, or when the result of an operation produces an overflow or an underflow, an interrupt may be taken and the interrupt handler is responsible for delivering IEEE 754–compliant behavior if desired.

When floating-point invalid input exceptions are disabled (SPEFSCR[FINVE] is cleared), default results are provided by the hardware when an infinity, denorm, or NaN input is received, or for the operation 0/0. When floating-point underflow exceptions are disabled (SPEFSCR[FUNFE] is cleared) and the result of a floating-point operation underflows, a signed zero result is produced. The inexact exception is also signaled for this condition. When floating-point overflow exceptions are disabled (EFSCR[FOVFE] is cleared) and the result of a floating-point operation overflows, a pmax or nmax result is produced. The inexact exception is also signaled for this condition. An exception enable flag (SPEFSCR[FINXE]) is also provided for generating an interrupt when an inexact result is produced, to allow a software handler to conform to the IEEE 754 standard. A divide-by-zero exception enable flag (SPEFSCR[FDBZE]) is provided for generating an 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 then delivers 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 –infinity. Thus, –0 + –0 = –0 is the only case in which the result is a –0; all other cases that result in a zero value give +0 unless the rounding mode is round to –infinity.

Note that when exceptions are disabled and default results computed, operations having input values that are denormalized may provide different bit-exact 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 on other implementations.

Sticky bit handling for exception conditions

The SPEFSCR defines sticky bits for retaining information about exception conditions that are detected. These sticky bits (FINXS, FINVS, FDBZS, FUNFS, and FOVFS) can be used to help provide IEEE 754 compliance. The sticky bits represent the combined OR of all previous status bits produced from any embedded floating-point operation before the last time software zeroed the sticky bit. Only software can zero a sticky bit; hardware can only set sticky bits.

Not all sticky bits are required to be updated by an implementation. Only the FINXS and FDBZS sticky bits are required to be set by hardware. Thus for FINVS, FUNFS and FOVFS, software is required to perform sticky bit setting unless software knows that a given implementation updates them in hardware. This can be achieved by enabling the appropriate exceptions and performing the sticky bit updating in the software interrupt handler. If an implementation provides sticky bit handling for any sticky bits other than FINXS and FDBZS, it must provide it for all sticky bits.

7.4.4 Implementation options summary

There are several options that may be chosen for a given implementation. This section summarizes all the items that are implementation dependent and should be used to help decide which implementation dependent features are chosen.

- APUs. Each of the APUs can be implemented independently of one another. The vector single-precision floating-point APU should be implemented only if the SPE APU is implemented; however, this is not required.
- Both the vector single-precision floating-point APU and the scalar double-precision floating-point APU allow the optional implementation of 64-bit load and store instructions as well as merge upper and lower instructions from the SPE APU. This allows data to be moved in and out of the upper half of a register for 32-bit implementations with 64-bit registers.
- Overflow and underflow conditions may be signaled by doing exponent evaluation of the operation. If by examining the exponents, an overflow or underflow could occur, the implementation may choose to signal an overflow or underflow. It is recommended that future implementations do not use this estimation and signal overflow or underflow when they actually occur.
- If an operand for a calculation or conversion is denormalized, the implementation may choose to use a same-signed zero value in place of the denormalized operand.
- The rounding modes of +Infinity and -Infinity are not required to handled by an implementation. 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 floating-point instruction for which rounding may occur

regardless of the value of FINXE unless an embedded floating-point data interrupt also occurs and is taken.

- For absolute value, negate, negative absolute value operations, an implementation may choose to either simply perform the sign bit operation ignoring exceptions, or to compute the operation and handle exceptions and saturation where appropriate.
- The FGH and FXH bits of the SPEFSCR are undefined upon the completion of a scalar floating-point operation. An implementation may choose to zero them or leave them unchanged.
- An implementation may choose to only implement sticky bit setting by hardware for FDBZS and FINXS allowing software to manage the other sticky bits. It is recommended that all future implementations implement all sticky bit setting in hardware.
- For 64-bit implementations, the upper 32 bits of the destination register are undefined when the result of a scalar floating-point operation is a 32-bit result. It is recommended that future 64-bit implementations produce 64-bit results for the results of 64-bit conversions to integer values.

7.5 Machine check APU

The machine check APU defines features for the machine check interrupt in addition to those defined by the PowerPC architecture and the Book E version of the PowerPC architecture. The machine check APU includes an enhanced definition of the machine check interrupt type similar to the Book E–defined critical interrupt.

7.5.1 Machine check APU programming model

The APU defines dedicated save and restore SPRs, MSRR0 and MSRR1, so a machine check interrupt does not affect the CSRR0, CSRR1, or ESR registers as defined by the Book E architecture.

The APU also defines a separate Return from Machine Check Interrupt instruction, **rfmci**, that restores context from MSRR0 and MSRR1 when the machine check interrupt handler completes.

Machine check APU register model

The machine check APU defines different register for the machine check interrupt resources than the Book E definition. These are as follows:

- Machine-check save/restore register 0 (MCSRR0)—SPR 570. Holds the instruction where fetching begins after **rfmci** executes, typically at the end of the machine check interrupt handler. See *Machine check save/restore register 0 (MCSRR0) on page 87*."
- Machine-check save/restore register 1 (MCSRR1)—SPR 571. Holds the machine state copied to the MSR when a machine check interrupt occurs. The MCSRR1 value is restored to the MSR when **rfmci** executes, typically at the end of the machine check interrupt handler. See *Machine check save/restore register 1 (MCSRR1) on page 87*."
- Machine check syndrome register (MCSR)—SPR 572. MCSR has fields that identify causes for a machine check interrupt along with an indication of whether the processor can recover from the machine check interrupt. See *Machine check syndrome register (MCSR) on page 88*."

Note, however, that the MSR[ME] bit, defined by the original PowerPC architecture, is also used in Book E and in the machine check APU to enable the machine check interrupt.

Machine check APU instruction model

The Return from Machine Check Interrupt instruction, **rfmci**, is context-synchronizing; it works its way to the final execute stage, updates architected registers, and redirects instruction flow. When **rfmci** executes, data is restored from MCSRR0 and MCSRR1. The **rfi** and **rfci** instructions do not affect MCSRR0 and MCSRR1. This instruction is described in *Chapter 3: Instruction model on page 133*."

Machine check interrupt

The machine check APU is consistent with the machine check exception as defined in Book E with the following differences:

- Machine check is no longer a critical interrupt but uses MCSRR0 and MCSRR1 for saving the return address and the MSR in case the machine check is recoverable.
- The Return from Machine Check Interrupt instruction (**rfmci**) is implemented to support the return to the address saved in MCSRR0.
- The machine check syndrome register, MCSR, is used (instead of ESR) to log the cause of the machine check.

7.6 Debug APU

This section describes the instruction set architecture of software accessible debug related items for Book E Implementations (EIS).

The debug APU defines an additional interrupt class for debug interrupts. This allows the debug features to be used in the software that is providing service for critical class interrupts. This is accomplished by providing specific save and restore registers for debug interrupts and providing a new return from interrupt instruction (return from debug interrupt).

The debug APU reassigns debug interrupts into its own interrupt class, adding a new set of registers used to save the machine context upon the occurrence of a debug interrupt, and adds a new instruction, Return From Debug Interrupt (**rfdi**), to return from a debug interrupt and restore the machine state from the new set of registers. This APU redefines PowerPC Book E debug interrupt behavior.

An implementation may choose to provide the debug APU and also provide a method to disable the debug APU, reverting to using the critical interrupt as defined in Book E. If such a capability is provided, HID0[DAPUEN] should be implemented.

7.6.1 Debug APU programming model

The following sections described the debug APU's extensions to the Book E interrupt, register, and interrupt models.

7.6.2 Debug APU register model

The debug interrupt defines the following registers:

- Debug save/restore register 0 (DSRR0). When a debug interrupt is taken, DSRR0 is set to the current or next instruction address. When **rfdi** is executed, instruction execution continues at the address in DSRR0.
- Debug save/restore register 1 (DSRR1), When a debug interrupt is taken, the contents of the MSR are placed into DSRR1. When **rfdi** is executed, the contents of DSRR1 are placed into the MSR. Bits of DSRR1 that correspond to reserved bits in the MSR are also reserved.

This instruction is fully described in *Chapter 6: Instruction set on page 330*."

The debug APU defines fields in the following Book E–defined registers:

- Debug status register (DBSR). New event fields, described in *Table 216*, have been added to DBSR to record critical interrupt taken events and critical interrupt return events.

Table 216. EIS-defined DBSR field descriptions

Bits	Name	Description
57	CIRPT	Critical interrupt taken debug event. A critical interrupt taken debug event occurs when DBCR0[CIRPT] = 1 and a critical interrupt (any interrupt that uses the critical class, that is, uses CSRR0 and CSRR1) occurs. 0No critical interrupt taken debug event has occurred. 1A critical interrupt taken debug event occurred.
58	CRET	Critical interrupt return debug event. A critical interrupt return debug event occurs when DBCR0[CRET] = 1 and a return from critical interrupt (an **rfci** instruction is executed) occurs. 0No critical interrupt return debug event has occurred. 1A critical interrupt return debug event occurred.

- The debug control register 0 (DBCR0), The debug APU adds event enable bits to DBCR0, described in *Table 217*, to control critical interrupt taken events, and critical interrupt return events.

Table 217. DBCR0 field descriptions

Bits	Name	Description
57	CIRPT	Critical interrupt taken debug event. A critical interrupt taken debug event occurs when DBCR0[CIRPT] = 1 and a critical interrupt (any interrupt that uses the critical class, that is, uses CSRR0 and CSRR1) occurs. 0 Critical interrupt taken debug events are disabled. 1 Critical interrupt taken debug events are enabled.
58	CRET	Critical interrupt return debug event. 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.

7.6.3 Debug APU instruction model

The debug APU defines the supervisor-level **rfdi** instruction to restore state after a debug interrupt. 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. 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 (that is, the address in DSRR0 at the time of the execution of the **rfdi**). This instruction is fully described in *Chapter 6*."

Debug APU interrupt model

A debug interrupt occurs when no higher priority exception exists, a debug exception is presented to the interrupt mechanism, and MSR[DE] = 1. The specific cause or causes of debug exceptions are unchanged from Book E.

DSRR0, DSRR1, MSR, debug address register, and debug status register are updated as follows:

Debug save/restore register 0 (DSRR0) is set to an instruction address. DSRR0 is set to the EA of an instruction that was executing or just completed execution when the debug exception occurred. DSRR0 is set the same as CSRR0 is defined to be set in Book E on a debug interrupt. CSRR0 is not changed as the result of a debug interrupt.

Debug save/restore register 1 (DSRR1) is set to the contents of the MSR at the time of the interrupt. CSRR1 is not changed as the result of a debug interrupt.

MSR[CM] is set to the value of MSR[ICM]. MSR[ICM] and MSR[ME] are unchanged and all other defined MSR bits are cleared.

The DBSR and the debug control registers (DBCR0–DBCR2) operate as described in Book E with the addition of a critical interrupt taken debug event and a critical return debug event.

Instruction execution resumes at address IVPR[0–47]||IVOR15[48–59]||0b0000.

7.7 Alternate time base

The alternate time base APU defines a time base counter similar to the time base defined in the PowerPC architecture. It is intended to be used for measuring time in implementation defined intervals. It differs from the time base defined by the PowerPC architecture in that it is not writable and always counts up, wrapping when the 64-bit count overflows.

7.7.1 Programming model

The alternate time base is simply a 64-bit counter that counts up at some implementation dependent rate. Although not required, it is recommended that the rate be at the core clock frequency or as small a multiple of the frequency as practical by the implementation. Consult the user documentation for devices that support this feature.

The counter can be read by executing an **mfspr** instruction specifying the ATB (or ATBL) register, but cannot be written. In 32-bit mode, reading the ATB (or ATBL) register will place the lower 32 bits of the counter into the target register. In 64-bit mode all 64 bits of the counter are placed in the target register. 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 regardless of computation mode.

The alternate time base is analogous to the time base in the PowerPC architecture except that it counts at a different frequency and is not writable.

The effect of power savings mode or core frequency changes on counting in the alternate time base is implementation dependent. See the user document for details.

Implementation Note: An implementation may choose to directly alias the alternate time base to the time base counter if the granularity of time base counting is acceptable.

Registers

The programming model consists of two SPRs, alternate time base lower and upper (ATBL and ATBU).

Alternate time base registers (ATBL and ATBU)

The ATBL and ATBU registers are described in *Chapter 2.15: Alternate time base registers (ATBL and ATBU) on page 123*." The alternate time base counter (ATB) is formed by concatenating the upper and lower alternate time base registers (ATBU and ATBL). ATBL (SPR 526) provides read-only access to the 64-bit alternate time base counter, which is incremented at an implementation-defined frequency. ATB registers are accessible in both user and supervisor mode.

Like the TB implementation, the ATBL register is an aliased name for ATB.

8 Storage-related APUs

This chapter describes the following APUs that are defined as part of the EIS storage architecture:

- *Chapter 8.1: Cache line locking APU*"
- *Chapter 8.2: Direct cache flush APU*"
- *Chapter 8.3: Cache way partitioning APU*"

8.1 Cache line locking APU

The cache line locking APU defines instructions and methods for locking frequently used instructions and data into their cache lines. Cache locking allows software to mark individual cache lines (blocks) as locked, instructing the cache to keep latency-sensitive data available for fast access.

Unlike normal cache lines, locked cache lines do not participate in the normal replacement policy.

8.1.1 Programming model

This section gives a general description of the instructions defined by the cache line locking APU. Full descriptions are provided in *Chapter 6: Instruction set on page 330*."

Lock setting and clearing

Lines are locked into the cache by software using a series of touch and lock set 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 **r**A and **r**B operands to these instructions form a effective address identifying the line to be locked. The CT field indicates which cache in the cache hierarchy should be targeted. These instructions are similar to the **dcbt**, **dcbtst**, and **icbt** instructions, but locking instructions can 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, lines are unlocked from the cache by software using a series of lock-clear instructions. The following instructions are provided to lock instructions into the instruction cache:

- **dcblc**—Data Cache Block Lock Clear
- **icblc**—Instruction Cache Block Lock Clear

The **r**A and **r**B operands to these instructions form an EA identifying the line to be unlocked. The CT field indicates which cache in the cache hierarchy should be targeted.

Additionally, software may clear all the locks in the cache. For the primary cache, this is accomplished by setting the CLFC (DCLFC, ICLFC) bit in L1CSR0 (L1CSR1).

Cache lines can also be implicitly unlocked in the following ways:

- A locked line is invalidated if it is targeted by a **dcbi**, **dcbf**, or **icbi** instruction.
- A snoop hit on a locked line that requires the line to be invalidated. This can occur because the data the line contains has been modified external to the processor, or another processor has explicitly invalidated the line.
- The entire cache containing the locked line is flash invalidated.

An implementation is not required to unlock lines if data is invalidated in the cache. Although the data may be invalidated (and thus not in the cache), the line can remain locked and be filled from the memory subsystem when the next access occurs. This method of not clearing locks when the associated line is invalidated, is called persistent locking. An implementation may choose to implement locks as persistent or not persistent; the preferred method is persistent.

Error conditions

Setting locks in the cache can fail for several reasons. An address specified with a lock set instruction that does not have the proper permission causes a data storage interrupt (DSI). Cache locking addresses are always translated as data references, therefore **icbtls** instructions that fail to translate or fail permissions cause DTLB and DSI errors respectively. Additionally, cache locking and clearing operations can fail due to restricted user mode access. See *Cache locking (user mode) exceptions on page 850*."

Overlocking

If no exceptions occur for the execution of an **dcbtls**, **dcbtstls**, or **icbtls** instruction an attempt is made to lock the corresponding line in the cache. If all of the available ways are already locked in the given cache set, the requested line is not locked. This is considered an overlocking situation and if the lock was targeted for the primary cache (CT = 0) then L1CSR0[DCLO] (or L1CSR1[ICLO] if **icbtls**) is set appropriately.

A processor may optionally allow victimizing a locked line in an overlocking situation. If L1CSR0[DCLOA] (L1CSR0[ICLOA] for the primary instruction cache,) is set, an overlocking condition causes the replacement of an existing locked line with the requested line. The selection of the line to replace in an overlocking situation is implementation dependent. The overlocking condition is still said to exist and is appropriatly reflected in the status bits for lock overflow.

An attempt to lock a line that is present and valid in the cache does not cause an overlocking condition.

A non–lock-setting cache-line fill or line replacement request to a cache that has all ways locked for a given set does not cause a lock to be cleared.

Unable-to-lock conditions

If no exceptions occur and no overlocking condition exists, an attempt to set a lock can fail if any of the following is true:

- The target address is marked cache-inhibited or the storage attributes of the address uses a coherency protocol that does not support locking.
- The target cache is disabled or not present.
- The CT field specifies a value not supported by the implementation.
- Any other implementation-specific error condition.

If an unable-to-lock condition occurs, the lock set instruction is treated as a NOP. If the lock targeted the data cache (**dcbtls**, **dcbtstls**), L1CSR0[DCUL] is set to indicate the unable-to-lock condition; if the lock targeted the instruction cache (**icbtls**), L1CSR1[ICUL] is set. L1CSR0[DCUL] or L1CSR0[ICUL] is set regardless of the CT value in the lock-setting instruction.

Cache locking (user mode) exceptions

Setting and clearing cache locks can be restricted to supervisor mode only access. If set, MSR[UCLE] allows cache locking operations to be performed in user mode. If MSR[UCLE] = 0 and MSR[PR] = 1 and execution of a cache lock or cache clear instruction occurs, a cache locking exception occurs. In this case the processor suppresses execution of the instruction causing the exception. A DSI interrupt is taken and SRR0, SRR1, MSR, and ESR are modified as follows:

- SRR0 is set to the EA of the instruction causing the interrupt.
- SRR1 is set to the contents of the MSR at the time of the interrupt.
- MSR[CE,ME,DE] are unchanged. All other bits are cleared.
- ESR[DLK] is set if the instruction was a **dcbtls**, **dcbtstls**, or a **dcblc**.
- ESR[ILK] is set if the instruction was a **icbtls** or a **icblc**.
- All other ESR bits are cleared.

Instruction execution resumes at address IVPR[0–47]||IVOR2[48–59]||0b0000.

8.2 Direct cache flush APU

8.2.1 Overview

To assist in software flush of the L1 cache, the direct cache flush APU allows the programmer to flush and/or invalidate the cache by specifying the cache set and cache way. Without such a feature, the programmer must either:

- Know the virtual addresses of the lines that need to be flushed and issue **dcbst** or dcbf instructions to those addresses.
- Flush the entire cache by causing all the lines to be replaced. This requires a virtual address range that is mapped as a contiguous physical address range, that the programmer knows and can manipulate the replacement policy of the cache, and the size and organization of the cache.

With the direct cache flush APU the program needs only specify the way and set of the cache to flush.

The direct cache flush APU available bit, L1CFG0[CFISWA], is set for implementations that contain the direct cache flush APU.

8.2.2 Programming model

To address a specific physical block of the cache, the L1 flush and invalidate control register 0 (L1FINV0) is written with the cache set (L1FINV0[CSET]) and cache way (L1FINV0[CWAY]) of the line that is to be flushed. L1FINV0 is written using a **mtspr** instruction specifying the L1FINV0 register. No tag match in the cache is required. An additional field, L1FINV0[CCMD], is used to specify the type of flush to be performed on the line addressed by L1FINV0[CWAY] and L1FINV0[CSET].

The available L1FINV0[CCMD] encodings are described in *Table 33 on page 96*.

Only the L1 data cache (or unified cache) is manipulated by the direct cache flush APU. The L1 instruction cache or any other caches in the cache hierarchy are not explicitly targeted by this APU.

Register model

The direct cache flush APU defined one register, the L1 flush and invalidate control register 0, described in *Chapter 2.11.5 on page 96*." L1FINV0 contains fields to provide the way and set selection of a cache line to flush and or invalidate.

8.3 Cache way partitioning APU

The cache way partitioning APU allows ways in a unified L1 cache to be configured to accept either data or instruction miss line-fill replacements.

8.3.1 Programming model

The cache way partitioning APU is comprised of bits in L1CSR0 and L1CFG0, as follows:

- Way instruction disable field (L1CSR0[WID]) is a 4-bit field that that determines which of ways 0–3 are available for replacement by instruction miss line refills.
- The additional ways instruction disable bit (L1CSR0[AWID]) determines whether ways 4 and above are available for replacement by instruction miss line refills.
- Way data disable field (L1CSR0[WDD]) is a 4-bit field that that determines which of ways 0–3 are available for replacement by data miss line refills.
- The additional ways data disable bit (L1CSR0[AWDD]) determines whether ways 4 and above are available for replacement by instruction miss line refills.
- See *Chapter 2.11.1: L1 cache control and status register 0 (L1CSR0) on page 90*."
- Way access mode bit, L1CSR0[WAM], Determines whether all ways are available for access or only ways partitioned for the specific type of access are used for a fetch or read operation. See *Chapter 2.11.1 on page 90*."
- Cache way partitioning APU available bit, L1CFG0[CWPA], indicates whether the cache way partitioning APU is available. See *Chapter 2.11.3 on page 94*."

These fields are described in detail in *Chapter 2.11.3 on page 94*," and in *Chapter 2.11.1: L1 cache control and status register 0 (L1CSR0) on page 90*."

8.3.2 Interaction with the cache locking APU

Note that the cache way partitioning APU can affect the cache line locking APU's ability to control replacement of lines. If any cache line locking instruction (**icbtls**, **dcbtls**, **dcbtstls**) is allowed to execute and finds a matching line in the cache, the line's lock bit is set regardless of the L1CSR0[WID,AWID,WDD,AWDD] settings. In this case, no replacement has been made.

However, for cache misses that occur while executing a cache line lock set instruction, the only candidate lines available for locking are those that correspond to ways of the cache that have not been disabled for the particular type of line locking instruction (controlled by WDD and AWDD for **dcbtls** and **dcbtstls**, controlled by WID and AWID for **icbtls**). Thus, an overlocking condition may result even though fewer than eight lines with the same index are locked.y

9 VLE introduction

This body of this document describes the VLE (variable length encoding) extension to the Book E architecture. The VLE extension 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. The intent of the VLE extension is not to define an entirely different ISA nor to supplant the PowerPC ISA; instead the VLE extension can be viewed as a supplement that is can be applied to an application or to part of an application to improve code density.

Chapter 11: VLE compatibility with the EIS on page 856," describes additional VLE extensions to the EIS.

The major objectives of the VLE extension are as follows:

- Coexistence and consistency with the Book E ISA and general architecture
- Maintain a common programming model and instruction operation model in the VLE extension
- Reduce overall code size by ~30% over existing PowerPC text segments
- Limit the increase in execution path length to under 10% for most important applications
- Limit the increase in hardware complexity for implementations containing the VLE extension

9.1 Compatibility with PowerPC Book E

VLE provides an extension to Book E. There are additional operations defined using an alternate instruction encoding to enable reduced code footprint. This alternate encoding set is selected on an instruction page basis. A single page attribute bit selects between standard Book E instruction encodings and VLE instructions for that page of memory. This attribute is an extension to the Book E page attributes. Pages can be freely intermixed, allowing for a mixture of both types of encodings.

Instruction encodings in pages marked as using the VLE extension 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 programmer's model uses the same register set with both instruction encodings, although certain registers are not accessible by VLE instructions using the 16-bit formats and not all condition register (CR) fields are used by condition setting or conditional branch instructions 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 instructions.

The VLE extension defines additional fields in registers defined by Book E and the EIS. These are described in *Chapter 11.2: VLE extension processor and storage control extensions on page 856*."

Other than the requirement of big-endian byte ordering for instruction pages and the additional page attribute to identify whether the instruction page corresponds to a VLE section of code, VLE complies with the memory model defined in Book E and the Book E Implementation Specifications (EIS). Likewise, the VLE extension complies with the Book E

and EIS definitions of the exception and interrupt model, the timer facilities, the debug facilities and the special-purpose registers (SPRs).

9.2 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 Book E instruction semantics. Where the semantics, side-effects, and binary encodings are identical, Book E mnemonics and formats are used. Where the semantics are similar but the binary encodings differ, the Book E mnemonic is typically preceded with an **e**_. To distinguish similar instructions available in both 16- and 32-bit forms under VLE and standard Book E instructions, VLE instructions encoded with 16 bits have an **se**_ prefix. Those VLE instructions encoded with 32 bits that have different binary encodings or semantics than the equivalent Book E instruction have an **e**_ prefix. The following are examples:

```
stw rS,D(rA)          // standard Book E instruction
e_stw rS,D(rA)        // 32-bit VLE instruction
se_stw rZ,SD4(rX)     // 16-bit VLE instruction
```

10 VLE storage addressing

A program references memory using the effective address (EA) computed by the processor when it executes a branch, storage access, storage control, or TLB management instruction, or when it fetches the next sequential instruction.

10.1 Data memory addressing modes

Table 218 lists data memory addressing modes supported by the VLE extension.

Table 218. Data storage addressing modes

Mode	Name	Description
Base+16-bit displacement (32-bit instruction format)	D-mode	The 16-bit D field is sign-extended and added to the contents of the GPR designated by **r**A or to zero if **r**A = 0 to produce the EA.
Base+8-bit displacement (32-bit instruction format)	D8-mode	The 8-bit D8 field is sign-extended and added to the contents of the GPR designated by **r**A or to zero if **r**A = 0 to produce the EA.
Base+scaled 4-bit displacement (16-bit instruction format)	SD4-mode	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 **r**X to produce the EA. (Note that **r**X = 0 is not a special case).
Base+Index (32-bit instruction format)	X-mode	The GPR contents designated by **r**B are added to the GPR contents designated by **r**A or to zero if **r**A = 0 to produce the EA.

10.2 Instruction memory addressing modes

Table 219 lists instruction memory addressing modes supported by the VLE extension.

Table 219. Instruction storage addressing modes

Mode	Description
I-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.
All 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_FFFC [0xFFFF_FFFE], the address of the next sequential instruction is undefined.

Table 219. Instruction storage addressing modes (continued)

Mode	Description
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_FFFC, 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_FFFE, the result placed into the LR is undefined.

11 VLE compatibility with the EIS

The body of this document addresses the relationship between VLE and Book E. It does not explicitly address EIS-defined features, such as the APUs or the use of MAS registers. However, the information in the previous chapters provides a model for how the VLE extension is integrated with features defined by the layer of architecture defined by the EIS.

11.1 Overview

The VLE extension uses the same semantics as the Book E architecture. Due to the limited instruction encoding formats, VLE instructions typically support reduced immediate fields and displacements, and not all Book E operations are encoded in the VLE extension. 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.

Book E floating-point registers (FPRs) are not accessible by VLE instructions. VLE instructions use Book E GPR and SPR registers 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 instructions using the 16-bit formats are limited to addressing CR0
- VLE instructions using the 32-bit formats are limited to addressing CR0–CR3

VLE instruction encodings are generally different than Book E instructions, except that most Book E instructions falling within Book E major opcode 31 are encoded identically in 32-bit VLE instructions and have identical semantics unless they affect or access a resource not supported by the VLE extension. Also, major opcode 4 is available to support additional APUs using identical encodings for both Book E and the VLE extension. This allows an implementation of the VLE extension to include additional APUs, such as the cache-line locking, single-precision floating-point, and SPE APUs, and to use the exact encodings.

Because future compatibility is desired, and to avoid confusion with Book E, register bit numbering remains the same as in Book E.

11.2 VLE extension processor and storage control extensions

This section describes additional functionality and extensions to the EIS to support the VLE extension.

11.2.1 EIS instruction extensions

This section describes extensions to EIS instructions to support VLE operations. Because instructions may reside on a half-word boundary, bit 62 is not masked by instructions that cause fetching from a register, such as the LR, CTR, or a save/restore register 0, that holds an instruction address:

- Return from interrupt instructions, such as **rfdi** (defined as part of the debug APU) and **rfmci** (defined as part of the machine check APU) no longer mask bit 62 of the respective save/restore register 0. The destination address is *x*SRR0[32–62] || 1’b0.

11.2.2 Book E instruction extensions

This section describes the various extensions to Book E instructions to support the VLE extension:

- **rfci**, **rfdi**, and **rfi** no longer mask bit 62 of CSRR0, DSRR0, or SRR0. The destination address is *x*SRR0[32–62] || 1'b0.
- **bclr**, **bclrl**, **bcctr**, and **bcctrl** no longer mask bit 62 of the LR or CTR. The destination address is [LR,CTR][32–62] || 1'b0.

11.2.3 EIS MMU extensions

The VLE assumes that the MMU implementation complies with the more general MMU definition provided by Book E and the more specific definition provided by the EIS. This section describes the differences and extensions to the MMU necessary to support the VLE extension.

TLB entries

Each TLB entry is augmented with an additional page attribute bit, the VLE bit. If set, VLE indicates the corresponding page of memory is a VLE page.

TLB load on reset

During reset, all TLB entries except entry 0 are invalidated. TLB entry 0 is loaded with the additional value shown in *Table 220*.

Table 220. TLB Entry 0 reset value

Field	Reset value	Comments
VLE	p_rst_vlemode value	Book E mode, not VLE if no *p_rst_vlemode* signal is available

Note that implementations may provide a *p_rst_vlemode* input to supply the value of the VLE field on reset. If not available, the default value should be 0, indicating a Book E page

VLE attribute bit

If set, the VLE attribute bit indicates the corresponding page of memory is a VLE page. The VLE attribute is used only for instruction access and is ignored for data accesses. The VLE bit may be set only for big-endian pages, otherwise a byte-ordering exception occurs on instruction fetches.

MMU assist registers (MAS*n*)

To support the VLE extension, additional bits are defined in MAS2 and MAS4. These are described in the following sections.

MAS2

The MAS2 register is shown below. The VLE page attribute has been added as MAS2[58]. If the VLE extension is not present, this bit is always read as zero and writes are ignored.

626 SPR Access: User read/write

	32 33 34 35	36 37 38 39	40 41 42 43	44 45 46 47	48 49 50 51	52 53 54 55	56 57	58	59	60	61	62	63
R W	*EPN*					—		VLE	*W*	*I*	*M*	*G*	*E*
Reset	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0	0	0	0	0	0	0

MAS2[VLE] is defined in *Table 221*.

Table 221. MAS2 field descriptions

Bits	Name	Comments, or function when set
58	VLE	VLE 0This page is a standard Book E page 1This page is a VLE page

MMU assist register 4 (MAS4)

When the VLE extension is implemented, MAS4[58] is defined as the VLED field, which contains the default MAS2[VLE] value. If the VLE extension is not present, this bit is always read as zero and writes are ignored. MAS4 is shown below.

628 SPR Access: user read/write

	32 33	34 35	36	45 46 47	48 51	52 55	56 57	58	59	60	61	62	63
R W	—	*TLBSELD*	—	*TIDSELD*	—	*TSIZED*	—	VLED	*WD*	*ID*	*MD*	*GD*	*ED*
Reset	All zeros												

MAS4[VLED]is described in *Table 222*.

Table 222. MAS4 field descriptions

Bits	Name	Comments, or function when set
58	VLED	Default VLE value. Defined by the EIS. 0 This page is a standard Book E page 1 This page is a VLE page

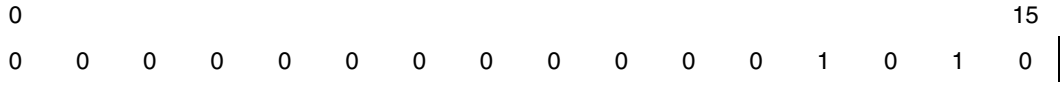
11.2.4 EIS debug APU extensions

The **se_rfdi** instruction is provided to support the EIS debug interrupt APU.

rfdi **rfdi**

Return from debug interrupt

se_rfdi



MSR ← DSRR1

NIA ← $DSRR0_{32:62}$ || 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[32–62]||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 SRR0 or CSRR0 by the interrupt processing mechanism (see Book 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**).

Execution of this instruction is privileged and restricted to supervisor mode only.

Execution of this instruction is context synchronizing.

When the debug APU is disabled, this instruction is treated as an illegal instruction.

Special Registers Altered: MSR

12 VLE instruction classes

This chapter lists instructions defined or supported by the VLE extension. Unless otherwise noted, instructions that are not prefixed with **e_** or **se_** have identical encodings and semantics as in Book E or in the Book E implementation standards (EIS). Full descriptions of these instructions are provided in the EREF: Programmers reference manual for ST's Book E processors.

A complete list of supported instructions is provided in *Chapter 12.6*."

12.1 Processor control instructions

This section lists processor control instructions that can be executed when a processor is in VLE mode. These instructions are grouped as follows:

- *Chapter 12.1.1: System linkage instructions on page 860*"
- *Chapter 12.1.2: Processor control register manipulation instructions on page 860*"
- *Chapter 12.1.3: Instruction synchronization instruction on page 861*"

12.1.1 System linkage instructions

se_sc, **se_rfi**, **se_rfci**, and **se_rfdi** are system linkage instructions that enable the program to call upon the system to perform a service (that is, invoke a system call interrupt), and by which the system can return from performing a service or from processing an interrupt. *Table 223* lists system linkage instructions.

Table 223. System linkage instruction set index

Mnemonic	Instruction	Reference
se_sc	System Call	*Page -954*
se_rfci`	Return From Critical Interrupt	*Page -949*
se_rfdi	Return From Debug Interrupt	*Page -859*
se_rfi	Return From Interrupt	*Page -950*

12.1.2 Processor control register manipulation instructions

In addition to the Book E processor control register manipulation instructions, the VLE extension provides 16-bit forms of instructions to move to/from the LR and CTR. *Table 224* lists the processor control register manipulation instructions.

Table 224. System register manipulation instruction set index

Mnemonic	Instruction	Reference
se_mfctr rX	Move From Count Register	*Page -938*
mfdcr rD,DCRN	Move From Device Control Register	*Book E*
se_mflr rX	Move From Link Register	*Page -939*
mfmsr rD	Move From Machine State Register	*Book E*

Table 224. System register manipulation instruction set index (continued)

Mnemonic	Instruction	Reference
mfspr rD,SPRN	Move From Special Purpose Register	*Book E*
se_mtctr rX	Move To Count Register	*Page -942*
mtdcr DCRN,rS	Move To Device Control Register	*Book E*
se_mtlr rX	Move To Link Register	*Page -943*
mtmsr rS	Move To Machine State Register	*Book E*
mtspr SPRN,rS	Move To Special Purpose Register	*Book E*
wrtee rA	Write MSR External Enable	*Book E*
wrteei E	Write MSR External Enable Immediate	*Book E*

12.1.3 Instruction synchronization instruction

Table 225 lists the VLE-defined **se_isync** instruction.

Table 225. Instruction Synchronization Instruction Set Index

Mnemonic	Instruction	Reference
se_isync	Instruction Synchronize	*Page -929*

12.2 Branch operation instructions

This section lists branch instructions that can be executed when a processor is in VLE mode and the registers that support them.

12.2.1 Registers for branch operations

The registers that support branch operations are grouped as follows:

- *Chapter 2.5.1: Condition register (CR) on page 61*"
- *Chapter 2.5.2: Link register (LR) on page 66*"
- *Chapter 2.5.3: Count register (CTR) on page 67*"

Condition register (CR)

The condition register (CR) is a 32-bit register. CR bits are numbered 32 (most-significant bit) to 63 (least-significant bit). The CR reflects the result of certain operations, and provides a mechanism for testing (and branching). The VLE extension implements the entire CR, but some comparison operations and all branch instructions are limited to using CR0–CR3. The full Book E condition register field and logical operations are provided however.

Access: User read/write

	31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
R W	CR0	CR1	CR2	CR3	CR4	CR5	CR6	CR7
Reset	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0

CR bits are grouped into eight 4-bit fields, CR0–CR7, which are set 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**).
- A specified CR field can be set by a move to the CR from another CR field (**e_mcrf**).
- CR field 0 can be set as the implicit result of an integer instruction.
- A specified condition register field can be set as the result of an integer compare instruction.
- CR field 0 can be set as the result of an integer bit test instruction.

Instructions are provided to perform logical operations on individual CR bits and to test individual condition register bits (see Book E).

Condition register settings for integer instructions

For all integer word instructions in which the Rc bit is defined and set, and for **addic.**, the first three bits of CR field 0 (CR[32–34]) are set by signed comparison of bits 32–63 of the result to zero, and the fourth bit of CR field 0 (CR[35]) is copied from the final state of XER[SO].

```
if      (target_register)32:63 < 0 then c ← 0b100
else if (target_register)32:63 > 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 in *Table 226*.

Table 226. CR0 encodings

CR Bit	Description
32	Negative (LT). Bit 32 of the result is equal to 1.
33	Positive (GT). Bit 32 of the result is equal to 0 and at least one of bits 33–63 of the result is non-zero.
34	Zero (EQ). Bits 32–63 of the result are equal to 0.
35	Summary overflow (SO). This is a copy of the final state XER[SO] at the completion of the instruction.

Condition register setting for compare instructions

For compare instructions, a CR field specified by the **cr**D operand in for the **e_cmph**, **e_cmphl**, **e_cmpi**, and **e_cmpli** instructions, or CR0 for the **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 in *Table 227*. A complete description of how the bits are set is given in the instruction descriptions and *Chapter 12.4.5: Integer compare and bit test instructions on page 872*."

Table 227. Condition register setting for compare instructions

CR Bit	Description
4×CRD + 32	Less than (LT) For signed-integer compare, GPR(**r**A or **r**X) < SCI8 or SI or GPR(**r**B or **r**Y). For unsigned-integer compare, GPR(**r**A or **r**X) $<_u$ SCI8 or UI or UI5 or GPR(**r**B or **r**Y).
4×CRD + 33	Greater than (GT) For signed-integer compare, GPR(**r**A or **r**X) > SCI8 or SI or UI5 or GPR(**r**B or **r**Y). For unsigned-integer compare, GPR(**r**A or **r**X) $>_u$ SCI8 or UI or UI5 or GPR(**r**B or **r**Y).
4×CRD + 34	Equal (EQ) For integer compare, GPR(**r**A or **r**X) = SCI8 or UI5 or SI or UI or GPR(**r**B or **r**Y).
4×CRD + 35	Summary overflow (SO) For integer compare, this is a copy of the final state of XER[SO] at the completion of the instruction.

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 *Chapter 12.4.5: Integer compare and bit test instructions on page 872*."

Link register (LR)

VLE instructions use the LR as defined in Book E, although the VLE extension defines a subset of all variants of Book E conditional branches involving the LR, as shown in *Table 228*.

Table 228. Branch to link register instruction comparison

Book E		VLE subset	
Instruction	**Syntax**	**Instruction**	**Syntax**
Branch Conditional to Link Register Branch Conditional to Link Register & Link	**bclr** BO,BI **bclrl** BO,BI	Branch (Absolute) to Link Register Branch (Absolute) to Link Register & Link	**se_blr** **se_blrl**
Branch Conditional & Link	**e_bcl** BO,BI,BD	Branch Conditional & Link	**e_bcl** BO32,BI32,BD15
		Branch (Absolute) & Link	**e_bl** BD24 **se_bl** BD8

Count register

VLE instructions use the count register (CTR) as defined in Book E, although the VLE extension defines a subset of the variants of Book E conditional branches involving the CTR, as shown in *Table 229*.

Table 229. Branch to count register instruction comparison

Book E		VLE	
Instruction	**Syntax**	**Instruction**	**Syntax**
Branch Conditional to Count Register Branch Conditional to Count Register & Link	**bcctr** BO,BI **bcctrl** BO,BI	Branch (Absolute) to Count Register Branch (Absolute) to Count Register & Link	**se_bctr** **se_bctrl**

12.2.2 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 *Chapter 10.2: Instruction memory addressing modes on page 854*."

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 whether or not 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 E's BO field.

If the BO32 field specifies that the CTR is to be decremented, CTR[32–63] are decremented. If BO[16,32] specifies a condition that must be TRUE or FALSE, that condition is obtained from the contents of CR[BI+32]. (Note that CR bits are numbered 32–63. BI refers to the BI field in the branch instruction encoding. For example, specifying BI = 2 refers to CR[34].)

Encodings for the BO32 field for the VLE extension are shown in *Table 230*.

Table 230. VLE extension BO32 encodings

BO32	Description
00	Branch if the condition is FALSE.
01	Branch if the condition is TRUE.
10	Decrement CTR[32–63], then branch if the decremented CTR[32–63]≠0.
11	Decrement CTR[32–63], then branch if the decremented CTR[32–63] = 0.

The encoding for the BO16 field for the VLE extension is shown in *Table 231*.

Table 231. VLE extension BO16 encodings

BO16	Description
0	Branch if the condition is FALSE.
1	Branch if the condition is TRUE.

The various branch instructions supported by the VLE extension are shown in *Table 232*.

Table 232. Branch instruction set index

Mnemonic	Instruction	Reference
e_b BD24 **e_bl** BD24	Branch Branch & Link	*Page -903*
se_b BD8 **se_bl** BD8	Branch Branch & Link	*Page -903*
e_bc BO32,BI32,BD15 **se_bc** BO16,BI16,BD8 **e_bcl** BO32,BI32,BD15	Branch Conditional Branch Conditional Branch Conditional & Link	*Page -904*
se_bctr **se_bctrl**	Branch to Count Register Branch to Count Register & Link	*Page -906*
se_blr **se_blrl**	Branch to Link Register Branch to Link Register & Link	*Page -908*

12.3 Condition register instructions

Condition register instructions are provided to transfer values to/from various portions of the CR. The VLE extension does not introduce any additional functionality beyond that defined in Book E for CR operations, but does remap the CR-logical and **mcrf** instruction functionality into major opcode 31. These instructions operate identically to the Book E instructions, but are encoded differently. *Table 233* lists condition register instructions supported in VLE mode.

Table 233. Condition register instruction set index

Mnemonic	Instruction	Reference
e_crand crbD,**crb**A,**crb**B	Condition Register AND	*Page -920*
e_crandc crbD,**crb**A,**crb**B	Condition Register AND with Complement	*Page -920*
e_creqv crbD,**crb**A,**crb**B	Condition Register Equivalent	*Page -920*
e_crnand crbD,**crb**A,**crb**B	Condition Register NAND	*Page -921*
e_crnor crbD,**crb**A,**crb**B	Condition Register NOR	*Page -922*
e_cror crbD,**crb**A,**crb**B	Condition Register OR	*Page -923*
e_crorc crbD,**crb**A,**crb**B	Condition Register OR with Complement	*Page -923*
e_crxor crbD,**crb**A,**crb**B	Condition Register XOR	*Page -925*
e_mcrf crD,**cr**S	Move Condition Register Field	*Page -936*
mcrxr crD	Move to Condition Register from Integer Exception Register	*Book E*
mfcr rD	Move From condition register	*Book E*
mtcrf FXM,**r**S	Move to Condition Register Fields	*Book E*

12.4 Integer instructions

This section lists the integer instructions supported by the VLE extension.

12.4.1 Integer load instructions

The integer load instructions compute the EA of the memory to be accessed as described in *Chapter 10.1: Data memory addressing modes on page 854*."

The byte, half word, or word in memory addressed by EA is loaded into GPR(**r**D) or GPR(**r**Z).

The VLE extension supports both big- and little-endian byte ordering for data accesses.

Some integer load instructions have an update form in which GPR(**r**A) is updated with the EA. For these forms, if **r**A ≠ 0 and **r**A ≠ **r**D, the EA is placed into GPR(**r**A) and the memory element (byte, half word, word, or double word) addressed by EA is loaded into GPR(**r**D). If **r**A = 0 or **r**A = **r**D, the instruction form is invalid. This is the same behavior as specified for load with update instructions in Book E.

Basic integer load instructions are listed in *Table 234*.

Table 234. Basic integer load instruction set index

Mnemonic	Instruction	Reference
e_lbz rD,D**(r**A**)** **e_lbzu r**D,D8**(r**A**)** **se_lbz r**Z,SD4**(r**X**)**	Load Byte and Zero Load Byte and Zero with Update Load Byte and Zero (16-bit form)	*Page -930*
lbzx rD,**r**A,**r**B **lbzux r**D,**r**A,**r**B	Load Byte and Zero Indexed Load Byte and Zero with Update Indexed	*Book E*

Table 234. Basic integer load instruction set index (continued)

Mnemonic	Instruction	Reference
e_lha rD,D(**r**A) **e_lhau** rD,D8(**r**A)	Load Halfword Algebraic Load Halfword Algebraic with Update	*Page -931*
lhax rD,**r**A,**r**B **lhaux** rD,**r**A,**r**B	Load Halfword Algebraic Indexed Load Halfword Algebraic with Update Indexed	*Book E*
e_lhz rD,D(**r**A) **e_lhzu** rD,D8(**r**A) **se_lhz** rZ,SD4(**r**X)	Load Halfword and Zero Load Halfword and Zero with Update Load Halfword and Zero (16-bit form)	*Page -932*
lhzx rD,**r**A,**r**B **lhzux** rD,**r**A,**r**B	Load Halfword and Zero Indexed Load Halfword and Zero with Update Indexed	*Book E*
e_lwz rD,D(**r**A) **e_lwzu** rD,D8(**r**A) **se_lwz** rZ,SD4(**r**X)	Load Word and Zero Load Word and Zero with Update Load Word and Zero (16-bit form)	*Page -935*
lwzx rD,**r**A,**r**B **lwzux** rD,**r**A,**r**B	Load Word and Zero Indexed Load Word and Zero with Update Indexed	*Book E*

Integer load byte-reversed instructions are listed in *Table 235*.

Table 235. Integer Load Byte-Reverse Instruction Set Index

Mnemonic	Instruction	Reference
lhbrx rD,**r**A,**r**B	Load Halfword Byte-Reverse Indexed	*Book E*
lwbrx rD,**r**A,**r**B	Load Word Byte-Reverse Indexed	*Book E*

The VLE-defined integer load multiple instruction is listed in *Table 236*.

Table 236. Integer load multiple instruction set index

Mnemonic	Instruction	Reference
e_lmw rD,D8(**r**A)	Load Multiple Word	*Page -934*

The VLE-defined integer load and reserve instruction is listed in *Table 237*.

Table 237. Integer load and reserve instruction set index

Mnemonic	Instruction	Reference
lwarx rD,**r**A,**r**B	Load Word And Reserve Indexed	*Book E*

12.4.2 Integer store instructions

The integer store instructions compute the EA of the memory to be accessed as described in *Chapter 10.1: Data memory addressing modes on page 854*."

The contents of GPR(**r**S) or GPR(**r**Z) are stored into the byte, half word, or word in memory addressed by EA.

The VLE extension supports both big- and little-endian byte ordering for data accesses.

Some integer store instructions have an update form, in which GPR(**r**A) is updated with the EA. For these forms, the following rules (from Book E) apply.

- If **r**A ≠ 0, the EA is placed into GPR(**r**A).
- If **r**S = **r**A, the contents of GPR(**r**S) are copied to the target memory element and then EA is placed into GPR(**r**A).

The basic integer store instructions are listed in *Table 238*.

Table 238. Basic integer store instruction set index

Mnemonic	Instruction	Reference
e_stb **r**S,D(**r**A) **e_stbu** **r**S,D8(**r**A) **se_stb** **r**Z,SD4(**r**X)	Store Byte Store Byte with Update Store Byte (16-bit form)	*Page -958*
stbx **r**S,**r**A,**r**B **stbux** **r**S,**r**A,**r**B	Store Byte Indexed Store Byte with Update Indexed	*Book E*
e_sth **r**S,D(**r**A) **e_sthu** **r**S,D8(**r**A) **se_sth** **r**Z,SD4(**r**X)	Store Halfword Store Halfword with Update Store Halfword (16-bit form)	*Page -959*
sthx **r**S,**r**A,**r**B **sthux** **r**S,**r**A,**r**B	Store Halfword Indexed Store Halfword with Update Indexed	*Book E*
e_stw **r**S,D(**r**A) **e_stwu** **r**S,D8(**r**A) **se_stw** **r**Z,SD4(**r**X)	Store Word Store Word with Update Store Word (16-bit form)	*Page -961*
stwx **r**S,**r**A,**r**B **stwux** **r**S,**r**A,**r**B	Store Word Indexed Store Word with Update Indexed	*Book E*

The integer store byte-reverse instructions are listed in *Table 239*.

Table 239. Integer store byte-reverse instruction set index

Mnemonic	Instruction	Reference
sthbrx **r**S,**r**A,**r**B	Store Halfword Byte-Reverse Indexed	*Book E*
stwbrx **r**S,**r**A,**r**B	Store Word Byte-Reverse Indexed	*Book E*

The integer store multiple instruction is listed in *Table 240*.

Table 240. Integer store multiple instruction set index

Mnemonic	Instruction	Reference
e_stmw **r**S,D8(**r**A)	Store Multiple Word	*Page -960*

The integer store conditional instruction is listed in *Table 241*.

Table 241. Integer store conditional instruction set index

Mnemonic	Instruction	Reference
stwcx. **r**S,**r**A,**r**B	Store Word Conditional Indexed	*Book E*

12.4.3 Integer arithmetic instructions

The integer 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 integer arithmetic instructions treat source operands as signed, two's complement integers unless the instruction is explicitly identified as performing an unsigned operation.

The **e_add2i.** instruction and the OIM5-form instruction, **se_subi.**, set the first three bits of CR0 to characterize bits 32–63 of the result. These bits are set by signed comparison of bits 32–63 of the result to zero.

e_addic[.] and **e_subfic**[.] always set CA to reflect the carry out of bit 32.

The integer arithmetic instructions are listed in *Table 242*.

Table 242. Integer arithmetic instruction set index

Mnemonic	Instruction	Reference
add rD,rA,rB **add.** rD,rA,rB **addo** rD,rA,rB **addo.** rD,rA,rB	Add	*Book E*
se_add rX,rY	Add	*Page -897*
addc rD,rA,rB **addc.** rD,rA,rB **addco** rD,rA,rB **addco.** rD,rA,rB	Add Carrying	*Book E*
adde rD,rA,rB **adde.** rD,rA,rB **addeo** rD,rA,rB **addeo.** rD,rA,rB	Add Extended	*Book E*
e_addi rD,rA,SCI8 **e_addi.** rD,rA,SCI8 **e_add16i** rD,rA,SI **e_add2i.** rD,SI **se_addi** rX,OIMM	Add Immediate	*Page -898*
e_addic rD,rA,SCI8 **e_addic.** rD,rA,SCI8	Add Immediate Carrying	*Page -900*
e_add2is rD,SI	Add Immediate Shifted	*Page -898*
divw rD,rA,rB **divw.** rD,rA,rB **divwo** rD,rA,rB **divwo.** rD,rA,rB	Divide Word	*Book E*
divwu rD,rA,rB **divwu.** rD,rA,rB **divwuo** rD,rA,rB **divwuo.** rD,rA,rB	Divide Word Unsigned	*Book E*

Table 242. Integer arithmetic instruction set index (continued)

Mnemonic	Instruction	Reference
mulhw rD,rA,rB **mulhw.** rD,rA,rB	Multiply High Word	*Book E*
mulhwu rD,rA,rB **mulhwu.** rD,rA,rB	Multiply High Word Unsigned	*Book E*
e_mulli rD,rA,SCI8 **e_mull2i** rD,SI	Multiply Low Immediate	*Page -944*
mullw rD,rA,rB **mullw.** rD,rA,rB **mullwo** rD,rA,rB **mullwo.** rD,rA,rB	Multiply Low Word	*Book E*
se_mullw rX,rY	Multiply Low Word	*Page -945*
neg rD,rA **se_neg** rX **neg.** rD,rA **nego** rD,rA **nego.** rD,rA	Negate	*Page -946*
se_sub rX,rY	Subtract	*Page -962*
subf rD,rA,rB **subf.** rD,rA,rB **subfo** rD,rA,rB **subfo.** rD,rA,rB	Subtract From	*Book E*
se_subf rX,rY	Subtract From	*Page -963*
subfc rD,rA,rB **subfc.** rD,rA,rB **subfco** rD,rA,rB **subfco.** rD,rA,rB	Subtract From Carrying	*Book E*
e_subfic rD,rA,SCI8 **e_subfic.** rD,rA,SCI8	Subtract From Immediate Carrying	*Page -964*
se_subi rX,OIMM **se_subi.** rX,OIMM	Subtract Immediate	*Page -965*

12.4.4 Integer logical and move instructions

Logical instructions perform bit-parallel operations on 32-bit operands or move register or immediate values into registers. The move instructions move values into a GP from either another GPR, or an immediate value.

The X-form logical instructions with Rc = 1 and the SCI8-form logical instructions with Rc = 1 set the first three bits of CR field 0 as described in *Chapter 12.4.3: Integer arithmetic instructions on page 869*." The logical instructions do not change XER[SO,OV,CA].

The integer logical instructions are listed in *Table 243*.

Table 243. Integer logical instruction set index

Mnemonic	Instruction	Reference
and[.] rA,rS,rB **se_and**[.] rX,rY	AND	*Page -901*
andc[.] rA,rS,rB **se_andc** rX,rY	AND with Complement	*Page -901*
e_andi[.] rA,rS,SCI8 **se_andi** rX,UI5 **e_and2i.** rD,UI	AND Immediate	*Page -901*
e_and2is. rD,UI	AND Immediate Shifted	*Page -901*
se_bclri rX,UI5	Bit Clear	*Page -905*
se_bgeni rX,UI5	Bit Generate	*Page -907*
se_bmski rX,UI5	Bit Mask Generate	*Page -909*
se_bseti rX,UI5	Bit Set	*Page -910*
cntlzw rA,rS **cntlzw.** rA,rS	Count Leading Zeros Word	*Book E*
eqv rA,rS,rB **eqv.** rA,rS,rB	Equivalent	*Book E*
extsb rA,rS **extsb.** rA,rS **se_extsb** rX	Extend Sign Byte	*Page -926*
extsh rA,rS **extsh.** rA,rS **se_extsh** rX	Extend Sign Halfword	*Page -926*
se_extzb rX	Extend with Zeros Byte	*Page -927*
se_extzh rX	Extend with Zeros Halfword	*Page -927*
e_li rD,LI20 **se_li** rX,UI7	Load Immediate	*Page -933*
e_lis rD,UI	Load Immediate Shifted	*Page -933*
se_mfar rX,arY	Move from Alternate Register	*Page -937*
se_mr rX,rY	Move Register	*Page -940*
se_mtar arX,rY	Move to Alternate Register	*Page -941*
nand rA,rS,rB **nand.** rA,rS,rB	NAND	*Book E*
nor rA,rS,rB **nor.** rA,rS,rB	NOR	*Book E*
or rA,rS,rB **or.** rA,rS,rB **se_or** rX,rY	OR	*Page -948*
se_not rX	NOT	*Page -947*

Table 243. Integer logical instruction set index (continued)

Mnemonic	Instruction	Reference
orc rA,**r**S,**r**B **orc.** rA,**r**S,**r**B	OR with Complement	*Book E*
e_ori[.] rA,**r**S,SCI8 **e_or2i** rD,UI	OR Immediate	*Page -966*
e_or2is rD,UI	OR Immediate Shifted	*Page -966*
xor rA,**r**S,**r**B **xor.** rA,**r**S,**r**B	XOR	*Book E*
e_xori[.] rA,**r**S,SCI8	XOR Immediate	*Page -966*

12.4.5 Integer compare and bit test instructions

The integer compare instructions compare the contents of GPR(rA) with one of the following:

- The value of the SCI8 field
- 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 GPR(**r**B) or GPR(**r**Y).

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**.

When operands are treated as 32-bit signed quantities, GPR*n*[32] is the sign bit. When operands are treated as 16-bit signed quantities, GPR*n*[48] is the sign bit.

For 32-bit implementations, the L field must be zero.

Compare instructions set one of the left-most three bits of the designated CR field and clears the other two. XER[SO] is copied to bit 3 of the designated CR field.

The CR field is set as shown in *Table 244*.

Table 244. CR settings for compare instructions

Bit	Name	Description
0	LT	(**r**A or **r**X) < SCI8, SI, UI5, or GPR(**r**B or **r**Y) (signed comparison) (**r**A or **r**X) $<_u$ SCI8, UI, UI5 or GPR(**r**B or **r**Y) (unsigned comparison)
1	GT	(**r**A or **r**X) > SCI8, SI, UI5, or GPR(**r**B or **r**Y) (signed comparison) (**r**A or **r**X) $>_u$ SCI8, UI, UI5 or GPR(**r**B or **r**Y) (unsigned comparison)
2	EQ	(**r**A or **r**X) = SCI8, SI, UI, UI5, or GPR(**r**B or **r**Y)
3	SO	Summary overflow from the XER

The integer bit test instruction tests the bit specified by the UI5 instruction field and sets the CR0 field as shown in *Table 245*.

Table 245. CR settings for integer bit test instructions

Bit	Name	Description
0	LT	Always cleared
1	GT	RX_{ui5} == 1
2	EQ	RX_{ui5} == 0
3	SO	Summary overflow from the XER

Table 246 is an index for integer compare and bit test operations.

Table 246. Integer compare and bit test instruction set index

Mnemonic	Instruction	Reference
se_btsti **r**X,UI5	Bit Test Immediate	*Page -911*
cmp **cr**D,L,**r**A,**r**B **se_cmp** **r**X,**r**Y	Compare	*Page -912*
e_cmph **cr**D,**r**A,**r**B **se_cmph** **r**X,**r**Y	Compare Halfword	*Page -914*
e_cmph16i **r**A,SI16	Compare Halfword Immediate	*Page -914*
e_cmphl **cr**D,**r**A,**r**B **se_cmphl** **r**X,**r**Y	Compare Halfword Logical	*Page -916*
e_cmphl16i **r**A,UI16	Compare Halfword Logical Immediate	*Page -916*
e_cmpi **cr**D,**r**A,SCI8 **e_cmp16i** **r**A,SI16 **se_cmpi** **r**X,UI5	Compare Immediate	*Page -912*
cmpl **cr**D,L,**r**A,**r**B **se_cmpl** **r**X,**r**Y	Compare Logical	*Page -918*
e_cmpli **cr**D,**r**A,SCI8 **e_cmpl16i** **r**A,UI16 **se_cmpli** **r**X,UI5	Compare Logical Immediate	*Page -918*

12.4.6 Integer select instruction

The **isel** 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 integer select instruction is listed in *Table 247*.

Table 247. Integer select instruction set index

Mnemonic	Instruction	Reference
isel **r**D,**r**A,**r**B,**cr**b	Integer Select	EIS

12.4.7 Integer trap instructions

Trap instructions test for a specified set of conditions by comparing the contents of one GPR with a second GPR. If any of the conditions tested by a Trap instruction are met, a trap

exception type program interrupt is invoked. If none of the tested conditions are met, instruction execution continues normally.

The contents of GPR(**r**A) are compared with the contents of GPR(**r**B). For **twi** and **tw**, only the contents of bits 32–63 of **r**A (and **r**B) participate in the comparison.

This comparison results in five conditions that are ANDed with TO. If the result is not 0, the trap exception type program interrupt is invoked. These conditions are as shown in *Table 248*.

Table 248. Integer trap conditions

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

The integer trap instruction is listed in *Table 249*.

Table 249. Integer trap instruction set index

Mnemonic	Instruction	Reference
tw TO,**r**A,**r**B	Trap Word	*Book E*

12.4.8 Integer rotate and shift instructions

Instructions are provided that perform shifts and rotates on data from a GPR and return the result, or a portion of the result, to a GPR.

The rotation operations rotate a 32-bit quantity left by a specified number of bit positions. Bits that exit from position 32 enter at position 63.

The $rotate_{32}$ operation is used to rotate a given 32-bit quantity.

Some rotate and shift instructions employ a mask generator. The mask is 32 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 32 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

$mask_{32:mstop}$ = 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 bits 32–63 of the mask.

The use of the mask is described in following sections.

The rotate word and shift word instructions with Rc = 1 set the first three bits of CR field 0 as described in Book E. 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.

The instructions in *Table 250* rotate the contents of a register. Depending on the instruction type, the amount of the rotation is either specified as an immediate, or contained in a GPR.

Table 250. Integer rotate instruction set index

Mnemonic	Instruction	Reference
e_rlw rA,rS,rB	Rotate Left Word	*Page -951*
e_rlwi rA,rS,SH	Rotate Left Word Immediate	*Page -951*

The instructions in *Table 251* rotate the contents of a register. Depending on the instruction type, the result of the rotation is either 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; if a 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 32-*n*, where *n* is the number of bits by which to rotate right. They allow right-rotation of the contents of bits 32–63 of a register to be performed (in concept) by a left-rotation of 32-*n*, where *n* is the number of bits by which to rotate right.

Table 251. Integer rotate with mask instruction set index

Mnemonic	Instruction	Reference
e_rlwimi rA,rS,SH,MB,ME	Rotate Left Word Immediate then Mask Insert	*Page -952*
e_rlwinm rA,rS,SH,MB,ME	Rotate Left Word Immediate then AND with Mask	*Page -953*

The integer shift instructions are listed in <Cross Refs>Table 252.

Table 252. Integer shift instruction set index

Mnemonic	Instruction	Reference
slw rA,rS,rB **slw.** rA,rS,rB **se_slw** rX,rY	Shift Left Word	*Page -955*
e_slwi rA,rS,SH **se_slwi** rX,UI5	Shift Left Word Immediate	*Page -955*
sraw rA,rS,rB **sraw.** rA,rS,rB **se_sraw** rX,rY	Shift Right Algebraic Word	*Page -956*
srawi rA,rS,SH **srawi.** rA,rS,SH **se_srawi** rX,UI5	Shift Right Algebraic Word Immediate	*Page -956*

Table 252. Integer shift instruction set index (continued)

Mnemonic	Instruction	Reference
srw rA,rS,rB **srw.** rA,rS,rB **se_srw** rX,rY	Shift Right Word	*Page -957*
e_srwi rA,rS,SH **se_srwi** rX,UI5	Shift Right Word Immediate	*Page -957*

12.5 Storage control instructions

This section lists storage control instructions, which include the following:

- *Chapter 12.5.1: Storage synchronization instructions on page 876"*
- *Chapter 12.5.2: Cache management instructions on page 876"*
- *Chapter 12.5.3: TLB management instructions on page 877*"

12.5.1 Storage synchronization instructions

The memory synchronization instructions implemented by the VLE extension are identical to those defined in Book E.

The storage synchronization instructions are listed in *Table 253*.

Table 253. Storage synchronization instruction set index

Mnemonic	Instruction	Reference
mbar	Memory Barrier	*Book E*
msync	Memory Synchronize	*Book E*

12.5.2 Cache management instructions

Cache management instructions implemented by the VLE extension are identical to those defined in Book E.

The cache management instructions are listed in *Table 254*.

Table 254. Cache management instruction set index

Mnemonic	Instruction	Reference
dcba rA,**r**B	Data Cache Block Allocate	*Book E*
dcbf rA,**r**B	Data Cache Block Flush	*Book E*
dcbi rA,**r**B	Data Cache Block Invalidate	*Book E*
dcbst rA,**r**B	Data Cache Block Store	*Book E*
dcbt CT,**r**A,**r**B	Data Cache Block Touch	*Book E*
dcbtls CT,rA,rB	Data Cache Block Touch and Lock Set	*Book E*
dcbtst CT,**r**A,**r**B	Data Cache Block Touch for Store	*Book E*
dcbz rA,**r**B	Data Cache Block set to Zero	*Book E*
icbi rA,**r**B	Instruction Cache Block Invalidate	*Book E*
icbt CT,**r**A,**r**B	Instruction Cache Block Touch	*Book E*

12.5.3 TLB management instructions

The TLB management instructions implemented by the VLE extension are identical to those defined in Book E and in the EIS. The TLB management instructions are listed in *Table 255*.

Table 255. TLB management instruction set index

Mnemonic	Instruction	Reference
tlbivax rA,**r**B	TLB Invalidate Virtual Address Indexed	Book E
tlbre	TLB Read Entry	Book E
tlbsx rA,**r**B	TLB Search Indexed	Book E
tlbsync	TLB Synchronize	Book E
tlbwe	TLB Write Entry	Book E

12.5.4 Instruction alignment and byte ordering

To be recognized by the instruction decoder, an instruction fetched from memory must be placed in the pipeline with its bytes in the proper order. Book E allows instructions to be placed into memory marked as either big- or little-endian. This is manageable because Book E instructions are always word-size aligned on word boundaries. VLE instructions can be either half word or word size, and are aligned on half-word boundaries. Because of this, only big-endian instruction memory is supported when executing from a page of VLE instructions. Attempts to execute VLE instructions from a page marked as little-endian generate an instruction storage interrupt byte-ordering exception.

12.6 Instruction listings

This section lists instructions either defined or supported by the VLE extension.

Table 256 lists instructions by instruction name.

Table 256. Instructions listed by name

Instruction	Mnemonic	Reference
Add	**add** rD,**r**A,**r**B **add.** rD,**r**A,**r**B **addo** rD,**r**A,**r**B **addo.** rD,**r**A,**r**B	*Book E*
Add Carrying	**addc** rD,**r**A,**r**B **addc.** rD,**r**A,**r**B **addco** rD,**r**A,**r**B **addco.** rD,**r**A,**r**B	*Book E*
Add Extended	**adde** rD,**r**A,**r**B **adde.** rD,**r**A,**r**B **addeo** rD,**r**A,**r**B **addeo.** rD,**r**A,**r**B	*Book E*
AND with Complement	**andc**[.] rA,**r**S,**r**B **se_andc** rX,**r**Y	*Book E* *Page -901*
AND	**and**[.] rA,**r**S,**r**B **se_and**[.] rX,**r**Y	*Book E* *Page -901*
Compare	**cmp** cr**r**D,L,**r**A,**r**B **se_cmp** rX,**r**Y	*Book E* *Page -912*
Compare Logical	**cmpl** cr**r**D,L,**r**A,**r**B **se_cmpl** rX,**r**Y	*Book E* *Page -918*
Count Leading Zeros Word	**cntlzw** rA,**r**S **cntlzw.** rA,**r**S	*Book E*
Data Cache Block Allocate	**dcba** rA,**r**B	*Book E*
Data Cache Block Flush	**dcbf** rA,**r**B	*Book E*
Data Cache Block Invalidate	**dcbi** rA,**r**B	*Book E*
Data Cache Block Store	**dcbst** rA,**r**B	*Book E*
Data Cache Block Touch	**dcbt** CT,**r**A,**r**B	*Book E*
Data Cache Block Touch for Store	**dcbtst** CT,**r**A,**r**B	*Book E*
Data Cache Block set to Zero	**dcbz** rA,**r**B	*Book E*
Divide Word	**divw** rD,**r**A,**r**B **divw.** rD,**r**A,**r**B **divwo** rD,**r**A,**r**B **divwo.** rD,**r**A,**r**B	*Book E*
Divide Word Unsigned	**divwu** rD,**r**A,**r**B **divwu.** rD,**r**A,**r**B **divwuo** rD,**r**A,**r**B **divwuo.** rD,**r**A,**r**B	*Book E*
Equivalent	**eqv** rA,**r**S,**r**B **eqv.** rA,**r**S,**r**B	*Book E*

Table 256. Instructions listed by name (continued)

Instruction	Mnemonic	Reference
Extend Sign Byte	**extsb** rA,rS **extsb.** rA,rS **se_extsb** rX	*Book E* *Book E* *Page -926*
Extend Sign Halfword	**extsh** rA,rS **extsh.** rA,rS **se_extsh** rX	*Book E* *Book E* *Page -926*
Add Immediate Shifted	**e_add2is** rD,SI	*Page -897*
Add Immediate	**e_addi** rD,rA,SCI8 **e_addi.** rD,rA,SCI8 **e_add16i** rD,rA,SI **e_add2i.** rD,SI **se_addi** rX,OIMM	*Page -897*
Add Immediate Carrying	**e_addic** rD,rA,SCI8 **e_addic.** rD,rA,SCI8	*Page -900*
AND Immediate Shifted	**e_and2is.** rD,UI	*Page -901*
AND Immediate	**e_andi**[.] rA,rS,SCI8 **se_andi** rX,UI5 **e_and2i.** rD,UI	*Page -901*
Branch Conditional Branch Conditional Branch Conditional & Link	**e_bc** BO32,BI32,BD15 **se_bc** BO16,BI16,BD8 **e_bcl** BO32,BI32,BD15	*Page -904*
Branch Branch & Link	**e_b** BD24 **e_bl** BD24	*Page -903*
Compare Halfword	**e_cmph** crD,rA,rB **se_cmph** rX,rY	*Page -914*
Compare Halfword Immediate	**e_cmph16i** rA,SI16	*Page -914*
Compare Halfword Logical	**e_cmphl** crD,rA,rB **se_cmphl** rX,rY	*Page -916*
Compare Halfword Logical Immediate	**e_cmphl16i** rA,UI16	*Page -916*
Compare Immediate	**e_cmpi** crD,rA,SCI8 **e_cmp16i** rA,SI16 **se_cmpi** rX,UI5	*Page -912*
Compare Logical Immediate	**e_cmpli** crD,rA,SCI8 **e_cmpl16i** rA,UI16 **se_cmpli** rX,UI5	*Page -918*
Condition Register AND	**e_crand** crbD,crbA,crbB	*Page -920*
Condition Register AND with Complement	**e_crandc** crbD,crbA,crbB	*Page -920*
Condition Register Equivalent	**e_creqv** crbD,crbA,crbB	*Page -920*
Condition Register NAND	**e_crnand** crbD,crbA,crbB	*Page -921*
Condition Register NOR	**e_crnor** crbD,crbA,crbB	*Page -922*

Table 256. Instructions listed by name (continued)

Instruction	Mnemonic	Reference
Condition Register OR	**e_cror crb**D,**crb**A,**crb**B	Page -923
Condition Register OR with Complement	**e_crorc crb**D,**crb**A,**crb**B	Page -924
Condition Register XOR	**e_crxor crb**D,**crb**A,**crb**B	Page -925
Load Byte and Zero Load Byte and Zero with Update Load Byte and Zero (16-bit form)	**e_lbz r**D,D**(r**A**)** **e_lbzu r**D,D8**(r**A**)** **se_lbz r**Z,SD4**(r**X**)**	Page -930
Load Halfword Algebraic Load Halfword Algebraic with Update	**e_lha r**D,D**(r**A**)** **e_lhau r**D,D8**(r**A**)**	Page -931
Load Halfword and Zero Load Halfword and Zero with Update Load Halfword and Zero (16-bit form)	**e_lhz r**D,D**(r**A**)** **e_lhzu r**D,D8**(r**A**)** **se_lhz r**Z,SD4**(r**X**)**	Page -932
Load Immediate	**e_li r**D,LI20 **se_li r**X,UI7	Page -933
Load Immediate Shifted	**e_lis r**D,UI	Page -933
Load Multiple Word	**e_lmw r**D,D8**(r**A**)**	Page -934
Load Word and Zero Load Word and Zero with Update Load Word and Zero (16-bit form)	**e_lwz r**D,D**(r**A**)** **e_lwzu r**D,D8**(r**A**)** **se_lwz r**Z,SD4**(r**X**)**	Page -935
Move Condition Register Field	**e_mcrf cr**D,**cr**S	Page -936
Multiply Low Immediate	**e_mulli r**D,**r**A,SCI8 **e_mull2i r**D,SI	Page -944
OR Immediate Shifted	**e_or2is r**D,UI	Page -948
OR Immediate	**e_ori**[.] **r**A,**r**S,SCI8 **e_or2i r**D,UI	Page -948
Rotate Left Word	**e_rlw r**A,**r**S,**r**B	Page -951
Rotate Left Word Immediate	**e_rlwi r**A,**r**S,SH	Page -951
Rotate Left Word Immediate then Mask Insert	**e_rlwimi r**A,**r**S,SH,MB,ME	Page -952
Rotate Left Word Immediate then AND with Mask	**e_rlwinm r**A,**r**S,SH,MB,ME	Page -953
Shift Left Word Immediate	**e_slwi r**A,**r**S,SH **se_slwi r**X,UI5	Page -955
Shift Right Word Immediate	**e_srwi r**A,**r**S,SH **se_srwi r**X,UI5	Page -957
Store Byte Store Byte with Update Store Byte (16-bit form)	**e_stb r**S,D**(r**A**)** **e_stbu r**S,D8**(r**A**)** **se_stb r**Z,SD4**(r**X**)**	Page -958

Table 256. Instructions listed by name (continued)

Instruction	Mnemonic	Reference
Store Halfword Store Halfword with Update Store Halfword (16-bit form)	**e_sth** rS,D(**r**A) **e_sthu** **r**S,D8(**r**A) **se_sth** **r**Z,SD4(**r**X)	*Page -959*
Store Multiple Word	**e_stmw** **r**S,D8(**r**A)	*Page -960*
Store Word Store Word with Update Store Word (16-bit form)	**e_stw** **r**S,D(**r**A) **e_stwu** **r**S,D8(**r**A) **se_stw** **r**Z,SD4(**r**X)	*Page -961*
Subtract From Immediate Carrying	**e_subfic** **r**D,**r**A,SCI8 **e_subfic.** **r**D,**r**A,SCI8	*Page -964*
XOR Immediate	**e_xori**[.] **r**A,**r**S,SCI8	*Page -966*
Instruction Cache Block Invalidate	**icbi** **r**A,**r**B	*Book E*
Instruction Cache Block Touch	**icbt** CT,**r**A,**r**B	*Book E*
Integer Select	**isel** **r**D,**r**A,**r**B,**cr**b	*EIS*
Load Byte and Zero Indexed Load Byte and Zero with Update Indexed	**lbzx** **r**D,**r**A,**r**B **lbzux** **r**D,**r**A,**r**B	*Book E*
Load Halfword Algebraic Indexed Load Halfword Algebraic with Update Indexed	**lhax** **r**D,**r**A,**r**B **lhaux** **r**D,**r**A,**r**B	*Book E*
Load Halfword Byte-Reverse Indexed	**lhbrx** **r**D,**r**A,**r**B	*Book E*
Load Halfword and Zero Indexed Load Halfword and Zero with Update Indexed	**lhzx** **r**D,**r**A,**r**B **lhzux** **r**D,**r**A,**r**B	*Book E*
Load Word And Reserve Indexed	**lwarx** **r**D,**r**A,**r**B	*Book E*
Load Word Byte-Reverse Indexed	**lwbrx** **r**D,**r**A,**r**B	*Book E*
Load Word and Zero Indexed Load Word and Zero with Update Indexed	**lwzx** **r**D,**r**A,**r**B **lwzux** **r**D,**r**A,**r**B	*Book E*
Memory Barrier	**mbar**	*Book E*
Move to Condition Register from Integer Exception Register	**mcrxr** **cr**D	*Book E*
Move From condition register	**mfcr** **r**D	*Book E*
Move From Device Control Register	**mfdcr** **r**D,DCRN	*Book E*
Move From Machine State Register	**mfmsr** **r**D	*Book E*
Move From Special Purpose Register	**mfspr** **r**D,SPRN	*Book E*
Memory Synchronize	**msync**	*Book E*
Move to Condition Register Fields	**mtcrf** FXM,**r**S	*Book E*
Move To Device Control Register	**mtdcr** **DCRN**,**r**S	*Book E*
Move To Machine State Register	**mtmsr** **r**S	*Book E*
Move To Special Purpose Register	**mtspr** **SPRN**,**r**S	*Book E*

Table 256. Instructions listed by name (continued)

Instruction	Mnemonic	Reference
Multiply High Word	**mulhw** rD,rA,rB **mulhw.** rD,rA,rB	Book E
Multiply High Word Unsigned	**mulhwu** rD,rA,rB **mulhwu.** rD,rA,rB	Book E
Multiply Low Word	**mullw** rD,rA,rB **mullw.** rD,rA,rB **mullwo** rD,rA,rB **mullwo.** rD,rA,rB	Book E
NAND	**nand** rA,rS,rB **nand.** rA,rS,rB	Book E
Negate	**neg** rD,rA **se_neg** rX **neg.** rD,rA **nego** rD,rA **nego.** rD,rA	Book E Page -946 Book E Book E Book E
NOR	**nor** rA,rS,rB **nor.** rA,rS,rB	Book E
OR	**or** rA,rS,rB **or.** rA,rS,rB **se_or** rX,rY	Book E Book E Page -948
OR with Complement	**orc** rA,rS,rB **orc.** rA,rS,rB	Book E
Add	**se_add** rX,rY	Page -897
Bit Clear	**se_bclri** rX,UI5	Page -905
Branch to Count Register Branch to Count Register & Link	**se_bctr** **se_bctrl**	Page -906
Bit Generate	**se_bgeni** rX,UI5	Page -907
Branch to Link Register Branch to Link Register & Link	**se_blr** **se_blrl**	Page -908
Bit Mask Generate	**se_bmski** rX,UI5	Page -909
Bit Set	**se_bseti** rX,UI5	Page -910
Branch Branch & Link	**se_b** BD8 **se_bl** BD8	Page -903
Bit Test Immediate	**se_btsti** rX,UI5	Page -911
Extend with Zeros Byte	**se_extzb** rX	Page -927
Extend with Zeros Halfword	**se_extzh** rX	Page -927
Instruction Synchronize	**se_isync**	Page -929
Move from Alternate Register	**se_mfar** rX,arY	Page -937
Move From Count Register	**se_mfctr** rX	Page -938

Table 256. Instructions listed by name (continued)

Instruction	Mnemonic	Reference
Move From Link Register	**se_mflr** rX	Page -939
Move Register	**se_mr** rX,rY	Page -940
Move to Alternate Register	**se_mtar** arX,rY	Page -941
Move To Count Register	**se_mtctr** rX	Page -942
Move To Link Register	**se_mtlr** rX	Page -943
Multiply Low Word	**se_mullw** rX,rY	Page -945
NOT	**se_not** rX	Page -947
Subtract	**se_sub** rX,rY	Page -962
Subtract From	**se_subf** rX,rY	Page -963
Subtract Immediate	**se_subi** rX,OIMM **se_subi.** rX,OIMM	Page -965
Shift Left Word	**slw** rA,rS,rB **slw.** rA,rS,rB **se_slw** rX,rY	Book E Book E Page -955
Shift Right Algebraic Word	**sraw** rA,rS,rB **sraw.** rA,rS,rB **se_sraw** rX,rY	Book E Book E Page -956
Shift Right Algebraic Word Immediate	**srawi** rA,rS,SH **srawi.** rA,rS,SH **se_srawi** rX,UI5	Book E Book E Page -956
Shift Right Word	**srw** rA,rS,rB **srw.** rA,rS,rB **se_srw** rX,rY	Book E Book E Page -957
Store Byte Indexed Store Byte with Update Indexed	**stbx** rS,rA,rB **stbux** rS,rA,rB	Book E
Store Halfword Byte-Reverse Indexed	**sthbrx** rS,rA,rB	Book E
Store Halfword Indexed Store Halfword with Update Indexed	**sthx** rS,rA,rB **sthux** rS,rA,rB	Book E
Store Word Byte-Reverse Indexed	**stwbrx** rS,rA,rB	Book E
Store Word Conditional Indexed	**stwcx.** rS,rA,rB	Book E
Store Word Indexed Store Word with Update Indexed	**stwx** rS,rA,rB **stwux** rS,rA,rB	Book E
Subtract From	**subf** rD,rA,rB **subf.** rD,rA,rB **subfo** rD,rA,rB **subfo.** rD,rA,rB	Book E

Table 256. Instructions listed by name (continued)

Instruction	Mnemonic	Reference
Subtract From Carrying	**subfc** rD,rA,rB **subfc.** rD,rA,rB **subfco** rD,rA,rB **subfco.** rD,rA,rB	*Book E*
TLB Invalidate Virtual Address Indexed	**tlbivax** rA,rB	*Book E*
TLB Read Entry	**tlbre**	*Book E*
TLB Search Indexed	**tlbsx** rA,rB	*Book E*
TLB Synchronize	**tlbsync**	*Book E*
TLB Write Entry	**tlbwe**	*Book E*
Trap Word	**tw** TO,rA,rB	*Book E*
Write MSR External Enable	**wrtee** rA	*Book E*
Write MSR External Enable Immediate	**wrteei E**	*Book E*
XOR	**xor** rA,rS,rB **xor.** rA,rS,rB	*Book E*

Table 257 lists instructions by mnemonic.

Table 257. Instructions listed by mnemonic

Mnemonic	Instruction	Reference
add rD,rA,rB **add.** rD,rA,rB **addo** rD,rA,rB **addo.** rD,rA,rB	Add	*Book E*
addc rD,rA,rB **addc.** rD,rA,rB **addco** rD,rA,rB **addco.** rD,rA,rB	Add Carrying	*Book E*
adde rD,rA,rB **adde.** rD,rA,rB **addeo** rD,rA,rB **addeo.** rD,rA,rB	Add Extended	*Book E*
andc[.] rA,rS,rB	AND with Complement	*Book E*
and[.] rA,rS,rB	AND	*Book E*
cmp crD,L,rA,rB	Compare	*Book E*
cmpl crD,L,rA,rB	Compare Logical	*Book E*
cntlzw rA,rS **cntlzw.** rA,rS	Count Leading Zeros Word	*Book E*
dcba rA,rB	Data Cache Block Allocate	*Book E*
dcbf rA,rB	Data Cache Block Flush	*Book E*

Table 257. Instructions listed by mnemonic (continued)

Mnemonic	Instruction	Reference
dcbi rA,rB	Data Cache Block Invalidate	*Book E*
dcbst rA,rB	Data Cache Block Store	*Book E*
dcbt CT,rA,rB	Data Cache Block Touch	*Book E*
dcbtst CT,rA,rB	Data Cache Block Touch for Store	*Book E*
dcbz rA,rB	Data Cache Block set to Zero	*Book E*
divw rD,rA,rB **divw.** rD,rA,rB **divwo** rD,rA,rB **divwo.** rD,rA,rB	Divide Word	*Book E*
divwu rD,rA,rB **divwu.** rD,rA,rB **divwuo** rD,rA,rB **divwuo.** rD,rA,rB	Divide Word Unsigned	*Book E*
eqv rA,rS,rB **eqv.** rA,rS,rB	Equivalent	*Book E*
extsb rA,rS **extsb.** rA,rS	Extend Sign Byte	*Book E*
extsh rA,rS **extsh.** rA,rS	Extend Sign Halfword	*Book E*
e_add2is rD,SI	Add Immediate Shifted	*Page -897*
e_addi rD,rA,SCI8 **e_addi.** rD,rA,SCI8 **e_add16i** rD,rA,SI **e_add2i.** rD,SI	Add Immediate	*Page -897*
e_addic rD,rA,SCI8 **e_addic.** rD,rA,SCI8	Add Immediate Carrying	*Page -900*
e_and2is. rD,UI	AND Immediate Shifted	*Page -901*
e_andi[.] rA,rS,SCI8 **e_and2i.** rD,UI	AND Immediate	*Page -901*
e_bc BO32,BI32,BD15 **e_bcl** BO32,BI32,BD15	Branch Conditional Branch Conditional & Link	*Page -904*
e_b BD24 **e_bl** BD24	Branch Branch & Link	*Page -903*
e_cmph crD,rA,rB	Compare Halfword	*Page -914*
e_cmph16i rA,SI16	Compare Halfword Immediate	*Page -914*
e_cmphl crD,rA,rB	Compare Halfword Logical	*Page -916*
e_cmphl16i rA,UI16	Compare Halfword Logical Immediate	*Page -916*
e_cmpi crD,rA,SCI8 **e_cmp16i** rA,SI16	Compare Immediate	*Page -912*

Table 257. Instructions listed by mnemonic (continued)

Mnemonic	Instruction	Reference
e_cmpli crD,rA,SCI8 **e_cmpl16i rA,UI16**	Compare Logical Immediate	*Page -918*
e_crand crbD,crbA,crbB	Condition Register AND	*Page -920*
e_crandc crbD,crbA,crbB	Condition Register AND with Complement	*Page -920*
e_creqv crbD,crbA,crbB	Condition Register Equivalent	*Page -920*
e_crnand crbD,crbA,crbB	Condition Register NAND	*Page -921*
e_crnor crbD,crbA,crbB	Condition Register NOR	*Page -922*
e_cror crbD,crbA,crbB	Condition Register OR	*Page -923*
e_crorc crbD,crbA,crbB	Condition Register OR with Complement	*Page -924*
e_crxor crbD,crbA,crbB	Condition Register XOR	*Page -925*
e_lbz rD,D(rA) **e_lbzu rD,D8(rA)**	Load Byte and Zero Load Byte and Zero with Update	*Page -930*
e_lha rD,D(rA) **e_lhau rD,D8(rA)**	Load Halfword Algebraic Load Halfword Algebraic with Update	*Page -931*
e_lhz rD,D(rA) **e_lhzu rD,D8(rA)**	Load Halfword and Zero Load Halfword and Zero with Update	*Page -932*
e_li rD,LI20	Load Immediate	*Page -933*
e_lis rD,UI	Load Immediate Shifted	*Page -933*
e_lmw rD,D8(rA)	Load Multiple Word	*Page -935*
e_lwz rD,D(rA) **e_lwzu rD,D8(rA)**	Load Word and Zero Load Word and Zero with Update	*Page -936*
e_mcrf crD,crS	Move Condition Register Field	*Page -944*
e_mulli rD,rA,SCI8 **e_mull2i rD,SI**	Multiply Low Immediate	*Page -948*
e_or2is rD,UI	OR Immediate Shifted	*Page -948*
e_ori[.] **rA,rS,SCI8** **e_or2i rD,UI**	OR Immediate	*Page -951*
e_rlw rA,rS,rB	Rotate Left Word	*Page -951*
e_rlwi rA,rS,SH	Rotate Left Word Immediate	*Page -952*
e_rlwimi rA,rS,SH,MB,ME	Rotate Left Word Immediate then Mask Insert	*Page -953*
e_rlwinm rA,rS,SH,MB,ME	Rotate Left Word Immediate then AND with Mask	*Page -955*
e_slwi rA,rS,SH	Shift Left Word Immediate	*Page -935*
e_srwi rA,rS,SH	Shift Right Word Immediate	*Book E*
e_stb rS,D(rA) **e_stbu rS,D8(rA)**	Store Byte Store Byte with Update	*Page -958*

Table 257. Instructions listed by mnemonic (continued)

Mnemonic	Instruction	Reference
e_sth rS,D(**r**A) **e_sthu** rS,D8(**r**A)	Store Halfword Store Halfword with Update	Page -959
e_stmw rS,D8(**r**A)	Store Multiple Word	Page -960
e_stw rS,D(**r**A) **e_stwu** rS,D8(**r**A)	Store Word Store Word with Update	Page -961
e_subfic rD,**r**A,SCI8 **e_subfic.** rD,**r**A,SCI8	Subtract From Immediate Carrying	Page -964
e_xori[.] rA,**r**S,SCI8	XOR Immediate	Page -966
icbi rA,**r**B	Instruction Cache Block Invalidate	Book E
icbt CT,**r**A,**r**B	Instruction Cache Block Touch	Book E
isel rD,**r**A,**r**B,**cr**b	Integer Select	EIS
lbzx rD,**r**A,**r**B **lbzux** rD,**r**A,**r**B	Load Byte and Zero Indexed Load Byte and Zero with Update Indexed	Book E
lhax rD,**r**A,**r**B **lhaux** rD,**r**A,**r**B	Load Halfword Algebraic Indexed Load Halfword Algebraic with Update Indexed	Book E
lhbrx rD,**r**A,**r**B	Load Halfword Byte-Reverse Indexed	Book E
lhzx rD,**r**A,**r**B **lhzux** rD,**r**A,**r**B	Load Halfword and Zero Indexed Load Halfword and Zero with Update Indexed	Book E
lwarx rD,**r**A,**r**B	Load Word And Reserve Indexed	Book E
lwbrx rD,**r**A,**r**B	Load Word Byte-Reverse Indexed	Book E
lwzx rD,**r**A,**r**B **lwzux** rD,**r**A,**r**B	Load Word and Zero Indexed Load Word and Zero with Update Indexed	Book E
mbar	Memory Barrier	Book E
mcrxr crD	Move to Condition Register from Integer Exception Register	Book E
mfcr rD	Move From condition register	Book E
mfdcr rD,DCRN	Move From Device Control Register	Book E
mfmsr rD	Move From Machine State Register	Book E
mfspr rD,SPRN	Move From Special Purpose Register	Book E
msync	Memory Synchronize	Book E
mtcrf FXM,**r**S	Move to Condition Register Fields	Book E
mtdcr DCRN,**r**S	Move To Device Control Register	Book E
mtmsr rS	Move To Machine State Register	Book E
mtspr SPRN,**r**S	Move To Special Purpose Register	Book E
mulhw rD,**r**A,**r**B **mulhw.** rD,**r**A,**r**B	Multiply High Word	Book E

Table 257. Instructions listed by mnemonic (continued)

Mnemonic	Instruction	Reference
mulhwu rD,rA,rB **mulhwu.** rD,rA,rB	Multiply High Word Unsigned	*Book E*
mullw rD,rA,rB **mullw.** rD,rA,rB **mullwo** rD,rA,rB **mullwo.** rD,rA,rB	Multiply Low Word	*Book E*
nand rA,rS,rB **nand.** rA,rS,rB	NAND	*Book E*
neg rD,rA **neg.** rD,rA **nego** rD,rA **nego.** rD,rA	Negate	*Book E*
nor rA,rS,rB **nor.** rA,rS,rB	NOR	*Book E*
or rA,rS,rB **or.** rA,rS,rB	OR	*Book E*
orc rA,rS,rB **orc.** rA,rS,rB	OR with Complement	*Book E*
se_add rX,rY	Add	*Page -897*
se_addi rX,OIMM	Add Immediate	*Page -897*
se_andc rX,rY	AND with Complement	*Page -901*
se_andi rX,UI5	AND Immediate	*Page -901*
se_and[.] rX,rY	AND	*Page -901*
se_bc BO16,BI16,BD8	Branch Conditional	*Page -904*
se_bclri rX,UI5	Bit Clear	*Page -905*
se_bctr **se_bctrl**	Branch to Count Register Branch to Count Register & Link	*Page -905*
se_bgeni rX,UI5	Bit Generate	*Page -906*
se_blr **se_blrl**	Branch to Link Register Branch to Link Register & Link	*Page -907*
se_bmski rX,UI5	Bit Mask Generate	*Page -908*
se_bseti rX,UI5	Bit Set	*Page -909*
se_b BD8 **se_bl** BD8	Branch Branch & Link	*Page -910*
se_btsti rX,UI5	Bit Test Immediate	*Page -903*
se_cmp rX,rY	Compare	*Page -912*
se_cmph rX,rY	Compare Halfword	*Page -914*
se_cmphl rX,rY	Compare Halfword Logical	*Page -916*

Table 257. Instructions listed by mnemonic (continued)

Mnemonic	Instruction	Reference
se_cmpi rX,UI5	Compare Immediate	*Page -912*
se_cmpl rX,rY	Compare Logical	*Page -918*
se_cmpli rX,UI5	Compare Logical Immediate	*Page -918*
se_extsb rX	Extend Sign Byte	*Page -926*
se_extsh rX	Extend Sign Halfword	*Page -926*
se_extzb rX	Extend with Zeros Byte	*Page -927*
se_extzh rX	Extend with Zeros Halfword	*Page -927*
se_isync	Instruction Synchronize	*Page -929*
se_lbz rZ,SD4(rX)	Load Byte and Zero (16-bit form)	*Page -930*
se_lhz rZ,SD4(rX)	Load Halfword and Zero (16-bit form)	*Page -932*
se_li rX,UI7	Load Immediate	*Page -933*
se_lwz rZ,SD4(rX)	Load Word and Zero (16-bit form)	*Page -935*
se_mfar rX,arY	Move from Alternate Register	*Page -937*
se_mfctr rX	Move From Count Register	*Page -938*
se_mflr rX	Move From Link Register	*Page -939*
se_mr rX,rY	Move Register	*Page -940*
se_mtar arX,rY	Move to Alternate Register	*Page -941*
se_mtctr rX	Move To Count Register	*Page -942*
se_mtlr rX	Move To Link Register	*Page -943*
se_mullw rX,rY	Multiply Low Word	*Page -945*
se_neg rX	Negate	*Page -946*
se_not rX	NOT	*Page -947*
se_or rX,rY	OR	*Page -948*
se_slw rX,rY	Shift Left Word	*Page -955*
se_slwi rX,UI5	Shift Left Word Immediate	*Page -955*
se_sraw rX,rY	Shift Right Algebraic Word	*Page -956*
se_srawi rX,UI5	Shift Right Algebraic Word Immediate	*Page -956*
se_srw rX,rY	Shift Right Word	*Page -957*
se_srwi rX,UI5	Shift Right Word Immediate	*Page -957*
se_stb rZ,SD4(rX)	Store Byte (16-bit form)	*Page -958*
se_sth rZ,SD4(rX)	Store Halfword (16-bit form)	*Page -959*
se_stw rZ,SD4(rX)	Store Word (16-bit form)	*Page -961*
se_sub rX,rY	Subtract	*Page -962*
se_subf rX,rY	Subtract From	*Page -963*

Table 257. Instructions listed by mnemonic (continued)

Mnemonic	Instruction	Reference
se_subi rX,OIMM **se_subi.** rX,OIMM	Subtract Immediate	*Page -965*
slw rA,rS,rB **slw.** rA,rS,rB	Shift Left Word	*Book E*
sraw rA,rS,rB **sraw.** rA,rS,rB	Shift Right Algebraic Word	*Book E*
srawi rA,rS,SH **srawi.** rA,rS,SH	Shift Right Algebraic Word Immediate	*Book E*
srw rA,rS,rB **srw.** rA,rS,rB	Shift Right Word	*Book E*
stbx rS,rA,rB **stbux** rS,rA,rB	Store Byte Indexed Store Byte with Update Indexed	*Book E*
sthbrx rS,rA,rB	Store Halfword Byte-Reverse Indexed	*Book E*
sthx rS,rA,rB **sthux** rS,rA,rB	Store Halfword Indexed Store Halfword with Update Indexed	*Book E*
stwbrx rS,rA,rB	Store Word Byte-Reverse Indexed	*Book E*
stwcx. rS,rA,rB	Store Word Conditional Indexed	*Book E*
stwx rS,rA,rB **stwux** rS,rA,rB	Store Word Indexed Store Word with Update Indexed	*Book E*
subf rD,rA,rB **subf.** rD,rA,rB **subfo** rD,rA,rB **subfo.** rD,rA,rB	Subtract From	*Book E*
subfc rD,rA,rB **subfc.** rD,rA,rB **subfco** rD,rA,rB **subfco.** rD,rA,rB	Subtract From Carrying	*Book E*
tlbivax rA,rB	TLB Invalidate Virtual Address Indexed	*Book E*
tlbre	TLB Read Entry	*Book E*
tlbsx rA,rB	TLB Search Indexed	*Book E*
tlbsync	TLB Synchronize	*Book E*
tlbwe	TLB Write Entry	*Book E*
tw TO,rA,rB	Trap Word	*Book E*
wrtee rA	Write MSR External Enable	*Book E*
wrteei E	Write MSR External Enable Immediate	*Book E*
xor rA,rS,rB **xor.** rA,rS,rB	XOR	*Book E*

13 VLE instruction set

The VLE extension ISA is defined in the instruction pages in this chapter. Because of the various immediate field and displacement field calculations used in the VLE extension, a description of the less obvious ones precedes the actual instruction pages, and the instruction descriptions generally assume the appropriate calculation has been performed.

*Note: The instructions in this section are listed in order of the root instruction. For example, **e_cmpi** and **se_cmpi** are both listed under **cmpi**.*

13.1 Book E– and EIS-defined instructions

Table 258 lists instructions that are used by the VLE extension that are defined by Book E or the EIS. Full descriptions of those instructions can be found in the EREF.

Descriptions in this chapter indicate any limitations on the behavior of VLE instructions as compared to their Book E and EIS equivalents.

Table 258. Book E– and EIS-defined instructions listed by mnemonic

Mnemonic	Instruction	Defining architecture
add rD,**r**A,**r**B **add.** rD,**r**A,**r**B **addo** rD,**r**A,**r**B **addo.** rD,**r**A,**r**B	Add	Book E
addc rD,**r**A,**r**B **addc.** rD,**r**A,**r**B **addco** rD,**r**A,**r**B **addco.** rD,**r**A,**r**B	Add Carrying	Book E
adde rD,**r**A,**r**B **adde.** rD,**r**A,**r**B **addeo** rD,**r**A,**r**B **addeo.** rD,**r**A,**r**B	Add Extended	Book E
andc[.] **r**A,**r**S,**r**B	AND with Complement	Book E
and[.] **r**A,**r**S,**r**B	AND	Book E
cmp crD,L,**r**A,**r**B	Compare	Book E
cmpl crD,L,**r**A,**r**B	Compare Logical	Book E
cntlzw **r**A,**r**S **cntlzw.** **r**A,**r**S	Count Leading Zeros Word	Book E
dcba **r**A,**r**B	Data Cache Block Allocate	Book E
dcbf **r**A,**r**B	Data Cache Block Flush	Book E
dcbi **r**A,**r**B	Data Cache Block Invalidate	Book E
dcbst **r**A,**r**B	Data Cache Block Store	Book E
dcbt CT,**r**A,**r**B	Data Cache Block Touch	Book E

Table 258. Book E– and EIS-defined instructions listed by mnemonic (continued)

Mnemonic	Instruction	Defining architecture
dcbtls CT,rA,rB	Data Cache Block Touch and Lock Set	Book E
dcbtst CT,**r**A,**r**B	Data Cache Block Touch for Store	Book E
dcbz **r**A,**r**B	Data Cache Block set to Zero	Book E
divw **r**D,**r**A,**r**B **divw.** **r**D,**r**A,**r**B **divwo** **r**D,**r**A,**r**B **divwo.** **r**D,**r**A,**r**B	Divide Word	Book E
divwu **r**D,**r**A,**r**B **divwu.** **r**D,**r**A,**r**B **divwuo** **r**D,**r**A,**r**B **divwuo.** **r**D,**r**A,**r**B	Divide Word Unsigned	Book E
eqv **r**A,**r**S,**r**B **eqv.** **r**A,**r**S,**r**B	Equivalent	Book E
extsb **r**A,**r**S **extsb.** **r**A,**r**S	Extend Sign Byte	Book E
extsh **r**A,**r**S **extsh.** **r**A,**r**S	Extend Sign Halfword	Book E
e_srwi **r**A,**r**S,SH	Shift Right Word Immediate	Book E
icbi **r**A,**r**B	Instruction Cache Block Invalidate	Book E
icbt CT,**r**A,**r**B	Instruction Cache Block Touch	Book E
lbzx **r**D,**r**A,**r**B **lbzux** **r**D,**r**A,**r**B	Load Byte and Zero Indexed Load Byte and Zero with Update Indexed	Book E
lhax **r**D,**r**A,**r**B **lhaux** **r**D,**r**A,**r**B	Load Halfword Algebraic Indexed Load Halfword Algebraic with Update Indexed	Book E
lhbrx **r**D,**r**A,**r**B	Load Halfword Byte-Reverse Indexed	Book E
lhzx **r**D,**r**A,**r**B **lhzux** **r**D,**r**A,**r**B	Load Halfword and Zero Indexed Load Halfword and Zero with Update Indexed	Book E
lwarx **r**D,**r**A,**r**B	Load Word And Reserve Indexed	Book E
lwbrx **r**D,**r**A,**r**B	Load Word Byte-Reverse Indexed	Book E
lwzx **r**D,**r**A,**r**B **lwzux** **r**D,**r**A,**r**B	Load Word and Zero Indexed Load Word and Zero with Update Indexed	Book E
mbar	Memory Barrier	Book E
mcrxr **cr**D	Move to Condition Register from Integer Exception Register	Book E
mfcr **r**D	Move From condition register	Book E
mfdcr **r**D,DCRN	Move From Device Control Register	Book E
mfmsr **r**D	Move From Machine State Register	Book E
mfspr **r**D,SPRN	Move From Special Purpose Register	Book E

Table 258. Book E– and EIS-defined instructions listed by mnemonic (continued)

Mnemonic	Instruction	Defining architecture
msync	Memory Synchronize	Book E
mtcrf FXM,**r**S	Move to Condition Register Fields	Book E
mtdcr DCRN,rS	Move To Device Control Register	Book E
mtmsr rS	Move To Machine State Register	Book E
mtspr SPRN,rS	Move To Special Purpose Register	Book E
mulhw rD,**r**A,**r**B **mulhw.** **r**D,**r**A,**r**B	Multiply High Word	Book E
mulhwu rD,**r**A,**r**B **mulhwu.** **r**D,**r**A,**r**B	Multiply High Word Unsigned	Book E
mullw rD,**r**A,**r**B **mullw.** **r**D,**r**A,**r**B **mullwo r**D,**r**A,**r**B **mullwo.** **r**D,**r**A,**r**B	Multiply Low Word	Book E
nand rA,**r**S,**r**B **nand.** **r**A,**r**S,**r**B	NAND	Book E
neg rD,**r**A **neg.** **r**D,**r**A **nego r**D,**r**A **nego.** **r**D,**r**A	Negate	Book E
nor rA,**r**S,**r**B **nor.** **r**A,**r**S,**r**B	NOR	Book E
or rA,**r**S,**r**B **or.** **r**A,**r**S,**r**B	OR	Book E
orc rA,**r**S,**r**B **orc.** **r**A,**r**S,**r**B	OR with Complement	Book E
slw rA,**r**S,**r**B **slw.** **r**A,**r**S,**r**B	Shift Left Word	Book E
sraw rA,**r**S,**r**B **sraw.** **r**A,**r**S,**r**B	Shift Right Algebraic Word	Book E
srawi rA,**r**S,SH **srawi.** **r**A,**r**S,SH	Shift Right Algebraic Word Immediate	Book E
srw rA,**r**S,**r**B **srw.** **r**A,**r**S,**r**B	Shift Right Word	Book E
stbx rS,**r**A,**r**B **stbux r**S,**r**A,**r**B	Store Byte Indexed Store Byte with Update Indexed	Book E
sthbrx rS,**r**A,**r**B	Store Halfword Byte-Reverse Indexed	Book E
sthx rS,**r**A,**r**B **sthux r**S,**r**A,**r**B	Store Halfword Indexed Store Halfword with Update Indexed	Book E

Table 258. Book E– and EIS-defined instructions listed by mnemonic (continued)

Mnemonic	Instruction	Defining architecture
stwbrx rS,rA,rB	Store Word Byte-Reverse Indexed	Book E
stwcx. rS,rA,rB	Store Word Conditional Indexed	Book E
stwx rS,rA,rB **stwux** rS,rA,rB	Store Word Indexed Store Word with Update Indexed	Book E
subf rD,rA,rB **subf.** rD,rA,rB **subfo** rD,rA,rB **subfo.** rD,rA,rB	Subtract From	Book E
subfc rD,rA,rB **subfc.** rD,rA,rB **subfco** rD,rA,rB **subfco.** rD,rA,rB	Subtract From Carrying	Book E
tlbivax rA,rB	TLB Invalidate Virtual Address Indexed	Book E
tlbre	TLB Read Entry	Book E
tlbsx rA,rB	TLB Search Indexed	Book E
tlbsync	TLB Synchronize	Book E
tlbwe	TLB Write Entry	Book E
tw TO,rA,rB	Trap Word	Book E
wrtee rA	Write MSR External Enable	Book E
wrteei E	Write MSR External Enable Immediate	Book E
xor rA,rS,rB **xor.** rA,rS,rB	XOR	Book E
isel rD,rA,rB,crb	Integer Select	EIS

13.2 Immediate field and displacement field encodings

Table 259 shows encodings for immediate and displacement fields.

Table 259. Immediate field and displacement field encodings

Encoding	Description
BD15	Format used by 32-bit branch conditional class instructions. The BD15 field is an 15-bit signed displacement which 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	Format used by 32-bit branch class instructions. The BD24 field is an 24-bit signed displacement which 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	Format used by 16-bit branch and branch conditional class instructions. The BD8 field is an 8-bit signed displacement which 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.
D	Format used by some 32-bit load and store class instructions. The D field is a 16-bit signed displacement which is sign-extended to 32 bits, and then added to the base register to form a 32-bit EA.
D8	Format used by some 32-bit load and store class instructions. The D8 field is a 8-bit signed displacement which is sign-extended to 32 bits, and then added to the base register to form a 32-bit EA.
F, SCL,UI8 (SCI8 format)	Format used by some 32-bit arithmetic, compare, and logical instructions. The UI8 field is an 8-bit immediate value shifted left 0, 1, 2, or 3 byte positions according to the value of the SCL field. The remaining bits in the 32-bit word are filled with the value of the F field, and the resulting 32-bit value is used as one operand of the instruction. More formally, if SCL=0 then imm_value ← ^{24}F || UI8 else if SCL=1 then imm_value ← ^{16}F || UI8 || ^{8}F else if SCL=2 then imm_value ← ^{8}F || UI8 || ^{16}F else imm_value ← UI8 || ^{24}F
LI20	Format used by 32-bit **e_li** instruction. The LI20 field is a 20-bit signed displacement which is sign-extended to 32 bits for the **e_li** instruction.
OIM5	Format used by the 16-bit **se_addi**, **se_cmpli**, and **se_subi[.]** instructions. The OIM5 instruction field is a 5-bit value in the range 0–31 and is used to represent immediate values in the range 1–32, thus the binary encoding of 0b00000 represents an immediate value of 1, 0b00001 represents an immediate value of 2, and so on. In the instruction descriptions, OIMM represents the immediate value, not the OIM5 instruction field binary encoding.
SCI8 format	Refer to F, SCL,UI8 (SCI8 format)
SD4	Format used by 16-bit load and store class instructions. The SD4 field is a 4-bit unsigned immediate value zero-extended to 32 bits, shifted left according to the size of the operation, and then added to the base register to form a 32-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). For future double-word operations, the immediate is shifted left three bits (concatenated with 0b000).
SI (D format, I16A format)	Format used by certain 32-bit arithmetic type instructions. The SI field is a 16-bit signed immediate value sign-extended to 32 bits and used as one operand of the instruction. The instruction encoding differs between the D and I16A instruction formats

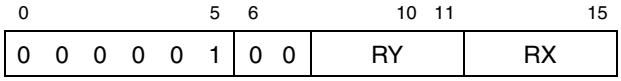
Table 259. Immediate field and displacement field encodings (continued)

Encoding	Description
UI (I16A, I16L formats)	Format used by certain 32-bit logical and arithmetic type instructions. The UI field is a 16-bit unsigned immediate value zero-extended to 32 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.
UI5	This format is used by some 16-bit Reg+Imm class instructions. The UI5 field is a 5-bit unsigned immediate value zero-extended to 32 bits and used as the second operand of the instruction. For other 16-bit Reg+Imm class instructions, the UI5 field is a 5-bit unsigned immediate value used to select a register bit in the range 0–31.
UI7	This format is used by the 16-bit **se_li** instructions. The UI7 field is a 7-bit unsigned immediate value zero-extended to 32 bits and used as the operand of the instruction.

_add*x* | VLE | User | **_add*x***

Add

se_add **r**X,**r**Y



$sum_{32:63} \leftarrow$ GPR(RX) + GPR(RY)

GPR(RX) $\leftarrow sum_{32:63}$

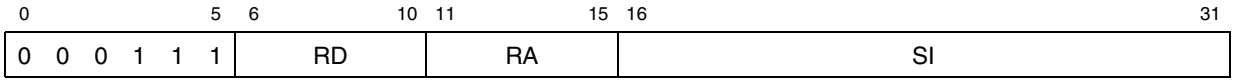
The sum of the contents of GPR(**r**X) and the contents of GPR(**r**Y) is placed into GPR(**r**X).

Special registers altered: None

_addi*x* | VLE | User | **_addi*x***

Add [2 operand] Immediate [Shifted] [and Record]

e_add16i **r**D,**r**A,SI



a ← GPR(RA)

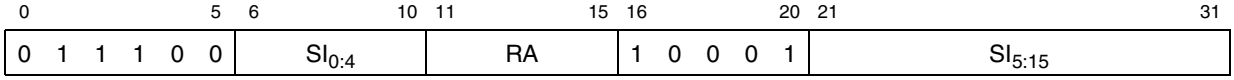
b ← EXTS(SI)

GPR(RD) ← a + b

The sum of the contents of GPR(**r**A) and the sign-extended value of field SI is placed into GPR(**r**D).

Special Registers Altered: None

e_add2i. **r**A,SI



SI ← $SI_{0:4}$ || $SI_{5:15}$

$sum_{32:63}$ ← GPR(RA) + EXTS(SI)

LT ← $sum_{32:63}$ < 0

GT ← $sum_{32:63}$ > 0

EQ ← $sum_{32:63}$ = 0

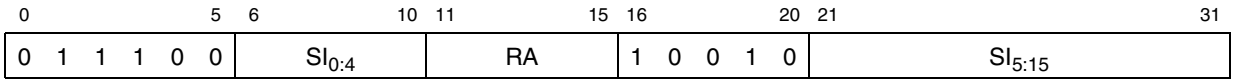
CR0 ← LT || GT || EQ || SO

GPR(RA) ← $sum_{32:63}$

The sum of the contents of GPR(**r**A) and the sign-extended value of SI is placed into GPR(**r**A).

Special Registers Altered: CR0

e_add2is **r**A,SI



SI ← $SI_{0:4}$ || $SI_{5:15}$

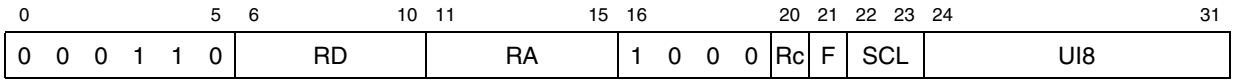
$sum_{32:63}$ ← GPR(RD) + (SI || $^{16}0$)

GPR(RA) ← $sum_{32:63}$

The sum of the contents of GPR(**r**A) and the value of SI concatenated with 16 zeros is placed into GPR(**r**A).

Special Registers Altered: None

e_addi **r**D,**r**A,SCI8 (Rc = 0)
e_addi. **r**D,**r**A,SCI8 (Rc = 1)



