Book E:
Enhanced PowerPC™ Architecture

Version 1.0

May 7, 2002

Third Edition (Dec 2001)

Address comments about this document to:

IBM Corporation
Department B5H / Building 667
3039 Cornwallis Road P.O. Box 12195
Research Triangle Park, NC 27709

Portions of the information in this document may have been published previously in the following related documents:
The PowerPC Architecture: A Specification for a New Family of RISC Processors, Second Edition (1994)
The IBM PowerPC Embedded Environment: Architectural Specifications for IBM PowerPC Embedded Controllers, Second Edition (1998)

Preface

This release represents the initial release of the Book E architecture specification.

Many thanks to those in Motorola and IBM who have reviewed this document and contributed so much to cleaning up after my carelessness.

The Editor

Table of Contents

Figures

Tables

Chapter 1 Introduction

1.1 Overview

This chapter describes computation models, compatibility with the PowerPC Architecture, document conventions, a processor overview, instruction formats, storage addressing, and instruction fetching.

1.2 Compatibility with the PowerPC Architecture

Book E provides binary compatibility for 32-bit PowerPC application programs. Binary compatibility is not necessarily provided for privileged PowerPC instructions.

1.3 32-bit Book E Implementations

While Book E is expressed as a 64-bit architecture, there remains a viable market for 32-bit processors where applications do not require extended addressing capabilities nor 64-bit integer processing, or their need for such capability does not outweigh the cost of a 64-bit processor. Appendix A, “Guidelines for 32-bit Book E”, on page 371 provides a set of guidelines for hardware developers to develop 32-bit implementations of 64-bit Book E. Likewise, a set of guidelines is also outlined for software developers. Application software written to these guidelines can be labelled 32-bit Book E applications and can expect to execute properly on all implementations of Book E, both 32-bit and 64-bit implementations.

32-bit Book E implementations will execute applications that adhere to the software guidelines for 32-bit Book E software outlined in Appendix A and are not

expected to properly execute 64-bit Book E applications, or any applications not adhering to these guidelines (i.e. 64-bit Book E applications).

1.4 Instruction Mnemonics and Operands

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

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

1.5 Document Conventions

1.5.1 Notes

The document employs several different forms of notes. Information contained in these notes are not considered part of the architecture proper, but do contain advice and strong recommendations for producing a Book E-compliant system.

Architecture Note
Used to convey the direction of the architecture definition with respect to a particular function or feature.

Programming Note
Used to convey recommendations and suggestions to software developers on how a particular function or feature should be used in an application or operating system.

Engineering Note
Used to convey information on implementation options or how a particular feature might be supported. While the primary audience is hardware developers, software developers should benefit as well.

Compiler Note
Used to convey information to compiler developers how best to support or denigrate a particular feature that is either being added to, is currently a part of, or is being evicted from Book E.

Compatibility Note
Used to convey information on compatibility with the PowerPC Architecture.

Note
Used to convey information on generic, miscellaneous issues.

1.5.2 Notation

The following definitions and notation are used throughout the Book E document.

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

  Underscores may be used between digits for clarity purposes.

- Bits in registers, instructions, and fields are specified as follows.
  - Bits are numbered, left to right, most-significant bit to least-significant bit, starting with bit 0.
  - Ranges of bits are specified by two numbers separated by a colon (:). The range p:q consists of bits p through q.

- X_p means bit p of register/field X.

- $X_{p:q}$ means bits p through q of register/field X.

- $X_{p\ q\ ...}$ means bits p, q, ... of register/field X.

- $\neg X$ means the one's complement of the contents of register/field X.

- Field i refers to bits $4\times i$ through $4\times i+3$ of a register.

- A period (.) as the last character of an instruction mnemonic means that the instruction records status information in certain fields of the Condition Register as a side effect of execution, as described in Chapter 3 through Chapter 5.

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

- x^n means x raised to the n^{th} power.

- ${}^nx$ means the replication of x, n times (i.e., x concatenated to itself n–1 times). ${}^n0$ and ${}^n1$ are special cases:
  - ${}^n0$ means a field of n bits with each bit equal to 0. Thus ^{50} is equivalent to 0b00000.
  - ${}^n1$ means a field of n bits with each bit equal to 1. Thus ^{51} is equivalent to 0b11111.

- /, //, ///, ... denotes a reserved[1] field in an instruction or in a register.

- ?, ???, ... denotes an allocated field in an instruction.

- A shaded field denotes a field that is reserved or allocated in an instruction or in a register.

1. Each bit and field in instructions, and in status and control registers (e.g. Integer Exception Register and Floating-Point Status and Control Register) and other Special Purpose Registers, is either defined, allocated, or reserved. See Sections 1.5.4, 1.5.5, and 1.5.6.

1.5.3 Definitions

The following definitions are used for Book E.

aligned storage access
A load or store is aligned if the address of the target storage location is a multiple of the size of the transfer effected by the instruction.

block
The aligned unit of storage operated on by each Cache Management instruction. The size of a *block* can vary by instruction and by implementation. The maximum *block* size is one page.

boundedly undefined
If the results of executing a given instruction could have been achieved by executing an arbitrary sequence of instructions, starting in the state the machine was in before executing the given instruction. Boundedly undefined results for a given instruction may vary between implementations, and between different executions on the same implementation, and are not further defined in this document.

byte
A 8-bit element of storage.

context of a program
The environment (e.g., privilege and relocation) in which the program executes. That context is controlled by the contents of certain system registers, such as the Machine State Register, and of the address translation tables.

data storage
The view of storage as seen by a *Storage Access* or *Cache Management* instruction.

doubleword
A 64-bit element of storage.

exception
An error, unusual condition, or external signal that may set a status bit and may or may not cause an interrupt, depending upon whether the corresponding interrupt is enabled.

halfword
A 16-bit element of storage.

hardware
Any combination of hard-wired implementation, emulation assist, or interrupt for software assistance. In the last case, the interrupt may be to an architected location or to an implementation-dependent location. Any use of emulation assists or interrupts to implement the architecture is described in User's Manual.

instruction completion
The point in time when the instruction causes no further effect on processor state, when all results have been recorded in architected state.

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

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

– *Trap* instructions for which the trap conditions are satisfied cause a Trap exception type Program interrupt to be taken.

– *System Call* instructions cause a System Call interrupt to be taken.

– Exceptions can cause interrupts to be taken, as described in Chapter 7 on page 143.

– Returning from an interrupt handler causes execution to continue at a specified address.

The model of program execution in which each instruction appears to complete before the next instruction starts is called the 'sequential execution model'. In general, from the view of the processor executing the instructions, the sequential execution model is obeyed. From the perspective of user mode, for the instructions and facilities defined in Book E, the only exceptions to this rule are the following.

– A floating-point exception occurs when the processor is running in one of the Imprecise floating-point exception modes (see Section 5.4 on page 81). The instruction that causes the exception does not complete before the next instruction starts, with respect to setting exception bits and (if the exception is enabled) invoking an Enabled exception type Program interrupt.

– A *Store* instruction modifies a storage location that contains an instruction. Software synchronization is required to ensure that subsequent instruction fetches from that location obtain the modified version of the instruction: see Section 6.3.2 on page 139.

> **Programming Note**
> If a program modifies the instructions it intends to execute, it should execute the sequence of instructions listed in Section 6.3.2 on page 139 before attempting to execute the modified instructions, to ensure that the modifications have taken effect with respect to instruction fetching.

instruction storage
The view of storage as seen by the mechanism that fetches instructions.

interrupt
The act of changing the machine state in response to an exception, as described in Section 7 on page 143.

interrupt handler
A component of the system software that receives control when an interrupt occurs. The interrupt handler includes a component for each of the various kinds of interrupts. These interrupt-specific components are referred to as the Alignment interrupt handler, the Data Storage interrupt handler, etc.

latency
Refers to the interval from the time an instruction begins execution until it produces a result that is available for use by a subsequent instruction.

main storage
The level of the storage hierarchy in which all storage state is visible to all processors and mechanisms in the system.

multiprocessor
A system that contains two or more Book E processors.

negative
Means less than zero.

page
A "power of 2"-aligned unit of storage for which protection and control attributes are independently specifiable and for which reference and change status are independently recorded.

performed
A load or instruction fetch by a processor or mechanism (P1) is *performed* with respect to any processor or mechanism (P2) when the value to be returned by the load or instruction fetch can no longer be changed by a store by P2. A store by P1 is *performed* with respect to P2 when a load by P2 from the location accessed by the store will return the value stored (or a value stored subsequently). An instruction cache block invalidation by P1 is *performed* with respect to P2 when an instruction fetch by P2 will not be satisfied from the copy of the block that existed in its instruction cache when the instruction causing the invalidation was executed, and similarly for a data cache block invalidation. The preceding definitions apply regardless of whether P1 and P2 are the same entity.

positive
Means greater than zero.

processor
A hardware component that executes Book E instructions specified in a program.

program
A sequence of related instructions.

program order
The execution of instructions in the order required by the sequential execution model (see below).

quadword
A 128-bit element of storage.

real page
A unit of real storage to which a virtual page is or could be mapped.

sequential execution model
The model of program execution described in 'instruction fetching' on page 4.

Additional exceptions to the rule that the processor obeys the *sequential execution model*, beyond those described in 'instruction fetching', are the following.

– A System Reset or Machine Check interrupt may occur. The determination of whether an instruction is required by the *sequential execution model* is not affected by the potential occurrence of a System Reset or Machine Check interrupt. (The determination is affected by the potential occurrence of any other kind of interrupt.)

– A context-altering instruction is executed (see Chapter 11 on page 225). The context alteration need not take effect until the required subsequent

synchronizing operation has occurred.

> **Engineering Note**
> Although External and imprecise interrupts must be considered in determining whether an instruction is required by the sequential execution model, the fact that these interrupts are not required to be recognized at any specific point in the instruction stream allows an implementation to halt instruction dispatching and delay recognition of the interrupt until the processor comes into a state consistent with the sequential execution model. Such an implementation need not consider these interrupts in determining whether an instruction is required by the sequential execution model.
>
> Instruction-caused precise interrupts must also be considered in determining whether an instruction is required by the sequential execution model. However, for these it is always possible to predict whether they might be caused by any given instruction and thus to determine whether subsequent instructions are sure to be required by the sequential execution model.

shared storage multiprocessor
A multiprocessor that contains some common storage, which all of the Book E processors in the system can access.

storage access
An access to a storage location caused by executing a *Storage Access* or *Cache Management* instruction ('data access') or by fetching an instruction, or an implicit access that occurs as a side effect of such an access (e.g., to translate the effective address).

storage location
One or more sequential bytes of storage beginning at the address specified by a *Storage Access* or *Cache Management* instruction or by the instruction fetching mechanism. The number of bytes comprising the location is based on the type of instruction being executed, or is four for instruction fetching.

system
A combination of processors, storage, and associated mechanisms that is capable of executing programs. Sometimes the reference to *system* includes services provided by the operating system.

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

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

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

trap interrupt
An interrupt that results from execution of a *Trap* instruction

unavailable
Refers to a resource that cannot be used by the program. Storage is *unavailable* if access to it is denied. Floating-point instructions are unavailable if use of them is denied. See Section 7.6.8 on page 165.

uniprocessor
A system that contains one Book E processor.

word
A 32-bit element of storage.

1.5.4 Reserved Fields

All reserved fields in instructions should be zero. If they are not, the instruction form is invalid: see Section 1.9.2, "Invalid Instruction Forms", on page 28.

The handling of reserved bits in System Registers (e.g. Integer Exception Register, Floating-Point Status and Control Register) is implementation-dependent. Software is permitted to write any value to such a bit with no visible effect on processors that implement this version of Book E. A subsequent reading of the bit returns a 0 if the last value written to the bit was 0 and returns an undefined value (0 or 1) otherwise.

Engineering Note
Reserved bits in System Registers need not be implemented.

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

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

1. Initialize each such register supplying zeros for all reserved bits.

2. Alter (defined) bit(s) in the register by reading the register, altering only the desired bit(s), and then writing the new value back to the register.

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

When a currently reserved bit is subsequently assigned a meaning, every effort will be made to have the value to which the system initializes the bit correspond to the 'old behavior'.

Certain System Registers are defined as 32-bit registers, with their bits numbered 32:63. These 32-bit registers, with the exception of the Floating-Point Status and Control Register and its unique behavior on *Move From FPSCR* instructions (see Section 5.6.7 on page 104), can be treated as 64-bit registers with the upper 32 bits being reserved. However, Book E guarantees that the upper 32 bits of these registers will remain reserved.

1.5.5 Preserved Fields

Preserved bits in System Registers are bits that were defined in the PowerPC Architecture, are not defined in Book E, but are preserved to allow implementations of Book E to support the legacy definition for software compatibility.

The handling of preserved bits in System Registers is implementation-dependent. While software is permitted to write any value to such a bit, the effect of writing a 1 to a preserved bit is implementation-dependent. Writing a 1 to a preserved bit either has no effect or causes an effect that adheres to the PowerPC Architecture definition of the bit. A subsequent reading of the bit returns an implementation-dependent value.

Engineering Note
Preserved bits in System Registers need not be implemented.

Programming Note
Software has the responsibility of maintaining the contents of preserved bits in System Registers. Preserved bits may be assigned a meaning in some future version of Book E.

In order to maintain the contents of preserved bits in an implementation-independent fashion, software should do either or both of the following.

1. Initialize each such register supplying zeros for all preserved bits.

2. Alter (defined) bit(s) in the register by reading the register, altering only the desired bit(s), and then writing the new value back to the register.

1.5.6 Allocated Fields

Allocated bits in System Registers are bits provided for implementation-dependent use. The effect of setting an allocated bit to a value other than 0 is implementation-dependent. Allocated bits return an implementation-dependent value when read.

Engineering Note
Allocated bits in System Registers need not be implemented.

Architecture Note
Allocated bits are provided to support implementation-dependent extensions to the Book E.

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

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

1. Initialize each such register supplying zeros for all allocated bits.

2. Alter (defined) bit(s) in the register by reading the register, altering only the desired bit(s), and then writing the new value back to the register.

1.5.7 Description of Instruction Operation

A formal description is given of the operation of each instruction. In addition, the operation of most instructions is described by a series of statements using a semi-formal language at the register transfer level (RTL). This RTL uses the notation given below, in addition to the definitions and notation described in Section 1.5.1 and Section 1.5.3. Some of this notation is also used in the formal descriptions of instructions. RTL notation not summarized here should be self-explanatory.

The RTL descriptions cover the normal execution of the instruction, except that "implicit" setting of the Condition Register, Integer Exception Register, and Floating-Point Status and Control Register, such as to reflect the final status of the execution of the instruction, is not always shown. (Explicit 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 of the cases in which the interrupt may be invoked, or for which the results are boundedly undefined, and may not cover all invalid forms.

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

Notation	Meaning
$\leftarrow$	Assignment
$\leftarrow_f$	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
$\times$	Multiplication
$\div$	Division (yielding quotient)
$+_{dp}$	Floating-point addition, result rounded to double-precision
$-_{dp}$	Floating-point subtraction, result rounded to double-precision
$\times_{dp}$	Floating-point multiplication, product rounded to double-precision
$\div_{dp}$	Floating-point division, quotient rounded to double-precision
$+_{sp}$	Floating-point addition, result rounded to single-precision
$-_{sp}$	Floating-point subtraction, result rounded to single-precision
$\times_{sp}$	Floating-point multiplication, product rounded to single-precision
$\div_{sp}$	Floating-point division, quotient rounded to single-precision
$\times_{fp}$	Floating-point multiplication to 'infinite' precision (no rounding)
FPSquareRoot-Double(x)	Floating-point $\sqrt{x}$, result rounded to double-precision
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 storage 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.

Notation	Meaning
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 storage 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 storage 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 storage 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 set to 0.
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 storage 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))
CEIL(x)	Least integer ≥ x
DCREG(x)	Device Control Register x
DOUBLE(x)	Result of converting x from floating-point single format to floating-point double format, using the model shown on page 98.
EXTS(x)	Result of extending x on the left with sign bits
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 storage located at address x.
MEM(x,y) (for y={2,4,8})	Contents of y bytes of storage starting at address x. If big-endian storage (see Section 6.2.5.5 on page 136), the byte at address x is the most-significant byte and the byte at address x+y-1 is the least-significant byte of the value being accessed. If little-endian storage (see Section 6.2.5.5 on page 136), the byte at address x is the least-significant byte and the byte at address x+y–1 is the most-significant byte of the value being accessed.
MOD(x,y)	Modulo y of x (remainder of x divided by y).
$ROTL_{64}$(x, y)	Result of rotating the 64-bit value x left y positions
$ROTL_{32}$(x, y)	Result of rotating the 64-bit value x||x left y positions, where x is 32 bits long
SINGLE(x)	Result of converting x from floating-point double format to floating-point single format, using the model shown on page 100.
SPREG(x)	Special Purpose Register x
TRAP	Invoke a Trap type Program interrupt
undefined	An undefined value. The value may vary between implementations, and between different executions on the same implementation.
CIA	Current Instruction Address, which is the 64-bit address of the instruction being described by a sequence of RTL. Used by relative branches to set the Next Instruction Address (NIA), and by *Branch* instructions with LK=1 to set the Link Register. CIA does not correspond to any architected register.

Notation	Meaning
NIA	Next Instruction Address, which is the 64-bit address of the next instruction to be executed. For a successful branch, the next instruction address is the branch target address: in RTL, this is indicated by assigning a value to NIA. For other instructions that cause non-sequential instruction fetching (see Section 2.2.1 on page 43), 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
do	Do loop, indenting shows range. 'To' and/or 'by' clauses specify incrementing an iteration variable, and a 'while' clause gives termination conditions.
leave	Leave innermost do loop, or do loop described in leave statement

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

Table 1-1. Operator precedence

Operators	Associativity
subscript, function evaluation	left to right
pre-superscript (replication), post-superscript (exponentiation)	right to left
unary –, ¬	right to left
×, ÷	left to right
+, –	left to right
||	left to right
=, ≠, <, ≤, >, ≥, $<_u$, $>_u$, ?	left to right
&, ⊕, ≡	left to right
|	left to right
: (range)	none
←	none

1.6 Book E Overview

The architecture defines the instruction set, the storage model, interrupt action, and other facilities. Instructions that the processor can execute fall into several classes:

Instruction Class	Section	Page
system linkage instructions	2.2.1	43
processor control register manipulation instructions	2.2.2	43
branch instructions	3.3	49
CR instructions	3.4	52
integer instructions	4.3	55
floating-point instructions (including FPSCR manipulation)	5.6	98
storage (i.e. synchronization, cache and TLB) control instructions	6.3	139
implementation-dependent instructions	See User's Manual	—i

Integer instructions operate on byte, halfword, word, and, in 64-bit implementations, doubleword operands. Floating-point instructions operate on single-precision and double-precision floating-point operands. Book E uses instructions that are four bytes long and word-aligned. It provides for byte, halfword, word, and, in 64-bit implementations, doubleword operand loads and stores between storage and a set of 32 General Purpose Registers (GPRs). It also provides for word and doubleword operand loads and stores between storage and a set of 32 Floating-Point Registers (FPRs).

Signed integers are represented in two's complement form.

There are no computational instructions that reference storage, except *Load Half Algebraic*, *Load Floating-Point Single*, *Load Floating-Point Double*, and *Store Floating-Point Single*. Normally, to use a storage operand in a computation and then modify the same or another storage location, the contents of storage must be loaded into a register, modified, and then stored back to the target location. Figure 1-3 shows the user-mode registers of Book E. Figure 1-4 shows the significant supervisor-mode registers of Book E. Figure 1-6 shows the interrupt-specific registers of Book E. Figure 1-7 shows the storage control-specific register of Book E. Figure 1-8 shows the timer-specific registers of Book E. Figure 1-9 shows the debug-specific registers of Book E. Note that bits for 32-bit registers are numbered 32:63 rather than 0:31 to indicate their true bit alignment with respect to 64-bit registers. 32-bit registers can be correctly interpreted as 64-bit registers with bits 0:31 permanently reserved.

Figure 1-1. Book E user-mode base register set

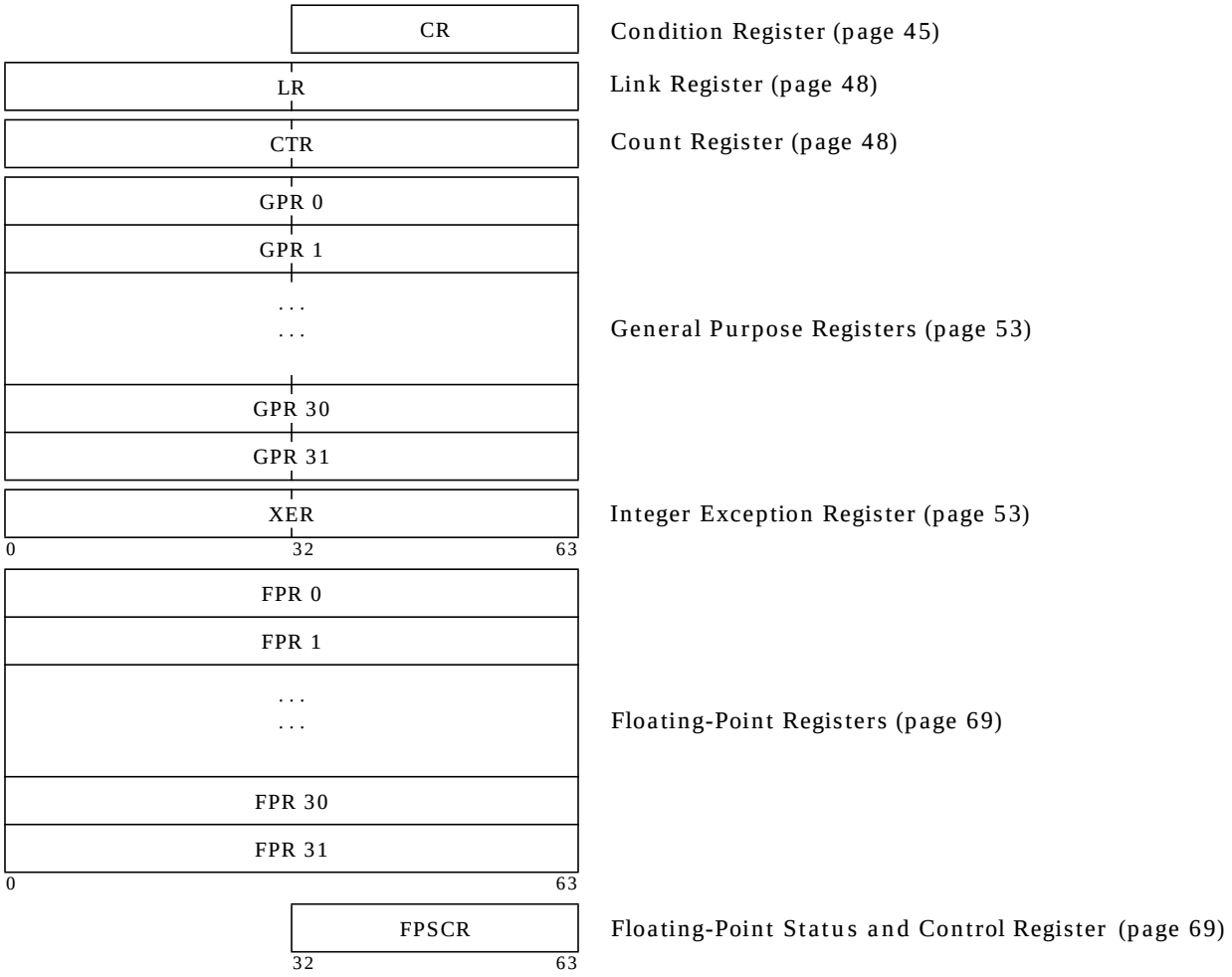
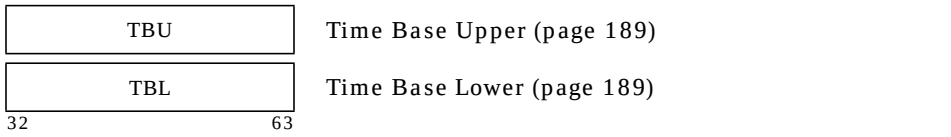


Figure 1-2. Book E user-mode timer facilities register set[1]



1. TBH, and TBL are user-mode read-access only.

Figure 1-3. Book E user-mode software-use register set[1]

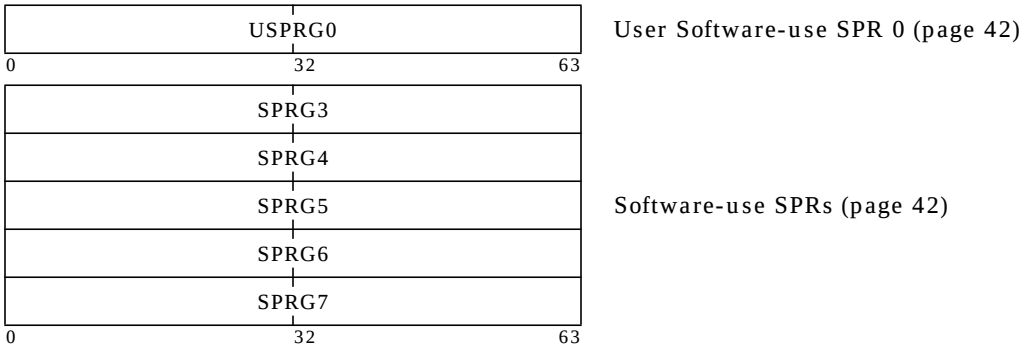


Figure 1-4. Book E supervisor-mode base register set

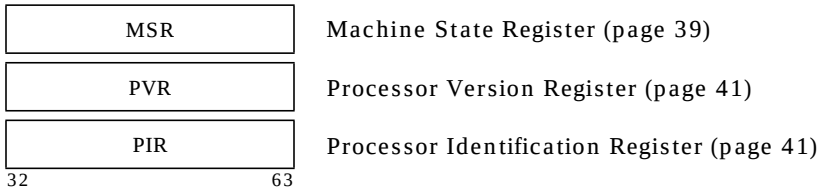
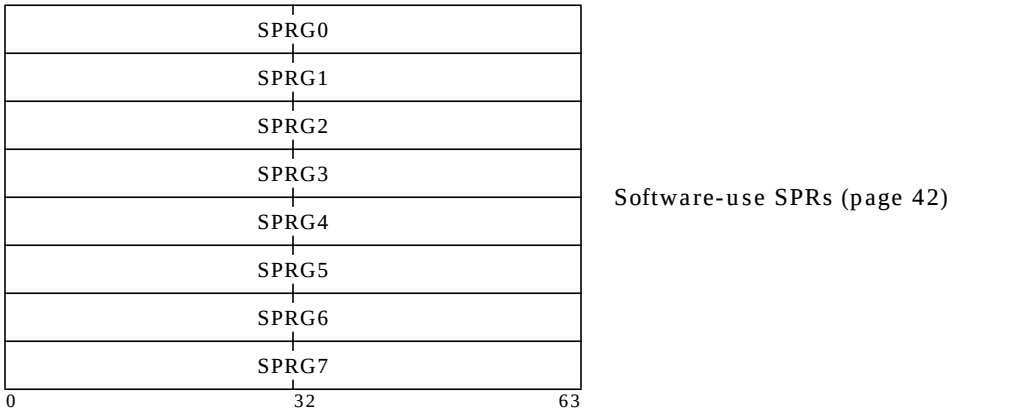


Figure 1-5. Book E supervisor-mode software-use register set



1. SPRG3 user-mode accessibility is implementation-dependent. SPRG4, SPRG5, SPRG6, and SPRG7 are user-mode read-access only.

Figure 1-6. Book E supervisor-mode interrupt register set

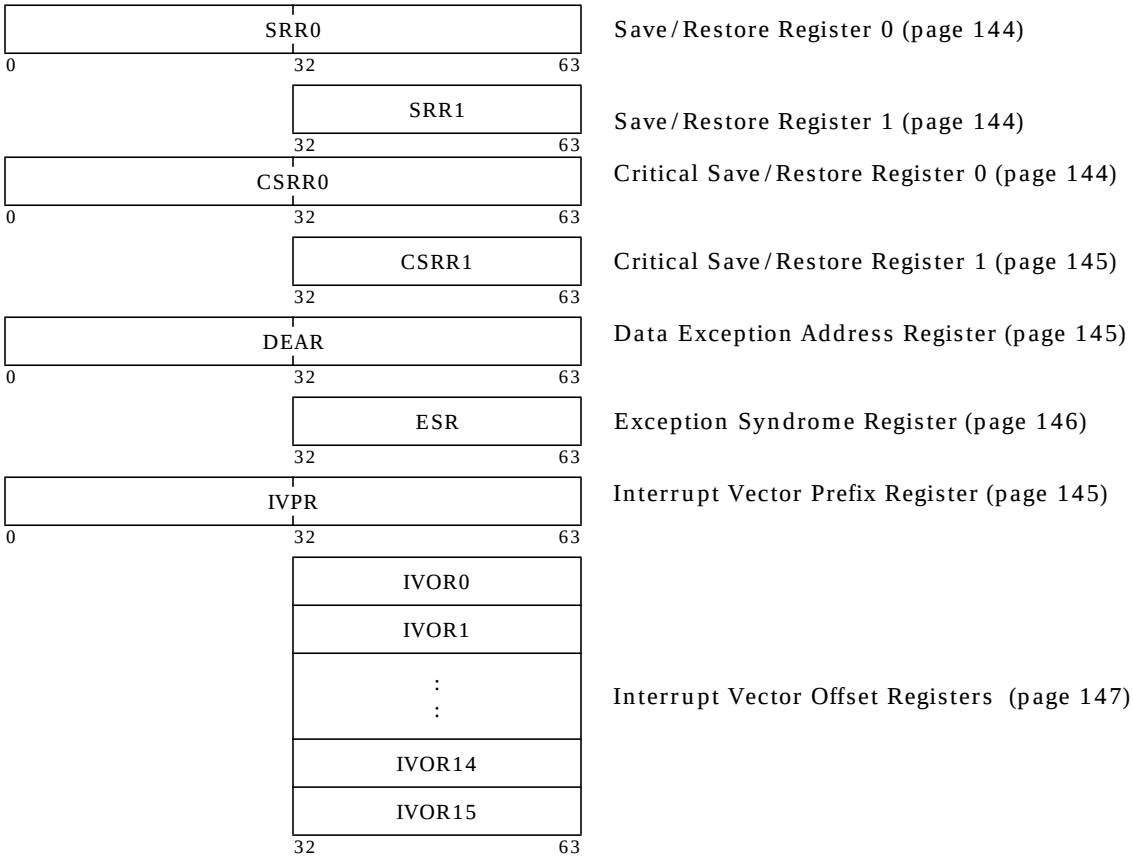


Figure 1-7. Book E supervisor-mode storage control register set

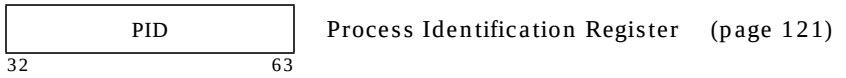


Figure 1-8. Book E supervisor-mode timer facilities register set

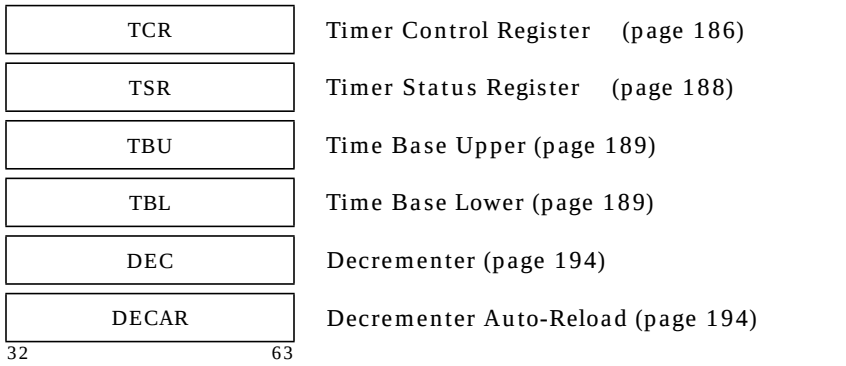
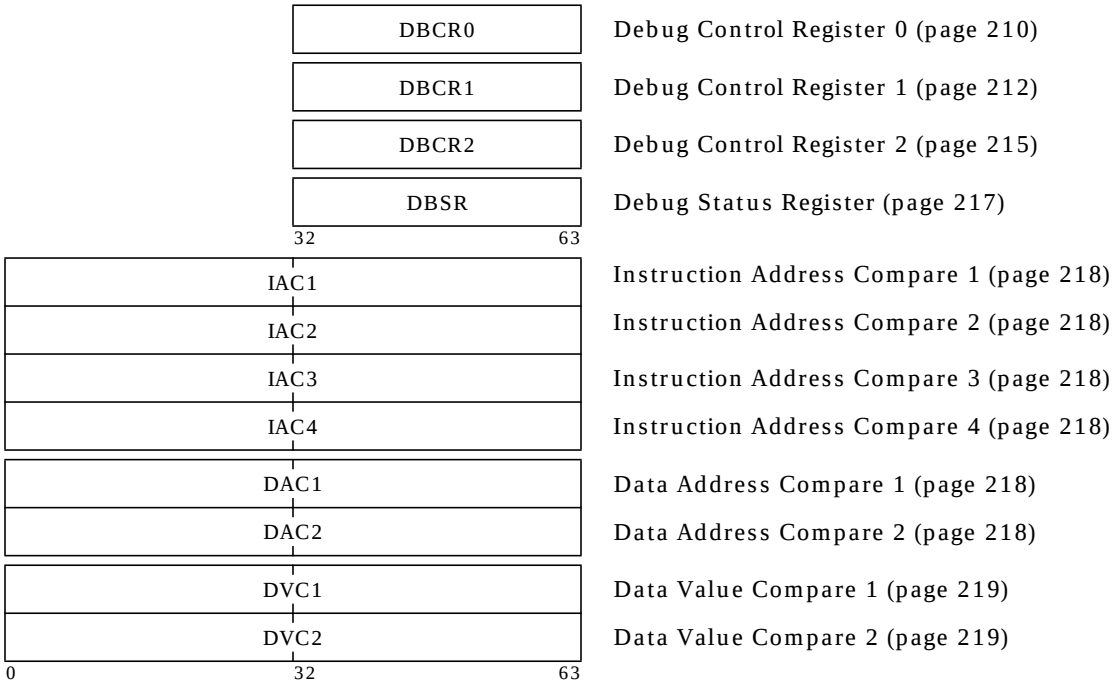


Figure 1-9. Book E debug facilities register set



1.7 Instruction Formats

All instructions to be executed are four bytes long and word-aligned in storage. Thus, whenever instruction addresses are presented to the processor (as in *Branch* and *Branch Extended* instructions) the two low-order bits are treated as 0s. Similarly, whenever the processor develops an instruction address its two low-order bits are zero.

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

The format diagrams given below show horizontally all valid combinations of instruction fields.

In some cases an instruction field is reserved, or must contain a particular value. If a reserved field does not have all bits set to 0, or if a field that must contain a particular value does not contain that value, the instruction form is invalid and the results are as described in Section 1.9.2, "Invalid Instruction Forms", on page 28.