```
imm ← SCI8(F,SCL,UI8)
sum32:63 ←      GPR(RA) + imm
if Rc=1 then do
  LT  ← sum32:63 < 0
  GT  ← sum32:63 > 0
  EQ  ← sum32:63 = 0
  CR0 ← LT || GT || EQ || SO
GPR(RD) ← sum32:63
```

The sum of the contents of GPR(**r**A) and the value of SCI8 is placed into GPR(**r**D).

Special Registers Altered: CR0 (if Rc = 1)

se_addi **r**X,OIMM

0					5	6	7		11	12		15
0	0	1	0	0	0	0	OIM5(1)			RX		

1. OIMM = OIM5 +1

```
GPR(RX) ← GPR(RX) + (27 0 || OFFSET(OIM5))
```

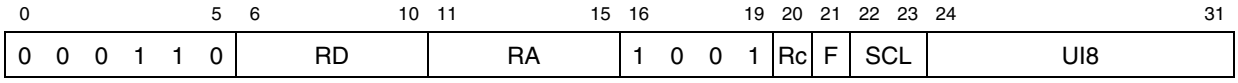
The sum of the contents of GPR(**r**X) and the zero-extended offset value of OIM5 (a final value in the range 1–32), is placed into GPR(**r**X).

Special Registers Altered: None

_addic*x* | VLE | User | **_addic*x***

Add Immediate Carrying [and Record]

e_addic **r**D,**r**A,SCI8 (Rc = 0)
e_addic. **r**D,**r**A,SCI8 (Rc = 1)



imm ← SCI8(F,SCL,UI8)
$carry_{32:63}$ ← Carry(GPR(RA) + imm)

$sum_{32:63}$ ← GPR(RA) + imm

if Rc=1 then do

LT ← $sum_{32:63}$ < 0

GT ← $sum_{32:63}$ > 0

EQ ← $sum_{32:63}$ = 0

CR0 ← LT || GT || EQ || SO

GPR(RD) ← $sum_{32:63}$
CA ← $carry_{32}$

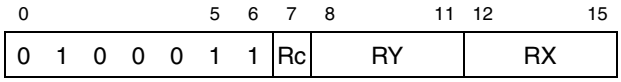
The sum of the contents of GPR(**r**A) and the value of SCI8 is placed into GPR(**r**D).

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

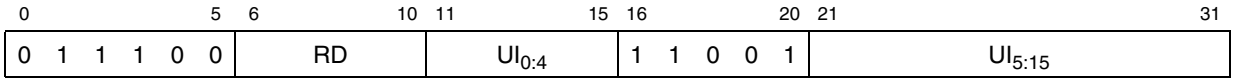
_and*x* | VLE | User | **_and*x***

AND [2 operand] [Immediate | with Complement] [and Record]

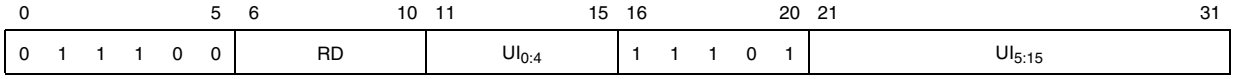
se_and **r**X,**r**Y (Rc = 0)
se_and. **r**X,**r**Y (Rc = 1)



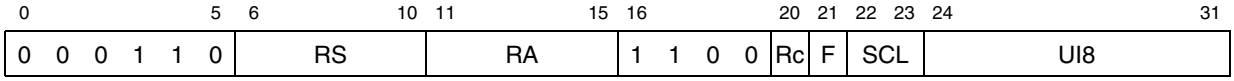
e_and2i. **r**D,UI



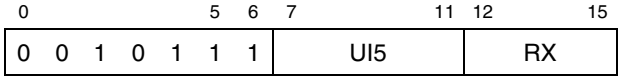
e_and2is. **r**D,UI



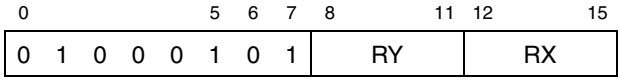
e_andi **r**A,**r**S,SCI8 (Rc = 0)
e_andi. **r**A,**r**S,SCI8 (Rc = 1)



se_andi **r**X,UI5



se_andc **r**X,**r**Y



if ‘e_andi[.]’ then b ← SCI8(F,SCL,UI8)
if ‘se_andi’ then b ← UI5
if ‘se_and[.]’ then b ← GPR(RY)
if ‘se_andc’ then b ← ¬GPR(RY)
if ‘e_and2i.’ then b ← $^{16}0$ || $UI_{0:4}$ || $UI_{5:15}$
if ‘e_and2is.’ then b ← $UI_{0:4}$ || $UI_{5:15}$ || $^{16}0$
$result_{32:63}$ ← GPR(RS or RD or RX) & b
if Rc=1 then do
  LT ← $result_{32:63}$ < 0
  GT ← $result_{32:63}$ > 0
  EQ ← $result_{32:63}$ = 0
  CR0 ← LT || GT || EQ || SO
if ‘se_and[ci]’ then GPR(RX) ← $result_{32:63}$ else GPR(RA or RD) ← $result_{32:63}$

For **e_andi**[**.**], the contents of GPR(**r**S) are ANDed with the value of SCI8.

For **e_and2i.**, the contents of GPR(**r**D) are ANDed with $^{16}0$ || UI.

For **e_and2is.**, the contents of GPR(**r**D) are ANDed with UI || $^{16}0$.

For **se_andi**, the contents of GPR(**r**X) are ANDed with the value of UI5.

For **se_and**[**.**], the contents of GPR(**r**X) are ANDed with the contents of GPR(**r**Y).

For **se_andc**, the contents of GPR(**r**X) are ANDed with the one's complement of the contents of GPR(**r**Y).

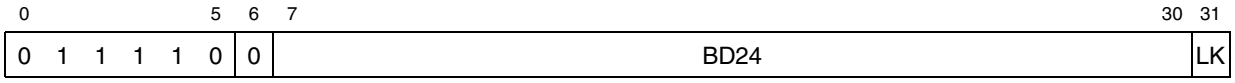
The result is placed into GPR(**r**A) or GPR(**r**X) (**se_and**[**ic**][**.**])

Special Registers Altered: CR0 (if Rc = 1)

_bx | VLE | User | **_bx**

Branch [and Link]

e_b BD24 (LK = 0)
e_bl BD24 (LK = 1)



a ← CIA

NIA ← (a + EXTS(BD24||0b0))$_{32:63}$

if LK=1 then LR ← CIA + 4

Let the BTEA be calculated as follows:

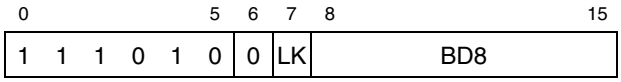
- For **e_b[l]**, let BTEA be the sum of the CIA and the sign-extended value of the BD24 instruction field concatenated with 0b0.

The BTEA is the address of the next instruction to be executed.

If LK = 1, the sum CIA+4 is placed into the LR.

Special Registers Altered: LR (if LK = 1)

se_b BD8 (LK = 0)
se_bl BD8 (LK = 1)



a ← CIA

NIA ← (a + EXTS(BD8||0b0))$_{32:63}$

if LK=1 then LR ← CIA + 2

Let the BTEA be calculated as follows:

- For **se_b[l]**, let BTEA be the sum of the CIA and the sign-extended value of the BD8 instruction field concatenated with 0b0.

The BTEA is the address of the next instruction to be executed.

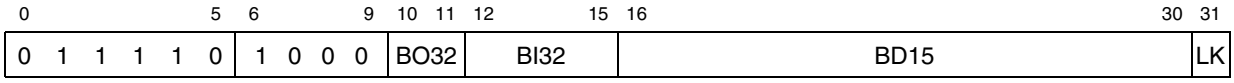
If LK = 1, the sum CIA+2 is placed into the LR.

Special Registers Altered: LR (if LK = 1)

_bcx | VLE | User | **_bcx**

Branch Conditional [and Link]

e_bc BO32,BI32,BD15 (LK = 0)
e_bcl BO32,BI32,BD15 (LK = 1)



if $BO32_0$ then $CTR_{32:63} \leftarrow CTR_{32:63} - 1$

ctr_ok $\leftarrow \neg BO32_0 \mid ((CTR_{32:63} \neq 0) \oplus BO32_1)$

cond_ok $\leftarrow BO32_0 \mid (CR_{BI32+32} \equiv BO32_1)$

if ctr_ok & cond_ok then

NIA $\leftarrow (CIA + EXTS(BD15 \parallel 0b0))_{32:63}$

else NIA $\leftarrow CIA + 4$

if LK=1 then LR $\leftarrow CIA + 4$

Let the BTEA be calculated as follows:

- For **e_bc**[**l**], let BTEA be the sum of the CIA and the sign-extended value of the BD15 instruction field concatenated with 0b0.

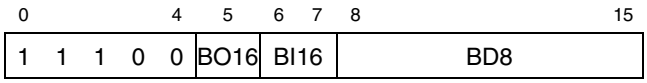
BO32 specifies any conditions that must be met for the branch to be taken, as defined in *Chapter 12.2.2: Branch instructions on page 864*." The sum BI32+32 specifies the CR bit. Only CR[32–47] may be specified.

If the branch conditions are met, the BTEA is the address of the next instruction to be executed.

If LK = 1, the sum CIA + 4 is placed into the LR.

Special Registers Altered: CTR (if $BO32_0 = 1$)
LR (if LK = 1)

se_bc BO16,BI16,BD8



cond_ok $\leftarrow (CR_{BI16+32} \equiv BO16)$

if cond_ok then

NIA $\leftarrow (CIA + EXTS(BD8 \parallel 0b0))_{32:63}$

else NIA $\leftarrow CIA + 2$

Let the BTEA be calculated as follows:

- For **se_bc**, BTEA is the sum of the CIA and the sign-extended value of the BD8 instruction field concatenated with 0b0.

BO16 specifies any conditions that must be met for the branch to be taken, as defined in *Chapter 12.2.2: Branch instructions on page 864*." The sum BI16+32 specifies CR bit; only CR[32–35] may be specified. If the branch conditions are met, the BTEA is the address of the next instruction to be executed.

Special Registers Altered: None

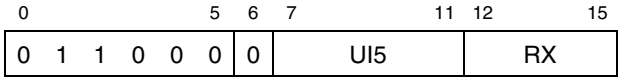
_bclri

VLE	User

_bclri

Bit Clear Immediate

se_bclri **r**X,UI5



a ← UI5
b ← $^{a}1$ || 0 || $^{31-a}1$

result$_{32:63}$ ← GPR(RX) & b

GPR(RX) ← result$_{32:63}$

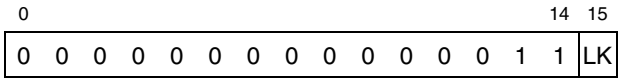
For **se_bclri**, the bit of GPR(**r**X) specified by the value of UI5 is cleared and all other bits in GPR(**r**X) remain unaffected.

Special Registers Altered: None

_bctr*x* | VLE | User | **_bctr*x***

Branch to Count Register [and Link]

se_bctr (LK = 0)
se_bctrl (LK = 1)



NIA ← $CTR_{32:62}$ || 0b0

if LK=1 then LR ← CIA + 2

Let the BTEA be calculated as follows:

- For **se_bctr**[**l**], let BTEA be bits 32–62 of the contents of the CTR concatenated with 0b0.

The BTEA is the address of the next instruction to be executed.

If LK = 1, the sum CIA + 2 is placed into the LR.

Special Registers Altered: LR (if LK = 1)

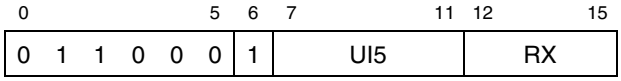
_bgeni

VLE	User

_bgeni

Bit Generate Immediate

se_bgeni **r**X,UI5



a ← UI5

b ← $^{a}0$ || 1 || $^{31-a}0$

GPR(RX) ← b

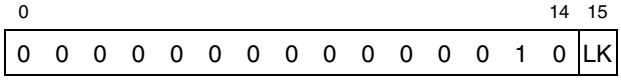
For **se_bgeni**, a constant value consisting of a single ’1’ bit surrounded by ’0’s is generated and the value is placed into GPR(**r**X). The position of the ’1’ bit is specified by the UI5 field.

Special Registers Altered: None

_blr*x* | VLE | User | **_blr*x***

Branch to Link Register [and Link]

se_blr (LK = 0)
se_blrl (LK = 1)



NIA ← $LR_{32:62}$ || 0b0

if LK=1 then LR ← CIA + 2

Let the BTEA be calculated as follows:

- For **se_blr**[**l**], let BTEA be bits 32–62 of the contents of the LR concatenated with 0b0.

The BTEA is the address of the next instruction to be executed.

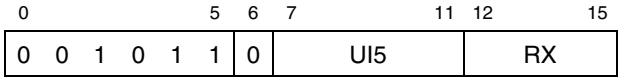
If LK = 1, the sum CIA + 2 is placed into the LR.

Special Registers Altered: LR (if LK = 1)

_bmaski | VLE | User | **_bmaski**

Bit Mask Generate Immediate

se_bmaski **r**X,UI5



a ← UI5

if a = 0 then b ← $^{32}1$ else b ← $^{32-a}0 \,||\, {}^{a}1$

GPR(RX) ← b

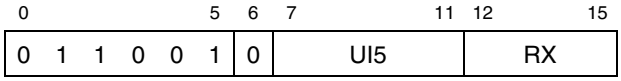
For **se_bmaski**, a constant value consisting of a mask of low-order ’1’ bits that is zero-extended to 32 bits is generated, and the value is placed into GPR(**r**X). The number of low-order ’1’ bits is specified by the UI5 field. If UI5 is 0b00000, a value of all ’1’s is generated

Special Registers Altered: None

_bseti | VLE | User | **_bseti**

Bit Set Immediate

se_bseti **r**X,UI5



a ← UI5
b ← $^{a}0$ || 1 || $^{31-a}0$

result$_{32:63}$ ← GPR(RX) | b

GPR(RX) ← result$_{32:63}$

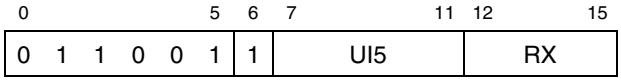
For **se_bseti**, the bit of GPR(**r**X) specified by the value of UI5 is set, and all other bits in GPR(**r**X) remain unaffected.

Special Registers Altered: None

_btsti | VLE | User | **_btsti**

Bit Test Immediate

se_btsti **r**X,UI5



a ← UI5
b ← $^{a}0$ || 1 || $^{31-a}0$

c ← GPR(RX) & b

if c = $^{32}0$ then d ← 0b001 else d ← 0b010

$CR_{0:3}$ ← d || XER_{SO}

For **se_btsti**, the bit of GPR(**r**X) specified by the value of UI5 is tested for equality to '1'. The result of the test is recorded in the CR. EQ is set if the tested bit is clear, LT is cleared, and GT is set to the inverse value of EQ.

Special Registers Altered: CR[0–3]

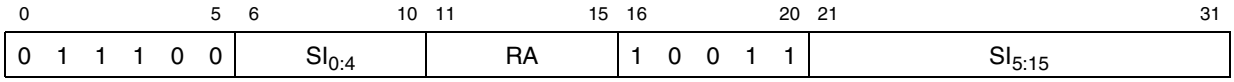
_cmp

VLE	User

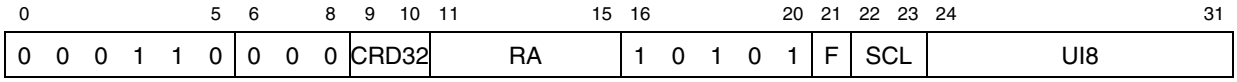
_cmp

Compare [Immediate]

e_cmp16i **r**A,SI



e_cmpi **cr**D32,**r**A,SCI8



a ← GPR(RA)$_{32:63}$

if ‘e_cmpi’ then b ← SCI8(F,SCL,UI8)

if ‘e_cmp16i’ then b ← EXTS(SI$_{0:4}$ || SI$_{5:15}$)

if a < b then c ← 0b100

if a > b then c ← 0b010

if a = b then c ← 0b001

if ‘e_cmpi’ then CR$_{4\times CRD32+32:4\times CRD32+35}$ ← c || XER$_{SO}$ // only CR0-CR3

if ‘e_cmp16i’ then CR$_{32:35}$ ← c || XER$_{SO}$ // only CR0

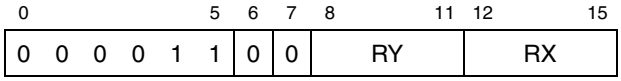
If **e_cmpi**, GPR(**r**A) contents are compared with the value of SCI8, treating operands as signed integers.

If **e_cmp16i**, GPR(**r**A) contents are compared with the sign-extended value of the SI field, treating operands as signed integers.

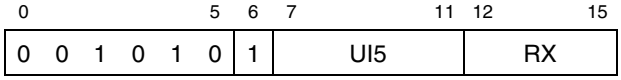
The result of the comparison is placed into CR field **cr**D (**cr**D32). For **e_cmpi**, only CR0–CR3 may be specified. For **e_cmp16i**, only CR0 may be specified.

Special Registers Altered: CR field **cr**D (**cr**D32) (CR0 for **e_cmp16i**)

se_cmp **r**X,**r**Y



se_cmpi **r**X,UI5



a ← GPR(RX)$_{32:63}$

if ‘se_cmpi’ then b ← ^{27}0 || UI5

if ‘se_cmp’ then b ← GPR(RY)$_{32:63}$

if a < b then c ← 0b100

if a > b then c ← 0b010

if a = b then c ← 0b001

CR$_{0:3}$ ← c || XER$_{SO}$

If **se_cmp**, the contents of GPR(**r**X) are compared with the contents of GPR(**r**Y), treating the operands as signed integers. The result of the comparison is placed into CR field 0.

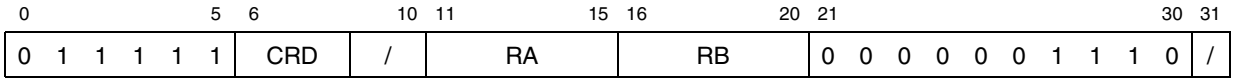
If **se_cmpi**, the contents of GPR(**r**X) are compared with the value of the zero-extended UI5 field, treating the operands as signed integers. The result of the comparison is placed into CR field 0.

Special Registers Altered: CR[0–3]

_cmph | VLE | User | **_cmph**

Compare Halfword [Immediate]

e_cmph **cr**D,**r**A,**r**B



a ← EXTS(GPR(RA)$_{48:63}$)

b ← EXTS(GPR(RB)$_{48:63}$)

if a < b then c ← 0b100

if a > b then c ← 0b010

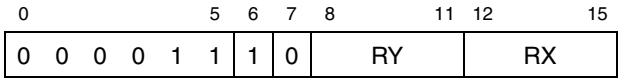
if a = b then c ← 0b001

$CR_{4\times CRD+32:4\times CRD+35}$ ← c || XER_{SO}

For **e_cmph**, the contents of the low-order 16 bits of GPR(**r**A) and GPR(**r**B) are compared, treating the operands as signed integers. The result of the comparison is placed into CR field CRD.

Special Registers Altered: CR field CRD

se_cmph **r**X,**r**Y



a ← EXTS(GPR(RX)$_{48:63}$)

b ← EXTS(GPR(RY)$_{48:63}$)

if a < b then c ← 0b100

if a > b then c ← 0b010

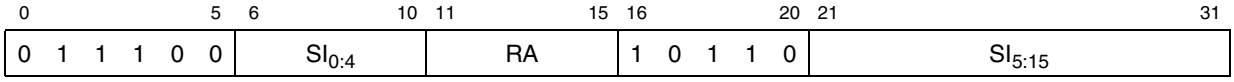
if a = b then c ← 0b001

$CR_{0:3}$ ← c || XER_{SO}

For **se_cmph**, the contents of the low-order 16 bits of GPR(**r**X) and GPR(**r**Y) are compared, treating the operands as signed integers. The result of the comparison is placed into CR field 0.

Special Registers Altered: CR[0–3]

e_cmph16i **r**A,SI



a ← EXTS(GPR(RA)$_{48:63}$)

b ← EXTS($SI_{0:4}$ || $SI_{5:15}$)

if a < b then c ← 0b100

if a > b then c ← 0b010

if a = b then c ← 0b001

$CR_{32:35}$ ← c || XER_{SO} // only CR0

The contents of the lower 16-bits of GPR(**r**A) are sign-extended and compared with the sign-extended value of the SI field, treating the operands as signed integers.

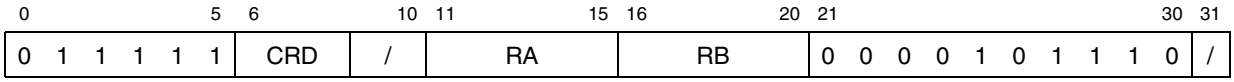
The result of the comparison is placed into CR0.

Special Registers Altered: CR0

_cmphl | VLE | User | **__cmphl _**

Compare Halfword Logical [Immediate]

e_cmphl **cr**D,**r**A,**r**B



a ← EXTZ(GPR(RA)$_{48:63}$)

b ← EXTZ(GPR(RB)$_{48:63}$)

if a < b then c ← 0b100

if a > b then c ← 0b010

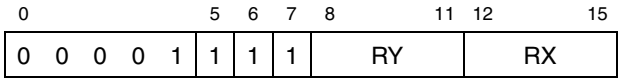
if a = b then c ← 0b001

$CR_{4\times CRD+32:4\times CRD+35}$ ← c || XER_{SO}

For **e_cmphl**, the contents of the low-order 16 bits of GPR(**r**A) and GPR(**r**B) are compared, treating the operands as unsigned integers. The result of the comparison is placed into CR field CRD.

Special Registers Altered: CR field CRD

se_cmphl **r**X,**r**Y



a ← GPR(RX)$_{48:63}$

b ← GPR(RY)$_{48:63}$

if a < b then c ← 0b100

if a > b then c ← 0b010

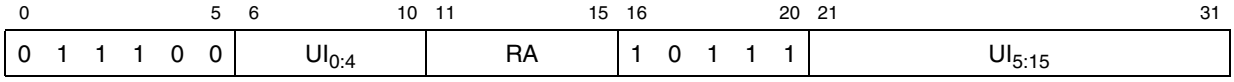
if a = b then c ← 0b001

$CR_{0:3}$ ← c || XER_{SO}

For **se_cmphl**, the contents of the low-order 16 bits of GPR(**r**X) and GPR(**r**Y) are compared, treating the operands as unsigned integers. The result of the comparison is placed into CR field 0.

Special Registers Altered: CR[0–3]

e_cmphl16i **r**A,UI



a ← $^{16}0$ || GPR(RA)$_{48:63)}$

b ← $^{16}0$ || $UI_{0:4}$ || $UI_{5:15}$

if a < b then c ← 0b100

if a > b then c ← 0b010

if a = b then c ← 0b001

$CR_{32:35}$ ← c || XER_{SO} // only CR0

The contents of the lower 16-bits of GPR(**r**A) are zero-extended and compared with the zero-extended value of the UI field, treating the operands as unsigned integers.

The result of the comparison is placed into CR0.

Special Registers Altered: CR0

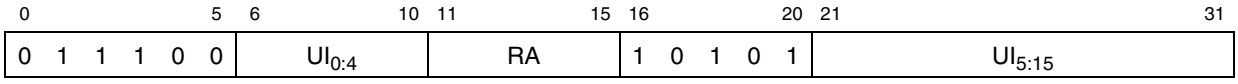
_cmpl

VLE	User

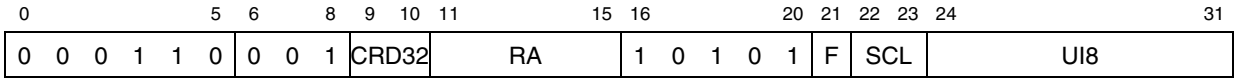
__cmpl _

Compare Logical [Immediate]

e_cmpl16i **r**A,UI



e_cmpli **cr**D32,**r**A,SCI8



a ← GPR(RA)$_{32:63}$

if ‘e_cmpli’ then b ← SCI8(F,SCL,UI8)

if ‘e_cmpl16i’ then b ← ^{16}0 || UI$_{0:4}$ || UI$_{5:15}$

if a <$_u$ b then c ← 0b100

if a >$_u$ b then c ← 0b010

if a = b then c ← 0b001

if ‘e_cmpli’ then CR$_{4\times CRD32+32:4\times CRD32+35}$ ← c || XER$_{SO}$ // only CR0-CR3

if ‘e_cmp16i’ then CR$_{32:35}$ ← c || XER$_{SO}$ // only CR0

If **e_cmpi**, the contents of bits 32–63 of GPR(**r**A) are compared with the value of SCI8, treating the operands as unsigned integers.

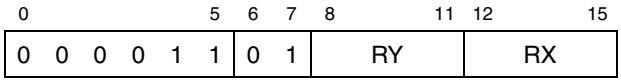
L must be 0 for 32-bit implementations

If **e_cmpl16i**, the contents of GPR(**r**A) are compared with the zero-extended value of the UI field, treating the operands as unsigned integers.

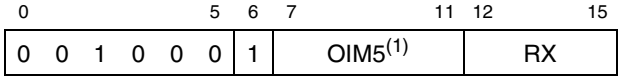
The result of the comparison is placed into CR field CRD (CRD32). For **e_cmpli**, only CR0–CR3 may be specified. For **e_cmpl16i**, only CR0 may be specified.

Special Registers Altered: CR field CRD (CRD32) (CR0 for **e_cmpl16i**)

se_cmpl **r**X,**r**Y



se_cmpli **r**X,OIMM



1. OIMM = OIM5 +1

a ← GPR(RX)$_{32:63}$

if ‘se_cmpli’ then b ← ^{27}0 || OFFSET(OIM5)

if ‘se_cmpl’ then b ← GPR(RY)$_{32:63}$

if a <$_u$ b then c ← 0b100

if a >$_u$ b then c ← 0b010

if a = b then c ← 0b001

$CR_{0:3} \leftarrow c \,||\, XER_{SO}$

If **se_cmpl**, the contents of GPR(**r**X) are compared with the contents of GPR(**r**Y), treating the operands as unsigned integers. The result of the comparison is placed into CR field 0.

If **se_cmpli**, the contents of GPR(**r**X) are compared with the value of the zero-extended offset value of the OIM5 field (a final value in the range 1–32), treating the operands as unsigned integers. The result of the comparison is placed into CR field 0.

Special Registers Altered: CR[0–3]

_crand

VLE	User

__crand _

Condition Register AND

e_crand **crb**D,**crb**A,**crb**B

0					5	6			10	11			15	16			20	21									30	31
0	1	1	1	1	1	CRBD				CRBA				CRBB				0	1	0	0	0	0	0	0	0	1	/

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

The content of bit CRBA+32 of the CR is ANDed with the content of bit CRBB+32 of the CR, and the result is placed into bit CRBD+32 of the CR.

Special Registers Altered: CR

Condition Register AND with Complement

e_crandc **crb**D,**crb**A,**crb**B

0					5	6			10	11			15	16			20	21									30	31
0	1	1	1	1	1	CRBD				CRBA				CRBB				0	0	1	0	0	0	0	0	0	1	/

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

The content of bit CRBA+32 of the CR is ANDed with the one's complement of the content of bit CRBB+32 of the CR, and the result is placed into bit CRBD+32 of the CR.

Special Registers Altered: CR

CR Equivalent

e_creqv **crb**D,**crb**A,**crb**B

0					5	6			10	11			15	16			20	21									30	31
0	1	1	1	1	1	CRBD				CRBA				CRBB				0	1	0	0	1	0	0	0	0	1	/

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

The content of bit CRBA+32 of the CR is XORed with the content of bit CRBB+32 of the CR, and the one's complement of result is placed into bit CRBD+32 of the CR.

Special Registers Altered: CR

_crnand

VLE	User

__crnand _

Condition Register NAND

e_crnand **crb**D,**crb**A,**crb**B

0					5	6			10	11			15	16			20	21									30	31
0	1	1	1	1	1	CRBD				CRBA				CRBB				0	0	1	1	1	0	0	0	0	1	/

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

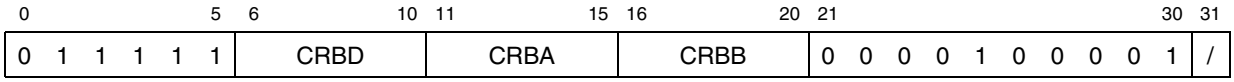
The content of bit CRBA+32 of the CR is ANDed with the content of bit CRBB+32 of the CR, and the one's complement of the result is placed into bit CRBD+32 of the CR.

Special Registers Altered: CR

_crnor | VLE | User | _ **crnor**

Condition Register NOR

e_crnor **crb**D,**crb**A,**crb**B



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

The content of bit CRBA+32 of the CR is ORed with the content of bit CRBB+32 of the CR, and the one's complement of the result is placed into bit CRBD+32 of the CR.

Special Registers Altered: CR

_cror

VLE	User

_cror

Condition Register OR

e_cror **crb**D,**crb**A,**crb**B

0					5	6			10	11			15	16			20	21									30	31
0	1	1	1	1	1	CRBD				CRBA				CRBB				0	1	1	1	0	0	0	0	0	1	/

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

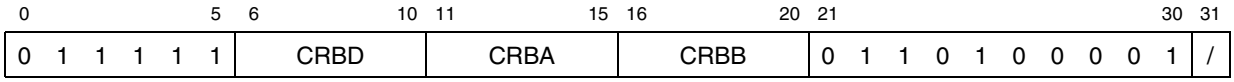
The content of bit CRBA+32 of the CR is ORed with the content of bit CRBB+32 of the CR, and the result is placed into bit CRBD+32 of the CR.

Special Registers Altered: CR

_cror | VLE | User | **__cror _**

Condition Register OR with Complement

e_crorc **crb**D,**crb**A,**crb**B



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

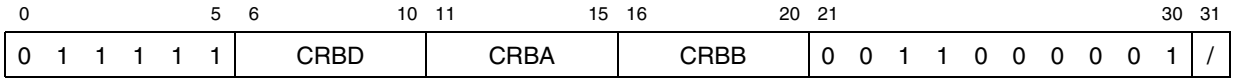
The content of bit CRBA+32 of the CR is ORed with the one's complement of the content of bit CRBB+32 of the CR, and the result is placed into bit CRBD+32 of the CR.

Special Registers Altered: CR

_crxor | VLE | User | **__crxor _**

Condition Register XOR

e_crxor **crb**D,**crb**A,**crb**B



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

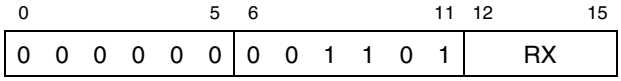
The content of bit CRBA+32 of the CR is XORed with the content of bit CRBB+32 of the CR, and the result is placed into bit CRBD+32 of the CR.

Special Registers Altered: CR

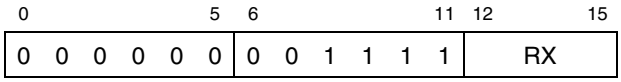
_extsb*x* | VLE | User | **_extsb*x***

Extend Sign (Byte | Halfword)

se_extsb **r**X



se_extsh0. **r**X



if se_extsb then n ← 56

if se_extsh then n ← 48

if ‘extsw’ then n ← 32

if Rc=1 then do

LT ← $GPR(RS)_{n:63} < 0$

GT ← $GPR(RS)_{n:63} > 0$

EQ ← $GPR(RS)_{n:63} = 0$

CR0 ← LT || GT || EQ || SO

s ← $GPR(RS \text{ or } RX)_n$

GPR(RA or RX) ← ^{n-32}s || $GPR(RS \text{ or } RX)_{n:63}$

For **se_extsb**, the contents of bits 56–63 of GPR(**r**X) are placed into bits 56–63 of GPR(**r**X). Bit 56 of the contents of GPR(**r**X) is copied into bits 32–55 of GPR(**r**X).

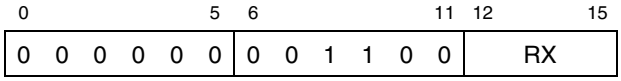
For **se_extsh**, the contents of bits 48–63 of GPR(**r**X) are placed into bits 48–63 of GPR(**r**X). Bit 48 of the contents of GPR(**r**X) is copied into bits 32–47 of GPR(**r**X).

Special Registers Altered: CR0 (if Rc=1)

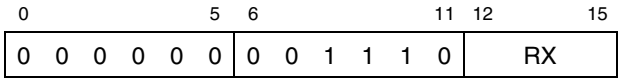
_extz*x* | VLE | User | **extz*x***

Extend Zero (Byte | Halfword)

se_extzb **r**X



se_extzh **r**X



if ‘se_extzb’ then n ← 56

if ‘se_extzh’ then n ← 48

GPR(RX) ← $^{n-32}0$ || $GPR(RX)_{n:63}$

For **se_extzb**, the contents of bits 56–63 of GPR(**r**X) are placed into bits 56–63 of GPR(**r**X). Bits 32–55 of GPR(**r**X) are cleared.

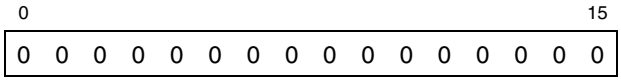
For **se_extzh**, the contents of bits 48–63 of GPR(**r**X) are placed into bits 48–63 of GPR(**r**X). Bits 32–47 of GPR(**r**X) are cleared.

Special Registers Altered: None

_illegal | VLE | User | **_illegal**

Illegal

se_illegal



SRR1 ← MSR

SRR0 ← CIA

NIA ← $IVPR_{32:47}$ || $IVOR6_{48:59}$ || 0b0000

$MSR_{WE,EE,PR,IS,DS,FP,FE0,FE1}$ ← 0b0000_0000

se_illegal is used to request an illegal instruction exception. A program interrupt is generated. The contents of the MSR are copied into SRR1 and the address of the **se_illegal** instruction is placed into SRR0.

MSR[WE,EE,PR,IS,DS,FP,FE0,FE1] are cleared.

The interrupt causes the next instruction to be fetched from address IVPR[32–47]||IVOR6[48–59]||0b0000

This instruction is context synchronizing.

Special Registers Altered: SRR0 SRR1 MSR[WE,EE,PR,IS,DS,FP,FE0,FE1]

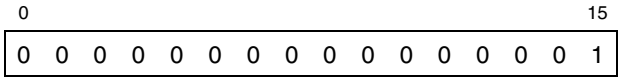
_isync

VLE	User

_isync

Instruction Synchronize

se_isync



The **se_isync** instruction provides an ordering function for the effects of all instructions executed by the processor executing the **se_isync** instruction. 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 causes any prefetched instructions to be discarded, with the effect that subsequent instructions are fetched and executed in the context established by the instructions preceding the **se_isync** instruction.

The **se_isync** instruction may complete before memory accesses associated with instructions preceding the **se_isync** instruction have been performed.

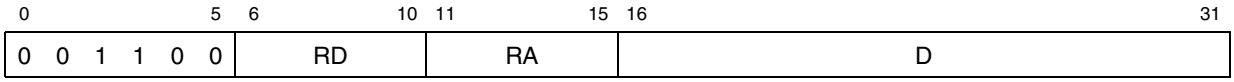
This instruction is context synchronizing (see Book E). It has identical semantics to Book E **isync**, just a different encoding.

Special Registers Altered: None

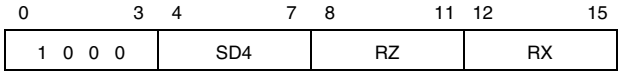
_lbz*x* | VLE | User | **_lbz*x***

Load Byte and Zero [with Update] [Indexed]

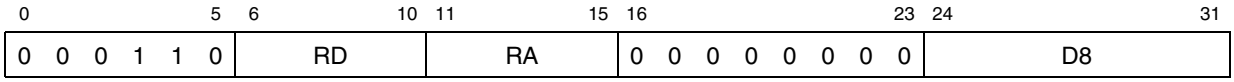
e_lbz **r**D,D(**r**A) (D-mode)



se_lbz **r**Z,SD4(**r**X) (SD4-mode)



e_lbzu **r**D,D8(**r**A) (D8-mode)



if (RA=0 & !se_lbz) then a ← $^{32}0$ else a ← GPR(RA or RX)

if D-mode then EA ← $(a + EXTS(D))_{32:63}$

if D8-mode then EA ← $(a + EXTS(D8))_{32:63}$

if SD4-mode then EA ← $(a + (^{28}0 \,||\, SD4))_{32:63}$

GPR(RD or RZ) ← $^{24}0$ || MEM(EA,1)

if e_lbzu then GPR(RA) ← EA

Let the EA be calculated as follows:

- For **e_lbz** and **e_lbzu**, let EA be the sum of the contents of GPR(**r**A), or 32 0s if **r**A = 0, and the sign-extended value of the D or D8 instruction field.
- For **se_lbz**, let EA be the sum of the contents of GPR(**r**X) and the zero-extended value of the SD4 instruction field.

The byte in memory addressed by EA is loaded into bits 56–63 of GPR(**r**D or **r**Z). Bits 32–55 of GPR(**r**D or **r**Z) are cleared.

If **e_lbzu**, EA is placed into GPR(**r**A).

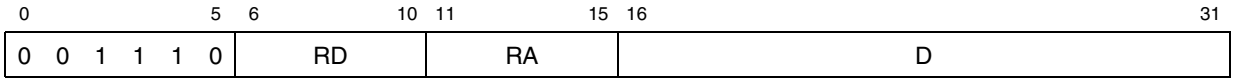
If **e_lbzu** and **r**A = 0 or **r**A = **r**D, the instruction form is invalid.

Special Registers Altered: None

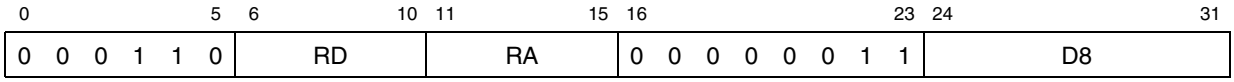
_lha*x* | VLE | User | **_lha*x***

Load Halfword Algebraic [with Update] [Indexed]

e_lha **r**D,D**(r**A**)** (D-mode)



e_lhau **r**D,D8**(r**A**)** (D8-mode)



if RA=0 then a ← $^{32}0$ else a ← GPR(RA)

if D-mode then EA ← $(a + EXTS(D))_{32:63}$

if D8-mode then EA ← $(a + EXTS(D8))_{32:63}$

$GPR(RD) \leftarrow EXTS(MEM(EA,2))_{32:63}$

if e_lhau then GPR(RA) ← EA

Let the EA be calculated as follows:

- For **e_lha** and **e_lhau**, let EA be the sum of the contents of GPR(**r**A), or 32 0s if **r**A = 0, and the sign-extended value of the D or D8 instruction field.

The half word in memory addressed by EA is loaded into bits 48–63 of GPR(**r**D). Bits 32–47 of GPR(**r**D) are filled with a copy of bit 0 of the loaded half word.

If **e_lhau**, EA is placed into GPR(**r**A).

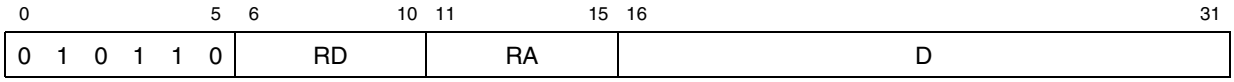
If **e_lhau** and **r**A = 0 or **r**A = **r**D, the instruction form is invalid.

Special Registers Altered: None

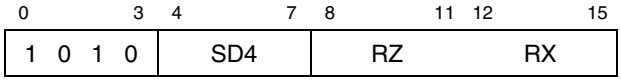
_lhz*x* | VLE | User | **_lhz*x***

Load Halfword and Zero [with Update] [Indexed]

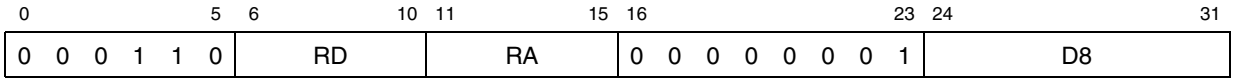
e_lhz **r**D,D(**r**A) (D-mode)



se_lhz **r**Z,SD4(**r**X) (SD4-mode)



e_lhzu **r**D,D8(**r**A) (D8-mode)



if (RA=0 & !se_lhz) then a ← $^{32}0$ else a ← GPR(RA or RX)

if D-mode then EA ← $(a + EXTS(D))_{32:63}$

if D8-mode then EA ← $(a + EXTS(D8))_{32:63}$

if SD4-mode then EA ← $(a + (^{27}0 \,||\, SD4 \,||\, 0))_{32:63}$

GPR(RD or RZ) ← $^{16}0$ || MEM(EA,2)

if e_lhzu then GPR(RA) ← EA

Let the EA be calculated as follows:

- For **e_lhz** and **e_lhzu**, let EA be the sum of the contents of GPR(**r**A), or 32 0s if **r**A = 0, and the sign-extended value of the D or D8 instruction field.
- For **se_lhz** let EA be the sum of the contents of GPR(**r**X) and the zero-extended value of the SD4 instruction field shifted left by 1 bit.

The half word in memory addressed by EA is loaded into bits 48–63 of GPR(**r**D). Bits 32–47 of GPR(**r**D) are cleared.

If **e_lhzu**, EA is placed into GPR(**r**A).

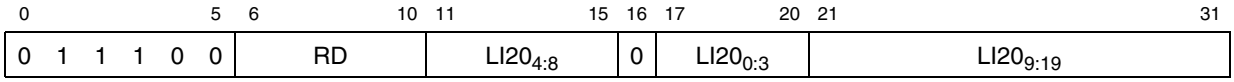
If **e_lhzu** and **r**A = 0 or **r**A = **r**D, the instruction form is invalid.

Special Registers Altered: None

_li*x* | VLE | User | **_li*x***

Load Immediate [Shifted]

e_li **r**D,LI20 (LI20-mode)

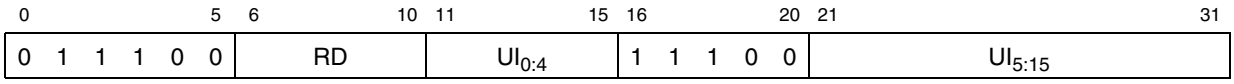


$LI20 \leftarrow LI20_{0:3} \,||\, LI20_{4:8} \,||\, LI20_{9:19}$
$GPR(RD) \leftarrow EXTS(LI20)$

For **e_li**, the sign-extended LI20 field is placed into GPR(**r**D).

Special Registers Altered: None

e_lis **r**D,UI

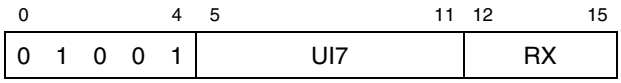


$UI \leftarrow UI_{0:4} \,||\, UI_{5:15}$
$GPR(RD) \leftarrow UI \,||\, {}^{16}0$

For **e_lis**, the UI field is concatenated on the right with 16 0's and placed into GPR(**r**D).

Special Registers Altered: None

se_li **r**X,UI7



$GPR(RX) \leftarrow {}^{25}0 \,||\, UI7$

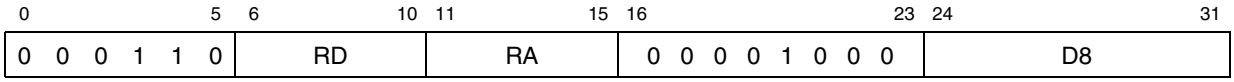
For **se_li**, the zero-extended UI7 field is placed into GPR(**r**X).

Special Registers Altered: None

_lmw | VLE | User | **_lmw**

Load Multiple Word

e_lmw **r**D,D8(**r**A)



if RA=0 then EA ← $EXTS(D8)_{32:63}$

else EA ← $(GPR(RA)+EXTS(D8))_{32:63}$

r ← RD

do while r ≤ 31

GPR(r) ← MEM(EA,4)

r ← r + 1

EA ← $(EA+4)_{32:63}$

Let the EA be the sum of the contents of GPR(**r**A), or 32 0s if **r**A = 0, and the sign-extended value of the D8 instruction field.

Let n = (32-rD). n consecutive words starting at EA are loaded into bits 32–63 of registers GPR(**r**D) through GPR(31).

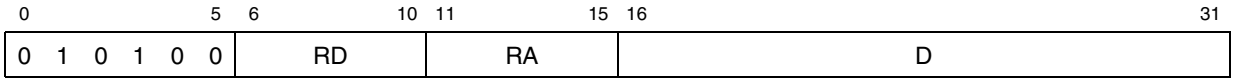
EA must be a multiple of 4. If it is not, either an alignment interrupt is invoked or the results are boundedly undefined. If **r**A is in the range of registers to be loaded, including the case in which **r**A = 0, the instruction form is invalid.

Special Registers Altered: None

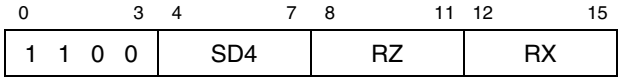
_lwz | VLE | User | **_lwz**

Load Word and Zero [with Update] [Indexed]

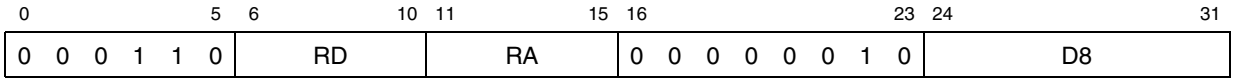
e_lwz **r**D,D**(r**A**)** (D-mode)



se_lwz **r**Z,SD4**(r**X**)** (SD4-mode)



e_lwzu **r**D,D8**(r**A**)** (D8-mode)



if (RA=0 & !se_lwz) then a ← $^{32}0$ else a ← GPR(RA or RX)

if D-mode then EA ← $(a + EXTS(D))_{32:63}$

if D8-mode then EA ← $(a + EXTS(D8))_{32:63}$

if SD4-mode then EA ← $(a + (^{26}0 \,||\, SD4 \,||\, ^{2}0))_{32:63}$

GPR(RD or RZ) ← MEM(EA,4)

if e_lwzu then GPR(RA) ← EA

Let the EA be calculated as follows:

- For **e_lwz** and **e_lwzu**, let EA be the sum of the contents of GPR(**r**A), or 32 0s if **r**A = 0, and the sign-extended value of the D or D8 instruction field.
- For **se_lwz** let EA be the sum of the contents of GPR(**r**X) and the zero-extended value of the SD4 instruction field shifted left by 2 bits.

The word in memory addressed by the EA is loaded into bits 32–63 of GPR(**r**D).

If **e_lwzu**, EA is placed into GPR(**r**A).

If **e_lwzu** and **r**A = 0 or **r**A = **r**D, the instruction form is invalid.

Special Registers Altered: None

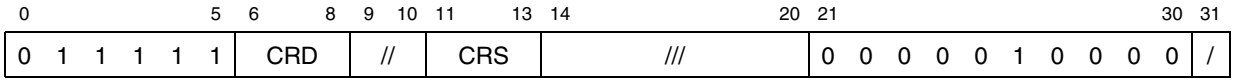
_mcrf

VLE	User

_mcrf

Move CR Field

e_mcrf **cr**D,**cr**S



$CR_{4xCRD+32:4xCRD+35} \leftarrow CR_{4xCRS+32:4xCRS+35}$

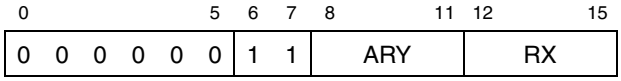
The contents of field **cr**S (bits 4×CRS+32 through 4×CRS+35) of the CR are copied to field **cr**D (bits 4×CRD+32 through 4×CRD+35) of the CR.

Special Registers Altered: CR

_mfar | VLE | User | **_mfar**

Move from Alternate Register

se_mfar **r**X,**ar**Y



GPR(RX) ← GPR(ARY)

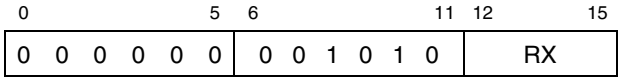
For **se_mfar**, the contents of GPR(**ar**Y) are placed into GPR(**r**X). **ar**Y specifies a GPR in the range R8–R23. The encoding 0000 specifies R8, 0001 specifies R9,…, 1111 specifies R23.

Special Registers Altered: None

_mfctr | VLE | User | **_mfctr**

Move From Count Register

se_mfctr **rX**



GPR(RX) ← CTR

The CTR contents are placed into bits 32–63 of GPR(**r**X).

Special Registers Altered: None

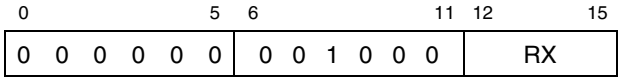
_mflr

VLE	User

_mflr

Move From Link Register

se_mflr **r**X



GPR(RX) ← LR

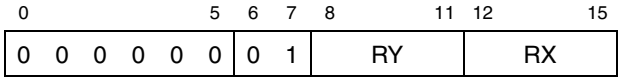
The LR contents are placed into bits 32–63 of GPR(**r**X).

Special Registers Altered: None

_mr | VLE | User | **_mr**

Move Register

se_mr **r**X,**r**Y



GPR(RX) ← GPR(RY)

For **se_mr**, the contents of GPR(**r**Y) are placed into GPR(**r**X).

Special Registers Altered: None

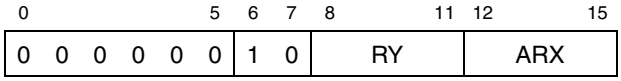
_mtar

VLE	User

_mtar

Move to Alternate Register

se_mtar **ar**X,**r**Y



GPR(ARX) ← GPR(RY)

For **se_mtar**, the contents of GPR(**r**Y) are placed into GPR(**ar**X). **ar**X specifies a GPR in the range R8–R23. The encoding 0000 specifies R8, 0001 specifies R9,…, 1111 specifies R23.

Special Registers Altered: None

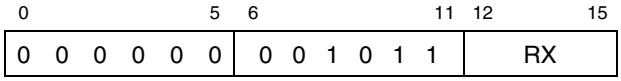
_mtctr

VLE	User

_mtctr

Move To Count Register

se_mtctr **rX**



CTR ← GPR(RX)

The contents of bits 32–63 of GPR(**r**X) are placed into the CTR.

Special Registers Altered: CTR

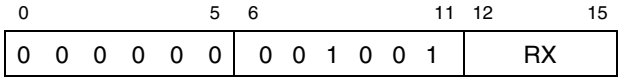
_mtlr

VLE	User

_mtlr

Move To Link Register

se_mtlr **r**X



LR ← GPR(RX)

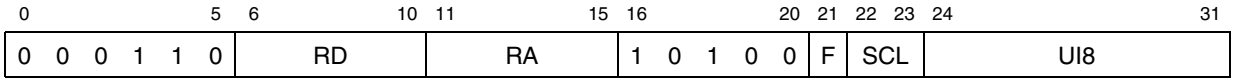
The contents of bits 32–63 of GPR(**r**X) are placed into the LR.

Special Registers Altered: LR

_mulli*x* | VLE | User | **_mulli*x***

Multiply Low [2 operand] Immediate

e_mulli **r**D,**r**A,SCI8



imm ← SCI8(F,SCL,UI8)
$prod_{0:63}$ ← GPR(RA) × imm

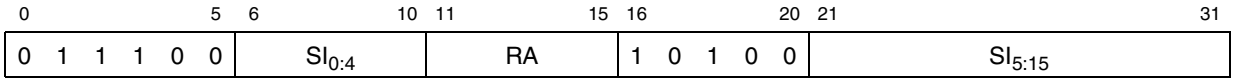
GPR(RD) ← $prod_{32:63}$

Bits 32–63 of the 64-bit product of the contents of GPR(**r**A) and the value of SCI8 are placed into GPR(**r**D).

Both operands and the product are interpreted as signed integers.

Special Registers Altered: None

e_mull2i **r**A,SI



$prod_{0:63}$ ← GPR(RA) × EXTS($SI_{0:4}$ || $SI_{5:15}$)

GPR(RA) ← $prod_{32:63}$

Bits 32–63 of the 64-bit product of the contents of GPR(**r**A) and the sign-extended value of the SI field are placed into GPR(**r**A).

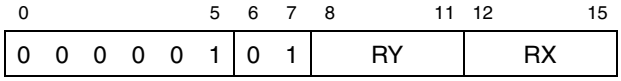
Both operands and the product are interpreted as signed integers.

Special Registers Altered: None

_mullw*x* | VLE | User | **_mullw*x***

Multiply Low Word

se_mullw **r**X,**r**Y



$prod_{0:63} \leftarrow GPR(RX)_{32:63} \times GPR(RY)_{32:63}$

$GPR(RX) \leftarrow prod_{32:63}$

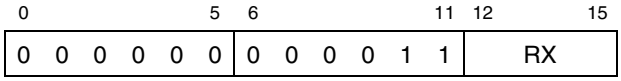
Bits 32–63 of the 64-bit product of the contents of bits 32–63 of GPR(**r**X) and the contents of bits 32–63 of GPR(**r**Y) is placed into GPR(**r**X).

Special Registers Altered: None

_neg*x* | VLE | User | **_neg*x***

Negate

se_neg **r**X



$result_{32:63} \leftarrow \neg GPR(RX) + 1$

$GPR(RX) \leftarrow result_{32:63}$

The sum of the one's complement of the contents of GPR(**r**X) and 1 is placed into GPR(**r**X).

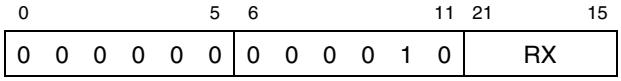
If bits 32–63 of GPR(**r**X) contain the most negative 32-bit number (0x8000_0000), bits 32–63 of the result contain the most negative 32-bit number

Special Registers Altered: None

_not*x* | VLE | User | **_not*x***

NOT

se_not **r**X



$result_{32:63} \leftarrow \neg GPR(RX)$

$GPR(RX) \leftarrow result_{32:63}$

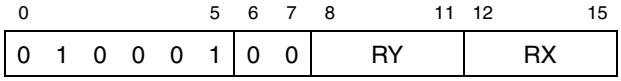
The contents of GPR(**r**X) are inverted.

Special Registers Altered: None

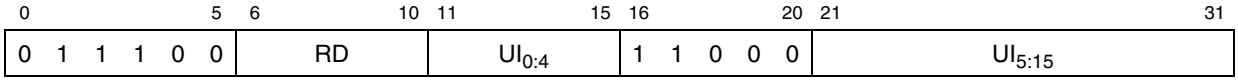
_or*x* | VLE | User | **_or*x***

OR [2 operand] [Immediate | with Complement] [Shifted][and Record]

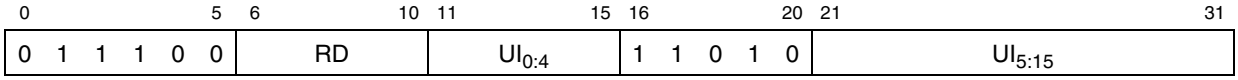
se_or **r**X,**r**Y



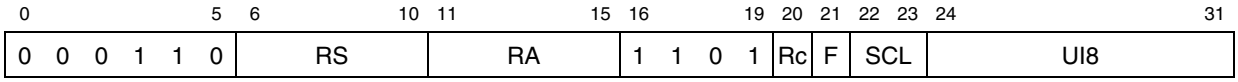
e_or2i **r**D,UI



e_or2is **r**D,UI



e_ori **r**A,**r**S,SCI8 (Rc = 0)
e_ori. **r**A,**r**S,SCI8 (Rc = 1)



if ‘e_ori[.]’ then b ← SCI8(F,SCL,UI8)

if ‘e_or2i’ then b ← $^{16}0$ || $UI_{0:4}$ || $UI_{5:15}$
if ‘e_or2is’ then b ← $UI_{0:4}$ || $UI_{5:15}$ || $^{16}0$
if ‘se_or’ then b ← GPR(RB)

$result_{0:63}$ ← GPR(RS or RD or RX) | b

if Rc=1 then do

LT ← $result_{32:63} < 0$

GT ← $result_{32:63} > 0$

EQ ← $result_{32:63} = 0$

CR0 ← LT || GT || EQ || SO

GPR(RA or RD or RX) ← result

For **e_ori[.]**, the contents of GPR(**r**S) are ORed with the value of SCI8.

For **e_or2i,** the contents of GPR(**r**D) are ORed with $^{16}0$ || UI.

For **e_or2is**, the contents of GPR(**r**D) are ORed with UI || $^{16}0$.

For **se_or**, the contents of GPR(**r**X) are ORed with the contents of GPR(**r**Y).

The result is placed into GPR(**r**A or **r**X).

The preferred ‘no-op’ (an instruction that does nothing) is:

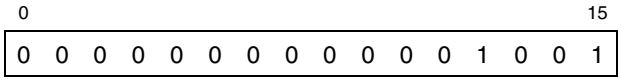
```
e_ori  0,0,0
```

Special Registers Altered: CR0 (if Rc = 1)

_rfci | VLE | Supervisor | **_rfci**

Return From Critical Interrupt

se_rfci



MSR ← CSRR1

NIA ← $CSRR0_{0:62}$ || 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[32–62]||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 SRR0 or CSRR0 by the interrupt processing mechanism (see Book E) is the address 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**).

Execution of this instruction is privileged and restricted to supervisor mode.

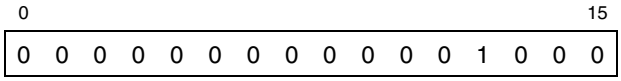
Execution of this instruction is context synchronizing.

Special Registers Altered: MSR

_rfi | VLE | Supervisor | **_rfi**

Return From Interrupt

se_rfi



MSR ← SRR1

NIA ← $SRR0_{0:62}$ || 0b0

The **se_rfi** instruction is used to return from a non-critical 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[32–62]||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 SRR0 or CSRR0 by the interrupt processing mechanism (see Book E) is the address 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**).

Execution of this instruction is privileged and restricted to supervisor mode.

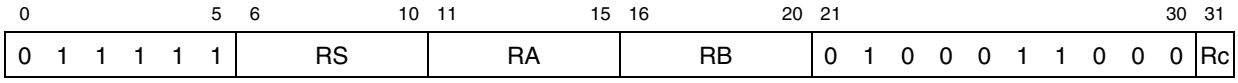
Execution of this instruction is context synchronizing.

Special Registers Altered: MSR

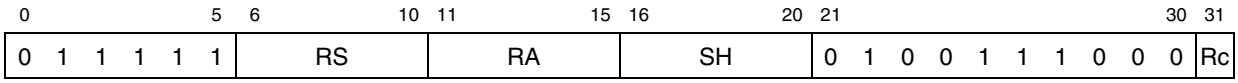
_rlw | VLE | User | **_rlw**

Rotate Left Word [Immediate]

e_rlw **r**A,**r**S,**r**B (Rc = 0)
e_rlw. **r**A,**r**S,**r**B (Rc = 1)



e_rlwi **r**A,**r**S,SH (Rc = 0)
e_rlwi. **r**A,**r**S,SH (Rc = 1)



if 'e_rlw[.]' then n ← $GPR(RB)_{59:63}$

else n ← SH

$result_{32:63}$ ← $ROTL_{32}(GPR(RS)_{32:63},n)$

if Rc=1 then do

LT ← $result_{32:63} < 0$

GT ← $result_{32:63} > 0$

EQ ← $result_{32:63} = 0$

CR0 ← LT || GT || EQ || SO

GPR(RA) ← $result_{32:63}$

If **e_rlw[.]**, let the shift count *n* be the contents of bits 59–63 of GPR(**r**B).

If **e_rlwi[.]**, let the shift count *n* be SH.

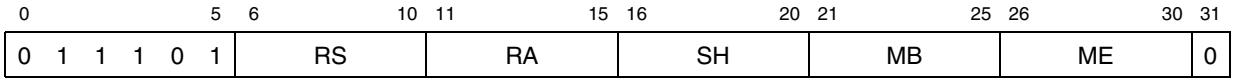
The contents of GPR(**r**S) are $rotated_{32}$ left *n* bits. The rotated data is placed into GPR(**r**A).

Special Registers Altered: CR0 (if Rc = 1)

_rlwimi | VLE | User | **_rlwimi**

Rotate Left Word Immediate then Mask Insert

e_rlwimi **r**A,**r**S,SH,MB,ME



n ← SH

b ← MB+32

e ← ME+32

r ← $ROTL_{32}$(GPR(RS)$_{32:63}$,n)

m ← MASK(b,e)

$result_{32:63}$ ← r&m | GPR(RA)&¬m

GPR(RA) ← $result_{32:63}$

Let the shift count *n* be the value SH.

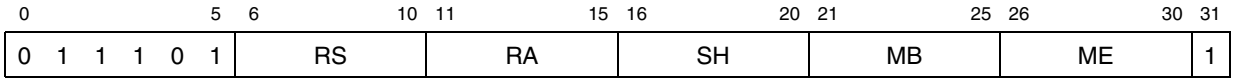
The contents of GPR(**r**S) are $rotated_{32}$ left *n* bits. A mask is generated having 1 bits from bit MB+32 through bit ME+32 and 0 bits elsewhere. The rotated data are inserted into GPR(**r**A) under control of the generated mask (if a mask bit is 1 the associated bit of the rotated data is placed into the target register, and if the mask bit is 0 the associated bit in the target register remains unchanged).

Special Registers Altered: None

_rlwinm | VLE | User | **_rlwinm**

Rotate Left Word Immediate then AND with Mask

e_rlwinm **r**A,**r**S,SH,MB,ME



n ← SH

b ← MB+32

e ← ME+32

r ← $ROTL_{32}$(GPR(RS)$_{32:63}$,n)

m ← MASK(b,e)

$result_{32:63}$ ← r & m

GPR(RA) ← $result_{32:63}$

Let the shift count *n* be SH.

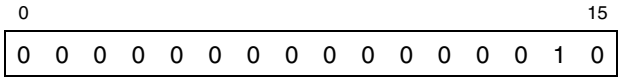
The contents of GPR(**r**S) are $rotated_{32}$ left *n* bits. A mask is generated having 1 bits from bit MB+32 through bit ME+32 and 0 bits elsewhere. The rotated data are ANDed with the generated mask and the result is placed into GPR(**r**A).

Special Registers Altered: None

_sc | VLE | User | **_sc**

System Call

se_sc



SRR1 ← MSR

SRR0 ← CIA+2

NIA ← $IVPR_{32:47}$ || $IOVR8_{48:59}$ || 0b0000

$MSR_{WE,EE,PR,IS,DS,FP,FE0,FE1}$ ← 0b0000_0000

se_sc is used to request a system service. A system call interrupt is generated. The contents of the MSR are copied into SRR1 and the address of the instruction after the **se_sc** instruction is placed into SRR0.

MSR[WE,EE,PR,IS,DS,FP,FE0,FE1] are cleared.

The interrupt causes the next instruction to be fetched from the address

IVPR[32–47]||IVOR8[48–59]||0b0000

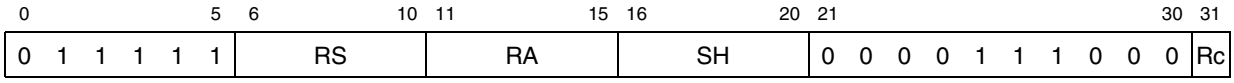
This instruction is context synchronizing.

Special Registers Altered: SRR0 SRR1 MSR[WE,EE,PR,IS,DS,FP,FE0,FE1]

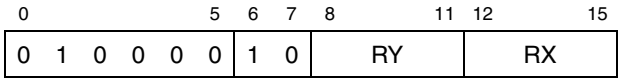
_slw*x* | VLE | User | **_slw*x***

Shift Left Word [Immediate] [and Record]

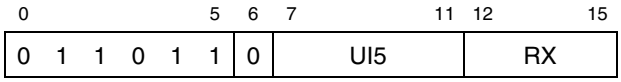
e_slwi **r**A,**r**S,SH (Rc = 0)
e_slwi. **r**A,**r**S,SH (Rc = 1)



se_slw **r**X,**r**Y



se_slwi **r**X,UI5



if ‘e_slwi[.]’ then n ← SH

if se_slw then n ← $GPR(RY)_{58:63}$
if se_slwi then n ← UI5

r ← $ROTL_{32}(GPR(RS \text{ or } RX)_{32:63}, n)$

if n<32 then m ← MASK(32,63-n)

else m ← $^{32}0$

$result_{32:63}$ ← r & m

if Rc=1 then do

LT ← $result_{32:63} < 0$

GT ← $result_{32:63} > 0$

EQ ← $result_{32:63} = 0$

CR0 ← LT || GT || EQ || SO

GPR(RA or RX) ← $result_{32:63}$

Let the shift count *n* be the value specified by the contents of bits 58–63 of GPR(**r**B or **r**Y), or by the value of the SH or UI5 field.

The contents of bits 32–63 of GPR(**r**S or **r**X) are shifted left *n* 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 bits 32–63 of GPR(**r**A or **r**X).

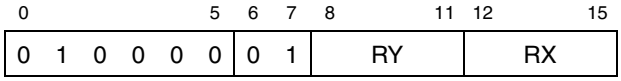
Shift amounts from 32 to 63 give a zero result.

Special Registers Altered: CR0 (if Rc = 1)

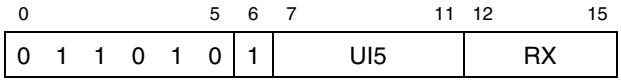
_sraw*x* | VLE | User | **_sraw*x***

Shift Right Algebraic Word [Immediate] [and Record]

se_sraw **r**X,**r**Y



se_srawi **r**X,UI5



if 'se_sraw' then n ← $GPR(RY)_{59:63}$
if 'se_srawi' then n ← UI5

r ← $ROTL_{32}(GPR(RS \text{ or } RX)_{32:63}, 32-n)$

if ((se_sraw & $GPR(RY)_{58}$=1) then m ← $^{32}0$

else m ← MASK(n+32,63)

s ← $GPR(RS \text{ or } RX)_{32}$

$result_{0:63}$ ← r&m | (^{32}s)&¬m

if Rc=1 then do

LT ← $result_{32:63} < 0$

GT ← $result_{32:63} > 0$

EQ ← $result_{32:63} = 0$

CR0 ← LT || GT || EQ || SO

GPR(RA or RX) ← $result_{32:63}$

CA ← s & $((r\&\neg m)_{32:63} \neq 0)$

If **se_sraw**, let the shift count *n* be the contents of bits 58–63 of GPR(**r**Y).

If **se_srawi**, let the shift count *n* be the value of the UI5 field.

The contents of bits 32–63 of GPR(**r**S or **r**X) are shifted right *n* bits. Bits shifted out of position 63 are lost. Bit 32 of **r**S or **r**X is replicated to fill vacated positions on the left. The 32-bit result is placed into bits 32–63 of GPR(**r**A or **r**X).

CA is set if bits 32–63 of GPR(**r**S or **r**X) contain a negative value and any 1 bits are shifted out of bit position 63; otherwise CA is cleared.

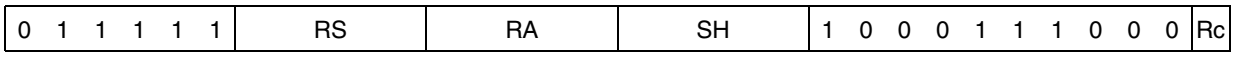
A shift amount of zero causes GPR(**r**A or **r**X) to receive $EXTS(GPR(\mathbf{r}S \text{ or } \mathbf{r}X)_{32:63})$, and CA to be cleared. For **se_sraw,** shift amounts from 32 to 63 give a result of 64 sign bits, and cause CA to receive bit 32 of the contents of GPR(**r**S or **r**X) (that is, sign bit of $GPR(\mathbf{r}S \text{ or } \mathbf{r}X)_{32:63}$).

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

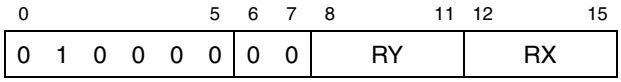
_srw*x* | VLE | User | **_srw*x***

Shift Right Word [Immediate] [and Record]

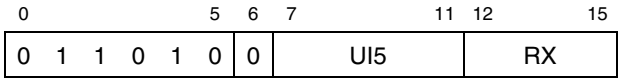
e_srwi **r**A,**r**S,SH (Rc = 0)
e_srwi. **r**A,**r**S,SH (Rc = 1)



se_srw **r**X,**r**Y



se_srwi **r**X,UI5



n ← GPR(RB)$_{59:63}$

if ‘e_srwi[.]’ then n ← SH
if ‘se_srw’ then n ← GPR(RY)$_{59:63}$
if ‘se_srwi’ then n ← UI5
r ← ROTL$_{32}$(GPR(RS or RX)$_{32:63}$,32-n)

if ((se_srw & GPR(RY)$_{58}$=1) then m ← ^{32}0

else m ← MASK(n+32,63)

result$_{32:63}$ ← r & m

if Rc=1 then do

LT ← result$_{32:63}$ < 0

GT ← result$_{32:63}$ > 0

EQ ← result$_{32:63}$ = 0

CR0 ← LT || GT || EQ || SO

GPR(RA or RX) ← result$_{32:63}$

If **e_srwi**, let the shift count *n* be the value of the SH field.

If **se_srw**, let the shift count *n* be the contents of bits 58–63 of GPR(**r**Y).

If **se_srwi**, let the shift count *n* be the value of the UI5 field.

The contents of bits 32–63 of GPR(**r**S or **r**X) are shifted right *n* 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 bits 32–63 of GPR(**r**A or **r**X).

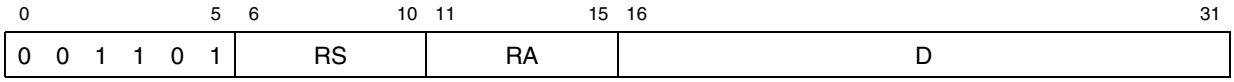
Shift amounts from 32 to 63 give a zero result.

Special Registers Altered: CR0 (if Rc = 1)

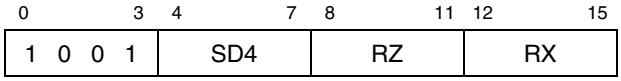
_stb*x* | VLE | User | **_stb*x***

Store Byte [with Update] [Indexed]

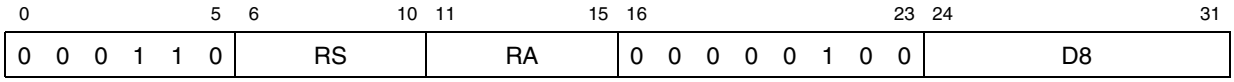
e_stb **r**S,D(**r**A) (D-mode)



se_stb **r**Z,SD4(**r**X) (SD4-mode)



e_stbu **r**S,D8(**r**A) (D8-mode)



if (RA=0 & !se_stb) then a ← $^{32}0$ else a ← GPR(RA or RX)

if D-mode then EA ← $(a + EXTS(D))_{32:63}$

if D8-mode then EA ← $(a + EXTS(D8))_{32:63}$

if SD4-mode then EA ← $(a + (^{28}0 \,||\, SD4))_{32:63}$

$MEM(EA,1) \leftarrow GPR(RS \text{ or } RZ)_{56:63}$

if e_stbu then GPR(RA) ← EA

Let the EA be calculated as follows:

- For **e_stb** and **e_stbu**, let EA be the sum of the contents of GPR(**r**A), or 32 0s if **r**A = 0, and the sign-extended value of the D or D8 instruction field.
- For **se_stb**, let EA be the sum of the contents of GPR(**r**X) and the zero-extended value of the SD4 instruction field.

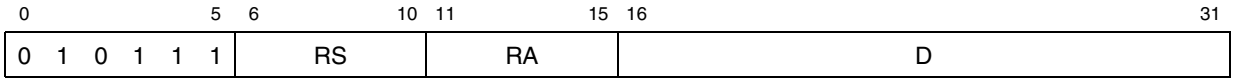
The contents of bits 56–63 of GPR(**r**S) are stored into the byte in memory addressed by EA.

- If **e_stbu**, EA is placed into GPR(**r**A).
- If **e_stbu** and **r**A = 0, the instruction form is invalid.
- None

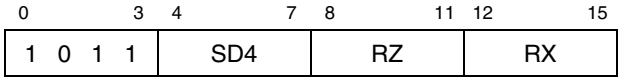
_sthx | VLE | User | **_sthx**

Store Halfword [with Update] [Indexed]

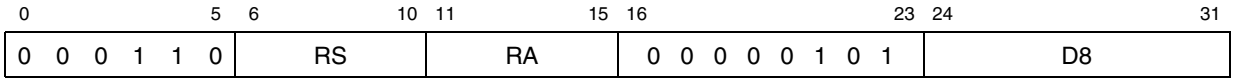
e_sth **r**S,D(**r**A) (D-mode)



se_sth **r**Z,SD4(**r**X) (SD4-mode)



e_sthu **r**S,D8(**r**A) (D8-mode)



if (RA=0 & !se_sth) then a ← $^{32}0$ else a ← GPR(RA or RX)

if D-mode then EA ← $(a + EXTS(D))_{32:63}$

if D8-mode then EA ← $(a + EXTS(D8))_{32:63}$

if SD4-mode then EA ← $(a + (^{27}0 \,||\, SD4 \,||\, 0))_{32:63}$

MEM(EA,2) ← $GPR(RS \text{ or } RZ)_{48:63}$

if e_sthu then GPR(RA) ← EA

Let the EA be calculated as follows:

- For **e_sth** and **e_sthu**, let EA be the sum of the contents of GPR(**r**A), or 32 0s if **r**A = 0, and the sign-extended value of the D or D8 instruction field.
- For **se_sth** let EA be the sum of the contents of GPR(**r**X) and the zero-extended value of the SD4 instruction field shifted left by 1 bit.

The contents of bits 48–63 of GPR(**r**S) are stored into the half word in memory addressed by EA.

If **e_sthu**, EA is placed into GPR(**r**A).

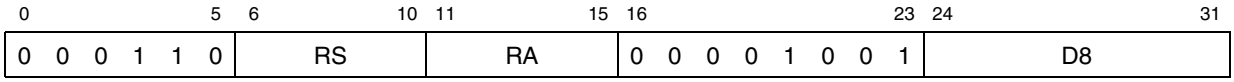
If **e_sthu** and **r**A = 0, the instruction form is invalid.

Special Registers Altered: None

_stmw | VLE | User | **_stmw**

Store Multiple Word

e_stmw **r**S,D8**(r**A**)** (D8-mode)



if RA=0 then EA ← $EXTS(D8)_{32:63}$

else EA ← $(GPR(RA)+EXTS(D8))_{32:63}$

r ← RS

do while r ≤ 31

MEM(EA,4) ← $GPR(r)_{32:63}$

r ← r + 1

EA ← $(EA+4)_{32:63}$

Let the EA be the sum of the contents of GPR(**r**A), or 32 0s if **r**A = 0, and the sign-extended value of the D8 instruction field.

Let n = (32 - **r**S). Bits 32–63 of registers GPR(**r**S) through GPR(31) are stored in n consecutive words in memory starting at address EA.

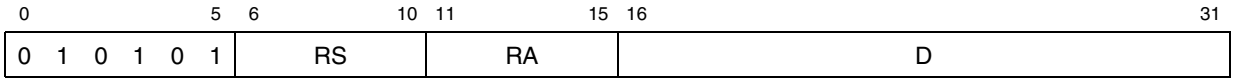
EA must be a multiple of 4. If it is not, either an alignment interrupt is invoked or the results are boundedly undefined.

Special Registers Altered: None

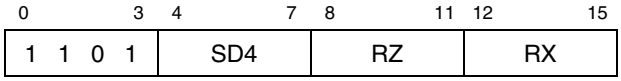
_stw*x* | VLE | User | **_stw*x***

Store Word [with Update] [Indexed]

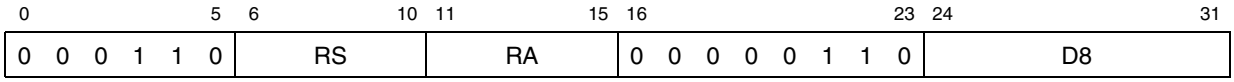
e_stw **r**S,D(**r**A) (D-mode)



se_stw **r**Z,SD4(**r**X) (SD4-mode)



e_stwu **r**S,D8(**r**A) (D8-mode)



if (RA=0 & !se_stw) then a ← $^{32}0$ else a ← GPR(RA or RX)

if D-mode then EA ← $(a + EXTS(D))_{32:63}$

if D8-mode then EA ← $(a + EXTS(D8))_{32:63}$

if SD4-mode then EA ← $(a + (^{26}0 \,||\, SD4 \,||\, ^{2}0))_{32:63}$

MEM(EA,4) ← $GPR(RS \text{ or } RZ)_{32:63}$

Let the EA be calculated as follows:

- For **e_stw** and **e_stwu**, let EA be the sum of the contents of GPR(**r**A), or 32 0s if **r**A = 0, and the sign-extended value of the D or D8 instruction field.
- For **se_stw**, let EA be the sum of the contents of GPR(**r**X) and the zero-extended value of the SD4 instruction field shifted left by 2 bits.

The contents of bits 32–63 of GPR(**r**S) are stored into the word in memory addressed by EA.

If **e_stwu**, EA is placed into GPR(**r**A).

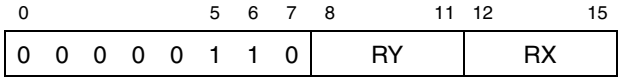
If **e_stwu** and **r**A = 0, the instruction form is invalid.

Special Registers Altered: None

_sub | VLE | User | **_sub**

Subtract

se_sub **r**X,**r**Y



$sum_{32:63} \leftarrow GPR(RX) + \neg GPR(RY) + 1$

$GPR(RX) \leftarrow sum_{32:63}$

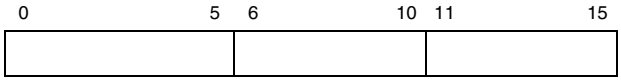
The sum of the contents of GPR(**r**X), the one's complement of contents of GPR(**r**Y), and 1 is placed into GPR(**r**X).

Special Registers Altered: None

_subf*x* | VLE | User | **_subf*x***

Subtract From

se_subf **r**X,**r**Y



$sum_{32:63} \leftarrow \neg GPR(RX) + GPR(RY) + 1$

$GPR(RX) \leftarrow sum_{32:63}$

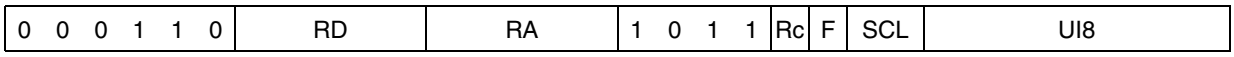
The sum of the one's complement of the contents of GPR(**r**X), the contents of GPR(**r**Y), and 1 is placed into GPR(**r**X).

Special Registers Altered: None

_subfic*x* | VLE | User | **_subfic*x***

Subtract From Immediate Carrying [and Record]

e_subfic **r**D,**r**A,SCI8 (Rc = 0)
e_subfic. **r**D,**r**A,SCI8 (Rc = 1)



imm ← SCI8(F,SCL,UI8)
$carry_{32:63}$ ← Carry(¬GPR(RA) + imm + 1)

$sum_{32:63}$ ← ¬GPR(RA) + imm + 1

if Rc=1 then do

LT ← $sum_{32:63}$ < 0

GT ← $sum_{32:63}$ > 0

EQ ← $sum_{32:63}$ = 0

CR0 ← LT || GT || EQ || SO

GPR(RD) ← $sum_{32:63}$
CA ← $carry_{32}$

The sum of the one's complement of the contents of GPR(**r**A), the value of SCI8, and 1 is placed into GPR(**r**D).

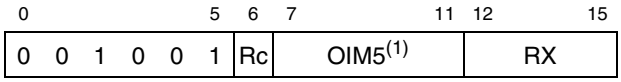
Special Registers Altered: CA CR0 (if Rc=1)

_subi*x* | VLE | User | **_subi*x***

Subtract Immediate [and Record]

se_subi **r**X,OIMM (Rc = 0)

se_subi. **r**X,OIMM (Rc = 1)



1. OIMM = OIM5 +1

$sum_{32:63} \leftarrow GPR(RX) + \neg(^{27}0 \,||\, OFFSET(OIM5)) + 1$

if Rc=1 then do

$LT \leftarrow sum_{32:63} < 0$

$GT \leftarrow sum_{32:63} > 0$

$EQ \leftarrow sum_{32:63} = 0$

$CR0 \leftarrow LT \,||\, GT \,||\, EQ \,||\, SO$

$GPR(RX) \leftarrow sum_{32:63}$

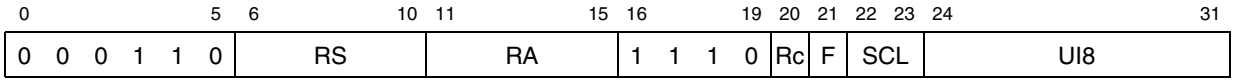
The sum of the contents of GPR(**r**X), the one's complement of the zero-extended value of the offseted OIM5 field (a final value in the range 1–32), and 1 is placed into GPR(**r**X).

Special Registers Altered: CR0 (if Rc = 1)

_xor*x* | VLE | User | **_xor*x***

XOR [Immediate] [and Record]

e_xori **r**A,**r**S,SCI8 (Rc = 0)
e_xori. **r**A,**r**S,SCI8 (Rc = 1)



if ‘e_xori[.]’ then b ← SCI8(F,SCL,UI8)

$result_{32:63}$ ← GPR(RS) ⊕ b

if Rc=1 then do

LT ← $result_{32:63}$ < 0

GT ← $result_{32:63}$ > 0

EQ ← $result_{32:63}$ = 0

CR0 ← LT || GT || EQ || SO

GPR(RA) ← result

For **e_xori[.]**, the contents of GPR(**r**S) are XORed with SCI8.

The result is placed into GPR(**r**A).

Special Registers Altered: CR0 (if Rc = 1)

14 VLE instruction index

The tables in this appendix use the following conventions:

Table 260. Notation conventions

Notation	Meaning
-	Don't care, usually part of an operand field
/	Reserved bit, invalid instruction form if encoded as 1
?	Allocated for implementation-dependent use. See the implementation documentation.

14.1 Instruction index sorted by opcode

Table 261 lists the 16-bit VLE instructions, sorted by opcode.

Table 261. Instruction index sorted by opcode

Format	16-Bit opcodes ($Inst_{0:15}$)	Mnemonic	Instruction	Page
C	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	**se_illegal**	Illegal	*-928*
C	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1	**se_isync**	Instruction Synchronize	*-929*
C	0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0	**se_sc**	System Call	*-954*
C	0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0	**se_blr**	Branch to Link Register	*-908*
C	0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1	**se_blrl**	Branch to link register & link	*-908*
C	0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0	**se_bctr**	Branch to Count Register	*-906*
C	0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1	**se_bctrl**	Branch to Count Register & Link	*-906*
C	0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0	**se_rfi**	Return From Interrupt	*-950*
C	0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1	**se_rfci**	Return From Critical Interrupt	*-949*
C	0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0	**se_rfdi**	Return From Debug Interrupt	*-949*
C	0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 1			
C	0 0 0 0 0 0 0 0 0 0 0 1 - - - -			
R	0 0 0 0 0 0 0 0 0 0 1 0 x x x x	**se_not**	NOT	*-947*
R	0 0 0 0 0 0 0 0 0 0 1 1 x x x x	**se_neg**	Negate	*-946*
R	0 0 0 0 0 0 0 0 0 1 - - x x x x			
R	0 0 0 0 0 0 0 0 1 0 0 0 x x x x	**se_mflr**	Move From Link Register	*-939*
R	0 0 0 0 0 0 0 0 1 0 0 1 x x x x	**se_mtlr**	Move To Link Register	*-943*
R	0 0 0 0 0 0 0 0 1 0 1 0 x x x x	**se_mfctr**	Move From Count Register	*-938*
R	0 0 0 0 0 0 0 0 1 0 1 1 x x x x	**se_mtctr**	Move To Count Register	*-942*
R	0 0 0 0 0 0 0 0 1 1 0 0 x x x x	**se_extzb**	Extend with Zeros Byte	*-927*

Table 261. Instruction index sorted by opcode (continued)

Format	16-Bit opcodes ($Inst_{0:15}$)	Mnemonic	Instruction	Page
R	0 0 0 0 0 0 0 0 1 1 0 1 x x x x	**se_extsb**	Extend Sign Byte	*-926*
R	0 0 0 0 0 0 0 0 1 1 1 0 x x x x	**se_extzh**	Extend with Zeros Halfword	*-927*
R	0 0 0 0 0 0 0 0 1 1 1 1 x x x x	**se_extsh**	Extend Sign Halfword	*-926*
R	0 0 0 0 0 0 0 1 y y y y x x x x	**se_mr**	Move Register	*-940*
RR	0 0 0 0 0 0 1 0 y y y y x x x x	**se_mtar**	Move to Alternate Register	*-941*
RR	0 0 0 0 0 0 1 1 y y y y x x x x	**se_mfar**	Move from Alternate Register	*-937*
RR	0 0 0 0 0 1 0 0 y y y y x x x x	**se_add**	Add	*-897*
RR	0 0 0 0 0 1 0 1 y y y y x x x x	**se_mullw**	Multiply Low Word	*-945*
RR	0 0 0 0 0 1 1 0 y y y y x x x x	**se_sub**	Subtract	*-962*
RR	0 0 0 0 0 1 1 1 y y y y x x x x	**se_subf**	Subtract From	*-963*
RR	0 0 0 0 1 0 - - y y y y x x x x			
RR	0 0 0 0 1 1 0 0 y y y y x x x x	**se_cmp**	Compare	*-912*
RR	0 0 0 0 1 1 0 1 y y y y x x x x	**se_cmpl**	Compare Logical	*-918*
RR	0 0 0 0 1 1 1 0 y y y y x x x x	**se_cmph**	Compare Halfword	*-914*
RR	0 0 0 0 1 1 1 1 y y y y x x x x	**se_cmphl**	Compare Halfword Logical	*-916*
IM5	0 0 1 0 0 0 0 i i i i i x x x x	**se_addi**	Add Immediate	*-897*
IM5	0 0 1 0 0 0 1 i i i i i x x x x	**se_cmpli**	Compare Logical Immediate	*-918*
IM5	0 0 1 0 0 1 0 i i i i i x x x x	**se_subi**	Subtract Immediate	*-965*
IM5	0 0 1 0 0 1 1 i i i i i x x x x	**se_subi.**	Subtract Immediate and Record	*-965*
IM5	0 0 1 0 1 0 0 i i i i i x x x x			
IM5	0 0 1 0 1 0 1 i i i i i x x x x	**se_cmpi**	Compare Immediate	*-912*
IM5	0 0 1 0 1 1 0 i i i i i x x x x	**se_bmaski**	Bit Mask Generate Immediate	*-909*
IM5	0 0 1 0 1 1 1 i i i i i x x x x	**se_andi**	And Immediate	*-901*
RR	0 1 0 0 0 0 0 0 y y y y x x x x	**se_srw**	Shift Right Word	*-957*
RR	0 1 0 0 0 0 0 1 y y y y x x x x	**se_sraw**	Shift Right Algebraic Word	*-956*
RR	0 1 0 0 0 0 1 0 y y y y x x x x	**se_slw**	Shift Left Word	*-955*
RR	0 1 0 0 0 0 1 1 y y y y x x x x			
RR	0 1 0 0 0 1 0 0 y y y y x x x x	**se_or**	OR	*-948*
RR	0 1 0 0 0 1 0 1 y y y y x x x x	**se_andc**	AND with Complement	*-901*
RR	0 1 0 0 0 1 1 0 y y y y x x x x	**se_and**	AND	*-901*
RR	0 1 0 0 0 1 1 1 y y y y x x x x	**se_and.**	AND and Record	*-901*
IM7	0 1 0 0 1 i i i i i i i x x x x	**se_li**	Load Immediate	*-933*

Table 261. Instruction index sorted by opcode (continued)

Format	16-Bit opcodes ($Inst_{0:15}$)	Mnemonic	Instruction	Page
IM5	0 1 1 0 0 0 0 i i i i i x x x x	**se_bclri**	Bit Clear Immediate	*-905*
IM5	0 1 1 0 0 0 1 i i i i i x x x x	**se_bgeni**	Bit Generate Immediate	*-906*
IM5	0 1 1 0 0 1 0 i i i i i x x x x	**se_bseti**	Bit Set Immediate	*-910*
IM5	0 1 1 0 0 1 1 i i i i i x x x x	**se_btsti**	Bit Test Immediate	*-911*
IM5	0 1 1 0 1 0 0 i i i i i x x x x	**se_srwi**	Shift Right Word Immediate	*-957*
IM5	0 1 1 0 1 0 1 i i i i i x x x x	**se_srawi**	Shift Right Algebraic Word Immediate	*-957*
IM5	0 1 1 0 1 1 0 i i i i i x x x x	**se_slwi**	Shift Left Word Immediate	*-955*
IM5	0 1 1 0 1 1 1 i i i i i x x x x			
SD4	1 0 0 0 i i i i z z z z x x x x	**se_lbz**	Load Byte and Zero	*-930*
SD4	1 0 0 1 i i i i z z z z x x x x	**se_stb**	Store Byte	*-958*
SD4	1 0 1 0 i i i i z z z z x x x x	**se_lhz**	Load Halfword and Zero	*-932*
SD4	1 0 1 1 i i i i z z z z x x x x	**se_sth**	Store Halfword	*-959*
SD4	1 1 0 0 i i i i z z z z x x x x	**se_lwz**	Load Word and Zero	*-935*
SD4	1 1 0 1 i i i i z z z z x x x x	**se_stw**	Store Word	*-961*
B8	1 1 1 0 0 o i i d d d d d d d d	**se_bc**	Branch Conditional	*-904*
B8	1 1 1 0 1 0 0 0 d d d d d d d d	**se_b**	Branch	*-910*
B8	1 1 1 0 1 0 0 1 d d d d d d d d	**se_bl**	Branch and Link	*-910*
	1 1 1 0 1 0 1 - - - - - - - - -			
	1 1 1 0 1 1 - - - - - - - - - -			

Table 262 shows 32-bit instruction encodings.

Table 262. 32-bit instruction encodings

Format	Opcode			Mnemonic	Instruction	Page
	Primary ($Inst_{0:5}$)	Intermediate ($Inst_{6:20}$)	Extended ($Inst_{21:31}$)			
APU	0 0 0 1 0 -	- - - - - - - - - - - - - - -	- - - - - - - - - - -	**apu**	Reserved for APUs	
D8	0 0 0 1 1 0	t t t t t a a a a a 0 0 0 0 0	0 0 0 d d d d d d d d	**e_lbzu**	Load Byte & Zero with Update	*-930*
D8	0 0 0 1 1 0	t t t t t a a a a a 0 0 0 0 0	0 0 1 d d d d d d d d	**e_lhzu**	Load Halfword & Zero with Update	*-932*
D8	0 0 0 1 1 0	t t t t t a a a a a 0 0 0 0 0	0 1 0 d d d d d d d d	**e_lwzu**	Load Word & Zero with Update	*-935*

Table 262. 32-bit instruction encodings (continued)

Format	Opcode			Mnemonic	Instruction	Page
	Primary ($Inst_{0:5}$)	Intermediate ($Inst_{6:20}$)	Extended ($Inst_{21:31}$)			
D8	000110	ttttt aaaaa 00000	011dd ddddd d	**e_lhau**	Load Halfword Algebraic With Update	-931
D8	000110	ttttt aaaaa 00000	100dd ddddd d	**e_stbu**	Store Byte with Update	-958
D8	000110	ttttt aaaaa 00000	101dd ddddd d	**e_sthu**	Store Halfword with Update	-959
D8	000110	ttttt aaaaa 00000	110dd ddddd d	**e_stwu**	Store Word with Update	-961
D8	000110	ttttt aaaaa 00000	111dd ddddd d			
D8	000110	ttttt aaaaa 00001	000dd ddddd d	**e_lmw**	Load Multiple Word	-934
D8	000110	ttttt aaaaa 00001	001dd ddddd d	**e_stmw**	Store Multiple Word	-960
D8	000110	ttttt aaaaa 00001	010dd ddddd d			
D8	000110	ttttt aaaaa 00001	011dd ddddd d			
	000110	ttttt aaaaa 00001	1--dd ddddd d			
	000110	ttttt aaaaa 0001-	---------- -			
	000110	ttttt aaaaa 001--	---------- -			
	000110	ttttt aaaaa 01---	---------- -			
SCI8	000110	ttttt aaaaa 10000	FSSii iiiii i	**e_addi**	Add Immediate	-897
SCI8	000110	ttttt aaaaa 10001	FSSii iiiii i	**e_addi.**	Add Immediate and Record	-897
SCI8	000110	ttttt aaaaa 10010	FSSii iiiii i	**e_addic**	Add Immediate Carrying	-900
SCI8	000110	ttttt aaaaa 10011	FSSii iiiii i	**e_addic.**	Add Immediate Carrying and Record	-900
SCI8	000110	ttttt aaaaa 10100	FSSii iiiii i	**e_mulli**	Multiply Low Immediate	-944
SCI8	000110	000bf aaaaa 10101	FSSii iiiii i	**e_cmpi**	Compare Immediate	-912
SCI8	000110	001bf aaaaa 10101	FSSii iiiii i	**e_cmpli**	Compare Logical Immediate	-918
SCI8	000110	ttttt aaaaa 10110	FSSii iiiii i	**e_subfic**	Subtract from Immediate Carrying	-964

Table 262. 32-bit instruction encodings (continued)

Format	Opcode			Mnemonic	Instruction	Page
	Primary ($Inst_{0:5}$)	Intermediate ($Inst_{6:20}$)	Extended ($Inst_{21:31}$)			
SCI8	0 0 0 1 1 0	t t t t t a a a a a 1 0 1 1 1	F S S i i i i i i i i	**e_subfic.**	Subtract from Immediate and Record	-964
SCI8	0 0 0 1 1 0	s s s s s a a a a a 1 1 0 0 0	F S S i i i i i i i i	**e_andi**	AND Immediate	-901
SCI8	0 0 0 1 1 0	s s s s s a a a a a 1 1 0 0 1	F S S i i i i i i i i	**e_andi.**	AND Immediate and Record	-901
SCI8	0 0 0 1 1 0	s s s s s a a a a a 1 1 0 1 0	F S S i i i i i i i i	**e_ori**	OR Immediate	-951
SCI8	0 0 0 1 1 0	s s s s s a a a a a 1 1 0 1 1	F S S i i i i i i i i	**e_ori.**	OR Immediate and Record	-951
SCI8	0 0 0 1 1 0	s s s s s a a a a a 1 1 1 0 0	F S S i i i i i i i i	**e_xori**	XOR Immediate	-966
SCI8	0 0 0 1 1 0	s s s s s a a a a a 1 1 1 0 1	F S S i i i i i i i i	**e_xori.**	XOR Immediate and Record	-966
SCI8	0 0 0 1 1 0	s s s s s a a a a a 1 1 1 1 0	F S S i i i i i i i i			
SCI8	0 0 0 1 1 0	s s s s s a a a a a 1 1 1 1 1	F S S i i i i i i i i			
D	0 0 0 1 1 1	t t t t t a a a a a i i i i i	i i i i i i i i i i i	**e_add16i**	Add Immediate	-897
D	0 0 1 1 0 0	t t t t t a a a a a d d d d d	d d d d d d d d d d d	**e_lbz**	Load Byte & Zero	-930
D	0 0 1 1 0 1	t t t t t a a a a a d d d d d	d d d d d d d d d d d	**e_stb**	Store Byte	-958
D	0 0 1 1 1 0	t t t t t a a a a a d d d d d	d d d d d d d d d d d	**e_lha**	Load Halfword Algebraic	-931
	0 0 1 1 1 1	- - - - - - - - - - - - - - -	- - - - - - - - - - -			
D	0 1 0 1 0 0	t t t t t a a a a a d d d d d	d d d d d d d d d d d	**e_lwz**	Load Word & Zero	-935
D	0 1 0 1 0 1	t t t t t a a a a a d d d d d	d d d d d d d d d d d	**e_stw**	Store Word	-961
D	0 1 0 1 1 0	t t t t t a a a a a d d d d d	d d d d d d d d d d d	**e_lhz**	Load Halfword & Zero	-932
D	0 1 0 1 1 1	t t t t t a a a a a d d d d d	d d d d d d d d d d d	**e_sth**	Store Halfword	-959
LI20	0 1 1 1 0 0	t t t t t i i i i i 0 i i i i	i i i i i i i i i i i	**e_li**	Load Immediate	-933
I16A	0 1 1 1 0 0	i i i i i a a a a a 1 0 0 0 0	i i i i i i i i i i i			
I16A	0 1 1 1 0 0	i i i i i a a a a a 1 0 0 0 1	i i i i i i i i i i i	**e_add2i.**	Add (2 operand) Immediate and Record CR	-898
I16A	0 1 1 1 0 0	i i i i i a a a a a 1 0 0 1 0	i i i i i i i i i i i	**e_add2is**	Add (2 operand) Immediate Shifted	-898
I16A	0 1 1 1 0 0	i i i i i a a a a a 1 0 0 1 1	i i i i i i i i i i i	**e_cmp16i**	Compare Immediate	-912

Table 262. 32-bit instruction encodings (continued)

Format	Opcode			Mnemonic	Instruction	Page
	Primary ($Inst_{0:5}$)	Intermediate ($Inst_{6:20}$)	Extended ($Inst_{21:31}$)			
I16A	0 1 1 1 0 0	i i i i i a a a a a	1 0 1 0 0 i i i i i i i i i i i	**e_mull2i**	Multiply Low Word (2 operand) Immediate	-944
I16A	0 1 1 1 0 0	i i i i i a a a a a	1 0 1 0 1 i i i i i i i i i i i	**e_cmpl16i**	Compare Logical Immediate	-918
I16A	0 1 1 1 0 0	i i i i i a a a a a	1 0 1 1 0 i i i i i i i i i i i	**e_cmph16i**	Compare Halfword Immediate	-914
I16A	0 1 1 1 0 0	i i i i i a a a a a	1 0 1 1 1 i i i i i i i i i i i	**e_cmphl16i**	Compare Halfword Logical Immediate	-916
I16L	0 1 1 1 0 0	t t t t t i i i i i	1 1 0 0 0 i i i i i i i i i i i	**e_or2i**	OR (2 operand) Immediate	-948
I16L	0 1 1 1 0 0	t t t t t i i i i i	1 1 0 0 1 i i i i i i i i i i i	**e_and2i.**	AND (2 operand) Immediate & record CR	-901
I16L	0 1 1 1 0 0	t t t t t i i i i i	1 1 0 1 0 i i i i i i i i i i i	**e_or2is**	OR (2 operand) Immediate Shifted	-966
I16L	0 1 1 1 0 0	t t t t t i i i i i	1 1 0 1 1 i i i i i i i i i i i			
I16L	0 1 1 1 0 0	t t t t t i i i i i	1 1 1 0 0 i i i i i i i i i i i	**e_lis**	Load Immediate Shifted	-933
I16L	0 1 1 1 0 0	t t t t t i i i i i	1 1 1 0 1 i i i i i i i i i i i	**e_and2is.**	AND (2 operand) Immediate Shifted & record CR	-901
I16L	0 1 1 1 0 0	t t t t t i i i i i	1 1 1 1 0 i i i i i i i i i i i			
I16L	0 1 1 1 0 0	t t t t t i i i i i	1 1 1 1 1 i i i i i i i i i i i			
RLWI	0 1 1 1 0 1	s s s s s a a a a a	h h h h h b b b b b e e e e e 0	**e_rlwimi**	Rotate Left Word Immed then Mask Insert	-952
RLWI	0 1 1 1 0 1	s s s s s a a a a a	h h h h h b b b b b e e e e e 1	**e_rlwinm**	Rotate Left Word Immed then AND with Mask	-953
BD24	0 1 1 1 1 0	0 d d d d d d d d d d	d d d d d d d d d d 0	**e_b**	Branch	-903
BD24	0 1 1 1 1 0	0 d d d d d d d d d d	d d d d d d d d d d 1	**e_bl**	Branch & Link	-903
BD15	0 1 1 1 1 0	1 0 0 0 o o i i i i	d d d d d d d d d d 0	**e_bc**	Branch Conditional	-904

Table 262. 32-bit instruction encodings (continued)

Format	Opcode			Mnemonic	Instruction	Page
	Primary ($Inst_{0:5}$)	Intermediate ($Inst_{6:20}$)	Extended ($Inst_{21:31}$)			
BD15	0 1 1 1 1 0	1 0 0 0 o o i i i i d d d d d	d d d d d d d d d d 1	**e_bcl**	Branch Conditional & Link	*-904*
X	0 1 1 1 1 1		- - - - - 0 1 1 1 1 /	**isel**	Integer Select	Book E
X	0 1 1 1 1 1		/ 0 0 0 0 0 1 0 1 1 0	**mulhwu**	Multiply High Word Unsigned	Book E
X	0 1 1 1 1 1		/ 0 0 0 0 0 1 0 1 1 1	**mulhwu.**	Multiply High Word Unsigned & Record	Book E
X	0 1 1 1 1 1		/ 0 0 1 0 0 1 0 1 1 0	**mulhw**	Multiply High Word	Book E
X	0 1 1 1 1 1		/ 0 0 1 0 0 1 0 1 1 1	**mulhw.**	Multiply High Word & record CR	Book E
X	0 1 1 1 1 1		0 0 0 0 0 0 0 0 0 0 /	**cmp**	Compare	Book E
X	0 1 1 1 1 1		0 0 0 0 0 0 0 1 0 0 /	**tw**	Trap Word	Book E
X	0 1 1 1 1 1		0 0 0 0 0 0 1 0 0 0 0	**subfc**	Subtract From Carrying	Book E
X	0 1 1 1 1 1		0 0 0 0 0 0 1 0 0 0 1	**subfc.**	Subtract From Carrying & record CR	Book E
X	0 1 1 1 1 1		0 0 0 0 0 0 1 0 1 0 0	**addc**	Add Carrying	Book E
X	0 1 1 1 1 1		0 0 0 0 0 0 1 0 1 0 1	**addc.**	Add Carrying & record CR	Book E
X	0 1 1 1 1 1		0 0 0 0 0 0 1 1 1 0 /	**e_cmph**	Compare Halfword	*-914*
XL	0 1 1 1 1 1		0 0 0 0 0 1 0 0 0 0 /	**e_mcrf**	Move Condition Register Field	*-944*
X	0 1 1 1 1 1		0 0 0 0 0 1 0 0 1 1 /	**mfcr**	Move From Condition Register	Book E
X	0 1 1 1 1 1		0 0 0 0 0 1 0 1 0 0 /	**lwarx**	Load Word & Reserve Indexed	Book E
X	0 1 1 1 1 1		0 0 0 0 0 1 0 1 1 0 /	**icbt**	Instruction Cache Block Touch Indexed	Book E

Table 262. 32-bit instruction encodings (continued)

Format	Opcode			Mnemonic	Instruction	Page
	Primary ($Inst_{0:5}$)	Intermediate ($Inst_{6:20}$)	Extended ($Inst_{21:31}$)			
X	0 1 1 1 1 1		0 0 0 0 0 1 0 1 1 1 /	**lwzx**	Load Word & Zero Indexed	Book E
X	0 1 1 1 1 1		0 0 0 0 0 1 1 0 0 0 0	**slw**	Shift Left Word	Book E
X	0 1 1 1 1 1		0 0 0 0 0 1 1 0 0 0 1	**slw.**	Shift Left Word & record CR	Book E
X	0 1 1 1 1 1		0 0 0 0 0 1 1 0 1 0 0	**cntlzw**	Count Leading Zeros Word	Book E
X	0 1 1 1 1 1		0 0 0 0 0 1 1 0 1 0 1	**cntlzw.**	Count Leading Zeros Word & record CR	Book E
X	0 1 1 1 1 1		0 0 0 0 0 1 1 1 0 0 0	**and**	AND	Book E
X	0 1 1 1 1 1		0 0 0 0 0 1 1 1 0 0 1	**and.**	AND & record CR	Book E
X	0 1 1 1 1 1		0 0 0 0 1 0 0 0 0 0 /	**cmpl**	Compare Logical	Book E
XL	0 1 1 1 1 1		0 0 0 0 1 0 0 0 0 1 /	**e_crnor**	Condition Register NOR	*-922*
X	0 1 1 1 1 1		0 0 0 0 1 0 1 0 0 0 0	**subf**	Subtract From	Book E
X	0 1 1 1 1 1		0 0 0 0 1 0 1 0 0 0 1	**subf.**	Subtract From & record CR	Book E
X	0 1 1 1 1 1		0 0 0 0 1 0 1 1 1 0 /	**e_cmphl**	Compare Halfword Logical	*-916*
X	0 1 1 1 1 1		0 0 0 0 1 1 0 1 1 0 /	**dcbst**	Data Cache Block Store Indexed	Book E
X	0 1 1 1 1 1		0 0 0 0 1 1 0 1 1 1 /	**lwzux**	Load Word & Zero with Update Indexed	Book E
X	0 1 1 1 1 1		0 0 0 0 1 1 1 0 0 0 0	**e_slwi**	Shift Left Word Immediate	*-955*
X	0 1 1 1 1 1		0 0 0 0 1 1 1 0 0 0 1	**e_slwi.**	Shift Left Word Immediate & record CR	*-955*
X	0 1 1 1 1 1		0 0 0 0 1 1 1 1 0 0 0	**andc**	AND with Complement	Book E
X	0 1 1 1 1 1		0 0 0 0 1 1 1 1 0 0 1	**andc.**	AND with Complement & record CR	Book E

Table 262. 32-bit instruction encodings (continued)

Format	Opcode			Mnemonic	Instruction	Page
	Primary ($Inst_{0:5}$)	Intermediate ($Inst_{6:20}$)	Extended ($Inst_{21:31}$)			
X	0 1 1 1 1 1		0 0 0 1 0 1 0 0 1 1 /	**mfmsr**	Move From Machine State Register	Book E
X	0 1 1 1 1 1		0 0 0 1 0 1 0 1 1 0 /	**dcbf**	Data Cache Block Flush Indexed	Book E
X	0 1 1 1 1 1		0 0 0 1 0 1 0 1 1 1 /	**lbzx**	Load Byte & Zero Indexed	Book E
X	0 1 1 1 1 1		0 0 0 1 1 0 1 0 0 0 0	**neg**	Negate	Book E
X	0 1 1 1 1 1		0 0 0 1 1 0 1 0 0 0 1	**neg.**	Negate & record CR	Book E
X	0 1 1 1 1 1		0 0 0 1 1 1 0 1 1 1 /	**lbzux**	Load Byte & Zero with Update Indexed	Book E
X	0 1 1 1 1 1		0 0 0 1 1 1 1 1 0 0 0	**nor**	NOR	Book E
X	0 1 1 1 1 1		0 0 0 1 1 1 1 1 0 0 1	**nor.**	NOR & record CR	Book E
XL	0 1 1 1 1 1		0 0 1 0 0 0 0 0 0 1 /	**e_crandc**	Condition Register AND with Complement	*-920*
X	0 1 1 1 1 1		0 0 1 0 0 0 0 0 1 1 /	**wrtee**	Write External Enable	Book E
X	0 1 1 1 1 1		0 0 1 0 0 0 1 0 0 0 0	**subfe**	Subtract From Extended with CA	Book E
X	0 1 1 1 1 1		0 0 1 0 0 0 1 0 0 0 1	**subfe.**	Subtract From Extended with CA & record CR	Book E
X	0 1 1 1 1 1		0 0 1 0 0 0 1 0 1 0 0	**adde**	Add Extended with CA	Book E
X	0 1 1 1 1 1		0 0 1 0 0 0 1 0 1 0 1	**adde.**	Add Extended with CA & record CR	Book E
XFX	0 1 1 1 1 1		0 0 1 0 0 1 0 0 0 0 /	**mtcrf**	Move To Condition Register Fields	Book E
X	0 1 1 1 1 1		0 0 1 0 0 1 0 0 1 0 /	**mtmsr**	Move To Machine State Register	Book E

Table 262. 32-bit instruction encodings (continued)

Format	Opcode			Mnemonic	Instruction	Page
	Primary ($Inst_{0:5}$)	Intermediate ($Inst_{6:20}$)	Extended ($Inst_{21:31}$)			
X	0 1 1 1 1 1		0 0 1 0 0 1 0 1 1 0 1	**stwcx.**	Store Word Conditional Indexed & record CR	Book E
X	0 1 1 1 1 1		0 0 1 0 0 1 0 1 1 1 /	**stwx**	Store Word Indexed	Book E
X	0 1 1 1 1 1		0 0 1 0 1 0 0 0 1 1 /	**wrteei**	Write External Enable Immediate	Book E
X	0 1 1 1 1 1		0 0 1 0 1 1 0 1 1 1 /	**stwux**	Store Word with Update Indexed	Book E
XL	0 1 1 1 1 1		0 0 1 1 0 0 0 0 1 /	**e_crxor**	Condition Register XOR	*-925*
X	0 1 1 1 1 1		0 0 1 1 0 0 1 0 0 0	**subfze**	Subtract From Zero Extended with CA	Book E
X	0 1 1 1 1 1		0 0 1 1 0 0 1 0 0 0 1	**subfze.**	Subtract From Zero Extended with CA & record CR	Book E
X	0 1 1 1 1 1		0 0 1 1 0 0 1 0 1 0	**addze**	Add to Zero Extended with CA	Book E
X	0 1 1 1 1 1		0 0 1 1 0 0 1 0 1 0 1	**addze.**	Add to Zero Extended with CA & record CR	Book E
X	0 1 1 1 1 1		0 0 1 1 0 1 0 1 1 1 /	**stbx**	Store Byte Indexed	Book E
XL	0 1 1 1 1 1		0 0 1 1 1 0 0 0 1 /	**e_crnand**	Condition Register NAND	Book E
X	0 1 1 1 1 1		0 0 1 1 1 0 1 0 0 0	**subfme**	Subtract From Minus One Extended with CA	Book E
X	0 1 1 1 1 1		0 0 1 1 1 0 1 0 0 0 1	**subfme.**	Subtract From Minus One Extended with CA & record CR	Book E
X	0 1 1 1 1 1		0 0 1 1 1 0 1 0 1 0	**addme**	Add to Minus One Extended with CA	Book E

Table 262. 32-bit instruction encodings (continued)

Format	Opcode			Mnemonic	Instruction	Page
	Primary ($Inst_{0:5}$)	Intermediate ($Inst_{6:20}$)	Extended ($Inst_{21:31}$)			
X	0 1 1 1 1 1		0 0 1 1 1 0 1 0 1 0 1	**addme.**	Add to Minus One Extended with CA & record CR	Book E
X	0 1 1 1 1 1		0 0 1 1 1 0 1 0 1 1 0	**mullw**	Multiply Low Word	Book E
X	0 1 1 1 1 1		0 0 1 1 1 0 1 0 1 1 1	**mullw.**	Multiply Low Word & record CR	Book E
X	0 1 1 1 1 1		0 0 1 1 1 1 0 1 1 0 /	**dcbtst**	Data Cache Block Touch for Store Indexed	Book E
X	0 1 1 1 1 1		0 0 1 1 1 1 0 1 1 1 /	**stbux**	Store Byte with Update Indexed	Book E
XL	0 1 1 1 1 1		0 1 0 0 0 0 0 0 0 1 /	**e_crand**	Condition Register AND	*-920*
X	0 1 1 1 1 1		0 1 0 0 0 0 1 0 1 0 0	**add**	Add	Book E
X	0 1 1 1 1 1		0 1 0 0 0 0 1 0 1 0 1	**add.**	Add & record CR	Book E
X	0 1 1 1 1 1		0 1 0 0 0 1 0 0 1 1 /	**mfapidi**	Move From APID Indirect	Book E
X	0 1 1 1 1 1		0 1 0 0 0 1 0 1 1 0 /	**dcbt**	Data Cache Block Touch Indexed	Book E
X	0 1 1 1 1 1		0 1 0 0 0 1 0 1 1 1 /	**lhzx**	Load Halfword & Zero Indexed	Book E
X	0 1 1 1 1 1		0 1 0 0 0 1 1 0 0 0 0	**e_rlw**	Rotate Left Word	*-951*
X	0 1 1 1 1 1		0 1 0 0 0 1 1 0 0 0 1	**e_rlw.**	Rotate Left Word & record CR	*-951*
X	0 1 1 1 1 1		0 1 0 0 0 1 1 1 0 0 0	**eqv**	Equivalent	Book E
X	0 1 1 1 1 1		0 1 0 0 0 1 1 1 0 0 1	**eqv.**	Equivalent & record CR	Book E
XL	0 1 1 1 1 1		0 1 0 0 1 0 0 0 0 1 /	**e_creqv**	Condition Register Equivalent	*-920*
X	0 1 1 1 1 1		0 1 0 0 1 1 0 1 1 1 /	**lhzux**	Load Halfword & Zero with Update Indexed	Book E

Table 262. 32-bit instruction encodings (continued)

Format	Opcode			Mnemonic	Instruction	Page
	Primary ($Inst_{0:5}$)	Intermediate ($Inst_{6:20}$)	Extended ($Inst_{21:31}$)			
X	0 1 1 1 1 1		0 1 0 0 1 1 1 0 0 0 0	**e_rlwi**	Rotate Left Word Immediate	-952
X	0 1 1 1 1 1		0 1 0 0 1 1 1 0 0 0 1	**e_rlwi.**	Rotate Left Word Immediate & record CR	-952
X	0 1 1 1 1 1		0 1 0 0 1 1 1 1 0 0 0	**xor**	XOR	Book E
X	0 1 1 1 1 1		0 1 0 0 1 1 1 1 0 0 1	**xor.**	XOR & record CR	Book E
XFX	0 1 1 1 1 1		0 1 0 1 0 0 0 0 1 1 /	**mfdcr**	Move From Device Control Register	Book E
XFX	0 1 1 1 1 1		0 1 0 1 0 1 0 0 1 1 /	**mfspr**	Move From Special Purpose Register	Book E
X	0 1 1 1 1 1		0 1 0 1 0 1 0 1 1 1 /	**lhax**	Load Halfword Algebraic Indexed	Book E
X	0 1 1 1 1 1		0 1 0 1 1 1 0 1 1 1 /	**lhaux**	Load Halfword Algebraic with Update Indexed	Book E
X	0 1 1 1 1 1		0 1 1 0 0 1 0 1 1 1 /	**sthx**	Store Halfword Indexed	Book E
X	0 1 1 1 1 1		0 1 1 0 0 1 1 1 0 0 0	**orc**	OR with Complement	Book E
X	0 1 1 1 1 1		0 1 1 0 0 1 1 1 0 0 1	**orc.**	OR with Complement & record CR	Book E
XL	0 1 1 1 1 1		0 1 1 0 1 0 0 0 0 1 /	**e_crorc**	Condition Register OR with Complement	-923
X	0 1 1 1 1 1		0 1 1 0 1 1 0 1 1 1 /	**sthux**	Store Halfword with Update Indexed	Book E
X	0 1 1 1 1 1		0 1 1 0 1 1 1 1 0 0 0	**or**	OR	Book E
X	0 1 1 1 1 1		0 1 1 0 1 1 1 1 0 0 1	**or.**	OR & record CR	Book E
XL	0 1 1 1 1 1		0 1 1 1 0 0 0 0 0 1 /	**e_cror**	Condition Register OR	-923

Table 262. 32-bit instruction encodings (continued)

Format	Opcode			Mnemonic	Instruction	Page
	Primary ($Inst_{0:5}$)	Intermediate ($Inst_{6:20}$)	Extended ($Inst_{21:31}$)			
XFX	0 1 1 1 1 1		0 1 1 1 0 0 0 0 1 1 /	**mtdcr**	Move To Device Control Register	Book E
X	0 1 1 1 1 1		0 1 1 1 0 0 1 0 1 1 0	**divwu**	Divide Word Unsigned	Book E
X	0 1 1 1 1 1		0 1 1 1 0 0 1 0 1 1 1	**divwu.**	Divide Word Unsigned & record CR	Book E
XFX	0 1 1 1 1 1		0 1 1 1 0 1 0 0 1 1 /	**mtspr**	Move To Special Purpose Register	Book E
X	0 1 1 1 1 1		0 1 1 1 0 1 0 1 1 0 /	**dcbi**	Data Cache Block Invalidate Indexed	Book E
X	0 1 1 1 1 1		0 1 1 1 0 1 1 1 0 0 0	**nand**	NAND	Book E
X	0 1 1 1 1 1		0 1 1 1 0 1 1 1 0 0 1	**nand.**	NAND & record CR	Book E
X	0 1 1 1 1 1		0 1 1 1 1 0 1 0 1 1 0	**divw**	Divide Word	Book E
X	0 1 1 1 1 1		0 1 1 1 1 0 1 0 1 1 1	**divw.**	Divide Word & record CR	Book E
X	0 1 1 1 1 1		1 0 0 0 0 0 0 0 0 0 /	**mcrxr**	Move to Condition Register from XER	Book E
X	0 1 1 1 1 1		1 0 0 0 0 0 1 0 0 0 0	**subfco**	Subtract From Carrying & record OV	Book E
X	0 1 1 1 1 1		1 0 0 0 0 0 1 0 0 0 1	**subfco.**	Subtract From Carrying & record OV & CR	Book E
X	0 1 1 1 1 1		1 0 0 0 0 0 1 0 1 0 0	**addco**	Add Carrying & record OV	Book E
X	0 1 1 1 1 1		1 0 0 0 0 0 1 0 1 0 1	**addco.**	Add Carrying & record OV & CR	Book E
X	0 1 1 1 1 1		1 0 0 0 0 1 0 1 1 0 /	**lwbrx**	Load Word Byte-Reverse Indexed	Book E
X	0 1 1 1 1 1		1 0 0 0 0 1 1 0 0 0 0	**srw**	Shift Right Word	Book E
X	0 1 1 1 1 1		1 0 0 0 0 1 1 0 0 0 1	**srw.**	Shift Right Word & record CR	Book E

Table 262. 32-bit instruction encodings (continued)

Format	Opcode			Mnemonic	Instruction	Page
	Primary ($Inst_{0:5}$)	Intermediate ($Inst_{6:20}$)	Extended ($Inst_{21:31}$)			
X	0 1 1 1 1 1		1 0 0 0 1 0 1 0 0 0 0	**subfo**	Subtract From & record OV	Book E
X	0 1 1 1 1 1		1 0 0 0 1 0 1 0 0 0 1	**subfo.**	Subtract From & record OV & CR	Book E
X	0 1 1 1 1 1		1 0 0 0 1 1 0 1 1 0 /	**tlbsync**	TLB Synchronize	Book E
X	0 1 1 1 1 1		1 0 0 0 1 1 1 0 0 0 0	**e_srwi**	Shift Right Word Immediate	-957
X	0 1 1 1 1 1		1 0 0 0 1 1 1 0 0 0 1	**e_srwi.**	Shift Right Word Immediate & record CR	-957
X	0 1 1 1 1 1		1 0 0 1 0 1 0 1 1 0 /	**msync**	Memory Synchronize	Book E
X	0 1 1 1 1 1		1 0 0 1 1 0 1 0 0 0 0	**nego**	Negate & record OV	Book E
X	0 1 1 1 1 1		1 0 0 1 1 0 1 0 0 0 1	**nego.**	Negate & record OV & record CR	Book E
X	0 1 1 1 1 1		1 0 1 0 0 0 1 0 0 0 0	**subfeo**	Subtract From Extended with CA & record OV	Book E
X	0 1 1 1 1 1		1 0 1 0 0 0 1 0 0 0 1	**subfeo.**	Subtract From Extended with CA & record OV & CR	Book E
X	0 1 1 1 1 1		1 0 1 0 0 0 1 0 1 0 0	**addeo**	Add Extended with CA & record OV	Book E
X	0 1 1 1 1 1		1 0 1 0 0 0 1 0 1 0 1	**addeo.**	Add Extended with CA & record OV & CR	Book E
X	0 1 1 1 1 1		1 0 1 0 0 1 0 1 0 1 /	**stswx**	Store String Word Indexed	Book E
X	0 1 1 1 1 1		1 0 1 0 0 1 0 1 1 0 /	**stwbrx**	Store Word Byte-Reverse Indexed	Book E
X	0 1 1 1 1 1		1 0 1 1 0 0 1 0 0 0 0	**subfzeo**	Subtract From Zero Extended with CA & record OV	Book E
X	0 1 1 1 1 1		1 0 1 1 0 0 1 0 0 0 1	**subfzeo.**	Subtract From Zero Extended with CA & record OV & CR	Book E

Table 262. 32-bit instruction encodings (continued)

Format	Opcode			Mnemonic	Instruction	Page
	Primary ($Inst_{0:5}$)	Intermediate ($Inst_{6:20}$)	Extended ($Inst_{21:31}$)			
X	0 1 1 1 1 1		1 0 1 1 0 0 1 0 1 0 0	**addzeo**	Add to Zero Extended with CA & record OV	Book E
X	0 1 1 1 1 1		1 0 1 1 0 0 1 0 1 0 1	**addzeo.**	Add to Zero Extended with CA & record OV & CR	Book E
X	0 1 1 1 1 1		1 0 1 1 0 1 0 1 0 1 /	**stswi**	Store String Word Immediate	Book E
X	0 1 1 1 1 1		1 0 1 1 1 0 1 0 0 0 0	**subfmeo**	Subtract From Minus One Extended with CA & record OV	Book E
X	0 1 1 1 1 1		1 0 1 1 1 0 1 0 0 0 1	**subfmeo.**	Subtract From Minus One Extended with CA & record OV & CR	Book E
X	0 1 1 1 1 1		1 0 1 1 1 0 1 0 1 0 0	**addmeo**	Add to Minus One Extended with CA & record OV	Book E
X	0 1 1 1 1 1		1 0 1 1 1 0 1 0 1 0 1	**addmeo.**	Add to Minus One Extended with CA & record OV & CR	Book E
X	0 1 1 1 1 1		1 0 1 1 1 0 1 0 1 1 0	**mullwo**	Multiply Low Word & record OV	Book E
X	0 1 1 1 1 1		1 0 1 1 1 0 1 0 1 1 1	**mullwo.**	Multiply Low Word & record OV & CR	Book E
X	0 1 1 1 1 1		1 0 1 1 1 1 0 1 1 0 /	**dcba**	Data Cache Block Allocate Indexed	Book E
X	0 1 1 1 1 1		1 1 0 0 0 0 1 0 1 0 0	**addo**	Add & record OV	Book E
X	0 1 1 1 1 1		1 1 0 0 0 0 1 0 1 0 1	**addo.**	Add & record OV & CR	Book E
X	0 1 1 1 1 1		1 1 0 0 0 1 0 0 1 0 /	**tlbivax**	TLB Invalidate Virtual Address Indexed	Book E

Table 262. 32-bit instruction encodings (continued)

Format	Opcode			Mnemonic	Instruction	Page
	Primary ($Inst_{0:5}$)	Intermediate ($Inst_{6:20}$)	Extended ($Inst_{21:31}$)			
X	0 1 1 1 1 1		1 1 0 0 0 1 0 1 1 0 /	**lhbrx**	Load Halfword Byte-Reverse Indexed	Book E
X	0 1 1 1 1 1		1 1 0 0 0 1 1 0 0 0 0	**sraw**	Shift Right Algebraic Word	Book E
X	0 1 1 1 1 1		1 1 0 0 0 1 1 0 0 0 1	**sraw.**	Shift Right Algebraic Word & record CR	Book E
X	0 1 1 1 1 1		1 1 0 0 1 1 1 0 0 0 0	**srawi**	Shift Right Algebraic Word Immediate	Book E
X	0 1 1 1 1 1		1 1 0 0 1 1 1 0 0 0 1	**srawi.**	Shift Right Algebraic Word Immediate & record CR	Book E
X	0 1 1 1 1 1		1 1 0 1 0 1 0 1 1 0 /	**mbar**	Memory Barrier	Book E
X	0 1 1 1 1 1		1 1 1 0 0 1 0 0 1 0 ?	**tlbsx**	TLB Search Indexed	Book E
X	0 1 1 1 1 1		1 1 1 0 0 1 0 1 1 0 /	**sthbrx**	Store Halfword Byte-Reverse Indexed	Book E
X	0 1 1 1 1 1		1 1 1 0 0 1 1 0 1 0 0	**extsh**	Extend Sign Halfword	Book E
X	0 1 1 1 1 1		1 1 1 0 0 1 1 0 1 0 1	**extsh.**	Extend Sign Halfword & record CR	Book E
X	0 1 1 1 1 1		1 1 1 0 1 1 0 0 1 0 /	**tlbre**	TLB Read Entry	Book E
X	0 1 1 1 1 1		1 1 1 0 1 1 1 0 1 0 0	**extsb**	Extend Sign Byte	Book E
X	0 1 1 1 1 1		1 1 1 0 1 1 1 0 1 0 1	**extsb.**	Extend Sign Byte & record CR	Book E
X	0 1 1 1 1 1		1 1 1 1 0 0 1 0 1 1 0	**divwuo**	Divide Word Unsigned & record OV	Book E
X	0 1 1 1 1 1		1 1 1 1 0 0 1 0 1 1 1	**divwuo.**	Divide Word Unsigned & record OV & CR	Book E
X	0 1 1 1 1 1		1 1 1 1 0 1 0 0 1 0 /	**tlbwe**	TLB Write Entry	Book E

Table 262. 32-bit instruction encodings (continued)

Format	Opcode			Mnemonic	Instruction	Page
	Primary ($Inst_{0:5}$)	Intermediate ($Inst_{6:20}$)	Extended ($Inst_{21:31}$)			
X	0 1 1 1 1 1		1 1 1 1 0 1 0 1 1 0 /	**icbi**	Instruction Cache Block Invalidate Indexed	Book E
X	0 1 1 1 1 1		1 1 1 1 1 0 1 0 1 1 0	**divwo**	Divide Word & Record OV	Book E
X	0 1 1 1 1 1		1 1 1 1 1 0 1 0 1 1 1	**divwo.**	Divide Word & Record OV & CR	Book E
X	0 1 1 1 1 1		1 1 1 1 1 1 0 1 1 0 /	**dcbz**	Data Cache Block Set to Zero Indexed	Book E
X	1 1 1 1 - -			Reserved		

14.2 Instruction index sorted by mnemonic

Table 263 lists all of the 16-bit VLE instructions, sorted by mnemonic.

Table 263. 16-Bit VLE instructions sorted by mnemonic

Format	16-Bit Opcodes ($Inst_{0:15}$)	Mnemonic	Instruction	Page
RR	0 0 0 0 0 1 0 0 y y y y x x x x	**se_add**	Add	*-897*
IM5	0 0 1 0 0 0 0 i i i i i x x x x	**se_addi**	Add Immediate	*-897*
RR	0 1 0 0 0 1 1 0 y y y y x x x x	**se_and**	AND	*-901*
RR	0 1 0 0 0 1 1 1 y y y y x x x x	**se_and.**	AND and Record	*-901*
RR	0 1 0 0 0 1 0 1 y y y y x x x x	**se_andc**	AND with Complement	*-901*
IM5	0 0 1 0 1 1 1 i i i i i x x x x	**se_andi**	And Immediate	*-901*
B8	1 1 1 0 0 o i i d d d d d d d d	**se_bc**	Branch Conditional	*-904*
IM5	0 1 1 0 0 0 0 i i i i i x x x x	**se_bclri**	Bit Clear Immediate	*-905*
C	0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0	**se_bctr**	Branch to Count Register	*-905*
C	0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1	**se_bctrl**	Branch to Count Register & Link	*-905*
IM5	0 1 1 0 0 0 1 i i i i i x x x x	**se_bgeni**	Bit Generate Immediate	*-906*
B8	1 1 1 0 1 0 0 1 d d d d d d d d	**se_bl**	Branch and Link	*-910*
C	0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0	**se_blr**	Branch to Link Register	*-908*
C	0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1	**se_blrl**	Branch to Link Register & Link	*-908*
IM5	0 0 1 0 1 1 0 i i i i i x x x x	**se_bmaski**	Bit Mask Generate Immediate	*-909*
B8	1 1 1 0 1 0 0 0 d d d d d d d d	**se_b**	Branch	*-903*
IM5	0 1 1 0 0 1 0 i i i i i x x x x	**se_bseti**	Bit Set Immediate	*-910*
IM5	0 1 1 0 0 1 1 i i i i i x x x x	**se_btsti**	Bit Test Immediate	*-911*
RR	0 0 0 0 1 1 0 0 y y y y x x x x	**se_cmp**	Compare	*-912*
RR	0 0 0 0 1 1 1 0 y y y y x x x x	**se_cmph**	Compare Halfword	*-914*
RR	0 0 0 0 1 1 1 1 y y y y x x x x	**se_cmphl**	Compare Halfword Logical	*-916*
IM5	0 0 1 0 1 0 1 i i i i i x x x x	**se_cmpi**	Compare Immediate	*-912*
RR	0 0 0 0 1 1 0 1 y y y y x x x x	**se_cmpl**	Compare Logical	*-918*
IM5	0 0 1 0 0 0 1 i i i i i x x x x	**se_cmpli**	Compare Logical Immediate	*-918*
R	0 0 0 0 0 0 0 0 1 1 0 1 x x x x	**se_extsb**	Extend Sign Byte	*-926*
R	0 0 0 0 0 0 0 0 1 1 1 1 x x x x	**se_extsh**	Extend Sign Halfword	*-926*
R	0 0 0 0 0 0 0 0 1 1 0 0 x x x x	**se_extzb**	Extend with Zeros Byte	*-927*

Table 263. 16-Bit VLE instructions sorted by mnemonic (continued)

Format	16-Bit Opcodes ($Inst_{0:15}$)	Mnemonic	Instruction	Page
R	0 0 0 0 0 0 0 0 1 1 1 0 x x x x	**se_extzh**	Extend with Zeros Halfword	*-927*
C	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	**se_illegal**	Illegal	*-928*
C	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1	**se_isync**	Instruction Synchronize	*-929*
SD4	1 0 0 0 i i i i z z z z x x x x	**se_lbz**	Load Byte and Zero	*-930*
SD4	1 0 1 0 i i i i z z z z x x x x	**se_lhz**	Load Halfword and Zero	*-932*
IM7	0 1 0 0 1 i i i i i i i x x x x	**se_li**	Load Immediate	*-933*
SD4	1 1 0 0 i i i i z z z z x x x x	**se_lwz**	Load Word and Zero	*-935*
RR	0 0 0 0 0 0 1 1 y y y y x x x x	**se_mfar**	Move from Alternate Register	*-937*
R	0 0 0 0 0 0 0 0 1 0 1 0 x x x x	**se_mfctr**	Move From Count Register	*-938*
R	0 0 0 0 0 0 0 0 1 0 0 0 x x x x	**se_mflr**	Move From Link Register	*-939*
RR	0 0 0 0 0 0 0 1 y y y y x x x x	**se_mr**	Move Register	*-940*
RR	0 0 0 0 0 0 1 0 y y y y x x x x	**se_mtar**	Move to Alternate Register	*-941*
R	0 0 0 0 0 0 0 0 1 0 1 1 x x x x	**se_mtctr**	Move To Count Register	*-942*
R	0 0 0 0 0 0 0 0 1 0 0 1 x x x x	**se_mtlr**	Move To Link Register	*-943*
RR	0 0 0 0 0 1 0 1 y y y y x x x x	**se_mullw**	Multiply Low Word	*-945*
R	0 0 0 0 0 0 0 0 0 0 1 1 x x x x	**se_neg**	Negate	*-946*
R	0 0 0 0 0 0 0 0 0 0 1 0 x x x x	**se_not**	NOT	*-947*
RR	0 1 0 0 0 1 0 0 y y y y x x x x	**se_or**	OR	*-948*
C	0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1	**se_rfci**	Return From Critical Interrupt	*-949*
C	0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0	**se_rfdi**	Return From Debug Interrupt	*-859*
C	0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0	**se_rfi**	Return From Interrupt	*-950*
C	0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0	**se_sc**	System Call	*-954*
RR	0 1 0 0 0 0 1 0 y y y y x x x x	**se_slw**	Shift Left Word	*-955*
IM5	0 1 1 0 1 1 0 i i i i i x x x x	**se_slwi**	Shift Left Word Immediate	*-955*
RR	0 1 0 0 0 0 0 1 y y y y x x x x	**se_sraw**	Shift Right Algebraic Word	*-956*
IM5	0 1 1 0 1 0 1 i i i i i x x x x	**se_srawi**	Shift Right Algebraic Word Immediate	*-956*
RR	0 1 0 0 0 0 0 0 y y y y x x x x	**se_srw**	Shift Right Word	*-957*
IM5	0 1 1 0 1 0 0 i i i i i x x x x	**se_srwi**	Shift Right Word Immediate	*-957*

Table 263. 16-Bit VLE instructions sorted by mnemonic (continued)

Format	16-Bit Opcodes ($Inst_{0:15}$)	Mnemonic	Instruction	Page
SD4	1 0 0 1 i i i i z z z z x x x x	**se_stb**	Store Byte	*-958*
SD4	1 0 1 1 i i i i z z z z x x x x	**se_sth**	Store Halfword	*-959*
SD4	1 1 0 1 i i i i z z z z x x x x	**se_stw**	Store Word	*-961*
RR	0 0 0 0 0 1 1 0 y y y y x x x x	**se_sub**	Subtract	*-962*
RR	0 0 0 0 0 1 1 1 y y y y x x x x	**se_subf**	Subtract From	*-963*
IM5	0 0 1 0 0 1 0 i i i i i x x x x	**se_subi**	Subtract Immediate	*-965*
IM5	0 0 1 0 0 1 1 i i i i i x x x x	**se_subi.**	Subtract Immediate and Record	*-962*

Table 264 outlines the 32-bit instruction encodings.

Table 264. 32-bit instruction encodings (by mnemonic)

Format	Opcode			Mnemonic	Instruction	Page
	Primary ($Inst_{0:5}$)	Intermediate ($Inst_{6:20}$)	Extended ($Inst_{21:31}$)			
X	0 1 1 1 1 1		0 1 0 0 0 0 1 0 1 0 0	**add**	Add	Book E
X	0 1 1 1 1 1		0 1 0 0 0 0 1 0 1 0 1	**add.**	Add & record CR	Book E
D	0 0 0 1 1 1	t t t t t a a a a a i i i i i	i i i i i i i i i i i	**e_add16i**	Add Immediate	*-897*
I16A	0 1 1 1 0 0	i i i i i a a a a a 1 0 0 0 1	i i i i i i i i i i i	**e_add2i.**	Add (2 operand) Immediate and Record CR	*-897*
I16A	0 1 1 1 0 0	i i i i i a a a a a 1 0 0 1 0	i i i i i i i i i i i	**e_add2is**	Add (2 operand) Immediate Shifted	*-897*
X	0 1 1 1 1 1		0 0 0 0 0 0 1 0 1 0 0	**addc**	Add Carrying	Book E
X	0 1 1 1 1 1		0 0 0 0 0 0 1 0 1 0 1	**addc.**	Add Carrying & record CR	Book E
X	0 1 1 1 1 1		1 0 0 0 0 0 1 0 1 0 0	**addco**	Add Carrying & record OV	Book E
X	0 1 1 1 1 1		1 0 0 0 0 0 1 0 1 0 1	**addco.**	Add Carrying & record OV & CR	Book E
X	0 1 1 1 1 1		0 0 1 0 0 0 1 0 1 0 0	**adde**	Add Extended with CA	Book E
X	0 1 1 1 1 1		0 0 1 0 0 0 1 0 1 0 1	**adde.**	Add Extended with CA & record CR	Book E

Table 264. 32-bit instruction encodings (by mnemonic) (continued)

Format	Opcode			Mnemonic	Instruction	Page
	Primary ($Inst_{0:5}$)	Intermediate ($Inst_{6:20}$)	Extended ($Inst_{21:31}$)			
X	0 1 1 1 1 1		1 0 1 0 0 0 1 0 1 0 0	**addeo**	Add Extended with CA & record OV	Book E
X	0 1 1 1 1 1		1 0 1 0 0 0 1 0 1 0 1	**addeo.**	Add Extended with CA & record OV & CR	Book E
SCI8	0 0 0 1 1 0	t t t t t a a a a a 1 0 0 0 0	F S S i i i i i i i i	**e_addi**	Add Immediate	-897
SCI8	0 0 0 1 1 0	t t t t t a a a a a 1 0 0 0 1	F S S i i i i i i i i	**e_addi.**	Add Immediate and Record	-897
SCI8	0 0 0 1 1 0	t t t t t a a a a a 1 0 0 1 0	F S S i i i i i i i i	**e_addic**	Add Immediate Carrying	-900
SCI8	0 0 0 1 1 0	t t t t t a a a a a 1 0 0 1 1	F S S i i i i i i i i	**e_addic.**	Add Immediate Carrying and Record	-900
X	0 1 1 1 1 1		0 0 1 1 1 0 1 0 1 0 0	**addme**	Add to Minus One Extended with CA	Book E
X	0 1 1 1 1 1		0 0 1 1 1 0 1 0 1 0 1	**addme.**	Add to Minus One Extended with CA & record CR	Book E
X	0 1 1 1 1 1		1 0 1 1 1 0 1 0 1 0 0	**addmeo**	Add to Minus One Extended with CA & record OV	Book E
X	0 1 1 1 1 1		1 0 1 1 1 0 1 0 1 0 1	**addmeo.**	Add to Minus One Extended with CA & record OV & CR	Book E
X	0 1 1 1 1 1		1 1 0 0 0 0 1 0 1 0 0	**addo**	Add & record OV	Book E
X	0 1 1 1 1 1		1 1 0 0 0 0 1 0 1 0 1	**addo.**	Add & record OV & CR	Book E
X	0 1 1 1 1 1		0 0 1 1 0 0 1 0 1 0 0	**addze**	Add to Zero Extended with CA	Book E
X	0 1 1 1 1 1		0 0 1 1 0 0 1 0 1 0 1	**addze.**	Add to Zero Extended with CA & record CR	Book E
X	0 1 1 1 1 1		1 0 1 1 0 0 1 0 1 0 0	**addzeo**	Add to Zero Extended with CA & record OV	Book E

Table 264. 32-bit instruction encodings (by mnemonic) (continued)

Format	Opcode			Mnemonic	Instruction	Page
	Primary ($Inst_{0:5}$)	Intermediate ($Inst_{6:20}$)	Extended ($Inst_{21:31}$)			
X	0 1 1 1 1 1		1 0 1 1 0 0 1 0 1 0 1	**addzeo.**	Add to Zero Extended with CA & record OV & CR	Book E
X	0 1 1 1 1 1		0 0 0 0 0 1 1 1 0 0 0	**and**	AND	Book E
X	0 1 1 1 1 1		0 0 0 0 0 1 1 1 0 0 1	**and.**	AND & record CR	Book E
I16L	0 1 1 1 0 0	t t t t t i i i i i 1 1 0 0 1	i i i i i i i i i i i	**e_and2i.**	AND (2 operand) Immediate & record CR	*-901*
I16L	0 1 1 1 0 0	t t t t t i i i i i 1 1 1 0 1	i i i i i i i i i i i	**e_and2is.**	AND (2 operand) Immediate Shifted & record CR	*-901*
X	0 1 1 1 1 1		0 0 0 0 1 1 1 1 0 0 0	**andc**	AND with Complement	Book E
X	0 1 1 1 1 1		0 0 0 0 1 1 1 1 0 0 1	**andc.**	AND with Complement & record CR	Book E
SCI8	0 0 0 1 1 0	s s s s s a a a a a 1 1 0 0 0	F S S i i i i i i i i	**e_andi**	AND Immediate	*-901*
SCI8	0 0 0 1 1 0	s s s s s a a a a a 1 1 0 0 1	F S S i i i i i i i i	**e_andi.**	AND Immediate and Record	*-901*
APU	0 0 0 1 0 -	- - - - - - - - - - - - - - -	- - - - - - - - - - -	**apu**	Reserved for APUs	
BD24	0 1 1 1 1 0	0 d d d d d d d d d d d	d d d d d d d d d d 0	**e_b**	Branch	*-903*
BD15	0 1 1 1 1 0	1 0 0 0 o o i i i i d d d d d	d d d d d d d d d d 0	**e_bc**	Branch Conditional	*-904*
BD15	0 1 1 1 1 0	1 0 0 0 o o i i i i d d d d d	d d d d d d d d d d 1	**e_bcl**	Branch Conditional & Link	*-904*
BD24	0 1 1 1 1 0	0 d d d d d d d d d d d	d d d d d d d d d d 1	**e_bl**	Branch & Link	*-903*
X	0 1 1 1 1 1		0 0 0 0 0 0 0 0 0 0 /	**cmp**	Compare	Book E
I16A	0 1 1 1 0 0	i i i i i a a a a a 1 0 0 1 1	i i i i i i i i i i i	**e_cmp16i**	Compare Immediate	*-912*
X	0 1 1 1 1 1		0 0 0 0 0 0 1 1 1 0 /	**e_cmph**	Compare Halfword	*-914*

Table 264. 32-bit instruction encodings (by mnemonic) (continued)

Format	Opcode			Mnemonic	Instruction	Page
	Primary ($Inst_{0:5}$)	Intermediate ($Inst_{6:20}$)	Extended ($Inst_{21:31}$)			
I16A	0 1 1 1 0 0	i i i i i a a a a a 1 0 1 1 0	i i i i i i i i i i i	**e_cmph16i**	Compare Halfword Immediate	*-914*
X	0 1 1 1 1 1		0 0 0 0 1 0 1 1 1 0 /	**e_cmphl**	Compare Halfword Logical	*-916*
I16A	0 1 1 1 0 0	i i i i i a a a a a 1 0 1 1 1	i i i i i i i i i i i	**e_cmphl16i**	Compare Halfword Logical Immediate	*-916*
SCI8	0 0 0 1 1 0	0 0 0 bf a a a a a 1 0 1 0 1	F S S i i i i i i i	**e_cmpi**	Compare Immediate	*-912*
X	0 1 1 1 1 1		0 0 0 0 1 0 0 0 0 0 /	**cmpl**	Compare Logical	Book E
I16A	0 1 1 1 0 0	i i i i i a a a a a 1 0 1 0 1	i i i i i i i i i i i	**e_cmpl16i**	Compare Logical Immediate	*-918*
SCI8	0 0 0 1 1 0	0 0 1 bf a a a a a 1 0 1 0 1	F S S i i i i i i i	**e_cmpli**	Compare Logical Immediate	*-918*
X	0 1 1 1 1 1		0 0 0 0 0 1 1 0 1 0 0	**cntlzw**	Count Leading Zeros Word	Book E
X	0 1 1 1 1 1		0 0 0 0 0 1 1 0 1 0 1	**cntlzw.**	Count Leading Zeros Word & record CR	Book E
XL	0 1 1 1 1 1		0 1 0 0 0 0 0 0 0 0 1 /	**e_crand**	Condition Register AND	*-920*
XL	0 1 1 1 1 1		0 0 1 0 0 0 0 0 0 0 1 /	**e_crandc**	Condition Register AND with Complement	*-920*
XL	0 1 1 1 1 1		0 1 0 0 1 0 0 0 0 1 /	**e_creqv**	Condition Register Equivalent	*-920*
XL	0 1 1 1 1 1		0 0 1 1 1 0 0 0 0 1 /	**e_crnand**	Condition Register NAND	Book E
XL	0 1 1 1 1 1		0 0 0 0 1 0 0 0 0 1 /	**e_crnor**	Condition Register NOR	*-922*
XL	0 1 1 1 1 1		0 1 1 1 0 0 0 0 0 1 /	**e_cror**	Condition Register OR	*-923*
XL	0 1 1 1 1 1		0 1 1 0 1 0 0 0 0 1 /	**e_crorc**	Condition Register OR with Complement	*-924*

Table 264. 32-bit instruction encodings (by mnemonic) (continued)

Format	Opcode			Mnemonic	Instruction	Page
	Primary ($Inst_{0:5}$)	Intermediate ($Inst_{6:20}$)	Extended ($Inst_{21:31}$)			
XL	0 1 1 1 1 1		0 0 1 1 0 0 0 0 1 /	**e_crxor**	Condition Register XOR	*-925*
X	0 1 1 1 1 1		1 0 1 1 1 1 0 1 1 0 /	**dcba**	Data Cache Block Allocate Indexed	Book E
X	0 1 1 1 1 1		0 0 0 1 0 1 0 1 1 0 /	**dcbf**	Data Cache Block Flush Indexed	Book E
X	0 1 1 1 1 1		0 1 1 1 0 1 0 1 1 0 /	**dcbi**	Data Cache Block Invalidate Indexed	Book E
X	0 1 1 1 1 1		0 0 0 0 1 1 0 1 1 0 /	**dcbst**	Data Cache Block Store Indexed	Book E
X	0 1 1 1 1 1		0 1 0 0 0 1 0 1 1 0 /	**dcbt**	Data Cache Block Touch Indexed	Book E
X	0 1 1 1 1 1		0 0 1 1 1 1 0 1 1 0 /	**dcbtst**	Data Cache Block Touch for Store Indexed	Book E
X	0 1 1 1 1 1		1 1 1 1 1 1 0 1 1 0 /	**dcbz**	Data Cache Block Set to Zero Indexed	Book E
X	0 1 1 1 1 1		0 1 1 1 1 0 1 0 1 1 0	**divw**	Divide Word	Book E
X	0 1 1 1 1 1		0 1 1 1 1 0 1 0 1 1 1	**divw.**	Divide Word & record CR	Book E
X	0 1 1 1 1 1		1 1 1 1 1 0 1 0 1 1 0	**divwo**	Divide Word & Record OV	Book E
X	0 1 1 1 1 1		1 1 1 1 1 0 1 0 1 1 1	**divwo.**	Divide Word & Record OV & CR	Book E
X	0 1 1 1 1 1		0 1 1 1 0 0 1 0 1 1 0	**divwu**	Divide Word Unsigned	Book E
X	0 1 1 1 1 1		0 1 1 1 0 0 1 0 1 1 1	**divwu.**	Divide Word Unsigned & record CR	Book E
X	0 1 1 1 1 1		1 1 1 1 0 0 1 0 1 1 0	**divwuo**	Divide Word Unsigned & record OV	Book E

Table 264. 32-bit instruction encodings (by mnemonic) (continued)

Format	Opcode			Mnemonic	Instruction	Page
	Primary ($Inst_{0:5}$)	Intermediate ($Inst_{6:20}$)	Extended ($Inst_{21:31}$)			
X	011111		11110 01011 1	**divwuo.**	Divide Word Unsigned & record OV & CR	Book E
X	011111		01000 11100 0	**eqv**	Equivalent	Book E
X	011111		01000 11100 1	**eqv.**	Equivalent & record CR	Book E
X	011111		11101 11010 0	**extsb**	Extend Sign Byte	Book E
X	011111		11101 11010 1	**extsb.**	Extend Sign Byte & record CR	Book E
X	011111		11100 11010 0	**extsh**	Extend Sign Halfword	Book E
X	011111		11100 11010 1	**extsh.**	Extend Sign Halfword & record CR	Book E
X	011111		11110 10110 /	**icbi**	Instruction Cache Block Invalidate Indexed	Book E
X	011111		00000 10110 /	**icbt**	Instruction Cache Block Touch Indexed	Book E
X	011111		----- 01111 /	**isel**	Integer Select	Book E
D	001100	ttttt aaaaa ddddd	ddddd ddddd d	**e_lbz**	Load Byte & Zero	*-930*
D8	000110	ttttt aaaaa 00000	000dd ddddd d	**e_lbzu**	Load Byte & Zero with Update	*-930*
X	011111		00011 10111 /	**lbzux**	Load Byte & Zero with Update Indexed	Book E
X	011111		00010 10111 /	**lbzx**	Load Byte & Zero Indexed	Book E
D	001110	ttttt aaaaa ddddd	ddddd ddddd d	**e_lha**	Load Halfword Algebraic	*-931*
D8	000110	ttttt aaaaa 00000	011dd ddddd d	**e_lhau**	Load Halfword Algebraic With Update	*-931*

Table 264. 32-bit instruction encodings (by mnemonic) (continued)

Format	Opcode			Mnemonic	Instruction	Page
	Primary ($Inst_{0:5}$)	Intermediate ($Inst_{6:20}$)	Extended ($Inst_{21:31}$)			
X	0 1 1 1 1 1		0 1 0 1 1 10111 /	**lhaux**	Load Halfword Algebraic with Update Indexed	Book E
X	0 1 1 1 1 1		0 1 0 1 0 10111 /	**lhax**	Load Halfword Algebraic Indexed	Book E
X	0 1 1 1 1 1		1 1 0 0 0 10110 /	**lhbrx**	Load Halfword Byte-Reverse Indexed	Book E
D	0 1 0 1 1 0	t t t t t a a a a a	d d d d d d d d d d d	**e_lhz**	Load Halfword & Zero	*-932*
D8	0 0 0 1 1 0	t t t t t a a a a a	0 0 0 0 0 0 0 1 d d	**e_lhzu**	Load Halfword & Zero with Update	*-932*
X	0 1 1 1 1 1		0 1 0 0 1 10111 /	**lhzux**	Load Halfword & Zero with Update Indexed	Book E
X	0 1 1 1 1 1		0 1 0 0 0 10111 /	**lhzx**	Load Halfword & Zero Indexed	Book E
LI20	0 1 1 1 0 0	t t t t t i i i i i	0 i i i i i i i i i	**e_li**	Load Immediate	*-933*
I16L	0 1 1 1 0 0	t t t t t i i i i i	1 1 1 0 0 i i i i i	**e_lis**	Load Immediate Shifted	*-933*
D8	0 0 0 1 1 0	t t t t t a a a a a	0 0 0 0 1 0 0 0 d d	**e_lmw**	Load Multiple Word	*-935*
X	0 1 1 1 1 1		0 0 0 0 0 10100 /	**lwarx**	Load Word & Reserve Indexed	Book E
X	0 1 1 1 1 1		1 0 0 0 0 10110 /	**lwbrx**	Load Word Byte-Reverse Indexed	Book E
D	0 1 0 1 0 0	t t t t t a a a a a	d d d d d d d d d d d	**e_lwz**	Load Word & Zero	*-935*
D8	0 0 0 1 1 0	t t t t t a a a a a	0 0 0 0 0 0 1 0 d d	**e_lwzu**	Load Word & Zero with Update	*-935*
X	0 1 1 1 1 1		0 0 0 0 1 10111 /	**lwzux**	Load Word & Zero with Update Indexed	Book E
X	0 1 1 1 1 1		0 0 0 0 0 10111 /	**lwzx**	Load Word & Zero Indexed	Book E
X	0 1 1 1 1 1		1 1 0 1 0 10110 /	**mbar**	Memory Barrier	Book E

Table 264. 32-bit instruction encodings (by mnemonic) (continued)

Format	Opcode			Mnemonic	Instruction	Page
	Primary ($Inst_{0:5}$)	Intermediate ($Inst_{6:20}$)	Extended ($Inst_{21:31}$)			
XL	0 1 1 1 1 1		0 0 0 0 0 1 0 0 0 0 /	**e_mcrf**	Move Condition Register Field	*-936*
X	0 1 1 1 1 1		1 0 0 0 0 0 0 0 0 0 /	**mcrxr**	Move to Condition Register from XER	Book E
X	0 1 1 1 1 1		0 1 0 0 0 1 0 0 1 1 /	**mfapidi**	Move From APID Indirect	Book E
X	0 1 1 1 1 1		0 0 0 0 0 1 0 0 1 1 /	**mfcr**	Move From Condition Register	Book E
XFX	0 1 1 1 1 1		0 1 0 1 0 0 0 0 1 1 /	**mfdcr**	Move From Device Control Register	Book E
X	0 1 1 1 1 1		0 0 0 1 0 1 0 0 1 1 /	**mfmsr**	Move From Machine State Register	Book E
XFX	0 1 1 1 1 1		0 1 0 1 0 1 0 0 1 1 /	**mfspr**	Move From Special Purpose Register	Book E
X	0 1 1 1 1 1		1 0 0 1 0 1 0 1 1 0 /	**msync**	Memory Synchronize	Book E
XFX	0 1 1 1 1 1		0 0 1 0 0 1 0 0 0 0 /	**mtcrf**	Move To Condition Register Fields	Book E
XFX	0 1 1 1 1 1		0 1 1 1 0 0 0 0 1 1 /	**mtdcr**	Move To Device Control Register	Book E
X	0 1 1 1 1 1		0 0 1 0 0 1 0 0 1 0 /	**mtmsr**	Move To Machine State Register	Book E
XFX	0 1 1 1 1 1		0 1 1 1 0 1 0 0 1 1 /	**mtspr**	Move To Special Purpose Register	Book E
X	0 1 1 1 1 1		/ 0 0 1 0 0 1 0 1 1 0	**mulhw**	Multiply High Word	Book E
X	0 1 1 1 1 1		/ 0 0 1 0 0 1 0 1 1 1	**mulhw.**	Multiply High Word & record CR	Book E
X	0 1 1 1 1 1		/ 0 0 0 0 0 1 0 1 1 0	**mulhwu**	Multiply High Word Unsigned	Book E

Table 264. 32-bit instruction encodings (by mnemonic) (continued)

Format	Opcode			Mnemonic	Instruction	Page
	Primary ($Inst_{0:5}$)	Intermediate ($Inst_{6:20}$)	Extended ($Inst_{21:31}$)			
X	011111		/ 0000 01011 1	**mulhwu.**	Multiply High Word Unsigned & Record	Book E
I16A	011100	iiiii aaaaa 10100	iiiii iiiii i	**e_mull2i**	Multiply Low Word (2 operand) Immediate	*-944*
SCI8	000110	ttttt aaaaa 10100	FSSii iiiii i	**e_mulli**	Multiply Low Immediate	*-944*
X	011111		00111 01011 0	**mullw**	Multiply Low Word	Book E
X	011111		00111 01011 1	**mullw.**	Multiply Low Word & record CR	Book E
X	011111		10111 01011 0	**mullwo**	Multiply Low Word & record OV	Book E
X	011111		10111 01011 1	**mullwo.**	Multiply Low Word & record OV & CR	Book E
X	011111		01110 11100 0	**nand**	NAND	Book E
X	011111		01110 11100 1	**nand.**	NAND & record CR	Book E
X	011111		00011 01000 0	**neg**	Negate	Book E
X	011111		00011 01000 1	**neg.**	Negate & record CR	Book E
X	011111		10011 01000 0	**nego**	Negate & record OV	Book E
X	011111		10011 01000 1	**nego.**	Negate & record OV & record CR	Book E
X	011111		00011 11100 0	**nor**	NOR	Book E
X	011111		00011 11100 1	**nor.**	NOR & record CR	Book E
X	011111		01101 11100 0	**or**	OR	Book E
X	011111		01101 11100 1	**or.**	OR & record CR	Book E

Table 264. 32-bit instruction encodings (by mnemonic) (continued)

Format	Opcode			Mnemonic	Instruction	Page
	Primary ($Inst_{0:5}$)	Intermediate ($Inst_{6:20}$)	Extended ($Inst_{21:31}$)			
I16L	0 1 1 1 0 0	t t t t t i i i i i 1 1 0 0 0	i i i i i i i i i i i	**e_or2i**	OR (2 operand) Immediate	*-948*
I16L	0 1 1 1 0 0	t t t t t i i i i i 1 1 0 1 0	i i i i i i i i i i i	**e_or2is**	OR (2 operand) Immediate Shifted	*-948*
X	0 1 1 1 1 1		0 1 1 0 0 1 1 1 0 0 0	**orc**	OR with Complement	Book E
X	0 1 1 1 1 1		0 1 1 0 0 1 1 1 0 0 1	**orc.**	OR with Complement & record CR	Book E
SCI8	0 0 0 1 1 0	s s s s s a a a a a 1 1 0 1 0	F S S i i i i i i i i	**e_ori**	OR Immediate	*-951*
SCI8	0 0 0 1 1 0	s s s s s a a a a a 1 1 0 1 1	F S S i i i i i i i i	**e_ori.**	OR Immediate and Record	*-951*
X	1 1 1 1 - -			Reserved		
X	0 1 1 1 1 1		0 1 0 0 0 1 1 0 0 0 0	**e_rlw**	Rotate Left Word	*-951*
X	0 1 1 1 1 1		0 1 0 0 0 1 1 0 0 0 1	**e_rlw.**	Rotate Left Word & record CR	*-951*
X	0 1 1 1 1 1		0 1 0 0 1 1 1 0 0 0 0	**e_rlwi**	Rotate Left Word Immediate	*-952*
X	0 1 1 1 1 1		0 1 0 0 1 1 1 0 0 0 1	**e_rlwi.**	Rotate Left Word Immediate & record CR	*-952*
RLWI	0 1 1 1 0 1	s s s s s a a a a a h h h h h	b b b b b e e e e e 0	**e_rlwimi**	Rotate Left Word Immed then Mask Insert	*-953*
RLWI	0 1 1 1 0 1	s s s s s a a a a a h h h h h	b b b b b e e e e e 1	**e_rlwinm**	Rotate Left Word Immed then AND with Mask	*-955*
X	0 1 1 1 1 1		0 0 0 0 0 1 1 0 0 0 0	**slw**	Shift Left Word	Book E
X	0 1 1 1 1 1		0 0 0 0 0 1 1 0 0 0 1	**slw.**	Shift Left Word & record CR	Book E
X	0 1 1 1 1 1		0 0 0 0 1 1 1 0 0 0 0	**e_slwi**	Shift Left Word Immediate	*-935*
X	0 1 1 1 1 1		0 0 0 0 1 1 1 0 0 0 1	**e_slwi.**	Shift Left Word Immediate & record CR	*-935*

Table 264. 32-bit instruction encodings (by mnemonic) (continued)

Format	Opcode			Mnemonic	Instruction	Page
	Primary ($Inst_{0:5}$)	Intermediate ($Inst_{6:20}$)	Extended ($Inst_{21:31}$)			
X	0 1 1 1 1 1		1 1 0 0 0 1 1 0 0 0 0	**sraw**	Shift Right Algebraic Word	Book E
X	0 1 1 1 1 1		1 1 0 0 0 1 1 0 0 0 1	**sraw.**	Shift Right Algebraic Word & record CR	Book E
X	0 1 1 1 1 1		1 1 0 0 1 1 1 0 0 0 0	**srawi**	Shift Right Algebraic Word Immediate	Book E
X	0 1 1 1 1 1		1 1 0 0 1 1 1 0 0 0 1	**srawi.**	Shift Right Algebraic Word Immediate & record CR	Book E
X	0 1 1 1 1 1		1 0 0 0 0 1 1 0 0 0 0	**srw**	Shift Right Word	Book E
X	0 1 1 1 1 1		1 0 0 0 0 1 1 0 0 0 1	**srw.**	Shift Right Word & record CR	Book E
X	0 1 1 1 1 1		1 0 0 0 1 1 1 0 0 0 0	**e_srwi**	Shift Right Word Immediate	*-957*
X	0 1 1 1 1 1		1 0 0 0 1 1 1 0 0 0 1	**e_srwi.**	Shift Right Word Immediate & record CR	*-957*
D	0 0 1 1 0 1	t t t t t a a a a a d d d d d	d d d d d d d d d d d	**e_stb**	Store Byte	*-958*
D8	0 0 0 1 1 0	t t t t t a a a a a 0 0 0 0 0	1 0 0 d d d d d d d	**e_stbu**	Store Byte with Update	*-958*
X	0 1 1 1 1 1		0 0 1 1 1 1 0 1 1 1 /	**stbux**	Store Byte with Update Indexed	Book E
X	0 1 1 1 1 1		0 0 1 1 0 1 0 1 1 1 /	**stbx**	Store Byte Indexed	Book E
D	0 1 0 1 1 1	t t t t t a a a a a d d d d d	d d d d d d d d d d d	**e_sth**	Store Halfword	*-959*
X	0 1 1 1 1 1		1 1 1 0 0 1 0 1 1 0 /	**sthbrx**	Store Halfword Byte-Reverse Indexed	Book E
D8	0 0 0 1 1 0	t t t t t a a a a a 0 0 0 0 0	1 0 1 d d d d d d d	**e_sthu**	Store Halfword with Update	*-959*
X	0 1 1 1 1 1		0 1 1 0 1 1 0 1 1 1 /	**sthux**	Store Halfword with Update Indexed	Book E
X	0 1 1 1 1 1		0 1 1 0 0 1 0 1 1 1 /	**sthx**	Store Halfword Indexed	Book E
D8	0 0 0 1 1 0	t t t t t a a a a a 0 0 0 0 1	0 0 1 d d d d d d d	**e_stmw**	Store Multiple Word	*-960*

Table 264. 32-bit instruction encodings (by mnemonic) (continued)

Format	Opcode			Mnemonic	Instruction	Page
	Primary ($Inst_{0:5}$)	Intermediate ($Inst_{6:20}$)	Extended ($Inst_{21:31}$)			
X	0 1 1 1 1 1		1 0 1 1 0 10101 /	**stswi**	Store String Word Immediate	Book E
X	0 1 1 1 1 1		1 0 1 0 0 10101 /	**stswx**	Store String Word Indexed	Book E
D	0 1 0 1 0 1	t t t t t aaaaa ddddd	d d d d d ddddd d	**e_stw**	Store Word	*-961*
X	0 1 1 1 1 1		1 0 1 0 0 10110 /	**stwbrx**	Store Word Byte-Reverse Indexed	Book E
X	0 1 1 1 1 1		0 0 1 0 0 10110 1	**stwcx.**	Store Word Conditional Indexed & record CR	Book E
D8	0 0 0 1 1 0	t t t t t aaaaa 00000	1 1 0 d d ddddd d	**e_stwu**	Store Word with Update	*-961*
X	0 1 1 1 1 1		0 0 1 0 1 10111 /	**stwux**	Store Word with Update Indexed	Book E
X	0 1 1 1 1 1		0 0 1 0 0 10111 /	**stwx**	Store Word Indexed	Book E
X	0 1 1 1 1 1		0 0 0 0 1 01000 0	**subf**	Subtract From	Book E
X	0 1 1 1 1 1		0 0 0 0 1 01000 1	**subf.**	Subtract From & record CR	Book E
X	0 1 1 1 1 1		0 0 0 0 0 01000 0	**subfc**	Subtract From Carrying	Book E
X	0 1 1 1 1 1		0 0 0 0 0 01000 1	**subfc.**	Subtract From Carrying & record CR	Book E
X	0 1 1 1 1 1		1 0 0 0 0 01000 0	**subfco**	Subtract From Carrying & record OV	Book E
X	0 1 1 1 1 1		1 0 0 0 0 01000 1	**subfco.**	Subtract From Carrying & record OV & CR	Book E
X	0 1 1 1 1 1		0 0 1 0 0 01000 0	**subfe**	Subtract From Extended with CA	Book E
X	0 1 1 1 1 1		0 0 1 0 0 01000 1	**subfe.**	Subtract From Extended with CA & record CR	Book E

Table 264. 32-bit instruction encodings (by mnemonic) (continued)

Format	Opcode			Mnemonic	Instruction	Page
	Primary ($Inst_{0:5}$)	Intermediate ($Inst_{6:20}$)	Extended ($Inst_{21:31}$)			
X	011111		10100 01000 0	**subfeo**	Subtract From Extended with CA & record OV	Book E
X	011111		10100 01000 1	**subfeo.**	Subtract From Extended with CA & record OV & CR	Book E
SCI8	000110	ttttt aaaaa 10110	FSSii iiiii i	**e_subfic**	Subtract from Immediate Carrying	*-964*
SCI8	000110	ttttt aaaaa 10111	FSSii iiiii i	**e_subfic.**	Subtract from Immediate and Record	*-964*
X	011111		00111 01000 0	**subfme**	Subtract From Minus One Extended with CA	Book E
X	011111		00111 01000 1	**subfme.**	Subtract From Minus One Extended with CA & record CR	Book E
X	011111		10111 01000 0	**subfmeo**	Subtract From Minus One Extended with CA & record OV	Book E
X	011111		10111 01000 1	**subfmeo.**	Subtract From Minus One Extended with CA & record OV & CR	Book E
X	011111		10001 01000 0	**subfo**	Subtract From & record OV	Book E
X	011111		10001 01000 1	**subfo.**	Subtract From & record OV & CR	Book E
X	011111		00110 01000 0	**subfze**	Subtract From Zero Extended with CA	Book E
X	011111		00110 01000 1	**subfze.**	Subtract From Zero Extended with CA & record CR	Book E

Table 264. 32-bit instruction encodings (by mnemonic) (continued)

Format	Opcode			Mnemonic	Instruction	Page
	Primary ($Inst_{0:5}$)	Intermediate ($Inst_{6:20}$)	Extended ($Inst_{21:31}$)			
X	0 1 1 1 1 1		1 0 1 1 0 0 1 0 0 0 0	**subfzeo**	Subtract From Zero Extended with CA & record OV	Book E
X	0 1 1 1 1 1		1 0 1 1 0 0 1 0 0 0 1	**subfzeo.**	Subtract From Zero Extended with CA & record OV & CR	Book E
X	0 1 1 1 1 1		1 1 0 0 0 1 0 0 1 0 /	**tlbivax**	TLB Invalidate Virtual Address Indexed	Book E
X	0 1 1 1 1 1		1 1 1 0 1 1 0 0 1 0 /	**tlbre**	TLB Read Entry	Book E
X	0 1 1 1 1 1		1 1 1 0 0 1 0 0 1 0 ?	**tlbsx**	TLB Search Indexed	Book E
X	0 1 1 1 1 1		1 0 0 0 1 1 0 1 1 0 /	**tlbsync**	TLB Synchronize	Book E
X	0 1 1 1 1 1		1 1 1 1 0 1 0 0 1 0 /	**tlbwe**	TLB Write Entry	Book E
X	0 1 1 1 1 1		0 0 0 0 0 0 0 1 0 0 /	**tw**	Trap Word	Book E
X	0 1 1 1 1 1		0 0 1 0 0 0 0 0 1 1 /	**wrtee**	Write External Enable	Book E
X	0 1 1 1 1 1		0 0 1 0 1 0 0 0 1 1 /	**wrteei**	Write External Enable Immediate	Book E
X	0 1 1 1 1 1		0 1 0 0 1 1 1 1 0 0 0	**xor**	XOR	Book E
X	0 1 1 1 1 1		0 1 0 0 1 1 1 1 0 0 1	**xor.**	XOR & record CR	Book E
SCI8	0 0 0 1 1 0	s s s s s a a a a a 1 1 1 0 0	F S S i i i i i i i i	**e_xori**	XOR Immediate	*-966*
SCI8	0 0 0 1 1 0	s s s s s a a a a a 1 1 1 0 1	F S S i i i i i i i i	**e_xori.**	XOR Immediate and Record	*-966*

14.3 Instruction index sorted by opcode

Table 265 lists all the 16-bit Power*Embedded instructions, sorted by opcode.

Table 265. Instruction index sorted by opcode

Format	16-Bit Opcodes ($Inst_{0:15}$)	Mnemonic	Instruction	Page
C	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1	**se_isync**	Instruction Synchronize	*-929*
C	0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0	**se_sc**	System Call	*-954*
C	0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0	**se_blr**	Branch to Link Register	*-908*
C	0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1	**se_blrl**	Branch to Link Register & Link	*-908*
C	0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0	**se_bctr**	Branch to Count Register	*-906*
C	0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1	**se_bctrl**	Branch to Count Register & Link	*-906*
C	0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0	**se_rfi**	Return From Interrupt	*-950*
C	0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1	**se_rfci**	Return From Critical Interrupt	*-949*
C	0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0	**se_rfdi**	Return From Debug Interrupt	*-859*
C	0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 1	unimp		
C	0 0 0 0 0 0 0 0 0 0 0 0 1 1 - -	unimp		
C	0 0 0 0 0 0 0 0 0 0 0 1 - - - -	unimp		
R	0 0 0 0 0 0 0 0 0 0 1 0 x x x x	**se_not**	NOT	*-947*
R	0 0 0 0 0 0 0 0 0 0 1 1 x x x x	unimp		
R	0 0 0 0 0 0 0 0 0 1 0 0 x x x x	**se_lmw**	Load Multiple Word	*-934*
R	0 0 0 0 0 0 0 0 0 1 0 1 x x x x	**se_stmw**	Store Multiple Word	*-960*
R	0 0 0 0 0 0 0 0 0 1 1 - x x x x	unimp		
R	0 0 0 0 0 0 0 0 1 0 0 0 x x x x	**se_mflr**	Move From Link Register	*-939*
R	0 0 0 0 0 0 0 0 1 0 0 1 x x x x	**se_mtlr**	Move To Link Register	*-943*
R	0 0 0 0 0 0 0 0 1 0 1 0 x x x x	**se_mfctr**	Move From Count Register	*-938*
R	0 0 0 0 0 0 0 0 1 0 1 1 x x x x	**se_mtctr**	Move To Count Register	*-942*
R	0 0 0 0 0 0 0 0 1 1 0 0 x x x x	**se_extzb**	Extend with Zeros Byte	*-927*
R	0 0 0 0 0 0 0 0 1 1 0 1 x x x x	**se_extsb**	Extend Sign Byte	*-926*
R	0 0 0 0 0 0 0 0 1 1 1 0 x x x x	**se_extzh**	Extend with Zeros Halfword	*-927*
R	0 0 0 0 0 0 0 0 1 1 1 1 x x x x	**se_extsh**	Extend Sign Halfword	*-926*
R	0 0 0 0 0 0 0 1 - - - - x x x x	unimp		

Table 265. Instruction index sorted by opcode (continued)

Format	16-Bit Opcodes ($Inst_{0:15}$)	Mnemonic	Instruction	Page
RR	0 0 0 0 0 0 1 0 y y y y x x x x	**se_mtar**	Move to Alternate Register	-941
RR	0 0 0 0 0 0 1 1 y y y y x x x x	**se_mfar**	Move from Alternate Register	-937
RR	0 0 0 0 0 1 0 0 y y y y x x x x	**se_add**	Add	-897
RR	0 0 0 0 0 1 0 1 y y y y x x x x	unimp		
RR	0 0 0 0 0 1 1 0 y y y y x x x x	**se_sub**	Subtract	-962
RR	0 0 0 0 0 1 1 1 y y y y x x x x	**se_sub.**	Subtract and Record	-962
RR	0 0 0 0 1 0 0 0 y y y y x x x x	unimp		
RR	0 0 0 0 1 0 0 1 y y y y x x x x	unimp		
RR	0 0 0 0 1 0 1 0 y y y y x x x x	**se_mullw**	Multiply Low Word	-945
RR	0 0 0 0 1 0 1 1 y y y y x x x x	**se_mr**	Move Register	-940
RR	0 0 0 0 1 1 0 0 y y y y x x x x	**se_cmp**	Compare	-912
RR	0 0 0 0 1 1 0 1 y y y y x x x x	**se_cmpl**	Compare Logical	-918
RR	0 0 0 0 1 1 1 0 y y y y x x x x	**se_xor**	XOR	-966
RR	0 0 0 0 1 1 1 1 y y y y x x x x	**se_or**	OR	-948
IM5	0 0 1 0 0 0 0 i i i i i x x x x	**se_addi**	Add Immediate	-897
IM5	0 0 1 0 0 0 1 i i i i i x x x x	**se_cmpli**	Compare Logical Immediate	-918
IM5	0 0 1 0 0 1 0 i i i i i x x x x	**se_subi**	Subtract Immediate	-965
IM5	0 0 1 0 0 1 1 i i i i i x x x x	**se_subi.**	Subtract Immediate and Record	-965
IM5	0 0 1 0 1 0 0 i i i i i x x x x	**se_subfic**	Subtract From Immediate Carrying	-964
IM5	0 0 1 0 1 0 1 i i i i i x x x x	**se_cmpi**	Compare Immediate	-912
IM5	0 0 1 0 1 1 0 i i i i i x x x x	**se_bmaski**	Bit Mask Generate Immediate	-909
IM5	0 0 1 0 1 1 1 i i i i i x x x x	**se_andi**	And Immediate	-901
RR	0 1 0 0 0 0 0 0 y y y y x x x x	**se_sraw**	Shift Right Algebraic Word	-956
RR	0 1 0 0 0 0 0 1 y y y y x x x x	**se_rlw**	Rotate Left Word	-951
RR	0 1 0 0 0 0 1 0 y y y y x x x x	**se_srw**	Shift Right Word	-957
RR	0 1 0 0 0 0 1 1 y y y y x x x x	**se_slw**	Shift Left Word	-955
RR	0 1 0 0 0 1 0 0 y y y y x x x x	**se_subf**	Subtract From	-963
RR	0 1 0 0 0 1 0 1 y y y y x x x x	**se_andc**	AND with Complement	-901
RR	0 1 0 0 0 1 1 0 y y y y x x x x	**se_and**	AND	-901
RR	0 1 0 0 0 1 1 1 y y y y x x x x	**se_and.**	AND and Record	-901

Table 265. Instruction index sorted by opcode (continued)

Format	16-Bit Opcodes ($Inst_{0:15}$)																Mnemonic	Instruction	Page
IM7	0	1	0	0	1	i	i	i	i	i	i	i	x	x	x	x	**se_li**	Load Immediate	*-933*
IM5	0	1	1	0	0	0	0	i	i	i	i	i	x	x	x	x	**se_bclri**	Bit Clear Immediate	*-905*
IM5	0	1	1	0	0	0	1	i	i	i	i	i	x	x	x	x	**se_bgeni**	Bit Generate Immediate	*-907*
IM5	0	1	1	0	0	1	0	i	i	i	i	i	x	x	x	x	**se_bseti**	Bit Set Immediate	*-910*
IM5	0	1	1	0	0	1	1	i	i	i	i	i	x	x	x	x	**se_btsti**	Bit Test Immediate	*-911*
IM5	0	1	1	0	1	0	0	i	i	i	i	i	x	x	x	x	**se_rlwi**	Rotate Left Word Immediate	*-951*
IM5	0	1	1	0	1	0	1	i	i	i	i	i	x	x	x	x	**se_srawi**	Shift Right Algebraic Word Immediate	*-956*
IM5	0	1	1	0	1	1	0	i	i	i	i	i	x	x	x	x	**se_slwi**	Shift Left Word Immediate	*-955*
IM5	0	1	1	0	1	1	1	i	i	i	i	i	x	x	x	x	**se_srwi**	Shift Right Word Immediate	*-957*
SD4	1	0	0	0	i	i	i	i	z	z	z	z	x	x	x	x	**se_lbz**	Load Byte and Zero	*-930*
SD4	1	0	0	1	i	i	i	i	z	z	z	z	x	x	x	x	**se_stb**	Store Byte	*-958*
SD4	1	0	1	0	i	i	i	i	z	z	z	z	x	x	x	x	**se_lhz**	Load Halfword and Zero	*-932*
SD4	1	0	1	1	i	i	i	i	z	z	z	z	x	x	x	x	**se_sth**	Store Halfword	*-959*
SD4	1	1	0	0	i	i	i	i	z	z	z	z	x	x	x	x	**se_lwz**	Load Word and Zero	*-935*
SD4	1	1	0	1	i	i	i	i	z	z	z	z	x	x	x	x	**se_stw**	Store Word	*-961*
UNIMP	1	1	1	0	-	-	-	-	-	-	-	-	-	-	-	-	unimp		
B8	1	1	1	1	0	o	i	i	d	d	d	d	d	d	d	d	**se_bc**	Branch Conditional	*-904*
B8	1	1	1	1	1	0	0	0	d	d	d	d	d	d	d	d	**se_b**	Branch	*-903*
B8	1	1	1	1	1	0	0	1	d	d	d	d	d	d	d	d	**se_bl**	Branch and Link	*-903*
UNIMP	1	1	1	1	1	0	1	-	-	-	-	-	-	-	-	-	unimp		
UNIMP	1	1	1	1	1	1	-	-	-	-	-	-	-	-	-	-	unimp		

Table 266 outlines the 32-bit instruction encodings.

Table 266. 32-bit instruction encodings

Format	Opcode			Mnemonic	Instruction	Page
	Primary ($Inst_{0:5}$)	Intermediate ($Inst_{6:20}$)	Extended ($Inst_{21:31}$)			
APU	0 - 0 1 - -	————————	- - - - - - - - - - -	**apu**	Reserved for APUs	
D14	0 0 1 1 0 0	t t t t t a a a a a	0 0 d d d d d d d d d d	**e_lbz**	Load Byte & Zero	*-930*
D14	0 0 1 1 0 0	t t t t t a a a a a	0 1 d d d d d d d d d d	**e_lhz**	Load Halfword & Zero	*-932*
D14	0 0 1 1 0 0	t t t t t a a a a a	1 0 d d d d d d d d d d	**e_lwz**	Load Word & Zero	*-936*
D14	0 0 1 1 0 0	t t t t t a a a a a	1 1 d d d d d d d d d d	**e_ld**	Load Doubleword & Zero (reserved for 64b GPR)	
D14	0 0 1 1 0 1	t t t t t a a a a a	0 0 d d d d d d d d d d	**e_stb**	Store Byte	*-958*
D14	0 0 1 1 0 1	t t t t t a a a a a	0 1 d d d d d d d d d d	**e_sth**	Store Halfword	*-959*
D14	0 0 1 1 0 1	t t t t t a a a a a	1 0 d d d d d d d d d d	**e_stw**	Store Word	*-961*
D14	0 0 1 1 0 1	t t t t t a a a a a	1 1 d d d d d d d d d d	**e_std**	Store Doubleword (reserved for 64b GPR)	
D8	0 0 1 1 1 0	t t t t t a a a a a	0 0 0 0 0 0 0 0 d d d d d	**e_lbzu**	Load Byte & Zero with Update	*-930*
D8	0 0 1 1 1 0	t t t t t a a a a a	0 0 0 0 0 0 0 1 d d d d d	**e_lhzu**	Load Halfword & Zero with Update	*-932*
D8	0 0 1 1 1 0	t t t t t a a a a a	0 0 0 0 0 0 1 0 d d d d d	**e_lwzu**	Load Word & Zero with Update	*-935*
D8	0 0 1 1 1 0	t t t t t a a a a a	0 0 0 0 0 0 1 1 d d d d d	**e_ldu**	Load Doubleword with Update (reserved for 64b GPR)	
D8	0 0 1 1 1 0	t t t t t a a a a a	0 0 0 0 0 1 0 0 d d d d d	**e_stbu**	Store Byte with Update	*-958*
D8	0 0 1 1 1 0	t t t t t a a a a a	0 0 0 0 0 1 0 1 d d d d d	**e_sthu**	Store Halfword with Update	*-959*
D8	0 0 1 1 1 0	t t t t t a a a a a	0 0 0 0 0 1 1 0 d d d d d	**e_stwu**	Store Word with Update	*-961*
D8	0 0 1 1 1 0	t t t t t a a a a a	0 0 0 0 0 1 1 1 d d d d d	**e_stdu**	Store Doubleword with Update (reserved for 64b GPR)	
D8	0 0 1 1 1 0	t t t t t a a a a a	0 0 0 0 1 0 0 0 d d d d d	**e_lmw**	Load Multiple Word	*-934*
D8	0 0 1 1 1 0	t t t t t a a a a a	0 0 0 0 1 0 0 1 d d d d d	**e_stmw**	Store Multiple Word	*-960*
D8	0 0 1 1 1 0	t t t t t a a a a a	0 0 0 0 1 0 1 0 d d d d d	**e_lha**	Load Halfword Algebraic	*-931*
D8	0 0 1 1 1 0	t t t t t a a a a a	0 0 0 0 1 0 1 1 d d d d d	**e_lhau**	Load Halfword Algebraic with Update	*-931*

Table 266. 32-bit instruction encodings (continued)

Format	Opcode			Mnemonic	Instruction	Page
	Primary ($Inst_{0:5}$)	Intermediate ($Inst_{6:20}$)	Extended ($Inst_{21:31}$)			
UNIMP	001110	ttttt aaaaa 00001	1--dd ddddd d	unimp		
UNIMP	001110	ttttt aaaaa 0001—	----- ----- -	unimp		
UNIMP	001110	ttttt aaaaa 001——	----- ----- -	unimp		
SCI8	001110	00000 aaaaa 010bF	Fssii iiiii i	**e_cmpi**	Compare Immediate	*-912*
SCI8	001110	00000 aaaaa 011bF	Fssii iiiii i	**e_cmpli**	Compare Logical Immediate	*-918*
SCI8	001110	ttttt aaaaa 10000	Fssii iiiii i	**e_addi**	Add Immediate	*-897*
SCI8	001110	ttttt aaaaa 10001	Fssii iiiii i	**e_addic**	Add Immediate Carrying	*-900*
SCI8	001110	ttttt aaaaa 10010	Fssii iiiii i	**e_andi**	AND Immediate	*-901*
SCI8	001110	ttttt aaaaa 10011	Fssii iiiii i	**e_ori**	OR Immediate	*-951*
SCI8	001110	ttttt aaaaa 10100	Fssii iiiii i	**e_subfic**	Subtract from Immediate Carrying	*-964*
SCI8	001110	ttttt aaaaa 10101	Fssii iiiii i	unimp		
SCI8	001110	ttttt aaaaa 10110	Fssii iiiii i	**e_mulli**	Multiply Low Immediate	*-944*
SCI8	001110	ttttt aaaaa 10111	Fssii iiiii i	**e_xori**	XOR Immediate	*-966*
SCI8	001110	ttttt aaaaa 11000	Fssii iiiii i	**e_addi.**	Add Immediate and Record	*-897*
SCI8	001110	ttttt aaaaa 11001	Fssii iiiii i	**e_addic.**	Add Immediate Carrying and Record	*-900*
SCI8	001110	ttttt aaaaa 11010	Fssii iiiii i	**e_andi.**	AND Immediate and Record	*-901*
SCI8	001110	ttttt aaaaa 11011	Fssii iiiii i	**e_ori.**	OR Immediate and Record	*-951*
SCI8	001110	ttttt aaaaa 11100	Fssii iiiii i	**e_subfic.**	Subtract from Immediate and Record	*-964*
SCI8	001110	ttttt aaaaa 10101	Fssii iiiii i	unimp		
SCI8	001110	ttttt aaaaa 11110	Fssii iiiii i	**e_mulli.**	Multiply Low Immediate and Record	*-944*
SCI8	001110	ttttt aaaaa 11111	Fssii iiiii i	**e_xori.**	XOR Immediate and Record	*-966*
D	001111	ttttt aaaaa iiiii	iiiii iiiii i	**e_add16i**	Add Immediate	*-898*
LI20	011100	ttttt 0iiii iiiii	iiiii iiiii i	**e_li**	Load Immediate	*-933*
LI20	011100	ttttt 1iiii iiiii	iiiii iiiii i	**e_lis**	Load Immediate Shifted	*-933*

Table 266. 32-bit instruction encodings (continued)

Format	Opcode			Mnemonic	Instruction	Page
	Primary ($Inst_{0:5}$)	Intermediate ($Inst_{6:20}$)	Extended ($Inst_{21:31}$)			
RLWI	011101	sssss aaaaa hhhhh	bbbbb eeeee 0	**e_rlwimi.**	Rotate Left Word Immed then Mask Insert & record CR	-952
RLWI	011101	sssss aaaaa hhhhh	bbbbb eeeee 1	**e_rlwinm.**	Rotate Left Word Immed then AND with Mask & record CR	-953
BD24	011110	0dddd ddddd ddddd	ddddd ddddd 0	**e_b**	Branch	-903
BD24	011110	0dddd ddddd ddddd	ddddd ddddd 1	**e_bl**	Branch & Link	-903
BD15	011110	1000o oiiii ddddd	ddddd ddddd 0	**e_bc**	Branch Conditional	-904
BD15	011110	1000o oiiii ddddd	ddddd ddddd 1	**e_bcl**	Branch Conditional & Link	-904
X	011111	----- ----- -----	----- 01111 /	**isel**	Integer Select	Book E
X	011111	----- ----- -----	/ 0000 01011 0	**mulhwu**	Multiply High Word Unsigned	Book E
X	011111	----- ----- -----	/ 0000 01011 1	**mulhwu.**	Multiply High Word Unsigned & Record	Book E
X	011111	----- ----- -----	/ 0010 01011 0	**mulhw**	Multiply High Word	Book E
X	011111	----- ----- -----	/ 0010 01011 1	**mulhw.**	Multiply High Word & record CR	Book E
X	011111	----- ----- -----	00000 00000 /	**cmp**	Compare	Book E
X	011111	----- ----- -----	00000 00100 /	**tw**	Trap Word	Book E
X	011111	----- ----- -----	00000 01000 0	**subfc**	Subtract From Carrying	Book E
X	011111	----- ----- -----	00000 01000 1	**subfc.**	Subtract From Carrying & record CR	Book E
X	011111	----- ----- -----	00000 01010 0	**addc**	Add Carrying	Book E
X	011111	----- ----- -----	00000 01010 1	**addc.**	Add Carrying & record CR	Book E
X	011111	----- ----- -----	00000 10011 /	**mfcr**	Move From Condition Register	Book E
X	011111	----- ----- -----	00000 10100 /	**lwarx**	Load Word & Reserve Indexed	Book E
X	011111	----- ----- -----	00000 10110 /	**icbt**	Instruction Cache Block Touch Indexed	Book E

Table 266. 32-bit instruction encodings (continued)

Format	Opcode			Mnemonic	Instruction	Page
	Primary ($Inst_{0:5}$)	Intermediate ($Inst_{6:20}$)	Extended ($Inst_{21:31}$)			
X	011111	----- ----- -----	00000 10111 /	**lwzx**	Load Word & Zero Indexed	Book E
X	011111	----- ----- -----	00000 11000 0	**slw**	Shift Left Word	Book E
X	011111	----- ----- -----	00000 11000 1	**slw.**	Shift Left Word & record CR	Book E
X	011111	----- ----- -----	00000 11010 0	**cntlzw**	Count Leading Zeros Word	Book E
X	011111	----- ----- -----	00000 11010 1	**cntlzw.**	Count Leading Zeros Word & record CR	Book E
X	011111	----- ----- -----	00000 11100 0	**and**	AND	Book E
X	011111	----- ----- -----	00000 11100 1	**and.**	AND & record CR	Book E
X	011111	----- ----- -----	00001 00000 /	**cmpl**	Compare Logical	Book E
X	011111	----- ----- -----	00001 01000 0	**subf**	Subtract From	Book E
X	011111	----- ----- -----	00001 01000 1	**subf.**	Subtract From & record CR	Book E
X	011111	----- ----- -----	00001 10110 /	**dcbst**	Data Cache Block Store Indexed	Book E
X	011111	----- ----- -----	00001 10111 /	**lwzux**	Load Word & Zero with Update Indexed	Book E
X	011111	----- ----- -----	00001 11100 0	**andc**	AND with Complement	Book E
X	011111	----- ----- -----	00001 11100 1	**andc.**	AND with Complement & record CR	Book E
X	011111	----- ----- -----	00010 10011 /	**mfmsr**	Move From Machine State Register	Book E
X	011111	----- ----- -----	00010 10110 /	**dcbf**	Data Cache Block Flush Indexed	Book E
X	011111	----- ----- -----	00010 10111 /	**lbzx**	Load Byte & Zero Indexed	Book E
X	011111	----- ----- -----	00011 01000 0	**neg**	Negate	Book E
X	011111	----- ----- -----	00011 01000 1	**neg.**	Negate & record CR	Book E

Table 266. 32-bit instruction encodings (continued)

Format	Opcode			Mnemonic	Instruction	Page
	Primary ($Inst_{0:5}$)	Intermediate ($Inst_{6:20}$)	Extended ($Inst_{21:31}$)			
X	011111	----- ----- -----	00011 10111 /	**lbzux**	Load Byte & Zero with Update Indexed	Book E
X	011111	----- ----- -----	00011 11100 0	**nor**	NOR	Book E
X	011111	----- ----- -----	00011 11100 1	**nor.**	NOR & record CR	Book E
X	011111	----- ----- -----	00100 00011 /	**wrtee**	Write External Enable	Book E
X	011111	----- ----- -----	00100 01000 0	**subfe**	Subtract From Extended with CA	Book E
X	011111	----- ----- -----	00100 01000 1	**subfe.**	Subtract From Extended with CA & record CR	Book E
X	011111	----- ----- -----	00100 01010 0	**adde**	Add Extended with CA	Book E
X	011111	----- ----- -----	00100 01010 1	**adde.**	Add Extended with CA & record CR	Book E
XFX	011111	----- ----- -----	00100 10000 /	**mtcrf**	Move To Condition Register Fields	Book E
X	011111	----- ----- -----	00100 10010 /	**mtmsr**	Move To Machine State Register	Book E
X	011111	----- ----- -----	00100 10110 1	**stwcx.**	Store Word Conditional Indexed & record CR	Book E
X	011111	----- ----- -----	00100 10111 /	**stwx**	Store Word Indexed	Book E
X	011111	----- ----- -----	00101 00011 /	**wrteei**	Write External Enable Immediate	Book E
X	011111	----- ----- -----	00101 10111 /	**stwux**	Store Word with Update Indexed	Book E
X	011111	----- ----- -----	00110 01000 0	**subfze**	Subtract From Zero Extended with CA	Book E
X	011111	----- ----- -----	00110 01000 1	**subfze.**	Subtract From Zero Extended with CA & record CR	Book E
X	011111	----- ----- -----	00110 01010 0	**addze**	Add to Zero Extended with CA	Book E
X	011111	----- ----- -----	00110 01010 1	**addze.**	Add to Zero Extended with CA & record CR	Book E

Table 266. 32-bit instruction encodings (continued)

Format	Opcode			Mnemonic	Instruction	Page
	Primary ($Inst_{0:5}$)	Intermediate ($Inst_{6:20}$)	Extended ($Inst_{21:31}$)			
X	011111	----- ----- -----	00110 10111 /	**stbx**	Store Byte Indexed	Book E
X	011111	----- ----- -----	00111 01000 0	**subfme**	Subtract From Minus One Extended with CA	Book E
X	011111	----- ----- -----	00111 01000 1	**subfme.**	Subtract From Minus One Extended with CA & record CR	Book E
X	011111	----- ----- -----	00111 01010 0	**addme**	Add to Minus One Extended with CA	Book E
X	011111	----- ----- -----	00111 01010 1	**addme.**	Add to Minus One Extended with CA & record CR	Book E
X	011111	----- ----- -----	00111 01011 0	**mullw**	Multiply Low Word	Book E
X	011111	----- ----- -----	00111 01011 1	**mullw.**	Multiply Low Word & record CR	Book E
X	011111	----- ----- -----	00111 10110 /	**dcbtst**	Data Cache Block Touch for Store Indexed	Book E
X	011111	----- ----- -----	00111 10111 /	**stbux**	Store Byte with Update Indexed	Book E
X	011111	----- ----- -----	01000 01010 0	**add**	Add	Book E
X	011111	----- ----- -----	01000 01010 1	**add.**	Add & record CR	Book E
X	011111	----- ----- -----	01000 10011 /	**mfapidi**	Move From APID Indirect	Book E
X	011111	----- ----- -----	01000 10110 /	**dcbt**	Data Cache Block Touch Indexed	Book E
X	011111	----- ----- -----	01000 10111 /	**lhzx**	Load Halfword & Zero Indexed	Book E
X	011111	----- ----- -----	01000 11100 0	**eqv**	Equivalent	Book E
X	011111	----- ----- -----	01000 11100 1	**eqv.**	Equivalent & record CR	Book E
X	011111	----- ----- -----	01001 10111 /	**lhzux**	Load Halfword & Zero with Update Indexed	Book E
X	011111	----- ----- -----	01001 11100 0	**xor**	XOR	Book E

Table 266. 32-bit instruction encodings (continued)

Format	Opcode			Mnemonic	Instruction	Page
	Primary ($Inst_{0:5}$)	Intermediate ($Inst_{6:20}$)	Extended ($Inst_{21:31}$)			
X	011111	----- ----- -----	01001 11100 1	**xor.**	XOR & record CR	Book E
XFX	011111	----- ----- -----	01010 00011 /	**mfdcr**	Move From Device Control Register	Book E
XFX	011111	----- ----- -----	01010 10011 /	**mfspr**	Move From Special Purpose Register	Book E
X	011111	----- ----- -----	01010 10111 /	**lhax**	Load Halfword Algebraic Indexed	Book E
X	011111	----- ----- -----	01011 10111 /	**lhaux**	Load Halfword Algebraic with Update Indexed	Book E
X	011111	----- ----- -----	01100 10111 /	**sthx**	Store Halfword Indexed	Book E
X	011111	----- ----- -----	01100 11100 0	**orc**	OR with Complement	Book E
X	011111	----- ----- -----	01100 11100 1	**orc.**	OR with Complement & record CR	Book E
X	011111	----- ----- -----	01101 10111 /	**sthux**	Store Halfword with Update Indexed	Book E
X	011111	----- ----- -----	01101 11100 0	**or**	OR	Book E
X	011111	----- ----- -----	01101 11100 1	**or.**	OR & record CR	Book E
XFX	011111	----- ----- -----	01110 00011 /	**mtdcr**	Move To Device Control Register	Book E
X	011111	----- ----- -----	01110 01011 0	**divwu**	Divide Word Unsigned	Book E
X	011111	----- ----- -----	01110 01011 1	**divwu.**	Divide Word Unsigned & record CR	Book E
XFX	011111	----- ----- -----	01110 10011 /	**mtspr**	Move To Special Purpose Register	Book E
X	011111	----- ----- -----	01110 10110 /	**dcbi**	Data Cache Block Invalidate Indexed	Book E
X	011111	----- ----- -----	01110 11100 0	**nand**	NAND	Book E
X	011111	----- ----- -----	01110 11100 1	**nand.**	NAND & record CR	Book E
X	011111	----- ----- -----	01111 01011 0	**divw**	Divide Word	Book E

Table 266. 32-bit instruction encodings (continued)

Format	Opcode			Mnemonic	Instruction	Page
	Primary ($Inst_{0:5}$)	Intermediate ($Inst_{6:20}$)	Extended ($Inst_{21:31}$)			
X	011111	----- ----- -----	01111 01011 1	**divw.**	Divide Word & record CR	Book E
X	011111	----- ----- -----	10000 00000 /	**mcrxr**	Move to Condition Register from XER	Book E
X	011111	----- ----- -----	10000 01000 0	**subfco**	Subtract From Carrying & record OV	Book E
X	011111	----- ----- -----	10000 01000 1	**subfco.**	Subtract From Carrying & record OV & CR	Book E
X	011111	----- ----- -----	10000 01010 0	**addco**	Add Carrying & record OV	Book E
X	011111	----- ----- -----	10000 01010 1	**addco.**	Add Carrying & record OV & CR	Book E
X	011111	----- ----- -----	10000 10110 /	**lwbrx**	Load Word Byte-Reverse Indexed	Book E
X	011111	----- ----- -----	10000 11000 0	**srw**	Shift Right Word	Book E
X	011111	----- ----- -----	10000 11000 1	**srw.**	Shift Right Word & record CR	Book E
X	011111	----- ----- -----	10001 01000 0	**subfo**	Subtract From & record OV	Book E
X	011111	----- ----- -----	10001 01000 1	**subfo.**	Subtract From & record OV & CR	Book E
X	011111	----- ----- -----	10001 10110 /	**tlbsync**	TLB Synchronize	Book E
X	011111	----- ----- -----	10010 10110 /	**msync**	Memory Synchronize	Book E
X	011111	----- ----- -----	10011 01000 0	**nego**	Negate & record OV	Book E
X	011111	----- ----- -----	10011 01000 1	**nego.**	Negate & record OV & record CR	Book E
X	011111	----- ----- -----	10100 01000 0	**subfeo**	Subtract From Extended with CA & record OV	Book E
X	011111	----- ----- -----	10100 01000 1	**subfeo.**	Subtract From Extended with CA & record OV & CR	Book E
X	011111	----- ----- -----	10100 01010 0	**addeo**	Add Extended with CA & record OV	Book E

Table 266. 32-bit instruction encodings (continued)

Format	Opcode			Mnemonic	Instruction	Page
	Primary ($Inst_{0:5}$)	Intermediate ($Inst_{6:20}$)	Extended ($Inst_{21:31}$)			
X	011111	----- ----- -----	10100 01010 1	**addeo.**	Add Extended with CA & record OV & CR	Book E
X	011111	----- ----- -----	10100 10101 /	**stswx**	Store String Word Indexed	Book E
X	011111	----- ----- -----	10100 10110 /	**stwbrx**	Store Word Byte-Reverse Indexed	Book E
X	011111	----- ----- -----	10110 01000 0	**subfzeo**	Subtract From Zero Extended with CA & record OV	Book E
X	011111	----- ----- -----	10110 01000 1	**subfzeo.**	Subtract From Zero Extended with CA & record OV & CR	Book E
X	011111	----- ----- -----	10110 01010 0	**addzeo**	Add to Zero Extended with CA & record OV	Book E
X	011111	----- ----- -----	10110 01010 1	**addzeo.**	Add to Zero Extended with CA & record OV & CR	Book E
X	011111	----- ----- -----	10110 10101 /	**stswi**	Store String Word Immediate	Book E
X	011111	----- ----- -----	10111 01000 0	**subfmeo**	Subtract From Minus One Extended with CA & record OV	Book E
X	011111	----- ----- -----	10111 01000 1	**subfmeo.**	Subtract From Minus One Extended with CA & record OV & CR	Book E
X	011111	----- ----- -----	10111 01010 0	**addmeo**	Add to Minus One Extended with CA & record OV	Book E
X	011111	----- ----- -----	10111 01010 1	**addmeo.**	Add to Minus One Extended with CA & record OV & CR	Book E
X	011111	----- ----- -----	10111 01011 0	**mullwo**	Multiply Low Word & record OV	Book E
X	011111	----- ----- -----	10111 01011 1	**mullwo.**	Multiply Low Word & record OV & CR	Book E
X	011111	----- ----- -----	10111 10110 /	**dcba**	Data Cache Block Allocate Indexed	Book E
X	011111	----- ----- -----	11000 01010 0	**addo**	Add & record OV	Book E
X	011111	----- ----- -----	11000 01010 1	**addo.**	Add & record OV & CR	Book E

Table 266. 32-bit instruction encodings (continued)

Format	Opcode			Mnemonic	Instruction	Page
	Primary ($Inst_{0:5}$)	Intermediate ($Inst_{6:20}$)	Extended ($Inst_{21:31}$)			
X	011111	----- ----- -----	11000 10010 /	**tlbivax**	TLB Invalidate Virtual Address Indexed	Book E
X	011111	----- ----- -----	11000 10110 /	**lhbrx**	Load Halfword Byte-Reverse Indexed	Book E
X	011111	----- ----- -----	11000 11000 0	**sraw**	Shift Right Algebraic Word	Book E
X	011111	----- ----- -----	11000 11000 1	**sraw.**	Shift Right Algebraic Word & record CR	Book E
X	011111	----- ----- -----	11001 11000 0	**srawi**	Shift Right Algebraic Word Immediate	Book E
X	011111	----- ----- -----	11001 11000 1	**srawi.**	Shift Right Algebraic Word Immediate & record CR	Book E
X	011111	----- ----- -----	11010 10110 /	**mbar**	Memory Barrier	Book E
X	011111	----- ----- -----	11100 10010 ?	**tlbsx**	TLB Search Indexed	Book E
X	011111	----- ----- -----	11100 10110 /	**sthbrx**	Store Halfword Byte-Reverse Indexed	Book E
X	011111	----- ----- -----	11100 11010 0	**extsh**	Extend Sign Halfword	Book E
X	011111	----- ----- -----	11100 11010 1	**extsh.**	Extend Sign Halfword & record CR	Book E
X	011111	----- ----- -----	11101 10010 /	**tlbre**	TLB Read Entry	Book E
X	011111	----- ----- -----	11101 11010 0	**extsb**	Extend Sign Byte	Book E
X	011111	----- ----- -----	11101 11010 1	**extsb.**	Extend Sign Byte & record CR	Book E
X	011111	----- ----- -----	11110 01011 0	**divwuo**	Divide Word Unsigned & record OV	Book E
X	011111	----- ----- -----	11110 01011 1	**divwuo.**	Divide Word Unsigned & record OV & CR	Book E
X	011111	----- ----- -----	11110 10010 /	**tlbwe**	TLB Write Entry	Book E
X	011111	----- ----- -----	11110 10110 /	**icbi**	Instruction Cache Block Invalidate Indexed	Book E
X	011111	----- ----- -----	11111 01011 0	**divwo**	Divide Word & record OV	Book E

Table 266. 32-bit instruction encodings (continued)

Format	Opcode			Mnemonic	Instruction	Page
	Primary ($Inst_{0:5}$)	Intermediate ($Inst_{6:20}$)	Extended ($Inst_{21:31}$)			
X	0 1 1 1 1 1	- - - - - - - - - - - - - - -	1 1 1 1 1 0 1 0 1 1 1	**divwo.**	Divide Word & record OV & CR	Book E
X	0 1 1 1 1 1	- - - - - - - - - - - - - - -	1 1 1 1 1 1 0 1 1 0 /	**dcbz**	Data Cache Block set to Zero Indexed	Book E

14.4 Instruction index sorted by mnemonic

Table 267 lists all the 16-bit Power*Embedded instructions, sorted by mnemonic.

Table 267. Instruction index sorted by mnemonic

Format	16-Bit Opcodes ($Inst_{0:15}$)																	Mnemonic	Instruction	Page
RR	0	0	0	0		0	1	0	0		y	y	y	y		x	x	x	x	**se_add**	Add	*-897*
IM5	0	0	1	0		0	0	0	i		i	i	i	i		x	x	x	x	**se_addi**	Add Immediate	*-898*
RR	0	1	0	0		0	1	1	0		y	y	y	y		x	x	x	x	**se_and**	AND	*-901*
RR	0	1	0	0		0	1	1	1		y	y	y	y		x	x	x	x	**se_and.**	AND and Record	*-901*
RR	0	1	0	0		0	1	0	1		y	y	y	y		x	x	x	x	**se_andc**	AND with Complement	*-901*
IM5	0	0	1	0		1	1	1	i		i	i	i	i		x	x	x	x	**se_andi**	And Immediate	*-901*
B8	1	1	1	1		0	o	i	i		d	d	d	d		d	d	d	d	**se_bc**	Branch Conditional	*-904*
IM5	0	1	1	0		0	0	0	i		i	i	i	i		x	x	x	x	**se_bclri**	Bit Clear Immediate	*-905*
C	0	0	0	0		0	0	0	0		0	0	0	0		0	1	1	0	**se_bctr**	Branch to Count Register	*-906*
C	0	0	0	0		0	0	0	0		0	0	0	0		0	1	1	1	**se_bctrl**	Branch to Count Register & Link	*-906*
IM5	0	1	1	0		0	0	1	i		i	i	i	i		x	x	x	x	**se_bgeni**	Bit Generate Immediate	*-907*
B8	1	1	1	1		1	0	0	1		d	d	d	d		d	d	d	d	**se_bl**	Branch and Link	*-903*
C	0	0	0	0		0	0	0	0		0	0	0	0		0	1	0	0	**se_blr**	Branch to Link Register	*-908*
C	0	0	0	0		0	0	0	0		0	0	0	0		0	1	0	1	**se_blrl**	Branch to Link Register & Link	*-908*
IM5	0	0	1	0		1	1	0	i		i	i	i	i		x	x	x	x	**se_bmaski**	Bit Mask Generate Immediate	*-909*
B8	1	1	1	1		1	0	0	0		d	d	d	d		d	d	d	d	**se_b**	Branch	*-903*
IM5	0	1	1	0		0	1	0	i		i	i	i	i		x	x	x	x	**se_bseti**	Bit Set Immediate	*-909*
IM5	0	1	1	0		0	1	1	i		i	i	i	i		x	x	x	x	**se_btsti**	Bit Test Immediate	*-911*
RR	0	0	0	0		1	1	0	0		y	y	y	y		x	x	x	x	**se_cmp**	Compare	*-912*
IM5	0	0	1	0		1	0	1	i		i	i	i	i		x	x	x	x	**se_cmpi**	Compare Immediate	*-912*
RR	0	0	0	0		1	1	0	1		y	y	y	y		x	x	x	x	**se_cmpl**	Compare Logical	*-918*
IM5	0	0	1	0		0	0	1	i		i	i	i	i		x	x	x	x	**se_cmpli**	Compare Logical Immediate	*-918*
R	0	0	0	0		0	0	0	0		1	1	0	1		x	x	x	x	**se_extsb**	Extend Sign Byte	*-926*
R	0	0	0	0		0	0	0	0		1	1	1	1		x	x	x	x	**se_extsh**	Extend Sign Halfword	*-926*
R	0	0	0	0		0	0	0	0		1	1	0	0		x	x	x	x	**se_extzb**	Extend with Zeros Byte	*-927*
R	0	0	0	0		0	0	0	0		1	1	1	0		x	x	x	x	**se_extzh**	Extend with Zeros Halfword	*-927*
C	0	0	0	0		0	0	0	0		0	0	0	0		0	0	0	1	**se_isync**	Instruction Synchronize	*-929*

Table 267. Instruction index sorted by mnemonic (continued)

Format	16-Bit Opcodes ($Inst_{0:15}$)																		Mnemonic	Instruction	Page
SD4	1	0	0	0		i	i	i	i		z	z	z	z		x	x	x	x	**se_lbz**	Load Byte and Zero	*-930*
R	0	0	0	0		0	0	0	0		0	1	0	0		x	x	x	x	**se_lmw**	Load Multiple Word	*-934*
SD4	1	0	1	0		i	i	i	i		z	z	z	z		x	x	x	x	**se_lhz**	Load Halfword and Zero	*-932*
IM7	0	1	0	0		1	i	i	i		i	i	i	i		x	x	x	x	**se_li**	Load Immediate	*-933*
SD4	1	1	0	0		i	i	i	i		z	z	z	z		x	x	x	x	**se_lwz**	Load Word and Zero	*-935*
RR	0	0	0	0		0	0	1	1		y	y	y	y		x	x	x	x	**se_mfar**	Move from Alternate Register	*-937*
R	0	0	0	0		0	0	0	0		1	0	1	0		x	x	x	x	**se_mfctr**	Move From Count Register	*-938*
R	0	0	0	0		0	0	0	0		1	0	0	0		x	x	x	x	**se_mflr**	Move From Link Register	*-939*
RR	0	0	0	0		1	0	1	1		y	y	y	y		x	x	x	x	**se_mr**	Move Register	*-940*
RR	0	0	0	0		0	0	1	0		y	y	y	y		x	x	x	x	**se_mtar**	Move to Alternate Register	*-941*
R	0	0	0	0		0	0	0	0		1	0	1	1		x	x	x	x	**se_mtctr**	Move To Count Register	*-942*
R	0	0	0	0		0	0	0	0		1	0	0	1		x	x	x	x	**se_mtlr**	Move To Link Register	*-943*
RR	0	0	0	0		1	0	1	0		y	y	y	y		x	x	x	x	**se_mullw**	Multiply Low Word	*-945*
R	0	0	0	0		0	0	0	0		0	0	1	0		x	x	x	x	**se_not**	NOT	*-947*
RR	0	0	0	0		1	1	1	1		y	y	y	y		x	x	x	x	**se_or**	OR	*-948*
C	0	0	0	0		0	0	0	0		0	0	0	0		1	0	0	1	**se_rfci**	Return From Critical Interrupt	*-949*
C	0	0	0	0		0	0	0	0		0	0	0	0		1	0	1	0	**se_rfdi**	Return From Debug Interrupt	*-859*
C	0	0	0	0		0	0	0	0		0	0	0	0		1	0	0	0	**se_rfi**	Return From Interrupt	*-950*
RR	0	1	0	0		0	0	0	1		y	y	y	y		x	x	x	x	**se_rlw**	Rotate Left Word	*-951*
IM5	0	1	1	0		1	0	0	i		i	i	i	i		x	x	x	x	**se_rlwi**	Rotate Left Word Immediate	*-951*
C	0	0	0	0		0	0	0	0		0	0	0	0		0	0	1	0	**se_sc**	System Call	*-954*
RR	0	1	0	0		0	0	1	1		y	y	y	y		x	x	x	x	**se_slw**	Shift Left Word	*-955*
IM5	0	1	1	0		1	1	0	i		i	i	i	i		x	x	x	x	**se_slwi**	Shift Left Word Immediate	*-955*
RR	0	1	0	0		0	0	0	0		y	y	y	y		x	x	x	x	**se_sraw**	Shift Right Algebraic Word	*-956*
IM5	0	1	1	0		1	0	1	i		i	i	i	i		x	x	x	x	**se_srawi**	Shift Right Algebraic Word Immediate	*-956*
RR	0	1	0	0		0	0	1	0		y	y	y	y		x	x	x	x	**se_srw**	Shift Right Word	*-957*
IM5	0	1	1	0		1	1	1	i		i	i	i	i		x	x	x	x	**se_srwi**	Shift Right Word Immediate	*-957*

Table 267. Instruction index sorted by mnemonic (continued)

Format	16-Bit Opcodes ($Inst_{0:15}$)																		Mnemonic	Instruction	Page
SD4	1	0	0	1		i	i	i	i		z	z	z	z		x	x	x	x	**se_stb**	Store Byte	*-958*
SD4	1	0	1	1		i	i	i	i		z	z	z	z		x	x	x	x	**se_sth**	Store Halfword	*-959*
R	0	0	0	0		0	0	0	0		0	1	0	1		x	x	x	x	**se_stmw**	Store Multiple Word	*-960*
SD4	1	1	0	1		i	i	i	i		z	z	z	z		x	x	x	x	**se_stw**	Store Word	*-961*
RR	0	0	0	0		0	1	1	0		y	y	y	y		x	x	x	x	**se_sub**	Subtract	*-961*
RR	0	0	0	0		0	1	1	1		y	y	y	y		x	x	x	x	**se_sub.**	Subtract and Record	*-961*
RR	0	1	0	0		0	1	0	0		y	y	y	y		x	x	x	x	**se_subf**	Subtract From	*-963*
IM5	0	0	1	0		1	0	0	i		i	i	i	i		x	x	x	x	**se_subfic**	Subtract From Immediate Carrying	*-964*
IM5	0	0	1	0		0	1	0	i		i	i	i	i		x	x	x	x	**se_subi**	Subtract Immediate	*-965*
IM5	0	0	1	0		0	1	1	i		i	i	i	i		x	x	x	x	**se_subi.**	Subtract Immediate and Record	*-965*
RR	0	0	0	0		1	1	1	0		y	y	y	y		x	x	x	x	**se_xor**	XOR	*-966*

Table 268 sorts 32-bit instructions by mnemonic, ignoring the e_ prefix.

Table 268. 32-bit instructions by mnemonic (ignoring the e_ prefix)

Format	Opcode			Mnemonic	Instruction	Page
	Primary ($Inst_{0:5}$)	Intermediate ($Inst_{6:20}$)	Extended ($Inst_{21:31}$)			
X	011111	----- ----- -----	01000 01010 0	**add**	Add	Book E
X	011111	----- ----- -----	01000 01010 1	**add.**	Add & record CR	Book E
X	011111	----- ----- -----	11000 01010 0	**addo**	Add & record OV	Book E
X	011111	----- ----- -----	11000 01010 1	**addo.**	Add & record OV & CR	Book E
X	011111	----- ----- -----	00000 01010 0	**addc**	Add Carrying	Book E
X	011111	----- ----- -----	00000 01010 1	**addc.**	Add Carrying & record CR	Book E
X	011111	----- ----- -----	10000 01010 0	**addco**	Add Carrying & record OV	Book E
X	011111	----- ----- -----	10000 01010 1	**addco.**	Add Carrying & record OV & CR	Book E
X	011111	----- ----- -----	00100 01010 0	**adde**	Add Extended with CA	Book E
X	011111	----- ----- -----	00100 01010 1	**adde.**	Add Extended with CA & record CR	Book E
X	011111	----- ----- -----	10100 01010 0	**addeo**	Add Extended with CA & record OV	Book E
X	011111	----- ----- -----	10100 01010 1	**addeo.**	Add Extended with CA & record OV & CR	Book E
D	001111	ttttt aaaaa iiiii	iiiii iiiii i	**e_add16i**	Add Immediate	*-898*
SCI8	001110	ttttt aaaaa 10000	FSSii iiiii i	**e_addi**	Add Immediate	*-898*
SCI8	001110	ttttt aaaaa 11000	FSSii iiiii i	**e_addi.**	Add Immediate and Record	*-898*
SCI8	001110	ttttt aaaaa 10001	FSSii iiiii i	**e_addic**	Add Immediate Carrying	*-900*
SCI8	001110	ttttt aaaaa 11001	FSSii iiiii i	**e_addic.**	Add Immediate Carrying and Record	*-900*
X	011111	----- ----- -----	00111 01010 0	**addme**	Add to Minus One Extended with CA	Book E

Table 268. 32-bit instructions by mnemonic (ignoring the e_ prefix) (continued)

Format	Opcode			Mnemonic	Instruction	Page
	Primary ($Inst_{0:5}$)	Intermediate ($Inst_{6:20}$)	Extended ($Inst_{21:31}$)			
X	011111	----- ----- -----	00111 01010 1	**addme.**	Add to Minus One Extended with CA & record CR	Book E
X	011111	----- ----- -----	10111 01010 0	**addmeo**	Add to Minus One Extended with CA & record OV	Book E
X	011111	----- ----- -----	10111 01010 1	**addmeo.**	Add to Minus One Extended with CA & record OV & CR	Book E
X	011111	----- ----- -----	00110 01010 0	**addze**	Add to Zero Extended with CA	Book E
X	011111	----- ----- -----	00110 01010 1	**addze.**	Add to Zero Extended with CA & record CR	Book E
X	011111	----- ----- -----	10110 01010 0	**addzeo**	Add to Zero Extended with CA & record OV	Book E
X	011111	----- ----- -----	10110 01010 1	**addzeo.**	Add to Zero Extended with CA & record OV & CR	Book E
X	011111	----- ----- -----	00000 11100 0	**and**	AND	Book E
X	011111	----- ----- -----	00000 11100 1	**and.**	AND & record CR	Book E
X	011111	----- ----- -----	00001 11100 0	**andc**	AND with Complement	Book E
X	011111	----- ----- -----	00001 11100 1	**andc.**	AND with Complement & record CR	Book E
SCI8	001110	ttttt aaaaa 10010	FSSii iiiii i	**e_andi**	AND Immediate	*-901*
SCI8	001110	ttttt aaaaa 11010	FSSii iiiii i	**e_andi.**	AND Immediate and Record	*-901*
BD24	011110	0dddd ddddd ddddd	ddddd ddddd 0	**e_b**	Branch	*-903*
BD15	011110	1000o oiiii ddddd	ddddd ddddd 0	**e_bc**	Branch Conditional	*-904*
BD15	011110	1000o oiiii ddddd	ddddd ddddd 1	**e_bcl**	Branch Conditional & Link	*-904*
BD24	011110	0dddd ddddd ddddd	ddddd ddddd 1	**e_bl**	Branch & Link	*-903*
X	011111	----- ----- -----	00000 00000 /	**cmp**	Compare	Book E

Table 268. 32-bit instructions by mnemonic (ignoring the e_ prefix) (continued)

Format	Opcode			Mnemonic	Instruction	Page
	Primary ($Inst_{0:5}$)	Intermediate ($Inst_{6:20}$)	Extended ($Inst_{21:31}$)			
SCI8	0 0 1 1 1 0	0 0 0 0 0 a a a a a 0 1 0 b F	F S S i i i i i i i i i	**e_cmpi**	Compare Immediate	*-912*
X	0 1 1 1 1 1	- - - - - - - - - - - - - - -	0 0 0 0 1 0 0 0 0 0 /	**cmpl**	Compare Logical	Book E
SCI8	0 0 1 1 1 0	0 0 0 0 0 a a a a a 0 1 1 b F	F S S i i i i i i i i i	**e_cmpli**	Compare Logical Immediate	*-918*
X	0 1 1 1 1 1	- - - - - - - - - - - - - - -	0 0 0 0 0 1 1 0 1 0 0	**cntlzw**	Count Leading Zeros Word	Book E
X	0 1 1 1 1 1	- - - - - - - - - - - - - - -	0 0 0 0 0 1 1 0 1 0 1	**cntlzw.**	Count Leading Zeros Word & record CR	Book E
X	0 1 1 1 1 1	- - - - - - - - - - - - - - -	1 0 1 1 1 1 0 1 1 0 /	**dcba**	Data Cache Block Allocate Indexed	Book E
X	0 1 1 1 1 1	- - - - - - - - - - - - - - -	0 0 0 1 0 1 0 1 1 0 /	**dcbf**	Data Cache Block Flush Indexed	Book E
X	0 1 1 1 1 1	- - - - - - - - - - - - - - -	0 1 1 1 0 1 0 1 1 0 /	**dcbi**	Data Cache Block Invalidate Indexed	Book E
X	0 1 1 1 1 1	- - - - - - - - - - - - - - -	0 0 0 0 1 1 0 1 1 0 /	**dcbst**	Data Cache Block Store Indexed	Book E
X	0 1 1 1 1 1	- - - - - - - - - - - - - - -	0 0 1 1 1 1 0 1 1 0 /	**dcbtst**	Data Cache Block Touch for Store Indexed	Book E
X	0 1 1 1 1 1	- - - - - - - - - - - - - - -	0 1 0 0 0 1 0 1 1 0 /	**dcbt**	Data Cache Block Touch Indexed	Book E
X	0 1 1 1 1 1	- - - - - - - - - - - - - - -	1 1 1 1 1 1 0 1 1 0 /	**dcbz**	Data Cache Block set to Zero Indexed	Book E
X	0 1 1 1 1 1	- - - - - - - - - - - - - - -	0 1 1 1 1 0 1 0 1 1 0	**divw**	Divide Word	Book E
X	0 1 1 1 1 1	- - - - - - - - - - - - - - -	0 1 1 1 1 0 1 0 1 1 1	**divw.**	Divide Word & record CR	Book E
X	0 1 1 1 1 1	- - - - - - - - - - - - - - -	1 1 1 1 1 0 1 0 1 1 0	**divwo**	Divide Word & record OV	Book E
X	0 1 1 1 1 1	- - - - - - - - - - - - - - -	1 1 1 1 1 0 1 0 1 1 1	**divwo.**	Divide Word & record OV & CR	Book E
X	0 1 1 1 1 1	- - - - - - - - - - - - - - -	0 1 1 1 0 0 1 0 1 1 0	**divwu**	Divide Word Unsigned	Book E
X	0 1 1 1 1 1	- - - - - - - - - - - - - - -	0 1 1 1 0 0 1 0 1 1 1	**divwu.**	Divide Word Unsigned & record CR	Book E

Table 268. 32-bit instructions by mnemonic (ignoring the e_ prefix) (continued)

Format	Opcode			Mnemonic	Instruction	Page
	Primary ($Inst_{0:5}$)	Intermediate ($Inst_{6:20}$)	Extended ($Inst_{21:31}$)			
X	011111	----- ----- -----	11110 01011 0	**divwuo**	Divide Word Unsigned & record OV	Book E
X	011111	----- ----- -----	11110 01011 1	**divwuo.**	Divide Word Unsigned & record OV & CR	Book E
X	011111	----- ----- -----	01000 11100 0	**eqv**	Equivalent	Book E
X	011111	----- ----- -----	01000 11100 1	**eqv.**	Equivalent & record CR	Book E
X	011111	----- ----- -----	11101 11010 0	**extsb**	Extend Sign Byte	Book E
X	011111	----- ----- -----	11101 11010 1	**extsb.**	Extend Sign Byte & record CR	Book E
X	011111	----- ----- -----	11100 11010 0	**extsh**	Extend Sign Halfword	Book E
X	011111	----- ----- -----	11100 11010 1	**extsh.**	Extend Sign Halfword & record CR	Book E
X	011111	----- ----- -----	11110 10110 /	**icbi**	Instruction Cache Block Invalidate Indexed	Book E
X	011111	----- ----- -----	00000 10110 /	**icbt**	Instruction Cache Block Touch Indexed	Book E
X	011111	----- ----- -----	----- 01111 /	**isel**	Integer Select	Book E
D14	001100	ttttt aaaaa 00ddd	ddddd ddddd d	**e_lbz**	Load Byte & Zero	*-930*
X	011111	----- ----- -----	00010 10111 /	**lbzx**	Load Byte & Zero Indexed	Book E
D8	001110	ttttt aaaaa 00000	000dd ddddd d	**e_lbzu**	Load Byte & Zero with Update	*-930*
X	011111	----- ----- -----	00011 10111 /	**lbzux**	Load Byte & Zero with Update Indexed	Book E
D14	001100	ttttt aaaaa 11ddd	ddddd ddddd d	**e_ld**	Load Doubleword & Zero (reserved for 64b GPR)	

Table 268. 32-bit instructions by mnemonic (ignoring the e_ prefix) (continued)

Format	Opcode			Mnemonic	Instruction	Page
	Primary ($Inst_{0:5}$)	Intermediate ($Inst_{6:20}$)	Extended ($Inst_{21:31}$)			
D8	0 0 1 1 1 0	t t t t t a a a a a	0 0 0 0 0 0 1 1 d d d d d d d d d	**e_ldu**	Load Doubleword with Update (reserved for 64b GPR)	
D14	0 0 1 1 0 0	t t t t t a a a a a	0 1 d d d d d d d d d d d d d d d	**e_lhz**	Load Halfword & Zero	-932
D8	0 0 1 1 1 0	t t t t t a a a a a	0 0 0 0 0 0 0 1 d d d d d d d d d	**e_lhzu**	Load Halfword & Zero with Update	-932
X	0 1 1 1 1 1	- - - - - - - - - -	- - - - - 0 1 0 0 0 1 0 1 1 1 /	**lhzx**	Load Halfword & Zero Indexed	Book E
X	0 1 1 1 1 1	- - - - - - - - - -	- - - - - 0 1 0 0 1 1 0 1 1 1 /	**lhzux**	Load Halfword & Zero with Update Indexed	Book E
D8	0 0 1 1 1 0	t t t t t a a a a a	0 0 0 0 0 1 0 1 0 d d d d d d d d	**e_lha**	Load Halfword Algebraic	-931
X	0 1 1 1 1 1	- - - - - - - - - -	- - - - - 0 1 0 1 0 1 0 1 1 1 /	**lhax**	Load Halfword Algebraic Indexed	Book E
D8	0 0 1 1 1 0	t t t t t a a a a a	0 0 0 0 1 0 1 1 d d d d d d d d d	**e_lhau**	Load Halfword Algebraic with Update	-931
X	0 1 1 1 1 1	- - - - - - - - - -	- - - - - 0 1 0 1 1 1 0 1 1 1 /	**lhaux**	Load Halfword Algebraic with Update Indexed	Book E
X	0 1 1 1 1 1	- - - - - - - - - -	- - - - - 1 1 0 0 0 1 0 1 1 0 /	**lhbrx**	Load Halfword Byte-Reverse Indexed	Book E
LI20	0 1 1 1 0 0	t t t t t 0 i i i i	i i i i i i i i i i i i i i i	**e_li**	Load Immediate	-933
LI20	0 1 1 1 0 0	t t t t t 1 i i i i	i i i i i i i i i i i i i i i	**e_lis**	Load Immediate Shifted	-933
D8	0 0 1 1 1 0	t t t t t a a a a a	0 0 0 0 0 1 0 0 0 d d d d d d d d	**e_lmw**	Load Multiple Word	-934
X	0 1 1 1 1 1	- - - - - - - - - -	- - - - - 0 0 0 0 0 1 0 1 0 0 /	**lwarx**	Load Word & Reserve Indexed	Book E
X	0 1 1 1 1 1	- - - - - - - - - -	- - - - - 1 0 0 0 0 1 0 1 1 0 /	**lwbrx**	Load Word Byte-Reverse Indexed	Book E
D14	0 0 1 1 0 0	t t t t t a a a a a	1 0 d d d d d d d d d d d d d d d	**e_lwz**	Load Word & Zero	-936
D8	0 0 1 1 1 0	t t t t t a a a a a	0 0 0 0 0 0 1 0 d d d d d d d d d	**e_lwzu**	Load Word & Zero with Update	-935
X	0 1 1 1 1 1	- - - - - - - - - -	- - - - - 0 0 0 0 0 1 0 1 1 1 /	**lwzx**	Load Word & Zero Indexed	Book E

Table 268. 32-bit instructions by mnemonic (ignoring the e_ prefix) (continued)

Format	Opcode			Mnemonic	Instruction	Page
	Primary ($Inst_{0:5}$)	Intermediate ($Inst_{6:20}$)	Extended ($Inst_{21:31}$)			
X	011111	----- ----- -----	00001 10111 /	**lwzux**	Load Word & Zero with Update Indexed	Book E
X	011111	----- ----- -----	11010 10110 /	**mbar**	Memory Barrier	Book E
X	011111	----- ----- -----	01000 10011 /	**mfapidi**	Move From APID Indirect	Book E
X	011111	----- ----- -----	00000 10011 /	**mfcr**	Move From Condition Register	Book E
XFX	011111	----- ----- -----	01010 00011 /	**mfdcr**	Move From Device Control Register	Book E
X	011111	----- ----- -----	00010 10011 /	**mfmsr**	Move From Machine State Register	Book E
XFX	011111	----- ----- -----	01010 10011 /	**mfspr**	Move From Special Purpose Register	Book E
X	011111	----- ----- -----	10010 10110 /	**msync**	Memory Synchronize	Book E
XFX	011111	----- ----- -----	00100 10000 /	**mtcrf**	Move To Condition Register Fields	Book E
X	011111	----- ----- -----	10000 00000 /	**mcrxr**	Move to Condition Register from XER	Book E
XFX	011111	----- ----- -----	01110 00011 /	**mtdcr**	Move To Device Control Register	Book E
X	011111	----- ----- -----	00100 10010 /	**mtmsr**	Move To Machine State Register	Book E
XFX	011111	----- ----- -----	01110 10011 /	**mtspr**	Move To Special Purpose Register	Book E
X	011111	----- ----- -----	/ 0010 01011 0	**mulhw**	Multiply High Word	Book E
X	011111	----- ----- -----	/ 0010 01011 1	**mulhw.**	Multiply High Word & record CR	Book E
X	011111	----- ----- -----	/ 0000 01011 0	**mulhwu**	Multiply High Word Unsigned	Book E
X	011111	----- ----- -----	/ 0000 01011 1	**mulhwu.**	Multiply High Word Unsigned & record CR	Book E
SCI8	001110	ttttt aaaaa 10110	FSSii iiiii i	**e_mulli**	Multiply Low Immediate	*-944*

Table 268. 32-bit instructions by mnemonic (ignoring the e_ prefix) (continued)

Format	Opcode			Mnemonic	Instruction	Page
	Primary ($Inst_{0:5}$)	Intermediate ($Inst_{6:20}$)	Extended ($Inst_{21:31}$)			
SCI8	0 0 1 1 1 0	t t t t t a a a a a 1 1 1 1 0	F S S i i i i i i i i i	**e_mulli.**	Multiply Low Immediate and Record	*-944*
X	0 1 1 1 1 1	- - - - - - - - - - - - - - -	0 0 1 1 1 0 1 0 1 1 0	**mullw**	Multiply Low Word	Book E
X	0 1 1 1 1 1	- - - - - - - - - - - - - - -	0 0 1 1 1 0 1 0 1 1 1	**mullw.**	Multiply Low Word & record CR	Book E
X	0 1 1 1 1 1	- - - - - - - - - - - - - - -	1 0 1 1 1 0 1 0 1 1 0	**mullwo**	Multiply Low Word & record OV	Book E
X	0 1 1 1 1 1	- - - - - - - - - - - - - - -	1 0 1 1 1 0 1 0 1 1 1	**mullwo.**	Multiply Low Word & record OV & CR	Book E
X	0 1 1 1 1 1	- - - - - - - - - - - - - - -	0 1 1 1 0 1 1 1 0 0 0	**nand**	NAND	Book E
X	0 1 1 1 1 1	- - - - - - - - - - - - - - -	0 1 1 1 0 1 1 1 0 0 1	**nand.**	NAND & record CR	Book E
X	0 1 1 1 1 1	- - - - - - - - - - - - - - -	0 0 0 1 1 0 1 0 0 0 0	**neg**	Negate	Book E
X	0 1 1 1 1 1	- - - - - - - - - - - - - - -	0 0 0 1 1 0 1 0 0 0 1	**neg.**	Negate & record CR	Book E
X	0 1 1 1 1 1	- - - - - - - - - - - - - - -	1 0 0 1 1 0 1 0 0 0 0	**nego**	Negate & record OV	Book E
X	0 1 1 1 1 1	- - - - - - - - - - - - - - -	1 0 0 1 1 0 1 0 0 0 1	**nego.**	Negate & record OV & record CR	Book E
X	0 1 1 1 1 1	- - - - - - - - - - - - - - -	0 0 0 1 1 1 1 1 0 0 0	**nor**	NOR	Book E
X	0 1 1 1 1 1	- - - - - - - - - - - - - - -	0 0 0 1 1 1 1 1 0 0 1	**nor.**	NOR & record CR	Book E
X	0 1 1 1 1 1	- - - - - - - - - - - - - - -	0 1 1 0 1 1 1 1 0 0 0	**or**	OR	Book E
X	0 1 1 1 1 1	- - - - - - - - - - - - - - -	0 1 1 0 1 1 1 1 0 0 1	**or.**	OR & record CR	Book E
X	0 1 1 1 1 1	- - - - - - - - - - - - - - -	0 1 1 0 0 1 1 1 0 0 0	**orc**	OR with Complement	Book E
X	0 1 1 1 1 1	- - - - - - - - - - - - - - -	0 1 1 0 0 1 1 1 0 0 1	**orc.**	OR with Complement & record CR	Book E
SCI8	0 0 1 1 1 0	t t t t t a a a a a 1 0 0 1 1	F S S i i i i i i i i i	**e_ori**	OR Immediate	*-948*
SCI8	0 0 1 1 1 0	t t t t t a a a a a 1 1 0 1 1	F S S i i i i i i i i i	**e_ori.**	OR Immediate and Record	*-951*

Table 268. 32-bit instructions by mnemonic (ignoring the e_ prefix) (continued)

Format	Opcode			Mnemonic	Instruction	Page
	Primary ($Inst_{0:5}$)	Intermediate ($Inst_{6:20}$)	Extended ($Inst_{21:31}$)			
	011101	SSSSS aaaaa hhhhh	bbbbb eeeee 1	**e_rlwinm.**	Rotate Left Word Immed then AND with Mask & record CR	*-953*
	011101	SSSSS aaaaa hhhhh	bbbbb eeeee 0	**e_rlwimi.**	Rotate Left Word Immed then Mask Insert & record CR	*-952*
X	011111	----- ----- -----	00000 11000 0	**slw**	Shift Left Word	Book E
X	011111	----- ----- -----	00000 11000 1	**slw.**	Shift Left Word & record CR	Book E
X	011111	----- ----- -----	11000 11000 0	**sraw**	Shift Right Algebraic Word	Book E
X	011111	----- ----- -----	11000 11000 1	**sraw.**	Shift Right Algebraic Word & record CR	Book E
X	011111	----- ----- -----	11001 11000 0	**srawi**	Shift Right Algebraic Word Immediate	Book E
X	011111	----- ----- -----	11001 11000 1	**srawi.**	Shift Right Algebraic Word Immediate & record CR	Book E
X	011111	----- ----- -----	10000 11000 0	**srw**	Shift Right Word	Book E
X	011111	----- ----- -----	10000 11000 1	**srw.**	Shift Right Word & record CR	Book E
D14	001101	ttttt aaaaa 00ddd	ddddd ddddd d	**e_stb**	Store Byte	*-958*
X	011111	----- ----- -----	00110 10111 /	**stbx**	Store Byte Indexed	Book E
D8	001110	ttttt aaaaa 00000	100dd ddddd d	**e_stbu**	Store Byte with Update	*-958*
X	011111	----- ----- -----	00111 10111 /	**stbux**	Store Byte with Update Indexed	*-958*
D14	001101	ttttt aaaaa 11ddd	ddddd ddddd d	**e_std**	Store Doubleword (reserved for 64b GPR)	Book E
D8	001110	ttttt aaaaa 00000	111dd ddddd d	**e_stdu**	Store Doubleword with Update (reserved for 64b GPR)	

Table 268. 32-bit instructions by mnemonic (ignoring the e_ prefix) (continued)

Format	Opcode			Mnemonic	Instruction	Page
	Primary ($Inst_{0:5}$)	Intermediate ($Inst_{6:20}$)	Extended ($Inst_{21:31}$)			
D14	0 0 1 1 0 1	t t t t t a a a a a 0 1 d d d	d d d d d d d d d d d	**e_sth**	Store Halfword	*-959*
X	0 1 1 1 1 1	- - - - - - - - - - - - - - -	1 1 1 0 0 1 0 1 1 0 /	**sthbrx**	Store Halfword Byte-Reverse Indexed	Book E
X	0 1 1 1 1 1	- - - - - - - - - - - - - - -	0 1 1 0 0 1 0 1 1 1 /	**sthx**	Store Halfword Indexed	Book E
D8	0 0 1 1 1 0	t t t t t a a a a a 0 0 0 0 0	1 0 1 d d d d d d d	**e_sthu**	Store Halfword with Update	*-959*
X	0 1 1 1 1 1	- - - - - - - - - - - - - - -	0 1 1 0 1 1 0 1 1 1 /	**sthux**	Store Halfword with Update Indexed	Book E
D8	0 0 1 1 1 0	t t t t t a a a a a 0 0 0 0 1	0 0 1 d d d d d d d	**e_stmw**	Store Multiple Word	Book E
D14	0 0 1 1 0 1	t t t t t a a a a a 1 0 d d d	d d d d d d d d d d d	**e_stw**	Store Word	*-961*
X	0 1 1 1 1 1	- - - - - - - - - - - - - - -	1 0 1 0 0 1 0 1 1 0 /	**stwbrx**	Store Word Byte-Reverse Indexed	Book E
X	0 1 1 1 1 1	- - - - - - - - - - - - - - -	0 0 1 0 0 1 0 1 1 0 1	**stwcx.**	Store Word Conditional Indexed & record CR	Book E
D8	0 0 1 1 1 0	t t t t t a a a a a 0 0 0 0 0	1 1 0 d d d d d d d	**e_stwu**	Store Word with Update	*-961*
X	0 1 1 1 1 1	- - - - - - - - - - - - - - -	0 0 1 0 1 1 0 1 1 1 /	**stwux**	Store Word with Update Indexed	Book E
X	0 1 1 1 1 1	- - - - - - - - - - - - - - -	0 0 1 0 0 1 0 1 1 1 /	**stwx**	Store Word Indexed	Book E
X	0 1 1 1 1 1	- - - - - - - - - - - - - - -	0 0 0 0 1 0 1 0 0 0 0	**subf**	Subtract From	Book E
X	0 1 1 1 1 1	- - - - - - - - - - - - - - -	0 0 0 0 1 0 1 0 0 0 1	**subf.**	Subtract From & record CR	Book E
X	0 1 1 1 1 1	- - - - - - - - - - - - - - -	0 0 0 0 0 0 1 0 0 0 0	**subfc**	Subtract From Carrying	Book E
X	0 1 1 1 1 1	- - - - - - - - - - - - - - -	0 0 0 0 0 0 1 0 0 0 1	**subfc.**	Subtract From Carrying & record CR	Book E
X	0 1 1 1 1 1	- - - - - - - - - - - - - - -	1 0 0 0 0 0 1 0 0 0 0	**subfco**	Subtract From Carrying & record OV	Book E
X	0 1 1 1 1 1	- - - - - - - - - - - - - - -	1 0 0 0 0 0 1 0 0 0 1	**subfco.**	Subtract From Carrying & record OV & CR	Book E

Table 268. 32-bit instructions by mnemonic (ignoring the e_ prefix) (continued)

Format	Opcode			Mnemonic	Instruction	Page
	Primary ($Inst_{0:5}$)	Intermediate ($Inst_{6:20}$)	Extended ($Inst_{21:31}$)			
X	011111	----- ----- -----	10001 01000 0	**subfo**	Subtract From & record OV	Book E
X	011111	----- ----- -----	10001 01000 1	**subfo.**	Subtract From & record OV & CR	Book E
X	011111	----- ----- -----	00100 01000 0	**subfe**	Subtract From Extended with CA	Book E
X	011111	----- ----- -----	00100 01000 1	**subfe.**	Subtract From Extended with CA & record CR	Book E
X	011111	----- ----- -----	10100 01000 0	**subfeo**	Subtract From Extended with CA & record OV	Book E
X	011111	----- ----- -----	10100 01000 1	**subfeo.**	Subtract From Extended with CA & record OV & CR	Book E
SCI8	001110	ttttt aaaaa	10100 FSSii iiiii i	**e_subfic**	Subtract from Immediate Carrying	*-964*
SCI8	001110	ttttt aaaaa	11100 FSSii iiiii i	**e_subfic.**	Subtract from Immediate Carrying and Record	*-964*
X	011111	----- ----- -----	00111 01000 0	**subfme**	Subtract From Minus One Extended with CA	Book E
X	011111	----- ----- -----	00111 01000 1	**subfme.**	Subtract From Minus One Extended with CA & record CR	Book E
X	011111	----- ----- -----	10111 01000 0	**subfmeo**	Subtract From Minus One Extended with CA & record OV	Book E
X	011111	----- ----- -----	10111 01000 1	**subfmeo.**	Subtract From Minus One Extended with CA & record OV & CR	Book E
X	011111	----- ----- -----	00110 01000 0	**subfze**	Subtract From Zero Extended with CA	Book E
X	011111	----- ----- -----	00110 01000 1	**subfze.**	Subtract From Zero Extended with CA & record CR	Book E

Table 268. 32-bit instructions by mnemonic (ignoring the e_ prefix) (continued)

Format	Opcode			Mnemonic	Instruction	Page
	Primary ($Inst_{0:5}$)	Intermediate ($Inst_{6:20}$)	Extended ($Inst_{21:31}$)			
X	011111	----- ----- -----	10110 01000 0	**subfzeo**	Subtract From Zero Extended with CA & record OV	Book E
X	011111	----- ----- -----	10110 01000 1	**subfzeo.**	Subtract From Zero Extended with CA & record OV & CR	Book E
X	011111	----- ----- -----	11000 10010 /	**tlbivax**	TLB Invalidate Virtual Address Indexed	Book E
X	011111	----- ----- -----	11101 10010 /	**tlbre**	TLB Read Entry	Book E
X	011111	----- ----- -----	11100 10010 ?	**tlbsx**	TLB Search Indexed	Book E
X	011111	----- ----- -----	10001 10110 /	**tlbsync**	TLB Synchronize	Book E
X	011111	----- ----- -----	11110 10010 /	**tlbwe**	TLB Write Entry	Book E
X	011111	----- ----- -----	00000 00100 /	**tw**	Trap Word	Book E
X	011111	----- ----- -----	00100 00011 /	**wrtee**	Write External Enable	Book E
X	011111	----- ----- -----	00101 00011 /	**wrteei**	Write External Enable Immediate	Book E
X	011111	----- ----- -----	01001 11100 0	**xor**	XOR	Book E
X	011111	----- ----- -----	01001 11100 1	**xor.**	XOR & record CR	Book E
SCI8	001110	ttttt aaaaa 10111	FSSii iiiii i	**e_xori**	XOR Immediate	*-966*
SCI8	001110	ttttt aaaaa 11111	FSSii iiiii i	**e_xori.**	XOR Immediate and Record	*-966*

Appendix A Instruction set listings

This appendix lists the instructions by both mnemonic and opcode, and includes a quick reference table with general information, such as the architecture level, privilege level, form, and whether the instruction is optional. The tables in the chapter are organized as follows:

- *Chapter A.1: Instructions sorted by mnemonic (decimal and hexadecimal)*"
- *Chapter A.2: Instructions sorted by primary opcodes (decimal and hexadecimal)*"
- *Chapter A.3: Instructions sorted by mnemonic (binary)*"
- *Chapter A.4: Instructions sorted by opcode (binary)*"
- *Chapter A.5: Instruction set legend*"

Note that this appendix does not include instructions defined by the VLE extension. These instructions are listed in *Chapter 14: VLE instruction index on page 862.*

A.1 Instructions sorted by mnemonic (decimal and hexadecimal)

Table 269 lists instructions in alphabetical order by mnemonic, showing decimal and hexadecimal values of the primary opcode (0–5) and binary values of the secondary opcode (21–31). This list also includes simplified mnemonics and their equivalents using standard mnemonics.

Table 269. Instructions sorted by mnemonic (decimal and hexadecimal)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
add	31 (0x1F)						rD					rA					rB					0	1	0	0	0	0	1	0	1	0	0	X	**add**
add.	31 (0x1F)						rD					rA					rB					0	1	0	0	0	0	1	0	1	0	1	X	**add.**
addc	31 (0x1F)						rD					rA					rB					0	0	0	0	0	0	1	0	1	0	0	X	**addc**
addc.	31 (0x1F)						rD					rA					rB					0	0	0	0	0	0	1	0	1	0	1	X	**addc.**
addco	31 (0x1F)						rD					rA					rB					1	0	0	0	0	0	1	0	1	0	0	X	**addco**
addco.	31 (0x1F)						rD					rA					rB					1	0	0	0	0	0	1	0	1	0	1	X	**addco.**
adde	31 (0x1F)						rD					rA					rB					0	0	1	0	0	0	1	0	1	0	0	X	**adde**
adde.	31 (0x1F)						rD					rA					rB					0	0	1	0	0	0	1	0	1	0	1	X	**adde.**
addeo	31 (0x1F)						rD					rA					rB					1	0	1	0	0	0	1	0	1	0	0	X	**addeo**
addeo.	31 (0x1F)						rD					rA					rB					1	0	1	0	0	0	1	0	1	0	1	X	**addeo.**
addi	14 (0x0E)						rD					rA					SIMM																D	**addi**
addic	12 (0x0C)						rD					rA					SIMM																D	**addic**
addic.	13 (0x0D)						rD					rA					SIMM																D	**addic.**
addis	15 (0x0F)						rD					rA					SIMM																D	**addis**
addme	31 (0x1F)						rD					rA					///					0	0	1	1	1	0	1	0	1	0	0	X	**addme**
addme.	31 (0x1F)						rD					rA					///					0	0	1	1	1	0	1	0	1	0	1	X	**addme.**
addmeo	31 (0x1F)						rD					rA					///					1	0	1	1	1	0	1	0	1	0	0	X	**addmeo**
addmeo.	31 (0x1F)						rD					rA					///					1	0	1	1	1	0	1	0	1	0	1	X	**addmeo.**
addo	31 (0x1F)						rD					rA					rB					1	1	0	0	0	0	1	0	1	0	0	X	**addo**
addo.	31 (0x1F)						rD					rA					rB					1	1	0	0	0	0	1	0	1	0	1	X	**addo.**

Table 269. Instructions sorted by mnemonic (decimal and hexadecimal) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
addze	31 (0x1F)						rD					rA					///					0	0	1	1	0	0	1	0	1	0	0	X	**addze**
addze.	31 (0x1F)						rD					rA					///					0	0	1	1	0	0	1	0	1	0	1	X	**addze.**
addzeo	31 (0x1F)						rD					rA					///					1	0	1	1	0	0	1	0	1	0	0	X	**addzeo**
addzeo.	31 (0x1F)						rD					rA					///					1	0	1	1	0	0	1	0	1	0	1	X	**addzeo.**
and	31 (0x1F)						rS					rA					rB					0	0	0	0	0	1	1	1	0	0	0	X	**and**
and.	31 (0x1F)						rS					rA					rB					0	0	0	0	0	1	1	1	0	0	1	X	**and.**
andc	31 (0x1F)						rS					rA					rB					0	0	0	0	1	1	1	1	0	0	0	X	**andc**
andc.	31 (0x1F)						rS					rA					rB					0	0	0	0	1	1	1	1	0	0	1	X	**andc.**
andi.	28 (0x1C)						rS					rA					UIMM																D	**andi.**
andis.	29 (0x1D)						rS					rA					UIMM																D	**andis.**
b	18 (0x12)						LI																								0	0	I	**b**
ba	18 (0x12)						LI																								1	0	I	**ba**
bc	16 (0x10)						BO					BI					BD														0	0	B	**bc**
bca	16 (0x10)						BO					BI					BD														1	0	B	**bca**
bcctr	19 (0x13)						BO					BI					///					1	0	0	0	0	1	0	0	0	0	0	XL	**bcctr**
bcctrl	19 (0x13)						BO					BI					///					1	0	0	0	0	1	0	0	0	0	1	XL	**bcctrl**
bcl	16 (0x10)						BO					BI					BD														0	1	B	**bcl**
bcla	16 (0x10)						BO					BI					BD														1	1	B	**bcla**
bclr	19 (0x13)						BO					BI					///					0	0	0	0	0	1	0	0	0	0	0	XL	**bclr**
bclrl	19 (0x13)						BO					BI					///					0	0	0	0	0	1	0	0	0	0	1	XL	**bclrl**
bctr	**bctr**(1)																equivalent to					**bcctr 20,0**												**bctr**
bctrl	**bctrl** [1]																equivalent to					**bcctrl 20,0**												**bctrl**
bdnz	**bdnz target** [1]																equivalent to					**bc 16,0,**target												**bdnz**
bdnza	**bdnza** target [1]																equivalent to					**bca 16,0,**target												**bdnza**
bdnzf	**bdnzf BI,**target																equivalent to					**bc 0,BI,**target												**bdnzf**
bdnzfa	**bdnzfa BI,**target																equivalent to					**bca 0,BI,**target												**bdnzfa**
bdnzfl	**bdnzfl BI,**target																equivalent to					**bcl 0,BI,**target												**bdnzfl**
bdnzfla	**bdnzfla BI,**target																equivalent to					**bcla 0,BI,**target												**bdnzfla**
bdnzflr	**bdnzflr BI**																equivalent to					**bclr 0,BI**												**bdnzflr**
bdnzflrl	**bdnzflrl BI**																equivalent to					**bclrl 0,BI**												**bdnzflrl**
bdnzl	**bdnzl** target [1]																equivalent to					**bcl 16,0,**target												**bdnzl**
bdnzla	**bdnzla** target [1]																equivalent to					**bcla 16,0,**target												**bdnzla**
bdnzlr	**bdnzlr BI**																equivalent to					**bclr 16,BI**												**bdnzlr**
bdnzlrl	**bdnzlrl** [1]																equivalent to					**bclrl 16,0**												**bdnzlrl**
bdnzt	**bdnzt BI,**target																equivalent to					**bc 8,BI,**target												**bdnzt**
bdnzta	**bdnzta BI,**target																equivalent to					**bca 8,BI,**target												**bdnzta**
bdnztl	**bdnztl BI,**target																equivalent to					**bcl 8,0,**target												**bdnztl**
bdnztla	**bdnztla BI,**target																equivalent to					**bcla 8,BI,**target												**bdnztla**
bdnztlr	**bdnztlr BI**																equivalent to					**bclr 8,BI**												**bdnztlr**
bdnztlr	**bdnztlr BI**																equivalent to					**bclr 8,BI**												**bdnztlr**

Table 269. Instructions sorted by mnemonic (decimal and hexadecimal) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
bdnztlrl	**bdnztlrl BI**																																equivalent to	**bclrl 8,BI**																																	**bdnztlrl**
bdz	**bdz** target [1]																																equivalent to	**bc 18,0,**target																																	**bdz**
bdza	**bdza** target [1]																																equivalent to	**bca 18,0,**target																																	**bdza**
bdzf	**bdzf BI,**target																																equivalent to	**bc 2,BI,**target																																	**bdzf**
bdzfa	**bdzfa BI,**target																																equivalent to	**bca 2,BI,**target																																	**bdzfa**
bdzfl	**bdzfl BI,**target																																equivalent to	**bcl 2,BI,**target																																	**bdzfl**
bdzfla	**bdzfla BI,**target																																equivalent to	**bcla 2,BI,**target																																	**bdzfla**
bdzflr	**bdzflr BI**																																equivalent to	**bclr 2,BI**																																	**bdzflr**
bdzflrl	**bdzflrl BI**																																equivalent to	**bclrl 2,BI**																																	**bdzflrl**
bdzl	**bdzl** target [1]																																equivalent to	**bcl 18,BI,**target																																	**bdzl**
bdzla	**bdzla** target [1]																																equivalent to	**bcla 18,BI,**target																																	**bdzla**
bdzlr	**bdzlr** [1]																																equivalent to	**bclr 18,0**																																	**bdzlr**
bdzlrl	**bdzlrl** [1]																																equivalent to	**bclrl 18,0**																																	**bdzlrl**
bdzt	**bdzt BI,**target																																equivalent to	**bc 10,BI,**target																																	**bdzt**
bdzta	**bdzta BI,**target																																equivalent to	**bca 10,BI,**target																																	**bdzta**
bdztl	**bdztl BI,**target																																equivalent to	**bcl 10,BI,**target																																	**bdztl**
bdztla	**bdztla BI,**target																																equivalent to	**bcla 10,BI,**target																																	**bdztla**
bdztlrl	**bdztlrl BI**																																equivalent to	**bclrl 10, BI**																																	**bdztlrl**
beq	**beq cr**S,target																																equivalent to	**bc 12,**BI[(2)],target																																	**beq**
beqa	**beqa cr**S,target																																equivalent to	**bca 12,**BI[2],target																																	**beqa**
beqctr	**beqctr cr**S,target																																equivalent to	**bcctr 12,**BI[2],target																																	**beqctr**
beqctrl	**beqctrl cr**S,target																																equivalent to	**bcctrl 12,**BI[2],target																																	**beqctrl**
beql	**beql cr**S,target																																equivalent to	**bcl 12,**BI[2],target																																	**beql**
beqla	**beqla cr**S,target																																equivalent to	**bcla 12,**BI[2],target																																	**beqla**
beqlr	**beqlr cr**S,target																																equivalent to	**bclr 12,**BI[2],target																																	**beqlr**
beqlrl	**beqlrl cr**S,target																																equivalent to	**bclrl 12,**BI[2],target																																	**beqlrl**
bf	**bf BI,**target																																equivalent to	**bc 4,BI,**target																																	**bf**
bfa	**bfa BI,**target																																equivalent to	**bca 4,BI,**target																																	**bfa**
bfctr	**bfctr BI**																																equivalent to	**bcctr 4,BI**																																	**bfctr**
bfctrl	**bfctrl BI**																																equivalent to	**bcctrl 4, BI**																																	**bfctrl**
bfl	**bfl BI,**target																																equivalent t	**bcl 4,BI,**target																																	**bfl**
bfla	**bfla BI,**target																																equivalent to	**bcla 4,BI,**target																																	**bfla**
bflr	**bflr BI**																																equivalent to	**bclr 4,BI**																																	**bflr**
bflrl	**bflrl BI**																																equivalent to	**bclrl 4,BI**																																	**bflrl**
bge	**bge cr**S,target																																equivalent to	**bc 4,**BI[(3)],target																																	**bge**
bgea	**bgea cr**S,target																																equivalent to	**bca 4,**BI[3],target																																	**bgea**
bgectr	**bgectr cr**S,target																																equivalent to	**bcctr 4,**BI[3],target																																	**bgectr**
bgectrl	**bgectrl cr**S,target																																equivalent to	**bcctrl 4,**BI[3],target																																	**bgectrl**
bgel	**bgel cr**S,target																																equivalent to	**bcl 4,**BI[3],target																																	**bgel**
bgela	**bgela cr**S,target																																equivalent to	**bcla 4,**BI[3],target																																	**bgela**

Table 269. Instructions sorted by mnemonic (decimal and hexadecimal) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
bgelr	**bgelr crS**,target													equivalent to					**bclr 4,BI**[3],target															**bgelr**
bgelrl	**bgelrl crS**,target													equivalent to					**bclrl 4,BI**[3],target															**bgelrl**
bgt	**bgt crS**,target													equivalent to					**bc 12,BI**[(4)],target															**bgt**
bgta	**bgta crS**,target													equivalent to					**bca 12,BI**[4],target															**bgta**
bgtctr	**bgtctr crS**,target													equivalent to					**bcctr 12,BI**[4],target															**bgtctr**
bgtctrl	**bgtctrl crS**,target													equivalent to					**bcctrl 12,BI**[4],target															**bgtctrl**
bgtl	**bgtl crS**,target													equivalent to					**bcl 12,BI**[4],target															**bgtl**
bgtla	**bgtla crS**,target													equivalent to					**bcla 12,BI**[4],target															**bgtla**
bgtlr	**bgtlr crS**,target													equivalent to					**bclr 12,BI**[4],target															**bgtlr**
bgtlrl	**bgtlrl crS**,target													equivalent to					**bclrl 12,BI**[4],target															**bgtlrl**
bl	18 (0x12)													LI																	0	1	I	**bl**
bla	18 (0x12)													LI																	1	1	I	**bla**
ble	**ble crS**,target													equivalent to					**bc 4,BI**[4],target															**ble**
blea	**blea crS**,target													equivalent to					**bca 4,BI**[4],target															**blea**
blectr	**blectr crS**,target													equivalent to					**bcctr 4,BI**[4],target															**blectr**
blectrl	**blectrl crS**,target													equivalent to					**bcctrl 4,BI**[4],target															**blectrl**
blel	**blel crS**,target													equivalent to					**bcl 4,BI**[4],target															**blel**
blela	**blela crS**,target													equivalent to					**bcla 4,BI**[4],target															**blela**
blelr	**blelr crS**,target													equivalent to					**bclr 4,BI**[4],target															**blelr**
blelrl	**blelrl crS**,target													equivalent to					**bclrl 4,BI**[4],target															**blelrl**
blr	**blr** [1]													equivalent to					**bclr 20,0**															**blr**
blrl	**blrl** [1]													equivalent to					**bclrl 20,0**															**blrl**
blt	**blt crS**,target													equivalent to					**bc 12,BI**,target															**blt**
blta	**blta crS**,target													equivalent to					**bca 12,BI**[3],target															**blta**
bltctr	**bltctr crS**,target													equivalent to					**bcctr 12,BI**[3],target															**bltctr**
bltctrl	**bltctrl crS**,target													equivalent to					**bcctrl 12,BI**[3],target															**bltctrl**
bltl	**bltl crS**,target													equivalent to					**bcl 12,BI**[3],target															**bltl**
bltla	**bltla crS**,target													equivalent to					**bcla 12,BI**[3],target															**bltla**
bltlr	**bltlr crS**,target													equivalent to					**bclr 12,BI**[3],target															**bltlr**
bltlrl	**bltlrl crS**,target													equivalent to					**bclrl 12,BI**[3],target															**bltlrl**
bne	**bne crS**,target													equivalent to					**bc 4,BI**[3],target															**bne**
bnea	**bnea crS**,target													equivalent to					**bca 4,BI**[3],target															**bnea**
bnectr	**bnectr crS**,target													equivalent to					**bcctr 4,BI**[3],target															**bnectr**
bnectrl	**bnectrl crS**,target													equivalent to					**bcctrl 4,BI**[3],target															**bnectrl**
bnel	**bnel crS**,target													equivalent to					**bcl 4,BI**[3],target															**bnel**
bnela	**bnela crS**,target													equivalent to					**bcla 4,BI**[3],target															**bnela**
bnelr	**bnelr crS**,target													equivalent to					**bclr 4,BI**[3],target															**bnelr**
bnelrl	**bnelrl crS**,target													equivalent to					**bclrl 4,BI**[3],target															**bnelrl**
bng	**bng crS**,target													equivalent to					**bc 4,BI**[4],target															**bng**
bnga	**bnga crS**,target													equivalent to					**bca 4,BI**[4],target															**bnga**

Table 269. Instructions sorted by mnemonic (decimal and hexadecimal) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
bngctr	**bngctr crS**,target												equivalent to			**bcctr 4,BI[4],target**																		**bngctr**
bngctrl	**bngctrl crS**,target												equivalent to			**bcctrl 4,BI[4],target**																		**bngctrl**
bngl	**bngl crS**,target												equivalent to			**bcl 4,BI[4],target**																		**bngl**
bngla	**bngla crS**,target												equivalent to			**bcla 4,BI[4],target**																		**bngla**
bnglr	**bnglr crS**,target												equivalent to			**bclr 4,BI[4],target**																		**bnglr**
bnglrl	**bnglrl crS**,target												equivalent to			**bclrl 4,BI[4],target**																		**bnglrl**
bnl	**bnl crS**,target												equivalent to			**bc 4,BI[3],target**																		**bnl**
bnla	**bnla crS**,target												equivalent to			**bca 4,BI[3],target**																		**bnla**
bnlctr	**bnlctr crS**,target												equivalent to			**bcctr 4,BI[3],target**																		**bnlctr**
bnlctrl	**bnlctrl crS**,target												equivalent to			**bcctrl 4,BI[3],target**																		**bnlctrl**
bnll	**bnll crS**,target												equivalent to			**bcl 4,BI[3],target**																		**bnll**
bnlla	**bnlla crS**,target												equivalent to			**bcla 4,BI[3],target**																		**bnlla**
bnllr	**bnllr crS**,target												equivalent to			**bclr 4,BI[3],target**																		**bnllr**
bnllrl	**bnllrl crS**,target												equivalent to			**bclrl 4,BI[3],target**																		**bnllrl**
bns	**bns crS**,target												equivalent to			**bc 4,BI[(5)],target**																		**bns**
bnsa	**bnsa crS**,target												equivalent to			**bca 4,BI[5],target**																		**bnsa**
bnsctr	**bnsctr crS**,target												equivalent to			**bcctr 4,BI[5],target**																		**bnsctr**
bnsctrl	**bnsctrl crS**,target												equivalent to			**bcctrl 4,BI[5],target**																		**bnsctrl**
bnsl	**bnsl crS**,target												equivalent to			**bcl 4,BI[5],target**																		**bnsl**
bnsla	**bnsla crS**,target												equivalent to			**bcla 4,BI[5],target**																		**bnsla**
bnslr	**bnslr crS**,target												equivalent to			**bclr 4,BI[5],target**																		**bnslr**
bnslrl	**bnslrl crS**,target												equivalent to			**bclrl 4,BI[5],target**																		**bnslrl**
bnu	**bnu crS**,target												equivalent to			**bc 4,BI[5],target**																		**bnu**
bnua	**bnua crS**,target												equivalent to			**bca 4,BI[5],target**																		**bnua**
bnuctr	**bnuctr crS**,target												equivalent to			**bcctr 4,BI[5],target**																		**bnuctr**
bnuctrl	**bnuctrl crS**,target												equivalent to			**bcctrl 4,BI[5],target**																		**bnuctrl**
bnul	**bnul crS**,target												equivalent to			**bcl 4,BI[5],target**																		**bnul**
bnula	**bnula crS**,target												equivalent to			**bcla 4,BI[5],target**																		**bnula**
bnulr	**bnulr crS**,target												equivalent to			**bclr 4,BI[5],target**																		**bnulr**
bnulrl	**bnulrl crS**,target												equivalent to			**bclrl 4,BI[5],target**																		**bnulrl**
brinc	04						rD					rA				rB					0	1	0	0	0	0	0	1	1	1	1	EVX	**brinc**
bso	**bso crS**,target												equivalent to			**bc 12,BI[5],target**																		**bso**
bsoa	**bsoa crS**,target												equivalent to			**bca 12,BI[5],target**																		**bsoa**
bsoctr	**bsoctr crS**,target												equivalent to			**bcctr 12,BI[5],target**																		**bsoctr**
bsoctrl	**bsoctrl crS**,target												equivalent to			**bcctrl 12,BI[5],target**																		**bsoctrl**
bsol	**bsol crS**,target												equivalent to			**bcl 12,BI[5],target**																		**bsol**
bsola	**bsola crS**,target												equivalent to			**bcla 12,BI[5],target**																		**bsola**
bsolr	**bsolr crS**,target												equivalent to			**bclr 12,BI[5],target**																		**bsolr**
bsolrl	**bsolrl crS**,target												equivalent to			**bclrl 12,BI[5],target**																		**bsolrl**
bt	**bt BI**,target												equivalent to			**bc 12,BI**,target																		**bt**

Table 269. Instructions sorted by mnemonic (decimal and hexadecimal) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
bta	**bta BI,**target											equivalent to					**bca 12,BI,**target																		**bta**
btctr	**btctr BI**											equivalent to					**bcctr 12,BI**																		**btctr**
btctrl	**btctrl BI**											equivalent to					**bcctrl 12,BI**																		**btctrl**
btl	**btl BI,**target											equivalent to					**bcl 12,BI,**target																		**btl**
btla	**btla BI,**target											equivalent to					**bcla 12,BI,**target																		**btla**
btlr	**btlr BI**											equivalent to					**bclr 12,BI**																		**btlr**
btlrl	**btlrl BI**											equivalent to					**bclrl 12,BI**																		**btlrl**
bun	**bun crS,**target											equivalent to					**bc 12,**BI[5]**,**target																		**bun**
buna	**buna crS,**target											equivalent to					**bca 12,**BI[5]**,**target																		**buna**
bunctr	**bunctr crS,**target											equivalent to					**bcctr 12,**BI[5]**,**target																		**bunctr**
bunctrl	**bunctrl crS,**target											equivalent to					**bcctrl 12,**BI[5]**,**target																		**bunctrl**
bunl	**bunl crS,**target											equivalent to					**bcl 12,**BI[5]**,**target																		**bunl**
bunla	**bunla crS,**target											equivalent to					**bcla 12,**BI[5]**,**target																		**bunla**
bunlr	**bunlr crS,**target											equivalent to					**bclr 12,**BI[5]**,**target																		**bunlr**
bunlrl	**bunlrl crS,**target											equivalent to					**bclrl 12,**BI[5]**,**target																		**bunlrl**
clrlslwi	**clrlslwi rA,rS,***b***,***n* (*n* £ *b* £ 31)											equivalent to					**rlwinm rA,rS,***n***,***b* − *n*,31 − *n*																		**clrlslwi**
clrlwi	**clrlwi rA,rS,***n* (n < 32)											equivalent to					**rlwinm rA,rS,0,***n***,31**																		**clrlwi**
clrrwi	**clrrwi rA,rS,***n* (n < 32)											equivalent to					**rlwinm rA,rS,0,0,31** − *n*																		**clrrwi**
cmp	31 (0x1F)			crfD			/	L		rA						rB					0	0	0	0	0	0	0	0	0	0	/	X	**cmp**
cmpi	11 (0x0B)			crfD			/	L		rA						SIMM																	D	**cmpi**
cmpl	31 (0x1F)			/	L		rA					rB					///				0	0	0	0	1	0	0	0	0	0	/	X	**cmpl**
cmpli	10 (0x0A)			crfD			/	L		rA						UIMM																	D	**cmpli**
cmplw	**cmplw crD,rA,rB**											equivalent to					**cmpl crD,0,rA,rB**																		**cmplw**
cmplwi	**cmplwi crD,rA,UIMM**											equivalent to					**cmpli crD,0,rA,**UIMM																		**cmplwi**
cmpw	**cmpw crD,rA,rB**											equivalent to					**cmp crD,0,rA,rB**																		**cmpw**
cmpwi	**cmpwi crD,rA,SIMM**											equivalent to					**cmpi crD,0,rA,**SIMM																		**cmpwi**
cntlzw	31 (0x1F)			rS						rA						///					0	0	0	0	0	1	1	0	1	0	0	X	**cntlzw**
cntlzw.	31 (0x1F)			rS						rA						///					0	0	0	0	0	1	1	0	1	0	1	X	**cntlzw.**
crand	19 (0x13)			crbD						crbA						crbB					0	1	0	0	0	0	0	0	0	1	/	XL	**crand**
crandc	19 (0x13)			crbD						crbA						crbB					0	0	1	0	0	0	0	0	0	1	/	XL	**crandc**
crclr	**crclr bx**											equivalent to					**crxor** bx,bx,bx																		**crclr**
creqv	19 (0x13)			crbD						crbA						crbB					0	1	0	0	1	0	0	0	0	1	/	XL	**creqv**
crmove	**crmove bx,by**											equivalent to					**cror** bx,by,by																		**crmove**
crnand	19 (0x13)			crbD						crbA						crbB					0	0	1	1	1	0	0	0	0	1	/	XL	**crnand**
crnor	19 (0x13)			crbD						crbA						crbB					0	0	0	0	1	0	0	0	0	1	/	XL	**crnor**
crnot	**crnot bx,by**											equivalent to					**crnor** bx,by,by																		**crnot**
cror	19 (0x13)			crbD						crbA						crbB					0	1	1	1	0	0	0	0	0	1	/	XL	**cror**
crorc	19 (0x13)			crbD						crbA						crbB					0	1	1	0	1	0	0	0	0	1	/	XL	**crorc**
crset	**crset bx**											equivalent to					**creqv** bx,bx,bx																		**crset**
crxor	19 (0x13)			crbD						crbA						crbB					0	0	1	1	0	0	0	0	0	1	/	XL	**crxor**

Table 269. Instructions sorted by mnemonic (decimal and hexadecimal) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
dcba(6)	31 (0x1F)						///					rA					rB					1	0	1	1	1	1	0	1	1	0	/	X	**dcba**
dcbf	31 (0x1F)						///					rA					rB					0	0	0	1	0	1	0	1	1	0	/	X	**dcbf**
dcbi (7)	31 (0x1F)						///					rA					rB					0	1	1	1	0	1	0	1	1	0	/	X	**dcbi**
dcblc	31 (0x1F)						CT					rA					rB					0	1	1	0	0	0	0	1	1	0	0	X	**dcblc**
dcbst	31 (0x1F)						///					rA					rB					0	0	0	0	1	1	0	1	1	0	/	X	**dcbst**
dcbt	31 (0x1F)						CT					rA					rB					0	1	0	0	0	1	0	1	1	0	/	X	**dcbt**
dcbtls	31 (0x1F)						CT					rA					rB					0	0	1	0	1	0	0	1	1	0	0	X	**dcbtls**
dcbtst	31 (0x1F)						CT					rA					rB					0	0	1	1	1	1	0	1	1	0	/	X	**dcbtst**
dcbtstls	31 (0x1F)						CT					rA					rB					0	0	1	0	0	0	0	1	1	0	0	X	**dcbtstls**
dcbz	31 (0x1F)						///					rA					rB					1	1	1	1	1	1	0	1	1	0	/	X	**dcbz**
divw	31 (0x1F)						rD					rA					rB					0	1	1	1	1	0	1	0	1	1	0	X	**divw**
divw.	31 (0x1F)						rD					rA					rB					0	1	1	1	1	0	1	0	1	1	1	X	**divw.**
divwo	31 (0x1F)						rD					rA					rB					1	1	1	1	1	0	1	0	1	1	0	X	**divwo**
divwo.	31 (0x1F)						rD					rA					rB					1	1	1	1	1	0	1	0	1	1	1	X	**divwo.**
divwu	31 (0x1F)						rD					rA					rB					0	1	1	1	0	0	1	0	1	1	0	X	**divwu**
divwu.	31 (0x1F)						rD					rA					rB					0	1	1	1	0	0	1	0	1	1	1	X	**divwu.**
divwuo	31 (0x1F)						rD					rA					rB					1	1	1	1	0	0	1	0	1	1	0	X	**divwuo**
divwuo.	31 (0x1F)						rD					rA					rB					1	1	1	1	0	0	1	0	1	1	1	X	**divwuo.**
dss	**dss STRM**						equivalent to					**dss** STRM,**0**																							**dss**
efdabs	04						rD					rA					///					0	1	0	1	1	1	0	0	1	0	0	EFX	**efdabs**
efdadd	04						rD					rA					rB					0	1	0	1	1	1	0	0	0	0	0	EFX	**efdadd**
efdcfs	04						rD					0	0	0	0	0	rB					0	1	0	1	1	1	0	1	1	1	1	EFX	**efdcfs**
efdcfsf	04						rD					///					rB					0	1	0	1	1	1	1	0	0	1	1	EFX	**efdcfsf**
efdcfsi	04						rD					///					rB					0	1	0	1	1	1	1	0	0	0	1	EFX	**efdcfsi**
efdcfuf	04						rD					///					rB					0	1	0	1	1	1	1	0	0	1	0	EFX	**efdcfuf**
efdcfui	04						rD					///					rB					0	1	0	1	1	1	1	0	0	0	0	EFX	**efdcfui**
efdcmpeq	04						crfD		/		/	rA					rB					0	1	0	1	1	1	0	1	1	1	0	EFX	**efdcmpeq**
efdcmpgt	04						crfD		/		/	rA					rB					0	1	0	1	1	1	0	1	1	0	0	EFX	**efdcmpgt**
efdcmplt	04						crfD		/		/	rA					rB					0	1	0	1	1	1	0	1	1	0	1	EFX	**efdcmplt**
efdctsf	04						rD					///					rB					0	1	0	1	1	1	1	0	1	1	1	EFX	**efdctsf**
efdctsi	04						rD					///					rB					0	1	0	1	1	1	1	0	1	0	1	EFX	**efdctsi**
efdctsiz	04						rD					///					rB					0	1	0	1	1	1	1	1	0	1	0	EFX	**efdctsiz**
efdctuf	04						rD					///					rB					0	1	0	1	1	1	1	0	1	1	0	EFX	**efdctuf**
efdctui	04						rD					///					rB					0	1	0	1	1	1	1	0	1	0	0	EFX	**efdctui**
efdctuiz	04						rD					///					rB					0	1	0	1	1	1	1	1	0	0	0	EFX	**efdctuiz**
efddiv	04						rD					rA					rB					0	1	0	1	1	1	0	1	0	0	1	EFX	**efddiv**
efdmul	04						rD					rA					rB					0	1	0	1	1	1	0	1	0	0	0	EFX	**efdmul**
efdnabs	04						rD					rA					///					0	1	0	1	1	1	0	0	1	0	1	EFX	**efdnabs**
efdneg	04						rD					rA					///					0	1	0	1	1	1	0	0	1	1	0	EFX	**efdneg**
efdsub	04						rD					rA					rB					0	1	0	1	1	1	0	0	0	0	1	EFX	**efdsub**

Table 269. Instructions sorted by mnemonic (decimal and hexadecimal) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
efdtsteq	04						crfD			/	/	rA					rB					0	1	0	1	1	1	1	1	1	0	EFX	**efdtsteq**
efdtstgt	04						crfD			/	/	rA					rB					0	1	0	1	1	1	1	1	1	0	EFX	**efdtstgt**
efdtstlt	04						crfD			/	/	rA					rB					0	1	0	1	1	1	1	1	1	1	EFX	**efdtstlt**
efsabs	04						rD					rA					///					0	1	0	1	1	0	0	0	1	0	EFX	**efsabs**
efsadd	04						rD					rA					rB					0	1	0	1	1	0	0	0	0	0	EFX	**efsadd**
efscfd	04						rD					0	0	0	0	0	rB					0	1	0	1	1	0	0	1	1	1	EFX	**efscfd**
efscfsf	04						rD					///					rB					0	1	0	1	1	0	1	0	0	1	EFX	**efscfsf**
efscfsi	04						rD					///					rB					0	1	0	1	1	0	1	0	0	0	EFX	**efscfsi**
efscfuf	04						rD					///					rB					0	1	0	1	1	0	1	0	0	1	EFX	**efscfuf**
efscfui	04						rD					///					rB					0	1	0	1	1	0	1	0	0	0	EFX	**efscfui**
efscmpeq	04						crfD			/	/	rA					rB					0	1	0	1	1	0	0	1	1	1	EFX	**efscmpeq**
efscmpgt	04						crfD			/	/	rA					rB					0	1	0	1	1	0	0	1	1	0	EFX	**efscmpgt**