Split Field Notation

In some cases an instruction field occupies more than one contiguous sequence of bits, or occupies one contiguous sequence of bits that are used in permuted order. Such a field is called a *split field*. In the format diagrams given below and in the individual instruction layouts, the name of a split field is shown in lower-case letters, once for each of the contiguous sequences, each with their respective bit numbering. In the RTL description of an instruction having a split field, and in certain other places where individual bits of a split field are identified, the name of the field in upper-case letters represents the bit-ordered concatenation of the sequences.

Figure 1-10. A instruction format

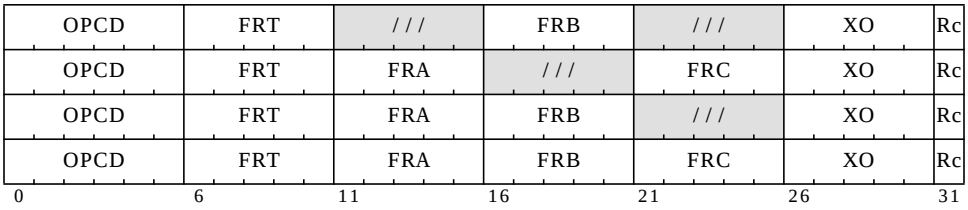


Figure 1-11. B instruction format

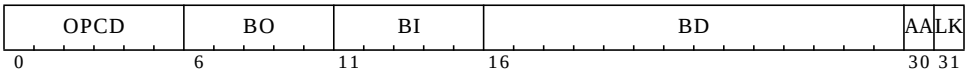


Figure 1-12. D instruction format

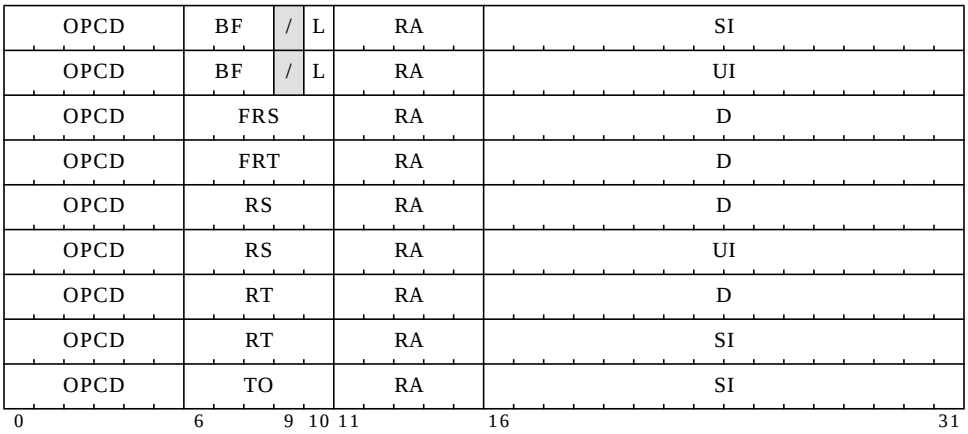


Figure 1-13. DE instruction format

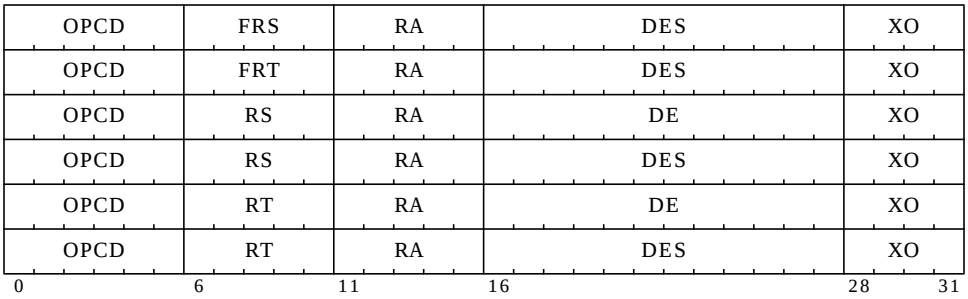


Figure 1-14. I instruction format

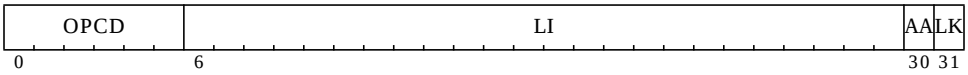


Figure 1-15. M instruction format

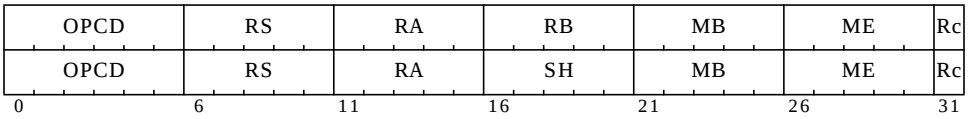


Figure 1-16. MD instruction format

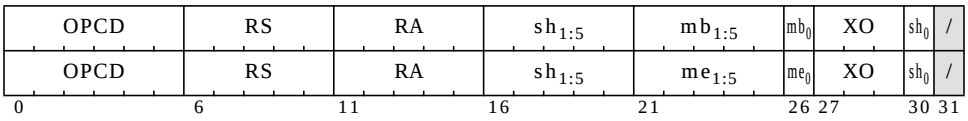


Figure 1-17. MDS instruction format

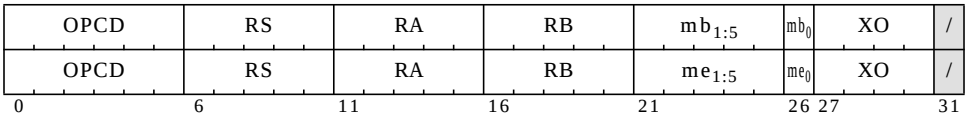


Figure 1-18. X instruction format

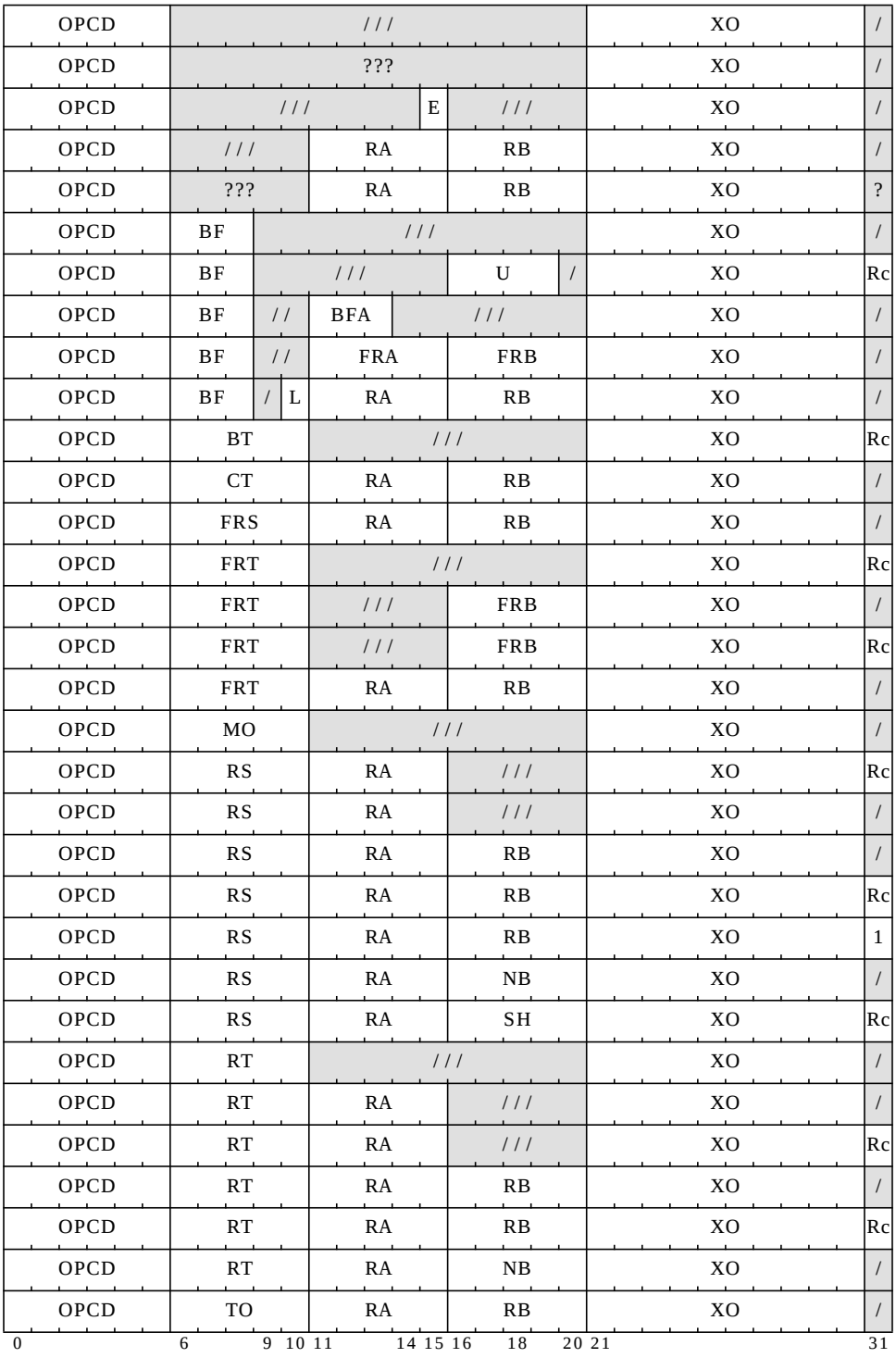


Figure 1-19. SC instruction format

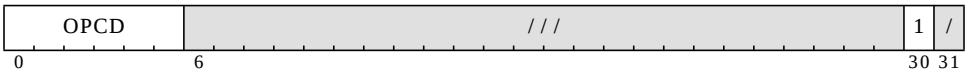


Figure 1-20. XFL instruction format

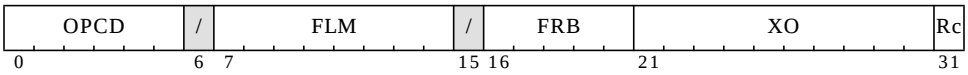


Figure 1-21. XFX instruction format

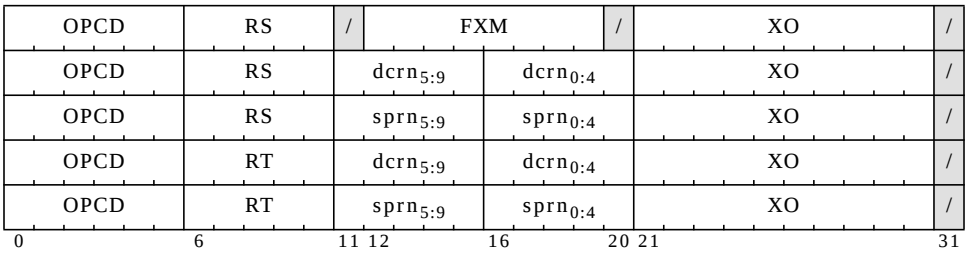


Figure 1-22. XL instruction format

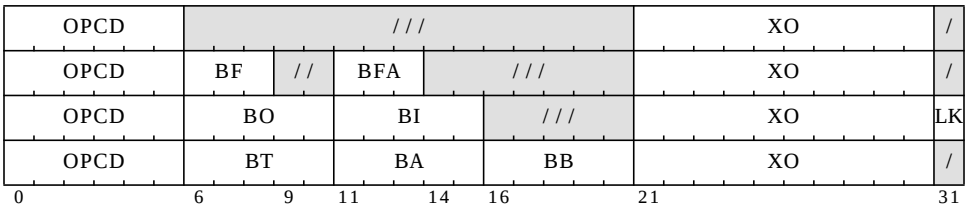
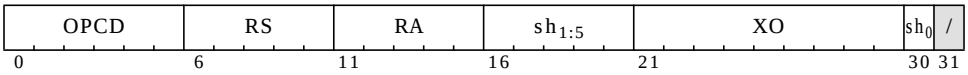


Figure 1-23. XS instruction format



1.7.1 Instruction Fields

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 sum $^{32}0 \,||\, (\text{CIA}+\text{EXTS}(\text{LI}||0b00))_{32:63}$.

– For B-form *Branch* instructions the effective address of the branch target is the sum $^{32}0 \,||\, (\text{CIA}+\text{EXTS}(\text{BD}||0b00))_{32:63}$.

– For I-form *Branch Extended* instructions the effective address of the branch target is the sum CIA+EXTS(LI||0b00).

– For B-form *Branch Extended* instructions the effective address of the branch target is the sum 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 $^{32}0 \,||\, \text{EXTS}(\text{LI}||0b00)_{32:63}$.

– For B-form *Branch* instructions the effective address of the branch target is the value $^{32}0 \,||\, \text{EXTS}(\text{BD}||0b00)_{32:63}$.

– 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).

BA (11:15)
Field used to specify a bit in the Condition Register to be used as a source.

BB (16:20)
Field used to specify a bit in the Condition Register to be used as a source.

BD (16:29)
Immediate field specifying a 14-bit signed two's complement branch displacement which is concatenated on the right with 0b00 and sign-extended to 64 bits.

BF (6:8)
Field used to specify one of the Condition Register fields or one of the Floating-Point Status and Control Register fields to be used as a target.

BFA (11:13)
Field used to specify one of the Condition Register fields or one of the Floating-Point Status and Control Register fields to be used as a source.

BI (11:15)
Field used to specify a bit in the Condition Register to be used as the condition of a *Branch Conditional* instruction.

BO (6:10)
Field used to specify options for the *Branch Conditional* instructions. The encoding is described in Section 3.3 on page 49.

BT (6:10)
Field used to specify a bit in the Condition Register or in the Floating-Point Status and Control Register to be used as a target.

CT (6:10)
Field used by the *Cache Touch* instructions (***dcbt***[***e***], ***dcbtst***[***e***], and ***icbt***[***e***]) 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 which is sign-extended to 64 bits.

dcrn(16:20||11:15)
Field used to specify a Device Control Register for the ***mtdcr*** and ***mfdcr*** instructions.

DE (16:27)
Immediate field used to specify a 12-bit signed two's complement integer which is sign-extended to 64 bits.

DES (16:27)
Immediate field used to specify a 12-bit signed two's complement integer which is concatenated on the right with 0b00 and sign-extended to 64 bits.

E (15)
Immediate field used to specify a 1-bit value used by ***wrteei*** to place in the EE (External Input Enable) bit of the Machine State Register.

FLM (7:14)
Field mask used to identify the Floating-Point Status and Control Register fields that are to be updated by the ***mtfsf*** instruction.

FRA (11:15)
Field used to specify a Floating-Point Register to be used as a source.

FRB (16:20)
Field used to specify a Floating-Point Register to be used as a source.

FRC (21:25)
Field used to specify a Floating-Point Register to be used as a source.

FRS (6:10)
Field used to specify a Floating-Point Register to be used as a source.

FRT (6:10)
Field used to specify a Floating-Point Register to be used as a target.

FXM (12:19)
Field mask used to identify the Condition Register fields that are to be updated by the ***mtcrf*** instruction.

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

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

LK (31)
LINK bit.

0 Do not set the Link Register.
1 Set the Link Register. The sum of the value 4 and the address of the *Branch* instruction is placed into the Link Register.

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, as described in Section 4.3.7 on page 63.

mb (26 || 21:25)
Field used in MD-form and MDS-form *Rotate* instructions to specify the first 1-bit of a 64-bit mask, as described in Section 4.3.7 on page 63.

me (26 || 21:25)
Field used in MD-form and MDS-form *Rotate* instructions to specify the last 1-bit of a 64-bit mask, as described in Section 4.3.7 on page 63.

MO (6:10)
Field used to specify the subset of storage accesses that are ordered by the *Memory Barrier* instruction.

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

OPCD (0:5)
Primary opcode field.

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

RB (16:20)
Field used to specify a General Purpose Register to be used as a source.

Rc (31)
RECORD bit.

0 Do not alter the Condition Register.
1 Set Condition Register Field 0 or Field 1 as described in Section 3.2.1 on page 45.

RS (6:10)
Field used to specify a General Purpose Register to be used as a source.

RT (6:10)
Field used to specify a General Purpose Register to be used as a target.

SH (16:20)
Field used to specify a shift amount in *Rotate Word Immediate* and *Shift Word Immediate* instructions.

sh (30 || 16:20)
Field used to specify a shift amount in *Rotate Doubleword Immediate* and *Shift Doubleword Immediate* instructions.

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

sprn (16:20||11:15)
Field used to specify a Special Purpose Register for the ***mtspr*** and ***mfspr*** instructions.

TO (6:10)
Field used to specify the conditions on which to trap. The encoding is described in Section 4.3.6 on page 62.

U (16:19)
Immediate field used as the data to be placed into a field in the Floating-Point Status and Control Register.

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

WS (18:20)
Field used to specify a word in the Translation Lookaside Buffer entry being accessed.

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

1.8 Classes of Instructions

An instruction falls into exactly one of the following four classes, which is determined by examining the primary opcode, and the extended opcode, if any.

1. Defined instructions (see Appendix H on page 419)
2. Allocated instructions (Section F.2 on page 404)
3. Preserved instructions (Section F.1 on page 403)
4. Reserved (-illegal or -nop) instructions (Section F.3 on page 404)

1.8.1 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.

One exception to this is that, for implementations which only provide the 32-bit subset of Book E, it is not expected (and likely not even possible) that emulation of the 64-bit behavior of the defined instructions will be provided by the system. See Appendix A, “Guidelines for 32-bit Book E”, on page 371.

Any attempt to execute a defined instruction will:

- cause an Illegal Instruction exception type Program interrupt, if the instruction is not recognized by the implementation; or

- cause an Unimplemented Instruction exception type Program interrupt, if the instruction is recognized but not supported by the implementation, and isn't a floating-point instruction; or

- cause 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; or

- cause a Floating-Point Unavailable interrupt if the instruction is recognized but is not supported by the implementation, is a floating-point instruction, and floating-point processing is disabled; or

- perform the actions described in the rest of this document, if the instruction is recognized and supported by the implementation. 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 re-classified as a preserved instruction (process of removal from the architecture) and eventually classified as reserved-illegal.

1.8.2 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) listed in Section F.2 on page 404.

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 will:

- cause an Illegal Instruction exception type Program interrupt, if the instruction is not recognized by the implementation; or

- cause an Unimplemented Instruction exception type Program interrupt, if the instruction is recognized and enabled for execution, but direct execution of the instruction is not supported by the implementation; or

- cause a Floating-Point Unavailable interrupt if the instruction is recognized but is not supported by the implementation, is a floating-point instruction, and floating-point processing is disabled; or

- perform the actions described in the User's Manual for the implementation. The 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. 'Enabled for execution' for these instructions implies any associated enable bit is set to allow, or enable, execution of these instructions. For these allocated instructions, the architecture provides an Auxiliary Processor Unavailable interrupt vector (see Section 7.6.10 on page 166) in the event execution of any of these instructions is attempted when not 'enabled for execution'.

Other allocated instructions may not have any associated new state and therefore may not require an associated enable bit. These instructions are assumed to always be 'enabled for execution' if they are supported by the implementation.

1.8.3 Preserved Instruction Class

Preserved instructions are provided to support backward compatibility with the PowerPC Architecture.

Any attempt to execute a preserved instruction will:

- perform the actions described in the previous version of this architecture, if the instruction is recognized and supported by the implementation; or
- cause an Illegal Instruction exception type Program interrupt, if the instruction is not recognized by the implementation.

A preserved instruction may be retained by future versions of Book E as a preserved instruction, may be subsequently re-classified as an allocated instruction, or may even be adopted into Book E. A preserved instruction (in the process of removal from the architecture) may also eventually be classified as reserved-illegal.

1.8.4 Reserved Instruction Class

This class of instructions consists of all instruction primary opcodes (and associated extended opcodes, if applicable) which 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.

Attempt to execute a reserved-illegal instruction will cause an Illegal Instruction exception type Program interrupt (see Section 7.6.7 on page 163) on implementations conforming to the current version of Book E. Reserved-illegal instructions are, therefore, available for future extensions to Book E which would affect architected state. Such extensions might include new forms of integer or floating-point arithmetic or new forms of load or store instructions which write their result in an architected register.

Attempt to execute a reserved-nop instruction will either have no effect on implementations conforming to the current version of Book E (i.e. treated as a no-operation instruction), or will cause either an Illegal Instruction exception type Program interrupt (see Section 7.6.7 on page 163). Reserved-nop instructions are available for future architecture extensions which have no effect on architected state. Such extensions might include performance-enhancing hints such as new forms of *Cache Touch* instructions, and would be able to be added while remaining functionally compatible with implementations of previous versions of Book E.

Engineering Note
Implementations are strongly encouraged to support reserved-nop instructions as a true no-operation instruction.

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

1.9 Forms of Defined Instructions

1.9.1 Preferred Instruction Forms

There is one defined instruction that has a preferred form. The *Or Immediate* instruction is the preferred form for expressing a no-operation.

1.9.2 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.

Any attempt to execute an invalid form of an instruction will either cause an Illegal Instruction type Program interrupt or yield boundedly undefined results. 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.

- the *Branch Conditional* and *Branch Conditional Extended* instructions (i.e. undefined encoding of BO field)
- the *Load with Update* instructions (i.e. RT=RA or RA=0)
- the *Store with Update* instructions (i.e. RA=0)
- the *Load Multiple* instruction (i.e. RA or RB in range of registers to be loaded)
- the *Load String Immediate* instruction (i.e. RA in range of registers to be loaded)
- the *Load String Indexed* instruction (i.e. RT=RA or RT=RB)
- the *Load/Store Floating-Point with Update* instructions (i.e. RA=0)

Engineering Note
Causing an Illegal Instruction type Program interrupt if an attempt is made to execute an invalid form of an instruction facilitates the debugging of software.

One exception is that Book E strongly recommends that hardware ignore bit 31 of the instruction encoding for X-form *Storage Access* instructions rather than causing an Illegal Instruction type Program interrupt when this reserved bit is set to 1. This facilitates subsequent definition of bit 31 for performance enhancement extensions to the architecture while remaining functionally compatible with implementations of previous versions of Book E.

1.10 Optionality

An implementation is allowed to trap on the use of any instruction or architectural feature and emulate in software. There are certain architectural features that cannot and are not expected to be supported in this manner. A 32-bit Book E implementation cannot be expected to emulate the behavior of a 64-bit Book E implementation. A Book E implementation that supports only a subset of the Book E debug facilities (see Chapter 9 on page 199) may not be able to emulate the remainder of the defined debug facilities. Implementations of this architecture must be capable of supporting all other defined Book E facilities either in hardware or with software assistance.

1.11 Storage Addressing

A program references storage using the effective address computed by the processor when it executes a *Storage Access* or *Branch* instruction (or certain other instructions described in Section 6.3.2 on page 139, and Section 6.3.3 on page 142), or when it fetches the next sequential instruction.

1.11.1 Storage Operands

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

Storage operands may be bytes, halfwords, words, or doublewords, or, for the *Load Multiple*, *Store Multiple*, *Load String* and *Store String* instructions, a sequence of words or bytes. The address of a storage operand is the address of its first byte (i.e., of its lowest-numbered byte). Byte ordering can be either big-endian or little-endian (see Section 1.11.3 on page 33).

Operand length is implicit for each instruction with respect to storage alignment. The operand of a scalar *Storage 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 storage operand is said to be *aligned* if it is aligned at its natural boundary: otherwise it is said to be *unaligned*.

Storage operands for single-register *Storage Access* instructions have the following characteristics.

Operand	Operand Length	$Addr_{60:63}$ if aligned
Byte (or String)	8 bits	xxxx
Halfword	2 bytes	xxx0
Word	4 bytes	xx00
Doubleword	8 bytes	x000

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.

The concept of alignment is also applied more generally, to any datum in storage. For example, a 12-byte datum in storage is said to be word-aligned if its address is an integral multiple of 4.

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

Instructions are always four bytes long and word-aligned.

1.11.2 Effective Address Calculation

The 64-bit address computed by the processor when executing a *Storage Access* or *Branch* instruction (or certain other instructions described in Section 6.3.2 on page 139, and Section 6.3.3 on page 142), or when fetching the next sequential instruction, is called the *effective address* and specifies a byte in storage. For a *Storage Access* instruction, if the sum of the effective address and the operand length exceeds the maximum effective address, the storage access is considered to be undefined.

Effective address arithmetic, except for next sequential instruction address computations, wraps around from the maximum address, $2^{64}-1$, to address 0.

1.11.2.1 Data Storage Addressing Modes

There are five data storage addressing modes supported by Book E.

Base+Displacement (D-mode) addressing mode:
The 16-bit D field is sign-extended and added to the contents of the GPR designated by RA or to zero if RA=0. 32 zeros are prepended to the low-order 32 bits of the sum to produce the 64-bit effective address.

Base+Index (X-mode) addressing mode:
The contents of the GPR designated by RB (or the value 0 for ***lswi*** and ***stswi***) are added to the contents of the GPR designated by RA or to zero if RA=0. 32 zeros are prepended to the low-order 32 bits of the sum to produce the 64-bit effective address.

Base+Displacement Extended (DE-mode) addressing mode:
The 12-bit DE field is sign-extended and added to the contents of the GPR designated by RA or to zero if RA=0 to produce the 64-bit effective address.

Base+Displacement Extended Scaled (DES-mode) addressing mode:
The 12-bit DES field is concatenated on the right with 0b00, sign-extended, and then added to the contents of the GPR designated by RA or to zero if RA=0 to produce the 64-bit effective address.

Base+Index Extended (XE-mode) addressing mode:
The contents of the GPR designated by RB are added to the contents of the GPR designated by RA or to zero if RA=0 to produce the 64-bit effective address

1.11.2.2 Instruction Storage Addressing Modes

I-form *Branch Extended* instructions:

The 24-bit LI field is concatenated on the right with 0b00, sign-extended, and then added to either the 64-bit address of the *Branch Extended* instruction if AA=0, or to 0 if AA=1, to form the 64-bit effective address of the next instruction.

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. 32 zeros are prepended to the low-order 32 bits of the sum to form the 64-bit effective address of the next instruction.

Taken B-form *Branch Extended* 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 Extended* instruction if AA=0, or to 0 if AA=1, to form the 64-bit effective address of the next instruction.

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. 32 zeros are prepended to the low-order 32 bits of the sum to form the 64-bit effective address of the next instruction.

Taken XL-form *Branch Extended* instructions:

The contents of bits 0:61 the Link Register or the Count Register are concatenated on the right with 0b00 to form the 64-bit effective address of the next instruction.

Taken XL-form *Branch* instructions:

The contents of bits 32:61 of the Link Register or the Count Register are concatenated on the right with 0b00 and prepended with 32 zeros on the left to form the 64-bit effective address of the next instruction.

Sequential instruction fetching (or non-taken *Branch* or *Branch Extended* instructions):

The value 4 is added to the address of the current instruction to form the 64-bit effective address of the next instruction. If the address of the current instruction is 0xFFFF_FFFF_FFFF_FFFC, the address of the next sequential instruction is undefined.

Any *Branch* or *Branch Extended* instruction with LK=1:

The value 4 is added to the address of the current instruction and the 64-bit result is placed into the LR. If the address of the current instruction is 0xFFFF_FFFF_FFFF_FFFC, the result placed into the LR is undefined.

Programming Note

While some implementations may support next sequential instruction address computations wrapping from the highest address 0xFFFF_FFFF_FFFF_FFFC to 0x0000_0000_0000_0000 as part of the instruction flow, portable software is strongly encouraged not to depend on this. If code must span this boundary, software should place a non-linking branch at address 0xFFFF_FFFF_FFFF_FFFC which always branches to address 0x0000_0000_0000_0000 (either absolute or relative branches will work). See also Section A.1.3 on page 372.

1.11.3 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 storage, 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 storage does the question of order arise.

For a machine in which the smallest addressable unit of storage is the 64-bit doubleword, there is no question of the ordering of bytes within doublewords. All transfers of individual scalars between registers and storage are of doublewords, and the address of the byte containing the high-order eight 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 storage is the 8-bit byte. Many scalars are halfwords, words, or doublewords, which consist of groups of bytes. When a word-length scalar is moved from a register to storage, 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 eight 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 four 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’) eight 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 end” (most-significant) of the scalar, considered as a binary number, comes first in storage. The Motorola 68000 is an example of a processor using this byte ordering.

- The ordering that assigns the lowest address to the lowest-order (‘right-most’) eight 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 end” (least-significant) of the scalar, considered as a binary number, comes first in storage. The Intel 8086 is an example of a processor using this byte ordering.

Book E provides support for both big-endian and little-endian byte ordering in the form of a storage attribute. See Section 6.2.5 on page 132 and Section 6.2.5.5 on page 136.

1.11.3.1 Structure Mapping Examples

The following C language structure, *s*, contains an assortment of scalars and a character string. The comments show the value assumed to be in each structure element; these values show how the bytes comprising each structure element are mapped into storage.

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

C structure mapping rules permit the use of padding (skipped bytes) to align scalars on desirable boundaries. The structure mapping examples below show each scalar aligned at its natural boundary. This alignment introduces padding of four bytes between *a* and *b*, one byte between *d* and *e*, and two bytes between *e* and *f*. The same amount of padding is present in both big-endian and little-endian mappings.

Big-Endian Mapping

The big-endian mapping of structure *s* follows. (The data is in boldface print in the structure mappings. Addresses, in hexadecimal, are below the data stored at the address. The contents of each byte, as defined in structure *s*, is shown as a (hexadecimal) number or character (for the string elements).

11 0x00	**12** 0x01	**13** 0x02	**14** 0x03	0x04	0x05	0x06	0x07
21 0x08	**22** 0x09	**23** 0x0A	**24** 0x0B	**25** 0x0C	**26** 0x0D	**27** 0x0E	**28** 0x0F
31 0x10	**32** 0x11	**33** 0x12	**34** 0x13	**'A'** 0x14	**'B'** 0x15	**'C'** 0x16	**'D'** 0x17
'E' 0x18	**'F'** 0x19	**'G'** 0x1A	0x1B	**51** 0x1C	**52** 0x1D	0x1E	0x1F
61 0x20	**62** 0x21	**63** 0x22	**64** 0x23	0x24	0x25	0x26	0x27

Little-Endian Mapping

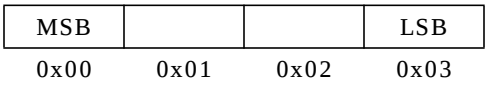
Structure *s* is shown mapped little-endian.

14 0x00	**13** 0x01	**12** 0x02	**11** 0x03	0x04	0x05	0x06	0x07
28 0x08	**27** 0x09	**26** 0x0A	**25** 0x0B	**24** 0x0C	**23** 0x0D	**22** 0x0E	**21** 0x0F
34 0x10	**33** 0x11	**32** 0x12	**31** 0x13	**'A'** 0x14	**'B'** 0x15	**'C'** 0x16	**'D'** 0x17
'E' 0x18	**'F'** 0x19	**'G'** 0x1A	0x1B	**52** 0x1C	**51** 0x1D	0x1E	0x1F
64 0x20	**63** 0x21	**62** 0x22	**61** 0x23	0x24	0x25	0x26	0x27

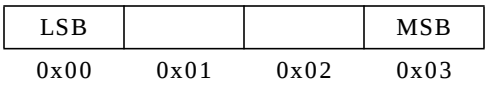
1.11.3.2 Instructions Byte Ordering

Book E defines instructions as aligned words (four bytes) in memory. As such, instructions in a big-endian program image are arranged with the most-significant byte (MSB) of the instruction word at the lowest numbered address.

Consider the big-endian mapping of instruction at address 0x00, where, for example, *p* = add r7, r7, r4:



On the other hand, in a little-endian program the same instruction is arranged with the least-significant byte (LSB) of the instruction word at the lowest numbered address:



When an instruction is fetched from memory, the instruction must be placed in the pipeline with its bytes in the proper order. Otherwise, the instruction decoder cannot recognize it. Book E numbers bits such that the most-significant byte of an instruction is the byte containing bits 0:7 of the instruction, the next most-significant byte of an instruction is the byte containing bits 8:15 of the instruction, the next most-significant byte of an instruction is the byte containing bits 16:23 of the instruction, and the least-significant byte of an instruction is the byte containing bits 24:31 of the instruction, as depicted in the instruction format diagrams (see Section 1.7, "Instruction Formats", on page 18). Given the difference in byte order between the two endian formats, the processor must perform whatever byte reversal is required (depending on the particular byte order in use) before attempting to execute the instruction. This reversal may occur between the memory interface and an instruction cache, or between the cache and the instruction decoder.

If storage is reprogrammed from one endian format to the other, the contents of the storage must be reloaded with program and data structures in the appropriate Endian format. If the contents of instruction memory change, the instruction cache must be made coherent with the updates. The instruction cache must be invalidated and the updated memory contents must be fetched in the new Endian format so that the proper byte ordering occurs in the event that this byte reversal is performed between the memory interface and the cache.

1.11.3.3 Data Byte Ordering

Unlike instruction fetches, data accesses cannot be byte-reversed between memory and a data cache. Data byte ordering in memory depends on the data type (byte, halfword, word, or doubleword) of a specific data item. It is only when moving a data item of a specific type from or to an architected register that it becomes known whether byte reversal is required due to the Endian format of the data item. Therefore, byte reversal during load or store accesses is performed between memory (or the data cache) and the architected register (e.g. GPR, FPR, etc.), depending on whether the load or store was a byte, halfword, word, or doubleword access.

Referring to the big-endian and little-endian mappings of structure *s*, as shown on page 34, the differences between the byte locations of any data item in the struc-

ture depends upon the size of the particular data item. For example (again referring to the big-endian and little-endian mappings of structure *s*):

- The word *a* has its four bytes reversed within the word spanning addresses 0x00–0x03.

- The halfword *e* has its two bytes reversed within the halfword spanning addresses 0x1C–0x1D.

Note that the array of bytes d, where each data item is a byte, is not reversed when the big-endian and little-endian mappings are compared. For example, the character 'A' is located at address 0x14 in both the big-endian and little-endian mappings.

The size of the data item being loaded or stored must be known before the processor can decide whether, and if so, how to reorder the bytes when moving them between a register and storage.

- For byte loads and stores, including strings, no reordering of bytes occurs.

- For halfword loads and stores, bytes may be reversed within the halfword, depending on the byte order.

- For word loads and stores, bytes may be reversed within the word, depending on the byte order.

- For doubleword loads and stores, bytes may be reversed within the doubleword, depending on the byte order.

Note that this mechanism applies, regardless of the alignment of data.

For example, when loading a data word from storage, all four bytes of the word are retrieved from memory (or the data cache) starting with the byte at the calculated effective address and continuing with the next three higher numbered bytes. Then, the bytes are placed in the register so that the byte from either the highest address or the lowest address is placed in the least-significant byte of the register for big-endian or little-endian storage, respectively.

1.11.3.4 Integer Load and Store Byte-Reverse Instructions

Book E provides a set of *Load Byte-Reverse* and *Store Byte-Reverse* instructions that access storage that is specified as being in one byte ordering in the same manner that a non-byte-reverse *Load* or *Store* instruction would access storage that is specified as being in the opposite byte ordering. For example, a *Load Halfword Byte-Reverse Indexed* instruction performs an halfword data storage access from a big-endian storage location in the same manner as a *Load Halfword Indexed* instruction performs an halfword data storage access from a little-endian storage location, and vice-versa.

The function of the *Load Byte-Reverse* and *Store Byte-Reverse* instructions is useful when a particular page in storage contains some data written in big-endian ordering and other data written in little-endian ordering. In such an environment, the Endianness storage attribute for the page would be set according to the predominant byte ordering for the page, and 'normal', non-byte-reverse *Load* and *Store* instructions would be used to access data operands which used this predominant byte ordering. Conversely, *Load Byte-Reverse* and *Store Byte-Reverse* instructions would be used to access the data operands which used the other byte ordering.

1.11.3.5 Origin of Endian

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

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

1.12 Synchronization

The synchronization described in this section refers to the state of the processor that is performing the synchronization.

1.12.1 Context Synchronization

An instruction or event is *context synchronizing* if it satisfies the requirements listed below. Such instructions and events are collectively called *context synchronizing operations*. Examples of context synchronizing operations include the ***sc*** instruction, the ***rfi*** and ***rfci*** instructions, 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 will cause.

2. The instructions that precede the operation complete execution in the context (privilege, address space, storage protection, etc.) in which they were initiated.

3. If the operation directly causes an interrupt (e.g., ***sc*** directly causes a System Call interrupt) or is an interrupt, the operation is not initiated until no interrupt-causing exception exists having higher priority than the exception associated with the interrupt (see Section 7.9 on page 178).

4. The instructions that follow the operation will be 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 out-of-order also be discarded, except as described in Section 6.1.5 on page 112.)

A context synchronizing operation is necessarily execution synchronizing; see Section 1.12.2. “Execution Synchronization”. Unlike the ***msync*** and ***mbar*** instructions, a context synchronizing operation does not affect the order in which storage accesses are performed with respect to other processors and mechanisms.

1.12.2 Execution Synchronization

An instruction is *execution synchronizing* if it satisfies items 1 and 2 of the definition of context synchronization (see Section 1.12.1). ***msync*** is treated like ***isync*** with respect to item 1 (i.e., the conditions described in item 1 apply to the completion of ***msync***). Examples of execution synchronizing instructions are ***msync***, ***mtmsr***, ***wrtee***, and ***wrteei***. Also, all context synchronizing instructions are execution synchronizing.

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

Chapter 2 Processor Control

2.1 Processor Control Registers

2.1.1 Machine State Register

The Machine State Register (MSR) is a 32-bit register. Machine State Register bits are numbered 32 (most-significant bit) to 63 (least-significant bit). This register defines the state of the processor (i.e. enabling and disabling of interrupts and debugging exceptions, selection of address space for instruction and data storage accesses, and specifying whether the processor is in supervisor or user mode).

The Machine State Register contents are automatically saved, altered, and restored by the interrupt-handling mechanism, as described in Section 7.5 on page 151. If a non-critical interrupt is taken, the contents of the Machine State Register are automatically copied into Save/Restore Register 1. If a critical interrupt is taken, the contents of the Machine State Register are automatically copied into Critical Save/Restore Register 1. When an ***rfi*** or ***rfci*** is executed, the contents of the Machine State Register are restored from Save/Restore Register 1 or Critical Save/Restore Register 1, respectively.

The contents of the Machine State Register can be read into bits 32:63 of a GPR using *mfmsr RT*, setting bits 0:31 of GPR(RT) to an undefined value. The contents of bits 32:63 of GPR(RS) can be written to the Machine State Register using *mtmsr RS*. MSR_{EE} may be set or cleared atomically using ***wrtee*** or ***wrteei***.

Table 2-1. Machine State Register Definition

Bit(s)	Description
32:37	Reserved
38	Allocated for implementation-dependent use
39:44	Reserved

Bit(s)	Description	
45	***Wait State Enable*** (WE)	
	=0	The processor is not in wait state and continues processing
	=1	The processor enters the wait state by ceasing to execute instructions and entering low power mode. The details of how the wait state is entered and exited, and how the processor behaves while in the wait state, are implementation-dependent. See the User's Manual for the implementation for complete details.
46	***Critical Enable*** (CE)	
	=0	Critical Input and Watchdog Timer interrupts are disabled
	=1	Critical Input and Watchdog Timer interrupts are enabled
47	Preserved for PowerPC ILE	
48	***External Enable*** (EE)	
	=0	External Input, Decrementer, and Fixed-Interval Timer interrupts are disabled.
	=1	External Input, Decrementer, and Fixed-Interval Timer interrupts are enabled.
49	***Problem State*** (PR)	
	=0	The processor is in supervisor mode, can execute any instruction, and can access any resource (e.g. GPRs, SPRs, MSR, etc.).
	=1	The processor is in user mode, cannot execute any privileged instruction, and cannot access any privileged resource.
	MSR_{PR} also affects storage access control, as described in Section 6.2.4.	
	Editorial Note The term "problem state" is synonymous with the term "user mode".	
50	***Floating-Point Available*** (FP)	
	=0	The processor cannot execute any floating-point instructions, including floating-point loads, stores and moves.
	=1	The processor can execute floating-point instructions.
51	***Machine Check Enable*** (ME)	
	=0	Machine Check interrupts are disabled.
	=1	Machine Check interrupts are enabled.
52	***Floating-Point Exception Mode 0*** (FE0) (See below)	
53	Allocated for implementation-dependent use	
54	***Debug Interrupt Enable*** (DE)	
	=0	Debug interrupts are disabled
	=1	Debug interrupts are enabled if $DBCR0_{IDM}$=1
55	***Floating-Point Exception Mode 1*** (FE1) (See below)	
56	Reserved	
57	Preserved for PowerPC IP	
58	***Instruction Address Space*** (IS)	
	=0	The processor directs all instruction fetches to address space 0 (TS=0 in the relevant TLB entry).
	=1	The processor directs all instruction fetches to address space 1 (TS=1 in the relevant TLB entry).
59	***Data Address Space*** (DS)	
	=0	The processor directs all data storage accesses to address space 0 (TS=0 in the relevant TLB entry).
	=1	The processor directs all data storage accesses to address space 1 (TS=1 in the relevant TLB entry).
60:61	Reserved	
62	Preserved for PowerPC RI	
63	Preserved for PowerPC LE	

Programming Note
An Machine State Register bit that is reserved may be altered by ***rfi***/***rfci***.

The Floating-Point Exception Mode bits FE0 and FE1 are interpreted as shown below. For further details see Section 5.4 on page 81.

FE0	FE1	Mode
0	0	Ignore Exceptions
0	1	Imprecise Nonrecoverable
1	0	Imprecise Recoverable
1	1	Precise

2.1.2 Processor Identification Register

The Processor Identification Register (PIR) is a 32-bit register. Processor Identification Register bits are numbered 32 (most-significant bit) to 63 (least-significant bit). The Processor Identification Register contains a value that can be used to distinguish the processor from other processors in the system.

The contents of the Processor Identification Register can be read into bits 32:63 of GPR(RT) using *mfspr RT,PIR*, setting bits 0:31 of GPR(RT) to an undefined value. The means by which the Processor Identification Register is initialized are implementation-dependent (see User's Manual).

2.1.3 Processor Version Register

The Processor Version Register (PVR) is a 32-bit read-only register. Processor Version Register bits are numbered 32 (most-significant bit) to 63 (least-significant bit). The Processor Version Register contains a value identifying the version and revision level of the processor.

The Processor Version Register distinguishes between processors that differ in attributes that may affect software. It contains two fields.

The contents of the Processor Version Register can be read into bits 32:63 of GPR(RT) using *mfspr RT,PVR*, setting bits 0:31 of GPR(RT) to an undefined value. Write access to the Processor Version Register is not provided.

Table 2-2. Processor Version Register Definition

Bit(s)	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.

Version numbers are assigned by Book E process. Revision numbers are assigned by an implementation-defined process.

Engineering Note
Although the classification of a given difference between processors as ‘major’ or ‘minor’ is somewhat arbitrary, the following are examples of differences that generally should be considered ‘major’.

- number and types of execution units
- optional facilities and instructions supported
- level of support of instructions (hard-wired or emulated)
- size, geometry, and management of caches and of TLBs

The following are examples of differences that generally should be considered ‘minor’.

- remapping a processor to a new technology
- redesigning a critical path to increase clock rate
- fixing bugs

In general, any change to a processor should cause a new PVR value to be assigned. Even a seemingly trivial change that is not expected to be apparent to software should cause a new revision number to be assigned, in case the change is later discovered to have introduced an error that software must circumvent.

2.1.4 Software-Use Special Purpose Registers

Software-use Special Purpose Registers (SPRG0 through SPRG7, and USPRG0) that have no defined functionality.

SPRG0 through SPRG2
These 64-bit registers can be accessed only in supervisor mode.

SPRG3
This 64-bit register can be written only in supervisor mode. SPRG3 can be read in supervisor mode. It is implementation-dependent whether or not SPRG3 can be read in user mode. See the User Manual for the implementation.

SPRG4 through SPRG7
These 64-bit registers can be written only in supervisor mode. These registers can be read in supervisor or user mode.

USPRG0
This 64-bit register can be accessed in supervisor or user mode.

The contents of SPRGi can be read into GPR(RT) using *mfspr RT,SPRGi*. The contents of GPR(RS) can be written into SPRGi using *mtspr SPRGi,RS*.

The contents of USPRG0 can be read into GPR(RT) using *mfspr RT,USPRG0*. The contents of GPR(RS) can be written into USPRG0 using *mtspr USPRG0,RS*.

2.1.5 Device Control Registers

Device Control Registers (DCRs) are on-chip registers that exist architecturally outside the processor core and thus are not actually part of Book E. Book E simply defines the existence of a Device Control Register ‘address space’ and the instructions to access them and does not define any particular Device Control Registers themselves.

Device Control Registers may control the use of on-chip peripherals, such as memory controllers (see the User's Manual for specific Device Control Register definitions).

The contents of Device Control Register DCRN can be read into GPR(RT) using *mfdcr RT,DCRN*. The contents of GPR(RS) can be written into Device Control Register DCRN using *mtdcr DCRN,RS*.

2.2 Processor Control Instructions

2.2.1 System Linkage Instructions

sc, ***rfi***, and ***rfci*** are system linkage instructions which enable the program to call upon the system to perform a service (i.e. invoke a System Call interrupt), and by which the system can return from performing a service or from processing an interrupt. System linkage instructions are context synchronizing, as defined in Section 1.12.1 on page 38.

Table 2-3. System Linkage Instruction Set Index

Mnemonic	Instruction	Page
sc	System Call	334
rfci	Return From Critical Interrupt	325
rfi	Return From Interrupt	326

2.2.2 Processor Control Register Manipulation Instructions

Table 2-4. System Register Manipulation Instruction Set Index

Mnemonic		Instruction	Page
mfdcr	RT,DCRN	Move From Device Control Register	307
mfmsr	RT	Move From Machine State Register	308
mfspr	RT,SPRN	Move From Special Purpose Register	309
mtdcr	DCRN,RS	Move To Device Control Register	311
mtmsr	RS	Move To Machine State Register	315
mtspr	SPRN,RS	Move To Special Purpose Register	316
wrtee	RA	Write MSR External Enable	368
wrteei	E	Write MSR External Enable Immediate	368

2.2.3 Instruction Synchronization Instruction

Table 2-5. Instruction Synchronization Instruction Set Index

Mnemonic	Instruction	Page
isync	Instruction Synchronize	288

2.2.4 Auxiliary Processing Query Instruction

Book E allows implementation-dependent extensions that enable processors to more optimally address a specific application area or even a broad range of application areas. Extensions can range from a single instruction to a new class of functionality (e.g. multimedia and 3-D graphics extensions).

Table 2-6. Auxiliary Processing Query Instruction Set Index

Mnemonic	Instruction	Page
mfapidi RT,RA	Move From APID Indirect	307

Chapter 3 Branch and Condition Register Operations

3.1 Branch Operations Overview

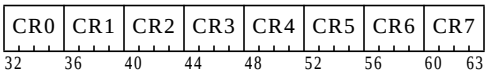
This section describes the registers and instructions that make up the branch and Condition Register operations facilities of Book E.

3.2 Registers for Branch Operations

3.2.1 Condition Register

The Condition Register (CR) is a 32-bit register. Condition Register bits are numbered 32 (most-significant bit) to 63 (least-significant bit). The Condition Register reflects the result of certain operations, and provides a mechanism for testing (and branching).

Figure 3-1. Condition Register



The bits in the Condition Register are grouped into eight 4-bit fields, named CR Field 0 (CR0), CR Field 1 (CR1),..., and CR Field 7 (CR7), which are set in one of the following ways.

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

- A specified field of the Condition Register can be set by a move to the Condition Register from another Condition Register field (***mcrf***), from the Integer Exception Register (***mcrxr***), or from the Floating-Point Status and Control Register (***mcrfs***).

- CR Field 0 can be set as the implicit result of an integer instruction.

- CR Field 1 can be set as the implicit result of a floating-point instruction.

- A specified Condition Register field can be set as the result of either an integer or a floating-point *Compare* instruction.

Instructions are provided to perform logical operations on individual Condition Register bits and to test individual Condition Register bits (see Section 3.4 on page 52).

3.2.1.1 Condition Register setting for integer instructions

For all integer word instructions in which the Rc bit is defined and set to 1, and for ***addic.***, ***andi.***, and ***andis.***, the first three bits of CR Field 0 (bits 32:34 of the Condition Register) are set by signed comparison of bits 32:63 of the result to zero, and the fourth bit of CR Field 0 (bit 35 of the Condition Register) is copied from the final state of the SO bit of the Integer Exception Register. The Rc bit is not defined for doubleword 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
```

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

The bits of CR Field 0 are interpreted as follows.

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 of the SO bit of the Integer Exception Register at the completion of the instruction.

Programming Note
CR Field 0 may not reflect the 'true' (infinitely precise) result if overflow occurs: see Section 4.3.3, "Integer Arithmetic Instructions", on page 59.

3.2.1.2 Condition Register setting for store conditional instructions

The *Integer Store Conditional* instructions ***stwcx.***, ***stwcxe.***, and ***stdcxe.*** also set CR Field 0. See instruction descriptions on page 342 and page 353 for a detailed description of how CR Field 0 is set.

3.2.1.3 Condition Register setting for floating-point instructions

For all floating-point instructions in which the Rc bit is defined and set to 1, CR Field 1 (bits 36:39 of the Condition Register) is set to the Floating-Point exception status, copied from bits 32:35 of the Floating-Point Status and Control Register. These bits are interpreted as follows.

CR Bit Description

36 ***Floating-Point Exception Summary*** (FX)
This is a copy of the final state of the FX bit of the Floating-Point Status and Control Register at the completion of the instruction.

37 ***Floating-Point Enabled Exception Summary*** (FEX)
This is a copy of the final state of the FEX bit of the Floating-Point Status and Control Register at the completion of the instruction.

38 ***Floating-Point Invalid Operation Exception Summary*** (VX)
This is a copy of the final state of the VX bit of the Floating-Point Status and Control Register at the completion of the instruction.

39 ***Floating-Point Overflow Exception*** (OX)
This is a copy of the final state of the OX bit of the Floating-Point Status and Control Register at the completion of the instruction.

3.2.1.4 Condition Register setting for compare instructions

For *Compare* instructions, a CR field specified by the BF field in the instruction is set to reflect the result of the comparison. The bits of CR Field BF are interpreted as follows. A complete description of how the bits are set is given in the instruction descriptions in Section 4.3.5, “Integer Compare Instructions”, on page 62 and Section 5.6.6, “Floating-Point Compare Instructions”, on page 104.

CR Bit Description

4×BF + 32 ***Less Than, Floating-Point Less Than*** (LT, FL)
For signed-integer *Compare*, GPR(RA) < SI or GPR(RB)
For unsigned-integer *Compare*, GPR(RA) $<_u$ UI or GPR(RB)
For floating-point *Compare*, FPR(FRA) $<_{fp}$ FPR(FRB).

4×BF + 33 ***Greater Than, Floating-Point Greater Than*** (GT, FG)
For signed-integer *Compare*, GPR(RA) > SI or GPR(RB).
For unsigned-integer *Compare*, GPR(RA) $>_u$ UI or GPR(RB).
For floating-point *Compare*, FPR(FRA) $>_{fp}$ FPR(FRB).

4×BF + 34 ***Equal, Floating-Point Equal*** (EQ, FE)
For integer *Compare*, GPR(RA) = SI, UI, or GPR(RB).
For floating-point *Compare*, FPR(FRA) $=_{fp}$ FPR(FRB).

4×BF + 35 ***Summary Overflow, Floating-Point Unordered*** (SO, FU)

For integer *Compare*, this is a copy of the final state of the SO bit of the Integer Exception Register at the completion of the instruction.

For floating-point *Compare*, one or both of FPR(FRA) and FPR(FRB) is a Not a Number.

3.2.2 Link Register

The Link Register (LR) is a 64-bit register. Link Register bits are numbered 0 (most-significant bit) to 63 (least-significant bit). The Link Register can be used to provide the branch target address for the *Branch Conditional to Link Register* instruction, and it holds the return address after *Branch and Link* instructions.

The contents of the Link Register can be read into a GPR using *mfspr RT,LR*. The contents of GPR(RS) can be written to the Link Register using *mtspr LR,RS*.

3.2.3 Count Register

The Count Register (CTR) is a 64-bit register. Count Register bits are numbered 0 (most-significant bit) to 63 (least-significant bit). Bits 32:63 of the Count Register can be used to hold a loop count that can be decremented during execution of *Branch* instructions that contain an appropriately encoded BO field. If the value in bits 32:63 of the Count Register is 0 before being decremented, it is –1 afterward and bits 0:31 are left unchanged. The entire 64-bit Count Register can also be used to provide the branch target address for the *Branch Conditional to Count Register* instruction.

The contents of the Count Register can be read into a GPR using *mfspr RT,CTR*. The contents of GPR(RS) can be written to the Count Register using *mtspr CTR,RS*.

3.3 Branch Instructions

The sequence of instruction execution can be changed by the *Branch* instructions. Because all instructions are on word boundaries, bits 62 and 63 of the generated branch target address are considered to be 0 by the processor in performing the branch.

The *Branch* instructions compute the effective address (EA) of the target in one of the following four ways, as described in Section 1.11.2.2, "Instruction Storage Addressing Modes", on page 32.

1. Adding a displacement to the address of the *Branch* instruction (*Branch* or *Branch Conditional* with AA=0).

2. Specifying an absolute address (*Branch* or *Branch Conditional* with AA=1).

3. Using the address contained in the Link Register (*Branch Conditional to Link Register*).

4. Using the address contained in the Count Register (*Branch Conditional to Count Register*).

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

Branching can be conditional or unconditional, and the return address can optionally be provided. If the return address is to be provided (LK=1), the effective address of the instruction following the *Branch* instruction is placed into the Link Register after the branch target address has been computed: this is done whether or not the branch is taken.

In *Branch Conditional* instructions, the BO field specifies the conditions under which the branch is taken. The first four bits of the BO field specify how the branch is affected by or affects the Condition Register and the Count Register. The fifth bit, shown below as having the value 'y', may be used by some implementations as described below.

The encoding for the BO field is as follows. If the BO field specifies that the Count Register is to be decremented, bits 32:63 of the Count Register are decremented. If the BO field specifies a condition that must be TRUE or FALSE, that condition is obtained from the contents of bit BI+32[1] of the Condition Register.

1. Note that bits in the Condition Register are numbered 32:63 and that 'BI' refers to the BI field in the *Branch* instruction encoding. For example, specifying BI=2 refers to bit 34 of the Condition Register.

Table 3-1. BO Encodings

BO	Description
0000y	Decrement $CTR_{32:63}$, then branch if the decremented $CTR_{32:63} \neq 0$ and the condition is FALSE.
0001y	Decrement $CTR_{32:63}$, then branch if the decremented $CTR_{32:63} = 0$ and the condition is FALSE.
001zy	Branch if the condition is FALSE.
0100y	Decrement $CTR_{32:63}$, then branch if the decremented $CTR_{32:63} \neq 0$ and the condition is TRUE.
0101y	Decrement $CTR_{32:63}$, then branch if the decremented $CTR_{32:63} = 0$ and the condition is TRUE.
011zy	Branch if the condition is TRUE.
1z00y	Decrement $CTR_{32:63}$, then branch if the decremented $CTR_{32:63} \neq 0$.
1z01y	Decrement $CTR_{32:63}$, then branch if the decremented $CTR_{32:63} = 0$.
1z1zz	Branch always.

Above, 'z' denotes a bit that is ignored.

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 field does not have a 'y' bit.

For *Branch Conditional* instructions that have a 'y' bit, using y=0 indicates that the following behavior is likely.

- If the instruction is ***bc***, ***bcl***, ***bca***, ***bcla***, ***bce***, ***bcel***, ***bcea***, or ***bcela*** with a negative value in the displacement field, the branch is taken.

- In all other cases (***bc***, ***bcl***, ***bca***, ***bcla***, ***bce***, ***bcel***, ***bcea***, or ***bcela*** with a nonnegative value in the displacement field, ***bclr***, ***bclrl***, ***bclre***, ***bclrel***, ***bcctr***, ***bcctrl***, ***bcctre***, or ***bcctrel***), the branch falls through (is not taken).

Using y=1 reverses the preceding indications.

The displacement field is used as described above even if the target is an absolute address.

Programming Note
The 'z' bits should be set to 0, as they may be assigned a meaning in some future version of the architecture.

The default value for the 'y' bit should be 0: the value 1 should be used 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.

Engineering Note
For all three *Branch Conditional* instructions, the branch should be statically predicted to be taken if the value of the following expression is 1, and to fall through if the value is 0.

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

Here 's' is bit 16 of the instruction, which is the sign bit of the displacement field if the instruction has a displacement field and is 0 otherwise. BO_4 is the 'y' bit, or a bit that is ignored for the 'branch always' encoding of the BO field. (Advantage is taken of the fact that, for ***bclr***, ***bclrl***, ***bclre***, ***bclrel***, ***bcctr***, ***bcctrl***, ***bcctre***, or ***bcctrel***, bit 16 of the instruction is part of a reserved field and therefore must be 0.)

Implementations should also consider dynamic, or run-time-driven branch prediction techniques in addition to static branch prediction, where appropriate.

Programming Note
In some implementations the processor may keep a stack of the Link Register values most recently set by *Branch and Link* instructions, with the possible exception of the form shown below for obtaining the address of the next instruction. To benefit from this stack, the following programming conventions should be used.

Let A, B, and Glue be programs.

- Obtaining the address of the next instruction:

  Use the following form of *Branch and Link*.