efscmplt	04						crfD			/	/	rA					rB					0	1	0	1	1	0	0	1	1	0	EFX	**efscmplt**
efsctsf	04						rD					///					rB					0	1	0	1	1	0	1	0	1	1	EFX	**efsctsf**
efsctsi	04						rD					///					rB					0	1	0	1	1	0	1	0	1	0	EFX	**efsctsi**
efsctsiz	04						rD					///					rB					0	1	0	1	1	0	1	1	0	1	EFX	**efsctsiz**
efsctuf	04						rD					///					rB					0	1	0	1	1	0	1	0	1	1	EFX	**efsctuf**
efsctui	04						rD					///					rB					0	1	0	1	1	0	1	0	1	0	EFX	**efsctui**
efsctuiz	04						rD					///					rB					0	1	0	1	1	0	1	1	0	0	EFX	**efsctuiz**
efsdiv	04						rD					rA					rB					0	1	0	1	1	0	0	1	0	0	EFX	**efsdiv**
efsmul	04						rD					rA					rB					0	1	0	1	1	0	0	1	0	0	EFX	**efsmul**
efsnabs	04						rD					rA					///					0	1	0	1	1	0	0	0	1	0	EFX	**efsnabs**
efsneg	04						rD					rA					///					0	1	0	1	1	0	0	0	1	1	EFX	**efsneg**
efssub	04						rD					rA					rB					0	1	0	1	1	0	0	0	0	1	EFX	**efssub**
efststeq	04						crfD			/	/	rA					rB					0	1	0	1	1	0	1	1	1	0	EFX	**efststeq**
efststgt	04						crfD			/	/	rA					rB					0	1	0	1	1	0	1	1	1	0	EFX	**efststgt**
efststlt	04						crfD			/	/	rA					rB					0	1	0	1	1	0	1	1	1	1	EFX	**efststlt**
eqv	31 (0x1F)						rD					rA					rB					0	1	0	0	0	1	1	1	0	0	X	**eqv**
eqv.	31 (0x1F)						rD					rA					rB					0	1	0	0	0	1	1	1	0	1	X	**eqv.**
evabs	31 (0x1F)						rD					rA					///					0	1	0	0	0	0	0	1	0	0	EVX	**evabs**
evaddiw	31 (0x1F)						rD					UIMM					rB					0	1	0	0	0	0	0	0	0	1	EVX	**evaddiw**
evaddsmiaaw	31 (0x1F)						rD					rA					///					1	0	0	1	1	0	0	1	0	1	EVX	**evaddsmiaaw**
evaddssiaaw	31 (0x1F)						rD					rA					///					1	0	0	1	1	0	0	0	0	1	EVX	**evaddssiaaw**
evaddumiaaw	31 (0x1F)						rD					rA					///					1	0	0	1	1	0	0	1	0	0	EVX	**evaddumiaaw**
evaddusiaaw	31 (0x1F)						rD					rA					///					1	0	0	1	1	0	0	0	0	0	EVX	**evaddusiaaw**
evaddw	31 (0x1F)						rD					rA					rB					0	1	0	0	0	0	0	0	0	0	EVX	**evaddw**
evand	31 (0x1F)						rD					rA					rB					0	1	0	0	0	0	1	0	0	0	EVX	**evand**

Table 269. Instructions sorted by mnemonic (decimal and hexadecimal) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
evandc	31 (0x1F)						rD					rA					rB					0	1	0	0	0	0	1	0	0	1	0	EVX	**evandc**
evcmpeq	31 (0x1F)						crfD			/	/	rA					rB					0	1	0	0	0	1	1	0	1	0	0	EVX	**evcmpeq**
evcmpgts	31 (0x1F)						crfD			/	/	rA					rB					0	1	0	0	0	1	1	0	0	0	1	EVX	**evcmpgts**
evcmpgtu	31 (0x1F)						crfD			/	/	rA					rB					0	1	0	0	0	1	1	0	0	0	0	EVX	**evcmpgtu**
evcmplts	31 (0x1F)						crfD			/	/	rA					rB					0	1	0	0	0	1	1	0	0	1	1	EVX	**evcmplts**
evcmpltu	31 (0x1F)						crfD			/	/	rA					rB					0	1	0	0	0	1	1	0	0	1	0	EVX	**evcmpltu**
evcntlsw	31 (0x1F)						rD					rA					///					0	1	0	0	0	0	0	1	1	1	0	EVX	**evcntlsw**
evcntlzw	31 (0x1F)						rD					rA					///					0	1	0	0	0	0	0	1	1	0	1	EVX	**evcntlzw**
evdivws	31 (0x1F)						rD					rA					rB					1	0	0	1	1	0	0	0	1	1	0	EVX	**evdivws**
evdivwu	31 (0x1F)						rD					rA					rB					1	0	0	1	1	0	0	0	1	1	1	EVX	**evdivwu**
eveqv	31 (0x1F)						rD					rA					rB					0	1	0	0	0	0	1	1	0	0	1	EVX	**eveqv**
evextsb	31 (0x1F)						rD					rA					///					0	1	0	0	0	0	0	1	0	1	0	EVX	**evextsb**
evextsh	31 (0x1F)						rD					rA					///					0	1	0	0	0	0	0	1	0	1	1	EVX	**evextsh**
evfsabs	31 (0x1F)						rD					rA					///					0	1	0	1	0	0	0	0	1	0	0	EVX	**evfsabs**
evfsadd	31 (0x1F)						rD					rA					rB					0	1	0	1	0	0	0	0	0	0	0	EVX	**evfsadd**
evfscfsf	31 (0x1F)						rD					///					rB					0	1	0	1	0	0	1	0	0	1	1	EVX	**evfscfsf**
evfscfsi	31 (0x1F)						rD					///					rB					0	1	0	1	0	0	1	0	0	0	1	EVX	**evfscfsi**
evfscfuf	31 (0x1F)						rD					///					rB					0	1	0	1	0	0	1	0	0	1	0	EVX	**evfscfuf**
evfscfui	31 (0x1F)						rD					///					rB					0	1	0	1	0	0	1	0	0	0	0	EVX	**evfscfui**
evfscmpeq	31 (0x1F)						crfD			/	/	rA					rB					0	1	0	1	0	0	0	1	1	1	0	EVX	**evfscmpeq**
evfscmpgt	31 (0x1F)						crfD			/	/	rA					rB					0	1	0	1	0	0	0	1	1	0	0	EVX	**evfscmpgt**
evfscmplt	31 (0x1F)						crfD			/	/	rA					rB					0	1	0	1	0	0	0	1	1	0	1	EVX	**evfscmplt**
evfsctsf	31 (0x1F)						rD					///					rB					0	1	0	1	0	0	1	0	1	1	1	EVX	**evfsctsf**
evfsctsi	31 (0x1F)						rD					///					rB					0	1	0	1	0	0	1	0	1	0	1	EVX	**evfsctsi**
evfsctsiz	31 (0x1F)						rD					///					rB					0	1	0	1	0	0	1	1	0	1	0	EVX	**evfsctsiz**
evfsctuf	31 (0x1F)						rD					///					rB					0	1	0	1	0	0	1	0	1	1	0	EVX	**evfsctuf**
evfsctui	31 (0x1F)						rD					///					rB					0	1	0	1	0	0	1	0	1	0	0	EVX	**evfsctui**
evfsctuiz	31 (0x1F)						rD					///					rB					0	1	0	1	0	0	1	1	0	0	0	EVX	**evfsctuiz**
evfsdiv	31 (0x1F)						rD					rA					rB					0	1	0	1	0	0	0	1	0	0	1	EVX	**evfsdiv**
evfsmul	31 (0x1F)						rD					rA					rB					0	1	0	1	0	0	0	1	0	0	0	EVX	**evfsmul**
evfsnabs	31 (0x1F)						rD					rA					///					0	1	0	1	0	0	0	0	1	0	1	EVX	**evfsnabs**
evfsneg	31 (0x1F)						rD					rA					///					0	1	0	1	0	0	0	0	1	1	0	EVX	**evfsneg**
evfssub	31 (0x1F)						rD					rA					rB					0	1	0	1	0	0	0	0	0	0	1	EVX	**evfssub**
evfststeq	31 (0x1F)						crfD			/	/	rA					rB					0	1	0	1	0	0	1	1	1	1	0	EVX	**evfststeq**
evfststgt	31 (0x1F)						crfD			/	/	rA					rB					0	1	0	1	0	0	1	1	1	0	0	EVX	**evfststgt**
evfststlt	31 (0x1F)						crfD			/	/	rA					rB					0	1	0	1	0	0	1	1	1	0	1	EVX	**evfststlt**
evldd	31 (0x1F)						rD					rA					UIMM [8]					0	1	1	0	0	0	0	0	0	0	1	EVX	**evldd**

Table 269. Instructions sorted by mnemonic (decimal and hexadecimal) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
evlddx	31 (0x1F)						rD					rA					rB					0	1	1	0	0	0	0	0	0	0	EVX	**evlddx**
evldh	31 (0x1F)						rD					rA					UIMM [8]					0	1	1	0	0	0	0	0	1	0	1	EVX	**evldh**
evldhx	31 (0x1F)						rD					rA					rB					0	1	1	0	0	0	0	0	1	0	0	EVX	**evldhx**
evldw	31 (0x1F)						rD					rA					UIMM [8]					0	1	1	0	0	0	0	0	0	1	1	EVX	**evldw**
evldwx	31 (0x1F)						rD					rA					rB					0	1	1	0	0	0	0	0	0	1	0	EVX	**evldwx**
evlhhesplat	31 (0x1F)						rD					rA					UIMM [8]					0	1	1	0	0	0	0	1	0	0	1	EVX	**evlhhesplat**
evlhhesplatx	31 (0x1F)						rD					rA					rB					0	1	1	0	0	0	0	1	0	0	0	EVX	**evlhhesplatx**
evlhhossplat	31 (0x1F)						rD					rA					UIMM [9]					0	1	1	0	0	0	0	1	1	1	1	EVX	**evlhhossplat**
evlhhossplatx	31 (0x1F)						rD					rA					rB					0	1	1	0	0	0	0	1	1	1	0	EVX	**evlhhossplatx**
evlhhousplat	31 (0x1F)						rD					rA					UIMM [9]					0	1	1	0	0	0	0	1	1	0	1	EVX	**evlhhousplat**
evlhhousplatx	31 (0x1F)						rD					rA					rB					0	1	1	0	0	0	0	1	1	0	0	EVX	**evlhhousplatx**
evlwhe	31 (0x1F)						rD					rA					UIMM [8]					0	1	1	0	0	0	1	0	0	0	1	EVX	**evlwhe**
evlwhex	31 (0x1F)						rD					rA					rB					0	1	1	0	0	0	1	0	0	0	0	EVX	**evlwhex**
evlwhos	31 (0x1F)						rD					rA					UIMM [10]					0	1	1	0	0	0	1	0	1	1	1	EVX	**evlwhos**
evlwhosx	31 (0x1F)						rD					rA					rB					0	1	1	0	0	0	1	0	1	1	0	EVX	**evlwhosx**
evlwhou	31 (0x1F)						rD					rA					UIMM [10]					0	1	1	0	0	0	1	0	1	0	1	EVX	**evlwhou**
evlwhoux	31 (0x1F)						rD					rA					rB					0	1	1	0	0	0	1	0	1	0	0	EVX	**evlwhoux**
evlwhsplat	31 (0x1F)						rD					rA					UIMM [10]					0	1	1	0	0	0	1	1	1	0	1	EVX	**evlwhsplat**
evlwhsplatx	31 (0x1F)						rD					rA					rB					0	1	1	0	0	0	1	1	1	0	0	EVX	**evlwhsplatx**
evlwwsplat	31 (0x1F)						rD					rA					UIMM [10]					0	1	1	0	0	0	1	1	0	0	1	EVX	**evlwwsplat**
evlwwsplatx	31 (0x1F)						rD					rA					rB					0	1	1	0	0	0	1	1	0	0	0	EVX	**evlwwsplatx**
evmergehi	31 (0x1F)						rD					rA					rB					0	1	0	0	0	1	0	1	1	0	0	EVX	**evmergehi**
evmergehilo	31 (0x1F)						rD					rA					rB					0	1	0	0	0	1	0	1	1	1	0	EVX	**evmergehilo**
evmergelo	31 (0x1F)						rD					rA					rB					0	1	0	0	0	1	0	1	1	0	1	EVX	**evmergelo**
evmergelohi	31 (0x1F)						rD					rA					rB					0	1	0	0	0	1	0	1	1	1	1	EVX	**evmergelohi**
evmhegsmfaa	31 (0x1F)						rD					rA					rB					1	0	1	0	0	1	0	1	0	1	1	EVX	**evmhegsmfaa**
evmhegsmfan	31 (0x1F)						rD					rA					rB					1	0	1	1	0	1	0	1	0	1	1	EVX	**evmhegsmfan**
evmhegsmiaa	31 (0x1F)						rD					rA					rB					1	0	1	0	0	1	0	1	0	0	1	EVX	**evmhegsmiaa**
evmhegsmian	31 (0x1F)						rD					rA					rB					1	0	1	1	0	1	0	1	0	0	1	EVX	**evmhegsmian**
evmhegumiaa	31 (0x1F)						rD					rA					rB					1	0	1	0	0	1	0	1	0	0	0	EVX	**evmhegumiaa**

Table 269. Instructions sorted by mnemonic (decimal and hexadecimal) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
evmhegumian	31 (0x1F)						rD					rA					rB					1	0	1	1	0	1	0	1	0	0	0	EVX	evmhegumian
evmhesmf	31 (0x1F)						rD					rA					rB					1	0	0	0	0	0	0	1	0	1	1	EVX	evmhesmf
evmhesmfa	31 (0x1F)						rD					rA					rB					1	0	0	0	0	1	0	1	0	1	1	EVX	evmhesmfa
evmhesmfaaw	31 (0x1F)						rD					rA					rB					1	0	1	0	0	0	0	1	0	1	1	EVX	evmhesmfaaw
evmhesmfanw	31 (0x1F)						rD					rA					rB					1	0	1	1	0	0	0	1	0	1	1	EVX	evmhesmfanw
evmhesmi	31 (0x1F)						rD					rA					rB					1	0	0	0	0	0	0	1	0	0	1	EVX	evmhesmi
evmhesmia	31 (0x1F)						rD					rA					rB					1	0	0	0	0	1	0	1	0	0	1	EVX	evmhesmia
evmhesmiaaw	31 (0x1F)						rD					rA					rB					1	0	1	0	0	0	0	1	0	0	1	EVX	evmhesmiaaw
evmhesmianw	31 (0x1F)						rD					rA					rB					1	0	1	1	0	0	0	1	0	0	1	EVX	evmhesmianw
evmhessf	31 (0x1F)						rD					rA					rB					1	0	0	0	0	0	0	0	0	1	1	EVX	evmhessf
evmhessfa	31 (0x1F)						rD					rA					rB					1	0	0	0	0	1	0	0	0	1	1	EVX	evmhessfa
evmhessfaaw	31 (0x1F)						rD					rA					rB					1	0	1	0	0	0	0	0	0	1	1	EVX	evmhessfaaw
evmhessfanw	31 (0x1F)						rD					rA					rB					1	0	1	1	0	0	0	0	0	1	1	EVX	evmhessfanw
evmhessiaaw	31 (0x1F)						rD					rA					rB					1	0	1	0	0	0	0	0	0	0	1	EVX	evmhessiaaw
evmhessianw	31 (0x1F)						rD					rA					rB					1	0	1	1	0	0	0	0	0	0	1	EVX	evmhessianw
evmheumi	31 (0x1F)						rD					rA					rB					1	0	0	0	0	0	0	1	0	0	0	EVX	evmheumi
evmheumia	31 (0x1F)						rD					rA					rB					1	0	0	0	0	1	0	1	0	0	0	EVX	evmheumia
evmheumiaaw	31 (0x1F)						rD					rA					rB					1	0	1	0	0	0	0	1	0	0	0	EVX	evmheumiaaw
evmheumianw	31 (0x1F)						rD					rA					rB					1	0	1	1	0	0	0	1	0	0	0	EVX	evmheumianw
evmheusiaaw	31 (0x1F)						rD					rA					rB					1	0	1	0	0	0	0	0	0	0	0	EVX	evmheusiaaw
evmheusianw	31 (0x1F)						rD					rA					rB					1	0	1	1	0	0	0	0	0	0	0	EVX	evmheusianw
evmhogsmfaa	31 (0x1F)						rD					rA					rB					1	0	1	0	0	1	0	1	1	1	1	EVX	evmhogsmfaa
evmhogsmfan	31 (0x1F)						rD					rA					rB					1	0	1	1	0	1	0	1	1	1	1	EVX	evmhogsmfan
evmhogsmiaa	31 (0x1F)						rD					rA					rB					1	0	1	0	0	1	0	1	1	0	1	EVX	evmhogsmiaa
evmhogsmian	31 (0x1F)						rD					rA					rB					1	0	1	1	0	1	0	1	1	0	1	EVX	evmhogsmian
evmhogumiaa	31 (0x1F)						rD					rA					rB					1	0	1	0	0	1	0	1	1	0	0	EVX	evmhogumiaa

Table 269. Instructions sorted by mnemonic (decimal and hexadecimal) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
evmhogumian	31 (0x1F)						rD					rA					rB					1	0	1	1	0	1	0	1	1	0	0	EVX	**evmhogumian**
evmhosmf	31 (0x1F)						rD					rA					rB					1	0	0	0	0	0	0	1	1	1	1	EVX	**evmhosmf**
evmhosmfa	31 (0x1F)						rD					rA					rB					1	0	0	0	0	1	0	1	1	1	1	EVX	**evmhosmfa**
evmhosmfaaw	31 (0x1F)						rD					rA					rB					1	0	1	0	0	0	0	1	1	1	1	EVX	**evmhosmfaaw**
evmhosmfanw	31 (0x1F)						rD					rA					rB					1	0	1	1	0	0	0	1	1	1	1	EVX	**evmhosmfanw**
evmhosmi	31 (0x1F)						rD					rA					rB					1	0	0	0	0	0	0	1	1	0	1	EVX	**evmhosmi**
evmhosmia	31 (0x1F)						rD					rA					rB					1	0	0	0	0	1	0	1	1	0	1	EVX	**evmhosmia**
evmhosmiaaw	31 (0x1F)						rD					rA					rB					1	0	1	0	0	0	0	1	1	0	1	EVX	**evmhosmiaaw**
evmhosmianw	31 (0x1F)						rD					rA					rB					1	0	1	1	0	0	0	1	1	0	1	EVX	**evmhosmianw**
evmhossf	31 (0x1F)						rD					rA					rB					1	0	0	0	0	0	0	0	1	1	1	EVX	**evmhossf**
evmhossfa	31 (0x1F)						rD					rA					rB					1	0	0	0	0	1	0	0	1	1	1	EVX	**evmhossfa**
evmhossfaaw	31 (0x1F)						rD					rA					rB					1	0	1	0	0	0	0	0	1	1	1	EVX	**evmhossfaaw**
evmhossfanw	31 (0x1F)						rD					rA					rB					1	0	1	1	0	0	0	0	1	1	1	EVX	**evmhossfanw**
evmhossiaaw	31 (0x1F)						rD					rA					rB					1	0	1	0	0	0	0	0	1	0	1	EVX	**evmhossiaaw**
evmhossianw	31 (0x1F)						rD					rA					rB					1	0	1	1	0	0	0	0	1	0	1	EVX	**evmhossianw**
evmhoumi	31 (0x1F)						rD					rA					rB					1	0	0	0	0	0	0	1	1	0	0	EVX	**evmhoumi**
evmhoumia	31 (0x1F)						rD					rA					rB					1	0	0	0	0	1	0	1	1	0	0	EVX	**evmhoumia**
evmhoumiaaw	31 (0x1F)						rD					rA					rB					1	0	1	0	0	0	0	1	1	0	0	EVX	**evmhoumiaaw**
evmhoumianw	31 (0x1F)						rD					rA					rB					1	0	1	1	0	0	0	1	1	0	0	EVX	**evmhoumianw**
evmhousiaaw	31 (0x1F)						rD					rA					rB					1	0	1	0	0	0	0	0	1	0	0	EVX	**evmhousiaaw**
evmhousianw	31 (0x1F)						rD					rA					rB					1	0	1	1	0	0	0	0	1	0	0	EVX	**evmhousianw**
evmr	**evmr rD,rA** equivalent to **evor rD,rA,rA**																																	**evmr**
evmra	31 (0x1F)						rD					rA					///					1	0	0	1	1	0	0	0	1	0	0	EVX	**evmra**
evmwhgsmfaa	31 (0x1F)						rD					rA					rB					1	0	1	0	1	1	0	1	1	1	1	EVX	**evmwhgsmfaa**
evmwhgsmfan	31 (0x1F)						rD					rA					rB					1	0	1	1	1	0	1	1	1	1	1	EVX	**evmwhgsmfan**
evmwhgsmiaa	31 (0x1F)						rD					rA					rB					1	0	1	0	1	1	0	1	1	0	1	EVX	**evmwhgsmiaa**

Table 269. Instructions sorted by mnemonic (decimal and hexadecimal) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
evmwhgsmian	31 (0x1F)						rD					rA					rB					1	0	1	1	1	0	1	1	1	0	1	EVX	evmwhgsmian
evmwhgssfaa	31 (0x1F)						rD					rA					rB					1	0	1	0	1	1	0	0	1	1	1	EVX	evmwhgssfaa
evmwhgssfan	31 (0x1F)						rD					rA					rB					1	0	1	1	1	0	1	0	1	1	1	EVX	evmwhgssfan
evmwhgumiaa	31 (0x1F)						rD					rA					rB					1	0	1	0	1	1	0	1	1	0	0	EVX	evmwhgumiaa
evmwhgumian	31 (0x1F)						rD					rA					rB					1	0	1	1	1	0	1	1	1	0	0	EVX	evmwhgumian
evmwhsmf	31 (0x1F)						rD					rA					rB					1	0	0	0	1	0	0	1	1	1	1	EVX	evmwhsmf
evmwhsmfa	31 (0x1F)						rD					rA					rB					1	0	0	0	1	1	0	1	1	1	1	EVX	evmwhsmfa
evmwhsmfaaw	31 (0x1F)						rD					rA					rB					1	0	1	0	1	0	0	1	1	1	1	EVX	evmwhsmfaaw
evmwhsmfanw	31 (0x1F)						rD					rA					rB					1	0	1	1	1	0	0	1	1	1	1	EVX	evmwhsmfanw
evmwhsmi	31 (0x1F)						rD					rA					rB					1	0	0	0	1	0	0	1	1	0	1	EVX	evmwhsmi
evmwhsmia	31 (0x1F)						rD					rA					rB					1	0	0	0	1	1	0	1	1	0	1	EVX	evmwhsmia
evmwhsmiaaw	31 (0x1F)						rD					rA					rB					1	0	1	0	1	0	0	1	1	0	1	EVX	evmwhsmiaaw
evmwhsmianw	31 (0x1F)						rD					rA					rB					1	0	1	1	1	0	0	1	1	0	1	EVX	evmwhsmianw
evmwhssf	31 (0x1F)						rD					rA					rB					1	0	0	0	1	0	0	0	1	1	1	EVX	evmwhssf
evmwhssfa	31 (0x1F)						rD					rA					rB					1	0	0	0	1	1	0	0	1	1	1	EVX	evmwhssfa
evmwhssfaaw	31 (0x1F)						rD					rA					rB					1	0	1	0	1	0	0	0	1	1	1	EVX	evmwhssfaaw
evmwhssfanw	31 (0x1F)						rD					rA					rB					1	0	1	1	1	0	0	0	1	1	1	EVX	evmwhssfanw
evmwhssianw	31 (0x1F)						rD					rA					rB					1	0	1	1	1	0	0	0	1	0	1	EVX	evmwhssianw
evmwhssmaaw	31 (0x1F)						rD					rA					rB					1	0	1	0	1	0	0	0	1	0	1	EVX	evmwhssmaaw
evmwhumi	31 (0x1F)						rD					rA					rB					1	0	0	0	1	0	0	1	1	0	0	EVX	evmwhumi
evmwhumia	31 (0x1F)						rD					rA					rB					1	0	0	0	1	1	0	1	1	0	0	EVX	evmwhumia
evmwhusiaaw	31 (0x1F)						rD					rA					rB					1	0	1	0	1	0	0	0	1	0	0	EVX	evmwhusiaaw
evmwhusianw	31 (0x1F)						rD					rA					rB					1	0	1	1	1	0	0	0	1	0	0	EVX	evmwhusianw
evmwlsmf	31 (0x1F)						rD					rA					rB					1	0	0	0	1	0	0	1	0	1	1	EVX	evmwlsmf
evmwlsmfa	31 (0x1F)						rD					rA					rB					1	0	0	0	1	1	0	1	0	1	1	EVX	evmwlsmfa
evmwlsmfaaw	31 (0x1F)						rD					rA					rB					1	0	1	0	1	0	0	1	0	1	1	EVX	evmwlsmfaaw

Table 269. Instructions sorted by mnemonic (decimal and hexadecimal) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
evmwlsmfanw	31 (0x1F)						rD					rA					rB					1	0	1	1	1	0	0	1	0	1	1	EVX	evmwlsmfanw
evmwlsmiaaw	31 (0x1F)						rD					rA					rB					1	0	1	0	1	0	0	1	0	0	1	EVX	evmwlsmiaaw
evmwlsmianw	31 (0x1F)						rD					rA					rB					1	0	1	1	1	0	0	1	0	0	1	EVX	evmwlsmianw
evmwlssf	31 (0x1F)						rD					rA					rB					1	0	0	0	1	0	0	0	0	1	1	EVX	evmwlssf
evmwlssfa	31 (0x1F)						rD					rA					rB					1	0	0	0	1	1	0	0	0	1	1	EVX	evmwlssfa
evmwlssfaaw	31 (0x1F)						rD					rA					rB					1	0	1	0	1	0	0	0	0	1	1	EVX	evmwlssfaaw
evmwlssfanw	31 (0x1F)						rD					rA					rB					1	0	1	1	1	0	0	0	0	1	1	EVX	evmwlssfanw
evmwlssiaaw	31 (0x1F)						rD					rA					rB					1	0	1	0	1	0	0	0	0	0	1	EVX	evmwlssiaaw
evmwlssianw	31 (0x1F)						rD					rA					rB					1	0	1	1	1	0	0	0	0	0	1	EVX	evmwlssianw
evmwlumi	31 (0x1F)						rD					rA					rB					1	0	0	0	1	0	0	1	0	0	0	EVX	evmwlumi
evmwlumia	31 (0x1F)						rD					rA					rB					1	0	0	0	1	1	0	1	0	0	0	EVX	evmwlumia
evmwlumiaaw	31 (0x1F)						rD					rA					rB					1	0	1	0	1	0	0	1	0	0	0	EVX	evmwlumiaaw
evmwlumianw	31 (0x1F)						rD					rA					rB					1	0	1	1	1	0	0	1	0	0	0	EVX	evmwlumianw
evmwlusiaaw	31 (0x1F)						rD					rA					rB					1	0	1	0	1	0	0	0	0	0	0	EVX	evmwlusiaaw
evmwlusianw	31 (0x1F)						rD					rA					rB					1	0	1	1	1	0	0	0	0	0	0	EVX	evmwlusianw
evmwsmf	31 (0x1F)						rD					rA					rB					1	0	0	0	1	0	1	1	0	1	1	EVX	evmwsmf
evmwsmfa	31 (0x1F)						rD					rA					rB					1	0	0	0	1	1	1	1	0	1	1	EVX	evmwsmfa
evmwsmfaa	31 (0x1F)						rD					rA					rB					1	0	1	0	1	0	1	1	0	1	1	EVX	evmwsmfaa
evmwsmfan	31 (0x1F)						rD					rA					rB					1	0	1	1	1	0	1	1	0	1	1	EVX	evmwsmfan
evmwsmi	31 (0x1F)						rD					rA					rB					1	0	0	0	1	0	1	1	0	0	1	EVX	evmwsmi
evmwsmia	31 (0x1F)						rD					rA					rB					1	0	0	0	1	1	1	1	0	0	1	EVX	evmwsmia
evmwsmiaa	31 (0x1F)						rD					rA					rB					1	0	1	0	1	0	1	1	0	0	1	EVX	evmwsmiaa
evmwsmian	31 (0x1F)						rD					rA					rB					1	0	1	1	1	0	1	1	0	0	1	EVX	evmwsmian
evmwssf	31 (0x1F)						rD					rA					rB					1	0	0	0	1	0	1	0	0	1	1	EVX	evmwssf
evmwssfa	31 (0x1F)						rD					rA					rB					1	0	0	0	1	1	1	0	0	1	1	EVX	evmwssfa
evmwssfaa	31 (0x1F)						rD					rA					rB					1	0	1	0	1	0	1	0	0	1	1	EVX	evmwssfaa

Table 269. Instructions sorted by mnemonic (decimal and hexadecimal) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
evmwssfan	31 (0x1F)						rD					rA					rB					1	0	1	1	1	0	1	0	0	1	1	EVX	**evmwssfan**
evmwumi	31 (0x1F)						rD					rA					rB					1	0	0	0	1	0	1	1	0	0	0	EVX	**evmwumi**
evmwumia	31 (0x1F)						rD					rA					rB					1	0	0	0	1	1	1	1	0	0	0	EVX	**evmwumia**
evmwumiaa	31 (0x1F)						rD					rA					rB					1	0	1	0	1	0	1	1	0	0	0	EVX	**evmwumiaa**
evmwumian	31 (0x1F)						rD					rA					rB					1	0	1	1	1	0	1	1	0	0	0	EVX	**evmwumian**
evnand	31 (0x1F)						rD					rA					rB					0	1	0	0	0	0	1	1	1	1	0	EVX	**evnand**
evneg	31 (0x1F)						rD					rA					///					0	1	0	0	0	0	0	1	0	0	1	EVX	**evneg**
evnor	31 (0x1F)						rD					rA					rB					0	1	0	0	0	0	1	1	0	0	0	EVX	**evnor**
evnot	**evnot rD,rA**						equivalent to					**evnor rD,rA,rA**																							**evnot**
evor	31 (0x1F)						rD					rA					rB					0	1	0	0	0	0	1	0	1	1	1	EVX	**evor**
evorc	31 (0x1F)						rD					rA					rB					0	1	0	0	0	0	1	1	0	1	1	EVX	**evorc**
evrlw	31 (0x1F)						rD					rA					rB					0	1	0	0	0	1	0	1	0	0	0	EVX	**evrlw**
evrlwi	31 (0x1F)						rD					rA					UIMM					0	1	0	0	0	1	0	1	0	1	0	EVX	**evrlwi**
evrndw	31 (0x1F)						rD					rA					UIMM					0	1	0	0	0	0	0	1	1	0	0	EVX	**evrndw**
evsel	31 (0x1F)						rD					rA					rB					0	1	0	0	1	1	1	1	crfS			EVX	**evsel**
evslw	31 (0x1F)						rD					rA					rB					0	1	0	0	0	1	0	0	1	0	0	EVX	**evslw**
evslwi	31 (0x1F)						rD					rA					UIMM					0	1	0	0	0	1	0	0	1	1	0	EVX	**evslwi**
evsplatfi	31 (0x1F)						rD					SIMM					///					0	1	0	0	0	1	0	1	0	1	1	EVX	**evsplatfi**
evsplati	31 (0x1F)						rD					SIMM					///					0	1	0	0	0	1	0	1	0	0	1	EVX	**evsplati**
evsrwis	31 (0x1F)						rD					rA					UIMM					0	1	0	0	0	1	0	0	0	1	1	EVX	**evsrwis**
evsrwiu	31 (0x1F)						rD					rA					UIMM					0	1	0	0	0	1	0	0	0	1	0	EVX	**evsrwiu**
evsrws	31 (0x1F)						rD					rA					rB					0	1	0	0	0	1	0	0	0	0	1	EVX	**evsrws**
evsrwu	31 (0x1F)						rD					rA					rB					0	1	0	0	0	1	0	0	0	0	0	EVX	**evsrwu**
evstdd	31 (0x1F)						rD					rA					UIMM [8]					0	1	1	0	0	1	0	0	0	0	1	EVX	**evstdd**
evstddx	31 (0x1F)						rS					rA					rB					0	1	1	0	0	1	0	0	0	0	0	EVX	**evstddx**
evstdh	31 (0x1F)						rS					rA					UIMM [8]					0	1	1	0	0	1	0	0	1	0	1	EVX	**evstdh**
evstdhx	31 (0x1F)						rS					rA					rB					0	1	1	0	0	1	0	0	1	0	0	EVX	**evstdhx**
evstdw	31 (0x1F)						rS					rA					UIMM [8]					0	1	1	0	0	1	0	0	0	1	1	EVX	**evstdw**
evstdwx	31 (0x1F)						rS					rA					rB					0	1	1	0	0	1	0	0	0	1	0	EVX	**evstdwx**
evstwhe	31 (0x1F)						rS					rA					UIMM [10]					0	1	1	0	0	1	1	0	0	0	1	EVX	**evstwhe**
evstwhex	31 (0x1F)						rS					rA					rB					0	1	1	0	0	1	1	0	0	0	0	EVX	**evstwhex**
evstwho	31 (0x1F)						rS					rA					UIMM [10]					0	1	1	0	0	1	1	0	1	0	1	EVX	**evstwho**
evstwhox	31 (0x1F)						rS					rA					rB					0	1	1	0	0	1	1	0	1	0	0	EVX	**evstwhox**
evstwwe	31 (0x1F)						rS					rA					UIMM [10]					0	1	1	0	0	1	1	1	0	0	1	EVX	**evstwwe**
evstwwex	31 (0x1F)						rS					rA					rB					0	1	1	0	0	1	1	1	0	0	0	EVX	**evstwwex**
evstwwo	31 (0x1F)						rS					rA					UIMM [10]					0	1	1	0	0	1	1	1	1	0	1	EVX	**evstwwo**
evstwwox	31 (0x1F)						rS					rA					rB					0	1	1	0	0	1	1	1	1	0	0	EVX	**evstwwox**

Table 269. Instructions sorted by mnemonic (decimal and hexadecimal) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
evsubfsmiaaw	31 (0x1F)						rD					rA					///					1	0	0	1	1	0	0	1	0	1	1	EVX	**evsubfsmiaaw**
evsubfssiaaw	31 (0x1F)						rD					rA					///					1	0	0	1	1	0	0	0	0	1	1	EVX	**evsubfssiaaw**
evsubfumiaaw	31 (0x1F)						rD					rA					///					1	0	0	1	1	0	0	1	0	1	0	EVX	**evsubfumiaaw**
evsubfusiaaw	31 (0x1F)						rD					rA					///					1	0	0	1	1	0	0	0	0	1	0	EVX	**evsubfusiaaw**
evsubfw	31 (0x1F)						rD					rA					rB					0	1	0	0	0	0	0	0	1	0	0	EVX	**evsubfw**
evsubifw	31 (0x1F)						rD					UIMM					rB					0	1	0	0	0	0	0	0	1	1	0	EVX	**evsubifw**
evsubiw	**evsubiw rD,rB,UIMM** equivalent to **evsubifw rD,UIMM,rB**																																	**evsubiw**
evsubw	**evsubw rD,rB,rA** equivalent to **evsubfw rD,rA,rB**																																	**evsubw**
evxor	31 (0x1F)						rD					rA					rB					0	1	0	0	0	0	1	0	1	1	0	EVX	**evxor**
extlwi	**extlwi rA,rS,*n*,*b*** (n > 0) equivalent to **rlwinm rA,rS,*b*,0,*n* – 1**																																	**extlwi**
extrwi	**extrwi rA,rS,*n*,*b* (*n* > 0)** equivalent to **rlwinm rA,rS,*b* + *n*, 32 – *n*,31**																																	**extrwi**
extsb	31 (0x1F)						rS					rA					///					1	1	1	0	1	1	1	0	1	0	0	X	**extsb**
extsb.	31 (0x1F)						rS					rA					///					1	1	1	0	1	1	1	0	1	0	1	X	**extsb.**
extsh	31 (0x1F)						rS					rA					///					1	1	1	0	0	1	1	0	1	0	0	X	**extsh**
extsh.	31 (0x1F)						rS					rA					///					1	1	1	0	0	1	1	0	1	0	1	X	**extsh.**
fres [6]	59(0x3B)						frD					///					frB					///					1	1	0	0	0	0	A	**fres**
fres. [6]	59(0x3B)						frD					///					frB					///					1	1	0	0	0	1	A	**fres.**
fsel [6]	63(0x3F)						frD					frA					frB					frC					1	0	1	1	1	0	A	**fsel**
fsel. [6]	63(0x3F)						frD					frA					frB					frC					1	0	1	1	1	1	A	**fsel.**
icbi	31 (0x1F)						///					rA					rB					1	1	1	1	0	1	0	1	1	0	/	X	**icbi**
icblc	31 (0x1F)						CT					rA					rB					0	0	1	1	1	0	0	1	1	0	0	X	**icblc**
icbt	31 (0x1F)						CT					rA					rB					0	0	0	0	0	1	0	1	1	0	/	X	**icbt**
icbtls	31 (0x1F)						CT					rA					rB					0	1	1	1	1	0	0	1	1	0	0	X	**icbtls**
inslwi	**inslwi rA,rS,*n*,*b* (*n* > 0)** equivalent to **rlwimi rA,rS,32 – *b*,*b*,(*b* + *n*) – 1**																																	**inslwi**
insrwi	**insrwi rA,rS,*n*,*b* (*n* > 0)** equivalent to **rlwimi rA,rS,32 – (*b* + *n*),*b*,(*b* + *n*) – 1**																																	**insrwi**
isel	31 (0x1F)						rD					rA					rB					crb					0	1	1	1	1	0	X	**isel**
iseleq	**iseleq rD,rA,rB** equivalent to **isel rD,rA,rB,2**																																	**iseleq**
iselgt	**iselgt rD,rA,rB** equivalent to **isel rD,rA,rB,1**																																	**iselgt**
iselit	**iselit rD,rA,rB** equivalent to **isel rD,rA,rB,0**																																	**iselit**
isync	19 (0x13)						///															0	0	1	0	0	1	0	1	1	0	/	XL	**isync**
la	**la rD,d(rA)** equivalent to **addi rD,rA,d**																																	**la**
lbz	34(0x22)						rD					rA					D																D	**lbz**
lbzu	35(0x23)						rD					rA					D																D	**lbzu**
lbzux	31 (0x1F)						rD					rA					rB					0	0	0	1	1	1	0	1	1	1	/	X	**lbzux**
lbzx	31 (0x1F)						rD					rA					rB					0	0	0	1	0	1	0	1	1	1	/	X	**lbzx**
lha	42(0x2A)						rD					rA					D																D	**lha**
lhau	43(0x2B)						rD					rA					D																D	**lhau**

Table 269. Instructions sorted by mnemonic (decimal and hexadecimal) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
lhaux	31 (0x1F)						rD					rA					rB					0	1	0	1	1	1	0	1	1	1	/	X	**lhaux**
lhax	31 (0x1F)						rD					rA					rB					0	1	0	1	0	1	0	1	1	1	/	X	**lhax**
lhbrx	31 (0x1F)						rD					rA					rB					1	1	0	0	0	1	0	1	1	0	/	X	**lhbrx**
lhz	40(0x28)						rD					rA					D															D	**lhz**
lhzu	41(0x29)						rD					rA					D															D	**lhzu**
lhzux	31 (0x1F)						rD					rA					rB					0	1	0	0	1	1	0	1	1	1	/	X	**lhzux**
lhzx	31 (0x1F)						rD					rA					rB					0	1	0	0	0	1	0	1	1	1	/	X	**lhzx**
li	**li rD,value**						equivalent to					**addi rD,0,**value																						**li**
lis	**lis rD,value**						equivalent to					**addis rD,0,**value																						**lis**
lmw	46(0x2E)						rD					rA					D															D	**lmw**
lwarx	31 (0x1F)						rD					rA					rB					0	0	0	0	0	1	0	1	0	0	/	X	**lwarx**
lwbrx	31 (0x1F)						rD					rA					rB					1	0	0	0	0	1	0	1	1	0	/	X	**lwbrx**
lwz	32 (0x20)						rD					rA					D															D	**lwz**
lwzu	33 (0x21)						rD					rA					D															D	**lwzu**
lwzux	31 (0x1F)						rD					rA					rB					0	0	0	0	1	1	0	1	1	1	/	X	**lwzux**
lwzx	31 (0x1F)						rD					rA					rB					0	0	0	0	0	1	0	1	1	1	/	X	**lwzx**
mbar	31 (0x1F)						MO					///										1	1	0	1	0	1	0	1	1	0	/	X	mbar
mcrf	19 (0x13)						crfD			//		crfS					///					0	0	0	0	0	0	0	0	0	0	/	XL	**mcrf**
mcrxr	31 (0x1F)						crfD					///										1	0	0	0	0	0	0	0	0	0	/	X	**mcrxr**
mfcr	**mtcr rS**						equivalent to					**mtcrf 0xFF,rS**																						**mfcr**
mfcr	31 (0x1F)						rD					///										0	0	0	0	0	1	0	0	1	1	/	X	**mfcr**
mfdcr	31 (0x1F)						rD					DCRN5–9					DCRN0–4					0	1	0	1	0	0	0	0	1	1	/	XFX	mfdcr
mfmsr [7]	31 (0x1F)						rD					///										0	0	0	1	0	1	0	0	1	1	/	X	**mfmsr**
mfpmr	31 (0x1F)						rD					PMRN5–9					PMRN0–4					0	1	0	1	0	0	1	1	1	0	0	XFX	mfpmr
mf*regname*	**mf*regname* rD**						equivalent to					**mfspr rD,SPR*n***																						**mf*regname***
mfspr [8]	31 (0x1F)						rD					SPR[5–9]					SPR[0–4]					0	1	0	1	0	1	0	0	1	1	/	XFX	**mfspr**
mr	**mr rA,rS**						equivalent to					**or rA,rS,rS**																						**mr**
msync	31 (0x1F)						///															1	0	0	1	0	1	0	1	1	0	/	X	msync
mtcr	**mtcr rS**						equivalent to					**mtcrf** 0xFF,**rS**																						**mtcr**
mtcrf	31 (0x1F)						rS					/			CRM						/	0	0	1	0	0	1	0	0	0	0	/	XFX	**mtcrf**
mtdcr	31 (0x1F)						rS					DCRN5–9					DCRN0–4					0	1	1	1	0	0	0	0	1	1	/	XFX	mtdcr
mtmsr 7	31 (0x1F)						rS					///										0	0	1	0	0	1	0	0	1	0	/	X	mtmsr
mtpmr	31 (0x1F)						rS					PMRN5–9					PMRN0–4					0	1	1	1	0	0	1	1	1	0	0	XFX	mtpmr
mt*regname*	**mt*regname* rS**						equivalent to					**mtspr SPR*n* rS**																						**mt*regname***
mtspr [8]	31 (0x1F)						rS					SPR[5–9]					SPR[0–4]					0	1	1	1	0	1	0	0	1	1	/	XFX	**mtspr**
mulhw	31 (0x1F)						rD					rA					rB					/	0	0	1	0	0	1	0	1	1	0	X	**mulhw**
mulhw.	31 (0x1F)						rD					rA					rB					/	0	0	1	0	0	1	0	1	1	1	X	**mulhw.**
mulhwu	31 (0x1F)						rD					rA					rB					/	0	0	0	0	0	1	0	1	1	0	X	**mulhwu**
mulhwu.	31 (0x1F)						rD					rA					rB					/	0	0	0	0	0	1	0	1	1	1	X	**mulhwu.**
mulli	07						rD					rA					SIMM																D	**mulli**

Table 269. Instructions sorted by mnemonic (decimal and hexadecimal) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
mullw	31 (0x1F)						rD					rA					rB					0	0	1	1	1	0	1	0	1	1	0	X	**mullw**
mullw.	31 (0x1F)						rD					rA					rB					0	0	1	1	1	0	1	0	1	1	1	X	**mullw.**
mullwo	31 (0x1F)						rD					rA					rB					1	0	1	1	1	0	1	0	1	1	0	X	**mullwo**
mullwo.	31 (0x1F)						rD					rA					rB					1	0	1	1	1	0	1	0	1	1	1	X	**mullwo.**
nand	31 (0x1F)						rS					rA					rB					0	1	1	1	0	1	1	1	0	0	0	X	**nand**
nand.	31 (0x1F)						rS					rA					rB					0	1	1	1	0	1	1	1	0	0	1	X	**nand.**
neg	31 (0x1F)						rD					rA					///					0	0	0	1	1	0	1	0	0	0	0	X	**neg**
neg.	31 (0x1F)						rD					rA					///					0	0	0	1	1	0	1	0	0	0	1	X	**neg.**
nego	31 (0x1F)						rD					rA					///					1	0	0	1	1	0	1	0	0	0	0	X	**nego**
nego.	31 (0x1F)						rD					rA					///					1	0	0	1	1	0	1	0	0	0	1	X	**nego.**
nop	**nop** equivalent to **ori 0,0,0**																																	**nop**
nor	31 (0x1F)						rS					rA					rB					0	0	0	1	1	1	1	1	0	0	0	X	**nor**
nor.	31 (0x1F)						rS					rA					rB					0	0	0	1	1	1	1	1	0	0	1	X	**nor.**
not	**not** r**A**,r**S** equivalent to **nor** r**A**,r**S**,r**S**																																	**not**
or	31 (0x1F)						rS					rA					rB					0	1	1	0	1	1	1	1	0	0	0	X	**or**
or.	31 (0x1F)						rS					rA					rB					0	1	1	0	1	1	1	1	0	0	1	X	**or.**
orc	31 (0x1F)						rS					rA					rB					0	1	1	0	0	1	1	1	0	0	0	X	**orc**
orc.	31 (0x1F)						rS					rA					rB					0	1	1	0	0	1	1	1	0	0	1	X	**orc.**
ori	24 (0x18)						rS					rA					UIMM																D	**ori**
oris	25 (0x19)						rS					rA					UIMM																D	**oris**
rfci	19 (0x13)						///															0	0	0	0	1	1	0	0	1	1	/	XL	**rfci**
rfdi [7]	0	1	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	1	1	0	X	**rfdi**
rfi [7]	19 (0x13)						///															0	0	0	0	1	1	0	0	1	0	/	XL	**rfi**
rfmci [7]	19 (0x13)						///															0	0	0	0	1	0	0	1	1	0	/	XL	**rfmci**
rlwimi	20 (0x14)						rS					rA					SH					MB					ME					0	M	**rlwimi**
rlwimi.	20 (0x14)						rS					rA					SH					MB					ME					1	M	**rlwimi.**
rlwinm	21 (0x15)						rS					rA					SH					MB					ME					0	M	**rlwinm**
rlwinm.	21 (0x15)						rS					rA					SH					MB					ME					1	M	**rlwinm.**
rlwnm	23 (0x17)						rS					rA					rB					MB					ME					0	M	**rlwnm**
rlwnm.	23 (0x17)						rS					rA					rB					MB					ME					1	M	**rlwnm.**
rotlw	**rotlw** r**A**,r**S**,r**B** equivalent to **rlwnm** r**A**,r**S**,r**B**,**0,31**																																	**rotlw**
rotlwi	**rotlwi** r**A**,r**S**,*n* equivalent to **rlwinm** r**A**,r**S**,*n*,**0,31**																																	**rotlwi**
rotrwi	**rotrwi** r**A**,r**S**,*n* equivalent to **rlwinm** r**A**,r**S**,32 – *n*,**0,31**																																	**rotrwi**
sc	17 (0x11)						///																								1	/	SC	**sc**
slw	31 (0x1F)						rS					rA					rB					0	0	0	0	0	1	1	0	0	0	0	X	**slw**
slw.	31 (0x1F)						rS					rA					rB					0	0	0	0	0	1	1	0	0	0	1	X	**slw.**
slwi	**slwi** r**A**,r**S**,*n* (n < 32) equivalent to **rlwinm** r**A**,r**S**,*n*,**0**,31 – *n*																																	**slwi**
sraw	31 (0x1F)						rS					rA					rB					1	1	0	0	0	1	1	0	0	0	0	X	**sraw**
sraw.	31 (0x1F)						rS					rA					rB					1	1	0	0	0	1	1	0	0	0	1	X	**sraw.**
srawi	31 (0x1F)						rS					rA					SH					1	1	0	0	1	1	1	0	0	0	0	X	**srawi**

Table 269. Instructions sorted by mnemonic (decimal and hexadecimal) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
srawi.	31 (0x1F)						rS					rA					SH					1	1	0	0	1	1	1	0	0	0	1	X	**srawi.**
srw	31 (0x1F)						rS					rA					rB					1	0	0	0	0	1	1	0	0	0	0	X	**srw**
srw.	31 (0x1F)						rS					rA					rB					1	0	0	0	0	1	1	0	0	0	1	X	**srw.**
srwi	**srwi rA,rS,***n* (n < 32)equivalent to **rlwinm rA,rS,32 – *n*,*n*,31**																																	**srwi**
stb	38(0x26)						rS					rA					D															D	**stb**
stbu	39(0x27)						rS					rA					D															D	**stbu**
stbux	31 (0x1F)						rS					rA					rB					0	0	1	1	1	1	0	1	1	1	0	X	**stbux**
stbx	31 (0x1F)						rS					rA					rB					0	0	1	1	0	1	0	1	1	1	0	X	**stbx**
sth	44(0x2C)						rS					rA					D															D	**sth**
sthbrx	31 (0x1F)						rS					rA					rB					1	1	1	0	0	1	0	1	1	0	/	X	**sthbrx**
sthu	45(0x2D)						rS					rA					D															D	**sthu**
sthux	31 (0x1F)						rS					rA					rB					0	1	1	0	1	1	0	1	1	1	/	X	**sthux**
sthx	31 (0x1F)						rS					rA					rB					0	1	1	0	0	1	0	1	1	1	/	X	**sthx**
stmw	47(0x2F)						rS					rA					D															D	**stmw**
stw	36(0x24)						rS					rA					D															D	**stw**
stwbrx	31 (0x1F)						rS					rA					rB					1	0	1	0	0	1	0	1	1	0	/	X	**stwbrx**
stwcx.	31 (0x1F)						rS					rA					rB					0	0	1	0	0	1	0	1	1	0	1	X	**stwcx.**
stwu	37(0x25)						rS					rA					D															D	**stwu**
stwux	31 (0x1F)						rS					rA					rB					0	0	1	0	1	1	0	1	1	1	/	D	**stwux**
stwx	31 (0x1F)						rS					rA					rB					0	0	1	0	0	1	0	1	1	1	/	D	**stwx**
sub	**sub rD,rA,rB** equivalent to **subf rD,rB,rA**																																	**sub**
subc	**subc rD,rA,rB** equivalent to **subfc rD,rB,rA**																																	**subc**
subf	31 (0x1F)						rD					rA					rB					0	0	0	0	1	0	1	0	0	0	0	X	**subf**
subf.	31 (0x1F)						rD					rA					rB					0	0	0	0	1	0	1	0	0	0	1	X	**subf.**
subfc	31 (0x1F)						rD					rA					rB					0	0	0	0	0	0	1	0	0	0	0	X	**subfc**
subfc.	31 (0x1F)						rD					rA					rB					0	0	0	0	0	0	1	0	0	0	1	X	**subfc.**
subfco	31 (0x1F)						rD					rA					rB					1	0	0	0	0	0	1	0	0	0	0	X	**subfco**
subfco.	31 (0x1F)						rD					rA					rB					1	0	0	0	0	0	1	0	0	0	1	X	**subfco.**
subfe	31 (0x1F)						rD					rA					rB					0	0	1	0	0	0	1	0	0	0	0	X	**subfe**
subfe.	31 (0x1F)						rD					rA					rB					0	0	1	0	0	0	1	0	0	0	1	X	**subfe.**
subfeo	31 (0x1F)						rD					rA					rB					1	0	1	0	0	0	1	0	0	0	0	X	**subfeo**
subfeo.	31 (0x1F)						rD					rA					rB					1	0	1	0	0	0	1	0	0	0	1	X	**subfeo.**
subfic	08						rD					rA					SIMM															D	**subfic**
subfme	31 (0x1F)						rD					rA					///					0	0	1	1	1	0	1	0	0	0	0	X	**subfme**
subfme.	31 (0x1F)						rD					rA					///					0	0	1	1	1	0	1	0	0	0	1	X	**subfme.**
subfmeo	31 (0x1F)						rD					rA					///					1	0	1	1	1	0	1	0	0	0	0	X	**subfmeo**
subfmeo.	31 (0x1F)						rD					rA					///					1	0	1	1	1	0	1	0	0	0	1	X	**subfmeo.**
subfo	31 (0x1F)						rD					rA					rB					1	0	0	0	1	0	1	0	0	0	0	X	**subfo**
subfo.	31 (0x1F)						rD					rA					rB					1	0	0	0	1	0	1	0	0	0	1	X	**subfo.**
subfze	31 (0x1F)						rD					rA					///					0	0	1	1	0	0	1	0	0	0	0	X	**subfze**

Table 269. Instructions sorted by mnemonic (decimal and hexadecimal) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
subfze.	31 (0x1F)						rD					rA					///					0	0	1	1	0	0	1	0	0	0	1	X	**subfze.**
subfzeo	31 (0x1F)						rD					rA					///					1	0	1	1	0	0	1	0	0	0	0	X	**subfzeo**
subfzeo.	31 (0x1F)						rD					rA					///					1	0	1	1	0	0	1	0	0	0	1	X	**subfzeo.**
subi	**subi** rD,rA,**value** equivalent to **addi** rD,rA,–value																																	**subi**
subic	**subic** rD,rA,**value** equivalent to **addic** rD,rA,–value																																	**subic**
subic.	**subic.** rD,rA,**value** equivalent to **addic.** rD,rA,–value																																	**subic.**
subis	**subis** rD,rA,**value** equivalent to **addis** rD,rA,–value																																	**subis**
tlbie 6,7	31 (0x1F)						///					///					rB					0	1	0	0	1	1	0	0	1	0	0	X	tlbie
tlbivax	31 (0x1F)						///					rA					rB					1	1	0	0	0	1	0	0	1	0	/	X	tlbivax
tlbre	31 (0x1F)						///9															1	1	1	0	1	1	0	0	1	0	/	X	tlbre
tlbsx	31 (0x1F)						///12					rA					rB					1	1	1	0	0	1	0	0	1	0	/12	X	tlbsx
tlbsync 6,7	31 (0x1F)						///					///					///					1	0	0	0	1	1	0	1	1	0	/	X	tlbsync
tlbwe	31 (0x1F)						///12															1	1	1	1	0	1	0	0	1	0	/	X	tlbwe
tw	31 (0x1F)						TO					rA					rB					0	0	0	0	0	0	0	1	0	0	/	X	**tw**
tweq	**tweq** rA,**SIMM** equivalent to **tw 4,**rA,SIMM																																	**tweq**
tweqi	**tweqi** rA,**SIMM** equivalent to **twi 4,**rA,SIMM																																	**tweqi**
twge	**twge** rA,**SIMM** equivalent to **tw 12,**rA,SIMM																																	**twge**
twgei	**twgei** rA,**SIMM** equivalent to **twi 12,**rA,SIMM																																	**twgei**
twgt	**twgt** rA,**SIMM** equivalent to **tw 8,**rA,SIMM																																	**twgt**
twgti	**twgti** rA,**SIMM** equivalent to **twi 8,**rA,SIMM																																	**twgti**
twi	03						TO					rA					SIMM																D	**twi**
twle	**twle** rA,**SIMM** equivalent to **tw 20,**rA,SIMM																																	**twle**
twlei	**twlei** rA,**SIMM** equivalent to **twi 20,**rA,SIMM																																	**twlei**
twlge	**twlge** rA,**SIMM** equivalent to **tw 12,**rA,SIMM																																	**twlge**
twlgei	**twlgei** rA,**SIMM** equivalent to **twi 12,**rA,SIMM																																	**twlgei**
twlgt	**twlgt** rA,**SIMM** equivalent to **tw 1,**rA,SIMM																																	**twlgt**
twlgti	**twlgti** rA,**SIMM** equivalent to **twi 1,**rA,SIMM																																	**twlgti**
twlle	**twlle** rA,**SIMM** equivalent to **tw 6,**rA,SIMM																																	**twlle**
twllei	**twllei** rA,**SIMM** equivalent to **twi 6,**rA,SIMM																																	**twllei**
twllt	**twllt** rA,**SIMM** equivalent to **tw 2,**rA,SIMM																																	**twllt**
twllti	**twllti** rA,**SIMM** equivalent to**twi 2,**rA,SIMM																																	**twllti**
twlng	**twlng** rA,**SIMM** equivalent to **tw 6,**rA,SIMM																																	**twlng**
twlngi	**twlngi** rA,**SIMM** equivalent to **twi 6,**rA,SIMM																																	**twlngi**
twlnl	**twlnl** rA,**SIMM** equivalent to **tw 5,**rA,SIMM																																	**twlnl**
twlnli	**twlnli** rA,**SIMM** equivalent to **twi 5,**rA,SIMM																																	**twlnli**
twlt	**twlt** rA,**SIMM** equivalent to **tw 16,**rA,SIMM																																	**twlt**
twlti	**twlti** rA,**SIMM** equivalent to **twi 16,**rA,SIMM																																	**twlti**
twne	**twne** rA,**SIMM** equivalent to **tw 24,**rA,SIMM																																	**twne**
twnei	**twnei** rA,**SIMM** equivalent to **twi 24,**rA,SIMM																																	**twnei**

Table 269. Instructions sorted by mnemonic (decimal and hexadecimal) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
twng	**twng rA,SIMM**						equivalent to					**tw 20,rA,SIMM**																							**twng**
twngi	**twngi rA,SIMM**						equivalent to					**twi 20,rA,SIMM**																							**twngi**
twnl	**twnl rA,SIMM**						equivalent to					**tw 12,rA,SIMM**																							**twnl**
twnli	**twnli rA,SIMM**						equivalent to					**twi 12,rA,SIMM**																							**twnli**
wait	31 (0x1F)						///															0	0	0	0	1	1	1	1	1	0	/		wait
wrtee	31 (0x1F)						rS					///										0	0	1	0	0	0	0	0	1	1	/	X	wrtee
wrteei	31 (0x1F)						///										E	///				0	0	1	0	1	0	0	0	1	1	/	X	wrteei
xor	31 (0x1F)						rS					rA					rB					0	1	0	0	1	1	1	1	0	0	0	X	**xor**
xor.	31 (0x1F)						rS					rA					rB					0	1	0	0	1	1	1	1	0	0	1	X	**xor.**
xori	26 (0x1A)						rS					rA					UIMM																D	**xori**
xoris	27 (0x1B)						rS					rA					UIMM																D	**xoris**

1. Simplified mnemonics for branch instructions that do not test a CR bit should not specify one; a programming error may occur.
2. The value in the BI operand selects CR*n*[2], the EQ bit.
3. The value in the BI operand selects CR*n*[0], the LT bit.
4. The value in the BI operand selects CR*n*[1], the GT bit.
5. The value in the BI operand selects CR*n*[3], the SO bit.
6. Optional to the PowerPC classic architecture.
7. Supervisor-level instruction
8. Access level is detemined by whether the SPR is defined as a user- or supervisor-level SPR.

A.2 Instructions sorted by primary opcodes (decimal and hexadecimal)

Table 270 lists instructions by their primary (0–5) opcodes in decimal and hexadecimal format.

Table 270. Instructions sorted by primary opcodes (decimal and hexadecimal)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
rfdi [1]	0	1	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	1	1	0	X	**rfdi**
twi	03						TO					rA					SIMM																D	**twi**
brinc	04						rD					rA					rB					0	1	0	0	0	0	0	1	1	1	1	EVX	**brinc**
efdabs	04						rD					rA					///					0	1	0	1	1	1	0	0	1	0	0	EFX	**efdabs**
efdadd	04						rD					rA					rB					0	1	0	1	1	1	0	0	0	0	0	EFX	**efdadd**
efdcfs	04						rD					0	0	0	0	0	rB					0	1	0	1	1	1	0	1	1	1	1	EFX	**efdcfs**
efdcfsf	04						rD					///					rB					0	1	0	1	1	1	1	0	0	1	1	EFX	**efdcfsf**
efdcfsi	04						rD					///					rB					0	1	0	1	1	1	1	0	0	0	1	EFX	**efdcfsi**
efdcfuf	04						rD					///					rB					0	1	0	1	1	1	1	0	0	1	0	EFX	**efdcfuf**
efdcfui	04						rD					///					rB					0	1	0	1	1	1	1	0	0	0	0	EFX	**efdcfui**
efdcmpeq	04						crfD			/	/	rA					rB					0	1	0	1	1	1	0	1	1	1	0	EFX	**efdcmpeq**
efdcmpgt	04						crfD			/	/	rA					rB					0	1	0	1	1	1	0	1	1	0	0	EFX	**efdcmpgt**
efdcmplt	04						crfD			/	/	rA					rB					0	1	0	1	1	1	0	1	1	0	1	EFX	**efdcmplt**

Table 270. Instructions sorted by primary opcodes (decimal and hexadecimal) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
efdctsf	04						rD					///					rB					0	1	0	1	1	1	1	0	1	1	1	EFX	**efdctsf**
efdctsi	04						rD					///					rB					0	1	0	1	1	1	1	0	1	0	1	EFX	**efdctsi**
efdctsiz	04						rD					///					rB					0	1	0	1	1	1	1	1	0	1	0	EFX	**efdctsiz**
efdctuf	04						rD					///					rB					0	1	0	1	1	1	1	0	1	1	0	EFX	**efdctuf**
efdctui	04						rD					///					rB					0	1	0	1	1	1	1	0	1	0	0	EFX	**efdctui**
efdctuiz	04						rD					///					rB					0	1	0	1	1	1	1	1	0	0	0	EFX	**efdctuiz**
efddiv	04						rD					rA					rB					0	1	0	1	1	1	0	1	0	0	1	EFX	**efddiv**
efdmul	04						rD					rA					rB					0	1	0	1	1	1	0	1	0	0	0	EFX	**efdmul**
efdnabs	04						rD					rA					///					0	1	0	1	1	1	0	0	1	0	1	EFX	**efdnabs**
efdneg	04						rD					rA					///					0	1	0	1	1	1	0	0	1	1	0	EFX	**efdneg**
efdsub	04						rD					rA					rB					0	1	0	1	1	1	0	0	0	0	1	EFX	**efdsub**
efdtsteq	04						crfD			/	/	rA					rB					0	1	0	1	1	1	1	1	1	1	0	EFX	**efdtsteq**
efdtstgt	04						crfD			/	/	rA					rB					0	1	0	1	1	1	1	1	1	0	0	EFX	**efdtstgt**
efdtstlt	04						crfD			/	/	rA					rB					0	1	0	1	1	1	1	1	1	0	1	EFX	**efdtstlt**
efsabs	04						rD					rA					///					0	1	0	1	1	0	0	0	1	0	0	EFX	**efsabs**
efsadd	04						rD					rA					rB					0	1	0	1	1	0	0	0	0	0	0	EFX	**efsadd**
efscfd	04						rD					0	0	0	0	0	rB					0	1	0	1	1	0	0	1	1	1	1	EFX	**efscfd**
efscfsf	04						rD					///					rB					0	1	0	1	1	0	1	0	0	1	1	EFX	**efscfsf**
efscfsi	04						rD					///					rB					0	1	0	1	1	0	1	0	0	0	1	EFX	**efscfsi**
efscfuf	04						rD					///					rB					0	1	0	1	1	0	1	0	0	1	0	EFX	**efscfuf**
efscfui	04						rD					///					rB					0	1	0	1	1	0	1	0	0	0	0	EFX	**efscfui**
efscmpeq	04						crfD			/	/	rA					rB					0	1	0	1	1	0	0	1	1	1	0	EFX	**efscmpeq**
efscmpgt	04						crfD			/	/	rA					rB					0	1	0	1	1	0	0	1	1	0	0	EFX	**efscmpgt**
efscmplt	04						crfD			/	/	rA					rB					0	1	0	1	1	0	0	1	1	0	1	EFX	**efscmplt**
efsctsf	04						rD					///					rB					0	1	0	1	1	0	1	0	1	1	1	EFX	**efsctsf**
efsctsi	04						rD					///					rB					0	1	0	1	1	0	1	0	1	0	1	EFX	**efsctsi**
efsctsiz	04						rD					///					rB					0	1	0	1	1	0	1	1	0	1	0	EFX	**efsctsiz**
efsctuf	04						rD					///					rB					0	1	0	1	1	0	1	0	1	1	0	EFX	**efsctuf**
efsctui	04						rD					///					rB					0	1	0	1	1	0	1	0	1	0	0	EFX	**efsctui**
efsctuiz	04						rD					///					rB					0	1	0	1	1	0	1	1	0	0	0	EFX	**efsctuiz**
efsdiv	04						rD					rA					rB					0	1	0	1	1	0	0	1	0	0	1	EFX	**efsdiv**
efsmul	04						rD					rA					rB					0	1	0	1	1	0	0	1	0	0	0	EFX	**efsmul**
efsnabs	04						rD					rA					///					0	1	0	1	1	0	0	0	1	0	1	EFX	**efsnabs**
efsneg	04						rD					rA					///					0	1	0	1	1	0	0	0	1	1	0	EFX	**efsneg**
efssub	04						rD					rA					rB					0	1	0	1	1	0	0	0	0	0	1	EFX	**efssub**
efststeq	04						crfD			/	/	rA					rB					0	1	0	1	1	0	1	1	1	1	0	EFX	**efststeq**
efststgt	04						crfD			/	/	rA					rB					0	1	0	1	1	0	1	1	1	0	0	EFX	**efststgt**
efststlt	04						crfD			/	/	rA					rB					0	1	0	1	1	0	1	1	1	0	1	EFX	**efststlt**
mulli	07						rD					rA					SIMM																D	**mulli**
subfic	08						rD					rA					SIMM																D	**subfic**

Table 270. Instructions sorted by primary opcodes (decimal and hexadecimal) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
cmpli	10 (0x0A)						crfD			/	L	rA					UIMM																D	**cmpli**
cmpi	11 (0x0B)						crfD			/	L	rA					SIMM																D	**cmpi**
addic	12 (0x0C)						rD					rA					SIMM																D	**addic**
addic.	13 (0x0D)						rD					rA					SIMM																D	**addic.**
addi	14 (0x0E)						rD					rA					SIMM																D	**addi**
addis	15 (0x0F)						rD					rA					SIMM																D	**addis**
bc	16 (0x10)						BO					BI					BD														0	0	B	**bc**
bca	16 (0x10)						BO					BI					BD														1	0	B	**bca**
bcl	16 (0x10)						BO					BI					BD														0	1	B	**bcl**
bcla	16 (0x10)						BO					BI					BD														1	1	B	**bcla**
sc	17 (0x11)						///																								1	/	SC	**sc**
b	18 (0x12)						LI																								0	0	I	**b**
ba	18 (0x12)						LI																								1	0	I	**ba**
bl	18 (0x12)						LI																								0	1	I	**bl**
bla	18 (0x12)						LI																								1	1	I	**bla**
rfci	19 (0x13)						///															0	0	0	0	1	1	0	0	1	1	/	XL	**rfci**
rfmci [1]	19 (0x13)						///															0	0	0	0	1	0	0	1	1	0	/	XL	**rfmci**
mcrf	19 (0x13)						crfD			//		crfS					///					0	0	0	0	0	0	0	0	0	0	/	XL	**mcrf**
bclr	19 (0x13)						BO					BI					///					0	0	0	0	0	1	0	0	0	0	0	XL	**bclr**
bclrl	19 (0x13)						BO					BI					///					0	0	0	0	0	1	0	0	0	0	1	XL	**bclrl**
crnor	19 (0x13)						crbD					crbA					crbB					0	0	0	0	1	0	0	0	0	1	/	XL	**crnor**
rfi [(1)]	19 (0x13)						///															0	0	0	0	1	1	0	0	1	0	/	XL	**rfi**
crandc	19 (0x13)						crbD					crbA					crbB					0	0	1	0	0	0	0	0	0	1	/	XL	**crandc**
isync	19 (0x13)						///															0	0	1	0	0	1	0	1	1	0	/	XL	**isync**
crxor	19 (0x13)						crbD					crbA					crbB					0	0	1	1	0	0	0	0	0	1	/	XL	**crxor**
crand	19 (0x13)						crbD					crbA					crbB					0	1	0	0	0	0	0	0	0	1	/	XL	**crand**
crnand	19 (0x13)						crbD					crbA					crbB					0	0	1	1	1	0	0	0	0	1	/	XL	**crnand**
creqv	19 (0x13)						crbD					crbA					crbB					0	1	0	0	1	0	0	0	0	1	/	XL	**creqv**
crorc	19 (0x13)						crbD					crbA					crbB					0	1	1	0	1	0	0	0	0	1	/	XL	**crorc**
cror	19 (0x13)						crbD					crbA					crbB					0	1	1	1	0	0	0	0	0	1	/	XL	**cror**
bcctr	19 (0x13)						BO					BI					///					1	0	0	0	0	1	0	0	0	0	0	XL	**bcctr**
bcctrl	19 (0x13)						BO					BI					///					1	0	0	0	0	1	0	0	0	0	1	XL	**bcctrl**
rlwimi	20 (0x14)						rS					rA					SH					MB					ME					0	M	**rlwimi**
rlwimi.	20 (0x14)						rS					rA					SH					MB					ME					1	M	**rlwimi.**
rlwinm	21 (0x15)						rS					rA					SH					MB					ME					0	M	**rlwinm**
rlwinm.	21 (0x15)						rS					rA					SH					MB					ME					1	M	**rlwinm.**
rlwnm	23 (0x17)						rS					rA					rB					MB					ME					0	M	**rlwnm**
rlwnm.	23 (0x17)						rS					rA					rB					MB					ME					1	M	**rlwnm.**
ori	24 (0x18)						rS					rA					UIMM																D	**ori**
oris	25 (0x19)						rS					rA					UIMM																D	**oris**

Table 270. Instructions sorted by primary opcodes (decimal and hexadecimal) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
xori	26 (0x1A)						rS					rA					UIMM																D	**xori**
xoris	27 (0x1B)						rS					rA					UIMM																D	**xoris**
andi.	28 (0x1C)						rS					rA					UIMM																D	**andi.**
andis.	29 (0x1D)						rS					rA					UIMM																D	**andis.**
dcblc	31 (0x1F)						CT					rA					rB					0	1	1	0	0	0	0	1	1	0	0	X	**dcblc**
dcbtls	31 (0x1F)						CT					rA					rB					0	0	1	0	1	0	0	1	1	0	0	X	**dcbtls**
dcbtstls	31 (0x1F)						CT					rA					rB					0	0	1	0	0	0	0	1	1	0	0	X	**dcbtstls**
evabs	31 (0x1F)						rD					rA					///					0	1	0	0	0	0	0	1	0	0	0	EVX	**evabs**
evaddiw	31 (0x1F)						rD					UIMM					rB					0	1	0	0	0	0	0	0	0	1	0	EVX	**evaddiw**
evaddsmiaaw	31 (0x1F)						rD					rA					///					1	0	0	1	1	0	0	1	0	0	1	EVX	**evaddsmiaaw**
evaddssiaaw	31 (0x1F)						rD					rA					///					1	0	0	1	1	0	0	0	0	0	1	EVX	**evaddssiaaw**
evaddumiaaw	31 (0x1F)						rD					rA					///					1	0	0	1	1	0	0	1	0	0	0	EVX	**evaddumiaaw**
evaddusiaaw	31 (0x1F)						rD					rA					///					1	0	0	1	1	0	0	0	0	0	0	EVX	**evaddusiaaw**
evaddw	31 (0x1F)						rD					rA					rB					0	1	0	0	0	0	0	0	0	0	0	EVX	**evaddw**
evand	31 (0x1F)						rD					rA					rB					0	1	0	0	0	0	1	0	0	0	1	EVX	**evand**
evandc	31 (0x1F)						rD					rA					rB					0	1	0	0	0	0	1	0	0	1	0	EVX	**evandc**
evcmpeq	31 (0x1F)						crfD			/	/	rA					rB					0	1	0	0	0	1	1	0	1	0	0	EVX	**evcmpeq**
evcmpgts	31 (0x1F)						crfD			/	/	rA					rB					0	1	0	0	0	1	1	0	0	0	1	EVX	**evcmpgts**
evcmpgtu	31 (0x1F)						crfD			/	/	rA					rB					0	1	0	0	0	1	1	0	0	0	0	EVX	**evcmpgtu**
evcmplts	31 (0x1F)						crfD			/	/	rA					rB					0	1	0	0	0	1	1	0	0	1	1	EVX	**evcmplts**
evcmpltu	31 (0x1F)						crfD			/	/	rA					rB					0	1	0	0	0	1	1	0	0	1	0	EVX	**evcmpltu**
evcntlsw	31 (0x1F)						rD					rA					///					0	1	0	0	0	0	0	1	1	1	0	EVX	**evcntlsw**
evcntlzw	31 (0x1F)						rD					rA					///					0	1	0	0	0	0	0	1	1	0	1	EVX	**evcntlzw**
evdivws	31 (0x1F)						rD					rA					rB					1	0	0	1	1	0	0	0	1	1	0	EVX	**evdivws**
evdivwu	31 (0x1F)						rD					rA					rB					1	0	0	1	1	0	0	0	1	1	1	EVX	**evdivwu**
eveqv	31 (0x1F)						rD					rA					rB					0	1	0	0	0	0	1	1	0	0	1	EVX	**eveqv**
evextsb	31 (0x1F)						rD					rA					///					0	1	0	0	0	0	0	1	0	1	0	EVX	**evextsb**
evextsh	31 (0x1F)						rD					rA					///					0	1	0	0	0	0	0	1	0	1	1	EVX	**evextsh**
evfsabs	31 (0x1F)						rD					rA					///					0	1	0	1	0	0	0	0	1	0	0	EVX	**evfsabs**
evfsadd	31 (0x1F)						rD					rA					rB					0	1	0	1	0	0	0	0	0	0	0	EVX	**evfsadd**
evfscfsf	31 (0x1F)						rD					///					rB					0	1	0	1	0	0	1	0	0	1	1	EVX	**evfscfsf**
evfscfsi	31 (0x1F)						rD					///					rB					0	1	0	1	0	0	1	0	0	0	1	EVX	**evfscfsi**
evfscfuf	31 (0x1F)						rD					///					rB					0	1	0	1	0	0	1	0	0	1	0	EVX	**evfscfuf**
evfscfui	31 (0x1F)						rD					///					rB					0	1	0	1	0	0	1	0	0	0	0	EVX	**evfscfui**
evfscmpeq	31 (0x1F)						crfD			/	/	rA					rB					0	1	0	1	0	0	0	1	1	1	0	EVX	**evfscmpeq**
evfscmpgt	31 (0x1F)						crfD			/	/	rA					rB					0	1	0	1	0	0	0	1	1	0	0	EVX	**evfscmpgt**
evfscmplt	31 (0x1F)						crfD			/	/	rA					rB					0	1	0	1	0	0	0	1	1	0	1	EVX	**evfscmplt**
evfsctsf	31 (0x1F)						rD					///					rB					0	1	0	1	0	0	1	0	1	1	1	EVX	**evfsctsf**

Table 270. Instructions sorted by primary opcodes (decimal and hexadecimal) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
evfsctsi	31 (0x1F)						rD					///					rB					0	1	0	1	0	0	1	0	1	0	1	EVX	**evfsctsi**
evfsctsiz	31 (0x1F)						rD					///					rB					0	1	0	1	0	0	1	1	0	1	0	EVX	**evfsctsiz**
evfsctuf	31 (0x1F)						rD					///					rB					0	1	0	1	0	0	1	0	1	1	0	EVX	**evfsctuf**
evfsctui	31 (0x1F)						rD					///					rB					0	1	0	1	0	0	1	0	1	0	0	EVX	**evfsctui**
evfsctuiz	31 (0x1F)						rD					///					rB					0	1	0	1	0	0	1	1	0	0	0	EVX	**evfsctuiz**
evfsdiv	31 (0x1F)						rD					rA					rB					0	1	0	1	0	0	0	1	0	0	1	EVX	**evfsdiv**
evfsmul	31 (0x1F)						rD					rA					rB					0	1	0	1	0	0	0	1	0	0	0	EVX	**evfsmul**
evfsnabs	31 (0x1F)						rD					rA					///					0	1	0	1	0	0	0	0	1	0	1	EVX	**evfsnabs**
evfsneg	31 (0x1F)						rD					rA					///					0	1	0	1	0	0	0	0	1	1	0	EVX	**evfsneg**
evfssub	31 (0x1F)						rD					rA					rB					0	1	0	1	0	0	0	0	0	0	1	EVX	**evfssub**
evfststeq	31 (0x1F)						crfD			/	/	rA					rB					0	1	0	1	0	0	1	1	1	1	0	EVX	**evfststeq**
evfststgt	31 (0x1F)						crfD			/	/	rA					rB					0	1	0	1	0	0	1	1	1	0	0	EVX	**evfststgt**
evfststlt	31 (0x1F)						crfD			/	/	rA					rB					0	1	0	1	0	0	1	1	1	0	1	EVX	**evfststlt**
evldd	31 (0x1F)						rD					rA					UIMM [2]					0	1	1	0	0	0	0	0	0	0	1	EVX	**evldd**
evlddx	31 (0x1F)						rD					rA					rB					0	1	1	0	0	0	0	0	0	0	0	EVX	**evlddx**
evldh	31 (0x1F)						rD					rA					UIMM [2]					0	1	1	0	0	0	0	0	1	0	1	EVX	**evldh**
evldhx	31 (0x1F)						rD					rA					rB					0	1	1	0	0	0	0	0	1	0	0	EVX	**evldhx**
evldw	31 (0x1F)						rD					rA					UIMM [2]					0	1	1	0	0	0	0	0	0	1	1	EVX	**evldw**
evldwx	31 (0x1F)						rD					rA					rB					0	1	1	0	0	0	0	0	0	1	0	EVX	**evldwx**
evlhhesplat	31 (0x1F)						rD					rA					UIMM [2]					0	1	1	0	0	0	0	1	0	0	1	EVX	**evlhhesplat**
evlhhesplatx	31 (0x1F)						rD					rA					rB					0	1	1	0	0	0	0	1	0	0	0	EVX	**evlhhesplatx**
evlhhossplat	31 (0x1F)						rD					rA					UIMM [3]					0	1	1	0	0	0	0	1	1	1	1	EVX	**evlhhossplat**
evlhhossplatx	31 (0x1F)						rD					rA					rB					0	1	1	0	0	0	0	1	1	1	0	EVX	**evlhhossplatx**
evlhhousplat	31 (0x1F)						rD					rA					UIMM [3]					0	1	1	0	0	0	0	1	1	0	1	EVX	**evlhhousplat**
evlhhousplatx	31 (0x1F)						rD					rA					rB					0	1	1	0	0	0	0	1	1	0	0	EVX	**evlhhousplatx**
evlwhe	31 (0x1F)						rD					rA					UIMM [2]					0	1	1	0	0	0	1	0	0	0	1	EVX	**evlwhe**
evlwhex	31 (0x1F)						rD					rA					rB					0	1	1	0	0	0	1	0	0	0	0	EVX	**evlwhex**
evlwhos	31 (0x1F)						rD					rA					UIMM [4]					0	1	1	0	0	0	1	0	1	1	1	EVX	**evlwhos**
evlwhosx	31 (0x1F)						rD					rA					rB					0	1	1	0	0	0	1	0	1	1	0	EVX	**evlwhosx**
evlwhou	31 (0x1F)						rD					rA					UIMM [4]					0	1	1	0	0	0	1	0	1	0	1	EVX	**evlwhou**
evlwhoux	31 (0x1F)						rD					rA					rB					0	1	1	0	0	0	1	0	1	0	0	EVX	**evlwhoux**
evlwhsplat	31 (0x1F)						rD					rA					UIMM [4]					0	1	1	0	0	0	1	1	1	0	1	EVX	**evlwhsplat**
evlwhsplatx	31 (0x1F)						rD					rA					rB					0	1	1	0	0	0	1	1	1	0	0	EVX	**evlwhsplatx**
evlwwsplat	31 (0x1F)						rD					rA					UIMM [4]					0	1	1	0	0	0	1	1	0	0	1	EVX	**evlwwsplat**

Table 270. Instructions sorted by primary opcodes (decimal and hexadecimal) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
evlwwsplatx	31 (0x1F)						rD					rA					rB					0	1	1	0	0	0	1	1	0	0	0	EVX	evlwwsplatx
evmergehi	31 (0x1F)						rD					rA					rB					0	1	0	0	0	1	0	1	1	0	0	EVX	evmergehi
evmergehilo	31 (0x1F)						rD					rA					rB					0	1	0	0	0	1	0	1	1	1	0	EVX	evmergehilo
evmergelo	31 (0x1F)						rD					rA					rB					0	1	0	0	0	1	0	1	1	0	1	EVX	evmergelo
evmergelohi	31 (0x1F)						rD					rA					rB					0	1	0	0	0	1	0	1	1	1	1	EVX	evmergelohi
evmhegsmfaa	31 (0x1F)						rD					rA					rB					1	0	1	0	0	1	0	1	0	1	1	EVX	evmhegsmfaa
evmhegsmfan	31 (0x1F)						rD					rA					rB					1	0	1	1	0	1	0	1	0	1	1	EVX	evmhegsmfan
evmhegsmiaa	31 (0x1F)						rD					rA					rB					1	0	1	0	0	1	0	1	0	0	1	EVX	evmhegsmiaa
evmhegsmian	31 (0x1F)						rD					rA					rB					1	0	1	1	0	1	0	1	0	0	1	EVX	evmhegsmian
evmhegumiaa	31 (0x1F)						rD					rA					rB					1	0	1	0	0	1	0	1	0	0	0	EVX	evmhegumiaa
evmhegumian	31 (0x1F)						rD					rA					rB					1	0	1	1	0	1	0	1	0	0	0	EVX	evmhegumian
evmhesmf	31 (0x1F)						rD					rA					rB					1	0	0	0	0	0	0	1	0	1	1	EVX	evmhesmf
evmhesmfa	31 (0x1F)						rD					rA					rB					1	0	0	0	0	1	0	1	0	1	1	EVX	evmhesmfa
evmhesmfaaw	31 (0x1F)						rD					rA					rB					1	0	1	0	0	0	0	1	0	1	1	EVX	evmhesmfaaw
evmhesmfanw	31 (0x1F)						rD					rA					rB					1	0	1	1	0	0	0	1	0	1	1	EVX	evmhesmfanw
evmhesmi	31 (0x1F)						rD					rA					rB					1	0	0	0	0	0	0	1	0	0	1	EVX	evmhesmi
evmhesmia	31 (0x1F)						rD					rA					rB					1	0	0	0	0	1	0	1	0	0	1	EVX	evmhesmia
evmhesmiaaw	31 (0x1F)						rD					rA					rB					1	0	1	0	0	0	0	1	0	0	1	EVX	evmhesmiaaw
evmhesmianw	31 (0x1F)						rD					rA					rB					1	0	1	1	0	0	0	1	0	0	1	EVX	evmhesmianw
evmhessf	31 (0x1F)						rD					rA					rB					1	0	0	0	0	0	0	0	0	1	1	EVX	evmhessf
evmhessfa	31 (0x1F)						rD					rA					rB					1	0	0	0	0	1	0	0	0	1	1	EVX	evmhessfa
evmhessfaaw	31 (0x1F)						rD					rA					rB					1	0	1	0	0	0	0	0	0	1	1	EVX	evmhessfaaw
evmhessfanw	31 (0x1F)						rD					rA					rB					1	0	1	1	0	0	0	0	0	1	1	EVX	evmhessfanw
evmhessiaaw	31 (0x1F)						rD					rA					rB					1	0	1	0	0	0	0	0	0	0	1	EVX	evmhessiaaw
evmhessianw	31 (0x1F)						rD					rA					rB					1	0	1	1	0	0	0	0	0	0	1	EVX	evmhessianw
evmheumi	31 (0x1F)						rD					rA					rB					1	0	0	0	0	0	0	1	0	0	0	EVX	evmheumi

Table 270. Instructions sorted by primary opcodes (decimal and hexadecimal) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
evmheumia	31 (0x1F)						rD					rA					rB					1	0	0	0	0	1	0	1	0	0	0	EVX	evmheumia
evmheumiaaw	31 (0x1F)						rD					rA					rB					1	0	1	0	0	0	0	1	0	0	0	EVX	evmheumiaaw
evmheumianw	31 (0x1F)						rD					rA					rB					1	0	1	1	0	0	0	1	0	0	0	EVX	evmheumianw
evmheusiaaw	31 (0x1F)						rD					rA					rB					1	0	1	0	0	0	0	0	0	0	0	EVX	evmheusiaaw
evmheusianw	31 (0x1F)						rD					rA					rB					1	0	1	1	0	0	0	0	0	0	0	EVX	evmheusianw
evmhogsmfaa	31 (0x1F)						rD					rA					rB					1	0	1	0	0	1	0	1	1	1	1	EVX	evmhogsmfaa
evmhogsmfan	31 (0x1F)						rD					rA					rB					1	0	1	1	0	1	0	1	1	1	1	EVX	evmhogsmfan
evmhogsmiaa	31 (0x1F)						rD					rA					rB					1	0	1	0	0	1	0	1	1	0	1	EVX	evmhogsmiaa
evmhogsmian	31 (0x1F)						rD					rA					rB					1	0	1	1	0	1	0	1	1	0	1	EVX	evmhogsmian
evmhogumiaa	31 (0x1F)						rD					rA					rB					1	0	1	0	0	1	0	1	1	0	0	EVX	evmhogumiaa
evmhogumian	31 (0x1F)						rD					rA					rB					1	0	1	1	0	1	0	1	1	0	0	EVX	evmhogumian
evmhosmf	31 (0x1F)						rD					rA					rB					1	0	0	0	0	0	0	1	1	1	1	EVX	evmhosmf
evmhosmfa	31 (0x1F)						rD					rA					rB					1	0	0	0	0	1	0	1	1	1	1	EVX	evmhosmfa
evmhosmfaaw	31 (0x1F)						rD					rA					rB					1	0	1	0	0	0	0	1	1	1	1	EVX	evmhosmfaaw
evmhosmfanw	31 (0x1F)						rD					rA					rB					1	0	1	1	0	0	0	1	1	1	1	EVX	evmhosmfanw
evmhosmi	31 (0x1F)						rD					rA					rB					1	0	0	0	0	0	0	1	1	0	1	EVX	evmhosmi
evmhosmia	31 (0x1F)						rD					rA					rB					1	0	0	0	0	1	0	1	1	0	1	EVX	evmhosmia
evmhosmiaaw	31 (0x1F)						rD					rA					rB					1	0	1	0	0	0	0	1	1	0	1	EVX	evmhosmiaaw
evmhosmianw	31 (0x1F)						rD					rA					rB					1	0	1	1	0	0	0	1	1	0	1	EVX	evmhosmianw
evmhossf	31 (0x1F)						rD					rA					rB					1	0	0	0	0	0	0	0	1	1	1	EVX	evmhossf
evmhossfa	31 (0x1F)						rD					rA					rB					1	0	0	0	0	1	0	0	1	1	1	EVX	evmhossfa
evmhossfaaw	31 (0x1F)						rD					rA					rB					1	0	1	0	0	0	0	0	1	1	1	EVX	evmhossfaaw
evmhossfanw	31 (0x1F)						rD					rA					rB					1	0	1	1	0	0	0	0	1	1	1	EVX	evmhossfanw
evmhossiaaw	31 (0x1F)						rD					rA					rB					1	0	1	0	0	0	0	0	1	0	1	EVX	evmhossiaaw
evmhossianw	31 (0x1F)						rD					rA					rB					1	0	1	1	0	0	0	0	1	0	1	EVX	evmhossianw
evmhoumi	31 (0x1F)						rD					rA					rB					1	0	0	0	0	0	0	1	1	0	0	EVX	evmhoumi

Table 270. Instructions sorted by primary opcodes (decimal and hexadecimal) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
evmhoumia	31 (0x1F)						rD					rA					rB					1	0	0	0	0	1	0	1	1	0	0	EVX	**evmhoumia**
evmhoumiaaw	31 (0x1F)						rD					rA					rB					1	0	1	0	0	0	0	1	1	0	0	EVX	**evmhoumiaaw**
evmhoumianw	31 (0x1F)						rD					rA					rB					1	0	1	1	0	0	0	1	1	0	0	EVX	**evmhoumianw**
evmhousiaaw	31 (0x1F)						rD					rA					rB					1	0	1	0	0	0	0	0	1	0	0	EVX	**evmhousiaaw**
evmhousianw	31 (0x1F)						rD					rA					rB					1	0	1	1	0	0	0	0	1	0	0	EVX	**evmhousianw**
evmra	31 (0x1F)						rD					rA					///					1	0	0	1	1	0	0	0	1	0	0	EVX	**evmra**
evmwhgsmfaa	31 (0x1F)						rD					rA					rB					1	0	1	0	1	1	0	1	1	1	1	EVX	**evmwhgsmfaa**
evmwhgsmfan	31 (0x1F)						rD					rA					rB					1	0	1	1	1	0	1	1	1	1	1	EVX	**evmwhgsmfan**
evmwhgsmiaa	31 (0x1F)						rD					rA					rB					1	0	1	0	1	1	0	1	1	0	1	EVX	**evmwhgsmiaa**
evmwhgsmian	31 (0x1F)						rD					rA					rB					1	0	1	1	1	0	1	1	1	0	1	EVX	**evmwhgsmian**
evmwhgssfaa	31 (0x1F)						rD					rA					rB					1	0	1	0	1	1	0	0	1	1	1	EVX	**evmwhgssfaa**
evmwhgssfan	31 (0x1F)						rD					rA					rB					1	0	1	1	1	0	1	0	1	1	1	EVX	**evmwhgssfan**
evmwhgumiaa	31 (0x1F)						rD					rA					rB					1	0	1	0	1	1	0	1	1	0	0	EVX	**evmwhgumiaa**
evmwhgumian	31 (0x1F)						rD					rA					rB					1	0	1	1	1	0	1	1	1	0	0	EVX	**evmwhgumian**
evmwhsmf	31 (0x1F)						rD					rA					rB					1	0	0	0	1	0	0	1	1	1	1	EVX	**evmwhsmf**
evmwhsmfa	31 (0x1F)						rD					rA					rB					1	0	0	0	1	1	0	1	1	1	1	EVX	**evmwhsmfa**
evmwhsmfaaw	31 (0x1F)						rD					rA					rB					1	0	1	0	1	0	0	1	1	1	1	EVX	**evmwhsmfaaw**
evmwhsmfanw	31 (0x1F)						rD					rA					rB					1	0	1	1	1	0	0	1	1	1	1	EVX	**evmwhsmfanw**
evmwhsmi	31 (0x1F)						rD					rA					rB					1	0	0	0	1	0	0	1	1	0	1	EVX	**evmwhsmi**
evmwhsmia	31 (0x1F)						rD					rA					rB					1	0	0	0	1	1	0	1	1	0	1	EVX	**evmwhsmia**
evmwhsmiaaw	31 (0x1F)						rD					rA					rB					1	0	1	0	1	0	0	1	1	0	1	EVX	**evmwhsmiaaw**
evmwhsmianw	31 (0x1F)						rD					rA					rB					1	0	1	1	1	0	0	1	1	0	1	EVX	**evmwhsmianw**
evmwhssf	31 (0x1F)						rD					rA					rB					1	0	0	0	1	0	0	0	1	1	1	EVX	**evmwhssf**
evmwhssfa	31 (0x1F)						rD					rA					rB					1	0	0	0	1	1	0	0	1	1	1	EVX	**evmwhssfa**
evmwhssfaaw	31 (0x1F)						rD					rA					rB					1	0	1	0	1	0	0	0	1	1	1	EVX	**evmwhssfaaw**
evmwhssfanw	31 (0x1F)						rD					rA					rB					1	0	1	1	1	0	0	0	1	1	1	EVX	**evmwhssfanw**

Table 270. Instructions sorted by primary opcodes (decimal and hexadecimal) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
evmwhssianw	31 (0x1F)						rD					rA					rB					1	0	1	1	1	0	0	0	1	0	1	EVX	evmwhssianw
evmwhssmaaw	31 (0x1F)						rD					rA					rB					1	0	1	0	1	0	0	0	1	0	1	EVX	evmwhssmaaw
evmwhumi	31 (0x1F)						rD					rA					rB					1	0	0	0	1	0	0	1	1	0	0	EVX	evmwhumi
evmwhumia	31 (0x1F)						rD					rA					rB					1	0	0	0	1	1	0	1	1	0	0	EVX	evmwhumia
evmwhusiaaw	31 (0x1F)						rD					rA					rB					1	0	1	0	1	0	0	0	1	0	0	EVX	evmwhusiaaw
evmwhusianw	31 (0x1F)						rD					rA					rB					1	0	1	1	1	0	0	0	1	0	0	EVX	evmwhusianw
evmwlsmf	31 (0x1F)						rD					rA					rB					1	0	0	0	1	0	0	1	0	1	1	EVX	evmwlsmf
evmwlsmfa	31 (0x1F)						rD					rA					rB					1	0	0	0	1	1	0	1	0	1	1	EVX	evmwlsmfa
evmwlsmfaaw	31 (0x1F)						rD					rA					rB					1	0	1	0	1	0	0	1	0	1	1	EVX	evmwlsmfaaw
evmwlsmfanw	31 (0x1F)						rD					rA					rB					1	0	1	1	1	0	0	1	0	1	1	EVX	evmwlsmfanw
evmwlsmiaaw	31 (0x1F)						rD					rA					rB					1	0	1	0	1	0	0	1	0	0	1	EVX	evmwlsmiaaw
evmwlsmianw	31 (0x1F)						rD					rA					rB					1	0	1	1	1	0	0	1	0	0	1	EVX	evmwlsmianw
evmwlssf	31 (0x1F)						rD					rA					rB					1	0	0	0	1	0	0	0	0	1	1	EVX	evmwlssf
evmwlssfa	31 (0x1F)						rD					rA					rB					1	0	0	0	1	1	0	0	0	1	1	EVX	evmwlssfa
evmwlssfaaw	31 (0x1F)						rD					rA					rB					1	0	1	0	1	0	0	0	0	1	1	EVX	evmwlssfaaw
evmwlssfanw	31 (0x1F)						rD					rA					rB					1	0	1	1	1	0	0	0	0	1	1	EVX	evmwlssfanw
evmwlssiaaw	31 (0x1F)						rD					rA					rB					1	0	1	0	1	0	0	0	0	0	1	EVX	evmwlssiaaw
evmwlssianw	31 (0x1F)						rD					rA					rB					1	0	1	1	1	0	0	0	0	0	1	EVX	evmwlssianw
evmwlumi	31 (0x1F)						rD					rA					rB					1	0	0	0	1	0	0	1	0	0	0	EVX	evmwlumi
evmwlumia	31 (0x1F)						rD					rA					rB					1	0	0	0	1	1	0	1	0	0	0	EVX	evmwlumia
evmwlumiaaw	31 (0x1F)						rD					rA					rB					1	0	1	0	1	0	0	1	0	0	0	EVX	evmwlumiaaw
evmwlumianw	31 (0x1F)						rD					rA					rB					1	0	1	1	1	0	0	1	0	0	0	EVX	evmwlumianw
evmwlusiaaw	31 (0x1F)						rD					rA					rB					1	0	1	0	1	0	0	0	0	0	0	EVX	evmwlusiaaw
evmwlusianw	31 (0x1F)						rD					rA					rB					1	0	1	1	1	0	0	0	0	0	0	EVX	evmwlusianw
evmwsmf	31 (0x1F)						rD					rA					rB					1	0	0	0	1	0	1	1	0	1	1	EVX	evmwsmf
evmwsmfa	31 (0x1F)						rD					rA					rB					1	0	0	0	1	1	1	1	0	1	1	EVX	evmwsmfa

Table 270. Instructions sorted by primary opcodes (decimal and hexadecimal) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
evmwsmfaa	31 (0x1F)						rD					rA					rB					1	0	1	0	1	0	1	1	0	1	1	EVX	**evmwsmfaa**
evmwsmfan	31 (0x1F)						rD					rA					rB					1	0	1	1	1	0	1	1	0	1	1	EVX	**evmwsmfan**
evmwsmi	31 (0x1F)						rD					rA					rB					1	0	0	0	1	0	1	1	0	0	1	EVX	**evmwsmi**
evmwsmia	31 (0x1F)						rD					rA					rB					1	0	0	0	1	1	1	1	0	0	1	EVX	**evmwsmia**
evmwsmiaa	31 (0x1F)						rD					rA					rB					1	0	1	0	1	0	1	1	0	0	1	EVX	**evmwsmiaa**
evmwsmian	31 (0x1F)						rD					rA					rB					1	0	1	1	1	0	1	1	0	0	1	EVX	**evmwsmian**
evmwssf	31 (0x1F)						rD					rA					rB					1	0	0	0	1	0	1	0	0	1	1	EVX	**evmwssf**
evmwssfa	31 (0x1F)						rD					rA					rB					1	0	0	0	1	1	1	0	0	1	1	EVX	**evmwssfa**
evmwssfaa	31 (0x1F)						rD					rA					rB					1	0	1	0	1	0	1	0	0	1	1	EVX	**evmwssfaa**
evmwssfan	31 (0x1F)						rD					rA					rB					1	0	1	1	1	0	1	0	0	1	1	EVX	**evmwssfan**
evmwumi	31 (0x1F)						rD					rA					rB					1	0	0	0	1	0	1	1	0	0	0	EVX	**evmwumi**
evmwumia	31 (0x1F)						rD					rA					rB					1	0	0	0	1	1	1	1	0	0	0	EVX	**evmwumia**
evmwumiaa	31 (0x1F)						rD					rA					rB					1	0	1	0	1	0	1	1	0	0	0	EVX	**evmwumiaa**
evmwumian	31 (0x1F)						rD					rA					rB					1	0	1	1	1	0	1	1	0	0	0	EVX	**evmwumian**
evnand	31 (0x1F)						rD					rA					rB					0	1	0	0	0	0	1	1	1	1	0	EVX	**evnand**
evneg	31 (0x1F)						rD					rA					///					0	1	0	0	0	0	0	1	0	0	1	EVX	**evneg**
evnor	31 (0x1F)						rD					rA					rB					0	1	0	0	0	0	1	1	0	0	0	EVX	**evnor**
evor	31 (0x1F)						rD					rA					rB					0	1	0	0	0	0	1	0	1	1	1	EVX	**evor**
evorc	31 (0x1F)						rD					rA					rB					0	1	0	0	0	0	1	1	0	1	1	EVX	**evorc**
evrlw	31 (0x1F)						rD					rA					rB					0	1	0	0	0	1	0	1	0	0	0	EVX	**evrlw**
evrlwi	31 (0x1F)						rD					rA					UIMM					0	1	0	0	0	1	0	1	0	1	0	EVX	**evrlwi**
evrndw	31 (0x1F)						rD					rA					UIMM					0	1	0	0	0	0	0	1	1	0	0	EVX	**evrndw**
evsel	31 (0x1F)						rD					rA					rB					0	1	0	0	1	1	1	1	crfS			EVX	**evsel**
evslw	31 (0x1F)						rD					rA					rB					0	1	0	0	0	1	0	0	1	0	0	EVX	**evslw**
evslwi	31 (0x1F)						rD					rA					UIMM					0	1	0	0	0	1	0	0	1	1	0	EVX	**evslwi**
evsplatfi	31 (0x1F)						rD					SIMM					///					0	1	0	0	0	1	0	1	0	1	1	EVX	**evsplatfi**
evsplati	31 (0x1F)						rD					SIMM					///					0	1	0	0	0	1	0	1	0	0	1	EVX	**evsplati**
evsrwis	31 (0x1F)						rD					rA					UIMM					0	1	0	0	0	1	0	0	0	1	1	EVX	**evsrwis**
evsrwiu	31 (0x1F)						rD					rA					UIMM					0	1	0	0	0	1	0	0	0	1	0	EVX	**evsrwiu**
evsrws	31 (0x1F)						rD					rA					rB					0	1	0	0	0	1	0	0	0	0	1	EVX	**evsrws**
evsrwu	31 (0x1F)						rD					rA					rB					0	1	0	0	0	1	0	0	0	0	0	EVX	**evsrwu**
evstdd	31 (0x1F)						rD					rA					UIMM [4]					0	1	1	0	0	1	0	0	0	0	1	EVX	**evstdd**
evstddx	31 (0x1F)						rS					rA					rB					0	1	1	0	0	1	0	0	0	0	0	EVX	**evstddx**

Table 270. Instructions sorted by primary opcodes (decimal and hexadecimal) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
evstdh	31 (0x1F)						rS					rA					UIMM [2]					0	1	1	0	0	1	0	0	1	0	1	EVX	**evstdh**
evstdhx	31 (0x1F)						rS					rA					rB					0	1	1	0	0	1	0	0	1	0	0	EVX	**evstdhx**
evstdw	31 (0x1F)						rS					rA					UIMM [2]					0	1	1	0	0	1	0	0	0	1	1	EVX	**evstdw**
evstdwx	31 (0x1F)						rS					rA					rB					0	1	1	0	0	1	0	0	0	1	0	EVX	**evstdwx**
evstwhe	31 (0x1F)						rS					rA					UIMM [4]					0	1	1	0	0	1	1	0	0	0	1	EVX	**evstwhe**
evstwhex	31 (0x1F)						rS					rA					rB					0	1	1	0	0	1	1	0	0	0	0	EVX	**evstwhex**
evstwho	31 (0x1F)						rS					rA					UIMM [4]					0	1	1	0	0	1	1	0	1	0	1	EVX	**evstwho**
evstwhox	31 (0x1F)						rS					rA					rB					0	1	1	0	0	1	1	0	1	0	0	EVX	**evstwhox**
evstwwe	31 (0x1F)						rS					rA					UIMM [4]					0	1	1	0	0	1	1	1	0	0	1	EVX	**evstwwe**
evstwwex	31 (0x1F)						rS					rA					rB					0	1	1	0	0	1	1	1	0	0	0	EVX	**evstwwex**
evstwwo	31 (0x1F)						rS					rA					UIMM [4]					0	1	1	0	0	1	1	1	1	0	1	EVX	**evstwwo**
evstwwox	31 (0x1F)						rS					rA					rB					0	1	1	0	0	1	1	1	1	0	0	EVX	**evstwwox**
evsubfsmiaaw	31 (0x1F)						rD					rA					///					1	0	0	1	1	0	0	1	0	1	1	EVX	**evsubfsmiaaw**
evsubfssiaaw	31 (0x1F)						rD					rA					///					1	0	0	1	1	0	0	0	0	1	1	EVX	**evsubfssiaaw**
evsubfumiaaw	31 (0x1F)						rD					rA					///					1	0	0	1	1	0	0	1	0	1	0	EVX	**evsubfumiaaw**
evsubfusiaaw	31 (0x1F)						rD					rA					///					1	0	0	1	1	0	0	0	0	1	0	EVX	**evsubfusiaaw**
evsubfw	31 (0x1F)						rD					rA					rB					0	1	0	0	0	0	0	0	1	0	0	EVX	**evsubfw**
evsubifw	31 (0x1F)						rD					UIMM					rB					0	1	0	0	0	0	0	0	1	1	0	EVX	**evsubifw**
evxor	31 (0x1F)						rD					rA					rB					0	1	0	0	0	0	1	0	1	1	0	EVX	**evxor**
icblc	31 (0x1F)						CT					rA					rB					0	0	1	1	1	0	0	1	1	0	0	X	**icblc**
icbt	31 (0x1F)						CT					rA					rB					0	0	0	0	0	1	0	1	1	0	/	X	**icbt**
icbtls	31 (0x1F)						CT					rA					rB					0	1	1	1	1	0	0	1	1	0	0	X	**icbtls**
isel	31 (0x1F)						rD					rA					rB					crb					0	1	1	1	1	0	X	**isel**
mbar	31 (0x1F)						MO					///										1	1	0	1	0	1	0	1	1	0	/	X	**mbar**
mfdcr	31 (0x1F)						rD					DCRN5–9					DCRN0–4					0	1	0	1	0	0	0	0	1	1	/	XFX	**mfdcr**
mfpmr	31 (0x1F)						rD					PMRN5–9					PMRN0–4					0	1	0	1	0	0	1	1	1	0	0	XFX	**mfpmr**
msync	31 (0x1F)						///															1	0	0	1	0	1	0	1	1	0	/	X	**msync**
mtdcr	31 (0x1F)						rS					DCRN5–9					DCRN0–4					0	1	1	1	0	0	0	0	1	1	/	XFX	**mtdcr**
mtpmr	31 (0x1F)						rS					PMRN5–9					PMRN0–4					0	1	1	1	0	0	1	1	1	0	0	XFX	**mtpmr**
tlbivax	31 (0x1F)						///					rA					rB					1	1	0	0	0	1	0	0	1	0	/	X	**tlbivax**
tlbre	31 (0x1F)						/// [2]															1	1	1	0	1	1	0	0	1	0	/	X	**tlbre**
tlbsx	31 (0x1F)						/// [5]					rA					rB					1	1	1	0	0	1	0	0	1	0	/ [5]	X	**tlbsx**
tlbwe	31 (0x1F)						/// [6]															1	1	1	1	0	1	0	0	1	0	/	X	**tlbwe**
wait	31 (0x1F)						///															0	0	0	0	1	1	1	1	1	0	/		**wait**

Table 270. Instructions sorted by primary opcodes (decimal and hexadecimal) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
wrtee	31 (0x1F)						rS					///										0	0	1	0	0	0	0	0	1	1	/	X	**wrtee**
wrteei	31 (0x1F)						///										E	///				0	0	1	0	1	0	0	0	1	1	/	X	**wrteei**
cmp	31 (0x1F)						crfD			/	L	rA					rB					0	0	0	0	0	0	0	0	0	0	/	X	**cmp**
tw	31 (0x1F)						TO					rA					rB					0	0	0	0	0	0	0	1	0	0	/	X	**tw**
subfc	31 (0x1F)						rD					rA					rB					0	0	0	0	0	0	1	0	0	0	0	X	**subfc**
subfc.	31 (0x1F)						rD					rA					rB					0	0	0	0	0	0	1	0	0	0	1	X	**subfc.**
addc	31 (0x1F)						rD					rA					rB					0	0	0	0	0	0	1	0	1	0	0	X	**addc**
addc.	31 (0x1F)						rD					rA					rB					0	0	0	0	0	0	1	0	1	0	1	X	**addc.**
mulhwu	31 (0x1F)						rD					rA					rB					/	0	0	0	0	0	1	0	1	1	0	X	**mulhwu**
mulhwu.	31 (0x1F)						rD					rA					rB					/	0	0	0	0	0	1	0	1	1	1	X	**mulhwu.**
mfcr	31 (0x1F)						rD					///										0	0	0	0	0	1	0	0	1	1	/	X	**mfcr**
lwarx	31 (0x1F)						rD					rA					rB					0	0	0	0	0	1	0	1	0	0	/	X	**lwarx**
lwzx	31 (0x1F)						rD					rA					rB					0	0	0	0	0	1	0	1	1	1	/	X	**lwzx**
slw	31 (0x1F)						rS					rA					rB					0	0	0	0	0	1	1	0	0	0	0	X	**slw**
slw.	31 (0x1F)						rS					rA					rB					0	0	0	0	0	1	1	0	0	0	1	X	**slw.**
cntlzw	31 (0x1F)						rS					rA					///					0	0	0	0	0	1	1	0	1	0	0	X	**cntlzw**
cntlzw.	31 (0x1F)						rS					rA					///					0	0	0	0	0	1	1	0	1	0	1	X	**cntlzw.**
and	31 (0x1F)						rS					rA					rB					0	0	0	0	0	1	1	1	0	0	0	X	**and**
and.	31 (0x1F)						rS					rA					rB					0	0	0	0	0	1	1	1	0	0	1	X	**and.**
cmpl	31 (0x1F)						/	L				rA					rB					///		0	0	0	0	1	0	0	0	0	0	/	X	**cmpl**
subf	31 (0x1F)						rD					rA					rB					0	0	0	0	1	0	1	0	0	0	0	X	**subf**
subf.	31 (0x1F)						rD					rA					rB					0	0	0	0	1	0	1	0	0	0	1	X	**subf.**
dcbst	31 (0x1F)						///					rA					rB					0	0	0	0	1	1	0	1	1	0	/	X	**dcbst**
lwzux	31 (0x1F)						rD					rA					rB					0	0	0	0	1	1	0	1	1	1	/	X	**lwzux**
andc	31 (0x1F)						rS					rA					rB					0	0	0	0	1	1	1	1	0	0	0	X	**andc**
andc.	31 (0x1F)						rS					rA					rB					0	0	0	0	1	1	1	1	0	0	1	X	**andc.**
mulhw	31 (0x1F)						rD					rA					rB					/	0	0	1	0	0	1	0	1	1	0	X	**mulhw**
mulhw.	31 (0x1F)						rD					rA					rB					/	0	0	1	0	0	1	0	1	1	1	X	**mulhw.**
mfmsr [1]	31 (0x1F)						rD					///										0	0	0	1	0	1	0	0	1	1	/	X	**mfmsr**
dcbf	31 (0x1F)						///					rA					rB					0	0	0	1	0	1	0	1	1	0	/	X	**dcbf**
lbzx	31 (0x1F)						rD					rA					rB					0	0	0	1	0	1	0	1	1	1	/	X	**lbzx**
neg	31 (0x1F)						rD					rA					///					0	0	0	1	1	0	1	0	0	0	0	X	**neg**
neg.	31 (0x1F)						rD					rA					///					0	0	0	1	1	0	1	0	0	0	1	X	**neg.**
lbzux	31 (0x1F)						rD					rA					rB					0	0	0	1	1	1	0	1	1	1	/	X	**lbzux**
nor	31 (0x1F)						rS					rA					rB					0	0	0	1	1	1	1	1	0	0	0	X	**nor**
nor.	31 (0x1F)						rS					rA					rB					0	0	0	1	1	1	1	1	0	0	1	X	**nor.**
subfe	31 (0x1F)						rD					rA					rB					0	0	1	0	0	0	1	0	0	0	0	X	**subfe**
subfe.	31 (0x1F)						rD					rA					rB					0	0	1	0	0	0	1	0	0	0	1	X	**subfe.**
adde	31 (0x1F)						rD					rA					rB					0	0	1	0	0	0	1	0	1	0	0	X	**adde**
adde.	31 (0x1F)						rD					rA					rB					0	0	1	0	0	0	1	0	1	0	1	X	**adde.**

Table 270. Instructions sorted by primary opcodes (decimal and hexadecimal) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
mtcrf	31 (0x1F)						rS					/	CRM								/	0	0	1	0	0	1	0	0	0	0	/	XFX	**mtcrf**
mtmsr [1]	31 (0x1F)						rS					///										0	0	1	0	0	1	0	0	1	0	/	X	**mtmsr**
stwcx.	31 (0x1F)						rS					rA					rB					0	0	1	0	0	1	0	1	1	0	1	X	**stwcx.**
stwx	31 (0x1F)						rS					rA					rB					0	0	1	0	0	1	0	1	1	1	/	D	**stwx**
stwux	31 (0x1F)						rS					rA					rB					0	0	1	0	1	1	0	1	1	1	/	D	**stwux**
subfze	31 (0x1F)						rD					rA					///					0	0	1	1	0	0	1	0	0	0	0	X	**subfze**
subfze.	31 (0x1F)						rD					rA					///					0	0	1	1	0	0	1	0	0	0	1	X	**subfze.**
addze	31 (0x1F)						rD					rA					///					0	0	1	1	0	0	1	0	1	0	0	X	**addze**
addze.	31 (0x1F)						rD					rA					///					0	0	1	1	0	0	1	0	1	0	1	X	**addze.**
stbx	31 (0x1F)						rS					rA					rB					0	0	1	1	0	1	0	1	1	1	0	X	**stbx**
subfme	31 (0x1F)						rD					rA					///					0	0	1	1	1	0	1	0	0	0	0	X	**subfme**
subfme.	31 (0x1F)						rD					rA					///					0	0	1	1	1	0	1	0	0	0	1	X	**subfme.**
addme	31 (0x1F)						rD					rA					///					0	0	1	1	1	0	1	0	1	0	0	X	**addme**
addme.	31 (0x1F)						rD					rA					///					0	0	1	1	1	0	1	0	1	0	1	X	**addme.**
mullw	31 (0x1F)						rD					rA					rB					0	0	1	1	1	0	1	0	1	1	0	X	**mullw**
mullw.	31 (0x1F)						rD					rA					rB					0	0	1	1	1	0	1	0	1	1	1	X	**mullw.**
dcbtst	31 (0x1F)						CT					rA					rB					0	0	1	1	1	1	0	1	1	0	/	X	**dcbtst**
stbux	31 (0x1F)						rS					rA					rB					0	0	1	1	1	1	0	1	1	1	0	X	**stbux**
add	31 (0x1F)						rD					rA					rB					0	1	0	0	0	0	1	0	1	0	0	X	**add**
add.	31 (0x1F)						rD					rA					rB					0	1	0	0	0	0	1	0	1	0	1	X	**add.**
dcbt	31 (0x1F)						CT					rA					rB					0	1	0	0	0	1	0	1	1	0	/	X	**dcbt**
lhzx	31 (0x1F)						rD					rA					rB					0	1	0	0	0	1	0	1	1	1	/	X	**lhzx**
eqv	31 (0x1F)						rD					rA					rB					0	1	0	0	0	1	1	1	0	0	0	X	**eqv**
eqv.	31 (0x1F)						rD					rA					rB					0	1	0	0	0	1	1	1	0	0	1	X	**eqv.**
tlbie [1, 2]	31 (0x1F)						///					///					rB					0	1	0	0	1	1	0	0	1	0	0	X	**tlbie**
lhzux	31 (0x1F)						rD					rA					rB					0	1	0	0	1	1	0	1	1	1	/	X	**lhzux**
xor	31 (0x1F)						rS					rA					rB					0	1	0	0	1	1	1	1	0	0	0	X	**xor**
xor.	31 (0x1F)						rS					rA					rB					0	1	0	0	1	1	1	1	0	0	1	X	**xor.**
mfspr[2]	31 (0x1F)						rD					SPR[5–9]					SPR[0–4]					0	1	0	1	0	1	0	0	1	1	/	XFX	**mfspr**
lhax	31 (0x1F)						rD					rA					rB					0	1	0	1	0	1	0	1	1	1	/	X	**lhax**
lhaux	31 (0x1F)						rD					rA					rB					0	1	0	1	1	1	0	1	1	1	/	X	**lhaux**
sthx	31 (0x1F)						rS					rA					rB					0	1	1	0	0	1	0	1	1	1	/	X	**sthx**
orc	31 (0x1F)						rS					rA					rB					0	1	1	0	0	1	1	1	0	0	0	X	**orc**
orc.	31 (0x1F)						rS					rA					rB					0	1	1	0	0	1	1	1	0	0	1	X	**orc.**
sthux	31 (0x1F)						rS					rA					rB					0	1	1	0	1	1	0	1	1	1	/	X	**sthux**
or	31 (0x1F)						rS					rA					rB					0	1	1	0	1	1	1	1	0	0	0	X	**or**
or.	31 (0x1F)						rS					rA					rB					0	1	1	0	1	1	1	1	0	0	1	X	**or.**
divwu	31 (0x1F)						rD					rA					rB					0	1	1	1	0	0	1	0	1	1	0	X	**divwu**
divwu.	31 (0x1F)						rD					rA					rB					0	1	1	1	0	0	1	0	1	1	1	X	**divwu.**

Table 270. Instructions sorted by primary opcodes (decimal and hexadecimal) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
mtspr [2]	31 (0x1F)						rS					SPR[5–9]					SPR[0–4]					0	1	1	1	0	1	0	0	1	1	/	XFX	**mtspr**
dcbi [1]	31 (0x1F)						///					rA					rB					0	1	1	1	0	1	0	1	1	0	/	X	**dcbi**
nand	31 (0x1F)						rS					rA					rB					0	1	1	1	0	1	1	1	0	0	0	X	**nand**
nand.	31 (0x1F)						rS					rA					rB					0	1	1	1	0	1	1	1	0	0	1	X	**nand.**
divw	31 (0x1F)						rD					rA					rB					0	1	1	1	1	0	1	0	1	1	0	X	**divw**
divw.	31 (0x1F)						rD					rA					rB					0	1	1	1	1	0	1	0	1	1	1	X	**divw.**
mcrxr	31 (0x1F)						crfD			///												1	0	0	0	0	0	0	0	0	0	/	X	**mcrxr**
subfco	31 (0x1F)						rD					rA					rB					1	0	0	0	0	0	1	0	0	0	0	X	**subfco**
subfco.	31 (0x1F)						rD					rA					rB					1	0	0	0	0	0	1	0	0	0	1	X	**subfco.**
addco	31 (0x1F)						rD					rA					rB					1	0	0	0	0	0	1	0	1	0	0	X	**addco**
addco.	31 (0x1F)						rD					rA					rB					1	0	0	0	0	0	1	0	1	0	1	X	**addco.**
lwbrx	31 (0x1F)						rD					rA					rB					1	0	0	0	0	1	0	1	1	0	/	X	**lwbrx**
srw	31 (0x1F)						rS					rA					rB					1	0	0	0	0	1	1	0	0	0	0	X	**srw**
srw.	31 (0x1F)						rS					rA					rB					1	0	0	0	0	1	1	0	0	0	1	X	**srw.**
subfo	31 (0x1F)						rD					rA					rB					1	0	0	0	1	0	1	0	0	0	0	X	**subfo**
subfo.	31 (0x1F)						rD					rA					rB					1	0	0	0	1	0	1	0	0	0	1	X	**subfo.**
tlbsync [1,6]	31 (0x1F)						///					///					///					1	0	0	0	1	1	0	1	1	0	/	X	**tlbsync**
nego **MBC**	31 (0x1F)						rD					rA					///					1	0	0	1	1	0	1	0	0	0	0	X	**nego**
nego.	31 (0x1F)						rD					rA					///					1	0	0	1	1	0	1	0	0	0	1	X	**nego.**
subfeo	31 (0x1F)						rD					rA					rB					1	0	1	0	0	0	1	0	0	0	0	X	**subfeo**
subfeo.	31 (0x1F)						rD					rA					rB					1	0	1	0	0	0	1	0	0	0	1	X	**subfeo.**
addeo	31 (0x1F)						rD					rA					rB					1	0	1	0	0	0	1	0	1	0	0	X	**addeo**
addeo.	31 (0x1F)						rD					rA					rB					1	0	1	0	0	0	1	0	1	0	1	X	**addeo.**
stwbrx	31 (0x1F)						rS					rA					rB					1	0	1	0	0	1	0	1	1	0	/	X	**stwbrx**
subfzeo	31 (0x1F)						rD					rA					///					1	0	1	1	0	0	1	0	0	0	0	X	**subfzeo**
subfzeo.	31 (0x1F)						rD					rA					///					1	0	1	1	0	0	1	0	0	0	1	X	**subfzeo.**
addzeo	31 (0x1F)						rD					rA					///					1	0	1	1	0	0	1	0	1	0	0	X	**addzeo**
addzeo.	31 (0x1F)						rD					rA					///					1	0	1	1	0	0	1	0	1	0	1	X	**addzeo.**
subfmeo	31 (0x1F)						rD					rA					///					1	0	1	1	1	0	1	0	0	0	0	X	**subfmeo**
subfmeo.	31 (0x1F)						rD					rA					///					1	0	1	1	1	0	1	0	0	0	1	X	**subfmeo.**
addmeo	31 (0x1F)						rD					rA					///					1	0	1	1	1	0	1	0	1	0	0	X	**addmeo**
addmeo.	31 (0x1F)						rD					rA					///					1	0	1	1	1	0	1	0	1	0	1	X	**addmeo.**
mullwo	31 (0x1F)						rD					rA					rB					1	0	1	1	1	0	1	0	1	1	0	X	**mullwo**
mullwo.	31 (0x1F)						rD					rA					rB					1	0	1	1	1	0	1	0	1	1	1	X	**mullwo.**
dcba [6]	31 (0x1F)						///					rA					rB					1	0	1	1	1	1	0	1	1	0	/	X	**dcba**
addo	31 (0x1F)						rD					rA					rB					1	1	0	0	0	0	1	0	1	0	0	X	**addo**
addo.	31 (0x1F)						rD					rA					rB					1	1	0	0	0	0	1	0	1	0	1	X	**addo.**
lhbrx	31 (0x1F)						rD					rA					rB					1	1	0	0	0	1	0	1	1	0	/	X	**lhbrx**

Table 270. Instructions sorted by primary opcodes (decimal and hexadecimal) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
sraw	31 (0x1F)						rS					rA					rB					1	1	0	0	0	1	1	0	0	0	0	X	**sraw**
sraw.	31 (0x1F)						rS					rA					rB					1	1	0	0	0	1	1	0	0	0	1	X	**sraw.**
srawi	31 (0x1F)						rS					rA					SH					1	1	0	0	1	1	1	0	0	0	0	X	**srawi**
srawi.	31 (0x1F)						rS					rA					SH					1	1	0	0	1	1	1	0	0	0	1	X	**srawi.**
sthbrx	31 (0x1F)						rS					rA					rB					1	1	1	0	0	1	0	1	1	0	/	X	**sthbrx**
extsh	31 (0x1F)						rS					rA					///					1	1	1	0	0	1	1	0	1	0	0	X	**extsh**
extsh.	31 (0x1F)						rS					rA					///					1	1	1	0	0	1	1	0	1	0	1	X	**extsh.**
extsb	31 (0x1F)						rS					rA					///					1	1	1	0	1	1	1	0	1	0	0	X	**extsb**
extsb.	31 (0x1F)						rS					rA					///					1	1	1	0	1	1	1	0	1	0	1	X	**extsb.**
divwuo	31 (0x1F)						rD					rA					rB					1	1	1	1	0	0	1	0	1	1	0	X	**divwuo**
divwuo.	31 (0x1F)						rD					rA					rB					1	1	1	1	0	0	1	0	1	1	1	X	**divwuo.**
icbi	31 (0x1F)						///					rA					rB					1	1	1	1	0	1	0	1	1	0	/	X	**icbi**
divwo	31 (0x1F)						rD					rA					rB					1	1	1	1	1	0	1	0	1	1	0	X	**divwo**
divwo.	31 (0x1F)						rD					rA					rB					1	1	1	1	1	0	1	0	1	1	1	X	**divwo.**
dcbz	31 (0x1F)						///					rA					rB					1	1	1	1	1	1	0	1	1	0	/	X	**dcbz**
lwz	32 (0x20)						rD					rA					D																D	**lwz**
lwzu	33 (0x21)						rD					rA					D																D	**lwzu**
lbz	34(0x22)						rD					rA					D																D	**lbz**
lbzu	35(0x23)						rD					rA					D																D	**lbzu**
stw	36(0x24)						rS					rA					D																D	**stw**
stwu	37(0x25)						rS					rA					D																D	**stwu**
stb	38(0x26)						rS					rA					D																D	**stb**
stbu	39(0x27)						rS					rA					D																D	**stbu**
lhz	40(0x28)						rD					rA					D																D	**lhz**
lhzu	41(0x29)						rD					rA					D																D	**lhzu**
lha	42(0x2A)						rD					rA					D																D	**lha**
lhau	43(0x2B)						rD					rA					D																D	**lhau**
sth	44(0x2C)						rS					rA					D																D	**sth**
sthu	45(0x2D)						rS					rA					D																D	**sthu**
lmw	46(0x2E)						rD					rA					D																D	**lmw**
stmw	47(0x2F)						rS					rA					D																D	**stmw**
fres [6]	59(0x3B)						frD					///					frB					///					1	1	0	0	0	0	A	**fres**
fres. [6]	59(0x3B)						frD					///					frB					///					1	1	0	0	0	1	A	**fres.**
fsel [6]	63(0x3F)						frD					frA					frB					frC					1	0	1	1	1	0	A	**fsel**
fsel. [6]	63(0x3F)						frD					frA					frB					frC					1	0	1	1	1	1	A	**fsel.**

1. Supervisor-level instruction

A.3 Instructions sorted by mnemonic (binary)

Table 271 lists instructions in alphabetical order by mnemonic with binary values. This list also includes simplified mnemonics and their equivalents using standard mnemonics.

Table 271. Instructions sorted by mnemonic (binary)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
add	0	1	1	1	1	1	rD					rA					rB					0	1	0	0	0	0	1	0	1	0	0	X	**add**
add.	0	1	1	1	1	1	rD					rA					rB					0	1	0	0	0	0	1	0	1	0	1	X	**add.**
addc	0	1	1	1	1	1	rD					rA					rB					0	0	0	0	0	0	1	0	1	0	0	X	**addc**
addc.	0	1	1	1	1	1	rD					rA					rB					0	0	0	0	0	0	1	0	1	0	1	X	**addc.**
addco	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	0	0	1	0	1	0	0	X	**addco**
addco.	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	0	0	1	0	1	0	1	X	**addco.**
adde	0	1	1	1	1	1	rD					rA					rB					0	0	1	0	0	0	1	0	1	0	0	X	**adde**
adde.	0	1	1	1	1	1	rD					rA					rB					0	0	1	0	0	0	1	0	1	0	1	X	**adde.**
addeo	0	1	1	1	1	1	rD					rA					rB					1	0	1	0	0	0	1	0	1	0	0	X	**addeo**
addeo.	0	1	1	1	1	1	rD					rA					rB					1	0	1	0	0	0	1	0	1	0	1	X	**addeo.**
addi	0	0	1	1	1	0	rD					rA					SIMM																D	**addi**
addic	0	0	1	1	0	0	rD					rA					SIMM																D	**addic**
addic.	0	0	1	1	0	1	rD					rA					SIMM																D	**addic.**
addis	0	0	1	1	1	1	rD					rA					SIMM																D	**addis**
addme	0	1	1	1	1	1	rD					rA					///					0	0	1	1	1	0	1	0	1	0	0	X	**addme**
addme.	0	1	1	1	1	1	rD					rA					///					0	0	1	1	1	0	1	0	1	0	1	X	**addme.**
addmeo	0	1	1	1	1	1	rD					rA					///					1	0	1	1	1	0	1	0	1	0	0	X	**addmeo**
addmeo.	0	1	1	1	1	1	rD					rA					///					1	0	1	1	1	0	1	0	1	0	1	X	**addmeo.**
addo	0	1	1	1	1	1	rD					rA					rB					1	1	0	0	0	0	1	0	1	0	0	X	**addo**
addo.	0	1	1	1	1	1	rD					rA					rB					1	1	0	0	0	0	1	0	1	0	1	X	**addo.**
addze	0	1	1	1	1	1	rD					rA					///					0	0	1	1	0	0	1	0	1	0	0	X	**addze**
addze.	0	1	1	1	1	1	rD					rA					///					0	0	1	1	0	0	1	0	1	0	1	X	**addze.**
addzeo	0	1	1	1	1	1	rD					rA					///					1	0	1	1	0	0	1	0	1	0	0	X	**addzeo**
addzeo.	0	1	1	1	1	1	rD					rA					///					1	0	1	1	0	0	1	0	1	0	1	X	**addzeo.**
and	0	1	1	1	1	1	rS					rA					rB					0	0	0	0	0	1	1	1	0	0	0	X	**and**
and.	0	1	1	1	1	1	rS					rA					rB					0	0	0	0	0	1	1	1	0	0	1	X	**and.**
andc	0	1	1	1	1	1	rS					rA					rB					0	0	0	0	1	1	1	1	0	0	0	X	**andc**
andc.	0	1	1	1	1	1	rS					rA					rB					0	0	0	0	1	1	1	1	0	0	1	X	**andc.**
andi.	0	1	1	1	0	0	rS					rA					UIMM																D	**andi.**
andis.	0	1	1	1	0	1	rS					rA					UIMM																D	**andis.**
b	0	1	0	0	1	0	LI																								0	0	I	**b**
ba	0	1	0	0	1	0	LI																								1	0	I	**ba**
bc	0	1	0	0	0	0	BO					BI					BD														0	0	B	**bc**
bca	0	1	0	0	0	0	BO					BI					BD														1	0	B	**bca**
bcctr	0	1	0	0	1	1	BO					BI					///					1	0	0	0	0	1	0	0	0	0	0	XL	**bcctr**
bcctrl	0	1	0	0	1	1	BO					BI					///					1	0	0	0	0	1	0	0	0	0	1	XL	**bcctrl**

Table 271. Instructions sorted by mnemonic (binary) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
bcl	0	1	0	0	0	0	BO					BI					BD														0	1	B	**bcl**
bcla	0	1	0	0	0	0	BO					BI					BD														1	1	B	**bcla**
bclr	0	1	0	0	1	1	BO					BI					///					0	0	0	0	0	1	0	0	0	0	0	XL	**bclr**
bclrl	0	1	0	0	1	1	BO					BI					///					0	0	0	0	0	1	0	0	0	0	1	XL	**bclrl**
bctr	**bctr**(1) equivalent to **bcctr 20,0**																																	**bctr**
bctrl	**bctrl** [1] equivalent to **bcctrl 20,0**																																	**bctrl**
bdnz	**bdnz target** [1] equivalent to **bc 16,0,**target																																	**bdnz**
bdnza	**bdnza** target [1] equivalent to **bca 16,0,**target																																	**bdnza**
bdnzf	**bdnzf BI,**target equivalent to **bc 0,BI,**target																																	**bdnzf**
bdnzfa	**bdnzfa BI,**target equivalent to **bca 0,BI,**target																																	**bdnzfa**
bdnzfl	**bdnzfl BI,**target equivalent to **bcl 0,BI,**target																																	**bdnzfl**
bdnzfla	**bdnzfla BI,**target equivalent to **bcla 0,BI,**target																																	**bdnzfla**
bdnzflr	**bdnzflr BI** equivalent to **bclr 0,BI**																																	**bdnzflr**
bdnzflrl	**bdnzflrl BI** equivalent to **bclrl 0,BI**																																	**bdnzflrl**
bdnzl	**bdnzl** target [1] equivalent to **bcl 16,0,**target																																	**bdnzl**
bdnzla	**bdnzla** target [1] equivalent to **bcla 16,0,**target																																	**bdnzla**
bdnzlr	**bdnzlr BI** equivalent to **bclr 16,BI**																																	**bdnzlr**
bdnzlrl	**bdnzlrl** [1] equivalent to **bclrl 16,0**																																	**bdnzlrl**
bdnzt	**bdnzt BI,**target equivalent to **bc 8,BI,**target																																	**bdnzt**
bdnzta	**bdnzta BI,**target equivalent to **bca 8,BI,**target																																	**bdnzta**
bdnztl	**bdnztl BI,**target equivalent to **bcl 8,0,**target																																	**bdnztl**
bdnztla	**bdnztla BI,**target equivalent to **bcla 8,BI,**target																																	**bdnztla**
bdnztlr	**bdnztlr BI** equivalent to **bclr 8,BI**																																	**bdnztlr**
bdnztlr	**bdnztlr BI** equivalent to **bclr 8,BI**																																	**bdnztlr**
bdnztlrl	**bdnztlrl BI** equivalent to **bclrl 8,BI**																																	**bdnztlrl**
bdz	**bdz** target [1] equivalent to **bc 18,0,**target																																	**bdz**
bdza	**bdza** target [1] equivalent to **bca 18,0,**target																																	**bdza**
bdzf	**bdzf BI,**target equivalent to **bc 2,BI,**target																																	**bdzf**
bdzfa	**bdzfa BI,**target equivalent to **bca 2,BI,**target																																	**bdzfa**
bdzfl	**bdzfl BI,**target equivalent to **bcl 2,BI,**target																																	**bdzfl**
bdzfla	**bdzfla BI,**target equivalent to **bcla 2,BI,**target																																	**bdzfla**
bdzflr	**bdzflr BI** equivalent to **bclr 2,BI**																																	**bdzflr**
bdzflrl	**bdzflrl BI** equivalent to **bclrl 2,BI**																																	**bdzflrl**
bdzl	**bdzl** target [1] equivalent to **bcl 18,BI,**target																																	**bdzl**
bdzla	**bdzla** target [1] equivalent to **bcla 18,BI,**target																																	**bdzla**
bdzlr	**bdzlr** [1] equivalent to **bclr 18,0**																																	**bdzlr**
bdzlrl	**bdzlrl** [1] equivalent to **bclrl 18,0**																																	**bdzlrl**
bdzt	**bdzt BI,**target equivalent to **bc 10,BI,**target																																	**bdzt**
bdzta	**bdzta BI,**target equivalent to **bca 10,BI,**target																																	**bdzta**
bdztl	**bdztl BI,**target equivalent to **bcl 10,BI,**target																																	**bdztl**

Table 271. Instructions sorted by mnemonic (binary) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
bdztla	**bdztla BI,**target equivalent to **bcla 10,BI,**target																																	**bdztla**
bdztlrl	**bdztlrl BI** equivalent to **bclrl 10, BI**																																	**bdztlrl**
beq	**beq crS,**target equivalent to **bc 12,**BI(2),target																																	**beq**
beqa	**beqa crS,**target equivalent to **bca 12,**BI2,target																																	**beqa**
beqctr	**beqctr crS,**target equivalent to **bcctr 12,**BI2,target																																	**beqctr**
beqctrl	**beqctrl crS,**target equivalent to **bcctrl 12,**BI2,target																																	**beqctrl**
beql	**beql crS,**target equivalent to **bcl 12,**BI2,target																																	**beql**
beqla	**beqla crS,**target equivalent to **bcla 12,**BI2,target																																	**beqla**
beqlr	**beqlr crS,**target equivalent to **bclr 12,**BI2,target																																	**beqlr**
beqlrl	**beqlrl crS,**target equivalent to **bclrl 12,**BI2,target																																	**beqlrl**
bf	**bf BI,**target equivalent to **bc 4,BI,**target																																	**bf**
bfa	**bfa BI,**target equivalent to **bca 4,BI,**target																																	**bfa**
bfctr	**bfctr BI** equivalent to **bcctr 4,BI**																																	**bfctr**
bfctrl	**bfctrl BI** equivalent to **bcctrl 4,BI**																																	**bfctrl**
bfl	**bfl BI,**target equivalent to **bcl 4,BI,**target																																	**bfl**
bfla	**bfla BI,**target equivalent to **bcla 4,BI,**target																																	**bfla**
bflr	**bflr BI** equivalent to **bclr 4,BI**																																	**bflr**
bflrl	**bflrl BI** equivalent to **bclrl 4,BI**																																	**bflrl**
bge	**bge crS,**target equivalent to **bc 4,**BI(3),target																																	**bge**
bgea	**bgea crS,**target equivalent to **bca 4,**BI3,target																																	**bgea**
bgectr	**bgectr crS,**target equivalent to **bcctr 4,**BI3,target																																	**bgectr**
bgectrl	**bgectrl crS,**target equivalent to **bcctrl 4,**BI3,target																																	**bgectrl**
bgel	**bgel crS,**target equivalent to **bcl 4,**BI3,target																																	**bgel**
bgela	**bgela crS,**target equivalent to **bcla 4,**BI3,target																																	**bgela**
bgelr	**bgelr crS,**target equivalent to **bclr 4,**BI3,target																																	**bgelr**
bgelrl	**bgelrl crS,**target equivalent to **bclrl 4,**BI3,target																																	**bgelrl**
bgt	**bgt crS,**target equivalent to **bc 12,**BI(4),target																																	**bgt**
bgta	**bgta crS,**target equivalent to **bca 12,**BI4,target																																	**bgta**
bgtctr	**bgtctr crS,**target equivalent to **bcctr 12,**BI4,target																																	**bgtctr**
bgtctrl	**bgtctrl crS,**target equivalent to **bcctrl 12,**BI4,target																																	**bgtctrl**
bgtl	**bgtl crS,**target equivalent to **bcl 12,**BI4,target																																	**bgtl**
bgtla	**bgtla crS,**target equivalent to **bcla 12,**BI4,target																																	**bgtla**
bgtlr	**bgtlr crS,**target equivalent to **bclr 12,**BI4,target																																	**bgtlr**
bgtlrl	**bgtlrl crS,**target equivalent to **bclrl 12,**BI4,target																																	**bgtlrl**
bl	0	1	0	0	1	0	LI																							0	1	I	**bl**
bla	0	1	0	0	1	0	LI																							1	1	I	**bla**
ble	**ble crS,**target equivalent to **bc 4,**BI4,target																																	**ble**
blea	**blea crS,**target equivalent to **bca 4,**BI4,target																																	**blea**
blectr	**blectr crS,**target equivalent to **bcctr 4,**BI4,target																																	**blectr**
blectrl	**blectrl crS,**target equivalent to **bcctrl 4,**BI4,target																																	**blectrl**

Table 271. Instructions sorted by mnemonic (binary) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
blel				**blel crS**,target					equivalent to				**bcl 4,**BI[4],target																					**blel**
blela				**blela crS**,target					equivalent to				**bcla 4,**BI[4],target																					**blela**
blelr				**blelr crS**,target					equivalent to				**bclr 4,**BI[4],target																					**blelr**
blelrl				**blelrl crS**,target					equivalent to				**bclrl 4,**BI[4],target																					**blelrl**
blr				**blr** [1]					equivalent to				**bclr 20,0**																					**blr**
blrl				**blrl** [1]					equivalent to				**bclrl 20,0**																					**blrl**
blt				**blt crS**,target					equivalent to				**bc 12,**BI,target																					**blt**
blta				**blta crS**,target					equivalent to				**bca 12,**BI[3],target																					**blta**
bltctr				**bltctr crS**,target					equivalent to				**bcctr 12,**BI[3],target																					**bltctr**
bltctrl				**bltctrl crS**,target					equivalent to				**bcctrl 12,**BI[3],target																					**bltctrl**
bltl				**bltl crS**,target					equivalent to				**bcl 12,**BI[3],target																					**bltl**
bltla				**bltla crS**,target					equivalent to				**bcla 12,**BI[3],target																					**bltla**
bltlr				**bltlr crS**,target					equivalent to				**bclr 12,**BI[3],target																					**bltlr**
bltlrl				**bltlrl crS**,target					equivalent to				**bclrl 12,**BI[3],target																					**bltlrl**
bne				**bne crS**,target					equivalent to				**bc 4,**BI[3],target																					**bne**
bnea				**bnea crS**,target					equivalent to				**bca 4,**BI[3],target																					**bnea**
bnectr				**bnectr crS**,target					equivalent to				**bcctr 4,**BI[3],target																					**bnectr**
bnectrl				**bnectrl crS**,target					equivalent to				**bcctrl 4,**BI[3],target																					**bnectrl**
bnel				**bnel crS**,target					equivalent to				**bcl 4,**BI[3],target																					**bnel**
bnela				**bnela crS**,target					equivalent to				**bcla 4,**BI[3],target																					**bnela**
bnelr				**bnelr crS**,target					equivalent to				**bclr 4,**BI[3],target																					**bnelr**
bnelrl				**bnelrl crS**,target					equivalent to				**bclrl 4,**BI[3],target																					**bnelrl**
bng				**bng crS**,target					equivalent to				**bc 4,**BI[4],target																					**bng**
bnga				**bnga crS**,target					equivalent to				**bca 4,**BI[4],target																					**bnga**
bngctr				**bngctr crS**,target					equivalent to				**bcctr 4,**BI[4],target																					**bngctr**
bngctrl				**bngctrl crS**,target					equivalent to				**bcctrl 4,**BI[4],target																					**bngctrl**
bngl				**bngl crS**,target					equivalent to				**bcl 4,**BI[4],target																					**bngl**
bngla				**bngla crS**,target					equivalent to				**bcla 4,**BI[4],target																					**bngla**
bnglr				**bnglr crS**,target					equivalent to				**bclr 4,**BI[4],target																					**bnglr**
bnglrl				**bnglrl crS**,target					equivalent to				**bclrl 4,**BI[4],target																					**bnglrl**
bnl				**bnl crS**,target					equivalent to				**bc 4,**BI[3],target																					**bnl**
bnla				**bnla crS**,target					equivalent to				**bca 4,**BI[3],target																					**bnla**
bnlctr				**bnlctr crS**,target					equivalent to				**bcctr 4,**BI[3],target																					**bnlctr**
bnlctrl				**bnlctrl crS**,target					equivalent to				**bcctrl 4,**BI[3],target																					**bnlctrl**
bnll				**bnll crS**,target					equivalent to				**bcl 4,**BI[3],target																					**bnll**
bnlla				**bnlla crS**,target					equivalent to				**bcla 4,**BI[3],target																					**bnlla**
bnllr				**bnllr crS**,target					equivalent to				**bclr 4,**BI[3],target																					**bnllr**
bnllrl				**bnllrl crS**,target					equivalent to				**bclrl 4,**BI[3],target																					**bnllrl**
bns				**bns crS**,target					equivalent to				**bc 4,**BI[(5)],target																					**bns**
bnsa				**bnsa crS**,target					equivalent to				**bca 4,**BI[5],target																					**bnsa**

Table 271. Instructions sorted by mnemonic (binary) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
bnsctr	**bnsctr crS**,target equivalent to **bcctr 4,BI**[5],target																																	**bnsctr**
bnsctrl	**bnsctrl crS**,target equivalent to **bcctrl 4,BI**[5],target																																	**bnsctrl**
bnsl	**bnsl crS**,target equivalent to **bcl 4,BI**[5],target																																	**bnsl**
bnsla	**bnsla crS**,target equivalent to **bcla 4,BI**[5],target																																	**bnsla**
bnslr	**bnslr crS**,target equivalent to **bclr 4,BI**[5],target																																	**bnslr**
bnslrl	**bnslrl crS**,target equivalent to **bclrl 4,BI**[5],target																																	**bnslrl**
bnu	**bnu crS**,target equivalent to **bc 4,BI**[5],target																																	**bnu**
bnua	**bnua crS**,target equivalent to **bca 4,BI**[5],target																																	**bnua**
bnuctr	**bnuctr crS**,target equivalent to **bcctr 4,BI**[5],target																																	**bnuctr**
bnuctrl	**bnuctrl crS**,target equivalent to **bcctrl 4,BI**[5],target																																	**bnuctrl**
bnul	**bnul crS**,target equivalent to **bcl 4,BI**[5],target																																	**bnul**
bnula	**bnula crS**,target equivalent to **bcla 4,BI**[5],target																																	**bnula**
bnulr	**bnulr crS**,target equivalent to **bclr 4,BI**[5],target																																	**bnulr**
bnulrl	**bnulrl crS**,target equivalent to **bclrl 4,BI**[5],target																																	**bnulrl**
brinc	0	0	0	1	0	0	rD						rA					rB				0	1	0	0	0	0	0	1	1	1	1	EVX	**brinc**
bso	**bso crS**,target equivalent to **bc 12,BI**[5],target																																	**bso**
bsoa	**bsoa crS**,target equivalent to **bca 12,BI**[5],target																																	**bsoa**
bsoctr	**bsoctr crS**,target equivalent to **bcctr 12,BI**[5],target																																	**bsoctr**
bsoctrl	**bsoctrl crS**,target equivalent to **bcctrl 12,BI**[5],target																																	**bsoctrl**
bsol	**bsol crS**,target equivalent to **bcl 12,BI**[5],target																																	**bsol**
bsola	**bsola crS**,target equivalent to **bcla 12,BI**[5],target																																	**bsola**
bsolr	**bsolr crS**,target equivalent to **bclr 12,BI**[5],target																																	**bsolr**
bsolrl	**bsolrl crS**,target equivalent to **bclrl 12,BI**[5],target																																	**bsolrl**
bt	**bt BI**,target equivalent to **bc 12,BI**,target																																	**bt**
bta	**bta BI**,target equivalent to **bca 12,BI**,target																																	**bta**
btctr	**btctr BI** equivalent to **bcctr 12,BI**																																	**btctr**
btctrl	**btctrl BI** equivalent to **bcctrl 12,BI**																																	**btctrl**
btl	**btl BI**,target equivalent to **bcl 12,BI**,target																																	**btl**
btla	**btla BI**,target equivalent to **bcla 12,BI**,target																																	**btla**
btlr	**btlr BI** equivalent to **bclr 12,BI**																																	**btlr**
btlrl	**btlrl BI** equivalent to **bclrl 12,BI**																																	**btlrl**
bun	**bun crS**,target equivalent to **bc 12,BI**[5],target																																	**bun**
buna	**buna crS**,target equivalent to **bca 12,BI**[5],target																																	**buna**
bunctr	**bunctr crS**,target equivalent to **bcctr 12,BI**[5],target																																	**bunctr**
bunctrl	**bunctrl crS**,target equivalent to **bcctrl 12,BI**[5],target																																	**bunctrl**
bunl	**bunl crS**,target equivalent to **bcl 12,BI**[5],target																																	**bunl**
bunla	**bunla crS**,target equivalent to **bcla 12,BI**[5],target																																	**bunla**
bunlr	**bunlr crS**,target equivalent to **bclr 12,BI**[5],target																																	**bunlr**
bunlrl	**bunlrl crS**,target equivalent to **bclrl 12,BI**[5],target																																	**bunlrl**
clrlslwi	**clrlslwi rA,rS**,*b*,*n* (*n* ≤ *b* ≤ 31) equivalent to **rlwinm rA,rS**,*n*,*b* − *n*,31 − *n*																																	**clrlslwi**

Table 271. Instructions sorted by mnemonic (binary) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
clrlwi	**clrlwi** rA,**r**S,*n* (n < 32) equivalent to **rlwinm** rA,**r**S,**0**,*n*,**31**																																	**clrlwi**
clrrwi	**clrrwi** rA,**r**S,*n* (n < 32) equivalent to **rlwinm** rA,**r**S,**0,0,31** – *n*																																	**clrrwi**
cmp	0	1	1	1	1	1	crfD			/	L	rA					rB					0	0	0	0	0	0	0	0	0	/	X	**cmp**
cmpi	0	0	1	0	1	1	crfD			/	L	rA					SIMM																D	**cmpi**
cmpl	0	1	1	1	1	1	/	L	rA						rB			///				0	0	0	0	1	0	0	0	0	/	X	**cmpl**
cmpli	0	0	1	0	1	0	crfD			/	L	rA					UIMM																D	**cmpli**
cmplw	**cmplw** **cr**D,**r**A,**r**B equivalent to **cmpl** **cr**D,**0**,**r**A,**r**B																																	**cmplw**
cmplwi	**cmplwi** **cr**D,**r**A,**UIMM** equivalent to **cmpli** **cr**D,**0**,**r**A,UIMM																																	**cmplwi**
cmpw	**cmpw** **cr**D,**r**A,**r**B equivalent to **cmp** **cr**D,**0**,**r**A,**r**B																																	**cmpw**
cmpwi	**cmpwi** **cr**D,**r**A,**SIMM** equivalent to **cmpi** **cr**D,**0**,**r**A,SIMM																																	**cmpwi**
cntlzw	0	1	1	1	1	1	rS					rA					///					0	0	0	0	0	1	1	0	1	0	0	X	**cntlzw**
cntlzw.	0	1	1	1	1	1	rS					rA					///					0	0	0	0	0	1	1	0	1	0	1	X	**cntlzw.**
crand	0	1	0	0	1	1	crbD					crbA					crbB					0	1	0	0	0	0	0	0	0	1	/	XL	**crand**
crandc	0	1	0	0	1	1	crbD					crbA					crbB					0	0	1	0	0	0	0	0	0	1	/	XL	**crandc**
crclr	**crclr** **bx** equivalent to **crxor** bx,bx,bx																																	**crclr**
creqv	0	1	0	0	1	1	crbD					crbA					crbB					0	1	0	0	1	0	0	0	0	1	/	XL	**creqv**
crmove	**crmove** **bx**,**by** equivalent to **cror** bx,by,by																																	**crmove**
crnand	0	1	0	0	1	1	crbD					crbA					crbB					0	0	1	1	1	0	0	0	0	1	/	XL	**crnand**
crnor	0	1	0	0	1	1	crbD					crbA					crbB					0	0	0	0	1	0	0	0	0	1	/	XL	**crnor**
crnot	**crnot** **bx**,**by** equivalent to **crnor** bx,by,by																																	**crnot**
cror	0	1	0	0	1	1	crbD					crbA					crbB					0	1	1	1	0	0	0	0	0	1	/	XL	**cror**
crorc	0	1	0	0	1	1	crbD					crbA					crbB					0	1	1	0	1	0	0	0	0	1	/	XL	**crorc**
crset	**crset** **bx** equivalent to **creqv** bx,bx,bx																																	**crset**
crxor	0	1	0	0	1	1	crbD					crbA					crbB					0	0	1	1	0	0	0	0	0	1	/	XL	**crxor**
dcba (6)	0	1	1	1	1	1	///					rA					rB					1	0	1	1	1	1	0	1	1	0	/	X	**dcba**
dcbf	0	1	1	1	1	1	///					rA					rB					0	0	0	1	0	1	0	1	1	0	/	X	**dcbf**
dcbi (7)	0	1	1	1	1	1	///					rA					rB					0	1	1	1	0	1	0	1	1	0	/	X	**dcbi**
dcblc	0	1	1	1	1	1	CT					rA					rB					0	1	1	0	0	0	0	1	1	0	0	X	**dcblc**
dcbst	0	1	1	1	1	1	///					rA					rB					0	0	0	0	1	1	0	1	1	0	/	X	**dcbst**
dcbt	0	1	1	1	1	1	CT					rA					rB					0	1	0	0	0	1	0	1	1	0	/	X	**dcbt**
dcbtls	0	1	1	1	1	1	CT					rA					rB					0	0	1	0	1	0	0	1	1	0	0	X	**dcbtls**
dcbtst	0	1	1	1	1	1	CT					rA					rB					0	0	1	1	1	1	0	1	1	0	/	X	**dcbtst**
dcbtstls	0	1	1	1	1	1	CT					rA					rB					0	0	1	0	0	0	0	1	1	0	0	X	**dcbtstls**
dcbz	0	1	1	1	1	1	///					rA					rB					1	1	1	1	1	1	0	1	1	0	/	X	**dcbz**
divw	0	1	1	1	1	1	rD					rA					rB					0	1	1	1	1	0	1	0	1	1	0	X	**divw**
divw.	0	1	1	1	1	1	rD					rA					rB					0	1	1	1	1	0	1	0	1	1	1	X	**divw.**
divwo	0	1	1	1	1	1	rD					rA					rB					1	1	1	1	1	0	1	0	1	1	0	X	**divwo**
divwo.	0	1	1	1	1	1	rD					rA					rB					1	1	1	1	1	0	1	0	1	1	1	X	**divwo.**
divwu	0	1	1	1	1	1	rD					rA					rB					0	1	1	1	0	0	1	0	1	1	0	X	**divwu**
divwu.	0	1	1	1	1	1	rD					rA					rB					0	1	1	1	0	0	1	0	1	1	1	X	**divwu.**

Table 271. Instructions sorted by mnemonic (binary) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
divwuo	0	1	1	1	1	1	rD					rA					rB					1	1	1	1	0	0	1	0	1	1	0	X	**divwuo**
divwuo.	0	1	1	1	1	1	rD					rA					rB					1	1	1	1	0	0	1	0	1	1	1	X	**divwuo.**
dss	**dss STRM** equivalent to **dss** STRM,**0**																																	**dss**
efdabs	0	0	0	1	0	0	rD					rA					///					0	1	0	1	1	1	0	0	1	0	0	EFX	**efdabs**
efdadd	0	0	0	1	0	0	rD					rA					rB					0	1	0	1	1	1	0	0	0	0	0	EFX	**efdadd**
efdcfs	0	0	0	1	0	0	rD					0	0	0	0	0	rB					0	1	0	1	1	1	0	1	1	1	1	EFX	**efdcfs**
efdcfsf	0	0	0	1	0	0	rD					///					rB					0	1	0	1	1	1	1	0	0	1	1	EFX	**efdcfsf**
efdcfsi	0	0	0	1	0	0	rD					///					rB					0	1	0	1	1	1	1	0	0	0	1	EFX	**efdcfsi**
efdcfuf	0	0	0	1	0	0	rD					///					rB					0	1	0	1	1	1	1	0	0	1	0	EFX	**efdcfuf**
efdcfui	0	0	0	1	0	0	rD					///					rB					0	1	0	1	1	1	1	0	0	0	0	EFX	**efdcfui**
efdcmpeq	0	0	0	1	0	0	crfD			/	/	rA					rB					0	1	0	1	1	1	0	1	1	1	0	EFX	**efdcmpeq**
efdcmpgt	0	0	0	1	0	0	crfD			/	/	rA					rB					0	1	0	1	1	1	0	1	1	0	0	EFX	**efdcmpgt**
efdcmplt	0	0	0	1	0	0	crfD			/	/	rA					rB					0	1	0	1	1	1	0	1	1	0	1	EFX	**efdcmplt**
efdctsf	0	0	0	1	0	0	rD					///					rB					0	1	0	1	1	1	1	0	1	1	1	EFX	**efdctsf**
efdctsi	0	0	0	1	0	0	rD					///					rB					0	1	0	1	1	1	1	0	1	0	1	EFX	**efdctsi**
efdctsiz	0	0	0	1	0	0	rD					///					rB					0	1	0	1	1	1	1	1	0	1	0	EFX	**efdctsiz**
efdctuf	0	0	0	1	0	0	rD					///					rB					0	1	0	1	1	1	1	0	1	1	0	EFX	**efdctuf**
efdctui	0	0	0	1	0	0	rD					///					rB					0	1	0	1	1	1	1	0	1	0	0	EFX	**efdctui**
efdctuiz	0	0	0	1	0	0	rD					///					rB					0	1	0	1	1	1	1	1	0	0	0	EFX	**efdctuiz**
efddiv	0	0	0	1	0	0	rD					rA					rB					0	1	0	1	1	1	0	1	0	0	1	EFX	**efddiv**
efdmul	0	0	0	1	0	0	rD					rA					rB					0	1	0	1	1	1	0	1	0	0	0	EFX	**efdmul**
efdnabs	0	0	0	1	0	0	rD					rA					///					0	1	0	1	1	1	0	0	1	0	1	EFX	**efdnabs**
efdneg	0	0	0	1	0	0	rD					rA					///					0	1	0	1	1	1	0	0	1	1	0	EFX	**efdneg**
efdsub	0	0	0	1	0	0	rD					rA					rB					0	1	0	1	1	1	0	0	0	0	1	EFX	**efdsub**
efdtsteq	0	0	0	1	0	0	crfD			/	/	rA					rB					0	1	0	1	1	1	1	1	1	1	0	EFX	**efdtsteq**
efdtstgt	0	0	0	1	0	0	crfD			/	/	rA					rB					0	1	0	1	1	1	1	1	1	0	0	EFX	**efdtstgt**
efdtstlt	0	0	0	1	0	0	crfD			/	/	rA					rB					0	1	0	1	1	1	1	1	1	0	1	EFX	**efdtstlt**
efsabs	0	0	0	1	0	0	rD					rA					///					0	1	0	1	1	0	0	0	1	0	0	EFX	**efsabs**
efsadd	0	0	0	1	0	0	rD					rA					rB					0	1	0	1	1	0	0	0	0	0	0	EFX	**efsadd**
efscfd	0	0	0	1	0	0	rD					0	0	0	0	0	rB					0	1	0	1	1	0	0	1	1	1	1	EFX	**efscfd**
efscfsf	0	0	0	1	0	0	rD					///					rB					0	1	0	1	1	0	1	0	0	1	1	EFX	**efscfsf**
efscfsi	0	0	0	1	0	0	rD					///					rB					0	1	0	1	1	0	1	0	0	0	1	EFX	**efscfsi**
efscfuf	0	0	0	1	0	0	rD					///					rB					0	1	0	1	1	0	1	0	0	1	0	EFX	**efscfuf**
efscfui	0	0	0	1	0	0	rD					///					rB					0	1	0	1	1	0	1	0	0	0	0	EFX	**efscfui**
efscmpeq	0	0	0	1	0	0	crfD			/	/	rA					rB					0	1	0	1	1	0	0	1	1	1	0	EFX	**efscmpeq**
efscmpgt	0	0	0	1	0	0	crfD			/	/	rA					rB					0	1	0	1	1	0	0	1	1	0	0	EFX	**efscmpgt**
efscmplt	0	0	0	1	0	0	crfD			/	/	rA					rB					0	1	0	1	1	0	0	1	1	0	1	EFX	**efscmplt**
efsctsf	0	0	0	1	0	0	rD					///					rB					0	1	0	1	1	0	1	0	1	1	1	EFX	**efsctsf**
efsctsi	0	0	0	1	0	0	rD					///					rB					0	1	0	1	1	0	1	0	1	0	1	EFX	**efsctsi**
efsctsiz	0	0	0	1	0	0	rD					///					rB					0	1	0	1	1	0	1	1	0	1	0	EFX	**efsctsiz**

Table 271. Instructions sorted by mnemonic (binary) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
efsctuf	0	0	0	1	0	0	rD					///					rB					0	1	0	1	1	0	1	0	1	1	0	EFX	efsctuf
efsctui	0	0	0	1	0	0	rD					///					rB					0	1	0	1	1	0	1	0	1	0	0	EFX	efsctui
efsctuiz	0	0	0	1	0	0	rD					///					rB					0	1	0	1	1	0	1	1	0	0	0	EFX	efsctuiz
efsdiv	0	0	0	1	0	0	rD					rA					rB					0	1	0	1	1	0	0	1	0	0	1	EFX	efsdiv
efsmul	0	0	0	1	0	0	rD					rA					rB					0	1	0	1	1	0	0	1	0	0	0	EFX	efsmul
efsnabs	0	0	0	1	0	0	rD					rA					///					0	1	0	1	1	0	0	0	1	0	1	EFX	efsnabs
efsneg	0	0	0	1	0	0	rD					rA					///					0	1	0	1	1	0	0	0	1	1	0	EFX	efsneg
efssub	0	0	0	1	0	0	rD					rA					rB					0	1	0	1	1	0	0	0	0	0	1	EFX	efssub
efststeq	0	0	0	1	0	0	crfD			/	/	rA					rB					0	1	0	1	1	0	1	1	1	1	0	EFX	efststeq
efststgt	0	0	0	1	0	0	crfD			/	/	rA					rB					0	1	0	1	1	0	1	1	1	0	0	EFX	efststgt
efststlt	0	0	0	1	0	0	crfD			/	/	rA					rB					0	1	0	1	1	0	1	1	1	0	1	EFX	efststlt
eqv	0	1	1	1	1	1	rD					rA					rB					0	1	0	0	0	1	1	1	0	0	0	X	eqv
eqv.	0	1	1	1	1	1	rD					rA					rB					0	1	0	0	0	1	1	1	0	0	1	X	eqv.
evabs	0	1	1	1	1	1	rD					rA					///					0	1	0	0	0	0	0	1	0	0	0	EVX	evabs
evaddiw	0	1	1	1	1	1	rD					UIMM					rB					0	1	0	0	0	0	0	0	0	1	0	EVX	evaddiw
evaddsmiaaw	0	1	1	1	1	1	rD					rA					///					1	0	0	1	1	0	0	1	0	0	1	EVX	evaddsmiaaw
evaddssiaaw	0	1	1	1	1	1	rD					rA					///					1	0	0	1	1	0	0	0	0	0	1	EVX	evaddssiaaw
evaddumiaaw	0	1	1	1	1	1	rD					rA					///					1	0	0	1	1	0	0	1	0	0	0	EVX	evaddumiaaw
evaddusiaaw	0	1	1	1	1	1	rD					rA					///					1	0	0	1	1	0	0	0	0	0	0	EVX	evaddusiaaw
evaddw	0	1	1	1	1	1	rD					rA					rB					0	1	0	0	0	0	0	0	0	0	0	EVX	evaddw
evand	0	1	1	1	1	1	rD					rA					rB					0	1	0	0	0	0	1	0	0	0	1	EVX	evand
evandc	0	1	1	1	1	1	rD					rA					rB					0	1	0	0	0	0	1	0	0	1	0	EVX	evandc
evcmpeq	0	1	1	1	1	1	crfD			/	/	rA					rB					0	1	0	0	0	1	1	0	1	0	0	EVX	evcmpeq
evcmpgts	0	1	1	1	1	1	crfD			/	/	rA					rB					0	1	0	0	0	1	1	0	0	0	1	EVX	evcmpgts
evcmpgtu	0	1	1	1	1	1	crfD			/	/	rA					rB					0	1	0	0	0	1	1	0	0	0	0	EVX	evcmpgtu
evcmplts	0	1	1	1	1	1	crfD			/	/	rA					rB					0	1	0	0	0	1	1	0	0	1	1	EVX	evcmplts
evcmpltu	0	1	1	1	1	1	crfD			/	/	rA					rB					0	1	0	0	0	1	1	0	0	1	0	EVX	evcmpltu
evcntlsw	0	1	1	1	1	1	rD					rA					///					0	1	0	0	0	0	0	1	1	1	0	EVX	evcntlsw
evcntlzw	0	1	1	1	1	1	rD					rA					///					0	1	0	0	0	0	0	1	1	0	1	EVX	evcntlzw
evdivws	0	1	1	1	1	1	rD					rA					rB					1	0	0	1	1	0	0	0	1	1	0	EVX	evdivws
evdivwu	0	1	1	1	1	1	rD					rA					rB					1	0	0	1	1	0	0	0	1	1	1	EVX	evdivwu
eveqv	0	1	1	1	1	1	rD					rA					rB					0	1	0	0	0	0	1	1	0	0	1	EVX	eveqv
evextsb	0	1	1	1	1	1	rD					rA					///					0	1	0	0	0	0	0	1	0	1	0	EVX	evextsb
evextsh	0	1	1	1	1	1	rD					rA					///					0	1	0	0	0	0	0	1	0	1	1	EVX	evextsh
evfsabs	0	1	1	1	1	1	rD					rA					///					0	1	0	1	0	0	0	0	1	0	0	EVX	evfsabs
evfsadd	0	1	1	1	1	1	rD					rA					rB					0	1	0	1	0	0	0	0	0	0	0	EVX	evfsadd
evfscfsf	0	1	1	1	1	1	rD					///					rB					0	1	0	1	0	0	1	0	0	1	1	EVX	evfscfsf
evfscfsi	0	1	1	1	1	1	rD					///					rB					0	1	0	1	0	0	1	0	0	0	1	EVX	evfscfsi

Table 271. Instructions sorted by mnemonic (binary) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
evfscfuf	0	1	1	1	1	1	rD					///					rB					0	1	0	1	0	0	1	0	0	1	0	EVX	**evfscfuf**
evfscfui	0	1	1	1	1	1	rD					///					rB					0	1	0	1	0	0	1	0	0	0	0	EVX	**evfscfui**
evfscmpeq	0	1	1	1	1	1	crfD			/	/	rA					rB					0	1	0	1	0	0	0	1	1	1	0	EVX	**evfscmpeq**
evfscmpgt	0	1	1	1	1	1	crfD			/	/	rA					rB					0	1	0	1	0	0	0	1	1	0	0	EVX	**evfscmpgt**
evfscmplt	0	1	1	1	1	1	crfD			/	/	rA					rB					0	1	0	1	0	0	0	1	1	0	1	EVX	**evfscmplt**
evfsctsf	0	1	1	1	1	1	rD					///					rB					0	1	0	1	0	0	1	0	1	1	1	EVX	**evfsctsf**
evfsctsi	0	1	1	1	1	1	rD					///					rB					0	1	0	1	0	0	1	0	1	0	1	EVX	**evfsctsi**
evfsctsiz	0	1	1	1	1	1	rD					///					rB					0	1	0	1	0	0	1	1	0	1	0	EVX	**evfsctsiz**
evfsctuf	0	1	1	1	1	1	rD					///					rB					0	1	0	1	0	0	1	0	1	1	0	EVX	**evfsctuf**
evfsctui	0	1	1	1	1	1	rD					///					rB					0	1	0	1	0	0	1	0	1	0	0	EVX	**evfsctui**
evfsctuiz	0	1	1	1	1	1	rD					///					rB					0	1	0	1	0	0	1	1	0	0	0	EVX	**evfsctuiz**
evfsdiv	0	1	1	1	1	1	rD					rA					rB					0	1	0	1	0	0	0	1	0	0	1	EVX	**evfsdiv**
evfsmul	0	1	1	1	1	1	rD					rA					rB					0	1	0	1	0	0	0	1	0	0	0	EVX	**evfsmul**
evfsnabs	0	1	1	1	1	1	rD					rA					///					0	1	0	1	0	0	0	0	1	0	1	EVX	**evfsnabs**
evfsneg	0	1	1	1	1	1	rD					rA					///					0	1	0	1	0	0	0	0	1	1	0	EVX	**evfsneg**
evfssub	0	1	1	1	1	1	rD					rA					rB					0	1	0	1	0	0	0	0	0	0	1	EVX	**evfssub**
evfststeq	0	1	1	1	1	1	crfD			/	/	rA					rB					0	1	0	1	0	0	1	1	1	1	0	EVX	**evfststeq**
evfststgt	0	1	1	1	1	1	crfD			/	/	rA					rB					0	1	0	1	0	0	1	1	1	0	0	EVX	**evfststgt**
evfststlt	0	1	1	1	1	1	crfD			/	/	rA					rB					0	1	0	1	0	0	1	1	1	0	1	EVX	**evfststlt**
evldd	0	1	1	1	1	1	rD					rA					UIMM(8)					0	1	1	0	0	0	0	0	0	0	1	EVX	**evldd**
evlddx	0	1	1	1	1	1	rD					rA					rB					0	1	1	0	0	0	0	0	0	0	0	EVX	**evlddx**
evldh	0	1	1	1	1	1	rD					rA					UIMM 8					0	1	1	0	0	0	0	0	1	0	1	EVX	**evldh**
evldhx	0	1	1	1	1	1	rD					rA					rB					0	1	1	0	0	0	0	0	1	0	0	EVX	**evldhx**
evldw	0	1	1	1	1	1	rD					rA					UIMM 8					0	1	1	0	0	0	0	0	0	1	1	EVX	**evldw**
evldwx	0	1	1	1	1	1	rD					rA					rB					0	1	1	0	0	0	0	0	0	1	0	EVX	**evldwx**
evlhhesplat	0	1	1	1	1	1	rD					rA					UIMM 9					0	1	1	0	0	0	0	1	0	0	1	EVX	**evlhhesplat**
evlhhesplatx	0	1	1	1	1	1	rD					rA					rB					0	1	1	0	0	0	0	1	0	0	0	EVX	**evlhhesplatx**
evlhhossplat	0	1	1	1	1	1	rD					rA					UIMM 9					0	1	1	0	0	0	0	1	1	1	1	EVX	**evlhhossplat**
evlhhossplatx	0	1	1	1	1	1	rD					rA					rB					0	1	1	0	0	0	0	1	1	1	0	EVX	**evlhhossplatx**
evlhhousplat	0	1	1	1	1	1	rD					rA					UIMM 9					0	1	1	0	0	0	0	1	1	0	1	EVX	**evlhhousplat**
evlhhousplatx	0	1	1	1	1	1	rD					rA					rB					0	1	1	0	0	0	0	1	1	0	0	EVX	**evlhhousplatx**
evlwhe	0	1	1	1	1	1	rD					rA					UIMM 9					0	1	1	0	0	0	1	0	0	0	1	EVX	**evlwhe**
evlwhex	0	1	1	1	1	1	rD					rA					rB					0	1	1	0	0	0	1	0	0	0	0	EVX	**evlwhex**
evlwhos	0	1	1	1	1	1	rD					rA					UIMM 10					0	1	1	0	0	0	1	0	1	1	1	EVX	**evlwhos**
evlwhosx	0	1	1	1	1	1	rD					rA					rB					0	1	1	0	0	0	1	0	1	1	0	EVX	**evlwhosx**

Table 271. Instructions sorted by mnemonic (binary) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
evlwhou	0	1	1	1	1	1	rD					rA					UIMM [10]					0	1	1	0	0	0	1	0	1	0	1	EVX	evlwhou
evlwhoux	0	1	1	1	1	1	rD					rA					rB					0	1	1	0	0	0	1	0	1	0	0	EVX	evlwhoux
evlwhsplat	0	1	1	1	1	1	rD					rA					UIMM [10]					0	1	1	0	0	0	1	1	1	0	1	EVX	evlwhsplat
evlwhsplatx	0	1	1	1	1	1	rD					rA					rB					0	1	1	0	0	0	1	1	1	0	0	EVX	evlwhsplatx
evlwwsplat	0	1	1	1	1	1	rD					rA					UIMM [10]					0	1	1	0	0	0	1	1	0	0	1	EVX	evlwwsplat
evlwwsplatx	0	1	1	1	1	1	rD					rA					rB					0	1	1	0	0	0	1	1	0	0	0	EVX	evlwwsplatx
evmergehi	0	1	1	1	1	1	rD					rA					rB					0	1	0	0	0	1	0	1	1	0	0	EVX	evmergehi
evmergehilo	0	1	1	1	1	1	rD					rA					rB					0	1	0	0	0	1	0	1	1	1	0	EVX	evmergehilo
evmergelo	0	1	1	1	1	1	rD					rA					rB					0	1	0	0	0	1	0	1	1	0	1	EVX	evmergelo
evmergelohi	0	1	1	1	1	1	rD					rA					rB					0	1	0	0	0	1	0	1	1	1	1	EVX	evmergelohi
evmhegsmfaa	0	1	1	1	1	1	rD					rA					rB					1	0	1	0	0	1	0	1	0	1	1	EVX	evmhegsmfaa
evmhegsmfan	0	1	1	1	1	1	rD					rA					rB					1	0	1	1	0	1	0	1	0	1	1	EVX	evmhegsmfan
evmhegsmiaa	0	1	1	1	1	1	rD					rA					rB					1	0	1	0	0	1	0	1	0	0	1	EVX	evmhegsmiaa
evmhegsmian	0	1	1	1	1	1	rD					rA					rB					1	0	1	1	0	1	0	1	0	0	1	EVX	evmhegsmian
evmhegumiaa	0	1	1	1	1	1	rD					rA					rB					1	0	1	0	0	1	0	1	0	0	0	EVX	evmhegumiaa
evmhegumian	0	1	1	1	1	1	rD					rA					rB					1	0	1	1	0	1	0	1	0	0	0	EVX	evmhegumian
evmhesmf	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	0	0	0	1	0	1	1	EVX	evmhesmf
evmhesmfa	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	0	1	0	1	0	1	1	EVX	evmhesmfa
evmhesmfaaw	0	1	1	1	1	1	rD					rA					rB					1	0	1	0	0	0	0	1	0	1	1	EVX	evmhesmfaaw
evmhesmfanw	0	1	1	1	1	1	rD					rA					rB					1	0	1	1	0	0	0	1	0	1	1	EVX	evmhesmfanw
evmhesmi	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	0	0	0	1	0	0	1	EVX	evmhesmi
evmhesmia	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	0	1	0	1	0	0	1	EVX	evmhesmia
evmhesmiaaw	0	1	1	1	1	1	rD					rA					rB					1	0	1	0	0	0	0	1	0	0	1	EVX	evmhesmiaaw
evmhesmianw	0	1	1	1	1	1	rD					rA					rB					1	0	1	1	0	0	0	1	0	0	1	EVX	evmhesmianw
evmhessf	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	0	0	0	0	0	1	1	EVX	evmhessf
evmhessfa	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	0	1	0	0	0	1	1	EVX	evmhessfa
evmhessfaaw	0	1	1	1	1	1	rD					rA					rB					1	0	1	0	0	0	0	0	0	1	1	EVX	evmhessfaaw

Table 271. Instructions sorted by mnemonic (binary) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
evmhessfanw	0	1	1	1	1	1	rD					rA					rB					1	0	1	1	0	0	0	0	0	1	1	EVX	evmhessfanw
evmhessiaaw	0	1	1	1	1	1	rD					rA					rB					1	0	1	0	0	0	0	0	0	0	1	EVX	evmhessiaaw
evmhessianw	0	1	1	1	1	1	rD					rA					rB					1	0	1	1	0	0	0	0	0	0	1	EVX	evmhessianw
evmheumi	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	0	0	0	1	0	0	0	EVX	evmheumi
evmheumia	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	0	1	0	1	0	0	0	EVX	evmheumia
evmheumiaaw	0	1	1	1	1	1	rD					rA					rB					1	0	1	0	0	0	0	1	0	0	0	EVX	evmheumiaaw
evmheumianw	0	1	1	1	1	1	rD					rA					rB					1	0	1	1	0	0	0	1	0	0	0	EVX	evmheumianw
evmheusiaaw	0	1	1	1	1	1	rD					rA					rB					1	0	1	0	0	0	0	0	0	0	0	EVX	evmheusiaaw
evmheusianw	0	1	1	1	1	1	rD					rA					rB					1	0	1	1	0	0	0	0	0	0	0	EVX	evmheusianw
evmhogsmfaa	0	1	1	1	1	1	rD					rA					rB					1	0	1	0	0	1	0	1	1	1	1	EVX	evmhogsmfaa
evmhogsmfan	0	1	1	1	1	1	rD					rA					rB					1	0	1	1	0	1	0	1	1	1	1	EVX	evmhogsmfan
evmhogsmiaa	0	1	1	1	1	1	rD					rA					rB					1	0	1	0	0	1	0	1	1	0	1	EVX	evmhogsmiaa
evmhogsmian	0	1	1	1	1	1	rD					rA					rB					1	0	1	1	0	1	0	1	1	0	1	EVX	evmhogsmian
evmhogumiaa	0	1	1	1	1	1	rD					rA					rB					1	0	1	0	0	1	0	1	1	0	0	EVX	evmhogumiaa
evmhogumian	0	1	1	1	1	1	rD					rA					rB					1	0	1	1	0	1	0	1	1	0	0	EVX	evmhogumian
evmhosmf	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	0	0	0	1	1	1	1	EVX	evmhosmf
evmhosmfa	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	0	1	0	1	1	1	1	EVX	evmhosmfa
evmhosmfaaw	0	1	1	1	1	1	rD					rA					rB					1	0	1	0	0	0	0	1	1	1	1	EVX	evmhosmfaaw
evmhosmfanw	0	1	1	1	1	1	rD					rA					rB					1	0	1	1	0	0	0	1	1	1	1	EVX	evmhosmfanw
evmhosmi	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	0	0	0	1	1	0	1	EVX	evmhosmi
evmhosmia	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	0	1	0	1	1	0	1	EVX	evmhosmia
evmhosmiaaw	0	1	1	1	1	1	rD					rA					rB					1	0	1	0	0	0	0	1	1	0	1	EVX	evmhosmiaaw
evmhosmianw	0	1	1	1	1	1	rD					rA					rB					1	0	1	1	0	0	0	1	1	0	1	EVX	evmhosmianw
evmhossf	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	0	0	0	0	1	1	1	EVX	evmhossf
evmhossfa	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	0	1	0	0	1	1	1	EVX	evmhossfa
evmhossfaaw	0	1	1	1	1	1	rD					rA					rB					1	0	1	0	0	0	0	0	1	1	1	EVX	evmhossfaaw

Table 271. Instructions sorted by mnemonic (binary) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
evmhossfanw	0	1	1	1	1	1	rD					rA					rB					1	0	1	1	0	0	0	0	1	1	1	EVX	**evmhossfanw**
evmhossiaaw	0	1	1	1	1	1	rD					rA					rB					1	0	1	0	0	0	0	0	1	0	1	EVX	**evmhossiaaw**
evmhossianw	0	1	1	1	1	1	rD					rA					rB					1	0	1	1	0	0	0	0	1	0	1	EVX	**evmhossianw**
evmhoumi	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	0	0	0	1	1	0	0	EVX	**evmhoumi**
evmhoumia	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	0	1	0	1	1	0	0	EVX	**evmhoumia**
evmhoumiaaw	0	1	1	1	1	1	rD					rA					rB					1	0	1	0	0	0	0	1	1	0	0	EVX	**evmhoumiaaw**
evmhoumianw	0	1	1	1	1	1	rD					rA					rB					1	0	1	1	0	0	0	1	1	0	0	EVX	**evmhoumianw**
evmhousiaaw	0	1	1	1	1	1	rD					rA					rB					1	0	1	0	0	0	0	0	1	0	0	EVX	**evmhousiaaw**
evmhousianw	0	1	1	1	1	1	rD					rA					rB					1	0	1	1	0	0	0	0	1	0	0	EVX	**evmhousianw**
evmr	**evmr rD,rA** equivalent to **evor rD,rA,rA**																																	**evmr**
evmra	0	1	1	1	1	1	rD					rA					///					1	0	0	1	1	0	0	0	1	0	0	EVX	**evmra**
evmwhgsmfaa	0	1	1	1	1	1	rD					rA					rB					1	0	1	0	1	1	0	1	1	1	1	EVX	**evmwhgsmfaa**
evmwhgsmfan	0	1	1	1	1	1	rD					rA					rB					1	0	1	1	1	0	1	1	1	1	1	EVX	**evmwhgsmfan**
evmwhgsmiaa	0	1	1	1	1	1	rD					rA					rB					1	0	1	0	1	1	0	1	1	0	1	EVX	**evmwhgsmiaa**
evmwhgsmian	0	1	1	1	1	1	rD					rA					rB					1	0	1	1	1	0	1	1	1	0	1	EVX	**evmwhgsmian**
evmwhgssfaa	0	1	1	1	1	1	rD					rA					rB					1	0	1	0	1	1	0	0	1	1	1	EVX	**evmwhgssfaa**
evmwhgssfan	0	1	1	1	1	1	rD					rA					rB					1	0	1	1	1	0	1	0	1	1	1	EVX	**evmwhgssfan**
evmwhgumiaa	0	1	1	1	1	1	rD					rA					rB					1	0	1	0	1	1	0	1	1	0	0	EVX	**evmwhgumiaa**
evmwhgumian	0	1	1	1	1	1	rD					rA					rB					1	0	1	1	1	0	1	1	1	0	0	EVX	**evmwhgumian**
evmwhsmf	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	1	0	0	1	1	1	1	EVX	**evmwhsmf**
evmwhsmfa	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	1	1	0	1	1	1	1	EVX	**evmwhsmfa**
evmwhsmfaaw	0	1	1	1	1	1	rD					rA					rB					1	0	1	0	1	0	0	1	1	1	1	EVX	**evmwhsmfaaw**
evmwhsmfanw	0	1	1	1	1	1	rD					rA					rB					1	0	1	1	1	0	0	1	1	1	1	EVX	**evmwhsmfanw**
evmwhsmi	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	1	0	0	1	1	0	1	EVX	**evmwhsmi**
evmwhsmia	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	1	1	0	1	1	0	1	EVX	**evmwhsmia**
evmwhsmiaaw	0	1	1	1	1	1	rD					rA					rB					1	0	1	0	1	0	0	1	1	0	1	EVX	**evmwhsmiaaw**
evmwhsmianw	0	1	1	1	1	1	rD					rA					rB					1	0	1	1	1	0	0	1	1	0	1	EVX	**evmwhsmianw**

Table 271. Instructions sorted by mnemonic (binary) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
evmwhssf	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	1	0	0	0	1	1	1	EVX	evmwhssf
evmwhssfa	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	1	1	0	0	1	1	1	EVX	evmwhssfa
evmwhssfaaw	0	1	1	1	1	1	rD					rA					rB					1	0	1	0	1	0	0	0	1	1	1	EVX	evmwhssfaaw
evmwhssfanw	0	1	1	1	1	1	rD					rA					rB					1	0	1	1	1	0	0	0	1	1	1	EVX	evmwhssfanw
evmwhssianw	0	1	1	1	1	1	rD					rA					rB					1	0	1	1	1	0	0	0	1	0	1	EVX	evmwhssianw
evmwhssmaaw	0	1	1	1	1	1	rD					rA					rB					1	0	1	0	1	0	0	0	1	0	1	EVX	evmwhssmaaw
evmwhumi	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	1	0	0	1	1	0	0	EVX	evmwhumi
evmwhumia	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	1	1	0	1	1	0	0	EVX	evmwhumia
evmwhusiaaw	0	1	1	1	1	1	rD					rA					rB					1	0	1	0	1	0	0	0	1	0	0	EVX	evmwhusiaaw
evmwhusianw	0	1	1	1	1	1	rD					rA					rB					1	0	1	1	1	0	0	0	1	0	0	EVX	evmwhusianw
evmwlsmf	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	1	0	0	1	0	1	1	EVX	evmwlsmf
evmwlsmfa	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	1	1	0	1	0	1	1	EVX	evmwlsmfa
evmwlsmfaaw	0	1	1	1	1	1	rD					rA					rB					1	0	1	0	1	0	0	1	0	1	1	EVX	evmwlsmfaaw
evmwlsmfanw	0	1	1	1	1	1	rD					rA					rB					1	0	1	1	1	0	0	1	0	1	1	EVX	evmwlsmfanw
evmwlsmiaaw	0	1	1	1	1	1	rD					rA					rB					1	0	1	0	1	0	0	1	0	0	1	EVX	evmwlsmiaaw
evmwlsmianw	0	1	1	1	1	1	rD					rA					rB					1	0	1	1	1	0	0	1	0	0	1	EVX	evmwlsmianw
evmwlssf	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	1	0	0	0	0	1	1	EVX	evmwlssf
evmwlssfa	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	1	1	0	0	0	1	1	EVX	evmwlssfa
evmwlssfaaw	0	1	1	1	1	1	rD					rA					rB					1	0	1	0	1	0	0	0	0	1	1	EVX	evmwlssfaaw
evmwlssfanw	0	1	1	1	1	1	rD					rA					rB					1	0	1	1	1	0	0	0	0	1	1	EVX	evmwlssfanw
evmwlssiaaw	0	1	1	1	1	1	rD					rA					rB					1	0	1	0	1	0	0	0	0	0	1	EVX	evmwlssiaaw
evmwlssianw	0	1	1	1	1	1	rD					rA					rB					1	0	1	1	1	0	0	0	0	0	1	EVX	evmwlssianw
evmwlumi	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	1	0	0	1	0	0	0	EVX	evmwlumi
evmwlumia	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	1	1	0	1	0	0	0	EVX	evmwlumia
evmwlumiaaw	0	1	1	1	1	1	rD					rA					rB					1	0	1	0	1	0	0	1	0	0	0	EVX	evmwlumiaaw
evmwlumianw	0	1	1	1	1	1	rD					rA					rB					1	0	1	1	1	0	0	1	0	0	0	EVX	evmwlumianw

Table 271. Instructions sorted by mnemonic (binary) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
evmwlusiaaw	0	1	1	1	1	1	rD					rA					rB					1	0	1	0	1	0	0	0	0	0	EVX	**evmwlusiaaw**
evmwlusianw	0	1	1	1	1	1	rD					rA					rB					1	0	1	1	1	0	0	0	0	0	EVX	**evmwlusianw**
evmwsmf	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	1	0	1	1	0	1	1	EVX	**evmwsmf**
evmwsmfa	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	1	1	1	1	0	1	1	EVX	**evmwsmfa**
evmwsmfaa	0	1	1	1	1	1	rD					rA					rB					1	0	1	0	1	0	1	1	0	1	1	EVX	**evmwsmfaa**
evmwsmfan	0	1	1	1	1	1	rD					rA					rB					1	0	1	1	1	0	1	1	0	1	1	EVX	**evmwsmfan**
evmwsmi	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	1	0	1	1	0	0	1	EVX	**evmwsmi**
evmwsmia	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	1	1	1	1	0	0	1	EVX	**evmwsmia**
evmwsmiaa	0	1	1	1	1	1	rD					rA					rB					1	0	1	0	1	0	1	1	0	0	1	EVX	**evmwsmiaa**
evmwsmian	0	1	1	1	1	1	rD					rA					rB					1	0	1	1	1	0	1	1	0	0	1	EVX	**evmwsmian**
evmwssf	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	1	0	1	0	0	1	1	EVX	**evmwssf**
evmwssfa	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	1	1	1	0	0	1	1	EVX	**evmwssfa**
evmwssfaa	0	1	1	1	1	1	rD					rA					rB					1	0	1	0	1	0	1	0	0	1	1	EVX	**evmwssfaa**
evmwssfan	0	1	1	1	1	1	rD					rA					rB					1	0	1	1	1	0	1	0	0	1	1	EVX	**evmwssfan**
evmwumi	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	1	0	1	1	0	0	0	EVX	**evmwumi**
evmwumia	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	1	1	1	1	0	0	0	EVX	**evmwumia**
evmwumiaa	0	1	1	1	1	1	rD					rA					rB					1	0	1	0	1	0	1	1	0	0	0	EVX	**evmwumiaa**
evmwumian	0	1	1	1	1	1	rD					rA					rB					1	0	1	1	1	0	1	1	0	0	0	EVX	**evmwumian**
evnand	0	1	1	1	1	1	rD					rA					rB					0	1	0	0	0	0	1	1	1	1	0	EVX	**evnand**
evneg	0	1	1	1	1	1	rD					rA					///					0	1	0	0	0	0	0	1	0	0	1	EVX	**evneg**
evnor	0	1	1	1	1	1	rD					rA					rB					0	1	0	0	0	0	1	1	0	0	0	EVX	**evnor**
evnot	**evnot rD,rA** equivalent to **evnor rD,rA,rA**																																	**evnot**
evor	0	1	1	1	1	1	rD					rA					rB					0	1	0	0	0	0	1	0	1	1	1	EVX	**evor**
evorc	0	1	1	1	1	1	rD					rA					rB					0	1	0	0	0	0	1	1	0	1	1	EVX	**evorc**
evrlw	0	1	1	1	1	1	rD					rA					rB					0	1	0	0	0	1	0	1	0	0	0	EVX	**evrlw**
evrlwi	0	1	1	1	1	1	rD					rA					UIMM					0	1	0	0	0	1	0	1	0	1	0	EVX	**evrlwi**
evrndw	0	1	1	1	1	1	rD					rA					UIMM					0	1	0	0	0	0	0	1	1	0	0	EVX	**evrndw**
evsel	0	1	1	1	1	1	rD					rA					rB					0	1	0	0	1	1	1	1	crfS			EVX	**evsel**
evslw	0	1	1	1	1	1	rD					rA					rB					0	1	0	0	0	1	0	0	1	0	0	EVX	**evslw**
evslwi	0	1	1	1	1	1	rD					rA					UIMM					0	1	0	0	0	1	0	0	1	1	0	EVX	**evslwi**
evsplatfi	0	1	1	1	1	1	rD					SIMM					///					0	1	0	0	0	1	0	1	0	1	1	EVX	**evsplatfi**
evsplati	0	1	1	1	1	1	rD					SIMM					///					0	1	0	0	0	1	0	1	0	0	1	EVX	**evsplati**
evsrwis	0	1	1	1	1	1	rD					rA					UIMM					0	1	0	0	0	1	0	0	0	1	1	EVX	**evsrwis**

Table 271. Instructions sorted by mnemonic (binary) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
evsrwiu	0	1	1	1	1	1	rD					rA					UIMM					0	1	0	0	0	1	0	0	0	1	0	EVX	**evsrwiu**
evsrws	0	1	1	1	1	1	rD					rA					rB					0	1	0	0	0	1	0	0	0	0	1	EVX	**evsrws**
evsrwu	0	1	1	1	1	1	rD					rA					rB					0	1	0	0	0	1	0	0	0	0	0	EVX	**evsrwu**
evstdd	0	1	1	1	1	1	rD					rA					UIMM [8]					0	1	1	0	0	1	0	0	0	0	1	EVX	**evstdd**
evstddx	0	1	1	1	1	1	rS					rA					rB					0	1	1	0	0	1	0	0	0	0	0	EVX	**evstddx**
evstdh	0	1	1	1	1	1	rS					rA					UIMM [8]					0	1	1	0	0	1	0	0	1	0	1	EVX	**evstdh**
evstdhx	0	1	1	1	1	1	rS					rA					rB					0	1	1	0	0	1	0	0	1	0	0	EVX	**evstdhx**
evstdw	0	1	1	1	1	1	rS					rA					UIMM [8]					0	1	1	0	0	1	0	0	0	1	1	EVX	**evstdw**
evstdwx	0	1	1	1	1	1	rS					rA					rB					0	1	1	0	0	1	0	0	0	1	0	EVX	**evstdwx**
evstwhe	0	1	1	1	1	1	rS					rA					UIMM [10]					0	1	1	0	0	1	1	0	0	0	1	EVX	**evstwhe**
evstwhex	0	1	1	1	1	1	rS					rA					rB					0	1	1	0	0	1	1	0	0	0	0	EVX	**evstwhex**
evstwho	0	1	1	1	1	1	rS					rA					UIMM [10]					0	1	1	0	0	1	1	0	1	0	1	EVX	**evstwho**
evstwhox	0	1	1	1	1	1	rS					rA					rB					0	1	1	0	0	1	1	0	1	0	0	EVX	**evstwhox**
evstwwe	0	1	1	1	1	1	rS					rA					UIMM [10]					0	1	1	0	0	1	1	1	0	0	1	EVX	**evstwwe**
evstwwex	0	1	1	1	1	1	rS					rA					rB					0	1	1	0	0	1	1	1	0	0	0	EVX	**evstwwex**
evstwwo	0	1	1	1	1	1	rS					rA					UIMM [10]					0	1	1	0	0	1	1	1	1	0	1	EVX	**evstwwo**
evstwwox	0	1	1	1	1	1	rS					rA					rB					0	1	1	0	0	1	1	1	1	0	0	EVX	**evstwwox**
evsubfsmiaaw	0	1	1	1	1	1	rD					rA					///					1	0	0	1	1	0	0	1	0	1	1	EVX	**evsubfsmiaaw**
evsubfssiaaw	0	1	1	1	1	1	rD					rA					///					1	0	0	1	1	0	0	0	0	1	1	EVX	**evsubfssiaaw**
evsubfumiaaw	0	1	1	1	1	1	rD					rA					///					1	0	0	1	1	0	0	1	0	1	0	EVX	**evsubfumiaaw**
evsubfusiaaw	0	1	1	1	1	1	rD					rA					///					1	0	0	1	1	0	0	0	0	1	0	EVX	**evsubfusiaaw**
evsubfw	0	1	1	1	1	1	rD					rA					rB					0	1	0	0	0	0	0	0	1	0	0	EVX	**evsubfw**
evsubifw	0	1	1	1	1	1	rD					UIMM					rB					0	1	0	0	0	0	0	0	1	1	0	EVX	**evsubifw**
evsubiw	**evsubiw rD,rB,UIMM** equivalent to **evsubifw rD,UIMM,rB**																																	**evsubiw**
evsubw	**evsubw rD,rB,rA** equivalent to **evsubfw rD,rA,rB**																																	**evsubw**
evxor	0	1	1	1	1	1	rD					rA					rB					0	1	0	0	0	0	1	0	1	1	0	EVX	**evxor**
extlwi	**extlwi rA,rS,*n*,*b*** (n > 0) equivalent to **rlwinm rA,rS,*b*,0,*n* – 1**																																	**extlwi**
extrwi	**extrwi rA,rS,*n*,*b* (*n* > 0)** equivalent to **rlwinm rA,rS,*b* + *n*, 32 – *n*,31**																																	**extrwi**
extsb	0	1	1	1	1	1	rS					rA					///					1	1	1	0	1	1	1	0	1	0	0	X	**extsb**
extsb.	0	1	1	1	1	1	rS					rA					///					1	1	1	0	1	1	1	0	1	0	1	X	**extsb.**
extsh	0	1	1	1	1	1	rS					rA					///					1	1	1	0	0	1	1	0	1	0	0	X	**extsh**
extsh.	0	1	1	1	1	1	rS					rA					///					1	1	1	0	0	1	1	0	1	0	1	X	**extsh.**
fres [6]	1	1	1	0	1	1	frD					///					frB					///					1	1	0	0	0	0	A	**fres**
fres. [6]	1	1	1	0	1	1	frD					///					frB					///					1	1	0	0	0	1	A	**fres.**
fsel [6]	1	1	1	1	1	1	frD					frA					frB					frC					1	0	1	1	1	0	A	**fsel**
fsel. [6]	1	1	1	1	1	1	frD					frA					frB					frC					1	0	1	1	1	1	A	**fsel.**

Table 271. Instructions sorted by mnemonic (binary) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
icbi	0	1	1	1	1	1	///					rA					rB					1	1	1	1	0	1	0	1	1	0	/	X	**icbi**
icblc	0	1	1	1	1	1	CT					rA					rB					0	0	1	1	1	0	0	1	1	0	0	X	**icblc**
icbt	0	1	1	1	1	1	CT					rA					rB					0	0	0	0	0	1	0	1	1	0	/	X	**icbt**
icbtls	0	1	1	1	1	1	CT					rA					rB					0	1	1	1	1	0	0	1	1	0	0	X	**icbtls**
inslwi	**inslwi rA,rS,*n*,*b* (*n* > 0)**											equivalent to					**rlwimi rA,rS,32 – *b*,*b*,(*b* + *n*) – 1**																	**inslwi**
insrwi	**insrwi rA,rS,*n*,*b* (*n* > 0)**											equivalent to					**rlwimi rA,rS,32 – (*b* + *n*),*b*,(*b* + *n*) – 1**																	**insrwi**
isel	0	1	1	1	1	1	rD					rA					rB					crb					0	1	1	1	1	0	X	**isel**
iseleq	**iseleq rD,rA,rB** equivalent to **isel rD,rA,rB,2**																																	**iseleq**
iselgt	**iselgt rD,rA,rB** equivalent to **isel rD,rA,rB,1**																																	**iselgt**
iselit	**iselit rD,rA,rB** equivalent to **isel rD,rA,rB,0**																																	**iselit**
isync	0	1	0	0	1	1	///															0	0	1	0	0	1	0	1	1	0	/	XL	**isync**
la	**la rD,d(rA)** equivalent to **addi rD,rA,d**																																	**la**
lbz	1	0	0	0	1	0	rD					rA					D																D	**lbz**
lbzu	1	0	0	0	1	1	rD					rA					D																D	**lbzu**
lbzux	0	1	1	1	1	1	rD					rA					rB					0	0	0	1	1	1	0	1	1	1	/	X	**lbzux**
lbzx	0	1	1	1	1	1	rD					rA					rB					0	0	0	1	0	1	0	1	1	1	/	X	**lbzx**
lha	1	0	1	0	1	0	rD					rA					D																D	**lha**
lhau	1	0	1	0	1	1	rD					rA					D																D	**lhau**
lhaux	0	1	1	1	1	1	rD					rA					rB					0	1	0	1	1	1	0	1	1	1	/	X	**lhaux**
lhax	0	1	1	1	1	1	rD					rA					rB					0	1	0	1	0	1	0	1	1	1	/	X	**lhax**
lhbrx	0	1	1	1	1	1	rD					rA					rB					1	1	0	0	0	1	0	1	1	0	/	X	**lhbrx**
lhz	1	0	1	0	0	0	rD					rA					D																D	**lhz**
lhzu	1	0	1	0	0	1	rD					rA					D																D	**lhzu**
lhzux	0	1	1	1	1	1	rD					rA					rB					0	1	0	0	1	1	0	1	1	1	/	X	**lhzux**
lhzx	0	1	1	1	1	1	rD					rA					rB					0	1	0	0	0	1	0	1	1	1	/	X	**lhzx**
li	**li rD,value** equivalent to **addi rD,0,**value																																	**li**
lis	**lis rD,value** equivalent to **addis rD,0,**value																																	**lis**
lmw	1	0	1	1	1	0	rD					rA					D																D	**lmw**
lwarx	0	1	1	1	1	1	rD					rA					rB					0	0	0	0	0	1	0	1	0	0	/	X	**lwarx**
lwbrx	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	0	1	0	1	1	0	/	X	**lwbrx**
lwz	1	0	0	0	0	0	rD					rA					D																D	**lwz**
lwzu	1	0	0	0	0	1	rD					rA					D																D	**lwzu**
lwzux	0	1	1	1	1	1	rD					rA					rB					0	0	0	0	1	1	0	1	1	1	/	X	**lwzux**
lwzx	0	1	1	1	1	1	rD					rA					rB					0	0	0	0	0	1	0	1	1	1	/	X	**lwzx**
mbar	0	1	1	1	1	1	MO					///										1	1	0	1	0	1	0	1	1	0	/	X	**mbar**
mcrf	0	1	0	0	1	1	crfD			//		crfS			///							0	0	0	0	0	0	0	0	0	0	/	XL	**mcrf**
mcrxr	0	1	1	1	1	1	crfD			///												1	0	0	0	0	0	0	0	0	0	/	X	**mcrxr**
mfcr	**mtcr rS** equivalent to **mtcrf 0xFF,rS**																																	**mfcr**
mfcr	0	1	1	1	1	1	rD					///										0	0	0	0	0	1	0	0	1	1	/	X	**mfcr**
mfdcr	0	1	1	1	1	1	rD					DCRN5–9					DCRN0–4					0	1	0	1	0	0	0	0	1	1	/	XFX	**mfdcr**

Table 271. Instructions sorted by mnemonic (binary) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
mfmsr [7]	0	1	1	1	1	1	rD					///										0	0	0	1	0	1	0	0	1	1	/	X	**mfmsr**
mfpmr	0	1	1	1	1	1	rD					PMRN5–9					PMRN0–4					0	1	0	1	0	0	1	1	1	0	0	XFX	**mfpmr**
mf*regname*	**mf*regname*** rD equivalent to **mfspr** rD,**SPR*n***																																	**mf*regname***
mfspr [9]	0	1	1	1	1	1	rD					SPR[5–9]					SPR[0–4]					0	1	0	1	0	1	0	0	1	1	/	XFX	**mfspr**
mr	**mr** rA,rS equivalent to **or** rA,rS,rS																																	**mr**
msync	0	1	1	1	1	1	///															1	0	0	1	0	1	0	1	1	0	/	X	**msync**
mtcr	**mtcr** rS equivalent to **mtcrf** 0xFF,rS																																	**mtcr**
mtcrf	0	1	1	1	1	1	rS					/	CRM								/	0	0	1	0	0	1	0	0	0	0	/	XFX	**mtcrf**
mtdcr	0	1	1	1	1	1	rS					DCRN5–9					DCRN0–4					0	1	1	1	0	0	0	0	1	1	/	XFX	**mtdcr**
mtmsr [7]	0	1	1	1	1	1	rS					///										0	0	1	0	0	1	0	0	1	0	/	X	**mtmsr**
mtpmr	0	1	1	1	1	1	rS					PMRN5–9					PMRN0–4					0	1	1	1	0	0	1	1	1	0	0	XFX	**mtpmr**
mt*regname*	**mt*regname*** rS equivalent to **mtspr SPR*n*** rS																																	**mt*regname***
mtspr (9)	0	1	1	1	1	1	rS					SPR[5–9]					SPR[0–4]					0	1	1	1	0	1	0	0	1	1	/	XFX	**mtspr**
mulhw	0	1	1	1	1	1	rD					rA					rB					/	0	0	1	0	0	1	0	1	1	0	X	**mulhw**
mulhw.	0	1	1	1	1	1	rD					rA					rB					/	0	0	1	0	0	1	0	1	1	1	X	**mulhw.**
mulhwu	0	1	1	1	1	1	rD					rA					rB					/	0	0	0	0	0	1	0	1	1	0	X	**mulhwu**
mulhwu.	0	1	1	1	1	1	rD					rA					rB					/	0	0	0	0	0	1	0	1	1	1	X	**mulhwu.**
mulli	0	0	0	1	1	1	rD					rA					SIMM																D	**mulli**
mullw	0	1	1	1	1	1	rD					rA					rB					0	0	1	1	1	0	1	0	1	1	0	X	**mullw**
mullw.	0	1	1	1	1	1	rD					rA					rB					0	0	1	1	1	0	1	0	1	1	1	X	**mullw.**
mullwo	0	1	1	1	1	1	rD					rA					rB					1	0	1	1	1	0	1	0	1	1	0	X	**mullwo**
mullwo.	0	1	1	1	1	1	rD					rA					rB					1	0	1	1	1	0	1	0	1	1	1	X	**mullwo.**
nand	0	1	1	1	1	1	rS					rA					rB					0	1	1	1	0	1	1	1	0	0	0	X	**nand**
nand.	0	1	1	1	1	1	rS					rA					rB					0	1	1	1	0	1	1	1	0	0	1	X	**nand.**
neg	0	1	1	1	1	1	rD					rA					///					0	0	0	1	1	0	1	0	0	0	0	X	**neg**
neg.	0	1	1	1	1	1	rD					rA					///					0	0	0	1	1	0	1	0	0	0	1	X	**neg.**
nego	0	1	1	1	1	1	rD					rA					///					1	0	0	1	1	0	1	0	0	0	0	X	**nego**
nego.	0	1	1	1	1	1	rD					rA					///					1	0	0	1	1	0	1	0	0	0	1	X	**nego.**
nop	**nop** equivalent to **ori 0,0,0**																																	**nop**
nor	0	1	1	1	1	1	rS					rA					rB					0	0	0	1	1	1	1	1	0	0	0	X	**nor**
nor.	0	1	1	1	1	1	rS					rA					rB					0	0	0	1	1	1	1	1	0	0	1	X	**nor.**
not	**not** rA,rS equivalent to **nor** rA,rS,rS																																	**not**
or	0	1	1	1	1	1	rS					rA					rB					0	1	1	0	1	1	1	1	0	0	0	X	**or**
or.	0	1	1	1	1	1	rS					rA					rB					0	1	1	0	1	1	1	1	0	0	1	X	**or.**
orc	0	1	1	1	1	1	rS					rA					rB					0	1	1	0	0	1	1	1	0	0	0	X	**orc**
orc.	0	1	1	1	1	1	rS					rA					rB					0	1	1	0	0	1	1	1	0	0	1	X	**orc.**
ori	0	1	1	0	0	0	rS					rA					UIMM																D	**ori**
oris	0	1	1	0	0	1	rS					rA					UIMM																D	**oris**
rfci	0	1	0	0	1	1	///															0	0	0	0	1	1	0	0	1	1	/	XL	**rfci**

Table 271. Instructions sorted by mnemonic (binary) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
rfdi [7]	0	1	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	1	1	0	X	**rfdi**
rfi [7]	0	1	0	0	1	1	///															0	0	0	0	1	1	0	0	1	0	/	XL	**rfi**
rfmci [7]	0	1	0	0	1	1	///															0	0	0	0	1	0	0	1	1	0	/	XL	**rfmci**
rlwimi	0	1	0	1	0	0	rS					rA					SH					MB					ME				0	M	**rlwimi**
rlwimi.	0	1	0	1	0	0	rS					rA					SH					MB					ME				1	M	**rlwimi.**
rlwinm	0	1	0	1	0	1	rS					rA					SH					MB					ME				0	M	**rlwinm**
rlwinm.	0	1	0	1	0	1	rS					rA					SH					MB					ME				1	M	**rlwinm.**
rlwnm	0	1	0	1	1	1	rS					rA					rB					MB					ME				0	M	**rlwnm**
rlwnm.	0	1	0	1	1	1	rS					rA					rB					MB					ME				1	M	**rlwnm.**
rotlw	**rotlw rA,rS,rB** equivalent to **rlwnm rA,rS,rB,0,31**																																	**rotlw**
rotlwi	**rotlwi rA,rS,*n*** equivalent to **rlwinm rA,rS,*n*,0,31**																																	**rotlwi**
rotrwi	**rotrwi rA,rS,*n*** equivalent to **rlwinm rA,rS,32 – *n*,0,31**																																	**rotrwi**
sc	0	1	0	0	0	1	///																								1	/	SC	**sc**
slw	0	1	1	1	1	1	rS					rA					rB					0	0	0	0	0	1	1	0	0	0	X	**slw**
slw.	0	1	1	1	1	1	rS					rA					rB					0	0	0	0	0	1	1	0	0	1	X	**slw.**
slwi	**slwi rA,rS,*n*** (n < 32)equivalent to **rlwinm rA,rS,*n*,0,31 – *n***																																	**slwi**
sraw	0	1	1	1	1	1	rS					rA					rB					1	1	0	0	0	1	1	0	0	0	X	**sraw**
sraw.	0	1	1	1	1	1	rS					rA					rB					1	1	0	0	0	1	1	0	0	1	X	**sraw.**
srawi	0	1	1	1	1	1	rS					rA					SH					1	1	0	0	1	1	1	0	0	0	X	**srawi**
srawi.	0	1	1	1	1	1	rS					rA					SH					1	1	0	0	1	1	1	0	0	1	X	**srawi.**
srw	0	1	1	1	1	1	rS					rA					rB					1	0	0	0	0	1	1	0	0	0	X	**srw**
srw.	0	1	1	1	1	1	rS					rA					rB					1	0	0	0	0	1	1	0	0	1	X	**srw.**
srwi	**srwi rA,rS,*n*** (n < 32)equivalent to **rlwinm rA,rS,32 – *n*,*n*,31**																																	**srwi**
stb	1	0	0	1	1	0	rS					rA					D																D	**stb**
stbu	1	0	0	1	1	1	rS					rA					D																D	**stbu**
stbux	0	1	1	1	1	1	rS					rA					rB					0	0	1	1	1	1	0	1	1	1	0	X	**stbux**
stbx	0	1	1	1	1	1	rS					rA					rB					0	0	1	1	0	1	0	1	1	1	0	X	**stbx**
sth	1	0	1	1	0	0	rS					rA					D																D	**sth**
sthbrx	0	1	1	1	1	1	rS					rA					rB					1	1	1	0	0	1	0	1	1	0	/	X	**sthbrx**
sthu	1	0	1	1	0	1	rS					rA					D																D	**sthu**
sthux	0	1	1	1	1	1	rS					rA					rB					0	1	1	0	1	1	0	1	1	1	/	X	**sthux**
sthx	0	1	1	1	1	1	rS					rA					rB					0	1	1	0	0	1	0	1	1	1	/	X	**sthx**
stmw	1	0	1	1	1	1	rS					rA					D																D	**stmw**
stw	1	0	0	1	0	0	rS					rA					D																D	**stw**
stwbrx	0	1	1	1	1	1	rS					rA					rB					1	0	1	0	0	1	0	1	1	0	/	X	**stwbrx**
stwcx.	0	1	1	1	1	1	rS					rA					rB					0	0	1	0	0	1	0	1	1	0	1	X	**stwcx.**
stwu	1	0	0	1	0	1	rS					rA					D																D	**stwu**
stwux	0	1	1	1	1	1	rS					rA					rB					0	0	1	0	1	1	0	1	1	1	/	D	**stwux**
stwx	0	1	1	1	1	1	rS					rA					rB					0	0	1	0	0	1	0	1	1	1	/	D	**stwx**
sub	**sub rD,rA,rB** equivalent to **subf rD,rB,rA**																																	**sub**

Table 271. Instructions sorted by mnemonic (binary) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
subc	**subc rD,rA,rB** equivalent to **subfc rD,rB,rA**																																	**subc**
subf	0	1	1	1	1	1	rD				rA						rB					0	0	0	0	1	0	1	0	0	0	0	X	**subf**
subf.	0	1	1	1	1	1	rD				rA						rB					0	0	0	0	1	0	1	0	0	0	1	X	**subf.**
subfc	0	1	1	1	1	1	rD				rA						rB					0	0	0	0	0	0	1	0	0	0	0	X	**subfc**
subfc.	0	1	1	1	1	1	rD				rA						rB					0	0	0	0	0	0	1	0	0	0	1	X	**subfc.**
subfco	0	1	1	1	1	1	rD				rA						rB					1	0	0	0	0	0	1	0	0	0	0	X	**subfco**
subfco.	0	1	1	1	1	1	rD				rA						rB					1	0	0	0	0	0	1	0	0	0	1	X	**subfco.**
subfe	0	1	1	1	1	1	rD				rA						rB					0	0	1	0	0	0	1	0	0	0	0	X	**subfe**
subfe.	0	1	1	1	1	1	rD				rA						rB					0	0	1	0	0	0	1	0	0	0	1	X	**subfe.**
subfeo	0	1	1	1	1	1	rD				rA						rB					1	0	1	0	0	0	1	0	0	0	0	X	**subfeo**
subfeo.	0	1	1	1	1	1	rD				rA						rB					1	0	1	0	0	0	1	0	0	0	1	X	**subfeo.**
subfic	0	0	1	0	0	0	rD				rA						SIMM																D	**subfic**
subfme	0	1	1	1	1	1	rD				rA						///					0	0	1	1	1	0	1	0	0	0	0	X	**subfme**
subfme.	0	1	1	1	1	1	rD				rA						///					0	0	1	1	1	0	1	0	0	0	1	X	**subfme.**
subfmeo	0	1	1	1	1	1	rD				rA						///					1	0	1	1	1	0	1	0	0	0	0	X	**subfmeo**
subfmeo.	0	1	1	1	1	1	rD				rA						///					1	0	1	1	1	0	1	0	0	0	1	X	**subfmeo.**
subfo	0	1	1	1	1	1	rD				rA						rB					1	0	0	0	1	0	1	0	0	0	0	X	**subfo**
subfo.	0	1	1	1	1	1	rD				rA						rB					1	0	0	0	1	0	1	0	0	0	1	X	**subfo.**
subfze	0	1	1	1	1	1	rD				rA						///					0	0	1	1	0	0	1	0	0	0	0	X	**subfze**
subfze.	0	1	1	1	1	1	rD				rA						///					0	0	1	1	0	0	1	0	0	0	1	X	**subfze.**
subfzeo	0	1	1	1	1	1	rD				rA						///					1	0	1	1	0	0	1	0	0	0	0	X	**subfzeo**
subfzeo.	0	1	1	1	1	1	rD				rA						///					1	0	1	1	0	0	1	0	0	0	1	X	**subfzeo.**
subi	**subi rD,rA,value** equivalent to **addi rD,rA,**–value																																	**subi**
subic	**subic rD,rA,value** equivalent to **addic rD,rA,**–value																																	**subic**
subic.	**subic. rD,rA,value** equivalent to **addic. rD,rA,**–value																																	**subic.**
subis	**subis rD,rA,value** equivalent to **addis rD,rA,**–value																																	**subis**
tlbie [6,7]	0	1	1	1	1	1	///				///						rB					0	1	0	0	1	1	0	0	1	0	0	X	**tlbie**
tlbivax	0	1	1	1	1	1	///				rA						rB					1	1	0	0	0	1	0	0	1	0	/	X	**tlbivax**
tlbre	0	1	1	1	1	1	///[10]															1	1	1	0	1	1	0	0	1	0	/	X	**tlbre**
tlbsx	0	1	1	1	1	1	///[12]				rA						rB					1	1	1	0	0	1	0	0	1	0	/12	X	**tlbsx**
tlbsync [6,7]	0	1	1	1	1	1	///				///						///					1	0	0	0	1	1	0	1	1	0	/	X	**tlbsync**
tlbwe	0	1	1	1	1	1	///[12]															1	1	1	1	0	1	0	0	1	0	/	X	**tlbwe**
tw	0	1	1	1	1	1	TO				rA						rB					0	0	0	0	0	0	0	1	0	0	/	X	**tw**
tweq	**tweq rA,SIMM** equivalent to **tw 4,rA,**SIMM																																	**tweq**
tweqi	**tweqi rA,SIMM** equivalent to **twi 4,rA,**SIMM																																	**tweqi**
twge	**twge rA,SIMM** equivalent to **tw 12,rA,**SIMM																																	**twge**
twgei	**twgei rA,SIMM** equivalent to **twi 12,rA,**SIMM																																	**twgei**
twgt	**twgt rA,SIMM** equivalent to **tw 8,rA,**SIMM																																	**twgt**
twgti	**twgti rA,SIMM** equivalent to **twi 8,rA,**SIMM																																	**twgti**

Table 271. Instructions sorted by mnemonic (binary) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
twi	0	0	0	0	1	1	TO					rA					SIMM																D	**twi**
twle	**twle rA,SIMM** equivalent to **tw 20,rA,SIMM**																																	**twle**
twlei	**twlei rA,SIMM** equivalent to **twi 20,rA,SIMM**																																	**twlei**
twlge	**twlge rA,SIMM** equivalent to **tw 12,rA,SIMM**																																	**twlge**
twlgei	**twlgei rA,SIMM** equivalent to **twi 12,rA,SIMM**																																	**twlgei**
twlgt	**twlgt rA,SIMM** equivalent to **tw 1,rA,SIMM**																																	**twlgt**
twlgti	**twlgti rA,SIMM** equivalent to **twi 1,rA,SIMM**																																	**twlgti**
twlle	**twlle rA,SIMM** equivalent to **tw 6,rA,SIMM**																																	**twlle**
twllei	**twllei rA,SIMM** equivalent to **twi 6,rA,SIMM**																																	**twllei**
twllt	**twllt rA,SIMM** equivalent to **tw 2,rA,SIMM**																																	**twllt**
twllti	**twllti rA,SIMM** equivalent to **twi 2,rA,SIMM**																																	**twllti**
twlng	**twlng rA,SIMM** equivalent to **tw 6,rA,SIMM**																																	**twlng**
twlngi	**twlngi rA,SIMM** equivalent to **twi 6,rA,SIMM**																																	**twlngi**
twlnl	**twlnl rA,SIMM** equivalent to **tw 5,rA,SIMM**																																	**twlnl**
twlnli	**twlnli rA,SIMM** equivalent to **twi 5,rA,SIMM**																																	**twlnli**
twlt	**twlt rA,SIMM** equivalent to **tw 16,rA,SIMM**																																	**twlt**
twlti	**twlti rA,SIMM** equivalent to **twi 16,rA,SIMM**																																	**twlti**
twne	**twne rA,SIMM** equivalent to **tw 24,rA,SIMM**																																	**twne**
twnei	**twnei rA,SIMM** equivalent to **twi 24,rA,SIMM**																																	**twnei**
twng	**twng rA,SIMM** equivalent to **tw 20,rA,SIMM**																																	**twng**
twngi	**twngi rA,SIMM** equivalent to **twi 20,rA,SIMM**																																	**twngi**
twnl	**twnl rA,SIMM** equivalent to **tw 12,rA,SIMM**																																	**twnl**
twnli	**twnli rA,SIMM** equivalent to **twi 12,rA,SIMM**																																	**twnli**
wait	0	1	1	1	1	1	///															0	0	0	0	1	1	1	1	1	0	/		**wait**
wrtee	0	1	1	1	1	1	rS					///										0	0	1	0	0	0	0	0	1	1	/	X	**wrtee**
wrteei	0	1	1	1	1	1	///										E	///				0	0	1	0	1	0	0	0	1	1	/	X	**wrteei**
xor	0	1	1	1	1	1	rS					rA					rB					0	1	0	0	1	1	1	1	0	0	0	X	**xor**
xor.	0	1	1	1	1	1	rS					rA					rB					0	1	0	0	1	1	1	1	0	0	1	X	**xor.**
xori	0	1	1	0	1	0	rS					rA					UIMM																D	**xori**
xoris	0	1	1	0	1	1	rS					rA					UIMM																D	**xoris**

1. Simplified mnemonics for branch instructions that do not test a CR bit should not specify one; a programming error may occur.
2. The value in the BI operand selects CRn[2], the EQ bit.
3. The value in the BI operand selects CRn[0], the LT bit.
4. The value in the BI operand selects CRn[1], the GT bit.
5. The value in the BI operand selects CRn[3], the SO bit.
6. Optional to the PowerPC classic architecture.
7. Supervisor-level instruction.
8. d = UIMM * 8
9. Access level is detemined by whether the SPR is defined as a user or supervisor level SPR.

A.4 Instructions sorted by opcode (binary)

Table 272 lists instructions by opcode, shown in binary.

Table 272. Instructions sorted by opcode (binary)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
rfdi [1]	0	1	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	1	1	0	X	**rfdi**
twi	0	0	0	0	1	1	TO					rA					SIMM																D	**twi**
brinc	0	0	0	1	0	0	rD					rA					rB					0	1	0	0	0	0	0	1	1	1	1	EVX	**brinc**
efdabs	0	0	0	1	0	0	rD					rA					///					0	1	0	1	1	1	0	0	1	0	0	EFX	**efdabs**
efdadd	0	0	0	1	0	0	rD					rA					rB					0	1	0	1	1	1	0	0	0	0	0	EFX	**efdadd**
efdcfs	0	0	0	1	0	0	rD					0	0	0	0	0	rB					0	1	0	1	1	1	0	1	1	1	1	EFX	**efdcfs**
efdcfsf	0	0	0	1	0	0	rD					///					rB					0	1	0	1	1	1	1	0	0	1	1	EFX	**efdcfsf**
efdcfsi	0	0	0	1	0	0	rD					///					rB					0	1	0	1	1	1	1	0	0	0	1	EFX	**efdcfsi**
efdcfuf	0	0	0	1	0	0	rD					///					rB					0	1	0	1	1	1	1	0	0	1	0	EFX	**efdcfuf**
efdcfui	0	0	0	1	0	0	rD					///					rB					0	1	0	1	1	1	1	0	0	0	0	EFX	**efdcfui**
efdcmpeq	0	0	0	1	0	0	crfD			/	/	rA					rB					0	1	0	1	1	1	0	1	1	1	0	EFX	**efdcmpeq**
efdcmpgt	0	0	0	1	0	0	crfD			/	/	rA					rB					0	1	0	1	1	1	0	1	1	0	0	EFX	**efdcmpgt**
efdcmplt	0	0	0	1	0	0	crfD			/	/	rA					rB					0	1	0	1	1	1	0	1	1	0	1	EFX	**efdcmplt**
efdctsf	0	0	0	1	0	0	rD					///					rB					0	1	0	1	1	1	1	0	1	1	1	EFX	**efdctsf**
efdctsi	0	0	0	1	0	0	rD					///					rB					0	1	0	1	1	1	1	0	1	0	1	EFX	**efdctsi**
efdctsiz	0	0	0	1	0	0	rD					///					rB					0	1	0	1	1	1	1	1	0	1	0	EFX	**efdctsiz**
efdctuf	0	0	0	1	0	0	rD					///					rB					0	1	0	1	1	1	1	0	1	1	0	EFX	**efdctuf**
efdctui	0	0	0	1	0	0	rD					///					rB					0	1	0	1	1	1	1	0	1	0	0	EFX	**efdctui**
efdctuiz	0	0	0	1	0	0	rD					///					rB					0	1	0	1	1	1	1	1	0	0	0	EFX	**efdctuiz**
efddiv	0	0	0	1	0	0	rD					rA					rB					0	1	0	1	1	1	0	1	0	0	1	EFX	**efddiv**
efdmul	0	0	0	1	0	0	rD					rA					rB					0	1	0	1	1	1	0	1	0	0	0	EFX	**efdmul**
efdnabs	0	0	0	1	0	0	rD					rA					///					0	1	0	1	1	1	0	0	1	0	1	EFX	**efdnabs**
efdneg	0	0	0	1	0	0	rD					rA					///					0	1	0	1	1	1	0	0	1	1	0	EFX	**efdneg**
efdsub	0	0	0	1	0	0	rD					rA					rB					0	1	0	1	1	1	0	0	0	0	1	EFX	**efdsub**
efdtsteq	0	0	0	1	0	0	crfD			/	/	rA					rB					0	1	0	1	1	1	1	1	1	1	0	EFX	**efdtsteq**
efdtstgt	0	0	0	1	0	0	crfD			/	/	rA					rB					0	1	0	1	1	1	1	1	1	0	0	EFX	**efdtstgt**
efdtstlt	0	0	0	1	0	0	crfD			/	/	rA					rB					0	1	0	1	1	1	1	1	1	0	1	EFX	**efdtstlt**
efsabs	0	0	0	1	0	0	rD					rA					///					0	1	0	1	1	0	0	0	1	0	0	EFX	**efsabs**
efsadd	0	0	0	1	0	0	rD					rA					rB					0	1	0	1	1	0	0	0	0	0	0	EFX	**efsadd**
efscfd	0	0	0	1	0	0	rD					0	0	0	0	0	rB					0	1	0	1	1	0	0	1	1	1	1	EFX	**efscfd**
efscfsf	0	0	0	1	0	0	rD					///					rB					0	1	0	1	1	0	1	0	0	1	1	EFX	**efscfsf**
efscfsi	0	0	0	1	0	0	rD					///					rB					0	1	0	1	1	0	1	0	0	0	1	EFX	**efscfsi**
efscfuf	0	0	0	1	0	0	rD					///					rB					0	1	0	1	1	0	1	0	0	1	0	EFX	**efscfuf**
efscfui	0	0	0	1	0	0	rD					///					rB					0	1	0	1	1	0	1	0	0	0	0	EFX	**efscfui**

Table 272. Instructions sorted by opcode (binary) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
efscmpeq	0	0	0	1	0	0	crfD			/	/	rA					rB					0	1	0	1	1	0	0	1	1	1	0	EFX	**efscmpeq**
efscmpgt	0	0	0	1	0	0	crfD			/	/	rA					rB					0	1	0	1	1	0	0	1	1	0	0	EFX	**efscmpgt**
efscmplt	0	0	0	1	0	0	crfD			/	/	rA					rB					0	1	0	1	1	0	0	1	1	0	1	EFX	**efscmplt**
efsctsf	0	0	0	1	0	0	rD					///					rB					0	1	0	1	1	0	1	0	1	1	1	EFX	**efsctsf**
efsctsi	0	0	0	1	0	0	rD					///					rB					0	1	0	1	1	0	1	0	1	0	1	EFX	**efsctsi**
efsctsiz	0	0	0	1	0	0	rD					///					rB					0	1	0	1	1	0	1	1	0	1	0	EFX	**efsctsiz**
efsctuf	0	0	0	1	0	0	rD					///					rB					0	1	0	1	1	0	1	0	1	1	0	EFX	**efsctuf**
efsctui	0	0	0	1	0	0	rD					///					rB					0	1	0	1	1	0	1	0	1	0	0	EFX	**efsctui**
efsctuiz	0	0	0	1	0	0	rD					///					rB					0	1	0	1	1	0	1	1	0	0	0	EFX	**efsctuiz**
efsdiv	0	0	0	1	0	0	rD					rA					rB					0	1	0	1	1	0	0	1	0	0	1	EFX	**efsdiv**
efsmul	0	0	0	1	0	0	rD					rA					rB					0	1	0	1	1	0	0	1	0	0	0	EFX	**efsmul**
efsnabs	0	0	0	1	0	0	rD					rA					///					0	1	0	1	1	0	0	0	1	0	1	EFX	**efsnabs**
efsneg	0	0	0	1	0	0	rD					rA					///					0	1	0	1	1	0	0	0	1	1	0	EFX	**efsneg**
efssub	0	0	0	1	0	0	rD					rA					rB					0	1	0	1	1	0	0	0	0	0	1	EFX	**efssub**
efststeq	0	0	0	1	0	0	crfD			/	/	rA					rB					0	1	0	1	1	0	1	1	1	1	0	EFX	**efststeq**
efststgt	0	0	0	1	0	0	crfD			/	/	rA					rB					0	1	0	1	1	0	1	1	1	0	0	EFX	**efststgt**
efststlt	0	0	0	1	0	0	crfD			/	/	rA					rB					0	1	0	1	1	0	1	1	1	0	1	EFX	**efststlt**
mulli	0	0	0	1	1	1	rD					rA					SIMM																D	**mulli**
subfic	0	0	1	0	0	0	rD					rA					SIMM																D	**subfic**
cmpli	0	0	1	0	1	0	crfD			/	L	rA					UIMM																D	**cmpli**
cmpi	0	0	1	0	1	1	crfD			/	L	rA					SIMM																D	**cmpi**
addic	0	0	1	1	0	0	rD					rA					SIMM																D	**addic**
addic.	0	0	1	1	0	1	rD					rA					SIMM																D	**addic.**
addi	0	0	1	1	1	0	rD					rA					SIMM																D	**addi**
addis	0	0	1	1	1	1	rD					rA					SIMM																D	**addis**
bc	0	1	0	0	0	0	BO					BI					BD														0	0	B	**bc**
bca	0	1	0	0	0	0	BO					BI					BD														1	0	B	**bca**
bcl	0	1	0	0	0	0	BO					BI					BD														0	1	B	**bcl**
bcla	0	1	0	0	0	0	BO					BI					BD														1	1	B	**bcla**
sc	0	1	0	0	0	1	///																								1	/	SC	**sc**
b	0	1	0	0	1	0	LI																								0	0	I	**b**
ba	0	1	0	0	1	0	LI																								1	0	I	**ba**
bl	0	1	0	0	1	0	LI																								0	1	I	**bl**
bla	0	1	0	0	1	0	LI																								1	1	I	**bla**
rfci	0	1	0	0	1	1	///															0	0	0	0	1	1	0	0	1	1	/	XL	**rfci**
rfmci [1]	0	1	0	0	1	1	///															0	0	0	0	1	0	0	1	1	0	/	XL	**rfmci**
mcrf	0	1	0	0	1	1	crfD			//		crfS					///					0	0	0	0	0	0	0	0	0	0	/	XL	**mcrf**
bclr	0	1	0	0	1	1	BO					BI					///					0	0	0	0	0	1	0	0	0	0	0	XL	**bclr**

Table 272. Instructions sorted by opcode (binary) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
bclrl	0	1	0	0	1	1	BO					BI					///					0	0	0	0	0	1	0	0	0	0	1	XL	**bclrl**
crnor	0	1	0	0	1	1	crbD					crbA					crbB					0	0	0	0	1	0	0	0	0	1	/	XL	**crnor**
rfi (1)	0	1	0	0	1	1	///															0	0	0	0	1	1	0	0	1	0	/	XL	**rfi**
crandc	0	1	0	0	1	1	crbD					crbA					crbB					0	0	1	0	0	0	0	0	0	1	/	XL	**crandc**
isync	0	1	0	0	1	1	///															0	0	1	0	0	1	0	1	1	0	/	XL	**isync**
crxor	0	1	0	0	1	1	crbD					crbA					crbB					0	0	1	1	0	0	0	0	0	1	/	XL	**crxor**
crand	0	1	0	0	1	1	crbD					crbA					crbB					0	1	0	0	0	0	0	0	0	1	/	XL	**crand**
crnand	0	1	0	0	1	1	crbD					crbA					crbB					0	0	1	1	1	0	0	0	0	1	/	XL	**crnand**
creqv	0	1	0	0	1	1	crbD					crbA					crbB					0	1	0	0	1	0	0	0	0	1	/	XL	**creqv**
crorc	0	1	0	0	1	1	crbD					crbA					crbB					0	1	1	0	1	0	0	0	0	1	/	XL	**crorc**
cror	0	1	0	0	1	1	crbD					crbA					crbB					0	1	1	1	0	0	0	0	0	1	/	XL	**cror**
bcctr	0	1	0	0	1	1	BO					BI					///					1	0	0	0	0	1	0	0	0	0	0	XL	**bcctr**
bcctrl	0	1	0	0	1	1	BO					BI					///					1	0	0	0	0	1	0	0	0	0	1	XL	**bcctrl**
rlwimi	0	1	0	1	0	0	rS					rA					SH					MB					ME					0	M	**rlwimi**
rlwimi.	0	1	0	1	0	0	rS					rA					SH					MB					ME					1	M	**rlwimi.**
rlwinm	0	1	0	1	0	1	rS					rA					SH					MB					ME					0	M	**rlwinm**
rlwinm.	0	1	0	1	0	1	rS					rA					SH					MB					ME					1	M	**rlwinm.**
rlwnm	0	1	0	1	1	1	rS					rA					rB					MB					ME					0	M	**rlwnm**
rlwnm.	0	1	0	1	1	1	rS					rA					rB					MB					ME					1	M	**rlwnm.**
ori	0	1	1	0	0	0	rS					rA					UIMM															D	**ori**
oris	0	1	1	0	0	1	rS					rA					UIMM															D	**oris**
xori	0	1	1	0	1	0	rS					rA					UIMM															D	**xori**
xoris	0	1	1	0	1	1	rS					rA					UIMM															D	**xoris**
andi.	0	1	1	1	0	0	rS					rA					UIMM															D	**andi.**
andis.	0	1	1	1	0	1	rS					rA					UIMM															D	**andis.**
dcblc	0	1	1	1	1	1	CT					rA					rB					0	1	1	0	0	0	0	1	1	0	0	X	**dcblc**
dcbtls	0	1	1	1	1	1	CT					rA					rB					0	0	1	0	1	0	0	1	1	0	0	X	**dcbtls**
dcbtstls	0	1	1	1	1	1	CT					rA					rB					0	0	1	0	0	0	0	1	1	0	0	X	**dcbtstls**
evabs	0	1	1	1	1	1	rD					rA					///					0	1	0	0	0	0	0	1	0	0	0	EVX	**evabs**
evaddiw	0	1	1	1	1	1	rD					UIMM					rB					0	1	0	0	0	0	0	0	0	1	0	EVX	**evaddiw**
evaddsmiaaw	0	1	1	1	1	1	rD					rA					///					1	0	0	1	1	0	0	1	0	0	1	EVX	**evaddsmiaaw**
evaddssiaaw	0	1	1	1	1	1	rD					rA					///					1	0	0	1	1	0	0	0	0	0	1	EVX	**evaddssiaaw**
evaddumiaaw	0	1	1	1	1	1	rD					rA					///					1	0	0	1	1	0	0	1	0	0	0	EVX	**evaddumiaaw**
evaddusiaaw	0	1	1	1	1	1	rD					rA					///					1	0	0	1	1	0	0	0	0	0	0	EVX	**evaddusiaaw**
evaddw	0	1	1	1	1	1	rD					rA					rB					0	1	0	0	0	0	0	0	0	0	0	EVX	**evaddw**
evand	0	1	1	1	1	1	rD					rA					rB					0	1	0	0	0	0	1	0	0	0	1	EVX	**evand**
evandc	0	1	1	1	1	1	rD					rA					rB					0	1	0	0	0	0	1	0	0	1	0	EVX	**evandc**

Table 272. Instructions sorted by opcode (binary) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
evcmpeq	0	1	1	1	1	1	crfD			/	/	rA					rB					0	1	0	0	0	1	1	0	1	0	0	EVX	evcmpeq
evcmpgts	0	1	1	1	1	1	crfD			/	/	rA					rB					0	1	0	0	0	1	1	0	0	0	1	EVX	evcmpgts
evcmpgtu	0	1	1	1	1	1	crfD			/	/	rA					rB					0	1	0	0	0	1	1	0	0	0	0	EVX	evcmpgtu
evcmplts	0	1	1	1	1	1	crfD			/	/	rA					rB					0	1	0	0	0	1	1	0	0	1	1	EVX	evcmplts
evcmpltu	0	1	1	1	1	1	crfD			/	/	rA					rB					0	1	0	0	0	1	1	0	0	1	0	EVX	evcmpltu
evcntlsw	0	1	1	1	1	1	rD					rA					///					0	1	0	0	0	0	0	1	1	1	0	EVX	evcntlsw
evcntlzw	0	1	1	1	1	1	rD					rA					///					0	1	0	0	0	0	0	1	1	0	1	EVX	evcntlzw
evdivws	0	1	1	1	1	1	rD					rA					rB					1	0	0	1	1	0	0	0	1	1	0	EVX	evdivws
evdivwu	0	1	1	1	1	1	rD					rA					rB					1	0	0	1	1	0	0	0	1	1	1	EVX	evdivwu
eveqv	0	1	1	1	1	1	rD					rA					rB					0	1	0	0	0	0	1	1	0	0	1	EVX	eveqv
evextsb	0	1	1	1	1	1	rD					rA					///					0	1	0	0	0	0	0	1	0	1	0	EVX	evextsb
evextsh	0	1	1	1	1	1	rD					rA					///					0	1	0	0	0	0	0	1	0	1	1	EVX	evextsh
evfsabs	0	1	1	1	1	1	rD					rA					///					0	1	0	1	0	0	0	0	1	0	0	EVX	evfsabs
evfsadd	0	1	1	1	1	1	rD					rA					rB					0	1	0	1	0	0	0	0	0	0	0	EVX	evfsadd
evfscfsf	0	1	1	1	1	1	rD					///					rB					0	1	0	1	0	0	1	0	0	1	1	EVX	evfscfsf
evfscfsi	0	1	1	1	1	1	rD					///					rB					0	1	0	1	0	0	1	0	0	0	1	EVX	evfscfsi
evfscfuf	0	1	1	1	1	1	rD					///					rB					0	1	0	1	0	0	1	0	0	1	0	EVX	evfscfuf
evfscfui	0	1	1	1	1	1	rD					///					rB					0	1	0	1	0	0	1	0	0	0	0	EVX	evfscfui
evfscmpeq	0	1	1	1	1	1	crfD			/	/	rA					rB					0	1	0	1	0	0	0	1	1	1	0	EVX	evfscmpeq
evfscmpgt	0	1	1	1	1	1	crfD			/	/	rA					rB					0	1	0	1	0	0	0	1	1	0	0	EVX	evfscmpgt
evfscmplt	0	1	1	1	1	1	crfD			/	/	rA					rB					0	1	0	1	0	0	0	1	1	0	1	EVX	evfscmplt
evfsctsf	0	1	1	1	1	1	rD					///					rB					0	1	0	1	0	0	1	0	1	1	1	EVX	evfsctsf
evfsctsi	0	1	1	1	1	1	rD					///					rB					0	1	0	1	0	0	1	0	1	0	1	EVX	evfsctsi
evfsctsiz	0	1	1	1	1	1	rD					///					rB					0	1	0	1	0	0	1	1	0	1	0	EVX	evfsctsiz
evfsctuf	0	1	1	1	1	1	rD					///					rB					0	1	0	1	0	0	1	0	1	1	0	EVX	evfsctuf
evfsctui	0	1	1	1	1	1	rD					///					rB					0	1	0	1	0	0	1	0	1	0	0	EVX	evfsctui
evfsctuiz	0	1	1	1	1	1	rD					///					rB					0	1	0	1	0	0	1	1	0	0	0	EVX	evfsctuiz
evfsdiv	0	1	1	1	1	1	rD					rA					rB					0	1	0	1	0	0	0	1	0	0	1	EVX	evfsdiv
evfsmul	0	1	1	1	1	1	rD					rA					rB					0	1	0	1	0	0	0	1	0	0	0	EVX	evfsmul
evfsnabs	0	1	1	1	1	1	rD					rA					///					0	1	0	1	0	0	0	0	1	0	1	EVX	evfsnabs
evfsneg	0	1	1	1	1	1	rD					rA					///					0	1	0	1	0	0	0	0	1	1	0	EVX	evfsneg
evfssub	0	1	1	1	1	1	rD					rA					rB					0	1	0	1	0	0	0	0	0	0	1	EVX	evfssub

Table 272. Instructions sorted by opcode (binary) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
evfststeq	0	1	1	1	1	1	crfD			/	/	rA					rB					0	1	0	1	0	0	1	1	1	1	0	EVX	evfststeq
evfststgt	0	1	1	1	1	1	crfD			/	/	rA					rB					0	1	0	1	0	0	1	1	1	0	0	EVX	evfststgt
evfststlt	0	1	1	1	1	1	crfD			/	/	rA					rB					0	1	0	1	0	0	1	1	1	0	1	EVX	evfststlt
evldd	0	1	1	1	1	1	rD					rA					UIMM (2)					0	1	1	0	0	0	0	0	0	0	1	EVX	evldd
evlddx	0	1	1	1	1	1	rD					rA					rB					0	1	1	0	0	0	0	0	0	0	0	EVX	evlddx
evldh	0	1	1	1	1	1	rD					rA					UIMM 2					0	1	1	0	0	0	0	0	1	0	1	EVX	evldh
evldhx	0	1	1	1	1	1	rD					rA					rB					0	1	1	0	0	0	0	0	1	0	0	EVX	evldhx
evldw	0	1	1	1	1	1	rD					rA					UIMM 2					0	1	1	0	0	0	0	0	0	1	1	EVX	evldw
evldwx	0	1	1	1	1	1	rD					rA					rB					0	1	1	0	0	0	0	0	0	1	0	EVX	evldwx
evlhhesplat	0	1	1	1	1	1	rD					rA					UIMM 3					0	1	1	0	0	0	0	1	0	0	1	EVX	evlhhesplat
evlhhesplatx	0	1	1	1	1	1	rD					rA					rB					0	1	1	0	0	0	0	1	0	0	0	EVX	evlhhesplatx
evlhhossplat	0	1	1	1	1	1	rD					rA					UIMM 3					0	1	1	0	0	0	0	1	1	1	1	EVX	evlhhossplat
evlhhossplatx	0	1	1	1	1	1	rD					rA					rB					0	1	1	0	0	0	0	1	1	1	0	EVX	evlhhossplatx
evlhhousplat	0	1	1	1	1	1	rD					rA					UIMM 3					0	1	1	0	0	0	0	1	1	0	1	EVX	evlhhousplat
evlhhousplatx	0	1	1	1	1	1	rD					rA					rB					0	1	1	0	0	0	0	1	1	0	0	EVX	evlhhousplatx
evlwhe	0	1	1	1	1	1	rD					rA					UIMM 3					0	1	1	0	0	0	1	0	0	0	1	EVX	evlwhe
evlwhex	0	1	1	1	1	1	rD					rA					rB					0	1	1	0	0	0	1	0	0	0	0	EVX	evlwhex
evlwhos	0	1	1	1	1	1	rD					rA					UIMM 4					0	1	1	0	0	0	1	0	1	1	1	EVX	evlwhos
evlwhosx	0	1	1	1	1	1	rD					rA					rB					0	1	1	0	0	0	1	0	1	1	0	EVX	evlwhosx
evlwhou	0	1	1	1	1	1	rD					rA					UIMM 4					0	1	1	0	0	0	1	0	1	0	1	EVX	evlwhou
evlwhoux	0	1	1	1	1	1	rD					rA					rB					0	1	1	0	0	0	1	0	1	0	0	EVX	evlwhoux
evlwhsplat	0	1	1	1	1	1	rD					rA					UIMM 4					0	1	1	0	0	0	1	1	1	0	1	EVX	evlwhsplat
evlwhsplatx	0	1	1	1	1	1	rD					rA					rB					0	1	1	0	0	0	1	1	1	0	0	EVX	evlwhsplatx
evlwwsplat	0	1	1	1	1	1	rD					rA					UIMM 4					0	1	1	0	0	0	1	1	0	0	1	EVX	evlwwsplat
evlwwsplatx	0	1	1	1	1	1	rD					rA					rB					0	1	1	0	0	0	1	1	0	0	0	EVX	evlwwsplatx
evmergehi	0	1	1	1	1	1	rD					rA					rB					0	1	0	0	0	1	0	1	1	0	0	EVX	evmergehi
evmergehilo	0	1	1	1	1	1	rD					rA					rB					0	1	0	0	0	1	0	1	1	1	0	EVX	evmergehilo
evmergelo	0	1	1	1	1	1	rD					rA					rB					0	1	0	0	0	1	0	1	1	0	1	EVX	evmergelo
evmergelohi	0	1	1	1	1	1	rD					rA					rB					0	1	0	0	0	1	0	1	1	1	1	EVX	evmergelohi

Table 272. Instructions sorted by opcode (binary) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
evmhegsmfaa	0	1	1	1	1	1	rD					rA					rB					1	0	1	0	0	1	0	1	0	1	1	EVX	evmhegsmfaa
evmhegsmfan	0	1	1	1	1	1	rD					rA					rB					1	0	1	1	0	1	0	1	0	1	1	EVX	evmhegsmfan
evmhegsmiaa	0	1	1	1	1	1	rD					rA					rB					1	0	1	0	0	1	0	1	0	0	1	EVX	evmhegsmiaa
evmhegsmian	0	1	1	1	1	1	rD					rA					rB					1	0	1	1	0	1	0	1	0	0	1	EVX	evmhegsmian
evmhegumiaa	0	1	1	1	1	1	rD					rA					rB					1	0	1	0	0	1	0	1	0	0	0	EVX	evmhegumiaa
evmhegumian	0	1	1	1	1	1	rD					rA					rB					1	0	1	1	0	1	0	1	0	0	0	EVX	evmhegumian
evmhesmf	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	0	0	0	1	0	1	1	EVX	evmhesmf
evmhesmfa	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	0	1	0	1	0	1	1	EVX	evmhesmfa
evmhesmfaaw	0	1	1	1	1	1	rD					rA					rB					1	0	1	0	0	0	0	1	0	1	1	EVX	evmhesmfaaw
evmhesmfanw	0	1	1	1	1	1	rD					rA					rB					1	0	1	1	0	0	0	1	0	1	1	EVX	evmhesmfanw
evmhesmi	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	0	0	0	1	0	0	1	EVX	evmhesmi
evmhesmia	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	0	1	0	1	0	0	1	EVX	evmhesmia
evmhesmiaaw	0	1	1	1	1	1	rD					rA					rB					1	0	1	0	0	0	0	1	0	0	1	EVX	evmhesmiaaw
evmhesmianw	0	1	1	1	1	1	rD					rA					rB					1	0	1	1	0	0	0	1	0	0	1	EVX	evmhesmianw
evmhessf	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	0	0	0	0	0	1	1	EVX	evmhessf
evmhessfa	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	0	1	0	0	0	1	1	EVX	evmhessfa
evmhessfaaw	0	1	1	1	1	1	rD					rA					rB					1	0	1	0	0	0	0	0	0	1	1	EVX	evmhessfaaw
evmhessfanw	0	1	1	1	1	1	rD					rA					rB					1	0	1	1	0	0	0	0	0	1	1	EVX	evmhessfanw
evmhessiaaw	0	1	1	1	1	1	rD					rA					rB					1	0	1	0	0	0	0	0	0	0	1	EVX	evmhessiaaw
evmhessianw	0	1	1	1	1	1	rD					rA					rB					1	0	1	1	0	0	0	0	0	0	1	EVX	evmhessianw
evmheumi	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	0	0	0	1	0	0	0	EVX	evmheumi
evmheumia	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	0	1	0	1	0	0	0	EVX	evmheumia
evmheumiaaw	0	1	1	1	1	1	rD					rA					rB					1	0	1	0	0	0	0	1	0	0	0	EVX	evmheumiaaw
evmheumianw	0	1	1	1	1	1	rD					rA					rB					1	0	1	1	0	0	0	1	0	0	0	EVX	evmheumianw
evmheusiaaw	0	1	1	1	1	1	rD					rA					rB					1	0	1	0	0	0	0	0	0	0	0	EVX	evmheusiaaw
evmheusianw	0	1	1	1	1	1	rD					rA					rB					1	0	1	1	0	0	0	0	0	0	0	EVX	evmheusianw

Table 272. Instructions sorted by opcode (binary) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
evmhogsmfaa	0	1	1	1	1	1	rD					rA					rB					1	0	1	0	0	1	0	1	1	1	1	EVX	evmhogsmfaa
evmhogsmfan	0	1	1	1	1	1	rD					rA					rB					1	0	1	1	0	1	0	1	1	1	1	EVX	evmhogsmfan
evmhogsmiaa	0	1	1	1	1	1	rD					rA					rB					1	0	1	0	0	1	0	1	1	0	1	EVX	evmhogsmiaa
evmhogsmian	0	1	1	1	1	1	rD					rA					rB					1	0	1	1	0	1	0	1	1	0	1	EVX	evmhogsmian
evmhogumiaa	0	1	1	1	1	1	rD					rA					rB					1	0	1	0	0	1	0	1	1	0	0	EVX	evmhogumiaa
evmhogumian	0	1	1	1	1	1	rD					rA					rB					1	0	1	1	0	1	0	1	1	0	0	EVX	evmhogumian
evmhosmf	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	0	0	0	1	1	1	1	EVX	evmhosmf
evmhosmfa	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	0	1	0	1	1	1	1	EVX	evmhosmfa
evmhosmfaaw	0	1	1	1	1	1	rD					rA					rB					1	0	1	0	0	0	0	1	1	1	1	EVX	evmhosmfaaw
evmhosmfanw	0	1	1	1	1	1	rD					rA					rB					1	0	1	1	0	0	0	1	1	1	1	EVX	evmhosmfanw
evmhosmi	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	0	0	0	1	1	0	1	EVX	evmhosmi
evmhosmia	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	0	1	0	1	1	0	1	EVX	evmhosmia
evmhosmiaaw	0	1	1	1	1	1	rD					rA					rB					1	0	1	0	0	0	0	1	1	0	1	EVX	evmhosmiaaw
evmhosmianw	0	1	1	1	1	1	rD					rA					rB					1	0	1	1	0	0	0	1	1	0	1	EVX	evmhosmianw
evmhossf	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	0	0	0	0	1	1	1	EVX	evmhossf
evmhossfa	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	0	1	0	0	1	1	1	EVX	evmhossfa
evmhossfaaw	0	1	1	1	1	1	rD					rA					rB					1	0	1	0	0	0	0	0	1	1	1	EVX	evmhossfaaw
evmhossfanw	0	1	1	1	1	1	rD					rA					rB					1	0	1	1	0	0	0	0	1	1	1	EVX	evmhossfanw
evmhossiaaw	0	1	1	1	1	1	rD					rA					rB					1	0	1	0	0	0	0	0	1	0	1	EVX	evmhossiaaw
evmhossianw	0	1	1	1	1	1	rD					rA					rB					1	0	1	1	0	0	0	0	1	0	1	EVX	evmhossianw
evmhoumi	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	0	0	0	1	1	0	0	EVX	evmhoumi
evmhoumia	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	0	1	0	1	1	0	0	EVX	evmhoumia
evmhoumiaaw	0	1	1	1	1	1	rD					rA					rB					1	0	1	0	0	0	0	1	1	0	0	EVX	evmhoumiaaw
evmhoumianw	0	1	1	1	1	1	rD					rA					rB					1	0	1	1	0	0	0	1	1	0	0	EVX	evmhoumianw
evmhousiaaw	0	1	1	1	1	1	rD					rA					rB					1	0	1	0	0	0	0	0	1	0	0	EVX	evmhousiaaw
evmhousianw	0	1	1	1	1	1	rD					rA					rB					1	0	1	1	0	0	0	0	1	0	0	EVX	evmhousianw

Table 272. Instructions sorted by opcode (binary) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
evmra	0	1	1	1	1	1	rD					rA					///					1	0	0	1	1	0	0	0	1	0	0	EVX	**evmra**
evmwhgsmfaa	0	1	1	1	1	1	rD					rA					rB					1	0	1	0	1	1	0	1	1	1	1	EVX	**evmwhgsmfaa**
evmwhgsmfan	0	1	1	1	1	1	rD					rA					rB					1	0	1	1	1	0	1	1	1	1	1	EVX	**evmwhgsmfan**
evmwhgsmiaa	0	1	1	1	1	1	rD					rA					rB					1	0	1	0	1	1	0	1	1	0	1	EVX	**evmwhgsmiaa**
evmwhgsmian	0	1	1	1	1	1	rD					rA					rB					1	0	1	1	1	0	1	1	1	0	1	EVX	**evmwhgsmian**
evmwhgssfaa	0	1	1	1	1	1	rD					rA					rB					1	0	1	0	1	1	0	0	1	1	1	EVX	**evmwhgssfaa**
evmwhgssfan	0	1	1	1	1	1	rD					rA					rB					1	0	1	1	1	0	1	0	1	1	1	EVX	**evmwhgssfan**
evmwhgumiaa	0	1	1	1	1	1	rD					rA					rB					1	0	1	0	1	1	0	1	1	0	0	EVX	**evmwhgumiaa**
evmwhgumian	0	1	1	1	1	1	rD					rA					rB					1	0	1	1	1	0	1	1	1	0	0	EVX	**evmwhgumian**
evmwhsmf	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	1	0	0	1	1	1	1	EVX	**evmwhsmf**
evmwhsmfa	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	1	1	0	1	1	1	1	EVX	**evmwhsmfa**
evmwhsmfaaw	0	1	1	1	1	1	rD					rA					rB					1	0	1	0	1	0	0	1	1	1	1	EVX	**evmwhsmfaaw**
evmwhsmfanw	0	1	1	1	1	1	rD					rA					rB					1	0	1	1	1	0	0	1	1	1	1	EVX	**evmwhsmfanw**
evmwhsmi	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	1	0	0	1	1	0	1	EVX	**evmwhsmi**
evmwhsmia	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	1	1	0	1	1	0	1	EVX	**evmwhsmia**
evmwhsmiaaw	0	1	1	1	1	1	rD					rA					rB					1	0	1	0	1	0	0	1	1	0	1	EVX	**evmwhsmiaaw**
evmwhsmianw	0	1	1	1	1	1	rD					rA					rB					1	0	1	1	1	0	0	1	1	0	1	EVX	**evmwhsmianw**
evmwhssf	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	1	0	0	0	1	1	1	EVX	**evmwhssf**
evmwhssfa	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	1	1	0	0	1	1	1	EVX	**evmwhssfa**
evmwhssfaaw	0	1	1	1	1	1	rD					rA					rB					1	0	1	0	1	0	0	0	1	1	1	EVX	**evmwhssfaaw**
evmwhssfanw	0	1	1	1	1	1	rD					rA					rB					1	0	1	1	1	0	0	0	1	1	1	EVX	**evmwhssfanw**
evmwhsianw	0	1	1	1	1	1	rD					rA					rB					1	0	1	1	1	0	0	0	1	0	1	EVX	**evmwhsianw**
evmwhssmaaw	0	1	1	1	1	1	rD					rA					rB					1	0	1	0	1	0	0	0	1	0	1	EVX	**evmwhssmaaw**
evmwhumi	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	1	0	0	1	1	0	0	EVX	**evmwhumi**
evmwhumia	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	1	1	0	1	1	0	0	EVX	**evmwhumia**
evmwhusiaaw	0	1	1	1	1	1	rD					rA					rB					1	0	1	0	1	0	0	0	1	0	0	EVX	**evmwhusiaaw**

Table 272. Instructions sorted by opcode (binary) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
evmwhusianw	0	1	1	1	1	1	rD					rA					rB					1	0	1	1	1	0	0	0	1	0	0	EVX	evmwhusianw
evmwlsmf	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	1	0	0	1	0	1	1	EVX	evmwlsmf
evmwlsmfa	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	1	1	0	1	0	1	1	EVX	evmwlsmfa
evmwlsmfaaw	0	1	1	1	1	1	rD					rA					rB					1	0	1	0	1	0	0	1	0	1	1	EVX	evmwlsmfaaw
evmwlsmfanw	0	1	1	1	1	1	rD					rA					rB					1	0	1	1	1	0	0	1	0	1	1	EVX	evmwlsmfanw
evmwlsmiaaw	0	1	1	1	1	1	rD					rA					rB					1	0	1	0	1	0	0	1	0	0	1	EVX	evmwlsmiaaw
evmwlsmianw	0	1	1	1	1	1	rD					rA					rB					1	0	1	1	1	0	0	1	0	0	1	EVX	evmwlsmianw
evmwlssf	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	1	0	0	0	0	1	1	EVX	evmwlssf
evmwlssfa	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	1	1	0	0	0	1	1	EVX	evmwlssfa
evmwlssfaaw	0	1	1	1	1	1	rD					rA					rB					1	0	1	0	1	0	0	0	0	1	1	EVX	evmwlssfaaw
evmwlssfanw	0	1	1	1	1	1	rD					rA					rB					1	0	1	1	1	0	0	0	0	1	1	EVX	evmwlssfanw
evmwlssiaaw	0	1	1	1	1	1	rD					rA					rB					1	0	1	0	1	0	0	0	0	0	1	EVX	evmwlssiaaw
evmwlssianw	0	1	1	1	1	1	rD					rA					rB					1	0	1	1	1	0	0	0	0	0	1	EVX	evmwlssianw
evmwlumi	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	1	0	0	1	0	0	0	EVX	evmwlumi
evmwlumia	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	1	1	0	1	0	0	0	EVX	evmwlumia
evmwlumiaaw	0	1	1	1	1	1	rD					rA					rB					1	0	1	0	1	0	0	1	0	0	0	EVX	evmwlumiaaw
evmwlumianw	0	1	1	1	1	1	rD					rA					rB					1	0	1	1	1	0	0	1	0	0	0	EVX	evmwlumianw
evmwlusiaaw	0	1	1	1	1	1	rD					rA					rB					1	0	1	0	1	0	0	0	0	0	0	EVX	evmwlusiaaw
evmwlusianw	0	1	1	1	1	1	rD					rA					rB					1	0	1	1	1	0	0	0	0	0	0	EVX	evmwlusianw
evmwsmf	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	1	0	1	1	0	1	1	EVX	evmwsmf
evmwsmfa	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	1	1	1	1	0	1	1	EVX	evmwsmfa
evmwsmfaa	0	1	1	1	1	1	rD					rA					rB					1	0	1	0	1	0	1	1	0	1	1	EVX	evmwsmfaa
evmwsmfan	0	1	1	1	1	1	rD					rA					rB					1	0	1	1	1	0	1	1	0	1	1	EVX	evmwsmfan
evmwsmi	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	1	0	1	1	0	0	1	EVX	evmwsmi
evmwsmia	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	1	1	1	1	0	0	1	EVX	evmwsmia
evmwsmiaa	0	1	1	1	1	1	rD					rA					rB					1	0	1	0	1	0	1	1	0	0	1	EVX	evmwsmiaa

Table 272. Instructions sorted by opcode (binary) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
evmwsmian	0	1	1	1	1	1	rD					rA					rB					1	0	1	1	1	0	1	1	0	0	1	EVX	**evmwsmian**
evmwssf	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	1	0	1	0	0	1	1	EVX	**evmwssf**
evmwssfa	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	1	1	1	0	0	1	1	EVX	**evmwssfa**
evmwssfaa	0	1	1	1	1	1	rD					rA					rB					1	0	1	0	1	0	1	0	0	1	1	EVX	**evmwssfaa**
evmwssfan	0	1	1	1	1	1	rD					rA					rB					1	0	1	1	1	0	1	0	0	1	1	EVX	**evmwssfan**
evmwumi	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	1	0	1	1	0	0	0	EVX	**evmwumi**
evmwumia	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	1	1	1	1	0	0	0	EVX	**evmwumia**
evmwumiaa	0	1	1	1	1	1	rD					rA					rB					1	0	1	0	1	0	1	1	0	0	0	EVX	**evmwumiaa**
evmwumian	0	1	1	1	1	1	rD					rA					rB					1	0	1	1	1	0	1	1	0	0	0	EVX	**evmwumian**
evnand	0	1	1	1	1	1	rD					rA					rB					0	1	0	0	0	0	1	1	1	1	0	EVX	**evnand**
evneg	0	1	1	1	1	1	rD					rA					///					0	1	0	0	0	0	0	1	0	0	1	EVX	**evneg**
evnor	0	1	1	1	1	1	rD					rA					rB					0	1	0	0	0	0	1	1	0	0	0	EVX	**evnor**
evor	0	1	1	1	1	1	rD					rA					rB					0	1	0	0	0	0	1	0	1	1	1	EVX	**evor**
evorc	0	1	1	1	1	1	rD					rA					rB					0	1	0	0	0	0	1	1	0	1	1	EVX	**evorc**
evrlw	0	1	1	1	1	1	rD					rA					rB					0	1	0	0	0	1	0	1	0	0	0	EVX	**evrlw**
evrlwi	0	1	1	1	1	1	rD					rA					UIMM					0	1	0	0	0	1	0	1	0	1	0	EVX	**evrlwi**
evrndw	0	1	1	1	1	1	rD					rA					UIMM					0	1	0	0	0	0	0	1	1	0	0	EVX	**evrndw**
evsel	0	1	1	1	1	1	rD					rA					rB					0	1	0	0	1	1	1	1	crfS			EVX	**evsel**
evslw	0	1	1	1	1	1	rD					rA					rB					0	1	0	0	0	1	0	0	1	0	0	EVX	**evslw**
evslwi	0	1	1	1	1	1	rD					rA					UIMM					0	1	0	0	0	1	0	0	1	1	0	EVX	**evslwi**
evsplatfi	0	1	1	1	1	1	rD					SIMM					///					0	1	0	0	0	1	0	1	0	1	1	EVX	**evsplatfi**
evsplati	0	1	1	1	1	1	rD					SIMM					///					0	1	0	0	0	1	0	1	0	0	1	EVX	**evsplati**
evsrwis	0	1	1	1	1	1	rD					rA					UIMM					0	1	0	0	0	1	0	0	0	1	1	EVX	**evsrwis**
evsrwiu	0	1	1	1	1	1	rD					rA					UIMM					0	1	0	0	0	1	0	0	0	1	0	EVX	**evsrwiu**
evsrws	0	1	1	1	1	1	rD					rA					rB					0	1	0	0	0	1	0	0	0	0	1	EVX	**evsrws**
evsrwu	0	1	1	1	1	1	rD					rA					rB					0	1	0	0	0	1	0	0	0	0	0	EVX	**evsrwu**
evstdd	0	1	1	1	1	1	rD					rA					UIMM [2]					0	1	1	0	0	1	0	0	0	0	1	EVX	**evstdd**
evstddx	0	1	1	1	1	1	rS					rA					rB					0	1	1	0	0	1	0	0	0	0	0	EVX	**evstddx**
evstdh	0	1	1	1	1	1	rS					rA					UIMM [2]					0	1	1	0	0	1	0	0	1	0	1	EVX	**evstdh**
evstdhx	0	1	1	1	1	1	rS					rA					rB					0	1	1	0	0	1	0	0	1	0	0	EVX	**evstdhx**
evstdw	0	1	1	1	1	1	rS					rA					UIMM [2]					0	1	1	0	0	1	0	0	0	1	1	EVX	**evstdw**
evstdwx	0	1	1	1	1	1	rS					rA					rB					0	1	1	0	0	1	0	0	0	1	0	EVX	**evstdwx**
evstwhe	0	1	1	1	1	1	rS					rA					UIMM [4]					0	1	1	0	0	1	1	0	0	0	1	EVX	**evstwhe**

Table 272. Instructions sorted by opcode (binary) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
evstwhex	0	1	1	1	1	1	rS					rA					rB					0	1	1	0	0	1	1	0	0	0	0	EVX	**evstwhex**
evstwho	0	1	1	1	1	1	rS					rA					UIMM [4]					0	1	1	0	0	1	1	0	1	0	1	EVX	**evstwho**
evstwhox	0	1	1	1	1	1	rS					rA					rB					0	1	1	0	0	1	1	0	1	0	0	EVX	**evstwhox**
evstwwe	0	1	1	1	1	1	rS					rA					UIMM [4]					0	1	1	0	0	1	1	1	0	0	1	EVX	**evstwwe**
evstwwex	0	1	1	1	1	1	rS					rA					rB					0	1	1	0	0	1	1	1	0	0	0	EVX	**evstwwex**
evstwwo	0	1	1	1	1	1	rS					rA					UIMM [4]					0	1	1	0	0	1	1	1	1	0	1	EVX	**evstwwo**
evstwwox	0	1	1	1	1	1	rS					rA					rB					0	1	1	0	0	1	1	1	1	0	0	EVX	**evstwwox**
evsubfsmiaaw	0	1	1	1	1	1	rD					rA					///					1	0	0	1	1	0	0	1	0	1	1	EVX	**evsubfsmiaaw**
evsubfssiaaw	0	1	1	1	1	1	rD					rA					///					1	0	0	1	1	0	0	0	0	1	1	EVX	**evsubfssiaaw**
evsubfumiaaw	0	1	1	1	1	1	rD					rA					///					1	0	0	1	1	0	0	1	0	1	0	EVX	**evsubfumiaaw**
evsubfusiaaw	0	1	1	1	1	1	rD					rA					///					1	0	0	1	1	0	0	0	0	1	0	EVX	**evsubfusiaaw**
evsubfw	0	1	1	1	1	1	rD					rA					rB					0	1	0	0	0	0	0	0	1	0	0	EVX	**evsubfw**
evsubifw	0	1	1	1	1	1	rD					UIMM					rB					0	1	0	0	0	0	0	0	1	1	0	EVX	**evsubifw**
evxor	0	1	1	1	1	1	rD					rA					rB					0	1	0	0	0	0	1	0	1	1	0	EVX	**evxor**
icblc	0	1	1	1	1	1	CT					rA					rB					0	0	1	1	1	0	0	1	1	0	0	X	**icblc**
icbt	0	1	1	1	1	1	CT					rA					rB					0	0	0	0	0	1	0	1	1	0	/	X	**icbt**
icbtls	0	1	1	1	1	1	CT					rA					rB					0	1	1	1	1	0	0	1	1	0	0	X	**icbtls**
isel	0	1	1	1	1	1	rD					rA					rB					crb					0	1	1	1	1	0	X	**isel**
mbar	0	1	1	1	1	1	MO					///										1	1	0	1	0	1	0	1	1	0	/	X	**mbar**
mfdcr	0	1	1	1	1	1	rD					DCRN5–9					DCRN0–4					0	1	0	1	0	0	0	0	1	1	/	XFX	**mfdcr**
mfpmr	0	1	1	1	1	1	rD					PMRN5–9					PMRN0–4					0	1	0	1	0	0	1	1	1	0	0	XFX	**mfpmr**
msync	0	1	1	1	1	1	///															1	0	0	1	0	1	0	1	1	0	/	X	**msync**
mtdcr	0	1	1	1	1	1	rS					DCRN5–9					DCRN0–4					0	1	1	1	0	0	0	0	1	1	/	XFX	**mtdcr**
mtpmr	0	1	1	1	1	1	rS					PMRN5–9					PMRN0–4					0	1	1	1	0	0	1	1	1	0	0	XFX	**mtpmr**
tlbivax	0	1	1	1	1	1	///					rA					rB					1	1	0	0	0	1	0	0	1	0	/	X	**tlbivax**
tlbre	0	1	1	1	1	1	///[3]															1	1	1	0	1	1	0	0	1	0	/	X	**tlbre**
tlbsx	0	1	1	1	1	1	///[5]					rA					rB					1	1	1	0	0	1	0	0	1	0	/[5]	X	**tlbsx**
tlbwe	0	1	1	1	1	1	///[5]															1	1	1	1	0	1	0	0	1	0	/	X	**tlbwe**
wait	0	1	1	1	1	1	///															0	0	0	0	1	1	1	1	1	0	/		**wait**
wrtee	0	1	1	1	1	1	rS					///										0	0	1	0	0	0	0	0	1	1	/	X	**wrtee**
wrteei	0	1	1	1	1	1	///										E	///				0	0	1	0	1	0	0	0	1	1	/	X	**wrteei**
cmp	0	1	1	1	1	1	crfD			/	L	rA					rB					0	0	0	0	0	0	0	0	0	0	/	X	**cmp**
tw	0	1	1	1	1	1	TO					rA					rB					0	0	0	0	0	0	0	1	0	0	/	X	**tw**

Table 272. Instructions sorted by opcode (binary) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
subfc	0	1	1	1	1	1	rD					rA					rB					0	0	0	0	0	0	1	0	0	0	0	X	**subfc**
subfc.	0	1	1	1	1	1	rD					rA					rB					0	0	0	0	0	0	1	0	0	0	1	X	**subfc.**
addc	0	1	1	1	1	1	rD					rA					rB					0	0	0	0	0	0	1	0	1	0	0	X	**addc**
addc.	0	1	1	1	1	1	rD					rA					rB					0	0	0	0	0	0	1	0	1	0	1	X	**addc.**
mulhwu	0	1	1	1	1	1	rD					rA					rB					/	0	0	0	0	0	1	0	1	1	0	X	**mulhwu**
mulhwu.	0	1	1	1	1	1	rD					rA					rB					/	0	0	0	0	0	1	0	1	1	1	X	**mulhwu.**
mfcr	0	1	1	1	1	1	rD					///										0	0	0	0	0	1	0	0	1	1	/	X	**mfcr**
lwarx	0	1	1	1	1	1	rD					rA					rB					0	0	0	0	0	1	0	1	0	0	/	X	**lwarx**
lwzx	0	1	1	1	1	1	rD					rA					rB					0	0	0	0	0	1	0	1	1	1	/	X	**lwzx**
slw	0	1	1	1	1	1	rS					rA					rB					0	0	0	0	0	1	1	0	0	0	0	X	**slw**
slw.	0	1	1	1	1	1	rS					rA					rB					0	0	0	0	0	1	1	0	0	0	1	X	**slw.**
cntlzw	0	1	1	1	1	1	rS					rA					///					0	0	0	0	0	1	1	0	1	0	0	X	**cntlzw**
cntlzw.	0	1	1	1	1	1	rS					rA					///					0	0	0	0	0	1	1	0	1	0	1	X	**cntlzw.**
and	0	1	1	1	1	1	rS					rA					rB					0	0	0	0	0	1	1	1	0	0	0	X	**and**
and.	0	1	1	1	1	1	rS					rA					rB					0	0	0	0	0	1	1	1	0	0	1	X	**and.**
cmpl	0	1	1	1	1	1	/	L				rA					rB					///				0	0	0	0	1	0	0	0	0	/	X	**cmpl**
subf	0	1	1	1	1	1	rD					rA					rB					0	0	0	0	1	0	1	0	0	0	0	X	**subf**
subf.	0	1	1	1	1	1	rD					rA					rB					0	0	0	0	1	0	1	0	0	0	1	X	**subf.**
dcbst	0	1	1	1	1	1	///					rA					rB					0	0	0	0	1	1	0	1	1	0	/	X	**dcbst**
lwzux	0	1	1	1	1	1	rD					rA					rB					0	0	0	0	1	1	0	1	1	1	/	X	**lwzux**
andc	0	1	1	1	1	1	rS					rA					rB					0	0	0	0	1	1	1	1	0	0	0	X	**andc**
andc.	0	1	1	1	1	1	rS					rA					rB					0	0	0	0	1	1	1	1	0	0	1	X	**andc.**
mulhw	0	1	1	1	1	1	rD					rA					rB					/	0	0	1	0	0	1	0	1	1	0	X	**mulhw**
mulhw.	0	1	1	1	1	1	rD					rA					rB					/	0	0	1	0	0	1	0	1	1	1	X	**mulhw.**
mfmsr [1]	0	1	1	1	1	1	rD					///										0	0	0	1	0	1	0	0	1	1	/	X	**mfmsr**
dcbf	0	1	1	1	1	1	///					rA					rB					0	0	0	1	0	1	0	1	1	0	/	X	**dcbf**
lbzx	0	1	1	1	1	1	rD					rA					rB					0	0	0	1	0	1	0	1	1	1	/	X	**lbzx**
neg	0	1	1	1	1	1	rD					rA					///					0	0	0	1	1	0	1	0	0	0	0	X	**neg**
neg.	0	1	1	1	1	1	rD					rA					///					0	0	0	1	1	0	1	0	0	0	1	X	**neg.**
lbzux	0	1	1	1	1	1	rD					rA					rB					0	0	0	1	1	1	0	1	1	1	/	X	**lbzux**
nor	0	1	1	1	1	1	rS					rA					rB					0	0	0	1	1	1	1	1	0	0	0	X	**nor**
nor.	0	1	1	1	1	1	rS					rA					rB					0	0	0	1	1	1	1	1	0	0	1	X	**nor.**
subfe	0	1	1	1	1	1	rD					rA					rB					0	0	1	0	0	0	1	0	0	0	0	X	**subfe**
subfe.	0	1	1	1	1	1	rD					rA					rB					0	0	1	0	0	0	1	0	0	0	1	X	**subfe.**
adde	0	1	1	1	1	1	rD					rA					rB					0	0	1	0	0	0	1	0	1	0	0	X	**adde**
adde.	0	1	1	1	1	1	rD					rA					rB					0	0	1	0	0	0	1	0	1	0	1	X	**adde.**
mtcrf	0	1	1	1	1	1	rS					/	CRM								/	0	0	1	0	0	1	0	0	0	0	/	XFX	**mtcrf**
mtmsr [1]	0	1	1	1	1	1	rS					///										0	0	1	0	0	1	0	0	1	0	/	X	**mtmsr**
stwcx.	0	1	1	1	1	1	rS					rA					rB					0	0	1	0	0	1	0	1	1	0	1	X	**stwcx.**

Table 272. Instructions sorted by opcode (binary) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
stwx	0	1	1	1	1	1	rS					rA					rB					0	0	1	0	0	1	0	1	1	1	/	D	**stwx**
stwux	0	1	1	1	1	1	rS					rA					rB					0	0	1	0	1	1	0	1	1	1	/	D	**stwux**
subfze	0	1	1	1	1	1	rD					rA					///					0	0	1	1	0	0	1	0	0	0	0	X	**subfze**
subfze.	0	1	1	1	1	1	rD					rA					///					0	0	1	1	0	0	1	0	0	0	1	X	**subfze.**
addze	0	1	1	1	1	1	rD					rA					///					0	0	1	1	0	0	1	0	1	0	0	X	**addze**
addze.	0	1	1	1	1	1	rD					rA					///					0	0	1	1	0	0	1	0	1	0	1	X	**addze.**
stbx	0	1	1	1	1	1	rS					rA					rB					0	0	1	1	0	1	0	1	1	1	0	X	**stbx**
subfme	0	1	1	1	1	1	rD					rA					///					0	0	1	1	1	0	1	0	0	0	0	X	**subfme**
subfme.	0	1	1	1	1	1	rD					rA					///					0	0	1	1	1	0	1	0	0	0	1	X	**subfme.**
addme	0	1	1	1	1	1	rD					rA					///					0	0	1	1	1	0	1	0	1	0	0	X	**addme**
addme.	0	1	1	1	1	1	rD					rA					///					0	0	1	1	1	0	1	0	1	0	1	X	**addme.**
mullw	0	1	1	1	1	1	rD					rA					rB					0	0	1	1	1	0	1	0	1	1	0	X	**mullw**
mullw.	0	1	1	1	1	1	rD					rA					rB					0	0	1	1	1	0	1	0	1	1	1	X	**mullw.**
dcbtst	0	1	1	1	1	1	CT					rA					rB					0	0	1	1	1	1	0	1	1	0	/	X	**dcbtst**
stbux	0	1	1	1	1	1	rS					rA					rB					0	0	1	1	1	1	0	1	1	1	0	X	**stbux**
add	0	1	1	1	1	1	rD					rA					rB					0	1	0	0	0	0	1	0	1	0	0	X	**add**
add.	0	1	1	1	1	1	rD					rA					rB					0	1	0	0	0	0	1	0	1	0	1	X	**add.**
dcbt	0	1	1	1	1	1	CT					rA					rB					0	1	0	0	0	1	0	1	1	0	/	X	**dcbt**
lhzx	0	1	1	1	1	1	rD					rA					rB					0	1	0	0	0	1	0	1	1	1	/	X	**lhzx**
eqv	0	1	1	1	1	1	rD					rA					rB					0	1	0	0	0	1	1	1	0	0	0	X	**eqv**
eqv.	0	1	1	1	1	1	rD					rA					rB					0	1	0	0	0	1	1	1	0	0	1	X	**eqv.**
tlbie [1,3]	0	1	1	1	1	1	///					///					rB					0	1	0	0	1	1	0	0	1	0	0	X	**tlbie**
lhzux	0	1	1	1	1	1	rD					rA					rB					0	1	0	0	1	1	0	1	1	1	/	X	**lhzux**
xor	0	1	1	1	1	1	rS					rA					rB					0	1	0	0	1	1	1	1	0	0	0	X	**xor**
xor.	0	1	1	1	1	1	rS					rA					rB					0	1	0	0	1	1	1	1	0	0	1	X	**xor.**
mfspr [3]	0	1	1	1	1	1	rD					SPR[5–9]					SPR[0–4]					0	1	0	1	0	1	0	0	1	1	/	XFX	**mfspr**
lhax	0	1	1	1	1	1	rD					rA					rB					0	1	0	1	0	1	0	1	1	1	/	X	**lhax**
lhaux	0	1	1	1	1	1	rD					rA					rB					0	1	0	1	1	1	0	1	1	1	/	X	**lhaux**
sthx	0	1	1	1	1	1	rS					rA					rB					0	1	1	0	0	1	0	1	1	1	/	X	**sthx**
orc	0	1	1	1	1	1	rS					rA					rB					0	1	1	0	0	1	1	1	0	0	0	X	**orc**
orc.	0	1	1	1	1	1	rS					rA					rB					0	1	1	0	0	1	1	1	0	0	1	X	**orc.**
sthux	0	1	1	1	1	1	rS					rA					rB					0	1	1	0	1	1	0	1	1	1	/	X	**sthux**
or	0	1	1	1	1	1	rS					rA					rB					0	1	1	0	1	1	1	1	0	0	0	X	**or**
or.	0	1	1	1	1	1	rS					rA					rB					0	1	1	0	1	1	1	1	0	0	1	X	**or.**
divwu	0	1	1	1	1	1	rD					rA					rB					0	1	1	1	0	0	1	0	1	1	0	X	**divwu**
divwu.	0	1	1	1	1	1	rD					rA					rB					0	1	1	1	0	0	1	0	1	1	1	X	**divwu.**
mtspr [2]	0	1	1	1	1	1	rS					SPR[5–9]					SPR[0–4]					0	1	1	1	0	1	0	0	1	1	/	XFX	**mtspr**
dcbi [1]	0	1	1	1	1	1	///					rA					rB					0	1	1	1	0	1	0	1	1	0	/	X	**dcbi**
nand	0	1	1	1	1	1	rS					rA					rB					0	1	1	1	0	1	1	1	0	0	0	X	**nand**

Table 272. Instructions sorted by opcode (binary) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
nand.	0	1	1	1	1	1	rS					rA					rB					0	1	1	1	0	1	1	1	0	0	1	X	**nand.**
divw	0	1	1	1	1	1	rD					rA					rB					0	1	1	1	1	0	1	0	1	1	0	X	**divw**
divw.	0	1	1	1	1	1	rD					rA					rB					0	1	1	1	1	0	1	0	1	1	1	X	**divw.**
mcrxr	0	1	1	1	1	1	crfD					///										1	0	0	0	0	0	0	0	0	0	/	X	**mcrxr**
subfco	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	0	0	1	0	0	0	0	X	**subfco**
subfco.	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	0	0	1	0	0	0	1	X	**subfco.**
addco	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	0	0	1	0	1	0	0	X	**addco**
addco.	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	0	0	1	0	1	0	1	X	**addco.**
lwbrx	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	0	1	0	1	1	0	/	X	**lwbrx**
srw	0	1	1	1	1	1	rS					rA					rB					1	0	0	0	0	1	1	0	0	0	0	X	**srw**
srw.	0	1	1	1	1	1	rS					rA					rB					1	0	0	0	0	1	1	0	0	0	1	X	**srw.**
subfo	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	1	0	1	0	0	0	0	X	**subfo**
subfo.	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	1	0	1	0	0	0	1	X	**subfo.**
tlbsync [1,6]	0	1	1	1	1	1	///					///					///					1	0	0	0	1	1	0	1	1	0	/	X	**tlbsync**
nego	0	1	1	1	1	1	rD					rA					///					1	0	0	1	1	0	1	0	0	0	0	X	**nego**
nego.	0	1	1	1	1	1	rD					rA					///					1	0	0	1	1	0	1	0	0	0	1	X	**nego.**
subfeo	0	1	1	1	1	1	rD					rA					rB					1	0	1	0	0	0	1	0	0	0	0	X	**subfeo**
subfeo.	0	1	1	1	1	1	rD					rA					rB					1	0	1	0	0	0	1	0	0	0	1	X	**subfeo.**
addeo	0	1	1	1	1	1	rD					rA					rB					1	0	1	0	0	0	1	0	1	0	0	X	**addeo**
addeo.	0	1	1	1	1	1	rD					rA					rB					1	0	1	0	0	0	1	0	1	0	1	X	**addeo.**
stwbrx	0	1	1	1	1	1	rS					rA					rB					1	0	1	0	0	1	0	1	1	0	/	X	**stwbrx**
subfzeo	0	1	1	1	1	1	rD					rA					///					1	0	1	1	0	0	1	0	0	0	0	X	**subfzeo**
subfzeo.	0	1	1	1	1	1	rD					rA					///					1	0	1	1	0	0	1	0	0	0	1	X	**subfzeo.**
addzeo	0	1	1	1	1	1	rD					rA					///					1	0	1	1	0	0	1	0	1	0	0	X	**addzeo**
addzeo.	0	1	1	1	1	1	rD					rA					///					1	0	1	1	0	0	1	0	1	0	1	X	**addzeo.**
subfmeo	0	1	1	1	1	1	rD					rA					///					1	0	1	1	1	0	1	0	0	0	0	X	**subfmeo**
subfmeo.	0	1	1	1	1	1	rD					rA					///					1	0	1	1	1	0	1	0	0	0	1	X	**subfmeo.**
addmeo	0	1	1	1	1	1	rD					rA					///					1	0	1	1	1	0	1	0	1	0	0	X	**addmeo**
addmeo.	0	1	1	1	1	1	rD					rA					///					1	0	1	1	1	0	1	0	1	0	1	X	**addmeo.**
mullwo	0	1	1	1	1	1	rD					rA					rB					1	0	1	1	1	0	1	0	1	1	0	X	**mullwo**
mullwo.	0	1	1	1	1	1	rD					rA					rB					1	0	1	1	1	0	1	0	1	1	1	X	**mullwo.**
dcba [6]	0	1	1	1	1	1	///					rA					rB					1	0	1	1	1	1	0	1	1	0	/	X	**dcba**
addo	0	1	1	1	1	1	rD					rA					rB					1	1	0	0	0	0	1	0	1	0	0	X	**addo**
addo.	0	1	1	1	1	1	rD					rA					rB					1	1	0	0	0	0	1	0	1	0	1	X	**addo.**
lhbrx	0	1	1	1	1	1	rD					rA					rB					1	1	0	0	0	1	0	1	1	0	/	X	**lhbrx**
sraw	0	1	1	1	1	1	rS					rA					rB					1	1	0	0	0	1	1	0	0	0	0	X	**sraw**
sraw.	0	1	1	1	1	1	rS					rA					rB					1	1	0	0	0	1	1	0	0	0	1	X	**sraw.**

Table 272. Instructions sorted by opcode (binary) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
srawi	0	1	1	1	1	1	rS					rA					SH					1	1	0	0	1	1	1	0	0	0	0	X	**srawi**
srawi.	0	1	1	1	1	1	rS					rA					SH					1	1	0	0	1	1	1	0	0	0	1	X	**srawi.**
sthbrx	0	1	1	1	1	1	rS					rA					rB					1	1	1	0	0	1	0	1	1	0	/	X	**sthbrx**
extsh	0	1	1	1	1	1	rS					rA					///					1	1	1	0	0	1	1	0	1	0	0	X	**extsh**
extsh.	0	1	1	1	1	1	rS					rA					///					1	1	1	0	0	1	1	0	1	0	1	X	**extsh.**
extsb	0	1	1	1	1	1	rS					rA					///					1	1	1	0	1	1	1	0	1	0	0	X	**extsb**
extsb.	0	1	1	1	1	1	rS					rA					///					1	1	1	0	1	1	1	0	1	0	1	X	**extsb.**
divwuo	0	1	1	1	1	1	rD					rA					rB					1	1	1	1	0	0	1	0	1	1	0	X	**divwuo**
divwuo.	0	1	1	1	1	1	rD					rA					rB					1	1	1	1	0	0	1	0	1	1	1	X	**divwuo.**
icbi	0	1	1	1	1	1	///					rA					rB					1	1	1	1	0	1	0	1	1	0	/	X	**icbi**
divwo	0	1	1	1	1	1	rD					rA					rB					1	1	1	1	1	0	1	0	1	1	0	X	**divwo**
divwo.	0	1	1	1	1	1	rD					rA					rB					1	1	1	1	1	0	1	0	1	1	1	X	**divwo.**
dcbz	0	1	1	1	1	1	///					rA					rB					1	1	1	1	1	1	0	1	1	0	/	X	**dcbz**
lwz	1	0	0	0	0	0	rD					rA					D																D	**lwz**
lwzu	1	0	0	0	0	1	rD					rA					D																D	**lwzu**
lbz	1	0	0	0	1	0	rD					rA					D																D	**lbz**
lbzu	1	0	0	0	1	1	rD					rA					D																D	**lbzu**