  ```
  bcl    20,31,$+4
  ```

- Loop counts:

  Keep them in the Count Register, and use one of the *Branch Conditional* instructions to decrement the count and to control branching (e.g., branching back to the start of a loop if the decremented counter value is nonzero).

- Computed goto's, case statements, etc.:

  Use the Count Register to hold the address to branch to, and use the ***bcctr*** instruction (LK=0) to branch to the selected address.

- Direct subroutine linkage:

  Here A calls B and B returns to A. The two branches should be as follows.

  – A calls B: use a *Branch* instruction that sets the Link Register (LK=1).
  – B returns to A: use the ***bclr*** instruction (LK=0) (the return address is in, or can be restored to, the Link Register).

- Indirect subroutine linkage:

  Here A calls Glue, Glue calls B, and B returns to A rather than to Glue. (Such a calling sequence is common in linkage code used when the subroutine that the programmer wants to call, here B, is in a different module from the caller: the Binder inserts 'glue' code to mediate the branch.) The three branches should be as follows.

  – A calls Glue: use a *Branch* instruction that sets the Link Register (LK=1).
  – Glue calls B: place the address of B into the Count Register, and use the ***bcctr*** instruction (LK=0).
  – B returns to A: use the ***bclr*** instruction (LK=0) (the return address is in, or can be restored to, the Link Register).

Table 3-2. Branch Instruction Set Index

Mnemonic		Instruction	Page
b	LI	Branch	237
bl	LI	Branch & Link	
ba	LI	Branch Absolute	
bla	LI	Branch & Link Absolute	
be	LI	Branch Extended	237
bel	LI	Branch Extended & Link	
bea	LI	Branch Extended Absolute	
bela	LI	Branch Extended & Link Absolute	
bc	BO,BI,BD	Branch Conditional	238
bcl	BO,BI,BD	Branch Conditional & Link	
bca	BO,BI,BD	Branch Conditional Absolute	
bcla	BO,BI,BD	Branch Conditional & Link Absolute	
bce	BO,BI,BD	Branch Conditional Extended	238
bcel	BO,BI,BD	Branch Conditional Extended & Link	
bcea	BO,BI,BD	Branch Conditional Extended Absolute	
bcela	BO,BI,BD	Branch Conditional Extended & Link Absolute	
bclr	BO,BI	Branch Conditional to Link Register	240
bclrl	BO,BI	Branch Conditional to Link Register & Link	
bclre	BO,BI	Branch Conditional to Link Register Extended	240
bclrel	BO,BI	Branch Conditional to Link Register Extended & Link	
bcctr	BO,BI	Branch Conditional to Count Register	239
bcctrl	BO,BI	Branch Conditional to Count Register & Link	
bcctre	BO,BI	Branch Conditional to Count Register Extended	239
bcctrel	BO,BI	Branch Conditional to Count Register Extended & Link	

3.4 Condition Register Instructions

Table 3-3. Condition Register Instruction Set Index

Mnemonic		Instruction	Page
crand	BT,BA,BB	Condition Register AND	244
crandc	BT,BA,BB	Condition Register AND with Complement	244
creqv	BT,BA,BB	Condition Register Equivalent	244
crnand	BT,BA,BB	Condition Register NAND	245
crnor	BT,BA,BB	Condition Register NOR	245
cror	BT,BA,BB	Condition Register OR	245
crorc	BT,BA,BB	Condition Register OR with Complement	246
crxor	BT,BA,BB	Condition Register XOR	246
mcrf	BF,BFA	Move Condition Register Field	305
mfcr	RT	Move From Condition Register	307
mtcrf	FXM,RS	Move To Condition Register Fields	311

Chapter 4 Integer Operations

4.1 Integer Operations Overview

This chapter describes the registers and instructions that make up the integer operations. Section 4.2 describes the registers associated with the integer operations. Section 4.3 describes the instructions associated with integer operations.

4.2 Registers for Integer Operations

4.2.1 General Purpose Registers

Implementations of this architecture provide 32 General Purpose Registers (GPRs) for integer operations. The instruction formats provide 5-bit fields for specifying the General Purpose Registers to be used in the execution of the instruction. The General Purpose Registers are numbered 0-31. Each General Purpose Register is a 64-bit register and can be used to contain address and integer data.

4.2.2 Integer Exception Register

The Integer Exception Register (XER) is a 64-bit register. Table 4-1 provides bit definitions for the Integer Exception Register.

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

The contents of the Integer Exception Register can be read into a General Purpose Register using *mfspr RT,XER*. The contents of a General Purpose Register can be written to the Integer Exception Register using *mtspr XER,RS*.

Table 4-1. Integer Exception Register Definition

Bit(s)	Description
0	***Summary Overflow 64*** (SO64) The Summary Overflow 64 bit is set to 1 whenever an instruction (except ***mtspr***) sets the Overflow 64 bit to 1. Once set to 1, the SO64 bit remains set until it is cleared by an ***mtspr*** instruction (specifying the Integer Exception Register) or an ***mcrxr*** instruction. The SO64 bit is not altered by *Compare* instructions, nor by other instructions (except ***mtspr*** to the Integer Exception Register, and ***mcrxr64***) that cannot overflow. Executing an ***mtspr*** instruction to the Integer Exception Register, supplying the values 0 for SO64 and 1 for OV64, causes SO64 to be set to 0 and OV64 to be set to 1.
1	***Overflow 64*** (OV64) The Overflow 64 bit is set to indicate that an overflow has occurred during execution of an instruction. X-form *Add, Subtract From*, and *Negate* instructions having OE=1 set OV64 to 1 if the carry out of bit 0 is not equal to the carry out of bit 1, and set OV64 to 0 otherwise. This condition reflects a signed overflow. XO-form *Multiply Low Doubleword* and *Divide Doubleword* instructions having OE=1 set OV64 to 1 if the result cannot be represented in 64 bits (***mulld, divd, divdu***), and set OV64 to 0 otherwise. The OV64 bit is not altered by *Compare* instructions, nor by other instructions (except ***mtspr*** to the Integer Exception Register, and ***mcrxr64***) that cannot overflow.
2	***Carry 64*** (CA64) The Carry 64 bit is set as follows during execution of certain instructions. *Add Carrying, Subtract From Carrying, Add Extended,* and *Subtract From Extended* instructions set CA64 to 1 if there is a carry out of bit 0, and set CA64 to 0 otherwise. CA64 can be used to indicate unsigned overflow for add and subtract operations that set CA64. *Shift Right Algebraic Doubleword* instructions set CA64 to 1 if any 1-bits have been shifted out of a negative operand, and set CA64 to 0 otherwise. The CA64 bit is not altered by *Compare* instructions, nor by other instructions (except *Shift Right Algebraic*, ***mtspr*** to the Integer Exception Register, and ***mcrxr64***) that cannot carry.
3:31	Reserved
32	***Summary Overflow*** (SO) The Summary Overflow bit is set to 1 whenever an instruction (except ***mtspr***) sets the Overflow bit. Once set, the SO bit remains set until it is cleared by an ***mtspr*** instruction (specifying the Integer Exception Register) or an ***mcrxr*** instruction. The SO bit is not altered by *Compare* instructions, nor by other instructions (except ***mtspr*** to the Integer Exception Register, and ***mcrxr***) that cannot overflow. Executing an ***mtspr*** instruction to the Integer Exception Register, supplying the values 0 for SO and 1 for OV, causes SO to be set to 0 and OV to be set to 1.
33	***Overflow*** (OV) The Overflow bit is set to indicate that an overflow has occurred during execution of an instruction. X-form *Add, Subtract From*, and *Negate* instructions having OE=1 set OV to 1 if the carry out of bit 32 is not equal to the carry out of bit 33, and set OV to 0 otherwise. This condition reflects a signed overflow. X-form *Multiply Low Word* and *Divide Word* instructions having OE=1 set OV to 1 if the result cannot be represented in 32 bits (***mullw, divw, divwu***), and set OV to 0 otherwise. The OV bit is not altered by *Compare* instructions, nor by other instructions (except ***mtspr*** to the Integer Exception Register, and ***mcrxr***) that cannot overflow.
34	***Carry*** (CA) The Carry bit is set as follows, during execution of certain instructions. *Add Carrying, Subtract From Carrying, Add Extended,* and *Subtract From Extended* instructions set it to 1 if there is a carry out of bit 32, and set it to 0 otherwise. CA can be used to indicate unsigned overflow for add and subtract operations that set CA. *Shift Right Algebraic Word* instructions set CA to 1 if any 1-bits have been shifted out of a negative operand, and set CA to 0 otherwise. The CA bit is not altered by *Compare* instructions, nor by other instructions (except *Shift Right Algebraic Word*, ***mtspr*** to the Integer Exception Register, and ***mcrxr***) that cannot carry.

Bit(s)	Description
35:56	Reserved
57:63	This field specifies the number of bytes to be transferred by a *Load String Indexed* or *Store String Indexed* instruction.

4.3 Integer Instructions

4.3.1 Integer Load Instructions

The *Integer Load* instructions compute the effective address (EA) of the storage to be accessed as described in Section 1.11.2, “Effective Address Calculation”, on page 31.

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

Many of the *Integer Load* instructions have an ‘update’ form, in which GPR(RA) is updated with the effective address. For these forms, if RA≠0 and RA≠RT, the effective address is placed into GPR(RA) and the storage element (byte, halfword, word, or doubleword) addressed by EA is loaded into GPR(RT). If RA=0 or RA=RT, the instruction form is invalid.

Integer Load storage accesses will cause a Data Storage interrupt if the program is not allowed to read the storage location. *Integer Load* storage accesses will cause a Data TLB Error interrupt if the program attempts to access storage that is unavailable (.e. not currently mapped by the TLB).

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

Book E supports both Big-Endian and Little-Endian byte ordering.

Programming Note
The DES field in DE-form *Integer Load* instructions is a word offset, not a byte offset like the DE field in DE-form *Integer Load* instructions and D field in D-form *Integer Load* instructions. However, for programming convenience, assemblers should support the specification of byte offsets for both forms of instruction.

Engineering Note
Implementations are strongly recommended to ignore bit 31 of instruction encodings for X-form *Integer Load* instructions.

Table 4-2. Basic Integer Load Instruction Set Index

Mnemonic		Instruction	Page
lbz	RT,D(RA)	Load Byte and Zero	289
lbzu	RT,D(RA)	Load Byte and Zero with Update	

Mnemonic		Instruction	Page
lbze	RT,DE(RA)	Load Byte and Zero Extended	289
lbzue	RT,DE(RA)	Load Byte and Zero with Update Extended	
lbzx	RT,RA,RB	Load Byte and Zero Indexed	289
lbzux	RT,RA,RB	Load Byte and Zero with Update Indexed	
lbzxe	RT,RA,RB	Load Byte and Zero Indexed Extended	289
lbzuxe	RT,RA,RB	Load Byte and Zero with Update Indexed Extended	
lde	RT,DES(RA)	Load Doubleword Extended	291
ldue	RT,DES(RA)	Load Doubleword with Update Extended	
ldxe	RT,RA,RB	Load Doubleword Indexed Extended	291
lduxe	RT,RA,RB	Load Doubleword with Update Indexed Extended	
lha	RT,D(RA)	Load Halfword Algebraic	294
lhau	RT,D(RA)	Load Halfword Algebraic with Update	
lhae	RT,DE(RA)	Load Halfword Algebraic Extended	294
lhaue	RT,DE(RA)	Load Halfword Algebraic with Update Extended	
lhax	RT,RA,RB	Load Halfword Algebraic Indexed	294
lhaux	RT,RA,RB	Load Halfword Algebraic with Update Indexed	
lhaxe	RT,RA,RB	Load Halfword Algebraic Indexed Extended	294
lhauxe	RT,RA,RB	Load Halfword Algebraic with Update Indexed Extended	
lhz	RT,D(RA)	Load Halfword and Zero	296
lhzu	RT,D(RA)	Load Halfword and Zero with Update	
lhze	RT,DE(RA)	Load Halfword and Zero Extended	296
lhzue	RT,DE(RA)	Load Halfword and Zero with Update Extended	
lhzx	RT,RA,RB	Load Halfword and Zero Indexed	296
lhzux	RT,RA,RB	Load Halfword and Zero with Update Indexed	
lhzxe	RT,RA,RB	Load Halfword and Zero Indexed Extended	296
lhzuxe	RT,RA,RB	Load Halfword and Zero with Update Indexed Extended	
lwz	RT,D(RA)	Load Word and Zero	303
lwzu	RT,D(RA)	Load Word and Zero with Update	
lwze	RT,DE(RA)	Load Word and Zero Extended	303
lwzue	RT,DE(RA)	Load Word and Zero with Update Extended	
lwzx	RT,RA,RB	Load Word and Zero Indexed	303
lwzux	RT,RA,RB	Load Word and Zero with Update Indexed	
lwzxe	RT,RA,RB	Load Word and Zero Indexed Extended	303
lwzuxe	RT,RA,RB	Load Word and Zero with Update Indexed Extended	

Table 4-3. Integer Load Byte-Reverse Instruction Set Index

Mnemonic		Instruction	Page
lhbrx	RT,RA,RB	Load Halfword Byte-Reverse Indexed	295
lhbrxe	RT,RA,RB	Load Halfword Byte-Reverse Indexed Extended	295
lwbrx	RT,RA,RB	Load Word Byte-Reverse Indexed	302
lwbrxe	RT,RA,RB	Load Word Byte-Reverse Indexed Extended	302

When used in a Book E system operating with Big-Endian byte order, these instructions have the effect of loading data in Little-Endian order. Likewise, when used in a Book E system operating with Little-Endian byte order, these instructions have the effect of loading data in Big-Endian order.

Table 4-4. Integer Load Multiple Instruction Set Index

Mnemonic		Instruction	Page
lmw	RT,D(RA)	Load Multiple Word	297

Table 4-5. Integer Load String Instruction Set Index

Mnemonic		Instruction	Page
lswi	RT,RA,NB	Load String Word Immediate	298
lswx	RT,RA,RB	Load String Word Indexed	298

The *Integer Load String* instructions, in combination with the *Integer Store String* instructions, allow movement of data from storage to registers or from registers to storage without concern for alignment. These instructions can be used for a short move between arbitrary storage locations or to initiate a long move between unaligned storage fields.

Table 4-6. Integer Load and Reserve Instruction Set Index

Mnemonic		Instruction	Page
ldarxe	RT,RA,RB	Load Doubleword And Reserve Indexed Extended	290
lwarx	RT,RA,RB	Load Word And Reserve Indexed	300
lwarxe	RT,RA,RB	Load Word And Reserve Indexed Extended	

The *Integer Load And Reserve* instructions (***lwarx***, ***lwarxe***, and ***ldarxe***), in combination with the *Integer Store Conditional* instructions (***stwcx.***, ***stwcxe.***, and ***stdcxe.***), permit the programmer to write a sequence of instructions that appear to perform an atomic update operation on a storage location. This operation depends upon 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 doublewords.

Programming Note
Because the *Integer Load And Reserve* instructions have implementation dependencies (e.g., the granularity at which reservations are managed), they must be used with care. The operating system should provide system library programs that use these instructions to implement the high-level synchronization functions (Test and Set, Compare and Swap, etc.) needed by application programs. Application programs should use these library programs, rather than use the *Integer Load And Reserve* instructions directly.

Architecture Note
The *Integer Load And Reserve* instructions require the EA to be aligned. Software should not attempt to emulate an unaligned *Load And Reserve* instruction, because there is no correct way to define the address associated with the reservation.

Engineering Note
Causing an Alignment interrupt to be invoked if an attempt is made to execute a *Load And Reserve* instruction having an incorrectly aligned effective address facilitates the debugging of software by signalling the exception when and where the exception occurs.

Programming Note
The granularity with which reservations are managed is implementation-dependent. Therefore the storage to be accessed by the *Load And Reserve* instructions should be allocated by a system library program. Additional information can be found in Section 6.1.6.2, “Atomic Update Primitives”, on page 117.

4.3.2 Integer Store Instructions

The *Integer Store* instructions compute the effective address (EA) of the storage to be accessed as described in Section 1.11.2, “Effective Address Calculation”, on page 31.

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

Many of the *Integer Store* instructions have an ‘update’ form, in which GPR(RA) is updated with the effective address. For these forms, the following rules apply.

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

Integer Store storage accesses will cause a Data Storage interrupt if the program is not allowed to write to the storage location. *Integer Store* storage accesses will cause a Data TLB Error interrupt if the program attempts to access storage that is unavailable.

Book E supports both Big-Endian and Little-Endian byte ordering.

Programming Note
The DES field in DE-form *Integer Store* instructions is a word offset, not a byte offset like the DE field in DE-form *Integer Store* instructions and D field in D-form *Integer Store* instructions. However, for programming convenience, assemblers should support the specification of byte offsets for both forms of instruction.

Engineering Note
Implementations are strongly recommended to ignore bit 31 of instruction encodings for X-form *Integer Store* instructions.

Table 4-7. Basic Integer Store Instruction Set Index

Mnemonic		Instruction	Page
stb	RS,D(RA)	Store Byte	341
stbu	RS,D(RA)	Store Byte with Update	
stbe	RS,DE(RA)	Store Byte Extended	341
stbue	RS,DE(RA)	Store Byte with Update Extended	
stbx	RS,RA,RB	Store Byte Indexed	341
stbux	RS,RA,RB	Store Byte with Update Indexed	
stbxe	RS,RA,RB	Store Byte Indexed Extended	341
stbuxe	RS,RA,RB	Store Byte with Update Indexed Extended	
stde	RS,DES(RA)	Store Doubleword Extended	343
stdue	RS,DES(RA)	Store Doubleword with Update Extended	
stdxe	RS,RA,RB	Store Doubleword Indexed Extended	343
stduxe	RS,RA,RB	Store Doubleword with Update Indexed Extended	
sth	RS,D(RA)	Store Halfword	347
sthu	RS,D(RA)	Store Halfword with Update	
sthe	RS,DE(RA)	Store Halfword Extended	347
sthue	RS,DE(RA)	Store Halfword with Update Extended	
sthx	RS,RA,RB	Store Halfword Indexed	347
sthux	RS,RA,RB	Store Halfword with Update Indexed	
sthxe	RS,RA,RB	Store Halfword Indexed Extended	347
sthuxe	RS,RA,RB	Store Halfword with Update Indexed Extended	
stw	RS,D(RA)	Store Word	351
stwu	RS,D(RA)	Store Word with Update	
stwe	RS,DE(RA)	Store Word Extended	351
stwue	RS,DE(RA)	Store Word with Update Extended	
stwx	RS,RA,RB	Store Word Indexed	351
stwux	RS,RA,RB	Store Word with Update Indexed	
stwxe	RS,RA,RB	Store Word Indexed Extended	351
stwuxe	RS,RA,RB	Store Word with Update Indexed Extended	

Table 4-8. Integer Store Byte-Reverse Instruction Set Index

Mnemonic		Instruction	Page
sthbrx	RS,RA,RB	Store Halfword Byte-Reverse Indexed	348
sthbrxe	RS,RA,RB	Store Halfword Byte-Reverse Indexed Extended	348
stwbrx	RS,RA,RB	Store Word Byte-Reverse Indexed	352
stwbrxe	RS,RA,RB	Store Word Byte-Reverse Indexed Extended	352

When used in a Book E system operating with Big-Endian byte order, these instructions have the effect of storing data in Little-Endian order. Likewise, when used in a Book E system operating with Little-Endian byte order, these instructions have the effect of storing data in Big-Endian order.

Table 4-9. Integer Store Multiple Instruction Set Index

Mnemonic		Instruction	Page
stmw	RS,D(RA)	Store Multiple Word	349

Table 4-10. Integer Store String Instruction Set Index

Mnemonic		Instruction	Page
stswi	RS,RA,NB	Store String Word Immediate	350
stswx	RS,RA,RB	Store String Word Indexed	350

The *Integer Store String* instructions, in combination with the *Integer Load String* instructions, allow movement of data from storage to registers or from registers to storage without concern for alignment. These instructions can be used for a short move between arbitrary storage locations or to initiate a long move between unaligned storage fields.

Table 4-11. Integer Store Conditional Instruction Set Index

Mnemonic		Instruction	Page
stdcxe.	RS,RA,RB	Store Doubleword Conditional Indexed Extended	342
stwcx.	RS,RA,RB	Store Word Conditional Indexed	353
stwcxe.	RS,RA,RB	Store Word Conditional Indexed Extended	

The *Integer Store Conditional* instructions (***stwcx.***, ***stwcxe.***, and ***stdcxe.***), in combination with the *Integer Load And Reserve* instructions (***lwarx***, ***lwarxe***, and ***ldarxe***), permit the programmer to write a sequence of instructions that appear to perform an atomic update operation on a storage location. This operation depends upon 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 doublewords.

Programming Note
Because the *Integer Store Conditional* instructions have implementation dependencies (e.g., the granularity at which reservations are managed), they must be used with care. The operating system should provide system library programs that use these instructions to implement the high-level synchronization functions (Test and Set, Compare and Swap, etc.) needed by application programs. Application programs should use these library programs, rather than use the *Integer Store Conditional* instructions directly.

Architecture Note
The *Integer Store Conditional* instructions require the EA to be aligned. Software should not attempt to emulate an unaligned *Integer Store Conditional* instruction, because there is no correct way to define the address associated with the reservation.

Engineering Note
Causing an Alignment interrupt to be invoked if an attempt is made to execute an *Integer Store Conditional* instruction having an incorrectly aligned effective address facilitates the debugging of software by signalling the exception when and where the exception occurs.

Programming Note
The granularity with which reservations are managed is implementation-dependent. Therefore the storage to be accessed by the *Integer Store Conditional* instructions should be allocated by a system library program. Additional information can be found in Section 6.1.6.2, “Atomic Update Primitives”, on page 117.

4.3.3 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 Integer Exception Register, and into CR Field 0. ***addi*** and ***addis*** use the value 0, not the contents of GPR(0), if RA=0.

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 X-form instructions with Rc=1, and the D-form instruction ***addic.*** set the first three bits of CR Field 0 to characterize bits 32:63 of the result that is placed in the target register. These bits are set by signed comparison of bits 32:63 of the result to zero.

addic[.], ***subfic***, ***addc***[***o***][.], ***subfc***[***o***][.], ***adde***[***o***][.], ***subfe***[***o***][.], ***addme***[***o***][.], ***subfme***[***o***][.], ***addze***[***o***][.], and ***subfze***[***o***][.] always set CA and CA64 to reflect the carry out of bit 32 and the carry out of bit 0, respectively.

Programming Note
Instructions with the OE bit set or that set CA may execute slowly or may prevent the execution of subsequent instructions until the instruction has completed.

The X-form *Arithmetic* instructions set SO and OV when OE=1 to reflect overflow of bits 32:63 of the result. X-form *Arithmetic* instructions also set SO64 and OV64 when OE=1 to reflect overflow of bits 0:63 of the result.

For ***mulld, divd***, and ***divdu***, SO, OV and CA are not modified. For ***mullw, divw***, and ***divwu***, SO64, OV64 and CA64 are not affected.

Programming Note
Notice that CR Field 0 may not reflect the 'true' (infinitely precise) result if overflow occurs.

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

Table 4-12. Integer Arithmetic Instruction Set Index

Mnemonic		Instruction	Page
add	RT,RA,RB	Add	229
add.	RT,RA,RB		
addo	RT,RA,RB		
addo.	RT,RA,RB		
addc	RT,RA,RB	Add Carrying	230
addc.	RT,RA,RB		
addco	RT,RA,RB		
addco.	RT,RA,RB		
adde	RT,RA,RB	Add Extended	231
adde.	RT,RA,RB		
addeo	RT,RA,RB		
addeo.	RT,RA,RB		
adde64	RT,RA,RB		
adde64o	RT,RA,RB		
addi	RT,RA,SI	Add Immediate	232
addis	RT,RA,SI	Add Immediate Shifted	
addic	RT,RA,SI	Add Immediate Carrying	233
addic.	RT,RA,SI		
addme	RT,RA,RB	Add to Minus One Extended	234
addme.	RT,RA,RB		
addmeo	RT,RA,RB		
addmeo.	RT,RA,RB		
addme64	RT,RA,RB		
addme64o	RT,RA,RB		
addze	RT,RA,RB	Add to Zero Extended	235
addze.	RT,RA,RB		
addzeo	RT,RA,RB		
addzeo.	RT,RA,RB		
addze64	RT,RA,RB		
addze64o	RT,RA,RB		
divd	RT,RA,RB	Divide Doubleword	255
divdo	RT,RA,RB		
divdu	RT,RA,RB	Divide Doubleword Unsigned	256
divduo	RT,RA,RB		
divw	RT,RA,RB	Divide Word	257
divw.	RT,RA,RB		
divwo	RT,RA,RB		
divwo.	RT,RA,RB		
divwu	RT,RA,RB	Divide Word Unsigned	258
divwu.	RT,RA,RB		
divwuo	RT,RA,RB		
divwuo.	RT,RA,RB		
mulhd	RT,RA,RB	Multiply High Doubleword	317
mulhdu	RT,RA,RB	Multiply High Doubleword Unsigned	317
mulhw	RT,RA,RB	Multiply High Word	318
mulhw.	RT,RA,RB		
mulhwu	RT,RA,RB	Multiply High Word Unsigned	318
mulhwu.	RT,RA,RB		
mulld	RT,RA,RB	Multiply Low Doubleword	319
mulldo	RT,RA,RB		
mulli	RT,RA,SI	Multiply Low Immediate	319
mullw	RT,RA,RB	Multiply Low Word	320
mullw.	RT,RA,RB		
mullwo	RT,RA,RB		
mullwo.	RT,RA,RB		
neg	RT,RA	Negate	322
neg.	RT,RA		
nego	RT,RA		
nego.	RT,RA		

Mnemonic		Instruction	Page
subf	RT,RA,RB	Subtract From	355
subf.	RT,RA,RB		
subfo	RT,RA,RB		
subfo.	RT,RA,RB		
subfc	RT,RA,RB	Subtract From Carrying	356
subfc.	RT,RA,RB		
subfco	RT,RA,RB		
subfco.	RT,RA,RB		
subfe	RT,RA,RB	Subtract From Extended	357
subfe.	RT,RA,RB		
subfeo	RT,RA,RB		
subfeo.	RT,RA,RB		
subfe64	RT,RA,RB		
subfe64o	RT,RA,RB		
subfic	RT,RA,SI	Subtract From Immediate Carrying	358
subfme	RT,RA,RB	Subtract From Minus One Extended	359
subfme.	RT,RA,RB		
subfmeo	RT,RA,RB		
subfmeo.	RT,RA,RB		
subfme64	RT,RA,RB		
subfme64o	RT,RA,RB		
subfze	RT,RA,RB	Subtract From Zero Extended	360
subfze.	RT,RA,RB		
subfzeo	RT,RA,RB		
subfzeo.	RT,RA,RB		
subfze64	RT,RA,RB		
subfze64o	RT,RA,RB		

4.3.4 Integer Logical Instructions

The *Logical* instructions perform bit-parallel operations on 64-bit operands.

The X-form *Logical* instructions with Rc=1, and the D-form *Logical* instructions ***andi.*** and ***andis.*** set the first three bits of CR Field 0 as described in Section 4.3.3, “Integer Arithmetic Instructions”, on page 59. The *Logical* instructions do not change the SO, OV, CA, SO64, OV64 and CA64 bits in the Integer Exception Register.

Table 4-13. Integer Logical Instruction Set Index

Mnemonic		Instruction	Page
and[.]	RA,RS,RB	AND	236
andc[.]	RA,RS,RB	AND with Complement	236
andi.	RA,RS,UI	AND Immediate	236
andis.	RA,RS,UI	AND Immediate Shifted	236
cntlzd	RA,RS	Count Leading Zeros Doubleword	243
cntlzw	RA,RS	Count Leading Zeros Word	243
cntlzw.	RA,RS		
eqv	RA,RS,RB	Equivalent	259
eqv.	RA,RS,RB		
extsb	RA,RS	Extend Sign Byte	260
extsb.	RA,RS		
extsh	RA,RS	Extend Sign Halfword	260
extsh.	RA,RS		
extsw	RA,RS	Extend Sign Word	260
nand	RA,RS,RB	NAND	321
nand.	RA,RS,RB		
nor	RA,RS,RB	NOR	323
nor.	RA,RS,RB		
or	RA,RS,RB	OR	324
or.	RA,RS,RB		
orc	RA,RS,RB	OR with Complement	324
orc.	RA,RS,RB		
ori	RA,RS,UI	OR Immediate	324
oris	RA,RS,UI	OR Immediate Shifted	324
xor	RA,RS,RB	XOR	369
xor.	RA,RS,RB		
xori	RA,RS,UI	XOR Immediate	369

Mnemonic		Instruction	Page
xoris	RA,RS,UI	XOR Immediate Shifted	369

4.3.5 Integer Compare Instructions

The integer *Compare* instructions compare the contents of GPR(RA) with (1) the sign-extended value of the SI field, (2) the zero-extended value of the UI field, or (3) the contents of GPR(RB). The comparison is signed for ***cmpi*** and ***cmp***, and unsigned for ***cmpli*** and ***cmpl***.

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

L	Operand length
0	32-bit operands
1	64-bit operands

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

For 32-bit implementations, the L field must be zero.

The *Compare* instructions set one bit in the left-most three bits of the designated CR field to 1, and the other two to 0. The SO bit of the Integer Exception Register is copied to bit 3 of the designated CR field.

The CR field is set as follows.

Bit	Name	Description
0	LT	(RA) < SI or GPR(RB) (signed comparison) (RA) $<_u$ UI or GPR(RB) (unsigned comparison)
1	GT	(RA) > SI or GPR(RB) (signed comparison) (RA) $>_u$ UI or GPR(RB) (unsigned comparison)
2	EQ	(RA) = SI, UI, or GPR(RB)
3	SO	Summary Overflow from the Integer Exception Register

Table 4-14. Integer Compare Instruction Set Index

Mnemonic		Instruction	Page
cmp	BF,L,RA,RB	Compare	241
cmpi	BF,L,RA,SI	Compare Immediate	241
cmpl	BF,L,RA,RB	Compare Logical	242
cmpli	BF,L,RA,SI	Compare Logical Immediate	242

4.3.6 Integer Trap Instructions

The *Trap* instructions are provided to test for a specified set of conditions from comparing the contents of one GPR with a second GPR or immediate data. If any of the conditions tested by a *Trap* instruction are met, a Trap exception type Pro-

gram interrupt is invoked. If none of the tested conditions are met, instruction execution continues normally.

The contents of GPR(RA) are compared with either the sign-extended value of the SI field or the contents of GPR(RB), depending on the *Trap* instruction. For ***tdi*** and ***td***, the entire contents of RA (and RB) participate in the comparison; for ***twi*** and ***tw***, only the contents of bits 32:63 of RA (and RB) participate in the comparison.

This comparison results in five conditions which are ANDed with TO. If the result is not 0 the Trap exception type Program interrupt is invoked. These conditions are as follows.

TO Bit	ANDed with Condition
0	Less Than, using signed comparison
1	Greater Than, using signed comparison
2	Equal
3	Less Than, using unsigned comparison
4	Greater Than, using unsigned comparison

Table 4-15. Integer Compare Instruction Set Index

Mnemonic		Instruction	Page
td	TO,RA,RB	Trap Doubleword	361
tdi	TO,RA,SI	Trap Doubleword Immediate	361
tw	TO,RA,RB	Trap Word	367
twi	TO,RA,SI	Trap Word Immediate	367

4.3.7 Integer Rotate and Shift Instructions

Instructions are provided that perform rotation operations on data from a GPR and return the result, or a portion of the result, to a GPR.

The rotation operations rotate a 64-bit quantity left by a specified number of bit positions. Bits that exit from position 0 enter at position 63.

Two types of rotation operation are supported.

For the first type, denoted $rotate_{64}$ or $ROTL_{64}$, the value rotated is the given 64-bit value. The $rotate_{64}$ operation is used to rotate a given 64-bit quantity.

For the second type, denoted $rotate_{32}$ or $ROTL_{32}$, the value rotated consists of two copies of the given 32-bit value, one copy in bits 0:31 and the other in bits 32:63. The $rotate_{32}$ operation is used to rotate a given 32-bit quantity employing the 64-bit rotator.

The *Rotate and Shift* instructions employ a mask generator. The mask is 64 bits long, and consists of 1-bits from a start bit, *mstart*, through and including a stop bit, *mstop*, and 0-bits elsewhere. The values of *mstart* and *mstop* range from 0 to 63. If *mstart* > *mstop*, the 1-bits wrap around from position 63 to position 0. Thus the mask is formed as follows:

```
if mstart ≤ mstop then
   mask[mstart:mstop]   = ones
   mask[all other bits] = zeros
else
   mask[mstart:63]      = ones
   mask[0:mstop]        = ones
   mask[all other bits] = zeros
```

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

For instructions that use the rotate$_{32}$ operation, the mask start and stop positions are always in 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 Section 4.3.3, “Integer Arithmetic Instructions”, on page 59. *Rotate and Shift* instructions do not change the OV, OV64, SO, and SO64 bits. *Rotate and Shift* instructions, except algebraic right shifts, do not change the CA or CA64 bits.

Table 4-16. Integer Rotate Instruction Set Index

Mnemonic		Instruction	Page
rldcl	RA,RS,RB,MB	Rotate Left Doubleword then Clear Left	327
rldcr	RA,RS,RB,ME	Rotate Left Doubleword then Clear Right	328
rldic	RA,RS,SH,MB	Rotate Left Doubleword Immediate then Clear	329
rldicl	RA,RS,SH,MB	Rotate Left Doubleword Immediate then Clear Left	327
rldicr	RA,RS,SH,ME	Rotate Left Doubleword Immediate then Clear Right	328
rldimi	RA,RS,SH,MB	Rotate Left Doubleword Immediate then Mask Insert	330
rlwimi rlwimi.	RA,RS,SH,MB,ME RA,RS,SH,MB,ME	Rotate Left Word Immediate then Mask Insert	331
rlwinm rlwinm.	RA,RS,SH,MB,ME RA,RS,SH,MB,ME	Rotate Left Word Immediate then AND with Mask	332
rlwnm rlwnm.	RA,RS,RB,MB,ME RA,RS,RB,MB,ME	Rotate Left Word then AND with Mask	332

These instructions 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, and if the mask bit is 0 the associated bit in the target register remains unchanged); or
- ANDed with a mask before being placed into the target register.

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

Architecture Note

For MD-form and MDS-form instructions, the MB and ME fields are used in permuted rather than sequential order because this is easier for the processor. Permuting the MB field permits the processor to obtain the low-order five bits of the MB value from the same place for all instructions having an MB field (M-form and MD-form instructions). Permuting the ME field permits the processor to treat bits 21:26 of all MD-form instructions uniformly.

Table 4-17. Integer Shift Instruction Set Index

Mnemonic		Instruction	Page
sld	RA,RS,RB	Shift Left Doubleword	335
slw slw.	RA,RS,RB RA,RS,RB	Shift Left Word	336
srad	RA,RS,RB	Shift Right Algebraic Doubleword	337
sradi	RA,RS,SH	Shift Right Algebraic Doubleword Immediate	337
sraw sraw.	RA,RS,RB RA,RS,RB	Shift Right Algebraic Word	338
srawi srawi.	RA,RS,SH RA,RS,SH	Shift Right Algebraic Word Immediate	338

Mnemonic		Instruction	Page
srd	RA,RS,RB	Shift Right Doubleword	339
srw srw.	RA,RS,RB RA,RS,RB	Shift Right Word	340

Programming Note
Any *Shift Right Algebraic* instruction, followed by ***addze***, can be used to divide quickly by 2^n.

Programming Note
Multiple-precision shifts can be programmed as shown in Section C.2, “Multiple-Precision Shifts”, on page 387.

Engineering Note
The instructions intended for use with 32-bit data are shown as doing a rotate$_{32}$ operation. This is strictly necessary only for setting the CA bit for ***srawi*** and ***sraw***. ***slw*** and ***srw*** could do a rotate$_{64}$ operation if that is easier.

4.3.8 Integer Exception Register Instructions

Table 4-18. Integer Exception Register Instruction Set Index

Mnemonic		Instruction	Page
mcrxr	BF	Move to Condition Register from Integer Exception Register	306
mcrxr64	BF	Move to Condition Register from Integer Exception Register 64	306

Chapter 5 Floating-Point Operations

5.1 Overview

This chapter describes the registers and instructions that make up the floating-point operations. Section 5.2 on page 69 describes the registers associated with floating-point operations. Section 5.6 on page 98 describes the instructions associated with floating-point operations.

This architecture specifies that the processor implement a floating-point system as defined in ANSI/IEEE Standard 754-1985, 'IEEE Standard for Binary Floating-Point Arithmetic' (hereafter referred to as 'the IEEE standard'), but requires software support in order to conform fully with that standard. That standard defines certain required 'operations' (addition, subtraction, etc.); the term 'floating-point operation' is used in this chapter to refer to one of these required operations, or to the operation performed by one of the *Multiply-Add* or *Reciprocal Estimate* instructions. All floating-point operations conform to that standard, except if software sets the Floating-Point Non-IEEE Mode (NI) bit in the Floating-Point Status and Control Register to 1 (see page 71), in which case floating-point operations do not necessarily conform to that standard.

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

These instructions are divided into two categories.

- computational instructions

  The computational instructions are those that perform addition, subtraction, multiplication, division, extracting the square root, rounding, conversion, comparison, and combinations of these operations. These instructions provide the floating-point operations. They place status information into the Floating-Point Status and Control Register. They are the instructions described in Section 5.6.4, Section 5.6.5, and Section 5.6.6.

- non-computational instructions

The non-computational instructions are those that perform loads and stores, move the contents of a floating-point register to another floating-point register possibly altering the sign, manipulate the Floating-Point Status and Control Register explicitly, and select the value from one of two floating-point registers based on the value in a third floating-point register. The operations performed by these instructions are not considered floating-point operations. With the exception of the instructions that manipulate the Floating-Point Status and Control Register explicitly, they do not alter the Floating-Point Status and Control Register. They are the instructions described in Sections 5.6.1 through 5.6.3, and 5.6.7.

A floating-point number consists of a signed exponent and a signed significand. The quantity expressed by this number is the product of the significand and the number $2^{exponent}$. Encodings are provided in the data format to represent finite numeric values, ±Infinity, and values that are 'Not a Number' (NaN). Operations involving infinities produce results obeying traditional mathematical conventions. NaNs have no mathematical interpretation. Their encoding permits a variable diagnostic information field. They may be used to indicate such things as uninitialized variables and can be produced by certain invalid operations.

There is one class of exceptional events that occur during instruction execution that is unique to floating-point operations: the Floating-Point Exception. Floating-point exceptions are signaled with bits set in the Floating-Point Status and Control Register (FPSCR). They can cause an Enabled exception type Program interrupt to be taken, precisely or imprecisely, if the proper control bits are set.

Floating-Point Exceptions

The following floating-point exceptions are detected by the processor:

- Invalid Operation Exception (VX)

SNaN	(VXSNAN)
Infinity-Infinity	(VXISI)
Infinity÷Infinity	(VXIDI)
Zero÷Zero	(VXZDZ)
Infinity×Zero	(VXIMZ)
Invalid Compare	(VXVC)
Software Request	(VXSOFT)
Invalid Square Root	(VXSQRT)
Invalid Integer Convert	(VXCVI)

- Zero Divide Exception (ZX)
- Overflow Exception (OX)
- Underflow Exception (UX)
- Inexact Exception (XX)

Each floating-point exception, and each category of Invalid Operation Exception, has an exception bit in the Floating-Point Status and Control Register. In addition, each floating-point exception has a corresponding enable bit in the Floating-Point Status and Control Register. See Section 5.2.2 on page 69 for a description of these exception and enable bits, and Section 5.4 on page 81 for a detailed discussion of floating-point exceptions, including the effects of the enable bits.

5.2 Registers for Floating-Point Operations

5.2.1 Floating-Point Registers

Implementations of this architecture provide 32 Floating-Point Registers (FPRs). The floating-point instruction formats provide 5-bit fields for specifying the Floating-Point Registers to be used in the execution of the instruction. The Floating-Point Registers are numbered 0-31.

Each Floating-Point Register contains 64 bits that support the floating-point double format. Every instruction that interprets the contents of a Floating-Point Register as a floating-point value uses the floating-point double format for this interpretation.

The computational instructions, and the *Move* and *Select* instructions, operate on data located in Floating-Point Registers and, with the exception of the *Compare* instructions, place the result value into a Floating-Point Register and optionally place status information into the Condition Register.

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

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

The computational instructions and the *Select* instruction accept values from the Floating-Point Registers 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 Floating-Point Register, and the setting of status bits in the Floating-Point Status and Control Register and in the Condition Register (if Rc=1), are undefined.

5.2.2 Floating-Point Status and Control Register

The Floating-Point Status and Control Register (FPSCR) controls the handling of floating-point exceptions and records status resulting from the floating-point operations. Bits 32:55 are status bits. Bits 56:63 are control bits.

The exception bits in the Floating-Point Status and Control Register (bits 35:45, 53:55) are sticky; that is, once set to 1 they remain set to 1 until they are set to 0 by an ***mcrfs, mtfsfi, mtfsf,*** or ***mtfsb0*** instruction. The exception summary bits in the Floating-Point Status and Control Register (FX, FEX, and VX, which are bits 32:34) are not considered to be ‘exception bits’, and only FX is sticky.

FEX and VX are simply the OR of other Floating-Point Status and Control Register bits. Therefore these two bits are not listed among the Floating-Point Status and Control Register bits affected by the various instructions.

Table 5-1. Floating-Point Status and Control Register Definition

Bit(s)	Description
32	***Floating-Point Exception Summary*** (FX) Every floating-point instruction, except ***mtfsfi*** and ***mtfsf***, implicitly sets $FPSCR_{FX}$ to 1 if that instruction causes any of the floating-point exception bits in the Floating-Point Status and Control Register to change from 0 to 1. ***mcrfs***, ***mtfsfi***, ***mtfsf***, ***mtfsb0,*** and ***mtfsb1*** can alter $FPSCR_{FX}$ explicitly.
33	***Floating-Point Enabled Exception Summary*** (FEX) This bit is the OR of all the floating-point exception bits masked by their respective enable bits. ***mcrfs***, ***mtfsfi***, ***mtfsf***, ***mtfsb0***, and ***mtfsb1*** cannot alter $FPSCR_{FEX}$ explicitly.
34	***Floating-Point Invalid Operation Exception Summary*** (VX) This bit is the OR of all the Invalid Operation exception bits. ***mcrfs***, ***mtfsfi***, ***mtfsf***, ***mtfsb0***, and ***mtfsb1*** cannot alter $FPSCR_{VX}$ explicitly.
35	***Floating-Point Overflow Exception*** (OX) See Section 5.4.3 on page 89.
36	***Floating-Point Underflow Exception*** (UX) See Section 5.4.4 on page 91.
37	***Floating-Point Zero Divide Exception*** (ZX) See Section 5.4.2 on page 88.
38	***Floating-Point Inexact Exception*** (XX) See Section 5.4.5 on page 93. $FPSCR_{XX}$ is a sticky version of $FPSCR_{FI}$ (see below). Thus the following rules completely describe how $FPSCR_{XX}$ is set by a given instruction. • If the instruction affects $FPSCR_{FI}$, the new value of $FPSCR_{XX}$ is obtained by ORing the old value of $FPSCR_{XX}$ with the new value of $FPSCR_{FI}$. • If the instruction does not affect $FPSCR_{FI}$, the value of $FPSCR_{XX}$ is unchanged.
39	***Floating-Point Invalid Operation Exception (SNaN)*** (VXSNAN) See Section 5.4.1 on page 85.
40	***Floating-Point Invalid Operation Exception (∞-∞)*** (VXISI) See Section 5.4.1 on page 85.
41	***Floating-Point Invalid Operation Exception (∞÷∞)*** (VXIDI) See Section 5.4.1 on page 85.
42	***Floating-Point Invalid Operation Exception (0÷0)*** (VXZDZ) See Section 5.4.1 on page 85.
43	***Floating-Point Invalid Operation Exception (∞×0)*** (VXIMZ) See Section 5.4.1 on page 85.
44	***Floating-Point Invalid Operation Exception (Invalid Compare)*** (VXVC) See Section 5.4.1 on page 85.
45	***Floating-Point Fraction Rounded*** (FR) The last *Arithmetic* or *Rounding and Conversion* instruction incremented the fraction during rounding. See Section 5.3.6 on page 79. This bit is not sticky.
46	***Floating-Point Fraction Inexact*** (FI) The last *Arithmetic* or *Rounding and Conversion* instruction either produced an inexact result during rounding or caused a disabled Overflow Exception. See Section 5.3.6 on page 79. This bit is not sticky. See the definition of $FPSCR_{XX}$, above, regarding the relationship between $FPSCR_{FI}$ and $FPSCR_{XX}$.

Bit(s)	Description
47:51	***Floating-Point Result Flags*** (FPRF) This field is set as described below. For arithmetic, rounding, and conversion instructions, the field is set based on the result placed into the target register, except that if any portion of the result is undefined then the value placed into FPRF is undefined. See Table 5-2 on page 72.
47	***Floating-Point Result Class Descriptor*** (C) Arithmetic, rounding, and conversion instructions may set this bit with the FPCC bits, to indicate the class of the result.
48:51	***Floating-Point Condition Code*** (FPCC) Floating-point *Compare* instructions set one of the FPCC bits to 1 and the other three FPCC bits to 0. Arithmetic, rounding, and conversion instructions may set the FPCC bits with the C bit, to indicate the class of the result. Note that in this case the high-order three bits of the FPCC retain their relational significance indicating that the value is less than, greater than, or equal to zero.
48	***Floating-Point Less Than or Negative*** (FL or <)
49	***Floating-Point Greater Than or Positive*** (FG or >)
50	***Floating-Point Equal or Zero*** (FE or =)
51	***Floating-Point Unordered or NaN*** (FU or ?)
52	Reserved
53	***Floating-Point Invalid Operation Exception (Software Request)*** (VXSOFT) This bit can be altered only by ***mcrfs, mtfsfi, mtfsf, mtfsb0,*** or ***mtfsb1***. See Section 5.4.1 on page 85.
54	***Floating-Point Invalid Operation Exception (Invalid Square Root)*** (VXSQRT) See Section 5.4.1 on page 85.
	Architecture Note This bit is defined even for implementations that do not support either of the two optional instructions that set it, namely *Floating Square Root* and *Floating Reciprocal Square Root Estimate*. Defining it for all implementations gives software a standard interface for handling square root exceptions.
	Programming Note If the implementation does not support the optional *Floating Square Root* or *Floating Reciprocal Square Root Estimate* instruction, software can simulate the instruction and set this bit to reflect the exception.
55	***Floating-Point Invalid Operation Exception (Invalid Integer Convert)*** (VXCVI) See Section 5.4.1 on page 85.
56	***Floating-Point Invalid Operation Exception Enable*** (VE) See Section 5.4.1 on page 85.
57	***Floating-Point Overflow Exception Enable*** (OE) See Section 5.4.1 on page 85.
58	***Floating-Point Underflow Exception Enable*** (UE) See Section 5.4.1 on page 85.
59	***Floating-Point Zero Divide Exception Enable*** (ZE) See Section 5.4.1 on page 85.
60	***Floating-Point Inexact Exception Enable*** (XE) See Section 5.4.1 on page 85.
61	***Floating-Point Non-IEEE Mode*** (NI) If this bit is set to 1, the remaining Floating-Point Status and Control Register bits may have meanings other than those given in this document, and the 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 this bit to 1 are described in the User's Manual and may differ between implementations.

Bit(s)	Description	
62:63	***Floating-Point Rounding Control*** (RN) See Section 5.3.6 on page 79.	
	RN=00	Round to Nearest
	RN=01	Round toward Zero
	RN=10	Round toward +Infinity
	RN=11	Round toward –Infinity

Table 5-2. 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

Architecture Note

Setting Floating-Point Non-IEEE Mode (NI) to 1 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 (i.e., without causing an Enabled exception type Program interrupt or a Floating-Point Unimplemented Instruction exception type Program interrupt, and without invoking an 'emulation assist': see Chapter 7 on page 143). The means may be controlled by one or more other Floating-Point Status and Control Register bits (recall that the other Floating-Point Status and Control Register bits have implementation-dependent meanings when NI=1).

5.3 Floating-Point Data

5.3.1 Data Format

This architecture defines the representation of a floating-point value in two different binary fixed-length formats. The format may be a 32-bit single format for a single-precision value or a 64-bit double format for a double-precision value. The single format may be used for data in storage. The double format may be used for data in storage and for data in floating-point registers.

The lengths of the exponent and the fraction fields differ between these two formats. The structure of the single and double formats is shown below.

Figure 5-1. Floating-point single format

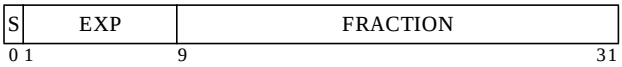
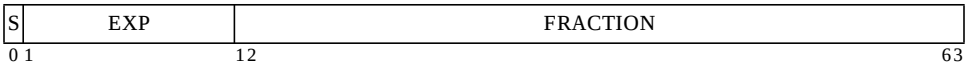


Figure 5-2. Floating-point double format



Values in floating-point format are composed of three fields:

S	sign bit
EXP	exponent+bias
FRACTION	fraction

If only a portion of a floating-point data item in storage is accessed, such as with a load or store instruction for a byte or halfword (or word in the case of floating-point double format), the value affected will depend on whether the Book E system is operating with Big-Endian byte order or Little-Endian byte order.

Representation of numeric values in the floating-point formats consists of a sign bit (S), a biased exponent (EXP), and the fraction portion (FRACTION) of the significand. The significand consists of a leading implied bit concatenated on the right with the FRACTION. This leading implied bit is 1 for normalized numbers and 0 for denormalized numbers and is located in the unit bit position (i.e., the first bit to the left of the binary point). Values representable within the two floating-point formats can be specified by the parameters listed in Table 5-3.

Table 5-3. IEEE floating-point fields

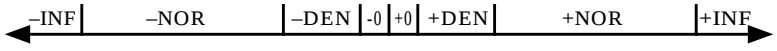
	Single	Double
Exponent Bias	+127	+1023
Maximum Exponent	+127	+1023
Minimum Exponent	–126	–1022
Widths (bits)		
Format	32	64
Sign	1	1
Exponent	8	11
Fraction	23	52
Significand	24	53

The architecture requires that the Floating-Point Registers support the floating-point double format only.

5.3.2 Value Representation

This architecture defines numeric and non-numeric values representable within each of the two supported formats. The numeric values are approximations to the real numbers and include the normalized numbers, denormalized numbers, and zero values. The non-numeric values representable are the infinities and the Not a Numbers (NaNs). The infinities are adjoined to the real numbers, but are not numbers themselves, and the standard rules of arithmetic do not hold when they are used in an operation. They are related to the real numbers by order alone. It is possible however to define restricted operations among numbers and infinities as defined below. The relative location on the real number line for each of the defined entities is shown in Figure 5-3.

Figure 5-3. Approximation to real numbers



The NaNs are not related to the numeric values or infinities by order or value but are encodings used to convey diagnostic information such as the representation of uninitialized variables.

The following is a description of the different floating-point values defined in the architecture:

Binary floating-point numbers
Machine representable values used as approximations to real numbers. Three categories of numbers are supported: normalized numbers, denormalized numbers, and zero values.

Normalized numbers (±NOR)
These are values that have an unbiased exponent value in the range:

–126 to 127 in single format
–1022 to 1023 in double format

They are values in which the implied unit bit is 1. Normalized numbers are interpreted as follows:

$$NOR = (-1)^s \times 2^E \times (1.fraction)$$

where s is the sign, E is the unbiased exponent, and 1.fraction is the significand, which is composed of a leading unit bit (implied bit) and a fraction part.

The ranges covered by the magnitude (M) of a normalized floating-point number are approximately equal to:

Single Format:

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

Double Format:

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

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

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

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

where Emin is the minimum representable exponent value (-126 for single-precision, -1022 for double-precision).

Infinities (±∞)
These are values that have the maximum biased exponent value:

255 in single format
2047 in double format

and a zero fraction value. They are used to approximate values greater in magnitude than the maximum normalized value.

Infinity arithmetic is defined as the limiting case of real arithmetic, with restricted operations defined among numbers and infinities. Infinities and the real numbers can be related by ordering in the affine sense:

$$-\infty < \text{every finite number} < +\infty$$

Arithmetic on infinities is always exact and does not signal any exception, except when an exception occurs due to the invalid operations as described in Section 5.4.1 on page 85.

Not a Numbers (NaNs)
These are values that have the maximum biased exponent value and a nonzero fraction value. The sign bit is ignored (i.e., NaNs are neither positive nor negative). If the high-order bit of the fraction field is 0 then the NaN is a *Signalling NaN*; otherwise it is a *Quiet NaN*.

Signaling NaNs are used to signal exceptions when they appear as operands of computational instructions.

Quiet NaNs are used to represent the results of certain invalid operations, such as invalid arithmetic operations on infinities or on NaNs, when Invalid Operation Exception is disabled ($FPSCR_{VE}$=0). Quiet NaNs propagate through all floating-point operations except comparison, *Floating Round to Single-Precision*, and conversion to integer. Quiet NaNs do not signal exceptions, except for ordered comparison and conversion to integer operations. Specific encodings in QNaNs can thus be preserved through a sequence of floating-point operations, and used to convey diagnostic information to help identify results from invalid operations.

When a QNaN is the result of a floating-point operation because one of the operands is a NaN or because a QNaN was generated due to a disabled Invalid Operation Exception, then the following rule is applied to determine the NaN with the high-order fraction bit set to 1 that is to be stored as the result.