stw	1	0	0	1	0	0	rS					rA					D																D	**stw**
stwu	1	0	0	1	0	1	rS					rA					D																D	**stwu**
stb	1	0	0	1	1	0	rS					rA					D																D	**stb**
stbu	1	0	0	1	1	1	rS					rA					D																D	**stbu**
lhz	1	0	1	0	0	0	rD					rA					D																D	**lhz**
lhzu	1	0	1	0	0	1	rD					rA					D																D	**lhzu**
lha	1	0	1	0	1	0	rD					rA					D																D	**lha**
lhau	1	0	1	0	1	1	rD					rA					D																D	**lhau**
sth	1	0	1	1	0	0	rS					rA					D																D	**sth**
sthu	1	0	1	1	0	1	rS					rA					D																D	**sthu**
lmw	1	0	1	1	1	0	rD					rA					D																D	**lmw**
stmw	1	0	1	1	1	1	rS					rA					D																D	**stmw**
fres [6]	1	1	1	0	1	1	frD					///					frB					///					1	1	0	0	0	0	A	**fres**
fres. [6]	1	1	1	0	1	1	frD					///					frB					///					1	1	0	0	0	1	A	**fres.**
fsel [6]	1	1	1	1	1	1	frD					frA					frB					frC					1	0	1	1	1	0	A	**fsel**
fsel. [6]	1	1	1	1	1	1	frD					frA					frB					frC					1	0	1	1	1	1	A	**fsel.**

1. Supervisor-level instruction
2. d = UIMM * 8

A.5 Instruction set legend

Table 273 provides general information on the instruction set (such as architectural level, privilege level, and form).

Table 273. PowerPC instruction set legend

	UISA	VEA	OEA	Supervisor level	Optional	Form	
addx	√					XO	addx
addcx	√					XO	addcx
addex	√					XO	addex
addi	√					D	addi
addic	√					D	addic
addic.	√					D	addic.
addis	√					D	addis
addmex	√					XO	addmex
addzex	√					XO	addzex
andx	√					X	andx
andcx	√					X	andcx
andi.	√					D	andi.
andis.	√					D	andis.
bx	√					I	bx
bcx	√					B	bcx
bcctrx	√					XL	bcctrx
bclrx	√					XL	bclrx
cmp	√					X	cmp
cmpi	√					D	cmpi
cmpl	√					X	cmpl
cmpli	√					D	cmpli
cntlzwx	√					X	cntlzwx
crand	√					XL	crand
crandc	√					XL	crandc
creqv	√					XL	creqv
crnand	√					XL	crnand
crnor	√					XL	crnor
cror	√					XL	cror
crorc	√					XL	crorc
crxor	√					XL	crxor
dcba		√			√	X	dcba
dcbf		√				X	dcbf
dcbi			√	√		X	dcbi

Table 273. PowerPC instruction set legend (continued)

	UISA	VEA	OEA	Supervisor level	Optional	Form	
dcbst		√				X	dcbst
dcbt		√				X	dcbt
dcbtst		√				X	dcbtst
dcbz		√				X	dcbz
divwx	√					XO	divwx
divwux	√					XO	divwux
eciwx		√			√	X	eciwx
ecowx		√			√	X	ecowx
eieio		√				X	eieio
eqvx	√					X	eqvx
extsbx	√					X	extsbx
extshx	√					X	extshx
fabsx	√					X	fabsx
faddx	√					A	faddx
faddsx	√					A	faddsx
fcmpo	√					X	fcmpo
fcmpu	√					X	fcmpu
fctiwx	√					X	fctiwx
fctiwzx	√					X	fctiwzx
fdivx	√					A	fdivx
fdivsx	√					A	fdivsx
fmaddx	√					A	fmaddx
fmaddsx	√					A	fmaddsx
fmrx	√					X	fmrx
fmsubx	√					A	fmsubx
fmsubsx	√					A	fmsubsx
fmulx	√					A	fmulx
fmulsx	√					A	fmulsx
fnabsx	√					X	fnabsx
fnegx	√					X	fnegx
fnmaddx	√					A	fnmaddx
fnmaddsx	√					A	fnmaddsx
fnmsubx	√					A	fnmsubx

Table 273. PowerPC instruction set legend (continued)

	UISA	VEA	OEA	Supervisor level	Optional	Form	
fnmsubsx	√					A	fnmsubsx
fresx	√				√	A	fresx
frspx	√					X	frspx
frsqrtex	√				√	A	frsqrtex
fselx	√				√	A	fselx
fsqrtx	√				√	A	fsqrtx
fsqrtsx	√				√	A	fsqrtsx
fsubx	√					A	fsubx
fsubsx	√					A	fsubsx
icbi		√				X	icbi
isync		√				XL	isync
lbz	√					D	lbz
lbzu	√					D	lbzu
lbzux	√					X	lbzux
lbzx	√					X	lbzx
lfd	√					D	lfd
lfdu	√					D	lfdu
lfdux	√					X	lfdux
lfdx	√					X	lfdx
lfs	√					D	lfs
lfsu	√					D	lfsu
lfsux	√					X	lfsux
lfsx	√					X	lfsx
lha	√					D	lha
lhau	√					D	lhau
lhaux	√					X	lhaux
lhax	√					X	lhax
lhbrx	√					X	lhbrx
lhz	√					D	lhz
lhzu	√					D	lhzu
lhzux	√					X	lhzux
lhzx	√					X	lhzx
lmw (1)	√					D	lmw (2)

Table 273. PowerPC instruction set legend (continued)

	UISA	VEA	OEA	Supervisor level	Optional	Form	
lswi [1]	√					X	lswi [1]
lswx [1]	√					X	lswx [1]
lwarx	√					X	lwarx
lwbrx	√					X	lwbrx
lwz	√					D	lwz
lwzu	√					D	lwzu
lwzux	√					X	lwzux
lwzx	√					X	lwzx
mcrf	√					XL	mcrf
mcrfs	√					X	mcrfs
mcrxr	√					X	mcrxr
mfcr	√					X	mfcr
mffs	√					X	mffs
mfmsr			√	√		X	mfmsr
mfspr [3]	√		√	√		XFX	mfspr [3]
mfsr			√	√		X	mfsr
mfsrin			√	√		X	mfsrin
mftb		√				XFX	mftb
mtcrf	√					XFX	mtcrf
mtfsb0x	√					X	mtfsb0x
mtfsb1x	√					X	mtfsb1x
mtfsfx	√					XFL	mtfsfx
mtfsfix	√					X	mtfsfix
mtmsr			√	√		X	mtmsr
mtspr [3]	√		√	√		XFX	mtspr [4]
mtsr			√	√		X	mtsr
mtsrin			√	√		X	mtsrin
mulhwx	√					XO	mulhwx
mulhwux	√					XO	mulhwux
mulli	√					D	mulli
nandx	√					X	nandx
negx	√					XO	negx
norx	√					X	norx

Table 273. PowerPC instruction set legend (continued)

	UISA	VEA	OEA	Supervisor level	Optional	Form	
orx	√					X	orx
orcx	√					X	orcx
ori	√					D	ori
oris	√					D	oris
rfi			√	√		XL	rfi
rlwimix	√					M	rlwimix
rlwinmx	√					M	rlwinmx
rlwnmx	√					M	rlwnmx
sc	√		√			SC	sc
slwx	√					X	slwx
srawx	√					X	srawx
srawix	√					X	srawix
srwx	√					X	srwx
stb	√					D	stb
stbu	√					D	stbu
stbux	√					X	stbux
stbx	√					X	stbx
stfd	√					D	stfd
stfdu	√					D	stfdu
stfdux	√					X	stfdux
stfdx	√					X	stfdx
stfiwx	√					X	stfiwx
stfs	√					D	stfs
stfsu	√					D	stfsu
stfsux	√					X	stfsux
stfsx	√					X	stfsx
sth	√					D	sth
sthbrx	√					X	sthbrx
sthu	√					D	sthu
sthux	√					X	sthux
sthx	√					X	sthx
stmw (1)	√					D	stmw (1)
stswi (1)	√					X	stswi (1)

Table 273. PowerPC instruction set legend (continued)

	UISA	VEA	OEA	Supervisor level	Optional	Form	
stswx (1)	√					X	stswx (1)
stw	√					D	stw
stwbrx	√					X	stwbrx
stwcx.	√					X	stwcx.
stwu	√					D	stwu
stwux	√					X	stwux
stwx	√					X	stwx
subfx	√					XO	subfx
subfcx	√					XO	subfcx
subfex	√					XO	subfex
subfic	√					D	subfic
subfmex	√					XO	subfmex
subfzex	√					XO	subfzex
sync	√					X	sync
tlbiax			√	√	√	X	tlbiax
tlbiex			√	√	√	X	tlbiex
tlbsync			√	√	√	X	tlbsync
tw	√					X	tw
twi	√					D	twi
xorx	√					X	xorx
xori	√					D	xori
xoris	√					D	xoris

1. Load/Store string or multiple.
2. Load/Store string or multiple.
3. Supervisor and user level instruction.
4. Supervisor and user level instruction.

Table 274. PowerPC instruction set legend

	UISA	VEA	OEA	Supervisor Level	Optional	Form	
add*x*	√					XO	**add***x*
addc*x*	√					XO	**addc***x*
adde*x*	√					XO	**adde***x*
addi	√					D	**addi**
addic	√					D	**addic**

Table 274. PowerPC instruction set legend (continued)

	UISA	VEA	OEA	Supervisor Level	Optional	Form	
addic.	√					D	**addic.**
addis	√					D	**addis**
addme*x*	√					XO	**addme***x*
addze*x*	√					XO	**addze***x*
and*x*	√					X	**and***x*
andc*x*	√					X	**andc***x*
andi.	√					D	**andi.**
andis.	√					D	**andis.**
b*x*	√					I	**b***x*
bc*x*	√					B	**bc***x*
bcctr*x*	√					XL	**bcctr***x*
bclr*x*	√					XL	**bclr***x*
cmp	√					X	**cmp**
cmpi	√					D	**cmpi**
cmpl	√					X	**cmpl**
cmpli	√					D	**cmpli**
cntlzw*x*	√					X	**cntlzw***x*
crand	√					XL	**crand**
crandc	√					XL	**crandc**
creqv	√					XL	**creqv**
crnand	√					XL	**crnand**
crnor	√					XL	**crnor**
cror	√					XL	**cror**
crorc	√					XL	**crorc**
crxor	√					XL	**crxor**
dcba		√			√	X	**dcba**
dcbf		√				X	**dcbf**
dcbi			√	√		X	**dcbi**
dcbst		√				X	**dcbst**
dcbt		√				X	**dcbt**
dcbtst		√				X	**dcbtst**
dcbz		√				X	**dcbz**
divw*x*	√					XO	**divw***x*
divwu*x*	√					XO	**divwu***x*
eciwx		√			√	X	**eciwx**

Table 274. PowerPC instruction set legend (continued)

	UISA	VEA	OEA	Supervisor Level	Optional	Form	
ecowx		√			√	X	**ecowx**
eieio		√				X	**eieio**
eqv*x*	√					X	**eqv***x*
extsb*x*	√					X	**extsb***x*
extsh*x*	√					X	**extsh***x*
fabs*x*	√					X	**fabs***x*
fadd*x*	√					A	**fadd***x*
fadds*x*	√					A	**fadds***x*
fcmpo	√					X	**fcmpo**
fcmpu	√					X	**fcmpu**
fctiw*x*	√					X	**fctiw***x*
fctiwz*x*	√					X	**fctiwz***x*
fdiv*x*	√					A	**fdiv***x*
fdivs*x*	√					A	**fdivs***x*
fmadd*x*	√					A	**fmadd***x*
fmadds*x*	√					A	**fmadds***x*
fmr*x*	√					X	**fmr***x*
fmsub*x*	√					A	**fmsub***x*
fmsubs*x*	√					A	**fmsubs***x*
fmul*x*	√					A	**fmul***x*
fmuls*x*	√					A	**fmuls***x*
fnabs*x*	√					X	**fnabs***x*
fneg*x*	√					X	**fneg***x*
fnmadd*x*	√					A	**fnmadd***x*
fnmadds *x*	√					A	**fnmadds***x*
fnmsub*x*	√					A	**fnmsub***x*
fnmsubs *x*	√					A	**fnmsubs***x*
fres*x*	√				√	A	**fres***x*
frsp*x*	√					X	**frsp***x*
frsqrte*x*	√				√	A	**frsqrte***x*
fsel*x*	√				√	A	**fsel***x*
fsqrt*x*	√				√	A	**fsqrt***x*
fsqrts*x*	√				√	A	**fsqrts***x*

Table 274. PowerPC instruction set legend (continued)

	UISA	VEA	OEA	Supervisor Level	Optional	Form	
fsub*x*	√					A	**fsub***x*
fsubs*x*	√					A	**fsubs***x*
icbi		√				X	**icbi**
isync		√				XL	**isync**
lbz	√					D	**lbz**
lbzu	√					D	**lbzu**
lbzux	√					X	**lbzux**
lbzx	√					X	**lbzx**
lfd	√					D	**lfd**
lfdu	√					D	**lfdu**
lfdux	√					X	**lfdux**
lfdx	√					X	**lfdx**
lfs	√					D	**lfs**
lfsu	√					D	**lfsu**
lfsux	√					X	**lfsux**
lfsx	√					X	**lfsx**
lha	√					D	**lha**
lhau	√					D	**lhau**
lhaux	√					X	**lhaux**
lhax	√					X	**lhax**
lhbrx	√					X	**lhbrx**
lhz	√					D	**lhz**
lhzu	√					D	**lhzu**
lhzux	√					X	**lhzux**
lhzx	√					X	**lhzx**
lmw [1]	√					D	**lmw** [1]
lswi [1]	√					X	**lswi** [1]
lswx[1]	√					X	**lswx**[1]
lwarx	√					X	**lwarx**
lwbrx	√					X	**lwbrx**
lwz	√					D	**lwz**
lwzu	√					D	**lwzu**
lwzux	√					X	**lwzux**
lwzx	√					X	**lwzx**
mcrf	√					XL	**mcrf**

Table 274. PowerPC instruction set legend (continued)

	UISA	VEA	OEA	Supervisor Level	Optional	Form	
mcrfs	√					X	**mcrfs**
mcrxr	√					X	**mcrx**r
mfcr	√					X	**mfcr**
mffs	√					X	**mffs**
mfmsr			√	√		X	**mfmsr**
mfspr[3]	√		√	√		XFX	**mfspr**[1]
mfsr			√	√		X	**mfsr**
mfsrin			√	√		X	**mfsrin**
mftb		√				XFX	**mftb**
mtcrf	√					XFX	**mtcrf**
mtfsb0*x*	√					X	**mtfsb0***x*
mtfsb1*x*	√					X	**mtfsb1***x*
mtfsf*x*	√					XFL	**mtfsf***x*
mtfsfi*x*	√					X	**mtfsfi***x*
mtmsr			√	√		X	**mtmsr**
mtspr[1]	√		√	√		XFX	**mtspr**[1]
mtsr			√	√		X	**mtsr**
mtsrin			√	√		X	**mtsrin**
mulhw*x*	√					XO	**mulhw***x*
mulhwu*x*	√					XO	**mulhwu***x*
mulli	√					D	**mulli**
mullw*x*	√					XO	**mullw***x*
nand*x*	√					X	**nand***x*
neg*x*	√					XO	**neg***x*
nor*x*	√					X	**nor***x*
or*x*	√					X	**or***x*
orc*x*	√					X	**orc***x*
ori	√					D	**ori**
oris	√					D	**oris**
rfi			√	√		XL	**rfi**
rlwimi*x*	√					M	**rlwimi***x*
rlwinm*x*	√					M	**rlwinm***x*
rlwnm*x*	√					M	**rlwnm***x*
sc	√		√			SC	**sc**
slw*x*	√					X	**slw***x*

Table 274. PowerPC instruction set legend (continued)

	UISA	VEA	OEA	Supervisor Level	Optional	Form	
sraw*x*	√					X	**sraw***x*
srawi*x*	√					X	**srawi***x*
srw*x*	√					X	**srw***x*
stb	√					D	**stb**
stbu	√					D	**stbu**
stbux	√					X	**stbux**
stbx	√					X	**stbx**
stfd	√					D	**stfd**
stfdu	√					D	**stfdu**
stfdux	√					X	**stfdux**
stfdx	√					X	**stfdx**
stfiwx	√					X	**stfiwx**
stfs	√					D	**stfs**
stfsu	√					D	**stfsu**
stfsux	√					X	**stfsux**
stfsx	√					X	**stfsx**
sth	√					D	**sth**
sthbrx	√					X	**sthbrx**
sthu	√					D	**sthu**
sthux	√					X	**sthux**
sthx	√					X	**sthx**
stmw [1]	√					D	**stmw** [1]
stswi [1]	√					X	**stswi** [1]
stswx [1]	√					X	**stswx** [1]
stw	√					D	**stw**
stwbrx	√					X	**stwbrx**
stwcx.	√					X	**stwcx.**
stwu	√					D	**stwu**
stwux	√					X	**stwux**
stwx	√					X	**stwx**
subf*x*	√					XO	**subf***x*
subfc*x*	√					XO	**subfc***x*
subfe*x*	√					XO	**subfe***x*
subfic	√					D	**subfic**
subfme*x*	√					XO	**subfme***x*

Table 274. PowerPC instruction set legend (continued)

	UISA	VEA	OEA	Supervisor Level	Optional	Form	
subfze*x*	√					XO	**subfze*x***
sync	√					X	**sync**
tlbia*x*			√	√	√	X	**tlbia*x***
tlbie*x*			√	√	√	X	**tlbie*x***
tlbsync			√	√	√	X	**tlbsync**
tw	√					X	**tw**
twi	√					D	**twi**
xor*x*	√					X	**xor*x***
xori	√					D	**xori**
xoris	√					D	**xoris**

Appendix B Simplified mnemonics for PowerPC instructions

This chapter describes simplified mnemonics, which are provided for easier coding of assembly language programs. Simplified mnemonics are defined for the most frequently used forms of branch conditional, compare, trap, rotate and shift, and certain other instructions defined by the PowerPC™ architecture and by implementations of and extensions to the PowerPC architecture.

Chapter B.11: Comprehensive list of simplified mnemonics on page 1133," provides an alphabetical listing of simplified mnemonics. Some assemblers may define additional simplified mnemonics not included here. The simplified mnemonics listed here should be supported by all compilers.

B.1 Overview

Simplified (or extended) mnemonics allow an assembly-language programmer to program using more intuitive mnemonics and symbols than the instructions and syntax defined by the instruction set architecture. For example, to code the conditional call “branch to an absolute target if CR4 specifies a greater than condition, setting the LR without simplified mnemonics, the programmer would write the branch conditional instruction, **bc 12,17,***target*. The simplified mnemonic, branch if greater than, **bgt cr4,** *target*, incorporates the conditions. Not only is it easier to remember the symbols than the numbers when programming, it is also easier to interpret simplified mnemonics when reading existing code.

Although the original PowerPC architecture documents include a set of simplified mnemonics, these are not a formal part of the architecture, but rather a recommendation for assemblers that support the instruction set.

Many simplified mnemonics have been added to those originally included in the architecture documentation. Some assemblers created their own, and others have been added to support extensions to the instruction set (for example, AltiVec instructions and Book E auxiliary processing units (APUs)). Simplified mnemonics have been added for new architecturally defined and new implementation-specific special-purpose registers (SPRs). These simplified mnemonics are described only in a very general way.

B.2 Subtract simplified mnemonics

This section describes simplified mnemonics for subtract instructions.

B.2.1 Subtract immediate

There is no subtract immediate instruction, however, its effect is achieved by negating the immediate operand of an Add Immediate instruction, **addi**. Simplified mnemonics include this negation, making the intent of the computation more clear. These are listed in *Table 275*.

Table 275. Subtract immediate simplified mnemonics

Simplified mnemonic	Standard mnemonic
subi **r**D,**r**A,value	**addi** **r**D,**r**A,–value
subis **r**D,**r**A,value	**addis** **r**D,**r**A,–value
subic **r**D,**r**A,value	**addic** **r**D,**r**A,–value
subic. **r**D,**r**A,value	**addic.** **r**D,**r**A,–value

B.2.2 Subtract

Subtract from instructions subtract the second operand (**r**A) from the third (**r**B). The simplified mnemonics in *Table 276* use the common order in which the third operand is subtracted from the second.

Table 276. Subtract simplified mnemonics

Simplified mnemonic	Standard mnemonic(1)
sub[**o**][.] **r**D,**r**A,**r**B	**subf**[**o**][.] **r**D,**r**B,**r**A
subc[**o**][.] **r**D,**r**A,**r**B	**subfc**[**o**][.] **r**D,**r**B,**r**A

1. **r**D,**r**B,**r**A is not the standard order for the operands. The order of **r**B and **r**A is reversed to show the equivalent behavior of the simplified mnemonic.

B.3 Rotate and shift simplified mnemonics

Rotate and shift instructions provide powerful, general ways to manipulate register contents, but can be difficult to understand. Simplified mnemonics are provided for the following operations:

- 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.
- Insert—Select a left- 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.
- 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 (logical shift).
- Clear—Clear the leftmost or rightmost *n* bits of a register.
- 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 non-negative) array index by the width of an element.

B.3.1 Operations on words

The simplified mnemonics in *Table 277* can be coded with a dot (**.**) suffix to cause the Rc bit to be set in the underlying instruction.

Table 277. Word rotate and shift simplified mnemonics

Operation	Simplified mnemonic	Equivalent to:
Extract and left justify word immediate	**extlwi** **r**A,**r**S,*n*,*b* (*n* > 0)	**rlwinm** **r**A,**r**S,*b*,**0**,*n* – 1
Extract and right justify word immediate	**extrwi** **r**A,**r**S,*n*,*b* (*n* > 0)	**rlwinm** **r**A,**r**S,*b* + *n*, 32 – *n*,**31**
Insert from left word immediate	**inslwi** **r**A,**r**S,*n*,*b* (*n* > 0)	**rlwimi** **r**A,**r**S,32 – *b*,*b*,(*b* + *n*) – 1
Insert from right word immediate	**insrwi** **r**A,**r**S,*n*,*b* (*n* > 0)	**rlwimi** **r**A,**r**S,32 – (*b* + *n*),*b*,(*b* + *n*) – 1
Rotate left word immediate	**rotlwi** **r**A,**r**S,*n*	**rlwinm** **r**A,**r**S,*n*,**0**,**31**
Rotate right word immediate	**rotrwi** **r**A,**r**S,*n*	**rlwinm** **r**A,**r**S,32 – *n*,**0**,**31**
Rotate word left	**rotlw** **r**A,**r**S,**r**B	**rlwnm** **r**A,**r**S,**r**B,**0**,**31**
Shift left word immediate	**slwi** **r**A,**r**S,*n* (*n* < 32)	**rlwinm** **r**A,**r**S,*n*,**0**,31 – *n*
Shift right word immediate	**srwi** **r**A,**r**S,*n* (*n* < 32)	**rlwinm** **r**A,**r**S,32 – *n*,*n*,**31**
Clear left word immediate	**clrlwi** **r**A,**r**S,*n* (*n* < 32)	**rlwinm** **r**A,**r**S,**0**,*n*,**31**
Clear right word immediate	**clrrwi** **r**A,**r**S,*n* (*n* < 32)	**rlwinm** **r**A,**r**S,**0**,**0**,31 – *n*
Clear left and shift left word immediate	**clrlslwi** **r**A,**r**S,*b*,*n* (*n* ≤ *b* ≤ 31)	**rlwinm** **r**A,**r**S,*n*,*b* – *n*,31 – *n*

Examples using word mnemonics follow:

1. Extract the sign bit (bit 0) of **r**S and place the result right-justified into **r**A.
   extrwi **r**A,**r**S,**1**,**0** equivalent to **rlwinm** **r**A,**r**S,**1**,**31**,**31**
2. Insert the bit extracted in (1) into the sign bit (bit 0) of **r**B.
   insrwi **r**B,**r**A,**1**,**0** equivalent to **rlwimi** **r**B,**r**A,**31**,**0**,**0**
3. Shift the contents of **r**A left 8 bits.
   slwi **r**A,**r**A,**8** equivalent to **rlwinm** **r**A,**r**A,**8**,**0**,**23**
4. Clear the high-order 16 bits of **r**S and place the result into **r**A.
   clrlwi **r**A,**r**S,**16** equivalent to **rlwinm** **r**A,**r**S,**0**,**16**,**31**

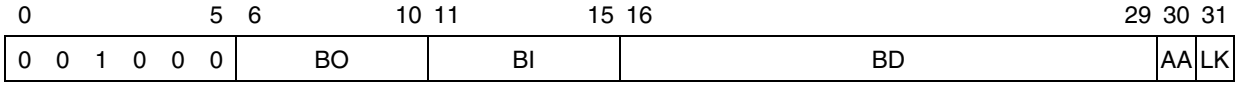
B.4 Branch instruction simplified mnemonics

Branch conditional instructions can be coded with the operations, a condition to be tested, and a prediction, as part of the mnemonic rather than as numeric BO and BI operands. *Table 278* shows the four general types of branch instructions. Simplified mnemonics are defined only for branch instructions that include BO and BI operands; there is no need to simplify unconditional branch mnemonics.

Table 278. Branch instructions

Instruction name	Mnemonic	Syntax
Branch	b (ba bl bla)	target_addr
Branch Conditional	bc (bca bcl bcla)	BO,BI,target_addr
Branch Conditional to Link Register	bclr (bclr**l**)	BO,BI
Branch Conditional to Count Register	bcctr (bcctr**l**)	BO,BI

The BO and BI operands correspond to two fields in the instruction opcode, as figure below shows for Branch Conditional (**bc**, **bca**, **bcl**, and **bcla**) instructions.



The BO operand specifies branch operations that involve decrementing CTR. It is also used to determine whether testing a CR bit causes a branch to occur if the condition is true or false.

The BI operand identifies a CR bit to test (whether a comparison is less than or greater than, for example). The simplified mnemonics avoid the need to memorize the numerical values for BO and BI.

For example, **bc 16,0,***target* is a conditional branch that, as a BO value of 16 (0b1_0000) indicates, decrements CTR, then branches if the decremented CTR is not zero. The operation specified by BO is abbreviated as **d** (for decrement) and **nz** (for not zero), which replace the **c** in the original mnemonic; so the simplified mnemonic for **bc** becomes **bdnz**. The branch does not depend on a condition in the CR, so BI can be eliminated, reducing the expression to **bdnz** *target*.

In addition to CTR operations, the BO operand provides an optional prediction bit and a true or false indicator can be added. For example, if the previous instruction should branch only on an equal condition in CR0, the instruction becomes **bc 8,2,***target*. To incorporate a true condition, the BO value becomes 8 (as shown in *Table 280*); the CR0 equal field is indicated by a BI value of 2 (as shown in *Table 281*). Incorporating the branch-if-true condition adds a '**t**' to the simplified mnemonic, **bdnzt.** The equal condition, that is specified by a BI value of 2 (indicating the EQ bit in CR0) is replaced by the **eq** symbol. Using the simplified mnemonic and the **eq** operand, the expression becomes **bdnzt eq,***target*.

This example tests CR0[EQ]; however, to test the equal condition in CR5 (CR bit 22), the expression becomes **bc 8,22,***target*. The BI operand of 22 indicates CR[22] (CR5[2], or BI field 0b10110), as shown in *Table 281*. This can be expressed as the simplified mnemonic. **bdnzt 4 * cr5 + eq,***target*.

The notation, **4 * cr5 + eq** may at first seem awkward, but it eliminates computing the value of the CR bit. It can be seen that (4 * 5) + 2 = 22. Note that although 32-bit registers in Book E processors are numbered 32–63, only values 0–31 are valid (or possible) for BI operands. As shown in *Table 282*, a Book E–compliant processor automatically translates the bit values; specifying a BI value of 22 selects bit 55 on a Book E processor, or CR5[2] = CR5[EQ].

B.4.1 Key facts about simplified branch mnemonics

The following key points are helpful in understanding how to use simplified branch mnemonics:

- All simplified branch mnemonics eliminate the BO operand, so if any operand is present in a branch simplified mnemonic, it is the BI operand (or a reduced form of it).
- If the CR is not involved in the branch, the BI operand can be deleted.
- If the CR is involved in the branch, the BI operand can be treated in the following ways:
  - It can be specified as a numeric value, just as it is in the architecturally defined instruction, or it can be indicated with an easier to remember formula, **4** * **cr***n* **+** [test bit symbol], where *n* indicates the CR field number.
  - The condition of the test bit (eq, lt, gt, and so) can be incorporated into the mnemonic, leaving the need for an operand that defines only the CR field.

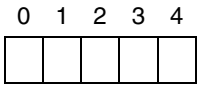
    - If the test bit is in CR0, no operand is needed.

    - If the test bit is in CR1–CR7, the BI operand can be replaced with a **cr**S operand (that is, **cr1**, **cr2**, **cr3**, and so forth).

B.4.2 Eliminating the BO operand

The 5-bit BO field, shown below, encodes the following operations in conditional branch instructions:

- Decrement count register (CTR)
  - And test if result is equal to zero
  - And test if result is not equal to zero
- Test condition register (CR)
  - Test condition true
  - Test condition false
- Branch prediction (taken, fall through). If the prediction bit, *y*, is needed, it is signified by appending a plus or minus sign as described in *Chapter B.4.3: Incorporating the BO branch prediction on page 1116*."



BO bits can be interpreted individually as described in *Table 279*.

Table 279. BO bit encodings

BO Bit	Description
0	If set, ignore the CR bit comparison.
1	If set, the CR bit comparison is against true, if not set the CR bit comparison is against false
2	If set, the CTR is not decremented.
3	If BO[2] is set, this bit determines whether the CTR comparison is for equal to zero or not equal to zero.
4	The *y* bit. If set, reverses the static prediction. Use of this bit is optional and independent from the interpretation of other BO bits. Because simplified branch mnemonics eliminate the BO operand, this bit is programmed by adding a plus or minus sign to the simplified mnemonic, as described in *Chapter B.4.3*."

Thus, a BO encoding of 10100 (decimal 20) means ignore the CR bit comparison and do not decrement the CTR—in other words, branch unconditionally. Encodings for the BO operand are shown in *Table 280*. A *z* bit indicates that the bit is ignored. However, these bits should be cleared, as they may be assigned a meaning in a future version of the architecture.

As shown in *Table 280*, the '**c**' in the standard mnemonic is replaced with the operations otherwise specified in the BO field, (**d** for decrement, **z** for zero, **nz** for non-zero, **t** for true, and **f** for false).

Table 280. BO operand encodings

BO field	Value(1) (decimal)	Description	Symbol
0000*y*	0	Decrement the CTR, then branch if the decremented CTR ≠ 0; condition is FALSE.	**dnzf**
0001*y*	2	Decrement the CTR, then branch if the decremented CTR = 0; condition is FALSE.	**dzf**
001*zy*	4	Branch if the condition is FALSE.(2) Note that 'false' and 'four' both start with 'f'.	**f**
0100*y*	8	Decrement the CTR, then branch if the decremented CTR ≠ 0; condition is TRUE.	**dnzt**
0101*y*	10	Decrement the CTR, then branch if the decremented CTR = 0; condition is TRUE.	**dzt**
011*z*(3)*y*	12	Branch if the condition is TRUE. 2 Note that 'true' and 'twelve' both start with 't'.	**t**
1*z*00*y*(4)	16	Decrement the CTR, then branch if the decremented CTR ≠ 0.	**dnz**(5)
1*z*01*y* 4	18	Decrement the CTR, then branch if the decremented CTR = 0.	**dz** 5
1*z*1*zz* 4	20	Branch always.	—

1. Assumes y = z = 0. *Chapter B.4.3: Incorporating the BO branch prediction*," describes how to use simplified mnemonics to program the *y* bit for static prediction.
2. Instructions for which B0 is 12 (branch if condition true) or 4 (branch if condition false) do not depend on the CTR value and can be alternately coded by incorporating the condition specified by the BI field, as described in *Chapter B.4.6*."
3. A *z* bit indicates a bit that is ignored. However, these bits should be cleared, as they may be assigned a meaning in a future version of the architecture.

4. Simplified mnemonics for branch instructions that do not test CR bits (BO = 16, 18, and 20) should specify only a target. Otherwise a programming error may occur.
5. Notice that these instructions do not use the branch if condition true or false operations. For that reason, simplified mnemonics for these should not specify a BI operand.

B.4.3 Incorporating the BO branch prediction

As shown in *Table 280*, the low-order bit (*y* bit) of the BO field provides a hint about whether the branch is likely to be taken (static branch prediction). Assemblers should clear this bit unless otherwise directed. This default action indicates the following:

- A branch conditional with a negative displacement field is predicted to be taken.
- A branch conditional with a non-negative displacement field is predicted not to be taken (fall through).
- A branch conditional to an address in the LR or CTR is predicted not to be taken (fall through).

If the likely outcome (branch or fall through) of a given branch conditional instruction is known, a suffix can be added to the mnemonic that tells the assembler how to set the *y* bit. That is, '+' indicates that the branch is to be taken and '–' indicates that the branch is not to be taken. This suffix can be added to any branch conditional mnemonic, standard or simplified.

For relative and absolute branches (**bc**[**l**][**a**]), the setting of the *y* bit depends on whether the displacement field is negative or non-negative. For negative displacement fields, coding the suffix '+' causes the bit to be cleared, and coding the suffix '–' causes the bit to be set. For non-negative displacement fields, coding the suffix '+' causes the bit to be set, and coding the suffix '–' causes the bit to be cleared.

For branches to an address in the LR or CTR (**bclr**[**l**] or **bcctr**[**l**]), coding the suffix '+' causes the *y* bit to be set, and coding the suffix '–' causes the bit to be cleared.

Examples of branch prediction follow:

1. Branch if CR0 reflects less than condition, specifying that the branch should be predicted as taken.

   blt+ *target*

2. Same as (1), but target address is in the LR and the branch should be predicted as not taken.

   bltlr–

B.4.4 The BI operand—CR bit and field representations

With standard branch mnemonics, the BI operand is used when it is necessary to test a CR bit, as shown in the example in *Chapter B.4: Branch instruction simplified mnemonics*."

With simplified mnemonics, the BI operand is handled differently depending on whether the simplified mnemonic incorporates a CR condition to test, as follows:

- Some branch simplified mnemonics incorporate only the BO operand. These simplified mnemonics can use the architecturally defined BI operand to specify the CR bit, as follows:
  - The BI operand can be presented exactly as it is with standard mnemonics—as a decimal number, 0–31.
  - Symbols can be used to replace the decimal operand, as shown in the example in *Chapter B.4: Branch instruction simplified mnemonics*," where **bdnzt 4 * cr5 + eq,***target* could be used instead of **bdnzt 22,***target*. This is described in *Specifying a CR bit on page 1118*."

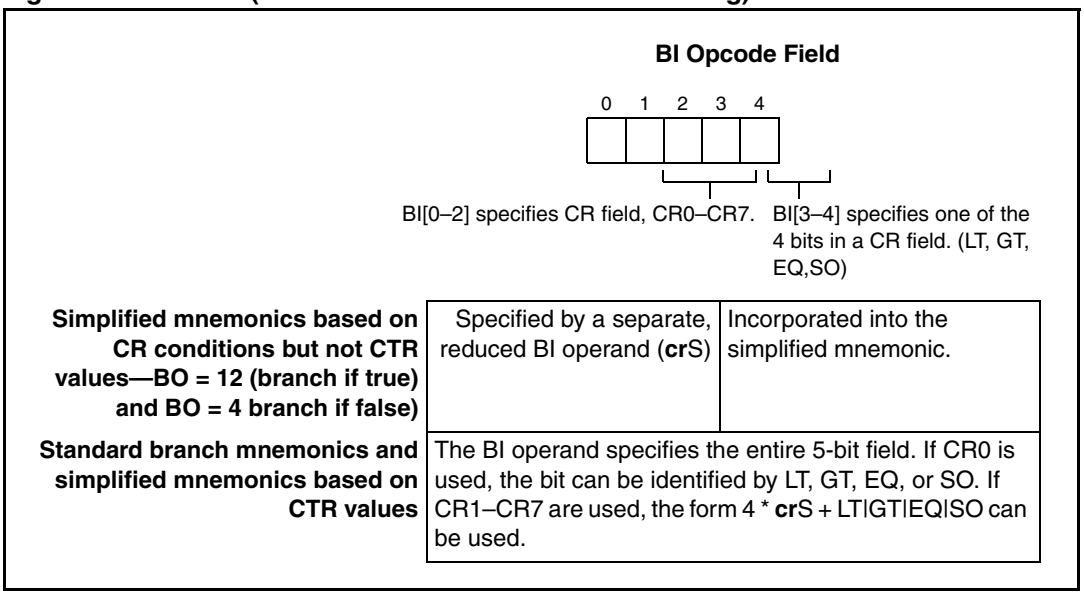
    The simplified mnemonics in *Chapter B.4.5: Simplified mnemonics that incorporate the BO operand*," use one of these two methods to specify a CR bit.
  - Additional simplified mnemonics are specified that incorporate CR conditions that would otherwise be specified by the BI operand, so the BI operand is replaced by the **cr**S operand to specify the CR field, CR0–CR7. See *BI operand instruction encoding on page 1117*."
    These mnemonics are described in *Chapter B.4.6: Simplified mnemonics that incorporate CR conditions (eliminates BO and replaces BI with crS)*."

BI operand instruction encoding

The entire 5-bit BI field, shown in *Figure 180*, represents the bit number for the CR bit to be tested. For standard branch mnemonics and for branch simplified mnemonics that do not incorporate a CR condition, the BI operand provides all 5 bits.

For simplified branch mnemonics described in *Chapter B.4.6*," the BI operand is replaced by a **cr**S operand. To understand this, it is useful to view the BI operand as comprised of two parts. As *Figure 180* shows, BI[0–2] indicates the CR field and BI[3–4] represents the condition to test.

Figure 180. BI field (Bits 11–14 of the instruction encoding)



Integer record-form instructions update CR0 as described in *Table 281*.

Specifying a CR bit

Note that the AIM version the PowerPC architecture numbers CR bits 0–31 and Book E numbers them 32–63. However, no adjustment is necessary to the code; in Book E devices, 32 is automatically added to the BI value, as shown in *Table 281* and *Table 282*.

Table 281. CR0 and CR1 fields as updated by integer instructions

CR*n* bit	CR bits		BI		Description
	AIM	Book E	0–2	3–4	
CR0[0]	0	32	000	00	Negative (LT)—Set when the result is negative.
CR0[1]	1	33	000	01	Positive (GT)—Set when the result is positive (and not zero).
CR0[2]	2	34	000	10	Zero (EQ)—Set when the result is zero.
CR0[3]	3	35	000	11	Summary overflow (SO). Copy of XER[SO] at the instruction's completion.

Some simplified mnemonics incorporate only the BO field (as described *Chapter B.4.2: Eliminating the BO operand*"). If one of these simplified mnemonics is used and the CR must be accessed, the BI operand can be specified either as a numeric value or by using the symbols in *Table 282*.

Compare word instructions (described in *Chapter B.5: Compare word simplified mnemonics*"), move to CR instructions, and others can also modify CR fields, so CR0 and CR1 may hold values that do not adhere to the meanings described in *Table 281*. CR logical instructions, described in *Chapter B.6: Condition register logical simplified mnemonics*," can update individual CR bits.

Table 282. BI operand settings for CR fields for branch comparisons

CR*n* bit	Bit expression	CR Bits		BI		Description
		AIM (BI operand)	Book E	0–2	3–4	
CR*n*[0]	**4 * cr0 + lt** (or **lt**)	0	32	000	00	Less than (LT). For integer compare instructions: **r**A < SIMM or **r**B (signed comparison) or **r**A < UIMM or **r**B (unsigned comparison).
	4 * cr1 + lt	4	36	001		
	4 * cr2 + lt	8	40	010		
	4 * cr3+ lt	12	44	011		
	4 * cr4 + lt	16	48	100		
	4 * cr5 + lt	20	52	101		
	4 * cr6 + lt	24	56	110		
	4 * cr7 + lt	28	60	111		
CR*n*[1]	**4 * cr0 + gt** (or **gt**)	1	33	000	01	Greater than (GT). For integer compare instructions: rA > SIMM or **r**B (signed comparison) or **r**A > UIMM or **r**B (unsigned comparison).
	4 * cr1 + gt	5	37	001		
	4 * cr2 + gt	9	41	010		
	4 * cr3+ gt	13	45	011		
	4 * cr4 + gt	17	49	100		
	4 * cr5 + gt	21	53	101		
	4 * cr6 + gt	25	57	110		
	4 * cr7 + gt	29	61	111		
CR*n*[2]	**4 * cr0 + eq** (or **eq**)	2	34	000	10	Equal (EQ). For integer compare instructions: **r**A = SIMM, UIMM, or **r**B.
	4 * cr1 + eq	6	38	001		
	4 * cr2 + eq	10	42	010		
	4 * cr3+ eq	14	46	011		
	4 * cr4 + eq	18	50	100		
	4 * cr5 + eq	22	54	101		
	4 * cr6 + eq	26	58	110		
	4 * cr7 + eq	30	62	111		
CR*n*[3]	**4 * cr0 + so** (or **so**)	3	35	000	11	Summary overflow (SO). For integer compare instructions, this is a copy of XER[SO] at instruction completion.
	4 * cr1 + so	7	39	001		
	4* cr2 + so	11	43	010		
	4* cr3 + so	15	47	011		
	4* cr4 + so	19	51	100		
	4* cr5 + so	23	55	101		
	4* cr6 + so	27	59	110		
	4* cr7 + so	31	63	111		

To provide simplified mnemonics for every possible combination of BO and BI (that is, including bits that identified the CR field) would require 2^{10} = 1024 mnemonics, most of that would be only marginally useful. The abbreviated set in *Chapter B.4.5: Simplified mnemonics that incorporate the BO operand*," covers useful cases. Unusual cases can be coded using a standard branch conditional syntax.

The crS operand

The **cr**S symbols are shown in *Table 283*. Note that either the symbol or the operand value can be used in the syntax used with the simplified mnemonic.

Table 283. CR field identification symbols

Symbol	BI[0–2]	CR bits
cr0 (default, can be eliminated from syntax)	000	32–35
cr1	001	36–39
cr2	010	40–43
cr3	011	44–47
cr4	100	48–51
cr5	101	52–55
cr6	110	56–59
cr7	111	60–63

To identify a CR bit, an expression in which a CR field symbol is multiplied by 4 and then added to a bit-number-within-CR-field symbol can be used, (for example, **cr0 * 4 + eq**).

B.4.5 Simplified mnemonics that incorporate the BO operand

The mnemonics in *Table 284* allow common BO operand encodings to be specified as part of the mnemonic, along with the absolute address (AA) and set link register bits (LK). There are no simplified mnemonics for relative and absolute unconditional branches. For these, the basic mnemonics **b**, **ba**, **bl**, and **bla** are used.

Table 284. Branch simplified mnemonics

Branch semantics	LR update not enabled				LR update enabled			
	bc	**bca**	**bclr**	**bcctr**	**bcl**	**bcla**	**bclrl**	**bcctrl**
Branch unconditionally (1)	—	—	**blr**	**bctr**	—	—	**blrl**	**bctrl**
Branch if condition true	**bt**	**bta**	**btlr**	**btctr**	**btl**	**btla**	**btlrl**	**btctrl**
Branch if condition false	**bf**	**bfa**	**bflr**	**bfctr**	**bfl**	**bfla**	**bflrl**	**bfctrl**
Decrement CTR, branch if CTR ≠ 0 [1]	**bdnz**	**bdnza**	**bdnzlr**	—	**bdnzl**	**bdnzla**	**bdnzlrl**	—
Decrement CTR, branch if CTR ≠ 0 and condition true	**bdnzt**	**bdnzta**	**bdnztlr**	—	**bdnztl**	**bdnztla**	**bdnztlrl**	—
Decrement CTR, branch if CTR ≠ 0 and condition false	**bdnzf**	**bdnzfa**	**bdnzflr**	—	**bdnzfl**	**bdnzfla**	**bdnzflrl**	—
Decrement CTR, branch if CTR = 0 [1]	**bdz**	**bdza**	**bdzlr**	—	**bdzl**	**bdzla**	**bdzlrl**	—
Decrement CTR, branch if CTR = 0 and condition true	**bdzt**	**bdzta**	**bdztlr**	—	**bdztl**	**bdztla**	**bdztlrl**	—
Decrement CTR, branch if CTR = 0 and condition false	**bdzf**	**bdzfa**	**bdzflr**	—	**bdzfl**	**bdzfla**	**bdzflrl**	—

1. Simplified mnemonics for branch instructions that do not test CR bits should specify only a target. Otherwise a programming error may occur.

Table 284 shows the syntax for basic simplified branch mnemonics

Table 285. Branch instructions

Instruction	Standard mnemonic	Syntax	Simplified mnemonic	Syntax
Branch	b (ba bl bla)	target_addr	N/A, syntax does not include BO	
Branch Conditional	bc (bca bcl bcla)	BO,BI,target_addr	**b***x*[1](**b***x***a b***x***l b***x***la**)	BI[2]target_addr
Branch Conditional to Link Register	bclr (bclr**l**)	BO,BI	b*x*lr (b*x*lr**l**)	BI
Branch Conditional to Count Register	bcctr (bcctr**l**)	BO,BI	b*x*ctr (b*x*ctr**l**)	BI

1. *x* stands for one of the symbols in *Table 280*, where applicable.
2. BI can be a numeric value or an expression as shown in *Table 283*.

The simplified mnemonics in *Table 284* that test a condition require a corresponding CR bit as the first operand (as examples 2–5 below show). The symbols in *Table 283* can be substituted for numeric values.

Examples that eliminate the BO operand

The simplified mnemonics in *Table 284* are used in the following examples:

1. Decrement CTR and branch if it is still nonzero (closure of a loop controlled by a count loaded into CTR) (note that no CR bits are tested).
   bdnz *target* equivalent to**bc 16,0,***target*
   Because this instruction does not test a CR bit, the simplified mnemonic should specify only a target operand. Specifying a CR (for example, **bdnz** 0,*target* or **bdnz cr0,***target*) may be considered a programming error. Subsequent examples test conditions).
2. Same as (1) but branch only if CTR is nonzero and equal condition in CR0.
   bdnzt eq,*target* equivalent to**bc 8,2,***target*
   Other equivalents include **bdnzt 2,***target* or the unlikely **bdnzt 4*cr0+eq,***target*
3. Same as (2), but equal condition is in CR5.
   bdnzt 4 * cr5 + eq,*target* equivalent to**bc 8,22,***target*
   bdnzt 22,*target* would also work
4. Branch if bit 59 of CR is false.
   bf 27,*target* equivalent to**bc 4,27,***target*
   bf 4*cr6+so,*target* would also work
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*

Table 286 lists simplified mnemonics and syntax for **bc** and **bca** without LR updating.

Table 286. Simplified mnemonics for bc and bca without LR update

Branch semantics	bc	Simplified mnemonic	bca	Simplified mnemonic
Branch unconditionally	—	—	—	—
Branch if condition true[1]	**bc 12,BI,**target	**bt BI,**target	**bca 12,BI,**target	**bta BI,**target
Branch if condition false [1]	**bc 4,BI,**target	**bf BI,**target	**bca 4,BI,**target	**bfa BI,**target
Decrement CTR, branch if CTR ≠ 0	**bc 16,0,**target	**bdnz** target[2]	**bca 16,0,**target	**bdnza** target [2]

Table 286. Simplified mnemonics for bc and bca without LR update (continued)

Branch semantics	bc	Simplified mnemonic	bca	Simplified mnemonic
Decrement CTR, branch if CTR ≠ 0 and condition true	**bc 8,BI,**target	**bdnzt BI,**target	**bca 8,BI,**target	**bdnzta BI,**target
Decrement CTR, branch if CTR ≠ 0 and condition false	**bc 0,BI,**target	**bdnzf BI,**target	**bca 0,BI,**target	**bdnzfa BI,**target
Decrement CTR, branch if CTR = 0	**bc 18,0,**target	**bdz** target [2]	**bca 18,0,**target	**bdza** target [2]
Decrement CTR, branch if CTR = 0 and condition true	**bc 10,BI,**target	**bdzt BI,**target	**bca 10,BI,**target	**bdzta BI,**target
Decrement CTR, branch if CTR = 0 and condition false	**bc 2,BI,**target	**bdzf BI,**target	**bca 2,BI,**target	**bdzfa BI,**target

1. Instructions for which B0 is either 12 (branch if condition true) or 4 (branch if condition false) do not depend on the CTR value and can be alternately coded by incorporating the condition specified by the BI field, as described in *Chapter B.4.6: Simplified mnemonics that incorporate CR conditions (eliminates BO and replaces BI with crS)*."
2. Simplified mnemonics for branch instructions that do not test CR bits should specify only a target. Otherwise a programming error may occur.

Table 287 lists simplified mnemonics and syntax for **bclr** and **bcctr** without LR updating.

Table 287. Simplified mnemonics for bclr and bcctr without LR update

Branch Semantics	bclr	Simplified mnemonic	bcctr	Simplified mnemonic
Branch unconditionally	**bclr 20,0**	**blr** (1)	**bcctr 20,0**	**bctr** [1]
Branch if condition true (2)	**bclr 12,BI**	**btlr BI**	**bcctr 12,BI**	**btctr BI**
Branch if condition false [2]	**bclr 4,BI**	**bflr BI**	**bcctr 4,BI**	**bfctr BI**
Decrement CTR, branch if CTR ≠ 0	**bclr 16,BI**	**bdnzlr BI**	—	—
Decrement CTR, branch if CTR ≠ 0 and condition true	**bclr 8,BI**	**bdnztlr BI**	—	—
Decrement CTR, branch if CTR ≠ 0 and condition false	**bclr 0,BI**	**bdnzflr BI**	—	—
Decrement CTR, branch if CTR = 0	**bclr 18,0**	**bdzlr** [1]	—	—
Decrement CTR, branch if CTR = 0 and condition true	**bclr 8,BI**	bdnztlr **BI**	—	—
Decrement CTR, branch if CTR = 0 and condition false	**bclr 2,BI**	**bdzflr BI**	—	—

1. Simplified mnemonics for branch instructions that do not test a CR bit should not specify one; a programming error may occur.
2. Instructions for which B0 is 12 (branch if condition true) or 4 (branch if condition false) do not depend on a CTR value and can be alternately coded by incorporating the condition specified by the BI field. See *Chapter B.4.6: Simplified mnemonics that incorporate CR conditions (eliminates BO and replaces BI with crS)*."

Table 288 provides simplified mnemonics and syntax for **bcl** and **bcla**.

Table 288. Simplified mnemonics for bcl and bcla with LR update

Branch semantics	bcl	Simplified mnemonic	bcla	Simplified mnemonic
Branch unconditionally	—	—	—	—
Branch if condition true (1)	**bcl 12,BI,**target	**btl BI,**target	**bcla 12,BI,**target	**btla BI,**target

Table 288. Simplified mnemonics for bcl and bcla with LR update (continued)

Branch semantics	bcl	Simplified mnemonic	bcla	Simplified mnemonic
Branch if condition false [1]	**bcl 4,BI,**target	**bfl BI,**target	**bcla 4,BI,**target	**bfla BI,**target
Decrement CTR, branch if CTR ≠ 0	**bcl 16,0,**target	**bdnzl** target [(2)]	**bcla 16,0,**target	**bdnzla** target [2]
Decrement CTR, branch if CTR ≠ 0 and condition true	**bcl 8,0,**target	**bdnztl BI,**target	**bcla 8,BI,**target	**bdnztla BI,**target
Decrement CTR, branch if CTR ≠ 0 and condition false	**bcl 0,BI,**target	**bdnzfl BI,**target	**bcla 0,BI,**target	**bdnzfla BI,**target
Decrement CTR, branch if CTR = 0	**bcl 18,BI,**target	**bdzl** target [2]	**bcla 18,BI,**target	**bdzla** target [2]
Decrement CTR, branch if CTR = 0 and condition true	**bcl 10,BI,**target	**bdztl BI,**target	**bcla 10,BI,**target	**bdztla BI,**target
Decrement CTR, branch if CTR = 0 and condition false	**bcl 2,BI,**target	**bdzfl BI,**target	**bcla 2,BI,**target	**bdzfla BI,**target

1. Instructions for which B0 is either 12 (branch if condition true) or 4 (branch if condition false) do not depend on the CTR value and can be alternately coded by incorporating the condition specified by the BI field. See *Chapter B.4.6: Simplified mnemonics that incorporate CR conditions (eliminates BO and replaces BI with crS)*."
2. Simplified mnemonics for branch instructions that do not test CR bits should specify only a target. A programming error may occur.

Table 289 provides simplified mnemonics and syntax for **bclrl** and **bcctrl** with LR updating.

Table 289. Simplified mnemonics for bclrl and bcctrl with LR update

Branch semantics	bclrl	Simplified mnemonic	bcctrl	simplified mnemonic
Branch unconditionally	**bclrl 20,0**	**blrl** [(1)]	**bcctrl 20,0**	**bctrl** [1]
Branch if condition true	**bclrl 12,BI**	**btlrl BI**	**bcctrl 12,BI**	**btctrl BI**
Branch if condition false	**bclrl 4,BI**	**bflrl BI**	**bcctrl 4,BI**	**bfctrl BI**
Decrement CTR, branch if CTR ≠ 0	**bclrl 16,0**	**bdnzlrl** [1]	—	—
Decrement CTR, branch if CTR ≠ 0, condition true	**bclrl 8,BI**	**bdnztlrl BI**	—	—
Decrement CTR, branch if CTR ≠ 0, condition false	**bclrl 0,BI**	**bdnzflrl BI**	—	—
Decrement CTR, branch if CTR = 0	**bclrl 18,0**	**bdzlrl** [1]	—	—
Decrement CTR, branch if CTR = 0, condition true	**bclrl 10, BI**	**bdztlrl BI**	—	—
Decrement CTR, branch if CTR = 0, condition false	**bclrl 2,BI**	bdzflrl **BI**	—	—

1. Simplified mnemonics for branch instructions that do not test a CR bit should not specify one. A programming error may occur.

B.4.6 Simplified mnemonics that incorporate CR conditions (eliminates BO and replaces BI with crS)

The mnemonics in *Table 292* are variations of the branch-if-condition-true (BO = 12) and branch-if-condition-false (BO = 4) encodings. Because these instructions do not depend on the CTR, the true/false conditions specified by BO can be combined with the CR test bit specified by BI to create a different set of simplified mnemonics that eliminates the BO operand and the portion of the BI operand (BI[3–4]) that specifies one of the four possible

test bits. However, the simplified mnemonic cannot specify in which of the eight CR fields the test bit falls, so the BI operand is replaced by a **cr**S operand.

The standard codes shown in *Table 290* are used for the most common combinations of branch conditions. Note that for ease of programming, these codes include synonyms; for example, less than or equal (**le**) and not greater than (**ng**) achieve the same result.

*Note: A CR field symbol, **cr0**–**cr7**, is used as the first operand after the simplified mnemonic. If CR0 is used, no **cr**S is necessary.*

Table 290. Standard coding for branch conditions

Code	Description	Equivalent	Bit tested
lt	Less than	—	LT
le	Less than or equal (equivalent to **ng**)	**ng**	GT
eq	Equal	—	EQ
ge	Greater than or equal (equivalent to **nl**)	**nl**	LT
gt	Greater than	—	GT
nl	Not less than (equivalent to **ge**)	**ge**	LT
ne	Not equal	—	EQ
ng	Not greater than (equivalent to **le**)	**le**	GT
so	Summary overflow	—	SO
ns	Not summary overflow	—	SO

Table 291 shows the syntax for simplified branch mnemonics that incorporate CR conditions. Here, **cr**S replaces a BI operand to specify only a CR field (because the specific CR bit within the field is now part of the simplified mnemonic. Note that the default is CR0; if no **cr**S is specified, CR0 is used.

Table 291. Branch instructions and simplified mnemonics that incorporate CR conditions

Instruction	Standard mnemonic	Syntax	Simplified mnemonic	Syntax
Branch	b (ba bl bla)	target_addr	—	
Branch Conditional	bc (bca bcl bcla)	BO,BI,target_addr	**b***x* (1)(**b***x***a b***x***l b***x***la**)	**cr**S(2),target_addr
Branch Conditional to Link Register	bclr (bclr**l**)	BO,BI	b*x*lr (b*x*lr**l**)	**cr**S
Branch Conditional to Count Register	bcctr (bcctr**l**)	BO,BI	b*x*ctr (b*x*ctr**l**)	**cr**S

1. *x* stands for one of the symbols in *Table 290*, where applicable.
2. BI can be a numeric value or an expression as shown in *Table 283*.

Table 292 shows the simplified branch mnemonics incorporating conditions.

Table 292. Simplified mnemonics with comparison conditions

Branch semantics	LR update not enabled				LR update enabled			
	bc	bca	bclr	bcctr	bcl	bcla	bclrl	bcctrl
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**

Instructions using the mnemonics in *Table 292* indicate the condition bit, but not the CR field. If no field is specified, CR0 is used. The CR field symbols defined in *Table 283* (**cr0**–**cr7**) are used for this operand, as shown in examples 2–4 below.

Branch simplified mnemonics that incorporate CR conditions: examples

The following examples use the simplified mnemonics shown in *Table 292*:

1. Branch if CR0 reflects not-equal condition.
   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 condition, setting the LR. 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 CTR.
   bgtctrl cr4 equivalent to **bcctrl 12,17**

Branch simplified mnemonics that incorporate CR conditions: listings

Table 293 shows simplified branch mnemonics and syntax for **bc** and **bca** without LR updating.

Table 293. Simplified mnemonics for bc and bca without comparison conditions or LR Update

Branch Semantics	bc	Simplified mnemonic	bca	Simplified mnemonic
Branch if less than	**bc 12,**BI(1),target	**blt cr**S target	**bca 12,**BI[1],target	**blta cr**S target
Branch if less than or equal	**bc 4,**BI(2),target	**ble cr**S target	**bca 4,**BI[2],target	**blea cr**S target
Branch if not greater than		**bng cr**S target		**bnga cr**S target
Branch if equal	**bc 12,**BI(3),target	**beq cr**S target	**bca 12,**BI[3],target	**beqa cr**S target
Branch if greater than or equal	**bc 4,**BI[1],target	**bge cr**S target	**bca 4,**BI[1],target	**bgea cr**S target
Branch if not less than		**bnl cr**S target		**bnla cr**S target
Branch if greater than	**bc 12,**BI[2],target	**bgt cr**S target	**bca 12,**BI[2],target	**bgta cr**S target
Branch if not equal	**bc 4,**BI[3],target	**bne cr**S target	**bca 4,**BI[3],target	**bnea cr**S target
Branch if summary overflow	**bc 12,**BI(4),target	**bso cr**S target	**bca 12,**BI[4],target	**bsoa cr**S target
Branch if unordered		**bun cr**S target		**buna cr**S target
Branch if not summary overflow	**bc 4,**BI[4],target	**bns cr**S target	**bca 4,**BI[4],target	**bnsa cr**S target
Branch if not unordered		**bnu cr**S target		**bnua cr**S target

1. The value in the BI operand selects CR*n*[0], the LT bit.
2. The value in the BI operand selects CR*n*[1], the GT bit.
3. The value in the BI operand selects CR*n*[2], the EQ bit.
4. The value in the BI operand selects CR*n*[3], the SO bit.

Table 294 shows simplified branch mnemonics and syntax for **bclr** and **bcctr** without LR updating.

Table 294. Simplified mnemonics for bclr and bcctr without comparison conditions or LR update

Branch semantics	bclr	Simplified mnemonic	bcctr	Simplified mnemonic
Branch if less than	**bclr 12,**BI(1),target	**bltlr cr**S target	**bcctr 12,**BI[1],target	**bltctr cr**S target
Branch if less than or equal	**bclr 4,**BI(2),target	**blelr cr**S target	**bcctr 4,**BI[2],target	**blectr cr**S target
Branch if not greater than		**bnglr cr**S target		**bngctr cr**S target
Branch if equal	**bclr 12,**BI(3),target	**beqlr cr**S target	**bcctr 12,**BI[3],target	**beqctr cr**S target
Branch if greater than or equal	**bclr 4,**BI[1],target	**bgelr cr**S target	**bcctr 4,**BI[1],target	**bgectr cr**S target
Branch if not less than		**bnllr cr**S target		**bnlctr cr**S target
Branch if greater than	**bclr 12,**BI[2],target	**bgtlr cr**S target	**bcctr 12,**BI[2],target	**bgtctr cr**S target
Branch if not equal	**bclr 4,**BI[3],target	**bnelr cr**S target	**bcctr 4,**BI[3],target	**bnectr cr**S target
Branch if summary overflow	**bclr 12,**BI(4),target	**bsolr cr**S target	**bcctr 12,**BI[4],target	**bsoctr cr**S target
Branch if not summary overflow	**bclr 4,**BI[4],target	**bnslr cr**S target	**bcctr 4,**BI[4],target	**bnsctr cr**S target

1. The value in the BI operand selects CR*n*[0], the LT bit.
2. The value in the BI operand selects CR*n*[1], the GT bit.
3. The value in the BI operand selects CR*n*[2], the EQ bit.
4. The value in the BI operand selects CR*n*[3], the SO bit.

Table 295 shows simplified branch mnemonics and syntax for **bcl** and **bcla**.

Table 295. Simplified mnemonics for bcl and bcla with comparison conditions, LR update

Branch semantics	bcl	Simplified mnemonic	bcla	Simplified mnemonic
Branch if less than	**bcl 12,**BI(1),target	**bltl crS** target	**bcla 12,**BI1,target	**bltla crS** target
Branch if less than or equal	**bcl 4,**BI(2),target	**blel crS** target	**bcla 4,**BI2,target	**blela crS** target
Branch if not greater than		**bngl crS** target		**bngla crS** target
Branch if equal	**bcl 12,**BI(3),target	**beql crS** target	**bcla 12,**BI3,target	**beqla crS** target
Branch if greater than or equal	**bcl 4,**BI1,target	**bgel crS** target	**bcla 4,**BI1,target	**bgela crS** target
Branch if not less than		**bnll crS** target		**bnlla crS** target
Branch if greater than	**bcl 12,**BI2,target	**bgtl crS** target	**bcla 12,**BI2,target	**bgtla crS** target
Branch if not equal	**bcl 4,**BI3,target	**bnel crS** target	**bcla 4,**BI3,target	**bnela crS** target
Branch if summary overflow	**bcl 12,**BI(4),target	**bsol crS** target	**bcla 12,**BI4,target	**bsola crS** target
Branch if not summary overflow	**bcl 4,**BI4,target	**bnsl crS** target	**bcla 4,**BI4,target	**bnsla crS** target

1. The value in the BI operand selects CR*n*[0], the LT bit.
2. The value in the BI operand selects CR*n*[1], the GT bit.
3. The value in the BI operand selects CR*n*[2], the EQ bit.
4. The value in the BI operand selects CR*n*[3], the SO bit.

Table 296 shows the simplified branch mnemonics and syntax for **bclrl** and **bcctrl** with LR updating.

Table 296. Simplified mnemonics for bclrl and bcctrl with comparison conditions, LR update

Branch semantics	bclrl	Simplified mnemonic	bcctrl	Simplified mnemonic
Branch if less than	**bclrl 12,**BI(1),target	**bltlrl crS** target	**bcctrl 12,**BI1,target	**bltctrl crS** target
Branch if less than or equal	**bclrl 4,**BI(2),target	**blelrl crS** target	**bcctrl 4,**BI2,target	**blectrl crS** target
Branch if not greater than		**bnglrl crS** target		**bngctrl crS** target
Branch if equal	**bclrl 12,**BI(3),target	**beqlrl crS** target	**bcctrl 12,**BI3,target	**beqctrl crS** target
Branch if greater than or equal	**bclrl 4,**BI1,target	**bgelrl crS** target	**bcctrl 4,**BI1,target	**bgectrl crS** target
Branch if not less than		**bnllrl crS** target		**bnlctrl crS** target
Branch if greater than	**bclrl 12,**BI2,target	**bgtlrl crS** target	**bcctrl 12,**BI2,target	**bgtctrl crS** target
Branch if not equal	**bclrl 4,**BI3,target	**bnelrl crS** target	**bcctrl 4,**BI3,target	**bnectrl crS** target
Branch if summary overflow	**bclrl 12,**B(4),target	**bsolrl crS** target	**bcctrl 12,**BI4,target	**bsoctrl crS** target
Branch if not summary overflow	**bclrl 4,**BI4,target	**bnslrl crS** target	**bcctrl 4,**BI4,target	**bnsctrl crS** target

1. The value in the BI operand selects CR*n*[0], the LT bit.
2. The value in the BI operand selects CR*n*[1], the GT bit.
3. The value in the BI operand selects CR*n*[2], the EQ bit.
4. The value in the BI operand selects CR*n*[3], the SO bit.

B.5 Compare word simplified mnemonics

In compare word instructions, the L operand indicates a word (L = 0) or a double-word (L = 1). Simplified mnemonics in *Table 297* eliminate the L operand for word comparisons.

Table 297. Word compare simplified mnemonics

Operation	Simplified mnemonic	Equivalent to:
Compare Word Immediate	**cmpwi** **cr**D,**r**A,SIMM	**cmpi** **cr**D,**0**,**r**A,SIMM
Compare Word	**cmpw** **cr**D,**r**A,**r**B	**cmp** **cr**D,**0**,**r**A,**r**B
Compare Logical Word Immediate	**cmplwi** **cr**D,**r**A,UIMM	**cmpli** **cr**D,**0**,**r**A,UIMM
Compare Logical Word	**cmplw** **cr**D,**r**A,**r**B	**cmpl** **cr**D,**0**,**r**A,**r**B

As with branch mnemonics, the **cr**D field of a compare instruction can be omitted if CR0 is used, as shown in examples 1 and 3 below. Otherwise, the target CR field must be specified as the first operand. The following examples use word compare mnemonics:

1. Compare **r**A with immediate value 100 as signed 32-bit integers and place result in CR0.
   cmpwi **r**A,**100** equivalent to **cmpi** **0**,**0**,**r**A,**100**
2. Same as (1), but place results in CR4.
   cmpwi **cr4**,**r**A,**100**equivalent to **cmpi** **4**,**0**,**r**A,**100**
3. Compare **r**A and **r**B as unsigned 32-bit integers and place result in CR0.
   cmplw **r**A,**r**B equivalent to **cmpl** **0**,**0**,**r**A,**r**B

B.6 Condition register logical simplified mnemonics

The CR logical instructions, shown in *Table 298*, can be used to set, clear, copy, or invert a given CR bit. Simplified mnemonics allow these operations to be coded easily. Note that the symbols defined in *Table 282* can be used to identify the CR bit.

Table 298. Condition register logical simplified mnemonics

Operation	Simplified mnemonic	Equivalent to
Condition register set	**crset** **b**x	**creqv** **b**x,**b**x,**b**x
Condition register clear	**crclr** **b**x	**crxor** **b**x,**b**x,**b**x
Condition register move	**crmove** **b**x,**b**y	**cror** **b**x,**b**y,**b**y
Condition register not	**crnot** **b**x,**b**y	**crnor** **b**x,**b**y,**b**y

Examples using the CR logical mnemonics follow:

1. Set CR[57].
   crset 25 equivalent to **creqv 25,25,25**
2. Clear CR0[SO].
   crclr so equivalent to **crxor 3,3,3**
3. Same as (2), but clear CR3[SO].
   crclr 4 * cr3 + so equivalent to **crxor 15,15,15**
4. Invert the CR0[EQ].
   crnot eq,eq equivalent to **crnor 2,2,2**
5. Same as (4), but CR4[EQ] is inverted and the result is placed into CR5[EQ].
   crnot 4 * cr5 + eq, 4 * cr4 + eq equivalent to **crnor 22,18,18**

B.7 Trap instructions simplified mnemonics

The codes in *Table 299* are for the most common combinations of trap conditions.

Table 299. Standard codes for trap instructions

Code	Description	TO encoding	<	>	=	<U (1)	>U (2)
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
—	Unconditional	31	1	1	1	1	1

1. The symbol ‘<U’ indicates an unsigned less-than evaluation is performed.
2. The symbol ‘>U’ indicates an unsigned greater-than evaluation is performed.

The mnemonics in *Table 300* are variations of trap instructions, with the most useful TO values represented in the mnemonic rather than specified as a numeric operand.

Table 300. Trap simplified mnemonics

Trap semantics	32-Bit Comparison	
	twi Immediate	tw Register
Trap unconditionally	—	**trap**
Trap if less than	**twlti**	**twlt**
Trap if less than or equal	**twlei**	**twle**
Trap if equal	**tweqi**	**tweq**
Trap if greater than or equal	**twgei**	**twge**
Trap if greater than	**twgti**	**twgt**
Trap if not less than	**twnli**	**twnl**
Trap if not equal	**twnei**	**twne**
Trap if not greater than	**twngi**	**twng**
Trap if logically less than	**twllti**	**twllt**
Trap if logically less than or equal	**twllei**	**twlle**
Trap if logically greater than or equal	**twlgei**	**twlge**
Trap if logically greater than	**twlgti**	**twlgt**
Trap if logically not less than	**twlnli**	**twlnl**
Trap if logically not greater than	**twlngi**	**twlng**

The following examples use the trap mnemonics shown in *Table 300*:

1. Trap if **r**A is not zero.
   twnei rA**,0** equivalent to **twi 24,r**A**,0**
2. Trap if **r**A is not equal to **r**B.
   twne rA**, r**B equivalent to **tw 24,r**A**,r**B
3. Trap if **r**A is logically greater than 0x7FF.
   twlgti rA**,** 0x7FF equivalent to **twi 1,r**A**,** 0x7FF
4. Trap unconditionally.
   trap equivalent to **tw 31,0,0**

Trap instructions evaluate a trap condition as follows: The contents of **r**A are compared with either the sign-extended SIMM field or the contents of **r**B, depending on the trap instruction.

The comparison results in five conditions that are ANDed with operand TO. If the result is not 0, the trap exception handler is invoked. See *Table 301* for these conditions.

Table 301. TO operand bit encoding

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

B.8 Simplified mnemonics for accessing SPRs

The **mtspr** and **mfspr** instructions specify a special-purpose register (SPR) as a numeric operand. Simplified mnemonics are provided that represent the SPR in the mnemonic rather than requiring it to be coded as a numeric operand. The pattern for **mtspr** and **mfspr** simplified mnemonics is straightforward: replace the -**spr** portion of the mnemonic with the abbreviation for the spr (for example XER, SRR0, or LR), eliminate the SPRN operand, leaving the source or destination GPR operand, **r**S or **r**D.

Following are examples using the SPR simplified mnemonics:

1. Copy the contents of **r**S to the XER.
   mtxer rS equivalent to **mtspr 1,r**S
2. Copy the contents of the LR to **r**D.
   mflr rD equivalent to **mfspr r**D**,8**
3. Copy the contents of **r**S to the CTR.
   mtctr rS equivalent to **mtspr 9,r**S

The examples above show simplified mnemonics for accessing SPRs defined by the AIM version of the PowerPC architecture; however, the same formula is used for Book E, EIS, and implementation-specific SPRs, as shown in the following examples:

1. Copy the contents of **r**S to CSRR0.
   mtcsrr0 rS equivalent to **mtspr 58,r**S
2. Copy the contents of IVOR0 to **r**D.
   mfivor0 rD equivalent to **mfspr r**D**,400**
3. Copy the contents of **r**S to the MAS1.
   mtmas1 rS equivalent to **mtspr 625,r**S

There is an additional simplified mnemonic formula for accessing SPRGs, although not all of these more complicated simplified mnemonics are supported by all assemblers. These are shown in *Table 302* along with the equivalent simplified mnemonic using the formula described above.

Table 302. Additional simplified mnemonics for accessing SPRGs

SPR	Move to SPR		Move from SPR	
	Simplified mnemonic	Equivalent to	Simplified mnemonic	Equivalent to
SPRGs	**mtsprg** *n,* **r**S	**mtspr** 272 + *n*,**r**S	**mfsprg r**D*, n*	**mfspr r**D,272 + *n*
	mtsprg*n,* **r**S		**mfsprg***n* **r**D	

B.9 Recommended simplified mnemonics

This section describes commonly-used operations (such as no-op, load immediate, load address, move register, and complement register).

B.9.1 No-op (nop)

Many instructions can be coded so that, effectively, no operation is performed. A 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 following:

nop equivalent to **ori 0,0,0**

B.9.2 Load immediate (li)

The **addi** and **addis** instructions can be used to load an immediate value into a register. Additional mnemonics are provided to convey the idea that no addition is being performed but that data is being moved from the immediate operand of the instruction to a register.

1. Load a 16-bit signed immediate value into **r**D.
   li rD,value equivalent to **addi r**D,**0,**value
2. Load a 16-bit signed immediate value, shifted left by 16 bits, into **r**D.
   lis rD,value equivalent to **addis r**D,**0,**value

B.9.3 Load address (la)

This mnemonic permits computing the value of a base-displacement operand, using the **addi** instruction that normally requires a separate register and immediate operands.

la rD,d(**r**A) equivalent to **addi r**D,**r**A,d

The **la** mnemonic is useful for obtaining the address of a variable specified by name, allowing the assembler to supply the base register number and compute the displacement. If the variable *v* is located at offset d*v* bytes from the address in **r***v,* and the assembler has been told to use **r***v* as a base for references to the data structure containing *v,* the following line causes the address of *v* to be loaded into **r**D:

la rD,*v* equivalent to **addi r**D,**r***v*,d*v*

B.9.4 Move register (mr)

Several instructions can be coded to copy the contents of one register to another. A simplified mnemonic is provided that signifies that no computation is being performed, but merely that data is being moved from one register to another.

The following instruction copies the contents of **r**S into **r**A. This mnemonic can be coded with a dot (**.**) suffix to cause the Rc bit to be set in the underlying instruction.

mr rA,**r**S equivalent to **or r**A,**r**S,**r**S

B.9.5 Complement register (not)

Several instructions can be coded in a way that they complement the contents of one register and place the result into another register. Simplified mnemonics allows this operation to be coded easily.

The following instruction complements the contents of **r**S and places the result into **r**A. This mnemonic can be coded with a dot (**.**) suffix to cause the Rc bit to be set in the underlying instruction.

not rA,**r**S equivalent to **nor r**A,**r**S,**r**S

B.9.6 Move to condition register (mtcr)

This mnemonic permits copying GPR contents to the CR, using the same syntax as the **mfcr** instruction.

mtcr rS equivalent to **mtcrf** 0xFF,**r**S

B.10 EIS-specific simplified mnemonics

This section describes simplified mnemonics for instructions defines by auxiliary processing units (APUs) defined as part of the Motorola Book E implementation standards.

B.10.1 Integer select (isel)

The following mnemonics simplify the most common variants of the **isel** instruction that access CR0:

Integer Select Less Than
isellt **r**D,**r**A,**r**B equivalent to **isel** **r**D,**r**A,**r**B,**0**

Integer Select Greater Than
iselgt **r**D,**r**A,**r**B equivalent to **isel** **r**D,**r**A,**r**B,**1**

Integer Select Equal
iseleq **r**D,**r**A,**r**B equivalent to **isel** **r**D,**r**A,**r**B,**2**

B.10.2 SPE mnemonics

The following mnemonic handles moving of the full 64-bit SPE GPR:

Vector Move
evmr **r**D,**r**A equivalent to **evor** **r**D,**r**A,**r**A

The following mnemonic performs a complement register:

Vector Not
evnot **r**D,**r**A equivalent to **evnor** **r**D,**r**A,**r**A

B.11 Comprehensive list of simplified mnemonics

Table 303 lists simplified mnemonics. Note that compiler designers may implement additional simplified mnemonics not listed here.

Table 303. Simplified mnemonics

Simplified mnemonic	Mnemonic	Instruction
bctr(1)	**bcctr 20,0**	Branch unconditionally (**bcctr** without LR update)
bctrl 1	**bcctrl 20,0**	Branch unconditionally (**bcctrl** with LR Update)
bdnz target 1	**bc 16,0,**target	Decrement CTR, branch if CTR ≠ 0 (**bc** without LR update)
bdnza target 1	**bca 16,0,**target	Decrement CTR, branch if CTR ≠ 0 (**bca** without LR update)
bdnzf BI,target	**bc 0,BI,**target	Decrement CTR, branch if CTR ≠ 0 and condition false (**bc** without LR update)
bdnzfa BI,target	**bca 0,BI,**target	Decrement CTR, branch if CTR ≠ 0 and condition false (**bca** without LR update)
bdnzfl BI,target	**bcl 0,BI,**target	Decrement CTR, branch if CTR ≠ 0 and condition false (**bcl** with LR update)
bdnzfla BI,target	**bcla 0,BI,**target	Decrement CTR, branch if CTR ≠ 0 and condition false (**bcla** with LR update)

Table 303. Simplified mnemonics (continued)

Simplified mnemonic	Mnemonic	Instruction
bdnzflr BI	**bclr 0,BI**	Decrement CTR, branch if CTR ≠ 0 and condition false (**bclr** without LR update)
bdnzflrl BI	**bclrl 0,BI**	Decrement CTR, branch if CTR ≠ 0 and condition false (**bclrl** with LR Update)
bdnzl target [1]	**bcl 16,0,**target	Decrement CTR, branch if CTR ≠ 0 (**bcl** with LR update)
bdnzla target [1]	**bcla 16,0,**target	Decrement CTR, branch if CTR ≠ 0 (**bcla** with LR update)
bdnzlr BI	**bclr 16,BI**	Decrement CTR, branch if CTR ≠ 0 (**bclr** without LR update)
bdnzlrl [1]	**bclrl 16,0**	Decrement CTR, branch if CTR ≠ 0 (**bclrl** with LR Update)
bdnzt BI,target	**bc 8,BI,**target	Decrement CTR, branch if CTR ≠ 0 and condition true (**bc** without LR update)
bdnzta BI,target	**bca 8,BI,**target	Decrement CTR, branch if CTR ≠ 0 and condition true (**bc**a without LR update)
bdnztl BI,target	**bcl 8,0,**target	Decrement CTR, branch if CTR ≠ 0 and condition true (**bcl** with LR update)
bdnztla BI,target	**bcla 8,BI,**target	Decrement CTR, branch if CTR ≠ 0 and condition true (**bcla** with LR update)
bdnztlr BI	**bclr 8,BI**	Decrement CTR, branch if CTR ≠ 0 and condition true (**bclr** without LR update)
bdnztlr **BI**	**bclr 8,BI**	Decrement CTR, branch if CTR = 0 and condition true (**bclr** without LR update)
bdnztlrl BI	**bclrl 8,BI**	Decrement CTR, branch if CTR ≠ 0 and condition true (**bclrl** with LR Update)
bdz target [1]	**bc 18,0,**target	Decrement CTR, branch if CTR = 0 (**bc** without LR update)
bdza target [1]	**bca 18,0,**target	Decrement CTR, branch if CTR = 0 (**bca** without LR update)
bdzf BI,target	**bc 2,BI,**target	Decrement CTR, branch if CTR = 0 and condition false (**bc** without LR update)
bdzfa BI,target	**bca 2,BI,**target	Decrement CTR, branch if CTR = 0 and condition false (**bca** without LR update)
bdzfl BI,target	**bcl 2,BI,**target	Decrement CTR, branch if CTR = 0 and condition false (**bcl** with LR update)
bdzfla BI,target	**bcla 2,BI,**target	Decrement CTR, branch if CTR = 0 and condition false (**bcla** with LR update)
bdzflr BI	**bclr 2,BI**	Decrement CTR, branch if CTR = 0 and condition false (**bclr** without LR update)
bdzflrl **BI**	**bclrl 2,BI**	Decrement CTR, branch if CTR = 0 and condition false (**bclrl** with LR Update)
bdzl target [1]	**bcl 18,BI,**target	Decrement CTR, branch if CTR = 0 (**bcl** with LR update)
bdzla target [1]	**bcla 18,BI,**target	Decrement CTR, branch if CTR = 0 (**bcla** with LR update)

Table 303. Simplified mnemonics (continued)

Simplified mnemonic	Mnemonic	Instruction
bdzlr [1]	**bclr 18,0**	Decrement CTR, branch if CTR = 0 (**bclr** without LR update)
bdzlrl [1]	**bclrl 18,0**	Decrement CTR, branch if CTR = 0 (**bclrl** with LR Update)
bdzt BI,target	**bc 10,BI,**target	Decrement CTR, branch if CTR = 0 and condition true (**bc** without LR update)
bdzta BI,target	**bca 10,BI,**target	Decrement CTR, branch if CTR = 0 and condition true (**bca** without LR update)
bdztl BI,target	**bcl 10,BI,**target	Decrement CTR, branch if CTR = 0 and condition true (**bcl** with LR update)
bdztla BI,target	**bcla 10,BI,**target	Decrement CTR, branch if CTR = 0 and condition true (**bcla** with LR update)
bdztlrl BI	**bclrl 10, BI**	Decrement CTR, branch if CTR = 0 and condition true (**bclrl** with LR Update)
beq crS target	**bc 12,**BI[(2)]**,**target	Branch if equal (**bc** without comparison conditions or LR updating)
beqa crS target	**bca 12,**BI[2]**,**target	Branch if equal (**bca** without comparison conditions or LR updating)
beqctr crS target	**bcctr 12,**BI[2]**,**target	Branch if equal (**bcctr** without comparison conditions and LR updating)
beqctrl crS target	**bcctrl 12,**BI[2]**,**target	Branch if equal (**bcctrl** with comparison conditions and LR update)
beql crS target	**bcl 12,**BI[2]**,**target	Branch if equal (**bcl** with comparison conditions and LR updating)
beqla crS target	**bcla 12,**BI[2]**,**target	Branch if equal (**bcla** with comparison conditions and LR updating)
beqlr crS target	**bclr 12,**BI[2]**,**target	Branch if equal (**bclr** without comparison conditions and LR updating)
beqlrl crS target	**bclrl 12,**BI[2]**,**target	Branch if equal (**bclrl** with comparison conditions and LR update)
bf BI,target	**bc 4,BI,**target	Branch if condition false [(3)] (**bc** without LR update)
bfa BI,target	**bca 4,BI,**target	Branch if condition false [3] (**bca** without LR update)
bfctr BI	**bcctr 4,BI**	Branch if condition false [3] (**bcctr** without LR update)
bfctrl BI	**bcctrl 4,BI**	Branch if condition false [3](**bcctrl** with LR Update)
bfl BI,target	**bcl 4,BI,**target	Branch if condition false [3] (**bcl** with LR update)
bfla BI,target	**bcla 4,BI,**target	Branch if condition false [3] (**bcla** with LR update)
bflr BI	**bclr 4,BI**	Branch if condition false [3] (**bclr** without LR update)
bflrl BI	**bclrl 4,BI**	Branch if condition false [3](**bclrl** with LR Update)
bge crS target	**bc 4,**BI[(4)]**,**target	Branch if greater than or equal (**bc** without comparison conditions or LR updating)

Table 303. Simplified mnemonics (continued)

Simplified mnemonic	Mnemonic	Instruction
bgea **cr**S target	**bca 4,**BI[4]**,**target	Branch if greater than or equal (**bca** without comparison conditions or LR updating)
bgectr **cr**S target	**bcctr 4,**BI[4]**,**target	Branch if greater than or equal (**bcctr** without comparison conditions and LR updating)
bgectrl **cr**S target	**bcctrl 4,**BI[4]**,**target	Branch if greater than or equal (**bcctrl** with comparison conditions and LR update)
bgel **cr**S target	**bcl 4,**BI[4]**,**target	Branch if greater than or equal (**bcl** with comparison conditions and LR updating)
bgela **cr**S target	**bcla 4,**BI[4]**,**target	Branch if greater than or equal (**bcla** with comparison conditions and LR updating)
bgelr **cr**S target	**bclr 4,**BI[4]**,**target	Branch if greater than or equal (**bclr** without comparison conditions and LR updating)
bgelrl **cr**S target	**bclrl 4,**BI[4]**,**target	Branch if greater than or equal (**bclrl** with comparison conditions and LR update)
bgt **cr**S target	**bc 12,**BI[(5)]**,**target	Branch if greater than (**bc** without comparison conditions or LR updating)
bgta **cr**S target	**bca 12,**BI[5]**,**target	Branch if greater than (**bca** without comparison conditions or LR updating)
bgtctr **cr**S target	**bcctr 12,**BI[5]**,**target	Branch if greater than (**bcctr** without comparison conditions and LR updating)
bgtctrl **cr**S target	**bcctrl 12,**BI[5]**,**target	Branch if greater than (**bcctrl** with comparison conditions and LR update)
bgtl **cr**S target	**bcl 12,**BI[5]**,**target	Branch if greater than (**bcl** with comparison conditions and LR updating)
bgtla **cr**S target	**bcla 12,**BI[5]**,**target	Branch if greater than (**bcla** with comparison conditions and LR updating)
bgtlr **cr**S target	**bclr 12,**BI[5]**,**target	Branch if greater than (**bclr** without comparison conditions and LR updating)
bgtlrl **cr**S target	**bclrl 12,**BI[5]**,**target	Branch if greater than (**bclrl** with comparison conditions and LR update)
ble **cr**S target	**bc 4,**BI[5]**,**target	Branch if less than or equal (**bc** without comparison conditions or LR updating)
blea **cr**S target	**bca 4,**BI[5]**,**target	Branch if less than or equal (**bca** without comparison conditions or LR updating)
blectr **cr**S target	**bcctr 4,**BI[5]**,**target	Branch if less than or equal (**bcctr** without comparison conditions and LR updating)
blectrl **cr**S target	**bcctrl 4,**BI[5]**,**target	Branch if less than or equal (**bcctrl** with comparison conditions and LR update)
blel **cr**S target	**bcl 4,**BI[5]**,**target	Branch if less than or equal (**bcl** with comparison conditions and LR updating)

Table 303. Simplified mnemonics (continued)

Simplified mnemonic	Mnemonic	Instruction
blela crS target	**bcla 4,**BI[5]**,**target	Branch if less than or equal (**bcla** with comparison conditions and LR updating)
blelr crS target	**bclr 4,**BI[5]**,**target	Branch if less than or equal (**bclr** without comparison conditions and LR updating)
blelrl crS target	**bclrl 4,**BI[5]**,**target	Branch if less than or equal (**bclrl** with comparison conditions and LR update)
blr [1]	**bclr 20,0**	Branch unconditionally (**bclr** without LR update)
blrl [1]	**bclrl 20,0**	Branch unconditionally (**bclrl** with LR Update)
blt crS target	**bc 12,**BI**,**target	Branch if less than (**bc** without comparison conditions or LR updating)
blta crS target	**bca 12,**BI[4]**,**target	Branch if less than (**bca** without comparison conditions or LR updating)
bltctr crS target	**bcctr 12,**BI[4]**,**target	Branch if less than (**bcctr** without comparison conditions and LR updating)
bltctrl crS target	**bcctrl 12,**BI[4]**,**target	Branch if less than (**bcctrl** with comparison conditions and LR update)
bltl crS target	**bcl 12,**BI[4]**,**target	Branch if less than (**bcl** with comparison conditions and LR updating)
bltla crS target	**bcla 12,**BI[4]**,**target	Branch if less than (**bcla** with comparison conditions and LR updating)
bltlr crS target	**bclr 12,**BI[4]**,**target	Branch if less than (**bclr** without comparison conditions and LR updating)
bltlrl crS target	**bclrl 12,**BI[4]**,**target	Branch if less than (**bclrl** with comparison conditions and LR update)
bne crS target	**bc 4,**BI[3]**,**target	Branch if not equal (**bc** without comparison conditions or LR updating)
bnea crS target	**bca 4,**BI[3]**,**target	Branch if not equal (**bca** without comparison conditions or LR updating)
bnectr crS target	**bcctr 4,**BI[3]**,**target	Branch if not equal (**bcctr** without comparison conditions and LR updating)
bnectrl crS target	**bcctrl 4,**BI[3]**,**target	Branch if not equal (**bcctrl** with comparison conditions and LR update)
bnel crS target	**bcl 4,**BI[3]**,**target	Branch if not equal (**bcl** with comparison conditions and LR updating)
bnela crS target	**bcla 4,**BI[3]**,**target	Branch if not equal (**bcla** with comparison conditions and LR updating)
bnelr crS target	**bclr 4,**BI[3]**,**target	Branch if not equal (**bclr** without comparison conditions and LR updating)
bnelrl crS target	**bclrl 4,**BI[3]**,**target	Branch if not equal (**bclrl** with comparison conditions and LR update)

Table 303. Simplified mnemonics (continued)

Simplified mnemonic	Mnemonic	Instruction
bng **cr**S target	**bc 4,**BI[5]**,**target	Branch if not greater than (**bc** without comparison conditions or LR updating)
bnga **cr**S target	**bca 4,**BI[5]**,**target	Branch if not greater than (**bca** without comparison conditions or LR updating)
bngctr **cr**S target	**bcctr 4,**BI[5]**,**target	Branch if not greater than (**bcctr** without comparison conditions and LR updating)
bngctrl **cr**S target	**bcctrl 4,**BI[5]**,**target	Branch if not greater than (**bcctrl** with comparison conditions and LR update)
bngl **cr**S target	**bcl 4,**BI[5]**,**target	Branch if not greater than (**bcl** with comparison conditions and LR updating)
bngla **cr**S target	**bcla 4,**BI[5]**,**target	Branch if not greater than (**bcla** with comparison conditions and LR updating)
bnglr **cr**S target	**bclr 4,**BI[5]**,**target	Branch if not greater than (**bclr** without comparison conditions and LR updating)
bnglrl **cr**S target	**bclrl 4,**BI[5]**,**target	Branch if not greater than (**bclrl** with comparison conditions and LR update)
bnl **cr**S target	**bc 4,**BI[4]**,**target	Branch if not less than (**bc** without comparison conditions or LR updating)
bnla **cr**S target	**bca 4,**BI[4]**,**target	Branch if not less than (**bca** without comparison conditions or LR updating)
bnlctr **cr**S target	**bcctr 4,**BI[4]**,**target	Branch if not less than (**bcctr** without comparison conditions and LR updating)
bnlctrl **cr**S target	**bcctrl 4,**BI[4]**,**target	Branch if not less than (**bcctrl** with comparison conditions and LR update)
bnll **cr**S target	**bcl 4,**BI[4]**,**target	Branch if not less than (**bcl** with comparison conditions and LR updating)
bnlla **cr**S target	**bcla 4,**BI[4]**,**target	Branch if not less than (**bcla** with comparison conditions and LR updating)
bnllr **cr**S target	**bclr 4,**BI[4]**,**target	Branch if not less than (**bclr** without comparison conditions and LR updating)
bnllrl **cr**S target	**bclrl 4,**BI[4]**,**target	Branch if not less than (**bclrl** with comparison conditions and LR update)
bns **cr**S target	**bc 4,**BI[(6)]**,**target	Branch if not summary overflow (**bc** without comparison conditions or LR updating)
bnsa **cr**S target	**bca 4,**BI[6]**,**target	Branch if not summary overflow (**bca** without comparison conditions or LR updating)
bnsctr **cr**S target	**bcctr 4,**BI[6]**,**target	Branch if not summary overflow (**bcctr** without comparison conditions and LR updating)
bnsctrl **cr**S target	**bcctrl 4,**BI[6]**,**target	Branch if not summary overflow (**bcctrl** with comparison conditions and LR update)

Table 303. Simplified mnemonics (continued)

Simplified mnemonic	Mnemonic	Instruction
bnsl crS target	**bcl 4,**BI[6]**,**target	Branch if not summary overflow (**bcl** with comparison conditions and LR updating)
bnsla crS target	**bcla 4,**BI[6]**,**target	Branch if not summary overflow (**bcla** with comparison conditions and LR updating)
bnslr crS target	**bclr 4,**BI[6]**,**target	Branch if not summary overflow (**bclr** without comparison conditions and LR updating)
bnslrl crS target	**bclrl 4,**BI[6]**,**target	Branch if not summary overflow (**bclrl** with comparison conditions and LR update)
bso crS target	**bc 12,**BI[6]**,**target	Branch if summary overflow (**bc** without comparison conditions or LR updating)
bsoa crS target	**bca 12,**BI[6]**,**target	Branch if summary overflow (**bca** without comparison conditions or LR updating)
bsoctr crS target	**bcctr 12,**BI[6]**,**target	Branch if summary overflow (**bcctr** without comparison conditions and LR updating)
bsoctrl crS target	**bcctrl 12,**BI[6]**,**target	Branch if summary overflow (**bcctrl** with comparison conditions and LR update)
bsol crS target	**bcl 12,**BI[6]**,**target	Branch if summary overflow (**bcl** with comparison conditions and LR updating)
bsola crS target	**bcla 12,**BI[6]**,**target	Branch if summary overflow (**bcla** with comparison conditions and LR updating)
bsolr crS target	**bclr 12,**BI[6]**,**target	Branch if summary overflow (**bclr** without comparison conditions and LR updating)
bsolrl crS target	**bclrl 12,**BI[6]**,**target	Branch if summary overflow (**bclrl** with comparison conditions and LR update)
bt BI,target	**bc 12,BI,**target	Branch if condition true[3] (**bc** without LR update)
bta BI,target	**bca 12,BI,**target	Branch if condition true[3] (**bca** without LR update)
btctr BI	**bcctr 12,BI**	Branch if condition true [3] (**bcctr** without LR update)
btctrl BI	**bcctrl 12,BI**	Branch if condition true [3] (**bcctrl** with LR Update)
btl BI,target	**bcl 12,BI,**target	Branch if condition true [3] (**bcl** with LR update)
btla BI,target	**bcla 12,BI,**target	Branch if condition true [3] (**bcla** with LR update)
btlr BI	**bclr 12,BI**	Branch if condition true [3] (**bclr** without LR update)
btlrl BI	**bclrl 12,BI**	Branch if condition true [3] (**bclrl** with LR Update)
clrlslwi rA,**r**S,*b*,*n* ($n \le b \le 31$)	**rlwinm r**A,**r**S,*n*,*b* – *n*,31 – *n*	Clear left and shift left word immediate
clrlwi rA,**r**S,*n* ($n < 32$)	**rlwinm r**A,**r**S,**0,***n*,**31**	Clear left word immediate
clrrwi rA,**r**S,*n* ($n < 32$)	**rlwinm r**A,**r**S,**0,0,**31 – *n*	Clear right word immediate
cmplw crD,**r**A,**r**B	**cmpl cr**D,**0,r**A,**r**B	Compare logical word

Table 303. Simplified mnemonics (continued)

Simplified mnemonic	Mnemonic	Instruction
cmplwi **cr**D,**r**A,UIMM	**cmpli** **cr**D,**0**,**r**A,UIMM	Compare logical word immediate
cmpw **cr**D,**r**A,**r**B	**cmp** **cr**D,**0**,**r**A,**r**B	Compare word
cmpwi **cr**D,**r**A,SIMM	**cmpi** **cr**D,**0**,**r**A,SIMM	Compare word immediate
crclr bx	**crxor** bx,bx,bx	Condition register clear
crmove bx,by	**cror** bx,by,by	Condition register move
crnot bx,by	**crnor** bx,by,by	Condition register not
crset bx	**creqv** bx,bx,bx	Condition register set
evmr **r**D,**r**A	**evor** **r**D,**r**A,**r**A	Vector Move Register
evnot **r**D,**r**A	**evnor** **r**D,**r**A,**r**A	Vector Complement Register
evsubiw **r**D,**r**B,UIMM	**evsubifw** **r**D,UIMM,**r**B	Vector subtract word immediate
evsubw **r**D,**r**B,**r**A	**evsubfw** **r**D,**r**A,**r**B	Vector subtract word
extlwi **r**A,**r**S,*n*,*b* (*n* > 0)	**rlwinm** **r**A,**r**S,*b*,**0**,*n* – 1	Extract and left justify word immediate
extrwi **r**A,**r**S,*n*,*b* (*n* > 0)	**rlwinm** **r**A,**r**S,*b* + *n*, 32 – *n*,**31**	Extract and right justify word immediate
inslwi **r**A,**r**S,*n*,*b* (*n* > 0)	**rlwimi** **r**A,**r**S,32 – *b*,*b*,(*b* + *n*) – 1	Insert from left word immediate
insrwi **r**A,**r**S,*n*,*b* (*n* > 0)	**rlwimi** **r**A,**r**S,32 – (*b* + *n*),*b*,(*b* + *n*) – 1	Insert from right word immediate
iseleq **r**D,**r**A,**r**B	**isel** **r**D,**r**A,**r**B,**2**	Integer Select Equal
iselgt **r**D,**r**A,**r**B	**isel** **r**D,**r**A,**r**B,**1**	Integer Select Greater Than
isellt **r**D,**r**A,**r**B	**isel** **r**D,**r**A,**r**B,**0**	Integer Select Less Than
la **r**D,d(**r**A)	**addi** **r**D,**r**A,d	Load address
li **r**D,value	**addi** **r**D,**0**,value	Load immediate
lis **r**D,value	**addis** **r**D,**0**,value	Load immediate signed
mf*spr* rD	**mfspr** rD,SPRN	Move from SPR (see *Chapter B.8: Simplified mnemonics for accessing SPRs on page 1131.*")
mr **r**A,**r**S	**or** **r**A,**r**S,**r**S	Move register
mtcr **r**S	**mtcrf** 0xFF,**r**S	Move to Condition Register
mt*spr* rS	**mfspr** SPRN,rS	Move to SPR (see *Chapter B.8: Simplified mnemonics for accessing SPRs on page 1131.*")
nop	**ori** **0**,**0**,**0**	No-op
not **r**A,**r**S	**nor** **r**A,**r**S,**r**S	NOT
not **r**A,**r**S	**nor** **r**A,**r**S,**r**S	Complement register

Table 303. Simplified mnemonics (continued)

Simplified mnemonic	Mnemonic	Instruction
rotlw r**A**,r**S**,r**B**	**rlwnm** r**A**,r**S**,r**B**,**0**,**31**	Rotate left word
rotlwi r**A**,r**S**,*n*	**rlwinm** r**A**,r**S**,*n*,**0**,**31**	Rotate left word immediate
rotrwi r**A**,r**S**,*n*	**rlwinm** r**A**,r**S**,32 – *n*,**0**,**31**	Rotate right word immediate
slwi r**A**,r**S**,*n* (*n* < 32)	**rlwinm** r**A**,r**S**,*n*,**0**,31 – *n*	Shift left word immediate
srwi r**A**,r**S**,*n* (*n* < 32)	**rlwinm** r**A**,r**S**,32 – *n*,*n*,**31**	Shift right word immediate
sub r**D**,r**A**,r**B**	**subf** r**D**,r**B**,r**A**	Subtract from
subc r**D**,r**A**,r**B**	**subfc** r**D**,r**B**,r**A**	Subtract from carrying
subi r**D**,r**A**,value	**addi** r**D**,r**A**,–value	Subtract immediate
subic r**D**,r**A**,value	**addic** r**D**,r**A**,–value	Subtract immediate carrying
subic. r**D**,r**A**,value	**addic.** r**D**,r**A**,–value	Subtract immediate carrying
subis r**D**,r**A**,value	**addis** r**D**,r**A**,–value	Subtract immediate signed
tweq r**A**,SIMM	**tw 4**,r**A**,SIMM	Trap if equal
tweqi r**A**,SIMM	**twi 4**,r**A**,SIMM	Trap immediate if equal
twge r**A**,SIMM	**tw 12**,r**A**,SIMM	Trap if greater than or equal
twgei r**A**,SIMM	**twi 12**,r**A**,SIMM	Trap immediate if greater than or equal
twgt r**A**,SIMM	**tw 8**,r**A**,SIMM	Trap if greater than
twgti r**A**,SIMM	**twi 8**,r**A**,SIMM	Trap immediate if greater than
twle r**A**,SIMM	**tw 20**,r**A**,SIMM	Trap if less than or equal
twlei r**A**,SIMM	**twi 20**,r**A**,SIMM	Trap immediate if less than or equal
twlge r**A**,SIMM	**tw 12**,r**A**,SIMM	Trap if logically greater than or equal
twlgei r**A**,SIMM	**twi 12**,r**A**,SIMM	Trap immediate if logically greater than or equal
twlgt r**A**,SIMM	**tw 1**,r**A**,SIMM	Trap if logically greater than
twlgti r**A**,SIMM	**twi 1**,r**A**,SIMM	Trap immediate if logically greater than
twlle r**A**,SIMM	**tw 6**,r**A**,SIMM	Trap if logically less than or equal
twllei r**A**,SIMM	**twi 6**,r**A**,SIMM	Trap immediate if logically less than or equal
twllt r**A**,SIMM	**tw 2**,r**A**,SIMM	Trap if logically less than
twllti r**A**,SIMM	**twi 2**,r**A**,SIMM	Trap immediate if logically less than
twlng r**A**,SIMM	**tw 6**,r**A**,SIMM	Trap if logically not greater than
twlngi r**A**,SIMM	**twi 6**,r**A**,SIMM	Trap immediate if logically not greater than
twlnl r**A**,SIMM	**tw 5**,r**A**,SIMM	Trap if logically not less than

Table 303. Simplified mnemonics (continued)

Simplified mnemonic	Mnemonic	Instruction
twlnli rA,SIMM	**twi 5,r**A,SIMM	Trap immediate if logically not less than
twlt rA,SIMM	**tw 16,r**A,SIMM	Trap if less than
twlti rA,SIMM	**twi 16,r**A,SIMM	Trap immediate if less than
twne rA,SIMM	**tw 24,r**A,SIMM	Trap if not equal
twnei rA,SIMM	**twi 24,r**A,SIMM	Trap immediate if not equal
twng rA,SIMM	**tw 20,r**A,SIMM	Trap if not greater than
twngi rA,SIMM	**twi 20,r**A,SIMM	Trap immediate if not greater than
twnl rA,SIMM	**tw 12,r**A,SIMM	Trap if not less than
twnli rA,SIMM	**twi 12,r**A,SIMM	Trap immediate if not less than

1. Simplified mnemonics for branch instructions that do not test a CR bit should not specify one; a programming error may occur.
2. The value in the BI operand selects CR*n*[2], the EQ bit.
3. Instructions for which B0 is either 12 (branch if condition true) or 4 (branch if condition false) do not depend on the CTR value and can be alternately coded by incorporating the condition specified by the BI field, as described in *Chapter B.4.6: Simplified mnemonics that incorporate CR conditions (eliminates BO and replaces BI with crS) on page 1123*."
4. The value in the BI operand selects CR*n*[0], the LT bit.
5. The value in the BI operand selects CR*n*[1], the GT bit.
6. The value in the BI operand selects CR*n*[3], the SO bit.

Appendix C Programming examples

This appendix gives examples of how memory synchronization instructions can be used to emulate various synchronization primitives and to provide more complex forms of synchronization. It also describes multiple precision shifts.

C.1 Synchronization

Examples in this appendix have a common form. After possible initialization, a conditional sequence begins with a load and reserve instruction that may be followed by memory accesses and computations that include neither a load and reserve nor a store conditional. The sequence ends with a store conditional with the same target address as the initial load and reserve. In most of the examples, failure of the store conditional causes a branch back to the load and reserve for a repeated attempt. On the assumption that contention is low, the conditional branch in the examples is optimized for the case in which the store conditional succeeds, by setting the branch-prediction bit appropriately. These examples focus on techniques for the correct modification of shared memory locations: see note 4 in *Chapter C.1.4: Synchronization notes on page 1147*," for a discussion of how the retry strategy can affect performance.

Load and reserve and store conditional instructions depend on the coherence mechanism of the system. Stores to a given location are coherent if they are serialized in some order, and no processor is able to observe a subset of those stores as occurring in a conflicting order.

Each load operation, whether ordinary or load and reserve, returns a value that has a well-defined source. The source can be the store or store conditional instruction that wrote the value, an operation by some other mechanism that accesses memory (for example, an I/O device), or the initial state of memory.

The function of an atomic read/modify/write operation is to read a location and write its next value, possibly as a function of its current value, all as a single atomic operation. We assume that locations accessed by read/modify/write operations are accessed coherently, so the concept of a value being the next in the sequence of values for a location is well defined. The conditional sequence, as defined above, provides the effect of an atomic read/modify/write operation, but not with a single atomic instruction. Let *addr* be the location that is the common target of the load and reserve and store conditional instructions. Then the guarantee the architecture makes for the successful execution of the conditional sequence is that no store into *addr* by another processor or mechanism has intervened between the source of the load and reserve and the store conditional.

For each of these examples, it is assumed that a similar sequence of instructions is used by all processes requiring synchronization on the accessed data.

Note: Because memory synchronization instructions have implementation dependencies (for example, 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 (such as, test and set or compare and swap) needed by application programs. Application programs should use these library programs, rather than use memory synchronization instructions directly.

C.1.1 Synchronization primitives

The following examples show how the **lwarx** and **stwcx.** instructions can be used to implement various synchronization primitives.

The sequences used to emulate the various primitives consist primarily of a loop using **lwarx** and **stwcx.**. No additional synchronization is necessary, because the **stwcx.** will fail, clearing EQ, if the word loaded by **lwarx** has changed before the **stwcx.** is executed: see *: Atomic update primitives using lwarx and stwcx. on page 176* for details.

Fetch and No-op

The fetch and no-op primitive atomically loads the current value in a word in memory.

In this example it is assumed that the address of the word to be loaded is in GPR3 and the data loaded are returned in GPR4.

```
loop:   lwarx   r4,0,r3   #load and reserve
        stwcx.  r4,0,r3   #store old value if still reserved
        bc      4,2,loop  #loop if lost reservation
```

If the **stwcx.** succeeds, it 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** was the current value, that is, that the source of the value loaded by the **lwarx** was the last store to the location that preceded the **stwcx.** in the coherence order for the location.

Fetch and store

The fetch and store primitive atomically loads and replaces a word in memory. In this example it is assumed that the address of the word to be loaded and replaced is in GPR3, the new value is in GPR4, and the old value is returned in GPR5.

```
loop:   lwarx   r5,0,r3   #load and reserve
        stwcx.  r4,0,r3   #store new value if still reserved
        bc      4,2,loop  #loop if lost reservation
```

Fetch and add

The fetch and add primitive atomically increments a word in memory. In this example it is assumed that the address of the word to be incremented is in GPR3, the increment is in GPR4, and the old value is returned in GPR5.

```
loop:   lwarx   r5,0,r3   #load and reserve
        add     r0,r4,r5  #increment word
        stwcx.  r0,0,r3   #store new value if still reserved
        bc      4,2,loop  #loop if lost reservation
```

Fetch and AND

The Fetch and AND primitive atomically ANDs a value into a word in memory.

In this example it is assumed that the address of the word to be ANDed is in GPR3, the value to AND into it is in GPR4, and the old value is returned in GPR5.