```
if FPR(FRA) is a NaN
    then FPR(FRT) ← FPR(FRA)
    else if FPR(FRB) is a NaN
        then if instruction is frsp
            then FPR(FRT) ← FPR(FRB)0:34 || 29 0
            else FPR(FRT) ← FPR(FRB)
        else if FPR(FRC) is a NaN
            then FPR(FRT) ← FPR(FRC)
            else if generated QNaN
            then FPR(FRT) ← generated QNaN
```

If the operand specified by FRA is a NaN, then that NaN is stored as the result. Otherwise, if the operand specified by FRB is a NaN (if the instruction specifies an FRB operand), then that NaN is stored as the result, with the low-order 29 bits of the result set to 0 if the instruction is ***frsp***. Otherwise, if the operand specified by FRC is a NaN (if the instruction specifies an FRC operand), then that NaN is stored as the result. Otherwise, if a QNaN was generated due to a disabled Invalid Operation Exception, then that QNaN is stored as the result. If a QNaN is to be generated as a result, then the QNaN generated has a sign bit of 0, an exponent field of all 1s, and a high-order fraction bit of 1 with all other fraction bits 0. Any instruction that generates a QNaN as the result of a disabled Invalid Operation must generate this QNaN (i.e., 0x7FF8_0000_0000_0000).

A double-precision NaN is representable in single format if and only if the low-order 29 bits of the double-precision NaNs fraction are zero.

5.3.3 Sign of Result

The following rules govern the sign of the result of an arithmetic, rounding, or conversion operation, when the operation does not yield an exception. They apply even when the operands or results are zeros or infinities.

- The sign of the result of an add operation is the sign of the operand having the larger absolute value. If both operands have the same sign, the sign of the result of an add operation is the same as the sign of the operands. The sign of the result of the subtract operation x-y is the same as the sign of the result of the add operation x+(-y).

  When the sum of two operands with opposite sign, or the difference of two operands with the same sign, is exactly zero, the sign of the result is positive

in all rounding modes except Round toward -Infinity, in which mode the sign is negative.

- The sign of the result of a multiply or divide operation is the Exclusive OR of the signs of the operands.

- The sign of the result of a *Square Root* or *Reciprocal Square Root Estimate* operation is always positive, except that the square root of -0 is -0 and the reciprocal square root of -0 is -Infinity.

- The sign of the result of a *Round to Single-Precision* or *Convert To/From Integer* operation is the sign of the operand being converted.

For the *Multiply-Add* instructions, the rules given above are applied first to the multiply operation and then to the add or subtract operation (one of the inputs to the add or subtract operation is the result of the multiply operation).

5.3.4 Normalization and Denormalization

The intermediate result of an arithmetic or ***frsp*** instruction may require normalization and/or denormalization as described below. Normalization and denormalization do not affect the sign of the result.

When an arithmetic or ***frsp*** instruction produces an intermediate result, consisting of a sign bit, an exponent, and a nonzero significand with a 0 leading bit, it is not a normalized number and must be normalized before it is stored.

A number is normalized by shifting its significand left while decrementing its exponent by 1 for each bit shifted, until the leading significand bit becomes 1. The Guard bit and the Round bit (see Section 5.5.1 on page 94) participate in the shift with zeros shifted into the Round bit. The exponent is regarded as if its range were unlimited.

After normalization, or if normalization was not required, the intermediate result may have a nonzero significand and an exponent value that is less than the minimum value that can be represented in the format specified for the result. In this case, the intermediate result is said to be 'Tiny' and the stored result is determined by the rules described in Section 5.4.4 on page 91. These rules may require denormalization.

A number is denormalized by shifting its significand right while incrementing its exponent by 1 for each bit shifted, until the exponent is equal to the format's minimum value. If any significant bits are lost in this shifting process then 'Loss of Accuracy' has occurred (See Section 5.4.4 on page 91) and Underflow Exception is signaled.

Engineering Note
When denormalized numbers are operands of multiply, divide, and square root operations, some implementations may prenormalize the operands internally before performing the operations.

5.3.5 Data Handling and Precision

Instructions are defined to move floating-point data between the Floating-Point Registers and storage. For double format data, the data are not altered during the move. For single format data, a format conversion from single to double is performed when loading from storage into an Floating-Point Register and a format conversion from double to single is performed when storing from an Floating-Point Register to storage. No floating-point exceptions are caused by these instructions.

All computational, *Move*, and *Select* instructions use the floating-point double format.

Floating-point single-precision is obtained with the implementation of four types of instruction.

1. *Load Floating-Point Single*

   This form of instruction accesses a single-precision operand in single format in storage, converts it to double format, and loads it into a Floating-Point Register. No floating-point exceptions are caused by these instructions.

2. *Round to Floating-Point Single-Precision*

   The *Floating Round to Single-Precision* instruction rounds a double-precision operand to single-precision, checking the exponent for single-precision range and handling any exceptions according to respective enable bits, and places that operand into a Floating-Point Register as a double-precision operand. For results produced by single-precision arithmetic instructions, single-precision loads, and other instances of the *Floating Round to Single-Precision* instruction, this operation does not alter the value.

3. *Single-Precision Arithmetic Instructions*

   This form of instruction takes operands from the Floating-Point Registers in double format, performs the operation as if it produced an intermediate result having infinite precision and unbounded exponent range, and then coerces this intermediate result to fit in single format. Status bits, in the Floating-Point Status and Control Register and optionally in the Condition Register, are set to reflect the single-precision result. The result is then converted to double format and placed into a Floating-Point Register. The result lies in the range supported by the single format.

   All input values must be representable in single format; if they are not, the result placed into the target Floating-Point Register, and the setting of status bits in the Floating-Point Status and Control Register and in the Condition Register (if Rc=1), are undefined.

4. *Store Floating-Point Single*

   This form of instruction converts a double-precision operand to single format and stores that operand into storage. No floating-point exceptions are caused by these instructions. (The value being stored is effectively assumed to be the result of an instruction of one of the preceding three types.)

When the result of a *Load Floating-Point Single*, *Floating Round to Single-Precision*, or single-precision arithmetic instruction is stored in a Floating-Point Register, the low-order 29 FRACTION bits are zero.

Programming Note
The *Floating Round to Single-Precision* instruction is provided to allow value conversion from double-precision to single-precision with appropriate exception checking and rounding. This instruction should be used to convert double-precision floating-point values (produced by double-precision load and arithmetic instructions and by ***fcfid***) to single-precision values prior to storing them into single format storage elements or using them as operands for single-precision arithmetic instructions. Values produced by single-precision load and arithmetic instructions are already single-precision values and can be stored directly into single format storage elements, or used directly as operands for single-precision arithmetic instructions, without preceding the store, or the arithmetic instruction, by a *Floating Round to Single-Precision* instruction.

Programming Note
A single-precision value can be used in double-precision arithmetic operations. The reverse is true only if the double-precision value is representable in single format.

Some implementations may execute single-precision arithmetic instructions faster than double-precision arithmetic instructions. Therefore, if double-precision accuracy is not required, single-precision data and instructions should be used.

5.3.6 Rounding

The material in this section applies to operations that have numeric operands (i.e., operands that are not infinities or NaNs). Rounding the intermediate result of such an operation may cause an Overflow Exception, an Underflow Exception, or an Inexact Exception. The remainder of this section assumes that the operation causes no exceptions and that the result is numeric. See Section 5.3.2 on page 74 and Section 5.4 on page 81 for the cases not covered here.

The arithmetic, rounding, and conversion instructions produce an intermediate result that can be regarded as having infinite precision and unbounded exponent range. This intermediate result is normalized or denormalized if required, then rounded to the destination format. The final result is then placed into the target Floating-Point Register in double format or in integer format, depending on the instruction.

The instructions that round their intermediate result are the *Arithmetic* and *Rounding and Conversion* instructions. Each of these instructions sets Floating-Point Status and Control Register bits FR and FI. If the fraction was incremented during rounding then FR is set to 1, otherwise FR is set to 0. If the rounded result is inexact then FI is set to 1, otherwise FI is set to 0.

The two *Estimate* instructions set FR and FI to undefined values. The remaining floating-point instructions do not alter FR and FI.

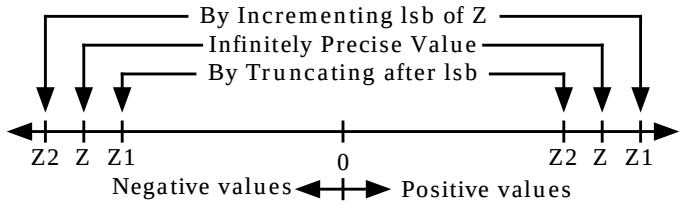
Four user-selectable rounding modes are provided through the Floating-Point Rounding Control field in the Floating-Point Status and Control Register. See Section 5.2.2 on page 69. These are encoded as follows:

RN	Rounding Mode
00	Round to Nearest
01	Round toward Zero
10	Round toward +Infinity
11	Round toward -Infinity

Let Z be the intermediate arithmetic result or the operand of a convert operation. If Z can be represented exactly in the target format, then the result in all rounding modes is Z as represented in the target format. If Z cannot be represented exactly in the target format, let Z1 and Z2 bound Z as the next larger and next smaller numbers representable in the target format. Then Z1 or Z2 can be used to approximate the result in the target format.

Figure 5-4 shows the relation of Z, Z1, and Z2 in this case. The following rules specify the rounding in the four modes. 'lsb' means 'least-significant bit'.

Figure 5-4. Selection of Z1 and Z2



Round to Nearest
Choose the value that is closer to Z (Z1 or Z2). In case of a tie, choose the one that is even (least significant bit 0).

Round toward Zero
Choose the smaller in magnitude (Z1 or Z2).

Round toward +Infinity
Choose Z1.

Round toward -Infinity
Choose Z2.

See Section 5.5.1 on page 94 for a detailed explanation of rounding.

5.4 Floating-Point Exceptions

This architecture defines the following floating-point exceptions:

- Invalid Operation Exception

  SNaN
  Infinity-Infinity
  Infinity÷Infinity
  Zero÷Zero
  Infinity×Zero
  Invalid Compare
  Software Request
  Invalid Square Root
  Invalid Integer Convert

- Zero Divide Exception
- Overflow Exception
- Underflow Exception
- Inexact Exception

These exceptions may occur during execution of computational instructions. In addition, an Invalid Operation Exception occurs when a *Move To Floating-Point Status and Control Register* instruction sets $FPSCR_{VXSOFT}$ to 1 (Software Request).

Each floating-point exception, and each category of Invalid Operation Exception, has an exception bit in the Floating-Point Status and Control Register. In addition, each floating-point exception has a corresponding enable bit in the Floating-Point Status and Control Register. The exception bit indicates occurrence of the corresponding exception. If an exception occurs, the corresponding enable bit governs the result produced by the instruction and, in conjunction with the FE0 and FE1 bits (see page 82), whether and how the Enabled exception type Program interrupt is taken. (In general, the enabling specified by the enable bit is of invoking the interrupt, not of permitting the exception to occur and the occurrence of an exception depends only on the instruction and its inputs, not on the setting of any control bits. The only deviations from this general rule are that the occurrence of an Underflow Exception may depend on the setting of the Underflow Exception enable bit , and the occurrence of an Inexact Exception may depend on the setting of the Overflow Exception and Underflow Exception enable bits.)

A single instruction, other than ***mtfsfi*** or ***mtfsf***, may set more than one exception bit only in the following cases:

- Inexact Exception may be set with Overflow Exception.

- Inexact Exception may be set with Underflow Exception.

- Invalid Operation Exception (SNaN) is set with Invalid Operation Exception (∞×0) for *Multiply-Add* instructions for which the values being multiplied are infinity and zero and the value being added is an SNaN.

- Invalid Operation Exception (SNaN) may be set with Invalid Operation Exception (Invalid Compare) for *Compare Ordered* instructions.

- Invalid Operation Exception (SNaN) may be set with Invalid Operation Exception (Invalid Integer Convert) for *Convert To Integer* instructions.

When an exception occurs the instruction execution may be suppressed or a result may be delivered, depending on the exception.

Instruction execution is suppressed for the following kinds of exception, so that there is no possibility that one of the operands is lost:

- Enabled Invalid Operation
- Enabled Zero Divide

For the remaining kinds of exception, a result is generated and written to the destination specified by the instruction causing the exception. The result may be a different value for the enabled and disabled conditions for some of these exceptions. The kinds of exception that deliver a result are the following:

- Disabled Invalid Operation
- Disabled Zero Divide
- Disabled Overflow
- Disabled Underflow
- Disabled Inexact
- Enabled Overflow
- Enabled Underflow
- Enabled Inexact

Subsequent sections define each of the floating-point exceptions and specify the action that is taken when they are detected.

The IEEE standard specifies the handling of exceptional conditions in terms of 'traps' and 'trap handlers'. In this architecture, an Floating-Point Status and Control Register exception enable bit of 1 causes generation of the result value specified in the IEEE standard for the 'trap enabled' case: the expectation is that the exception will be detected by software, which will revise the result. An Floating-Point Status and Control Register exception enable bit of 0 causes generation of the 'default result' value specified for the 'trap disabled' (or 'no trap occurs' or 'trap is not implemented') case: the expectation is that the exception will not be detected by software, which will simply use the default result. The result to be delivered in each case for each exception is described in the sections below.

The IEEE default behavior when an exception occurs is to generate a default value and not to notify software. In this architecture, if the IEEE default behavior when an exception occurs is desired for all exceptions, all Floating-Point Status and Control Register exception enable bits should be set to 0 and Ignore Exceptions Mode (see below) should be used. In this case the Enabled exception type Program interrupt is not taken, even if floating-point exceptions occur: software can inspect the Floating-Point Status and Control Register exception bits if necessary, to determine whether exceptions have occurred.

In this architecture, if software is to be notified that a given kind of exception has occurred, the corresponding Floating-Point Status and Control Register exception enable bit must be set to 1 and a mode other than Ignore Exceptions Mode must be used. In this case the Enabled exception type Program interrupt is taken if an enabled floating-point exception occurs. An Enabled exception type Program interrupt is also taken if a *Move To Floating-Point Status and Control Register* instruction causes an exception bit and the corresponding enable bit both to be 1; the *Move To Floating-Point Status and Control Register* instruction is considered to cause the enabled exception.

The FE0 and FE1 bits control whether and how an Enabled exception type Program interrupt is taken if an enabled floating-point exception occurs. The location of these bits and the requirements for altering them are described in Section 2.1.1 on page 39. (An Enabled exception type Program interrupt is never taken because

of a disabled floating-point exception.) The effects of the four possible settings of these bits are as follows.

FE0	FE1	Description
0	0	**Ignore Exceptions Mode** Floating-point exceptions do not cause an Enabled exception type Program interrupt to be taken.
0	1	**Imprecise Nonrecoverable Mode** An Enabled exception type Program interrupt is taken at some point at or beyond the instruction that caused the enabled exception. It may not be possible to identify the excepting instruction or the data that caused the exception. Results produced by the excepting instruction may have been used by or may have affected subsequent instructions that are executed before the interrupt is taken.
1	0	**Imprecise Recoverable Mode** An Enabled exception type Program interrupt is taken at some point at or beyond the instruction that caused the enabled exception. Sufficient information is provided to the interrupt that it can identify the excepting instruction and the operands, and correct the result. No results produced by the excepting instruction have been used by or have affected subsequent instructions that are executed before the interrupt is taken.
1	1	**Precise Mode** An Enabled exception type Program interrupt is taken precisely at the instruction that caused the enabled exception.

Architecture Note
The FE0 and FE1 bits of the Machine State Register are defined in Section 2.1.1 on page 39 in a manner such that they can be changed dynamically and can easily be treated as part of a process' state.

In all cases, the question of whether a floating-point result is stored, and what value is stored, is governed by the Floating-Point Status and Control Register exception enable bits, as described in subsequent sections, and is not affected by the value of the FE0 and FE1 bits.

In all cases in which an Enabled exception type Program interrupt is taken, all instructions before the instruction at which the Enabled exception type Program interrupt is taken have completed, and no instruction after the instruction at which the Enabled exception type Program interrupt is taken has begun execution. (Recall that, for the two Imprecise modes, the instruction at which the Enabled exception type Program interrupt is taken need not be the instruction that caused the exception.) The instruction at which the Enabled exception type Program interrupt is taken has not been executed unless it is the excepting

instruction, in which case it has been executed if the exception is not among those listed on page 82 as suppressed.

Programming Note
In any of the three non-Precise modes, a *Floating-Point Status and Control Register* instruction can be used to force any exceptions, due to instructions initiated before the *Floating-Point Status and Control Register* instruction, to be recorded in the Floating-Point Status and Control Register. (This forcing is superfluous for Precise Mode.)

In either of the Imprecise modes, a *Floating-Point Status and Control Register* instruction can be used to force any invocations of the Enabled exception type Program interrupt, due to instructions initiated before the *Floating-Point Status and Control Register* instruction, to occur. (This forcing has no effect in Ignore Exceptions Mode, and is superfluous for Precise Mode.)

A ***sync*** instruction, or any other execution synchronizing instruction or event (e.g., ***isync***), also has the effects described above. However, in order to obtain the best performance across the widest range of implementations, a *Floating-Point Status and Control Register* instruction should be used to obtain these effects.

In order to obtain the best performance across the widest range of implementations, the programmer should obey the following guidelines.

- If the IEEE default results are acceptable to the application, Ignore Exceptions Mode should be used with all Floating-Point Status and Control Register exception enable bits set to 0.

- If the IEEE default results are not acceptable to the application, Imprecise Nonrecoverable Mode should be used, or Imprecise Recoverable Mode if recoverability is needed, with Floating-Point Status and Control Register exception enable bits set to 1 for those exceptions for which the Enabled exception type Program interrupt is to be taken.

- Ignore Exceptions Mode should not, in general, be used when any Floating-Point Status and Control Register exception enable bits are set to 1.

- Precise Mode may degrade performance in some implementations, perhaps substantially, and therefore should be used only for debugging and other specialized applications.

Engineering Note
It is permissible for the implementation to be precise in any of the three modes that permit interrupts, or to be recoverable in Nonrecoverable Mode.

5.4.1 Invalid Operation Exception

Definition

An Invalid Operation Exception occurs when an operand is invalid for the specified operation. The invalid operations are:

- Any floating-point operation on a signaling NaN (SNaN)
- For add or subtract operations, magnitude subtraction of infinities (∞-∞)
- Division of infinity by infinity (∞÷∞)
- Division of zero by zero (0÷0)
- Multiplication of infinity by zero (∞×0)
- Ordered comparison involving a NaN (Invalid Compare)
- Square root or reciprocal square root of a negative (and nonzero) number (Invalid Square Root)
- Integer convert involving a number too large in magnitude to be represented in the target format, or involving an infinity or a NaN (Invalid Integer Convert)

In addition, an Invalid Operation Exception occurs if software explicitly requests this by executing an ***mtfsfi***, ***mtfsf***, or ***mtfsb1*** instruction that sets $FPSCR_{VXSOFT}$ to 1 (Software Request).

Programming Note
The purpose of $FPSCR_{VXSOFT}$ is to allow software to cause an Invalid Operation Exception for a condition that is not necessarily associated with the execution of a floating-point instruction. For example, it might be set by a program that computes a square root, if the source operand is negative.

Action

The action to be taken depends on the setting of the Invalid Operation Exception Enable bit of the Floating-Point Status and Control Register.

When Invalid Operation Exception is enabled ($FPSCR_{VE}$=1) and Invalid Operation occurs or software explicitly requests the exception, the following actions are taken:

1. One or two Invalid Operation Exceptions are set

$FPSCR_{VXSNAN}$	(if SNaN)
$FPSCR_{VXISI}$	(if ∞-∞)
$FPSCR_{VXIDI}$	(if ∞÷∞)
$FPSCR_{VXZDZ}$	(if 0÷0)
$FPSCR_{VXIMZ}$	(if ∞×0)
$FPSCR_{VXVC}$	(if invalid compare)
$FPSCR_{VXSOFT}$	(if software request)
$FPSCR_{VXSQRT}$	(if invalid square root)
$FPSCR_{VXCVI}$	(if invalid integer convert)

2. If the operation is an arithmetic, *Floating Round to Single-Precision*, or convert

to integer operation,

the target Floating-Point Register is unchanged
$FPSCR_{FR\ FI}$ are set to zero
$FPSCR_{FPRF}$ is unchanged

3. If the operation is a compare,

$FPSCR_{FR\ FI\ C}$ are unchanged
$FPSCR_{FPCC}$ is set to reflect unordered

4. If software explicitly requests the exception,

$FPSCR_{FR\ FI\ FPRF}$ are as set by the ***mtfsfi, mtfsf,*** or ***mtfsb1*** instruction

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

1. One or two Invalid Operation Exceptions are set

$FPSCR_{VXSNAN}$	(if SNaN)
$FPSCR_{VXISI}$	(if ∞-∞)
$FPSCR_{VXIDI}$	(if ∞÷∞)
$FPSCR_{VXZDZ}$	(if 0÷0)
$FPSCR_{VXIMZ}$	(if ∞×0)
$FPSCR_{VXVC}$	(if invalid compare)
$FPSCR_{VXSOFT}$	(if software request)
$FPSCR_{VXSQRT}$	(if invalid square root)
$FPSCR_{VXCVI}$	(if invalid integer convert)

2. If the operation is an arithmetic or *Floating Round to Single-Precision* operation,

the target Floating-Point Register is set to a Quiet NaN
$FPSCR_{FR\ FI}$ are set to zero
$FPSCR_{FPRF}$ is set to indicate the class of the result (Quiet NaN)

3. If the operation is a convert to 64-bit integer operation,

the target Floating-Point Register is set as follows:

FPR(FRT) is set to the most positive 64-bit integer if the operand in FPR(FRB) is a positive number or +∞, and to the most negative 64-bit integer if the operand in FPR(FRB) is a negative number, -∞, or NaN

$FPSCR_{FR\ FI}$ are set to zero

$FPSCR_{FPRF}$ is undefined

4. If the operation is a convert to 32-bit integer operation,

the target Floating-Point Register is set as follows:

$FPR(FRT)_{0:31} \leftarrow$ undefined

$FPR(FRT)_{32:63}$ are set to the most positive 32-bit integer if the operand in FPR(FRB) is a positive number or +∞, and to the most negative 32-bit integer if the operand in FPR(FRB) is a negative number, -∞, or NaN

$FPSCR_{FR\ FI}$ are set to zero

$FPSCR_{FPRF}$ is undefined

5. If the operation is a compare,

   $FPSCR_{FR\ FI\ C}$ are unchanged
   $FPSCR_{FPCC}$ is set to reflect unordered

6. If software explicitly requests the exception,

   $FPSCR_{FR\ FI\ FPRF}$ are as set by the ***mtfsfi, mtfsf,*** or ***mtfsb1*** instruction

5.4.2 Zero Divide Exception

Definition

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

Architecture Note
The name is a misnomer used for historical reasons. The proper name for this exception should be 'Exact Infinite Result from Finite Operands' corresponding to what mathematicians call a 'pole'.

Action

The action to be taken depends on the setting of the Zero Divide Exception Enable bit of the Floating-Point Status and Control Register.

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

1. Zero Divide Exception is set

   $FPSCR_{ZX} \leftarrow 1$

2. The target Floating-Point Register is unchanged

3. $FPSCR_{FR\ FI}$ are set to zero

4. $FPSCR_{FPRF}$ is unchanged

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

1. Zero Divide Exception is set

   $FPSCR_{ZX} \leftarrow 1$

2. The target Floating-Point Register is set to ±Infinity, where the sign is determined by the XOR of the signs of the operands

3. $FPSCR_{FR\ FI}$ are set to zero

4. $FPSCR_{FPRF}$ is set to indicate the class and sign of the result (±Infinity)

5.4.3 Overflow Exception

Definition

Overflow occurs when the magnitude of what would have been the rounded result if the exponent range were unbounded exceeds that of the largest finite number of the specified result precision.

Action

The action to be taken depends on the setting of the Overflow Exception Enable bit of the Floating-Point Status and Control Register.

When Overflow Exception is enabled ($FPSCR_{OE}$=1) and exponent overflow occurs, the following actions are taken:

1. Overflow Exception is set

   $FPSCR_{OX} \leftarrow 1$

2. For double-precision arithmetic instructions, the exponent of the normalized intermediate result is adjusted by subtracting 1536

3. For single-precision arithmetic instructions and the *Floating Round to Single-Precision* instruction, the exponent of the normalized intermediate result is adjusted by subtracting 192

4. The adjusted rounded result is placed into the target Floating-Point Register

5. $FPSCR_{FPRF}$ is set to indicate the class and sign of the result (±Normal Number)

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

1. Overflow Exception is set

   $FPSCR_{OX} \leftarrow 1$

2. Inexact Exception is set

   $FPSCR_{XX} \leftarrow 1$

3. The result is determined by the rounding mode ($FPSCR_{RN}$) and the sign of the intermediate result as follows:

   A. Round to Nearest
   Store ± Infinity, where the sign is the sign of the intermediate result

   B. Round toward Zero
   Store the format's largest finite number with the sign of the intermediate result

   C. Round toward +Infinity
   For negative overflow, store the format's most negative finite number; for positive overflow, store +Infinity

   D. Round toward -Infinity

For negative overflow, store -Infinity; for positive overflow, store the format's largest finite number

4. The result is placed into the target Floating-Point Register

5. $FPSCR_{FR}$ is undefined

6. $FPSCR_{FI}$ is set to 1

7. $FPSCR_{FPRF}$ is set to indicate the class and sign of the result (±Infinity or ±Normal Number)

5.4.4 Underflow Exception

Definition

Underflow Exception is defined separately for the enabled and disabled states:

- Enabled:
  Underflow occurs when the intermediate result is 'Tiny'.

- Disabled:
  Underflow occurs when the intermediate result is 'Tiny' and there is 'Loss of Accuracy'.

A 'Tiny' result is detected before rounding, when a nonzero intermediate result computed as though both the precision and the exponent range were unbounded would be less in magnitude than the smallest normalized number.

If the intermediate result is 'Tiny' and Underflow Exception is disabled ($FPSCR_{UE}$=0) then the intermediate result is denormalized (see Section 5.3.4 on page 77) and rounded (see Section 5.3.6 on page 79) before being placed into the target Floating-Point Register.

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

Action

The action to be taken depends on the setting of the Underflow Exception Enable bit of the Floating-Point Status and Control Register.

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

1. Underflow Exception is set

   $FPSCR_{UX} \leftarrow 1$

2. For double-precision arithmetic instructions, the exponent of the normalized intermediate result is adjusted by adding 1536

3. For single-precision arithmetic instructions and the *Floating Round to Single-Precision* instruction, the exponent of the normalized intermediate result is adjusted by adding 192

4. The adjusted rounded result is placed into the target Floating-Point Register

5. $FPSCR_{FPRF}$ is set to indicate the class and sign of the result (±Normalized Number)

Programming Note
The FR and FI bits are provided to allow the Enabled exception type Program interrupt, when taken because of an Underflow Exception, to simulate a 'trap disabled' environment. That is, the FR and FI bits allow the Enabled exception type Program interrupt to unround the result, thus allowing the result to be denormalized.

When Underflow Exception is disabled ($FPSCR_{UE}$=0) and underflow occurs, the following actions are taken:

1. Underflow Exception is set

   $FPSCR_{UX} \leftarrow 1$

2. The rounded result is placed into the target Floating-Point Register

3. $FPSCR_{FPRF}$ is set to indicate the class and sign of the result (±Normalized Number, ±Denormalized Number, or ±Zero)

5.4.5 Inexact Exception

Definition

An Inexact Exception occurs when one of two conditions occur during rounding:

1. The rounded result differs from the intermediate result assuming both the precision and the exponent range of the intermediate result to be unbounded. In this case the result is said to be inexact. (If the rounding causes an enabled Overflow Exception or an enabled Underflow Exception, an Inexact Exception also occurs only if the significands of the rounded result and the intermediate result differ.)

2. The rounded result overflows and Overflow Exception is disabled.

Action

The action to be taken does not depend on the setting of the Inexact Exception Enable bit of the Floating-Point Status and Control Register.

When Inexact Exception occurs, the following actions are taken:

1. Inexact Exception is set

   $FPSCR_{XX} \leftarrow 1$

2. The rounded or overflowed result is placed into the target Floating-Point Register

3. $FPSCR_{FPRF}$ is set to indicate the class and sign of the result

Programming Note
In some implementations, enabling Inexact Exceptions may degrade performance more than does enabling other types of floating-point exception.

5.5 Floating-Point Execution Models

All implementations of this architecture must provide the equivalent of the following execution models to ensure that identical results are obtained.

Special rules are provided in the definition of the computational instructions for the infinities, denormalized numbers and NaNs. The material in the remainder of this section applies to instructions that have numeric operands and a numeric result (i.e., operands and result that are not infinities or NaNs), and that cause no exceptions. See Section 5.3.2 on page 74 and Section 5.4 on page 81 for the cases not covered here.

Although the double format specifies an 11-bit exponent, exponent arithmetic makes use of two additional bits to avoid potential transient overflow conditions. One extra bit is required when denormalized double-precision numbers are prenormalized. The second bit is required to permit the computation of the adjusted exponent value in the following cases when the corresponding exception enable bit is 1:

- Underflow during multiplication using a denormalized operand.
- Overflow during division using a denormalized divisor.

The IEEE standard includes 32-bit and 64-bit arithmetic. The standard requires that single-precision arithmetic be provided for single-precision operands. The standard permits double-precision floating-point operations to have either (or both) single-precision or double-precision operands, but states that single-precision floating-point operations should not accept double-precision operands. Book E follows these guidelines: double-precision arithmetic instructions can have operands of either or both precisions, while single-precision arithmetic instructions require all operands to be single-precision. Double-precision arithmetic instructions and ***fcfid*** produce double-precision values, while single-precision arithmetic instructions produce single-precision values.

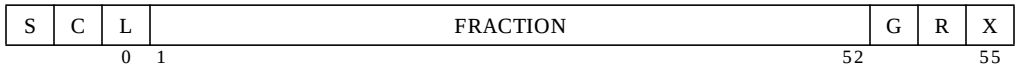
For arithmetic instructions, conversions from double-precision to single-precision must be done explicitly by software, while conversions from single-precision to double-precision are done implicitly.

5.5.1 Execution Model for IEEE Operations

The following description uses 64-bit arithmetic as an example. 32-bit arithmetic is similar except that the FRACTION is a 23-bit field, and the single-precision Guard, Round, and Sticky bits (described in this section) are logically adjacent to the 23-bit FRACTION field.

IEEE-conforming significand arithmetic is considered to be performed with a floating-point accumulator having the following format, where bits 0:55 comprise the significand of the intermediate result.

Figure 5-5. IEEE 64-bit execution model



The S bit is the sign bit.

The C bit is the carry bit, which captures the carry out of the significand.

The L bit is the leading unit bit of the significand, which receives the implicit bit from the operand.

The FRACTION is a 52-bit field that accepts the fraction of the operand.

The Guard (G), Round (R), and Sticky (X) bits are extensions to the low-order bits of the accumulator. The G and R bits are required for post-normalization of the result. The G, R, and X bits are required during rounding to determine if the intermediate result is equally near the two nearest representable values. The X bit serves as an extension to the G and R bits by representing the logical OR of all bits that may appear to the low-order side of the R bit, due either to shifting the accumulator right or to other generation of low-order result bits. The G and R bits participate in the left shifts with zeros being shifted into the R bit. Table 5-4 shows the significance of the G, R, and X bits with respect to the intermediate result (IR), the representable number next lower in magnitude (NL), and the representable number next higher in magnitude (NH).

Table 5-4. Interpretation of G, R, and X bits

G	R	X	Interpretation
0	0	0	IR is exact
0	0	1	IR closer to NL
0	1	0	
0	1	1	
1	0	0	IR midway between NL and NH
1	0	1	IR closer to NH
1	1	0	
1	1	1	

After normalization, the intermediate result is rounded, using the rounding mode specified by $FPSCR_{RN}$. If rounding results in a carry into C, the significand is shifted right one position and the exponent incremented by one. This yields an inexact result and possibly also exponent overflow. Fraction bits to the left of the bit position used for rounding are stored into the Floating-Point Register and low-order bit positions, if any, are set to zero.

Four user-selectable rounding modes are provided through $FPSCR_{RN}$ as described in Section 5.3.6 on page 79. For rounding, the conceptual Guard, Round, and Sticky bits are defined in terms of accumulator bits. Table 5-5 shows the positions of the Guard, Round, and Sticky bits for double-precision and single-precision floating-point numbers in the IEEE execution model.

Table 5-5. Location of the Guard, Round, and Sticky bits in the IEEE execution model

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

Rounding can be treated as though the significand were shifted right, if required, until the least significant bit to be retained is in the low-order bit position of the FRACTION. If any of the Guard, Round, or Sticky bits is nonzero, then the result is inexact.