```
loop:   lwarx   r5,0,r3   #load and reserve
        and     r0,r4,r5  #AND word
```

```
        stwcx.   r0,0,r3    #store new value if still reserved
        bc       4,2,loop   #loop if lost reservation
```

This sequence can be changed to perform another Boolean operation atomically on a word in memory by changing the **and** to the desired Boolean instruction (**or**, **xor**, etc.).

Test and set

This version of the test and set primitive atomically loads a word from memory, sets the word in memory 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 GPR3, the new value (nonzero) is in GPR4, and the old value is returned in GPR5.

```
loop:   lwarx    r5,0,r3    #load and reserve
        cmpwi    r5,0       #done if word
        bc       4,2,done   #not equal to 0
        stwcx.   r4,0,r3    #try to store non-0
        bc       4,2,loop   #loop if lost reservation
done:
```

Compare and swap

The compare and swap primitive atomically compares a value in a register with a word in memory, if they are equal stores the value from a second register into the word in memory, if they are unequal loads the word from memory into the first register, and sets CR0[EQ] to indicate the result of the comparison.

In this example it is assumed that the address of the word to be tested is in GPR3, the comparand is in GPR4 and the old value is returned there, and the new value is in GPR5.

```
loop:   lwarx    r6,0,r3    #load and reserve
        cmpw     r4,r6      #1st 2 operands equal?
        bc       4,2,exit   #skip if not
        stwcx.   r5,0,r3    #store new value if still reserved
        bc       4,2,loop   #loop if lost reservation
exit:   or       r4,r6,r6   #return value from memory
```

Note: 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 **bc** and/or the **or** can be omitted. The **bc** is needed only if the application requires that if CR0[EQ] on exit indicates not equal then GPR4 and GPR6 are not equal. The **or** is needed only if the application requires that if the comparands are not equal then the word from memory is loaded into the register with which it was compared (rather than into a third register). If any of these instructions is omitted, the resulting compare and swap does not obey System/370 semantics.*

C.1.2 Lock acquisition and release

This example gives an algorithm for locking that demonstrates the use of synchronization with an atomic read/modify/write operation. A shared memory location, the address of which is an argument of the lock and unlock procedures, given by GPR3, is used as a lock, to control access to some shared resource such as a shared data structure. The lock is open when its value is 0 and closed (locked) when its value is 1. Before accessing the shared resource the program executes the lock procedure, which sets the lock by changing its value from 0 to 1. To do this, the lock procedure calls test_and_set, which executes the code sequence shown in the test and set example of *Chapter C.1.1 on page 1144*," thereby atomically loading the old value of the lock, writing to the lock the new value (1) given in GPR4, returning the old value in GPR5 (not used below), and setting the EQ bit of CR Field 0 according to whether the value loaded is 0. The lock procedure repeats the test_and_set until it succeeds in changing the value of the lock from 0 to 1.

Because the shared resource must not be accessed until the lock has been set, the lock procedure contains an **isync** after the **bc** that checks for the success of test_and_set. The **isync** delays all subsequent instructions until all preceding instructions have completed.

```
lock:     mfspr     r6,LR       #save Link Register
          addi      r4,r0,1     #obtain lock:
loop:     bl        test_and_set#  test-and-set
          bc        4,2,loop    #  retry til old = 0
# Delay subsequent instructions til prior instructions finish
          isync
          mtspr     LR,r6       #restore Link Register
          blr                   #return
```

The unlock procedure stores a 0 to the lock location. Most applications that use locking require, for correctness, that if the access to the shared resource includes stores, the program must execute an **msync** before releasing the lock. The **msync** ensures that the program's modifications are performed with respect to other processors before the store that releases the lock is performed with respect to those processors. In this example, the unlock procedure begins with an **msync** for this purpose.

```
unlock:   msync                         #order prior stores
          addi      r1,r0,0     #before lock release
          stw       r1,0(r3)    #store 0 to lock location
          blr                   #return
```

C.1.3 List insertion

This example shows how **lwarx** and **stwcx.** 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 GPR3, the address of the new element is in GPR4, 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: see: *Atomic update primitives using lwarx and stwcx. on page 176*

```
loop:   lwarx   r2,0,r3     #get next pointer
        stw     r2,0(r4)    #store in new element
        msync               #order stw before stwcx.(can omit if not MP)
        stwcx.  r4,0,r3     #add new element to list
        bc      4,2,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 progress.)

If list elements cannot be allocated such that each element's next element pointer is in a different reservation granule, livelock can be avoided with this more complicated sequence:

```
        lwz     r2,0(r3)    #get next pointer
loop1:  or      r5,r2,r2    #keep a copy
        stw     r2,0(r4)    #store in new element
        msync               #order stw before stwcx.
loop2:  lwarx   r2,0,r3     #get it again
        cmpw    r2,r5       #loop if changed (someone
        bc      4,2,loop1   #  else progressed)
        stwcx.  r4,0,r3     #add new element to list
        bc      4,2,loop    #loop if failed
```

C.1.4 Synchronization notes

1. In general, **lwarx** and **stwcx.** should be paired, with the same effective address used for both. The only exception is that an unpaired **stwcx.** to any (scratch) effective address can be used to clear any reservation held by the processor.
2. It is acceptable to execute a **lwarx** for which no **stwcx.** is executed. For example, this occurs in the test and set sequence shown above if the value loaded is not zero.
3. To increase the likelihood that forward progress is made, it is important that looping on **lwarx**/**stwcx.** pairs be minimized. For example, in the sequence shown above for test and set, 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.**.
4. The manner in which **lwarx** and **stwcx.** are communicated to other processors and mechanisms, and between levels of the memory subsystem within a given processor is implementation-dependent (see: *Atomic update primitives using lwarx and stwcx. on page 176*). 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 example shown above, to stay in the loop until the word loaded is zero, `bne- $+12` can be changed to `bne- loop`. However, in some implementations better performance may be obtained by using an ordinary load instruction to do the initial checking of the value, as follows.

```
loop:   lwz     r5,0(r3)    #load the word
        cmpi    cr0,0,r5,0  #loop back if word
        bc      4,2,loop    #  not equal to 0
        lwarx   r5,0,r3     #try again, reserving
        cmpi    cr0,0,r5,0  #  (likely to succeed)
```

```
        bc      4,2,loop
        stwcx.  r4,0,r3     #try to store non-0
        bc      4,2,loop    #loop if lost reservation
```

5. In a multiprocessor, livelock is possible if a loop containing a **lwarx**/**stwcx.** pair also contains an ordinary store instruction for which any byte of the affected memory area is in the reservation granule: see*: Atomic update primitives using lwarx and stwcx. on page 176*. For example, the first code sequence shown in *Chapter C.1.3 on page 1146*," can cause livelock if two list elements have next element pointers in the same reservation granule.

C.2 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-word quantity (32-bit implementations), 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 0–63, which are the maximum ranges supported by the *Shift* instructions used. However if N>2, the shift amount must be in the range 0–31 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. 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 GPR6. 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).

Table 304. Shifts

Left shifts	Right shifts
Shift Left Immediate, N=3 (shift amount < 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 Right Immediate, N=3 (shift amount < 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 Left, N=2 (shift amount < 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 Right, N=2 (shift amount < 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 Left, N=3 (shift amount < 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, N=3 (shift amount < 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
	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

Table 304. Shifts (continued)

Left shifts	Right shifts
	Shift Right Algebraic, N=2 (shift amount < 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 bc 4,1,$+8 ori r3,r0,0 sraw r2,r2,r6

	Shift Right Algebraic, N=3 (shift amount < 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`

C.3 Floating point conversions

This section gives examples of how floating-point conversion instructions can be used to perform various conversions.

Note: Some of the examples use the optional 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.

C.3.1 Conversion from floating-point number to signed integer word

The full convert to signed integer word function can be implemented with the sequence shown below, assuming the floating-point value to be converted is in FPR1, the result is returned in GPR3, and a double word at displacement ‘disp’ from the address in GPR1 can be used as scratch space.

```
fctiw[z]    f2,f1          #convert to integer
stfd        f2,disp(r1)    #store float
lwa         r3,disp+4(r1)  #load word algebraic
                           #(use lwz on a 32-bit implementation)
```

C.3.2 Conversion from floating-point number to unsigned integer word

In a 32-bit implementation

The full convert to unsigned integer word function can be implemented with the sequence shown below, assuming the floating-point value to be converted is in FPR1, the value 0 is in FPR0, the value 232–1 is in FPR3, the value 231 is in FPR4, the result is returned in GPR3, and a double word at displacement ‘disp’ from the address in GPR1 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 2^31
fcmpu     cr2,f2,f4        #use diff if ≥ 2^31
fsel      f2,f5,f5,f2
fctiw[z]  f2,f2            #convert to integer
stfd      f2,disp(r1)      #store float
lwz       r3,disp+4(r1)    #load word
bc        12,8,$+8         #add 2^31 if input
xoris     r3,r3,0x8000     #  was ≥ 2^31
```

C.4 Floating point selection

This section gives examples of how the optional floating select instruction (fsel) 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 Book E instructions. In the examples, a, b, x, y, and z are floating-point variables, which are assumed to be in FPRs fa, fb, fx, fy, and fz. FPR fs is assumed to be available for scratch space.

Warning: Care must be taken in using fsel if IEEE compatibility is required, or if the values being tested can be NaNs or infinities: see Section C.4.1, “Notes.”

Table 305. Comparison to zero

High-Level Language:	Book E:	Notes
if a Š 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)	

Table 306. Minimum and maximum

High-Level Language:	Book E:	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)	

Table 307. Simple if-then-else constructions

High-Level Language:	Book E:	Notes
if a Š 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)	

C.4.1 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 Book E program shown, and the unoptimized program (not shown) is the corresponding Book E program that uses **fcmpu** and branch conditional instructions instead of **fsel**.

1. The unoptimized program affects FPSCR[VXSNAN] and therefore may cause the system error handler to be invoked if the corresponding exception is enabled; 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 FPSCR[OX, UX, XX,VXISI], and therefore may cause the system error handler to be invoked if the corresponding exceptions are enabled; the unoptimized program does not affect these bits. This property of the optimized program is incompatible with the IEEE standard.

Appendix D Guidelines for 32-bit book E

This appendix provides guidelines used by 32-bit Book E implementations; a set of guidelines is also outlined for software developers. Application software written to these guidelines can be labeled 32-bit Book E applications and can be expected to execute properly on all implementations of Book E, both 32-bit and 64-bit implementations.

32-bit Book E implementations execute applications that adhere to the software guidelines for 32-bit Book E software outlined in this appendix and are not expected to properly execute 64-bit Book E applications or any applications not adhering to these guidelines (that is, 64-bit Book E applications).

D.1 Registers on 32-bit book E implementations

Book E defines 32- and 64-bit registers. All 32-bit registers are supported as defined in Book E. However, except for the 64-bit FPRs, only bits 32–63 of Book E's 64-bit registers are required to be implemented in hardware in 32-bit Book E implementation. Such 64-bit registers include LR, CTR, 32 GPRs, SRR0, and CSRR0. Book E makes no restrictions regarding implementing a subset of the 64-bit floating-point architecture.

Likewise, other than floating-point instructions, all instructions defined to return a 64-bit result return only bits 32–63 of the result on a 32-bit Book E implementation.

D.2 Addressing on 32-bit book E implementations

Only bits 32–63 of the 64-bit Book E instruction and data memory effective addresses need to be calculated and presented to main memory, so a 32-bit implementation can bypass prepending the 32 zeros when implementing these instructions. For branch to LR and branch to CR instructions, given that LR and CTR are implemented as 32-bit registers, only 2 zeros need to be concatenated to the right of bits 32–61 of these registers to form the 32-bit branch target address.

The simplest implementation of next sequential instruction address computation suggests allowing effective address computations to wrap from 0xFFFF_FFFC to 0x0000_0000. This wrapping is required of PowerPC implementations. For 32-bit Book E applications, there appears little if any benefit to allowing this wrapping behavior. Book E specifies that the situation where the computation of the next sequential instruction address after address 0xFFFF_FFFC is undefined. (Note that the next sequential instruction address after address 0xFFFF_FFFC on a 64-bit Book E implementation is 0x0000_0001_0000_0000.)

D.3 TLB fields on 32-bit book E implementations

32-bit Book E 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 PowerPC ABIs may have come to expect to be available. 32-bit Book E 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.

D.4 32-bit book E software guidelines

D.4.1 32-bit instruction selection

Generally speaking, 32-bit software should avoid instructions that depend on any particular setting of bits 0–31 of any 64-bit application-accessible system register, including GPRs, for producing the correct 32-bit results. Context switching is not required to 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.

D.4.2 32-bit addressing

Book E provides a complete set of data memory access instructions that perform a modulo 2^{32} on the computed effective address and then prepend 32 zeros to produce the full 64-bit address. Book E also provides a complete set of branch instructions that perform a modulo 2^{32} on the computed branch target effective address and then prepend 32 zeros to produce the full 64-bit branch target address. On a 32-bit Book E implementation, these instructions are executed as defined, but without prepending the 32 zeros (only the low-order 32 bits of the address are calculated). On a 64-bit implementation, executing these instructions as defined provides the effect of restricting the application to the lowest 32-bit address space.

However, there is one exception. Next sequential instruction address computations (not a taken branch) are not defined for 32-bit Book E applications when the current instruction address is 0xFFFF_FFFC. On a 32-bit Book E implementation, the instruction address could simply wrap to 0x0000_0000, providing the same effect that is required in the PowerPC Architecture. However, when the 32-bit Book E application is executed on a 64-bit Book E implementation, the next sequential instruction address calculated will be 0x0000_0001_0000_0000 and not 0x0000_0000_0000_0000. To avoid this problem the 32-bit Book E application must either avoid this situation by not allowing code to span this address boundary, or requiring a branch absolute to address 0 be placed at address 0xFFFF_FFFC to emulate the wrap. Either of these approaches allows the application to execute on 32-bit and 64-bit Book E implementations.

Appendix E Embedded floating-point results

This appendix summarizes results of various types of floating-point operations on various combinations of input operands. Flag settings are performed on appropriate element flags.

E.1 Notation conventions and general rules

For all tables in this appendix, the annotation and general rules in *Table 308* apply.

Table 308. Notation conventions and general rules

Notation	Description
*	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 0x7F7_FFFFF. The encoding for double-precision is 0x7FEF_FFFF_FFFF_FFFF.
nmax	Denotes the maximum normalized negative number. The encoding for single-precision is 0xFF7F_FFFF. The encoding for double-precision is 0xFFEF_FFFF_FFFF_FFFF.
pmin	Denotes the minimum normalized positive number. The encoding for single-precision is 0x00800000. The encoding for double-precision is 0x0010_0000_0000_0000.
nmin	Denotes the minimum normalized negative number. The encoding for single-precision is 0x8080_0000. The encoding for double-precision is 0x8010_0000_0000_0000.
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 0x7FFF_FFFF (positive) or 0x8000_0000 (negative). Overflow for operations that have an unsigned integer result force the result to 0xFFFF_FFFF (positive) or 0x0000_0000 (negative).
1 (superscript)	Denotes that the sign of the result is positive when the signs of Operand A and Operand B are different, for all rounding modes except round to minus infinity, where the sign of the result is then negative
2 (superscript)	Denotes that the sign of the result is positive when the signs of Operand A and Operand B are the same, for all rounding modes except round to minus 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

E.2 Add, subtract, multiply, and divide results

Table 309 lists results for add, subtract, multiply, and divide operations.

Table 309. 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
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

Table 309. Floating-point results summary—add, sub, mul, div (continued)

Operation	Operand A	Operand B	Result	FINV	FOVF	FUNF	FDBZ	FINX
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
Mul	denorm	∞	zero	1	0	0	0	0
Mul	denorm	NaN	zero	1	0	0	0	0

Table 309. Floating-point results summary—add, sub, mul, div (continued)

Operation	Operand A	Operand B	Result	FINV	FOVF	FUNF	FDBZ	FINX
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

Table 309. Floating-point results summary—add, sub, mul, div (continued)

Operation	Operand A	Operand B	Result	FINV	FOVF	FUNF	FDBZ	FINX
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	*

E.3 Double- to single-precision conversion

Table 310 lists results for double- to single-precision conversion.

Table 310. 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	*

E.4 Single- to double-precision conversion

Table 311 lists results for single- to double-precision conversion.

Table 311. 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

E.5 Conversion to unsigned

Table 312 lists results for conversion to unsigned operations.

Table 312. Floating-point results summary—convert to unsigned

Operand B	Integer result ctui[d][z]	Fractional result ctuf	FINV	FOVF	FUNF	FDBZ	FINX
+∞	0xFFFF_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	*

E.6 Conversion to signed

Table 313 lists results for conversion to signed operations.

Table 313. 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	1	0	0	0	0
–∞	0x8000_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	*

E.7 Conversion from unsigned

Table 314 lists results for conversion from unsigned operations.

Table 314. 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	*

E.8 Conversion from signed

Table 315 lists results for conversion from signed operations.

Table 315. 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	*

E.9 *abs, *nabs, and *neg operations

Table 316 lists results for *abs, *nabs, and *neg operations.

Table 316. Floating-point results summary—*abs, *nabs, *neg

Operand A	*abs	*nabs	*neg	FINV	FOVF	FUNF	FDBZ	FINX
+∞	pmax I +∞	nmax I –∞	–amax I –∞	1	0	0	0	0
–∞	pmax I +∞	nmax I –∞	–amax I +∞	1	0	0	0	0
+NaN	pmax I NaN	nmax I –NaN	–amax I –NaN	1	0	0	0	0
–NaN	pmax I NaN	nmax I –NaN	–amax I +NaN	1	0	0	0	0
+denorm	+zero I +denorm	–zero I –denorm	–zero I –denorm	1	0	0	0	0
–denorm	+zero I +denorm	–zero I –denorm	+zero I +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

15 Glossary

The glossary contains an alphabetical list of terms, phrases, and abbreviations used in this book. Some of the terms and definitions included in the glossary are reprinted from IEEE Standard 754-1985, *IEEE Standard for Binary Floating-Point Arithmetic*, copyright ©1985 by the Institute of Electrical and Electronics Engineers, Inc. with the permission of the IEEE.

A

Architecture.

A detailed specification of requirements for a processor or computer system. It does not specify details of how the processor or computer system must be implemented; instead it provides a template for a family of compatible *implementations*.

Asynchronous interrupt.

interrupts that are caused by events external to the processor's execution. In this document, the term *asynchronous interrupt* is used interchangeably with the word *interrupt*.

Atomic access.

A bus access that attempts to be part of a read-write operation to the same address uninterrupted by any other access to that address (the term refers to the fact that the transactions are indivisible). The PowerPC architecture implements *atomic accesses* through the **lwarx**/**stwcx.** instruction pair.

B

Biased exponent.

An *exponent* whose range of values is shifted by a constant (bias). Typically a bias is provided to allow a range of positive values to express a range that includes both positive and negative values.

Big-endian.

A byte-ordering method in memory where the address *n* of a word corresponds to the *most-significant byte*. In an addressed memory word, the bytes are ordered (left to right) 0, 1, 2, 3, with 0 being the *most-significant byte*. See *Little-endian*.

Boundedly undefined.

A characteristic of certain operation results that are not rigidly prescribed by the PowerPC architecture. Boundedly-undefined results for a given operation may vary among implementations and between execution attempts in the same implementation.

Although the architecture does not prescribe the exact behavior for when results are allowed to be *boundedly undefined*, the results of executing instructions in contexts where results are allowed to be *boundedly undefined* are constrained to ones that could have been achieved by executing an arbitrary sequence of defined instructions, in valid form, starting in the state the machine was in before attempting to execute the given instruction.

Branch prediction.

The process of guessing whether a branch will be taken. Such predictions can be correct or incorrect; the term 'predicted' as it is used here does not imply that the prediction is correct

(successful). The PowerPC architecture defines a means for *static branch* prediction as part of the instruction encoding.

Branch resolution.

The determination of whether a branch is taken or not taken. A branch is said to be resolved when the processor can determine which instruction path to take. If the branch is resolved as predicted, the instructions following the predicted branch that may have been speculatively executed can complete (see *Completion*). If the branch is not resolved as predicted, instructions on the mispredicted path, and any results of speculative execution, are purged from the pipeline and fetching continues from the nonpredicted path.

C

Cache. High-speed memory containing recently accessed data or instructions (subset of main memory).

Cache block. A small region of contiguous memory that is copied from memory into a *cache*. The size of a *cache block* may vary among processors; the maximum block size is one *page*. In PowerPC processors, *cache coherency* is maintained on a cache-block basis. Note that the term *cache block* is often used interchangeably with 'cache line.'

Cache coherency. An attribute wherein an accurate and common view of memory is provided to all devices that share the same memory system. Caches are coherent if a processor performing a read from its cache is supplied with data corresponding to the most recent value written to memory or to another processor's cache.

Cache flush. An operation that removes from a cache any data from a specified address range. This operation ensures that any modified data within the specified address range is written back to main memory. This operation is generated typically by a Data Cache Block Flush (**dcbf**) instruction.

Caching-inhibited. A memory update policy in which the cache is bypassed and the load or store is performed to or from main memory.

Cast out. A *cache block* that must be written to memory when a cache miss causes a *cache block* to be replaced.

Changed bit. One of two *page history bits* found in each *page table entry* (PTE). The processor sets the changed bit if any store is performed into the *page*. See also *Page access history bits* and *Referenced bit*.

Clean. An operation that causes a *cache block* to be written to memory, if modified, and then left in a valid, unmodified state in the cache.

Clear. To cause a bit or bit field to register a value of zero. See also *Set*.

Completion. Completion occurs when an instruction has finished executing, written back any results, and is removed from the completion queue (CQ). When an instruction completes, it is guaranteed that this instruction and all previous instructions can cause no interrupts.

Context synchronization. An operation that ensures that all instructions in execution complete past the point where they can produce an *interrupt*, that all instructions in execution complete in the context in which they began execution, and that all subsequent instructions are *fetched* and executed in the new context. Context synchronization may result from executing specific instructions (such as **isync** or **rfi**) or when certain events occur (such as an *interrupt*).

D

Denormalized number.

A nonzero floating-point number whose exponent has a reserved value, usually the format's minimum, and whose explicit or implicit leading significand bit is zero.

E

Effective address (EA). The 32-bit address specified for a load, store, or an instruction fetch. This address is then submitted to the MMU for translation to either a *physical memory* address or an I/O address.

Exception. A condition that, if enabled, generates an interrupt.

Execution synchronization. A mechanism by which all instructions in execution are architecturally complete before beginning execution (appearing to begin execution) of the next instruction. Similar to context synchronization but doesn't force the contents of the instruction buffers to be deleted and refetched.

Exponent. In the binary representation of a floating-point number, the exponent is the component that normally signifies the integer power to which the value two is raised in determining the value of the represented number. See also *Biased exponent*.

F

Fetch. Instruction retrieval from either the cache or main memory and placing them into the instruction queue.

Finish. Finishing occurs in the last cycle of execution. In this cycle, the CQ entry is updated to indicate that the instruction has finished executing.

Floating-point register (FPR). Any of the 32 registers in the floating-point register file. These registers provide the source operands and destination results for floating-point instructions. Load instructions move data from memory to FPRs and store instructions move data from FPRs to memory. The FPRs are 64 bits wide and store floating-point values in double-precision format.

Floating-point unit. The functional unit in a processor responsible for executing all floating-point instructions.

Flush. An operation that causes a cache block to be invalidated and the data, if modified, to be written to memory.

Fraction. In the binary representation of a floating-point number, the field of the *significand* that lies to the right of its implied binary point.

G

General-purpose register (GPR). Any of the 32 registers in the general-purpose register file. These registers provide the source operands and destination results for all integer data manipulation instructions. Integer load instructions move data from memory to GPRs and store instructions move data from GPRs to memory.

Guarded. The guarded attribute pertains to out-of-order execution. When a page is designated as guarded, instructions and data cannot be accessed out-of-order.

H

Harvard architecture. An architectural model featuring separate caches and other memory management resources for instructions and data.

I

IEEE 754.

A standard written by the Institute of Electrical and Electronics Engineers that defines operations and representations of binary floating-point numbers.

Illegal instructions.

A class of instructions that are not implemented for a particular PowerPC processor. These include instructions not defined by the PowerPC architecture. In addition, for 32-bit implementations, instructions that are defined only for 64-bit implementations are considered to be illegal instructions. For 64-bit implementations instructions that are defined only for 32-bit implementations are considered to be illegal instructions.

Implementation.

A particular processor that conforms to the PowerPC architecture, but may differ from other architecture-compliant implementations for example in design, feature set, and implementation of *optional* features. The PowerPC architecture has many different implementations.

Imprecise interrupt.

A type of *synchronous interrupt* that is allowed not to adhere to the precise interrupt model (see *Precise interrupt*). The PowerPC architecture allows only floating-point exceptions to be handled imprecisely.

Integer unit.

The functional unit responsible for executing all integer instructions.

In order.

An aspect of an operation that adheres to a sequential model. 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. See *Out-of-order*.

Instruction latency.

The total number of clock cycles necessary to execute an instruction and make ready the results of that instruction.

Interrupt.

A condition encountered by the processor that requires special, supervisor-level processing.

Interrupt handler.

A software routine that executes when an interrupt is taken. Normally, the interrupt handler corrects the condition that caused the interrupt, or performs some other meaningful task (that may include aborting the program that caused the interrupt).

K

Kill.

An operation that causes a *cache block* to be invalidated without writing any modified data to memory.

L

Latency.

The number of clock cycles necessary to execute an instruction and make ready the results of that execution for a subsequent instruction.

L2 cache.

See *Secondary cache*.

Least-significant bit (lsb).

The bit of least value in an address, register, field, data element, or instruction encoding.

Least-significant byte (LSB).

The byte of least value in an address, register, data element, or instruction encoding.

Little-endian.

A byte-ordering method in memory where the address *n* of a word corresponds to the *least-significant byte*. In an addressed memory word, the bytes are ordered (left to right) 3, 2, 1, 0, with 3 being the *most-significant byte*. See *Big-endian*.

M

Mantissa.

The decimal part of logarithm.

Memory access ordering.

The specific order in which the processor performs load and store memory accesses and the order in which those accesses complete.

Memory-mapped accesses.

Accesses whose addresses use the page or block address translation mechanisms provided by the MMU and that occur externally with the bus protocol defined for memory.

Memory coherency.

An aspect of caching in which it is ensured that an accurate view of memory is provided to all devices that share system memory.

Memory consistency.

Refers to agreement of levels of memory with respect to a single processor and system memory (for example, on-chip cache, secondary cache, and system memory).

Memory management unit (MMU).

The functional unit that is capable of translating an *effective* (logical) *address* to a physical address, providing protection mechanisms, and defining caching methods.

Most-significant bit (msb).

The highest-order bit in an address, registers, data element, or instruction encoding.

Most-significant byte (MSB).

The highest-order byte in an address, registers, data element, or instruction encoding.

N

NaN.

An abbreviation for not a number; a symbolic entity encoded in floating-point format. There are two types of NaNs—signaling NaNs and quiet NaNs.

No-op.

No-operation. A single-cycle operation that does not affect registers or generate bus activity.

Normalization.

A process by which a floating-point value is manipulated such that it can be represented in the format for the appropriate precision (single- or double-precision). For a floating-point value to be representable in the single- or double-precision format, the leading implied bit must be a 1.

O

OEA (operating environment architecture).

The level of the architecture that describes PowerPC memory management model, supervisor-level registers, synchronization requirements, and the interrupt model. It also defines the time-base feature from a supervisor-level perspective. Implementations that conform to the PowerPC OEA also conform to the PowerPC UISA and VEA.

Optional.

A feature, such as an instruction, a register, or an interrupt, that is defined by the PowerPC architecture but not required to be implemented.

Out-of-order.

An aspect of an operation that allows it to be performed ahead of one that may have preceded it in the sequential model, for example, speculative operations. 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. See *In-order*.

Out-of-order execution.

A technique that allows instructions to be issued and completed in an order that differs from their sequence in the instruction stream.

Overflow.

An condition that occurs during arithmetic operations when the result cannot be stored accurately in the destination register(s). For example, if two 32-bit numbers are multiplied, the result may not be representable in 32 bits. Since 32-bit registers cannot represent this sum, an overflow condition occurs.

P

Page.

A region in memory. The OEA defines a page as a 4-Kbyte area of memory, aligned on a 4-Kbyte boundary.

Page fault.

A page fault is a condition that occurs when the processor attempts to access a memory location that does not reside within a *page* not currently resident in *physical memory*. On PowerPC processors, a page fault interrupt condition occurs when a matching, valid *page table entry* (PTE[V] = 1) cannot be located.

Physical memory.

The actual memory that can be accessed through the system's memory bus.

Pipelining.

A technique that breaks operations, such as instruction processing or bus transactions, into smaller distinct stages or tenures (respectively) so that a subsequent operation can begin before the previous one has completed.

Precise interrupts.

A category of interrupt for which the pipeline can be stopped so instructions that preceded the faulting instruction can complete and subsequent instructions can be flushed and redispatched after interrupt handling has completed. See *Imprecise interrupts*.

Primary opcode.

The most-significant 6 bits (bits 0–5) of the instruction encoding that identifies the type of instruction.

Program order.

The order of instructions in an executing program. More specifically, this term is used to refer to the original order in which program instructions are fetched into the instruction queue from the cache.

Protection boundary.

A boundary between *protection domains*.

Q

Quiet NaN.

A type of *NaN* that can propagate through most arithmetic operations without signaling interrupts. A quiet NaN is used to represent the results of certain invalid operations, such as invalid arithmetic operations on infinities or on NaNs, when invalid. See *Signaling NaN*.

R

Record bit.

Bit 31 (or the Rc bit) in the instruction encoding. When it is set, updates the condition register (CR) to reflect the result of the operation.

Referenced bit.

One of two *page history bits* found in each *page table entry*. The processor sets the *referenced bit* whenever the page is accessed for a read or write. See also *Page access history bits*.

Register indirect addressing.

A form of addressing that specifies one GPR that contains the address for the load or store.

Register indirect with immediate index addressing.

A form of addressing that specifies an immediate value to be added to the contents of a specified GPR to form the target address for the load or store.

Register indirect with index addressing.

A form of addressing that specifies that the contents of two GPRs be added together to yield the target address for the load or store.

Rename register.

Temporary buffers used by instructions that have finished execution but have not completed.

Reservation.

The processor establishes a reservation on a *cache block* of memory space when it executes an **lwarx** instruction to read a memory semaphore into a GPR.

Reservation station.

A buffer between the dispatch and execute stages that allows instructions to be dispatched even though the results of instructions on which the dispatched instruction may depend are not available.

RISC (reduced instruction set computing).

An *architecture* characterized by fixed-length instructions with nonoverlapping functionality and by a separate set of load and store instructions that perform memory accesses.

S

Secondary cache.

A cache memory that is typically larger and has a longer access time than the primary cache. A secondary cache may be shared by multiple devices. Also referred to as L2, or level-2, cache.

Set (*v*)**.**

To write a nonzero value to a bit or bit field; the opposite of *clear*. The term ‘set’ may also be used to generally describe the updating of a bit or bit field.

Set (*n*)**.**

A subdivision of a *cache*. Cacheable data can be stored in a given location in one of the sets, typically corresponding to its lower-order address bits. Because several memory locations can map to the same location, cached data is typically placed in the set whose *cache block* corresponding to that address was used least recently. See *Set-associative*.

Set-associative.

Aspect of cache organization in which the cache space is divided into sections, called *sets*. The cache controller associates a particular main memory address with the contents of a particular set, or region, within the cache.

Signaling NaN.

A type of *NaN* that generates an invalid operation program interrupt when it is specified as arithmetic operands. See *Quiet NaN*.

Significand.

The component of a binary floating-point number that consists of an explicit or implicit leading bit to the left of its implied binary point and a fraction field to the right.

Simplified mnemonics.

Assembler mnemonics that represent a more complex form of a common operation.

Snooping.

Monitoring addresses driven by a bus master to detect the need for coherency actions.

Split-transaction.

A transaction with independent request and response tenures.

Stall.

An occurrence when an instruction cannot proceed to the next stage.

Static branch prediction.

Mechanism by which software (for example, compilers) can hint to the machine hardware about the direction a branch is likely to take.

Superscalar.

A superscalar processor is one that can dispatch multiple instructions concurrently from a conventional linear instruction stream. In a superscalar implementation, multiple instructions can be in the same stage at the same time.

Supervisor mode.

The privileged operation state of a processor. In supervisor mode, software, typically the operating system, can access all control registers and can access the supervisor memory space, among other privileged operations.

Synchronization.

A process to ensure that operations occur strictly *in order*. See *Context synchronization* and *Execution synchronization*.

Synchronous interrupt.

An *interrupt* that is generated by the execution of a particular instruction or instruction sequence. There are two types of synchronous interrupts, *precise* and *imprecise*.

System memory.

The physical memory available to a processor.

T

TLB (translation lookaside buffer).

A cache that holds recently-used *page table entries*.

Throughput.

The measure of the number of instructions that are processed per clock cycle.

U

UISA (user instruction set architecture).

The level of the architecture to which user-level software should conform. The UISA defines the base user-level instruction set, user-level registers, data types, floating-point memory conventions and interrupt model as seen by user programs, and the memory and programming models.

Underflow.

A condition that occurs during arithmetic operations when the result cannot be represented accurately in the destination register. For example, underflow can happen if two floating-point fractions are multiplied and the result requires a smaller *exponent* and/or *mantissa* than the single-precision format can provide. In other words, the result is too small to be represented accurately.

User mode.

The operating state of a processor used typically by application software. In user mode, software can access only certain control registers and can access only user memory space. No privileged operations can be performed. Also referred to as problem state.

V

VEA (virtual environment architecture).

The level of the *architecture* that describes the memory model for an environment in which multiple devices can access memory, defines aspects of the cache model, defines cache control instructions, and defines the time-base facility from a user-level perspective. *Implementations* that conform to the PowerPC VEA also adhere to the UISA, but may not necessarily adhere to the OEA.

Virtual address.

An intermediate address used in the translation of an *effective address* to a physical address.

Virtual memory.

The address space created using the memory management facilities of the processor. Program access to *virtual memory* is possible only when it coinc

W

Way.

A location in the cache that holds a cache block, its tags and status bits.

Word.

A 32-bit data element.

Write-back.

A cache memory update policy in which processor write cycles are directly written only to the cache. External memory is updated only indirectly, for example, when a modified cache block is *cast out* to make room for newer data.

Write-through.

A cache memory update policy in which all processor write cycles are written to both the cache and memory.

16 Revision history

Table 317. Document revision history

Date	Revision	Changes
29-Nov-2007	1	Initial release.

Please Read Carefully:

Information in this document is provided solely in connection with ST products. STMicroelectronics NV and its subsidiaries (“ST”) reserve the right to make changes, corrections, modifications or improvements, to this document, and the products and services described herein at any time, without notice.

All ST products are sold pursuant to ST’s terms and conditions of sale.

Purchasers are solely responsible for the choice, selection and use of the ST products and services described herein, and ST assumes no liability whatsoever relating to the choice, selection or use of the ST products and services described herein.

No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted under this document. If any part of this document refers to any third party products or services it shall not be deemed a license grant by ST for the use of such third party products or services, or any intellectual property contained therein or considered as a warranty covering the use in any manner whatsoever of such third party products or services or any intellectual property contained therein.

UNLESS OTHERWISE SET FORTH IN ST’S TERMS AND CONDITIONS OF SALE ST DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY WITH RESPECT TO THE USE AND/OR SALE OF ST PRODUCTS INCLUDING WITHOUT LIMITATION IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE (AND THEIR EQUIVALENTS UNDER THE LAWS OF ANY JURISDICTION), OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

UNLESS EXPRESSLY APPROVED IN WRITING BY AN AUTHORIZED ST REPRESENTATIVE, ST PRODUCTS ARE NOT RECOMMENDED, AUTHORIZED OR WARRANTED FOR USE IN MILITARY, AIR CRAFT, SPACE, LIFE SAVING, OR LIFE SUSTAINING APPLICATIONS, NOR IN PRODUCTS OR SYSTEMS WHERE FAILURE OR MALFUNCTION MAY RESULT IN PERSONAL INJURY, DEATH, OR SEVERE PROPERTY OR ENVIRONMENTAL DAMAGE. ST PRODUCTS WHICH ARE NOT SPECIFIED AS "AUTOMOTIVE GRADE" MAY ONLY BE USED IN AUTOMOTIVE APPLICATIONS AT USER’S OWN RISK.

Resale of ST products with provisions different from the statements and/or technical features set forth in this document shall immediately void any warranty granted by ST for the ST product or service described herein and shall not create or extend in any manner whatsoever, any liability of ST.

ST and the ST logo are trademarks or registered trademarks of ST in various countries.

Information in this document supersedes and replaces all information previously supplied.

STMicroelectronics group of companies

Australia - Belgium - Brazil - Canada - China - Czech Republic - Finland - France - Germany - Hong Kong - India - Israel - Italy - Japan - Malaysia - Malta - Morocco - Singapore - Spain - Sweden - Switzerland - United Kingdom - United States of America

www.st.com