Z1 and Z2, as defined on page 80, can be used to approximate the result in the target format when one of the following rules is used.

- Round to Nearest

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

  Guard bit = 1
  Depends on Round and Sticky bits:

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

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

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

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

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

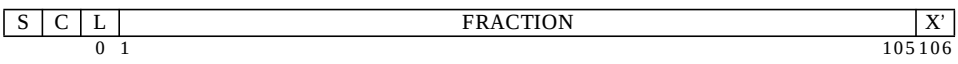
Where the result is to have fewer than 53 bits of precision because the instruction is a *Floating Round to Single-Precision* or single-precision arithmetic instruction, the intermediate result is either normalized or placed in correct denormalized form before being rounded.

5.5.2 Execution Model for Multiply-Add Type Instructions

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

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

Figure 5-6. Multiply-Add 64-bit execution model



The first part of the operation is a multiplication. The multiplication has two 53-bit significands as inputs, which are assumed to be prenormalized, and produces a result conforming to the above model. If there is a carry out of the significand (into the C bit), then the significand is shifted right one position, shifting the L bit

(leading unit bit) into the most significant bit of the FRACTION and shifting the C bit (carry out) into the L bit. All 106 bits (L bit, the FRACTION) of the product take part in the add operation. If the exponents of the two inputs to the adder are not equal, the significand of the operand with the smaller exponent is aligned (shifted) to the right by an amount that is added to that exponent to make it equal to the other input's exponent. Zeros are shifted into the left of the significand as it is aligned and bits shifted out of bit 105 of the significand are ORed into the X' bit. The add operation also produces a result conforming to the above model with the X' bit taking part in the add operation.

The result of the addition is then normalized, with all bits of the addition result, except the X' bit, participating in the shift. The normalized result serves as the intermediate result that is input to the rounder.

For rounding, the conceptual Guard, Round, and Sticky bits are defined in terms of accumulator bits. Table 5-6 shows the positions of the Guard, Round, and Sticky bits for double-precision and single-precision floating-point numbers in the multiply-add execution model.

Table 5-6. Location of the Guard, Round, and Sticky bits in the multiply-add execution model

Format	Guard	Round	Sticky
Double	53	54	OR of 55:105, X'
Single	24	25	OR of 26:105, X'

The rules for rounding the intermediate result are the same as those given in Section 5.5.1 on page 94.

If the instruction is *Floating Negative Multiply-Add* or *Floating Negative Multiply-Subtract*, the final result is negated.

5.6 Floating-Point Instructions

Architecture Note
The rules followed in assigning new primary and extended opcodes.

1. Primary opcode 63 is used for the double-precision arithmetic instructions as well as miscellaneous instructions (e.g. *Floating-Point Status and Control Register Manipulation* instructions). Primary opcode 59 is used for the single-precision arithmetic instructions.

2. 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.

3. 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 = 0b11.

   - Bit 26 = 1 iff the instruction is A-form.

   - Bits 26:29 = 0b0000 iff the instruction is a comparison or *mcrfs* (i.e., iff the instruction sets an explicitly-designated CR field).

   - Bits 26:28 = 0b001 iff the instruction explicitly refers to or sets the Floating-Point Status and Control Register (i.e., is a *Floating-Point Status and Control Register* instruction) and is not *mcrfs*.

   - Bits 26:30 = 0b01000 iff the instruction is a *Move Register* instruction, or any other instruction that does not refer to or set the Floating-Point Status and Control Register.

4. 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 *Floating-Point Status and Control Register* 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 iff the instruction is A-form.

   - Bits 26:30 = 0b01000 iff the instruction is a *Move Register* instruction, or any other instruction that does not refer to or set the Floating-Point Status and Control Register.

5.6.1 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 Floating-Point Register. The conversion and loading steps are as follows.

Let $WORD_{0:31}$ be the floating-point single-precision operand accessed from storage.

Normalized Operand

if $WORD_{1:8} > 0$ and $WORD_{1:8} < 255$ then
  $FPR(FRT)_{0:1} \leftarrow WORD_{0:1}$
  $FPR(FRT)_{2} \leftarrow \neg WORD_{1}$
  $FPR(FRT)_{3} \leftarrow \neg WORD_{1}$
  $FPR(FRT)_{4} \leftarrow \neg WORD_{1}$
  $FPR(FRT)_{5:63} \leftarrow WORD_{2:31} \| {}^{29}0$

Denormalized Operand

if $WORD_{1:8} = 0$ and $WORD_{9:31} \neq 0$ then
  sign $\leftarrow WORD_{0}$
  exp $\leftarrow$ -126
  $frac_{0:52} \leftarrow 0b0 \| WORD_{9:31} \| {}^{29}0$
  normalize the operand
    do while $frac_{0} = 0$
      $frac \leftarrow frac_{1:52} \| 0b0$
      exp $\leftarrow$ exp - 1
  $FPR(FRT)_{0} \leftarrow$ sign
  $FPR(FRT)_{1:11} \leftarrow$ exp + 1023
  $FPR(FRT)_{12:63} \leftarrow frac_{1:52}$

Zero / Infinity / NaN

if $WORD_{1:8} = 255$ or $WORD_{1:31} = 0$ then
  $FPR(FRT)_{0:1} \leftarrow WORD_{0:1}$
  $FPR(FRT)_{2} \leftarrow WORD_{1}$
  $FPR(FRT)_{3} \leftarrow WORD_{1}$
  $FPR(FRT)_{4} \leftarrow WORD_{1}$
  $FPR(FRT)_{5:63} \leftarrow WORD_{2:31} \| {}^{29}0$

Engineering Note
The above description of the conversion steps is a model only. The actual implementation may vary from this but must produce results equivalent to what this model would produce.

For double-precision *Load Floating-Point* instructions no conversion is required, as the data from storage are copied directly into the Floating-Point Register.

Many of the *Floating-Point Load* instructions have an 'update' form, in which GPR(RA) is updated with the effective address. For these forms, if RA≠0 and RA≠RT, the effective address is placed into GPR(RA) and the storage element (byte, halfword, word, or doubleword) addressed by EA is loaded into FPR(RT). If RA=0 or RA=RT, the instruction form is invalid.

Floating-Point Load storage accesses will cause a Data Storage interrupt if the program is not allowed to read the storage location. *FLoating-Point Load* storage accesses will cause a Data TLB Error interrupt if the program attempts to access storage that is unavailable.

Note: Recall that RA and RB denote General Purpose Registers, while FRT denotes a Floating-Point Register.

Book E supports both Big-Endian and Little-Endian byte ordering.

Engineering Note
Implementations are strongly recommended to ignore bit 31 of instruction encodings for X-form *Floating-Point Load* instructions.

Table 5-7. Floating-Point Load Instruction Set Index

Mnemonic		Instruction	Page
lfd	FRT,D(RA)	Load Floating-Point Double	292
lfdu	FRT,D(RA)	Load Floating-Point Double with Update	
lfde	FRT,DES(RA)	Load Floating-Point Double Extended	292
lfdue	FRT,DES(RA)	Load Floating-Point Double with Update Extended	
lfdx	FRT,RA,RB	Load Floating-Point Double Indexed	292
lfdux	FRT,RA,RB	Load Floating-Point Double with Update Indexed	
lfdxe	FRT,RA,RB	Load Floating-Point Double Indexed Extended	292
lfduxe	FRT,RA,RB	Load Floating-Point Double with Update Indexed Extended	
lfs	FRT,D(RA)	Load Floating-Point Single	293
lfsu	FRT,D(RA)	Load Floating-Point Single with Update	
lfse	FRT,DES(RA)	Load Floating-Point Single Extended	293
lfsue	FRT,DES(RA)	Load Floating-Point Single with Update Extended	
lfsx	FRT,RA,RB	Load Floating-Point Single Indexed	293
lfsux	FRT,RA,RB	Load Floating-Point Single with Update Indexed	
lfsxe	FRT,RA,RB	Load Floating-Point Single Indexed Extended	293
lfsuxe	FRT,RA,RB	Load Floating-Point Single with Update Indexed Extended	

5.6.2 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, described on page 345. Because the FPRs support only floating-point double format for floating-point data, single-precision *Store Floating-Point* instructions convert double-precision data to single format prior to storing the operand into storage. The conversion steps are as follows.

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

No Denormalization Required (includes Zero / Infinity / NaN)
if $FPR(FRS)_{1:11} > 896$ or $FPR(FRS)_{1:63} = 0$ then
    $WORD_{0:1} \leftarrow FPR(FRS)_{0:1}$
    $WORD_{2:31} \leftarrow FPR(FRS)_{5:34}$

Denormalization Required
if $874 \le FRS_{1:11} \le 896$ then
    sign $\leftarrow FPR(FRS)_0$
    exp $\leftarrow FPR(FRS)_{1:11} - 1023$
    frac $\leftarrow$ 0b1 || $FPR(FRS)_{12:63}$
    denormalize operand
        do while exp < –126
            frac $\leftarrow$ 0b0 || $frac_{0:62}$
            exp $\leftarrow$ exp + 1
    $WORD_0 \leftarrow$ sign
    $WORD_{1:8} \leftarrow$ 0x00
    $WORD_{9:31} \leftarrow frac_{1:23}$
else WORD $\leftarrow$ undefined

Notice that if the value to be stored by a single-precision *Store Floating-Point* instruction is larger in magnitude than the maximum number representable in single format, the first case above ('No Denormalization Required') applies. The result stored in WORD is then a well-defined value, but is not numerically equal to the value in the source register (i.e., the result of a single-precision *Load Floating-Point* from WORD will not compare equal to the contents of the original source register).

Engineering Note
The above description of the conversion steps is a model only. The actual implementation may vary from this but must produce results equivalent to what this model would produce.

For double-precision *Store Floating-Point* instructions and for the *Store Floating-Point as Integer Word* instruction no conversion is required, as the data from the Floating-Point Register are copied directly into storage.

Many of the *Floating-Point Store* instructions have an 'update' form, in which GPR(RA) is updated with the effective address. For these forms, if RA≠0, the effective address is placed into GPR(RA).

Floating-Point Store storage accesses will cause a Data Storage interrupt if the program is not allowed to write to the storage location. *Floating-Point Store* storage accesses will cause a Data TLB Error interrupt if the program attempts to access storage that is unavailable.

Note: Recall that RA and RB denote General Purpose Registers, while FRS denotes a Floating-Point Register.

Book E supports both Big-Endian and Little-Endian byte ordering.

Engineering Note
Implementations are strongly recommended to ignore bit 31 of instruction encodings for X-form *Floating-Point Store* instructions.

Table 5-8. Floating-Point Store Instruction Set Index

Mnemonic		Instruction	Page
stfd	FRS,D(RA)	Store Floating-Point Double	344
stfdu	FRS,D(RA)	Store Floating-Point Double with Update	
stfde	FRS,DES(RA)	Store Floating-Point Double Extended	344
stfdue	FRS,DES(RA)	Store Floating-Point Double with Update Extended	
stfdx	FRS,RA,RB	Store Floating-Point Double Indexed	344
stfdux	FRS,RA,RB	Store Floating-Point Double with Update Indexed	
stfdxe	FRS,RA,RB	Store Floating-Point Double Indexed Extended	344
stfduxe	FRS,RA,RB	Store Floating-Point Double with Update Indexed Extended	
stfiwx	FRS,RA,RB	Store Floating-Point as Integer Word Indexed	345
stfiwxe	FRS,RA,RB	Store Floating-Point as Integer Word Indexed Extended	345
stfs	FRS,D(RA)	Store Floating-Point Single	346
stfsu	FRS,D(RA)	Store Floating-Point Single with Update	
stfse	FRS,DES(RA)	Store Floating-Point Single Extended	346
stfsue	FRS,DES(RA)	Store Floating-Point Single with Update Extended	
stfsx	FRS,RA,RB	Store Floating-Point Single Indexed	346
stfsux	FRS,RA,RB	Store Floating-Point Single with Update Indexed	
stfsxe	FRS,RA,RB	Store Floating-Point Single Indexed Extended	346
stfsuxe	FRS,RA,RB	Store Floating-Point Single with Update Indexed Extended	

5.6.3 Floating-Point Move Instructions

These instructions copy data from one floating-point register to another, altering the sign bit (bit 0) as described below for ***fneg, fabs***, and ***fnabs***. These instructions treat NaNs just like any other kind of value (e.g., the sign bit of a NaN may be altered by ***fneg, fabs***, and ***fnabs***). These instructions do not alter the Floating-Point Status and Control Register.

Table 5-9. Floating-Point Move Instruction Set Index

Mnemonic		Instruction	Page
fabs	FRT,FRB	Floating Absolute Value	261
fabs.	FRT,FRB		
fmr	FRT,FRB	Floating Move Register	272
fmr.	FRT,FRB		
fnabs	FRT,FRB	Floating Negative Absolute Value	275
fnabs.	FRT,FRB		
fneg	FRT,FRB	Floating Negate	275
fneg.	FRT,FRB		

5.6.4 Floating-Point Arithmetic Instructions

5.6.4.1 Floating-Point Elementary Arithmetic Instructions

Table 5-10. Floating-Point Elementary Arithmetic Instruction Set Index

Mnemonic		Instruction	Page
fadd	FRT,FRA,FRB	Floating Add	262
fadd.	FRT,FRA,FRB		
fadds	FRT,FRA,FRB	Floating Add Single	262
fadds.	FRT,FRA,FRB		
fdiv	FRT,FRA,FRB	Floating Divide	270
fdiv.	FRT,FRA,FRB		
fdivs	FRT,FRA,FRB	Floating Divide Single	270
fdivs.	FRT,FRA,FRB		
fmul	FRT,FRA,FRC	Floating Multiply	274
fmul.	FRT,FRA,FRC		
fmuls	FRT,FRA,FRC	Floating Multiply Single	274
fmuls.	FRT,FRA,FRC		
fres	FRT,FRB	Floating Reciprocal Estimate Single	278
fres.	FRT,FRB		
frsqrte	FRT,FRB	Floating Reciprocal Square Root Estimate	282
frsqrte.	FRT,FRB		
fsqrt	FRT,FRB	Floating Square Root	284
fsqrt.	FRT,FRB		
fsqrts	FRT,FRB	Floating Square Root Single	284
fsqrts.	FRT,FRB		
fsub	FRT,FRA,FRB	Floating Subtract	285
fsub.	FRT,FRA,FRB		
fsubs	FRT,FRA,FRB	Floating Subtract Single	285
fsubs.	FRT,FRA,FRB		

5.6.4.2 Floating-Point Multiply-Add Instructions

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

Status bits are set as follows.

- Overflow, Underflow, and Inexact Exception bits, the FR and FI bits, and the FPRF field are set based on the final result of the operation, and not on the result of the multiplication.

- Invalid Operation Exception bits are set as if the multiplication and the addition were performed using two separate instructions (***fmul[s]***, followed by ***fadd[s]*** or ***fsub[s]***). That is, multiplication of infinity by 0 or of anything by an SNaN, and/or addition of an SNaN, cause the corresponding exception bits to be set.

Table 5-11. Floating-Point Multiply-Add Instruction Set Index

Mnemonic		Instruction	Page
fmadd	FRT,FRA,FRB,FRC	Floating Multiply-Add	271
fmadd.	FRT,FRA,FRB,FRC		
fmadds	FRT,FRA,FRB,FRC	Floating Multiply-Add Single	271
fmadds.	FRT,FRA,FRB,FRC		
fmsub	FRT,FRA,FRB,FRC	Floating Multiply-Subtract	273
fmsub.	FRT,FRA,FRB,FRC		
fmsubs	FRT,FRA,FRB,FRC	Floating Multiply-Subtract Single	273
fmsubs.	FRT,FRA,FRB,FRC		
fnmadd	FRT,FRA,FRB,FRC	Floating Negative Multiply-Add	276
fnmadd.	FRT,FRA,FRB,FRC		
fnmadds	FRT,FRA,FRB,FRC	Floating Negative Multiply-Add Single	276
fnmadds.	FRT,FRA,FRB,FRC		
fnmsub	FRT,FRA,FRB,FRC	Floating Negative Multiply-Subtract	277
fnmsub.	FRT,FRA,FRB,FRC		
fnmsubs	FRT,FRA,FRB,FRC	Floating Negative Multiply-Subtract Single	277
fnmsubs.	FRT,FRA,FRB,FRC		

5.6.5 Floating-Point Rounding and Conversion Instructions

Programming Note
Examples of uses of these instructions to perform various conversions can be found in Section C.3 on page 389.

Table 5-12. Floating-Point Rounding and Conversion Instruction Set Index

Mnemonic		Instruction	Page
fcfid	FRT,FRB	Floating Convert From Integer Doubleword	263
fctid	FRT,FRB	Floating Convert To Integer Doubleword	266
fctidz	FRT,FRB	Floating Convert To Integer Doubleword and round to Zero	266
fctiw	FRT,FRB	Floating Convert To Integer Word	268
fctiw.	FRT,FRB		
fctiwz	FRT,FRB	Floating Convert To Integer Word and round to Zero	268
fctiwz.	FRT,FRB		
frsp	FRT,FRB	Floating Round to Single-Precision	279
frsp.	FRT,FRB		

5.6.6 Floating-Point Compare Instructions

The floating-point *Compare* instructions compare the contents of two floating-point registers. Comparison ignores the sign of zero (i.e., regards +0 as equal to -0). The comparison result can be ordered or unordered.

The comparison sets one bit in the designated CR field to 1 and the other three to 0. The FPCC is set in the same way.

The CR field and the FPCC are set as follows.

Bit	NameDescription
0	FL(FRA) < (FRB)
1	FG(FRA) > (FRB)
2	FE(FRA) = (FRB)
3	FU(FRA) ? (FRB) (unordered)

Table 5-13. Floating-Point Compare and Select Instruction Set Index

Mnemonic		Instruction	Page
fcmpo	BF,FRA,FRB	Floating Compare Ordered	265
fcmpu	BF,FRA,FRB	Floating Compare Unordered	265
fsel fsel.	FRT,FRA,FRB,FRC FRT,FRA,FRB,FRC	Floating Select	283

5.6.7 Floating-Point Status and Control Register Instructions

Every *Floating-Point Status and Control Register* instruction synchronizes the effects of all floating-point instructions executed by a given processor. Executing a *Floating-Point Status and Control Register* instruction ensures that all floating-point instructions previously initiated by the given processor have completed before the *Floating-Point Status and Control Register* instruction is initiated, and that no subsequent floating-point instructions are initiated by the given processor until the *Floating-Point Status and Control Register* instruction has completed. In particular:

- All exceptions that will be caused by the previously initiated instructions are recorded in the Floating-Point Status and Control Register before the *Floating-Point Status and Control Register* instruction is initiated.
- All invocations of the Enabled exception type Program interrupt that will be caused by the previously initiated instructions have occurred before the *Floating-Point Status and Control Register* instruction is initiated.
- No subsequent floating-point instruction that depends on or alters the settings of any Floating-Point Status and Control Register bits is initiated until the *Floating-Point Status and Control Register* instruction has completed.

(*Floating-Point Load* and *Floating-Point Store* instructions are not affected.)

Table 5-14. Floating-Point Status and Control Register Instruction Set Index

Mnemonic		Instruction	Page
mcrfs	BF,BFA	Move to Condition Register from FPSCR	306
mffs mffs.	FRT FRT	Move From FPSCR	308
mtfsb0 mtfsb0.	BT BT	Move To FPSCR Bit 0	312
mtfsb1 mtfsb1.	BT BT	Move To FPSCR Bit 1	312
mtfsf mtfsf.	FLM,FRB FLM,FRB	Move To FPSCR Fields	313
mtfsfi mtfsfi.	BF,U BF,U	Move To FPSCR Field Immediate	314

Chapter 6 Storage

6.1 Storage Model

6.1.1 Introduction

Section 1.11 defines storage as a linear array of bytes indexed from 0 to a maximum of $2^{64} - 1$. Each byte is identified by its index, called its address, and each byte contains a value. This information is sufficient to allow the programming of applications that require no special features of any particular system environment. This chapter expands this simple storage model to include caches, virtual storage, and shared storage multiprocessors, and in conjunction with services provided by the operating system, describes a mechanism that permits explicit control of this expanded storage model. A simple model for sequential execution allows at most one storage access to be performed at a time and requires that all storage accesses appear to be performed in program order. In contrast to this simple model, Book E specifies a relaxed model of memory consistency. In a multiprocessor system that allows multiple copies of a location, aggressive implementations of Book E can permit intervals of time during which different copies of a location have different values. This chapter describes features of Book E that enable programmers to write correct programs for this memory model.

6.1.2 Storage Addressing

A program references storage using the effective address computed by the processor when it executes a load, store, branch, or cache management instruction, and when it fetches the next sequential instruction. The effective address is translated to a real address according to procedures described in Section 6.2.2 and Section 6.2.3. The real address is sent to the memory subsystem to perform the storage access (see Figure 6-2 on page 128).

For a complete discussion of storage addressing and effective address calculation, see Section 1.11.

The storage model provides the following features:

- Book E allows storage implementations to take advantage of the performance benefits of weak ordering of storage accesses between processors or between processors and devices.
- Processor ordering: storage accesses by a single processor appear to complete sequentially from the view of the programming model but may complete out of order with respect to the ultimate destination in the storage hierarchy. Order is guaranteed at each level of the storage hierarchy for accesses to the same address from the same processor.
- Book E provides the following instructions that allow the programmer to ensure a consistent and ordered storage state: ***dcbf***[***e***], ***dcbst***[***e***], ***dcbz***[***e***], ***icbi***[***e***], ***isync***, ***lwarx***[***e***], ***ldarxe***, ***msync***, ***mbar***, ***stwcx***[***e***]., ***stdcxe.***, ***tlbsync***.
- Storage consistency between processors and between a processor and devices is controlled by software through mode bits in the TLB entry.

6.1.2.1 Virtual Storage

The Book E system implements a virtual storage model for applications. This means that a combination of hardware and software can present a storage model that allows applications to exist within a 'virtual' address space larger than either the effective address space or the real address space.

Each program can access 2^{64} bytes of 'effective address' (EA) space, subject to limitations imposed by the operating system. In a typical 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 address is translated to a real address (i.e., to an address of a byte in real storage or on an I/O device) before being used to access storage. The hardware accomplishes this using the address translation mechanism described in Section 6.2.3. The operating system manages the real (physical) storage resources of the system by setting up the tables and other information used by the hardware address translation mechanism.

Storage access instruction descriptions deal primarily with effective addresses. Each such effective address lies in a 'virtual page', which is mapped to a 'real page' before data in the virtual page are accessed.

In general, real storage may not be large enough to map all the virtual pages used by the currently active applications. With support provided by hardware, the operating system can attempt to use the available real pages to map a sufficient set of

virtual pages of the applications. If a sufficient set is maintained, 'paging' activity is minimized. If not, performance degradation is likely.

The operating system can support restricted access to virtual pages (including individual enables for user state read, write, and execute, and supervisor state read, write, and execute: see Section 6.2.4), based on system standards (e.g. program code might be execute only, data structures mapped as read/write/no execute) and application requests.

6.1.2.2 Instruction Fetch

Instructions are fetched from the address space specified by MSR_{IS} (see Section 6.2.2 on page 125). When any context synchronizing event occurs, any prefetched instructions are discarded and then instructions are re-fetched using the then-current state of MSR_{IS} and the then-current program counter.

Instructions are fetched using the address translated by the TLB mechanism. Instructions are not fetched from no-execute storage (UX=0 or SX=0, see Section 6.2.4.1). If the effective address of the current instruction is mapped to no-execute storage, an Instruction Storage interrupt is generated.

However, it is permissible for an instruction from no-execute storage to be in the instruction cache if it was fetched into that cache when its effective address was mapped to execute permitted storage. However, attempted execution of such instructions will still result in an Instruction Storage interrupt. Thus, for example, the operating system can mark an application's instruction pages as no-execute without having to first flush them from the instruction cache.

6.1.2.3 Implicit Branch

Explicitly altering certain Machine State Register bits (using ***mtmsr***), or explicitly altering TLB entries or certain system registers may have the side effect of changing the addresses, virtual or real, from which the current instruction stream is being fetched. This side effect is called an implicit branch. For example, an ***mtmsr*** instruction that changes the value of MSR_{IS} may change the real address from which the current instruction stream is being fetched. The Machine State Register bits and system registers for which alteration can cause an implicit branch are indicated as such in Chapter 11 on page 225. Implicit branches are not supported by Book E. If an implicit branch occurs, the results are boundedly undefined. Software is required to precede or follow any implicit branch operation with the appropriate synchronization operation, as specified in Chapter 11 on page 225.

6.1.2.4 Data Storage Access

Data accesses are performed to or from the address space specified by MSR_{DS}. When the state of MSR_{DS} changes, subsequent accesses are made using the new state of MSR_{DS} following a context synchronizing operation (see Chapter 11 on page 225).

The effective address is translated by the TLB mechanism.

6.1.2.5 Invalid Real Address

An attempt to fetch from, load from, or store to a real address that is not physically present in the machine may result in a Machine Check interrupt (see Section 7.4.4 on page 151). This can occur by having the translation mechanism set up in a way that causes nonexistent storage to be addressed.

6.1.3 Single-Copy Atomicity

An access is *single-copy atomic*, or simply *atomic*, if it is always performed in its entirety with no visible fragmentation. Atomic accesses are thus *serialized*: each happens in its entirety in some order, even when that order is not specified in the program or enforced between processors.

In Book E the following single-register accesses (i.e. aligned scalar accesses less than or equal to the implemented width of the storage interface) are always atomic:

- byte accesses (all bytes are aligned on byte boundaries)
- halfword accesses aligned on halfword boundaries
- word accesses aligned on word boundaries
- doubleword accesses aligned on doubleword boundaries

No other accesses are guaranteed to be atomic. For example, the access caused by the following instructions is not guaranteed to be atomic.

- any *Load* or *Store* instruction for which the operand is unaligned
- ***lmw***, ***stmw***, ***lswi***, ***lswx***, ***stswi***, ***stswx***
- any *Cache Management* instruction

An access that is not atomic is performed as a set of smaller disjoint atomic accesses. The number and alignment of these accesses are implementation-dependent, as is the relative order in which they are performed.

The results for several combinations of loads and stores to the same or overlapping locations are described below.

- When two processors execute atomic stores to locations that do not overlap, and no other stores are performed to those locations, the contents of those locations are the same as if the two stores were performed by a single processor.
- When two processors execute atomic stores to the same storage location, and no other store is performed to that location, the contents of that location are the result stored by one of the processors.
- When two processors execute stores that have the same target location and are not guaranteed to be atomic, and no other store is performed to that location, the result is some combination of the bytes stored by both processors.
- When two processors execute multiple-byte stores to overlapping locations, and no other store is performed to those locations, the result is some combination of the bytes stored by the processors to the overlapping bytes. The portions of the locations that do not overlap contain the bytes stored by the processor storing to the location.
- When a processor executes an atomic store to a location, a second processor executes an atomic load from that location, and no other store is performed to that location, the value returned by the load is the contents of the location prior to the store or the contents of the location subsequent to the store.

- When a load and a store with the same target location can be executed simultaneously, and no other store is performed to the location, the value returned by the load is some combination, at the granularity of an atomic access, of the contents of the location before the store and after the store.

Engineering Note
Atomicity of storage accesses is provided by the processor in conjunction with the storage subsystem. The processor must provide a storage subsystem interface that is sufficient to allow a storage subsystem to meet the atomicity requirements specified here.

6.1.4 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, e.g., in the descriptions of the *Cache Management* instructions in Section 6.3.2. Alternative cache models may be implemented (e.g., a 'combined cache' model, in which a single cache is used for both instructions and data, or a model in which there are several levels of caches), but they must support the programming model implied by a Harvard-style cache.

The processor is not required to maintain copies of storage locations in the instruction cache that are consistent with modifications to those storage locations (e.g., modifications caused 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 (e.g., by a *Store* instruction) and the modified data have not been written to main storage. The only exception to this rule is described in Section 6.2.5.1.

Cache Management instructions are provided so that programs can manage the caches when needed. For example, program management of the caches is needed when a program generates or modifies code that will be executed (i.e., when the program modifies data in storage and then attempts to execute the modified data as instructions). The *Cache Management* instructions are also useful in optimizing the use of memory bandwidth in such applications as graphics and numerically intensive computing. The functions performed by these instructions depend on the storage attributes associated with the specified storage location (see Section 6.2.5).

The *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***[***e***])
- invalidate the copy of storage in an instruction cache block (***icbi***[***e***])
- discard prefetched instructions (***isync***)
- invalidate the copy of storage in a data cache block (***dcbi***[***e***])
- 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***[***e***], ***dcbtst***[***e***])

- 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***[***e***])

- set the contents of a data cache block to zeros (***dcbz***[***e***])

- copy the contents of a modified data cache block to main storage (***dcbst***[***e***])

- 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***[***e***]).

6.1.5 Performing Operations Out-of-Order

An operation is said to be performed 'in-order' if, at the time that it is performed, it is known to be required by the sequential execution model. An operation is said to be performed 'out-of-order' if, at the time that it is performed, it is not known to be required by the sequential execution model.

Architecture Note
In earlier versions of the architecture specification, 'speculative' was used instead of 'out-of-order'. The terminology was changed to be consistent with the technical literature, where 'speculative execution' often means the execution of instructions past unresolved branches and 'out-of-order execution' means execution of an instruction before it is known to be required by the sequential execution model. Because the meaning of 'speculative' in the literature differs from ordinary English usage the term would cause confusion no matter how the architecture specification defined it, so the term is no longer used here at all.

Operations are performed out-of-order by the 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 really needed is contingent on everything that might divert the control flow away from the instruction, such as *Branch*, *Trap*, *System Call*, ***rfi*** and ***rfci*** instructions, and interrupts, and on everything that might change the context in which the instruction is executed.

Typically, the hardware performs operations out-of-order when it has resources that would otherwise be idle, so the operation incurs little or no cost. If subsequent events such as branches or interrupts indicate that the operation would not have been performed in the sequential execution model, the processor abandons any results of the operation (except as described below).

Most operations can be performed out-of-order, as long as the machine appears to follow the sequential execution model. Certain out-of-order operations are restricted, as follows.

- **Stores**

  A *Store* instruction may not be executed out-of-order in a manner such that the alteration of the target location can be observed by other processors or mechanisms.

- **Accessing Guarded storage**

  The restrictions for this case are given in Section 6.2.5.4.

No error of any kind other than Machine Check may be reported due to an operation that is performed out-of-order, until such time as it is known that the opera-

tion is required by the sequential execution model. The only other permitted side effect (other than Machine Check) of performing an operation out-of-order is non-Guarded storage locations that could be fetched into a cache by in-order execution may be fetched out-of-order into that cache.

Engineering Note
Out-of-order execution of the *Storage Synchronization* instructions ***lwarx***[***e***], ***ldarxe***, ***stwcx***[***e***]., and ***stdcxe.*** is extremely complex and is not recommended.

Engineering Note
Because an asynchronous exception can become pending at any time, it might seem that, for example, if MSR_{EE}=1 then fetching or executing any instruction beyond the current instruction is an out-of-order operation. However, these operations need not be treated as out-of-order if the taking of the interrupt is delayed until after they have completed. Similar considerations apply to Floating-Point Enabled Exception type Program interrupts when one of the Imprecise floating-point exception modes is in effect.

Engineering Note
Implementations that perform operations out-of-order must take care to obey the sequential execution model except as permitted by Book E. Examples of cases that may require special attention include the following.

- changes of control flow, including ***sc***, *Trap*, ***rfi***, ***rfci***, and interrupts as well as branches
- changes of context due to changes of control flow. For example, the code at a branch target location, or the handler for System Call or Trap interrupts, may change the context and then return, so that the instructions immediately following the *Branch*, ***sc***, or *Trap* execute in a new context
- changes to resources, including but not limited to $MSR_{PR\ IS\ DS}$ and TLB entries, that affect address translation, access control, or storage control attributes, when the change is followed by the appropriate software synchronization
- execution synchronizing and context synchronizing operations

In general, Guarded storage is not accessed out-of-order, except in the following cases.

- **Load Instruction**

  If a copy of the location is in a cache then the location may be accessed in the cache or in main storage.

- **Instruction Fetch**

  There is no restriction on how a processor may perform instruction fetching from guarded space, as long as the storage is execute permitted (UX/SX=1).

  Programming Note
  Software should mark guarded space as no-execute (UX=0 and SX=0) to prevent inadvertent instruction fetch from guarded areas of storage.

6.1.6 Shared Storage

Book E supports the sharing of storage between programs, between different instances of the same program, and between processors and other mechanisms. It also supports access to a storage location by one or more programs using different effective addresses. All these cases are considered storage sharing. Storage is shared at a page granularity.

When the same storage location has different effective addresses, the addresses are said to be *aliases*. Each application can be granted separate access privileges to aliased pages.

Engineering Note
Page-level aliasing can be implemented in many ways, such as with real-addressed caches, L2 directories, or an external signal to an inverse directory. Each processor implementation will decide on its level of implementation in support of its system requirements.

6.1.6.1 Storage Access Ordering

The storage model for the ordering of storage accesses is *weakly consistent*. This model provides an opportunity for improved performance over a model that has stronger consistency rules, but places the responsibility on the program to ensure that ordering or synchronization instructions are properly placed when necessary for the correct execution of the program.

The order in which the processor performs storage accesses, the order in which those accesses are performed with respect to another processor or mechanism, and the order in which those accesses are performed in main storage may all be different. Several means of enforcing an ordering of storage accesses are provided to allow programs to share storage with other programs, or with mechanisms such as I/O devices. These means are listed below. The phrase 'to the extent required by the associated Memory Coherence Required attributes' refers to the Memory Coherence Required attribute, if any, associated with each access.

- If two *Store* instructions specify storage locations that are both Caching Inhibited and Guarded, the corresponding storage accesses are performed in program order with respect to any processor or mechanism.

- If a *Load* instruction depends on the value returned by a preceding *Load* instruction (because the value is used to compute the effective address specified by the second *Load*), the corresponding storage accesses are performed in program order with respect to any processor or mechanism to the extent required by the associated Memory Coherence Required attributes. This applies even if the dependency has no effect on program logic (e.g., the value returned by the first *Load* is ANDed with zero and then added to the effective address specified by the second *Load*).

- When a processor (P1) executes an ***msync*** or ***mbar*** instruction a *memory barrier* is created, which separates applicable storage accesses into two groups, G1 and G2. G1 includes all applicable storage accesses associated with instructions preceding the barrier-creating instruction, and G2 includes all applicable storage accesses associated with instructions following the barrier-creating instruction. The memory barrier ensures that all storage accesses in G1 are performed with respect to any processor or mechanism, to the extent required by the associated Memory Coherence Required attributes, before any storage accesses in G2 are performed with respect to that processor or mechanism.

The ordering done by a memory barrier is said to be 'cumulative' if it also orders storage accesses that are performed by processors and mechanisms other than P1, as follows.

– G1 includes all applicable storage accesses by any such processor or mechanism that have been performed with respect to P1 before the memory barrier is created.

– G2 includes all applicable storage accesses by any such processor or mechanism that are performed after a *Load* instruction executed by that processor or mechanism has returned the value stored by a store that is in G2.

The memory barrier created by ***msync*** is cumulative, and applies to all storage accesses except those associated with fetching instructions following the ***msync*** instruction. See the definition of ***mbar*** on page 304 for a description of the corresponding properties of the memory barrier created by that instruction.

No ordering should be assumed among the storage accesses caused by a single instruction (i.e, by an instruction for which the access is not atomic), and no means are provided for controlling that order.

Programming Note
The first example below illustrates cumulative ordering of storage accesses preceding a memory barrier, and the second illustrates cumulative ordering of storage accesses following a memory barrier. Assume that locations X, Y, and Z initially contain the value 0.

Example 1:

Processor A:	stores the value 1 to location X
Processor B:	loads from location X obtaining the value 1, executes an ***msync*** instruction, then stores the value 2 to location Y
Processor C:	loads from location Y obtaining the value 2, executes an ***msync*** instruction, then loads from location X

Example 2:

Processor A:	stores the value 1 to location X, executes an ***msync*** instruction, then stores the value 2 to location Y
Processor B:	loops loading from location Y until the value 2 is obtained, then stores the value 3 to location Z
Processor C:	loads from location Z obtaining the value 3, executes an ***msync*** instruction, then loads from location X

In both cases, cumulative ordering dictates that the value loaded from location X by processor C is 1.

Engineering Note
It is permissible to perform a dependent load before the load on which it depends, if software accessing shared storage cannot tell the difference.

It is always permissible to prefetch a data cache block from non-Guarded storage based on predicting the effective address specified by a *Load* or *Store* instruction.

Programming Note

Because stores cannot be performed 'out-of-order' (see Section 6.1.5), if a *Store* instruction depends on the value returned by a preceding *Load* instruction (because the value returned by the *Load* is used to compute either the effective address specified by the *Store* or the value to be stored), the corresponding storage accesses are performed in program order. The same applies if the *Store* instruction that is executed depends on a conditional *Branch* instruction that in turn depends on the value returned by a preceding *Load* instruction.

Because an ***isync*** instruction prevents the execution of instructions following the ***isync*** until instructions preceding the ***isync*** have completed, if an ***isync*** follows a conditional *Branch* instruction that depends on the value returned by a preceding *Load* instruction, the load on which the *Branch* depends is performed before any loads caused by instructions following the ***isync***. This applies even if the effects of the 'dependency' are independent of the value loaded (e.g., the value is compared to itself and the *Branch* tests the EQ bit in the selected CR field), and even if the branch target is the next sequential instruction to be executed.

With the exception of the cases described above and earlier in this section, data dependencies and control dependencies do not order storage accesses. Examples include the following.

- If a *Load* instruction specifies the same storage location as a preceding *Store* instruction and the location is in storage that is not Caching Inhibited, the load may be satisfied from a 'store queue' (a buffer into which the processor places stored values before presenting them to the storage subsystem), and not be visible to other processors and mechanisms. A consequence is that if a subsequent *Store* depends on the value returned by the *Load*, the two stores need not be performed in program order with respect to other processors and mechanisms.

- Because a *Store Conditional* instruction may complete before its store has been performed, a conditional *Branch* instruction that depends on the CR0 value set by a *Store Conditional* instruction does not order the *Store Conditional*'s store with respect to storage accesses caused by instructions that follow the *Branch*.

- Because processors may predict branch target addresses and branch condition resolution, control dependencies (e.g., branches) do not order storage accesses except as described above. For example, when a subroutine returns to its caller the return address may be predicted, with the result that loads caused by instructions at or after the return address may be performed before the load that obtains the return address is performed.

Because processors may implement non-architected duplicates of architected resources (e.g., GPRs, CR fields, and the Link Register), resource dependencies (e.g., specification of the same target register for two *Load* instructions) do not order storage accesses.

Examples of correct uses of dependencies, ***msync***, and ***mbar*** to order storage accesses can be found in Appendix D on page 397.

Because the storage model is weakly consistent, the sequential execution model as applied to instructions that cause storage accesses guarantees only that those accesses appear to be performed in program order with respect to the processor executing the instructions. For example, an instruction may complete, and subsequent instructions may be executed, before storage accesses caused by the first instruction have been performed. However, for a sequence of atomic accesses to the same storage location, if the location is in Memory Coherence Required storage, 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. The same applies if the location is in Caching Inhibited storage.

Because accesses to storage that is Caching Inhibited are performed in main storage, memory barriers and dependencies on *Load* instructions order such accesses with respect to any processor or mechanism even if the storage is not Memory Coherence Required.

Engineering Note
The correct operation of ***msync*** and ***mbar*** depends on both the processor and the storage subsystem.

The definition of memory barriers is not intended to preclude address pipelining. If two applicable *Storage Access* instructions are separated by ***msync*** or ***mbar***, it is permissible for the address associated with the second instruction to be presented to a given level of the storage hierarchy before the data access caused by the first instruction has completed at that level. However, if such pipelining is done, the processor must provide sufficient information so that the storage subsystem can keep the storage accesses in the correct order, and the storage subsystem must do so.

6.1.6.2 Atomic Update Primitives

The *Load And Reserve* and *Store Conditional* instructions together permit atomic update of a storage location. Book E provides word and doubleword forms of each of these instructions. Described here is the operation of ***lwarx*** and ***stwcx.***; operation of the other word forms, ***lwarxe*** and ***stwcxe.***, and the doubleword forms ***ldarxe*** and ***stdcxe.*** is the same except for obvious substitutions.

The specified storage location must be in storage that is Memory Coherence Required if the location may be modified by other processors or mechanisms. If the specified location is in storage that is Write-Through Required or Caching Inhibited, it is implementation-dependent whether these instructions function correctly or cause the system data storage error handler to be invoked.

Programming Note
The Memory Coherence Required attribute on other processors and mechanisms ensures that their stores to the specified storage location will cause the reservation created by the ***lwarx***, ***lwarxe***, or ***ldarxe*** to be lost.

Programming Note
Warning: Support for *Load and Reserve* and *Store Conditional* instructions for which the specified storage location is in storage that is Caching Inhibited 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 storage.

Engineering Note
For a given implementation, decisions regarding whether to support *Load and Reserve* and *Store Conditional* instructions that specify a Caching Inhibited storage location, and how well to make such instructions perform, must include consideration of migration plans for existing software that uses these instructions in this manner.

The ***lwarx*** instruction is a load from a word-aligned location that has two side effects.

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

The ***stwcx.*** instruction is a store to a word-aligned location that is conditioned on the existence of the reservation created by the ***lwarx*** and on whether the same storage location is specified by both instructions. To emulate an atomic operation with these instructions, it is necessary that both the ***lwarx*** and the ***stwcx.*** access the same storage 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 storage accesses caused by these two instructions.

Engineering Note
Both ***lwarx*** and ***stwcx.*** have a data dependence on the processor reservation resource.

A ***stwcx.*** performs a store to the target storage location only if the storage 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 storage locations specified by the two instructions differ the store is not necessarily performed. CR0 is set to indicate whether the store was performed.

If a ***stwcx.*** completes but does not perform the store because a reservation no longer exists, CR0 is set to indicate that the ***stwcx.*** completed but storage was not altered.

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

Examples of the use of ***lwarx*** and ***stwcx.*** are given in Appendix C on page 379.

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 storage called a *reservation granule*. The size of the reservation granule is implementation-dependent, but is a multiple of 4 bytes for ***lwarx*** and ***lwarxe***, and a multiple of 8 bytes for ***ldarxe***. The reservation granule associated with effective address EA contains the real address to which EA maps. ('real_addr(EA)' in the RTL for the *Load And Reserve* (page 290 and page 300) and *Store Conditional* instructions (page 342 and page 353) 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.

Programming Note
One use of ***lwarx*** and ***stwcx.*** is to emulate a 'Compare and Swap' primitive like that provided by the IBM System/370 Compare and Swap instruction: see Appendix C on page 379. A System/370-style Compare and Swap checks only that the old and current values of the word being tested are equal, with the result that programs that use such a Compare and Swap to control a shared resource can err if the word has been modified and the old value subsequently restored. The combination of ***lwarx*** and ***stwcx.*** improves on such a Compare and Swap, because the reservation reliably binds the ***lwarx*** and ***stwcx.*** together. The reservation is always lost if the word is modified by another processor or mechanism between the ***lwarx*** and ***stwcx.***, so the ***stwcx.*** never succeeds unless the word has not been stored into (by another processor or mechanism) since the ***lwarx***.

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

- 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***.

- Some other processor executes a *Store*, or ***dcbz***[***e***] to the same reservation granule.

- Some other processor executes a ***dcbtst***[***e***], ***dcbst***[***e***], or ***dcbf***[***e***] to the same reservation granule: whether the reservation is lost is undefined.

- Some other processor executes a ***dcba***[***e***] to the same 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 storage location in the same reservation granule.

Interrupts (see Chapter 7 on page 143) do not clear reservations (however, system software invoked by interrupts may clear reservations).

Programming Note
In general, programming conventions must ensure that ***lwarx*** and ***stwcx.*** specify addresses that match; a ***stwcx.*** should be paired with a specific ***lwarx*** to the same storage location. Situations in which a ***stwcx.*** may erroneously be issued after some ***lwarx*** other than that with which it is intended to be paired must be scrupulously avoided. For example, there must not be a context switch in which the processor holds a reservation in behalf of the old context, and the new context resumes after a ***lwarx*** and before the paired ***stwcx.***. The ***stwcx.*** in the new context might succeed, which is not what was intended by the programmer.

Such a situation must be prevented by 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 suffices to clear the reservation, whether it was obtained by ***lwarx***, ***lwarxe***, or ***ldarxe***.

Engineering Note
Reservations must take part in storage coherence. A reservation must be cleared if another processor receives authorization from the coherence mechanism to store to the reservation granule.

If an implementation continues to hold a reservation when the cache block in which the reservation lies is evicted, the reservation must continue to participate in the coherence protocol. In a snooping implementation, it must join in snooping. In a directory-based implementation, it must register its interest in the reserved block with the directory (shared-read access).

If an implementation demands that the reserved block be held in the cache, one way to satisfy the architectural requirements is the following. The implementation must be able to protect that block from eviction except by explicit invalidation (e.g., execution of ***dcbf***[***e***]) by the processor holding the reservation, and by cross-invalidates received from other processors, as long as the reservation persists. Caches in such an implementation must be sufficiently associative that the machine can continue to run with eviction of the reserved block inhibited.

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 that when a processor executes a ***lwarx*** to obtain a reservation for location X and then a ***stwcx.*** to store a value to location X, either

1. the ***stwcx.*** succeeds and the value is written to location X, or

2. the ***stwcx.*** fails because some other processor or mechanism modified location X, or

3. the ***stwcx.*** fails because the processor's reservation was lost for some other reason.

In Cases 1 and 2, the system as a whole makes progress in the sense that some processor successfully modifies location X. Case 3 covers reservation loss required for correct operation of the rest of the system. This includes cancellation caused by some other processor writing elsewhere in the reservation granule for X, as well as cancellation caused by the operating system in managing certain limited resources such as real memory. It may also include implementation-dependent causes of reservation loss.

An implementation may make a forward progress guarantee, defining the conditions under which the system as a whole makes progress. Such a guarantee must specify the possible causes of reservation loss in Case 3. While Book E alone cannot provide such a guarantee, the characteristics listed in Cases 1 and 2 are necessary conditions for any forward progress guarantee. An implementation and operating system can build on them to provide such a guarantee.

Architecture Note
Book E does not include a 'fairness guarantee.' 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 will lose 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 storage for locks to guarantee absence of granularity-induced reservation loss.

6.2 Storage Management

This section describes the address translation facility, access control, and storage attributes and control for Book E storage.

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. Translation misses and protection faults cause precise exceptions. Sufficient information is available to correct the fault and restart the faulting instruction.

Book E divides the effective address space into pages. The page represents the granularity of effective address translation, access control, and storage attributes. Up to sixteen page sizes (1KB, 4KB, 16KB, 64KB, 256KB, 1MB, 4MB, 16MB, 64MB, 256MB, 1GB, 4GB, 16GB, 64GB, 256GB, 1TB) may be simultaneously supported. In order for an effective to real translation to exist, a valid entry for the page containing the effective address must be in the Translation Lookaside Buffer (TLB). Addresses for which no TLB entry exists cause TLB Miss exceptions.

6.2.1 Storage Control Registers

In addition to the registers described below, the Machine State Register provides the IS and DS bits, that specify which of the two address spaces the respective instruction or data storage accesses are directed towards. MSR_{PR} bit is also used by the Book E storage access control mechanism.

6.2.1.1 Process ID Register

The Process ID Register (PID) is a 32-bit register. Process ID Register bits are numbered 32 (most-significant bit) to 63 (least-significant bit). The Process ID Register provides a value that is used to construct a virtual address for accessing storage.

The contents of bits 32:63 of the Process ID Register can be read into bits 32:63 of GPR(RT) using *mfspr RT,PID*, setting bits 0:31 of GPR(RT) to 0. The contents of bits 32:63 of GPR(RS) can be written into the Process ID Register using *mtspr RS,PID*. An implementation may opt to implement only the least-significant n bits of the Process ID Register, where $0 \le n \le 32$, and n must be the same as the number of implemented bits in the TID field of the TLB entry. The most-significant $32-n$ bits of the Process ID Register are treated as reserved. See the User's Manual for the implementation.

Some implementations may support more than one Process ID Register. See User's Manual for the implementation.

6.2.1.2 Translation Lookaside Buffer

The Translation Lookaside Buffer (TLB) is the hardware resource that controls translation, protection, and storage attributes. The organization of the TLB (e.g. unified versus separate instruction and data, hierarchies, associativity, number of entries, etc.) is implementation-dependent. Thus, the software for updating the TLB is also implementation-dependent. For the purposes of this discussion, a unified TLB organization is assumed. The differences for an implementation with separate instruction and data TLB's are for the most part obvious (e.g. separate

instructions or separate index ranges for reading, writing, searching, and invalidating each TLB). For details on how to synchronize TLB updates with instruction execution see Chapter 11 on page 225.

Maintenance of TLB entries is under software control. System software determines TLB entry replacement strategy and the format and use of any page state information. The TLB entry contains all the information required to identify the page, to specify the translation, to specify access controls, and to specify the storage attributes. The format of the TLB entry is implementation-dependent.

While the TLB is managed by software, Book E does not prohibit an implementation from implementing partial or full hardware assist for TLB management (e.g. support of PowerPC Architecture's virtual memory architecture). However, such implementations should be able to disable such support with implementation-dependent software or hardware configuration mechanisms.

A TLB entry is written by copying information from a GPR or other implementation-dependent source, using a series of ***tlbwe*** instructions (see page 366). A TLB entry is read by copying information to a GPR or other implementation-dependent target, using a series of ***tlbre*** instructions (see page 363). Software can also search for specific TLB entries using the ***tlbsx***[***e***] instruction (see page 364). Writing, reading and searching the TLB is implementation-dependent.

Each TLB entry describes a page that is eligible for translation and access controls. Fields in the TLB entry fall into four categories:

- Page identification fields (information required to identify the page to the hardware translation mechanism).
- Address translation fields
- Access control fields
- Storage attribute fields

While Book E requires the fields prescribed in Tables 6-1, 6-2, 6-3, and 6-4 to be implemented, no particular TLB entry format is formally specified. Book E does provide the ability to read or write portions of individual entries using the ***tlbre*** and ***tlbwe*** instructions.

Table 6-1. TLB Entry Page Identification Fields

Field	Description
V	**Valid** (1 bit) This bit indicates that this TLB entry is valid and may be used for translation. The Valid bit for a given entry can be set or cleared with a ***tlbwe*** instruction; alternatively, the Valid bit for an entry may be cleared by a ***tlbivax***[***e***] instruction.
EPN	**Effective Page Number** (54 bits) Bits 0:*n*–1 of the EPN field are compared to bits 0:*n*–1 of the effective address (EA) of the storage access (where *n*=64–log2(*page size* in bytes) and *page size* is specified by the SIZE field of the TLB entry). See Table 6-5.
	Note Implementations may implement bits N:53 of the EPN field, where N≥0. See User's Manual for the implementation.

Field	Description
TS	**Translation Address Space** (1 bit) This bit indicates the address space this TLB entry is associated with. For instruction storage accesses, MSR_{IS} must match the value of TS in the TLB entry for that TLB entry to provide the translation. Likewise, for data storage accesses, MSR_{DS} must match the value of TS in the TLB entry. For ***tlbsx***[***e***] and ***tlbivax***[***e***] instructions, an implementation-dependent source provides the address space specification that must match the value of TS.
SIZE	**Page Size** (4 bits) The SIZE field specifies the size of the page associated with the TLB entry as 4^{SIZE}KB, where SIZE ∈ {0, 1, ..., 15}. Implementations may implement any one or more of these page sizes. See Table 6-5.
TID	**Translation ID** (implementation-dependent size) Field used to identify a shared page (TID=0) or the owner's process ID of a private page (TID≠0). See Section 6.2.2.

Table 6-2. TLB Entry Translation Field

Field	Description
RPN	**Real Page Number** (up to 54 bits) Bits 0:*n*–1 of the RPN field are used to replace bits 0:*n*–1 of the effective address to produce the real address for the storage access (where *n*=64–log2(*page size* in bytes) and *page size* is specified by the SIZE field of the TLB entry). Software must set unused low-order RPN bits (i.e. bits *n*:53) to 0. See Section 6.2.3.
	Note Implementations may implement bits M:53 of the RPN field, where M≥0. See User's Manual for the implementation.

Table 6-3. TLB Entry Access Control Fields

Bit	Description	
UX	**User State Execute Enable** (1 bit) See Section 6.2.4.1.	
	=0	Instruction fetch and execution is not permitted from this page while MSR_{PR}=1 and will cause an Execute Access Control exception type Instruction Storage interrupt.
	=1	Instruction fetch and execution is permitted from this page while MSR_{PR}=1.
SX	**Supervisor State Execute Enable** (1 bit) See Section 6.2.4.1.	
	=0	Instruction fetch and execution is not permitted from this page while MSR_{PR}=0 and will cause an Execute Access Control exception type Instruction Storage interrupt.
	=1	Instruction fetch and execution is permitted from this page while MSR_{PR}=0.
UW	**User State Write Enable** (1 bit) See Section 6.2.4.2.	
	=0	*Store* operations, including ***dcba***[***e***] and ***dcbz***[***e***], are not permitted to this page when MSR_{PR}=1 and will cause a Write Access Control exception. Except as noted in Table 6-7 on page 131, a Write Access Control exception will cause a Data Storage interrupt.
	=1	*Store* operations, including ***dcba***[***e***] and ***dcbz***[***e***], are permitted to this page when MSR_{PR}=1.
SW	**Supervisor State Write Enable** (1 bit) See Section 6.2.4.2.	
	=0	*Store* operations, including ***dcba***[***e***], ***dcbi***[***e***], and ***dcbz***[***e***], are not permitted to this page when MSR_{PR}=0. *Store* operations, including ***dcbi***[***e***] and ***dcbz***[***e***], will cause a Write Access Control exception. Except as noted in Table 6-7 on page 131, a Write Access Control exception will cause a Data Storage interrupt.
	=1	*Store* operations, including ***dcba***[***e***], ***dcbi***[***e***], and ***dcbz***[***e***], are permitted to this page when MSR_{PR}=0.
UR	**User State Read Enable** (1 bit) See Section 6.2.4.3.	
	=0	*Load* operations (including load-class *Cache Management* instructions) are not permitted from this page when MSR_{PR}=1 and will cause a Read Access Control exception. Except as noted in Table 6-7 on page 131, a Read Access Control exception will cause a Data Storage interrupt.
	=1	*Load* operations (including load-class *Cache Management* instructions) are permitted from this page when MSR_{PR}=1.
SR	**Supervisor State Read Enable** (1 bit) See Section 6.2.4.3.	
	=0	*Load* operations (including load-class *Cache Management* instructions) are not permitted from this page when MSR_{PR}=0 and will cause a Read Access Control exception. Except as noted in Table 6-7 on page 131, a Read Access Control exception will cause a Data Storage interrupt.
	=1	*Load* operations (including load-class *Cache Management* instructions) are permitted from this page when MSR_{PR}=0.

Table 6-4. TLB Entry Storage Attribute Bits

Bit(s)	Description	
W	**Write-Through Required** (1 bit) See Section 6.2.5.1.	
	=0	The page is not Write-Through Required.
	=1	The page is Write-Through Required.
I	**Caching Inhibited** (1 bit) See Section 6.2.5.2.	
	=0	The page is not Caching Inhibited.
	=1	The page is Caching Inhibited.

Bit(s)	Description	
M	**Memory Coherence Required** (1 bit) See Section 6.2.5.3.	
	=0	The page is not Memory Coherence Required.
	=1	The page is Memory Coherence Required.
	Hardware support for Memory Coherence Required storage is optional for implementations that do not support multiprocessing. If the implementation does not support this storage attribute, then all storage access will behave as if M=0, and setting M=1 in a TLB entry will have no effect.	
G	**Guarded** (1 bit) See Section 6.2.5.4 and Section 6.1.5.	
	=0	The page is not Guarded.
	=1	The page is Guarded.
E	**Endianness** (1 bit) See Section 6.2.5.5.	
	=0	All accesses to the page are performed in a big-endian fashion, which means that, for all multiple-byte scalar accesses, the byte at the lowest numbered address is treated as the most-significant byte.
	=1	All accesses to the page are performed in a little-endian fashion, which means that, for all multiple-byte scalar accesses, the byte at the lowest numbered address is treated as the least-significant byte
U0-U3	**User-Definable Storage Attributes** (4 bits) See Section 6.2.5.6. Specifies implementation-dependent and system-dependent storage attributes for the page associated with the TLB entry. See the User's Manual for the implementation.	

6.2.2 Page Identification

Instruction effective addresses are generated for sequential instruction fetches and for addresses that correspond to a change in program flow (branches, interrupts). Data effective addresses are generated by *Load*, *Store*, and *Cache Management* instructions. *TLB Management* generate effective addresses to determine the presence of or to invalidate a specific TLB entry associated with that address.

The Valid (V) bit, Effective Page Number (EPN) field, Translation Space Identifier (TS) bit, Page Size (SIZE) field, and Translation ID (TID) field of a particular TLB entry identify the page associated with that TLB entry. Except as noted, all comparisons must succeed to validate this entry for subsequent translation and access control processing. Failure to locate a matching TLB entry based on this criteria for instruction fetches will result in an Instruction TLB Miss exception type Instruction TLB Error interrupt. Failure to locate a matching TLB entry based on this criteria for data storage accesses will result in a Data TLB Miss exception which may result in a Data TLB Error interrupt. Figure 6-1 on page 127 illustrates the criteria for a virtual address to match a specific TLB entry.

There are two address spaces, one typically associated with interrupt-related storage accesses and one typically associated with non-interrupt-related storage accesses. There are two bits in the Machine State Register, the Instruction Address Space bit (IS) and the Data Address Space bit (DS), that control which address space instruction and data storage accesses, respectively, are performed in, and a bit in the TLB entry (TS) that specifies which address space that TLB entry is associated with.

Load, *Store*, *Cache Management*, *Branch*, ***tlbsx***[***e***], and ***tlbivax***[***e***] instructions and next-sequential-instruction fetches produce a 64-bit effective address. The virtual address space is extended from this 64-bit effective address space by prepending a one-bit address space identifier and a process identifier. For instruction fetches, the address space identifier is provided by MSR_{IS} and the process identifier is provided by the contents of the Process ID Register. For data storage accesses, the

address space identifier is provided by the MSR_{DS} and the process identifier is provided by the contents of the Process ID Register. For ***tlbsx***[***e***], and ***tlbivax***[***e***] instructions, the address space identifier and the process identifier are provided by implementation-dependent sources.

This virtual address is used to locate the associated entry in the TLB. The address space identifier, the process identifier, and the effective address of the storage access are compared to the Translation Address Space bit (TS), the Translation ID field (TID), and the value in the Effective Page Number field (EPN), respectively, of each TLB entry.

The virtual address of a storage access matches a TLB entry if:

for every TLB entry i in the congruence class specified by EA:

- the value of the address specifier for the storage access (MSR_{IS} for instruction fetches, MSR_{DS} for data storage accesses, and implementation-dependent source for ***tlbsx***[***e***] and ***tlbivax***[***e***]) is equal to the value of the TS bit of the TLB entry, and

- either the value of the process identifier (Process ID Register for instruction and data storage accesses, and implementation-dependent source for ***tlbsx***[***e***] and ***tlbivax***[***e***]) is equal to the value in the TID field of the TLB entry, or the value of the TID field of the TLB entry is equal to 0, and

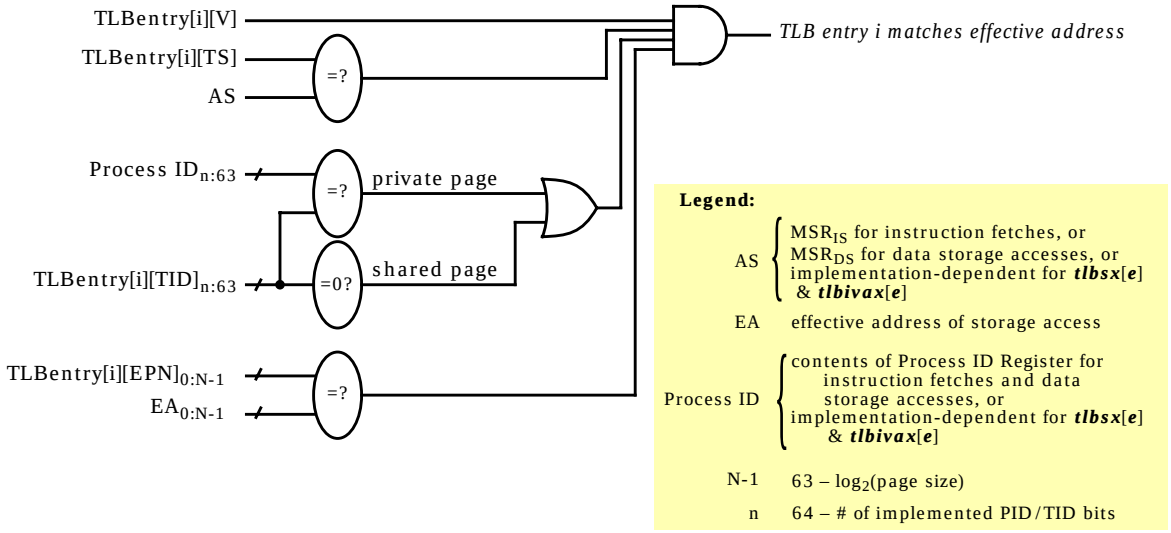
- the contents of bits 0:*n*–1 of the effective address of the storage or TLB access are equal to the value of bits 0:*n*-1 of the EPN field of the TLB entry (where n=64-$\log_2$(*page size* in bytes) and *page size* is specified by the value of the SIZE field of the TLB entry). See Table 6-5.

A TLB Miss exception occurs if there is no valid entry in the TLB for the page specified by the virtual address (Instruction or Data TLB Error interrupt). Although the possibility to place multiple entries into the TLB that match a specific virtual address exists, assuming a set-associative or fully-associative organization, doing so is a programming error and the results are undefined.

Table 6-5. Page Size and Effective Address to EPN Comparison

SIZE	Page Size (4^{SIZE}KB)	EA to EPN Comparison (bits 0:53–2×SIZE)
=0b0000	1KB	$EPN_{0:53}$ =? $EA_{0:53}$
=0b0001	4KB	$EPN_{0:51}$ =? $EA_{0:51}$
=0b0010	16KB	$EPN_{0:49}$ =? $EA_{0:49}$
=0b0011	64KB	$EPN_{0:47}$ =? $EA_{0:47}$
=0b0100	256KB	$EPN_{0:45}$ =? $EA_{0:45}$
=0b0101	1MB	$EPN_{0:43}$ =? $EA_{0:43}$
=0b0110	4MB	$EPN_{0:41}$ =? $EA_{0:41}$
=0b0111	16MB	$EPN_{0:39}$ =? $EA_{0:39}$
=0b1000	64MB	$EPN_{0:37}$ =? $EA_{0:37}$
=0b1001	256MB	$EPN_{0:35}$ =? $EA_{0:35}$
=0b1010	1GB	$EPN_{0:33}$ =? $EA_{0:33}$
=0b1011	4GB	$EPN_{0:31}$ =? $EA_{0:31}$
=0b1100	16GB	$EPN_{0:29}$ =? $EA_{0:29}$
=0b1101	64GB	$EPN_{0:27}$ =? $EA_{0:27}$
=0b1110	256GB	$EPN_{0:25}$ =? $EA_{0:25}$
=0b1111	1TB	$EPN_{0:23}$ =? $EA_{0:23}$

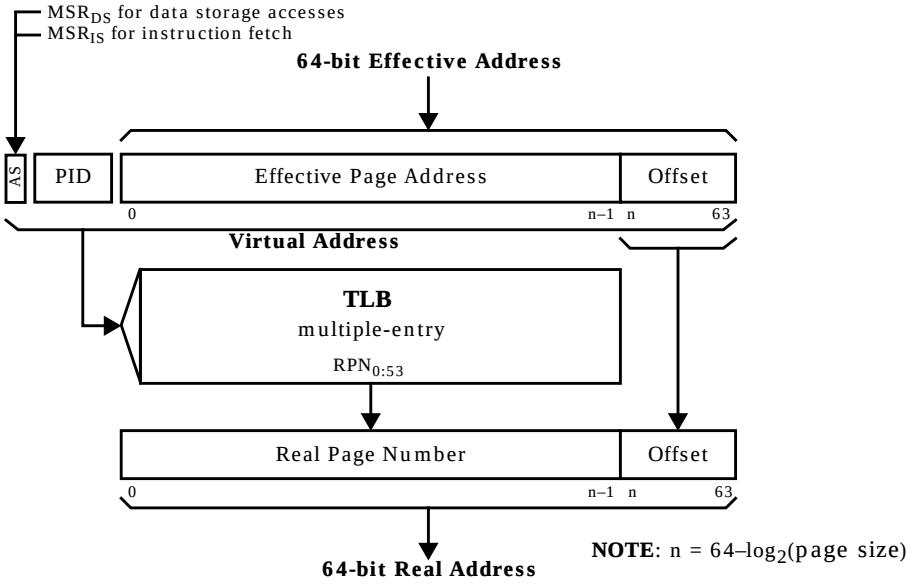
Figure 6-1. Virtual Address to TLB Entry Match Process



6.2.3 Address Translation

A program references memory by using the effective address computed by the processor when it executes a *Load*, *Store*, *Cache Management*, or *Branch* instruction, and when it fetches the next instruction. The effective address is translated to a real address according to the procedures described in this section. The storage subsystem uses the real address for the access. All storage access effective addresses are translated to real addresses using the TLB mechanism. See Figure 6-2.

Figure 6-2. Effective-to-Real Address Translation Flow



If the virtual address of the storage access matches a TLB entry in accordance with the selection criteria specified in Section 6.2.2, the value of the Real Page Number field (RPN) of the selected TLB entry provides the real page number portion of the real address. Let n=64–log$_2$(*page size* in bytes) where *page size* is specified by the SIZE field of the TLB entry. Bits n:63 of the effective address are appended to bits 0:n–1 of the 54-bit RPN field of the selected TLB entry to produce the 64-bit real address (i.e. RA = $RPN_{0:n-1}$ || $EA_{n:63}$). The page size is determined by the value of the SIZE field of the selected TLB entry. See Table 6-6.

The rest of the selected TLB entry provides the access control bits (UX, SX, UW, SW, UR, SR), and storage attributes (U0, U1, U2, U3, W, I, M, G, E) for the storage access. The access control bits and storage attribute bits specify whether or not the access is allowed and how the access is to be performed. See Sections 6.2.4 and 6.2.5.

The Real Page Number field (RPN) of the matching TLB entry provides the translation for the effective address of the storage access. Based on the setting of the SIZE field of the matching TLB entry, the RPN field replaces the corresponding most-significant N bits of the effective address (where N = 64 – log$_2$(page size)), as shown in Figure 6-6, to produce the 64-bit real address that is to be presented to main storage to perform the storage access.

Table 6-6. Effective Address to Real Address

SIZE	Page Size (4^{SIZE}KB)	RPN Bits Required to be Equal to 0	Real Address
=0b0000	1KB	none	$RPN_{0:53} \| EA_{54:63}$
=0b0001	4KB	$RPN_{52:53}$=0	$RPN_{0:51} \| EA_{52:63}$
=0b0010	16KB	$RPN_{50:53}$=0	$RPN_{0:49} \| EA_{50:63}$
=0b0011	64KB	$RPN_{48:53}$=0	$RPN_{0:47} \| EA_{48:63}$
=0b0100	256KB	$RPN_{46:53}$=0	$RPN_{0:45} \| EA_{46:63}$
=0b0101	1MB	$RPN_{44:53}$=0	$RPN_{0:43} \| EA_{44:63}$
=0b0110	4MB	$RPN_{42:53}$=0	$RPN_{0:41} \| EA_{42:63}$
=0b0111	16MB	$RPN_{40:53}$=0	$RPN_{0:39} \| EA_{40:63}$
=0b1000	64MB	$RPN_{38:53}$=0	$RPN_{0:37} \| EA_{38:63}$
=0b1001	256MB	$RPN_{36:53}$=0	$RPN_{0:35} \| EA_{36:63}$
=0b1010	1GB	$RPN_{34:53}$=0	$RPN_{0:33} \| EA_{34:63}$
=0b1011	4GB	$RPN_{32:53}$=0	$RPN_{0:31} \| EA_{32:63}$
=0b1100	16GB	$RPN_{30:53}$=0	$RPN_{0:29} \| EA_{30:63}$
=0b1101	64GB	$RPN_{28:53}$=0	$RPN_{0:27} \| EA_{28:63}$
=0b1110	256GB	$RPN_{26:53}$=0	$RPN_{0:25} \| EA_{26:63}$
=0b1111	1TB	$RPN_{24:53}$=0	$RPN_{0:23} \| EA_{24:63}$

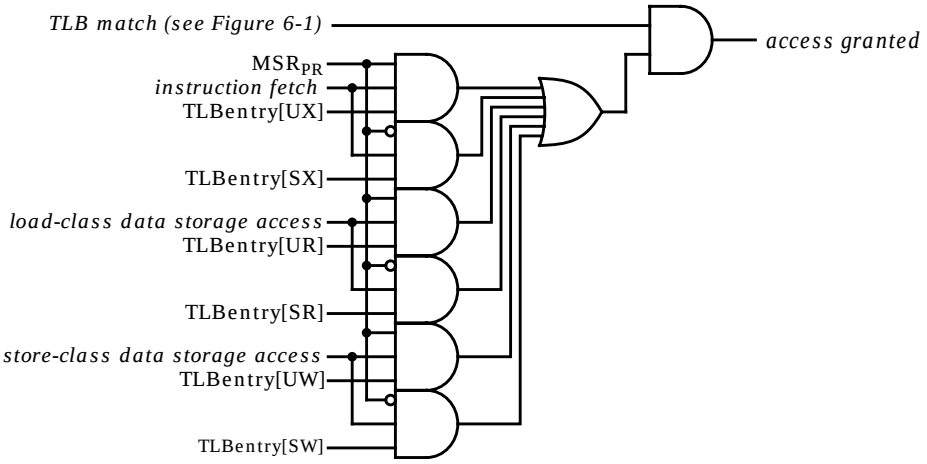
6.2.4 Storage Access Control

After a matching TLB entry has been identified, Book E provides an access control mechanism for selectively granting shared access, granting execute access, granting read access, granting write access, and prohibiting access to areas of storage based on a number of criteria. Figure 6-3 illustrates the access control process and is described in detail in Sections 6.2.4.1, 6.2.4.2, 6.2.4.3, 6.2.4.4, and 6.2.4.5.

An Execute, Read, or Write Access Control exception occurs if the appropriate TLB entry is found but the access is not allowed by the access control mechanism (Instruction or Data Storage interrupt). See Section 7.6 for additional information about these and other interrupt types. In certain cases, Execute, Read, and Write Access Control exceptions may result in the restart of (re-execution of at least part of) a *Load* or *Store* instruction.

Some implementation may provide additional access control capabilities beyond that described here. See the User's Manual for the implementation.

Figure 6-3. Access Control Process



6.2.4.1 Execute Access

The UX and SX bits of the TLB entry control *execute* access to the page (see Table 6-3 on page 124).

Instructions may be fetched and executed from a page in storage while in user state (MSR_{PR}=1) if the UX access control bit for that page is equal to 1. If the UX access control bit is equal to 0, then instructions from that page will not be fetched, and will not be placed into any cache as the result of a fetch request to that page while in user state.

Instructions may be fetched and executed from a page in storage while in supervisor state (MSR_{PR}=0) if the SX access control bit for that page is equal to 1. If the SX access control bit is equal to 0, then instructions from that page will not be fetched, and will not be placed into any cache as the result of a fetch request to that page while in supervisor state.

Furthermore, if the sequential execution model calls for the execution of an instruction from a page that is not enabled for execution (i.e. UX=0 when MSR_{PR}=1 or SX=0 when MSR_{PR}=0), an Execute Access Control exception type Instruction Storage interrupt is taken.

6.2.4.2 Write Access

The UW and SW bits of the TLB entry control *write* access to the page (see Table 6-3 on page 124).

Store operations (including *Store*-class *Cache Management* instructions) are permitted to a page in storage while in user state (MSR_{PR}=1) if the UW access control bit for that page is equal to 1. If the UW access control bit is equal to 0, then execution of the *Store* instruction is suppressed and a Write Access Control exception type Data Storage interrupt is taken.

Store operations (including *Store*-class *Cache Management* instructions) are permitted to a page in storage while in supervisor state (MSR_{PR}=0) if the SW access control bit for that page is equal to 1. If the SW access control bit is equal to 0, then execution of the *Store* instruction is suppressed and a Write Access Control exception type Data Storage interrupt is taken.

6.2.4.3 Read Access

The UR and SR bits of the TLB entry control *read* access to the page (see Table 6-3 on page 124).

Load operations (including *Load*-class *Cache Management* instructions) are permitted from a page in storage while in user state (MSR_{PR}=1) if the UR access control bit for that page is equal to 1. If the UR access control bit is equal to 0, then execution of the *Load* instruction is suppressed and a Read Access Control exception type Data Storage interrupt is taken.

Load operations (including *Load*-class *Cache Management* instructions) are permitted from a page in storage while in supervisor state (MSR_{PR}=0) if the SR access control bit for that page is equal to 1. If the SR access control bit is equal to 0, then execution of the *Load* instruction is suppressed and a Read Access Control exception type Data Storage interrupt is taken.

6.2.4.4 Storage Access Control Applied to Cache Management Instructions

dcbi[***e***] and ***dcbz***[***e***] instructions are treated as *Stores* since they can change data (or cause loss of data by invalidating a dirty line). As such, they both can cause Write Access Control exception type Data Storage interrupts.

dcba[***e***] instructions are treated as *Stores* since they can change data. As such, they can cause Write Access Control exceptions. However, such exceptions will not result in a Data Storage interrupt.

icbi[***e***] instructions are treated as *Loads* with respect to protection. As such, they can cause Read Access Control exception type Data Storage interrupts.

dcbt[***e***], ***dcbtst***[***e***], and ***icbt***[***e***] instructions are treated as *Loads* with respect to protection. As such, they can cause Read Access Control exceptions. However, such exceptions will not result in a Data Storage interrupt.

dcbf[***e***] and ***dcbst***[***e***] instructions are treated as *Loads* with respect to protection. Flushing or storing a line from the cache is not considered a *Store* since the store has already been done to update the cache and the ***dcbf***[***e***] or ***dcbst***[***e***] instruction is only updating the copy in main storage. As a *Load*, they can cause Read Access Control exception type Data Storage interrupts.

Table 6-7. Storage Access Control Applied to Cache Instructions

Instruction	Read Protection Violation Exception?	Write Protection Violation Exception?
dcba[***e***]	No	Yes[2]
dcbf[***e***]	Yes	No
dcbi[***e***]	No	Yes
dcbst[***e***]	Yes	No
dcbt[***e***]	Yes[1]	No
dcbtst[***e***]	Yes[1]	No
dcbz[***e***]	No	Yes
icbi[***e***]	Yes	No
icbt[***e***]	Yes[1]	No

1.***dcbt***[***e***], ***dcbtst***[***e***], & ***icbt***[***e***] may cause a Read Access Control exception but does not result in a Data Storage interrupt
2.***dcba***[***e***] may cause a Write Access Control exception but does not result in a Data Storage interrupt

6.2.4.5 Storage Access Control Applied to String Instructions

When the string length is zero, neither ***lswx*** nor ***stswx*** can cause Data Storage interrupts.

6.2.5 Storage Attributes

Some operating systems may provide a means to allow programs to specify the storage attributes described in this section. Because the support provided for these attributes by the operating system may vary between systems, the details of the specific system being used must be known before these attributes can be used.

Storage attributes are associated with units of storage called pages. Each storage access is performed according to the storage control attributes of the specified storage location, as described below. The storage control attributes are the following.

- Write-Through Required (W)
- Caching Inhibited (I)
- Memory Coherence Required (M)
- Guarded (G)
- Endianness (E)
- User-Definable (U0, U1, U2, U3)

The W, I, M, G, E, U0, U1, U2, and U3 bits in the TLB entry control the way in which the processor performs storage accesses in the page associated with the TLB entry.

All combinations of these attributes are supported except combinations which simultaneously specify a region as Write-Through Required and Caching Inhibited.

Programming Note
The Write-Through Required and Caching Inhibited attributes are mutually exclusive because, as described below, the Write-Through Required attribute permits the storage location to be in the data cache while the Caching Inhibited attribute does not.

Storage that is Write-Through Required or Caching Inhibited is not intended to be used for general-purpose programming. For example, the ***lwarx***[***e***], ***ldarxe***, ***stwcx***[***e***]**.** and ***stdcxe.*** instructions may cause the system data storage error handler to be invoked if they specify a location in storage having either of these attributes; see Section 6.1.6.2.

In the remainder of this chapter, '*Load* instruction' includes the *Cache Management* and other instructions that are stated in the instruction descriptions to be 'treated as a *Load*', and similarly for '*Store* instruction'.

6.2.5.1 Write-Through Required

A store to a Write-Through Required (W) storage location is performed in main storage. A *Store* instruction that specifies a location in Write-Through Required storage may cause additional locations in main storage to be accessed. If a copy of the block containing the specified location is retained in the data cache, the store is also performed in the data cache. The store does not cause the block to be considered to be modified in the data cache.

If some *Store* instructions executed by a given processor access locations in a block as Write-Through Required and other *Store* instructions executed by the same processor access locations in the block as not Write-Through Required, software must ensure that the block is not in storage that is accessed by another processor or mechanism. Also, if a *Store* instruction that accesses a location in the block as Write-Through Required 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 storage.

In general, accesses caused by separate *Store* instructions that specify locations in Write-Through Required storage may be combined into one access. Such combining does not occur if the *Store* instructions are separated by an ***msync*** instruction or by an ***mbar*** instruction.

6.2.5.2 Caching Inhibited

An access to a Caching Inhibited (I) storage location is performed in main storage. A *Load* instruction that specifies a location in Caching Inhibited storage may cause additional locations in main storage to be accessed unless the specified location is also Guarded. An instruction fetch from Caching Inhibited storage may cause additional words in main storage to be accessed. No copy of the accessed locations is placed into the caches.

In general, non-overlapping accesses caused by separate *Load* instructions that specify locations in Caching Inhibited storage may be combined into one access, as may non-overlapping accesses caused by separate *Store* instructions that specify locations in Caching Inhibited storage. Such combining does not occur if the *Load* or *Store* instructions are separated by an ***msync*** instruction, or by an ***mbar*** instruction if the storage is also Guarded.

6.2.5.3 Memory Coherence Required

An access to a Memory Coherence Required (M) storage location is performed coherently, as follows.

> Memory coherence refers to the ordering of stores to a single location. Atomic stores to a given location are *coherent* if they are serialized in some order, and no processor or mechanism is able to observe any subset of those stores as occurring in a conflicting order. This serialization order is an abstract sequence of values; the physical storage location need not assume each of the values written to it. For example, a processor may update a location several times before the value is written to physical storage. The result of a store operation is not available to every processor or mechanism at the same instant, and it may be that a processor or mechanism observes only some of the values that are written to a location. However, when a location is accessed atomically and coherently by all processor and mechanisms, the sequence of values loaded from the location by any processor or mechanism during any interval of time forms a subsequence of the sequence of values that the location logically held during that interval. That is, a processor or mechanism can never load a 'newer' value first and then, later, load an 'older' value.

Memory coherence is managed in blocks called *coherence blocks*. Their size is implementation-dependent (see the User's Manual for the implementation), but is usually larger than a word and often the size of a cache block.

For storage that is not Memory Coherence Required, software must explicitly manage memory coherence to the extent required by program correctness. The operations required to do this may be system-dependent.

Because the Memory Coherence Required attribute for a given storage location is of little use unless all processors that access the location do so coherently, in statements about Memory Coherence Required storage elsewhere in this document it is generally assumed that the storage has the Memory Coherence Required attribute for all processors that access it.

Programming Note
Operating systems that allow programs to request that storage not be Memory Coherence Required should provide services to assist in managing memory coherence for such storage, including all system-dependent aspects thereof.

In most systems the default is that all storage is Memory Coherence Required. For some applications in some systems, software management of coherence may yield better performance. In such cases, a program can request that a given unit of storage not be Memory Coherence Required, and can manage the coherence of that storage by using the ***msync*** instruction, the *Cache Management* instructions, and services provided by the operating system.

Engineering Note
Memory coherence can be implemented, for example, by an ownership protocol that allows at most one processor at a time to store to a given location in Memory Coherence Required storage.

A processor observing a storage access initiated by another processor or mechanism must honor the coherence requirements of that access, even if the observing processor last accessed the affected storage location as not Memory Coherence Required.

The ability to disable Memory Coherence Required is provided (see Table 6-4 on page 124) to allow improved performance in systems in which accesses to storage kept consistent by hardware are slower than accesses to storage not kept consistent by hardware, and in which software is able to enforce the required consistency. When the Storage attribute is off (M=0), the hardware need not enforce data coherence for storage accesses initiated by the processor. When the Storage attribute is on (M=1), the hardware must enforce data coherence for storage accesses initiated by the processor.

When an access is performed for which data coherence is required, the processor performing the access must inform the coherence mechanism that the access requires memory coherence. Other processors affected by the access must respond to the coherence mechanism. However since the mode control bits have no direct relation to data or instructions in the cache, processors responding to the coherence request are able to respond without knowledge of the state of this bit. Because instruction storage need not be consistent with data storage, it is permissible for an implementation to ignore the M bit for instruction fetches.

System Note
Entities other than processors can request that their memory transactions obey memory coherence.

Engineering Note
Treating instruction fetches as non-coherent can result in better performance in an implementation in which a coherent storage request has greater latency or overhead than a non-coherent storage request.

Hardware support for Memory Coherence Required storage attribute is optional for implementations that do not support multiprocessing.

6.2.5.4 Guarded

Storage is said to be 'well-behaved' if the corresponding real storage exists and is not defective, and if the effects of a single access to it are indistinguishable from the effects of multiple identical accesses to it. Data and instructions can be fetched out-of-order from well-behaved storage without causing undesired side effects.

A data access to a Guarded storage location is performed only if either the access is caused by an instruction that is known to be required by the sequential execution model, or the access is a load and the storage location is already in a cache. If the storage is also Caching Inhibited, only the storage location specified by the instruction is accessed; otherwise any storage location in the cache block containing the specified storage location may be accessed.

Instruction fetch is not affected by Guarded storage. While Book E does not prevent instructions from being fetched out-of-order from Guarded storage, system software should prevent all instruction fetching from Guarded storage by making Guarded pages 'no-execute' (see Table 6-4 on page 124). Then, if the effective address of the current instruction is in such storage, an Execute Access Control type Instruction Storage interrupt is invoked.

In general, storage that is not well-behaved should be Guarded. Because such storage may represent a control register on an I/O device or may include locations that do not exist, an out-of-order access to such storage may cause an I/O device to perform unintended operations or may result in a Machine Check.

Programming Note
In some implementations, instructions may be executed before they are known to be required by the sequential execution model. Because the results of instructions executed in this manner are discarded if it is later determined that those instructions would not have been executed in the sequential execution model, this behavior does not affect most programs.

This behavior does affect programs that access storage locations that are not 'well-behaved' (e.g., a storage location that represents a control register on an I/O device that, when accessed, causes the device to perform an operation). To avoid unintended results, programs that access such storage locations should request that the storage be Guarded, and should prevent such storage locations from being in a cache (e.g., by requesting that the storage also be Caching Inhibited).

If an aligned, elementary load or store to storage that is both Caching Inhibited and Guarded has accessed main storage and an asynchronous or imprecise mode Floating-Point Enabled exception is pending, the *Load* or *Store* instruction completes before the interrupt occurs.

Architecture Note
The rules for accessing Guarded storage when an Imprecise mode Floating-Point Enabled exception is pending should be revisited when Book E is clarified with respect to those modes. For example, it may be acceptable to require software synchronization between any instruction that could cause a floating-point enabled exception in Imprecise mode and a subsequent instruction that accesses Guarded storage. (A *Floating-Point Status and Control Register* instruction might provide sufficient synchronization.)

6.2.5.5 Endianness

Objects may be loaded from or stored to memory in byte, halfword, word, or doubleword units. For a particular data length, the loading and storing operations are symmetric; a store followed by a load of the same data object will yield an unchanged value. There is no information in the process about the order in which the bytes which comprise the multiple-byte data object are stored in memory.

Big Endian

If a stored multiple-byte object is probed by reading its component bytes one at a time using load-byte instructions, then the storage 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, then the multiple-byte object is stored in Big Endian form.

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 storage, 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 storage, starting at the lowest numbered address.

Little Endian

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, then the multiple-byte object is stored in Little Endian form.

6.2.5.6 User-Definable

User-definable storage attributes control user-definable and implementation-dependent behavior of the storage system. These bits are both implementation-dependent and system-dependent in their effect. They may be used in any combination and also in combination with the other storage attribute bits. See User's Manual for the implementation.

6.2.5.7 Supported Storage Attribute Combinations

Support for M=1 storage is optional. Storage modes where both W=1 and I=1 (which would represent Write-Through Required but Caching Inhibited storage) are not supported. For all supported combinations of the W, I and M bits, both G and E may be 0 or 1.

Engineering Note
Some implementations may only support the Endianness storage attribute in a static manner and may require software assistance as well as a context-synchronizing event between successive accesses to little-endian and big-endian storage (e.g. support the Endianness storage attribute as a static mode). These implementations may rely on the Byte Ordering exception type Data Storage or Instruction Storage interrupt to switch between 'little-endian mode' and 'big-endian mode'.

6.2.5.8 Mismatched Storage Attributes

Accesses to the same storage location using two effective addresses for which the Write-Through Required storage attribute (W bit) differs meet the memory coherence requirements described in Section 6.2.5.3, 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 Required storage attribute is the same for all the accesses.

> **Engineering Note**
> If an implementation uses a 'MESI' coherence protocol, a store addressed to a write-through page may find the addressed cache block in the cache and modified. If so, the store should update the location in both the cache block and main storage (the normal operation of a store to Write-Through Required storage). It is acceptable for the implementation to write the block back to main storage, in which case it can change the state to 'unmodified.' It is also acceptable for the implementation to leave the state of the cache block 'modified' after updating the location in cache and main storage.

Loads, *Stores*, ***dcbz***[***e***] instructions, and instruction fetches to the same storage location using two effective addresses for which the Caching Inhibited storage attribute (I bit) differs must meet the requirement that a copy of the target location of an access to Caching Inhibited storage 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 storage is in the cache.

Accesses to the same storage location using two effective addresses for which the Guarded storage attribute (G bit) differs are always permitted.

Except for instruction fetches, accesses to the same storage location using two effective addresses for which the endian storage attribute (E bit) differs are always permitted. Instruction storage locations must be flushed before the endian storage attribute can be changed for those addresses.

The specification of mismatched user storage attributes (U0 through U3) is implementation-dependent. See the User's Manual for the implementation.

Accesses to the same storage location using two effective addresses for which the memory coherence Storage 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 'snooping bus' based systems no software synchronization may be required. In some 'directory based' systems, software may be required to execute ***dcbf***[***e***] instructions on each processor to flush all storage locations accessed with M=0 before accessing those locations with M=1.

6.2.6 TLB Management

Book E does not imply any format for the page tables or the page table entries. Software has significant flexibility in implementing a custom replacement strategy. For example, software may choose to lock TLB entries that correspond to frequently used storage, so that those entries are never cast out of the TLB and TLB Miss exceptions to those pages never occur. At a minimum, software must maintain an entry or entries for the Instruction and Data TLB Error interrupt handlers.

TLB management is performed in software with some hardware assist. This hardware assist consists of a minimum of:

- Automatic recording of the effective address causing a TLB Miss exception. For Instruction TLB Miss exceptions, the address is saved in the Save/Restore Register 0. For Data TLB Miss exceptions, the address is saved in the Data Exception Address Register.

- Instructions for reading, writing, searching, invalidating, and synchronizing the TLB (see Section 6.3.3).

Programming Note
This note suggests one example for managing reference and change recording in a Book E system.

When performing physical page management, it is useful to know whether a given physical page has been referenced or altered. Note that this may be more involved than whether a given TLB entry has been used to reference or alter memory, since multiple TLB entries may translate to the same physical page. If it is necessary to replace the contents of some physical page with other contents, a page which has been referenced (accessed for any purpose) is more likely to be maintained than a page which has never been referenced. If the contents of a given physical page are to be replaced, then the contents of that page must be written to the backing store before replacement, if anything in that page has been changed. Software must maintain records to control this process.

Similarly, when performing TLB management, it is useful to know whether a given TLB entry has been referenced. When making a decision about which entry to cast-out of the TLB, an entry which has been referenced is more likely to be maintained in the TLB than an entry which has never been referenced.

Execute, Read and Write Access Control exceptions may be used to allow software to maintain reference information for a TLB entry and for its associated physical page. The entry is built, with its UX, SX, UR, SR, UW, and SW bits off, and the index and effective page number of the entry retained by software. The first attempt of application code to use the page will cause an Access Control exception (because the entry is marked 'No Execute', 'No Read', and 'No Write'). The Instruction or Data Storage interrupt handler records the reference to the TLB entry and to the associated physical page in a software table, and then turns on the appropriate access control bit. An initial read from the page could be handled by only turning on the appropriate UR or SR access control bits, leaving the page 'read-only'. Subsequent execute, read, or write accesses to the page via this TLB entry will proceed normally.

In a demand-paged environment, when the contents of a physical page are to be replaced, if any storage in that physical page has been altered, then the backing storage must be updated. The information that a physical page is dirty is typically recorded in a 'change' bit for that page.

Write Access Control exceptions may be used to allow software to maintain change information for a physical page. For the example just given for reference recording, the first write access to the page via the TLB entry will create a Write Access Control exception type Data Storage interrupt. The Data Storage interrupt handler records the change status to the physical page in a software table, and then turns on the appropriate UW and SW bits. All subsequent accesses to the page via this TLB entry will proceed normally.

6.3 Storage Control Instructions

Architecture Note
All processors in a symmetric multiprocessor must be identical with respect to the cache model, the coherence block size, and the reservation granule sizes.

6.3.1 Storage Synchronization Instructions

The *Storage Synchronization* instructions can be used to control the order in which storage accesses are performed with respect to other processors and with respect to other mechanisms that access storage.

Table 6-8. Storage Synchronization Instruction Set Index

Mnemonic	Instruction	Page
mbar	Memory Barrier	304
msync	Memory Synchronize	310

6.3.2 Cache Management Instructions

The *Cache Management* instructions obey the sequential execution model except as described in the example on page 141 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, storage protection, and the storage access ordering done by ***msync***, ***mbar***, and the other means described in Section 6.1.6.1.

Engineering Note
An example of the requirements of the sequential execution model with respect to *Cache Management* instructions is that a *Load* instruction that specifies a storage location in the block specified by a preceding ***dcbf***[***e***] instruction must be satisfied from main storage (if the location is in storage that is not Memory Coherence Required) or from coherent storage (if the location is in storage that is Memory Coherence Required), and not from the copy of the location that existed in the cache when the ***dcbf***[***e***] instruction was executed.

Similar requirements apply to cache reload buffers. For example, if a cache reload request for a given instruction cache block is pending when an ***icbi***[***e***] instruction is executed specifying the same block, the results of the reload request must not be used to satisfy a subsequent instruction fetch.

An example of the requirements of data dependencies with respect to *Cache Management* instructions is that if a ***dcbf***[***e***] instruction depends on the value returned by a preceding *Load* instruction, the invalidation caused by the ***dcbf***[***e***] must be performed after the load has been performed.

Engineering Note
If the applicable cache does not exist, all of the *Cache Management* instructions except ***dcbz***[***e***] must be treated as no-operations. If the data cache does not exist, ***dcbz***[***e***] must either (a) set to zero all bytes of the area of main storage that corresponds to the specified block or (b) cause an Alignment interrupt to be taken.

If, at any level of the storage hierarchy, a combined cache is implemented such that locations in that cache lack an indication of whether they were fetched as data or as instructions, the locations must be treated as if they were fetched as data. E.g., ***dcbf***[***e***] must flush and invalidate them, and ***icbi***[***e***] must not invalidate them. (Permitting ***icbi***[***e***] to invalidate a block that was fetched as data would make ***icbi***[***e***] act as an user mode ***dcbi***[***e***], and thereby create a security and data integrity exposure.)

Programming Note
It is suggested that the operating system provide a service that allows an application program to obtain the following information.

- Page sizes supported by the implementation (see User's Manual)
- Coherence block size
- Granule sizes for reservations
- An indication of the cache model implemented (e.g., separate instruction and data caches versus a combined cache)
- Instruction cache size
- Data cache size
- Instruction cache block size (see User's Manual)
- Data cache block size (see User's Manual)
- Block size for ***icbi***[***e***] (if no instruction cache, number of bytes zeroed by ***dcbz***[***e***])
- Block size for ***dcbt***[***e***] and ***dcbtst***[***e***] (if no data cache, number of bytes zeroed by ***dcbz***[***e***])
- Block size for ***dcbz***[***e***], ***dcbst***[***e***], ***dcbf***[***e***], and ***dcba***[***e***] (if no data cache, number of bytes zeroed by ***dcbz***[***e***])
- Instruction cache associativity
- Data cache associativity
- Factors for converting the Time Base to seconds

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

Each implementation provides an efficient means by which software can ensure that all blocks that are considered to be modified in the data cache have been copied to main storage before the processor enters any power conserving mode in which data cache contents are not maintained. The means are described in the User's Manual for the implementation.

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

The instruction cache is not necessarily kept consistent with the data cache or with main storage. When instructions are modified by processors or by other mechanisms, software must ensure that the instruction cache is made consistent with data storage 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 storage 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 instructions that have been modified.

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

Programming Note
Because the optimal instruction sequence may vary between systems, many operating systems will provide a system service to perform the function described above.

Engineering Note
Correct operation of the instruction sequence shown above, and of any corresponding system-dependent sequence, may require that an instruction fetch request not bypass a writeback of the same storage location caused by the sequence (including a writeback by another processor).

Programming Note
As stated above, the effective address 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.

Table 6-9. Cache Management Instruction Set Index

Mnemonic		Instruction	Page
dcba	RA,RB	Data Cache Block Allocate	247
dcbae	RA,RB	Data Cache Block Allocate Extended	247
dcbf	RA,RB	Data Cache Block Flush	248
dcbfe	RA,RB	Data Cache Block Flush Extended	248
dcbi	RA,RB	Data Cache Block Invalidate	249
dcbie	RA,RB	Data Cache Block Invalidate Extended	249
dcbst	RA,RB	Data Cache Block Store	251
dcbste	RA,RB	Data Cache Block Store Extended	251
dcbt	CT,RA,RB	Data Cache Block Touch	252
dcbte	CT,RA,RB	Data Cache Block Touch Extended	252
dcbtst	CT,RA,RB	Data Cache Block Touch for Store	253
dcbtste	CT,RA,RB	Data Cache Block Touch for Store Extended	253
dcbz	RA,RB	Data Cache Block set to Zero	254
dcbze	RA,RB	Data Cache Block set to Zero Extended	254
icbi	RA,RB	Instruction Cache Block Invalidate	286
icbie	RA,RB	Instruction Cache Block Invalidate Extended	286
icbt	CT,RA,RB	Instruction Cache Block Touch	287
icbte	CT,RA,RB	Instruction Cache Block Touch Extended	287

6.3.3 TLB Management Instructions

While Book E describes logically separate instruction fetch and integer (including effective address computation) operations, the programming model is that there is a common translation mechanism. Separate instruction and data TLBs as well as multi-level TLBs are allowed in Book E at the discretion of the implementation.

Table 6-10. TLB Management Instruction Set Index

Mnemonic		Instruction	Page
tlbivax	RA,RB	TLB Invalidate Virtual Address Indexed	362
tlbivaxe	RA,RB	TLB Invalidate Virtual Address Indexed Extended	362
tlbre	RT,RA,WS	TLB Read Entry	363
tlbsx	RA,RB	TLB Search Indexed	364
tlbsxe	RA,RB	TLB Search Indexed Extended	362
tlbsync		TLB Synchronize	365
tlbwe	RT,RA,WS	TLB Write Entry	366

Chapter 7 Interrupts and Exceptions

7.1 Overview

An *interrupt* is the action in which the processor saves its old context (Machine State Register and next instruction address) and begins execution at a pre-determined interrupt-handler address, with a modified Machine State Register. *Exceptions* are the events that will, if enabled, cause the processor to take an interrupt.

In Book E, exceptions are generated by signals from internal and external peripherals, instructions, the internal timer facility, debug events, or error conditions.

All interrupts, except Machine Check, are ordered within the two categories of non-critical and critical, such that only one interrupt of each category is reported, and when it is processed (taken) no program state is lost. Since Save/Restore Register pairs SRR0/SRR1 and CSRR0/CSRR1 are serially reusable resources used by all non-critical and critical interrupts respectively, program state may be lost when an unordered interrupt is taken (see Section 7.8 on page 174).

All interrupts, except Machine Check, are context synchronizing as defined in Section 1.12.1 on page 38. 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 1-2 of Section 1.12.1 but does satisfy items 3-4.

7.2 Interrupt Registers

7.2.1 Save/Restore Register 0

Save/Restore Register 0 (SRR0) is a 64-bit register. Save/Restore Register 0 bits are numbered 0 (most-significant bit) to 63 (least-significant bit). The register is used to save machine state on non-critical interrupts, and to restore machine state when an ***rfi*** is executed. On a non-critical interrupt, Save/Restore Register 0 is set to the current or next instruction address. When ***rfi*** is executed, instruction execution continues at the address in Save/Restore Register 0.

In general, Save/Restore Register 0 contains the address of the instruction that caused the non-critical interrupt, or the address of the instruction to return to after a non-critical interrupt is serviced.

The contents of Save/Restore Register 0 can be read into GPR(RT) using *mfspr RT,SRR0*. The contents of GPR(RS) can be written into Save/Restore Register 0 using *mtspr SRR0,RS*.

7.2.2 Save/Restore Register 1

Save/Restore Register 1 (SRR1) is a 32-bit register. Save/Restore Register 1 bits are numbered 32 (most-significant bit) to 63 (least-significant bit). The register is used to save machine state on non-critical interrupts, and to restore machine state when an ***rfi*** is executed. When a non-critical interrupt is taken, the contents of the Machine State Register are placed into Save/Restore Register 1. When ***rfi*** is executed, the contents of Save/Restore Register 1 are placed into the Machine State Register.

Bits of Save/Restore Register 1 that correspond to reserved bits in the Machine State Register are also reserved.

Programming Note
A Machine State Register bit that is reserved may be altered by ***rfi***/***rfci***.

The contents of Save/Restore Register 1 can be read into bits 32:63 of GPR(RT) using *mfspr RT,SRR1*, setting bits 0:31 of GPR(RT) to zero. The contents of bits 32:63 of GPR(RS) can be written into the Save/Restore Register 1 using *mtspr SRR1,RS*.

7.2.3 Critical Save/Restore Register 0

Critical Save/Restore Register 0 (CSRR0) is a 64-bit register. Critical Save/Restore Register 0 bits are numbered 0 (most-significant bit) to 63 (least-significant bit). The register is used to save machine state on critical interrupts, and to restore machine state when an ***rfci*** is executed. When a critical interrupt is taken, the Critical Save/Restore Register 0 is set to the current or next instruction address. When ***rfci*** is executed, instruction execution continues at the address in Critical Save/Restore Register 0.

In general, Critical Save/Restore Register 0 contains the address of the instruction that caused the critical interrupt, or the address of the instruction to return to after a critical interrupt is serviced.

The contents of Critical Save/Restore Register 0 can be read into GPR(RT) using *mfspr RT,CSRR0*. The contents of GPR(RS) can be written into Critical Save/Restore Register 0 using *mtspr CSRR0,RS*.

7.2.4 Critical Save/Restore Register 1

Critical Save/Restore Register 1 (CSRR1) is a 32-bit register. Critical Save/Restore Register 1 bits are numbered 32 (most-significant bit) to 63 (least-significant bit). The register is used to save machine state on critical interrupts, and to restore machine state when an ***rfci*** is executed. When a critical interrupt is taken, the contents of the Machine State Register are placed into Critical Save/Restore Register 1. When ***rfci*** is executed, the contents of Critical Save/Restore Register 1 are placed into the Machine State Register.

Bits of Critical Save/Restore Register 1 that correspond to reserved bits in the Machine State Register are also reserved.

Programming Note
A Machine State Register bit that is reserved may be altered by ***rfi***/***rfci***.

The contents of Critical Save/Restore Register 1 can be read into bits 32:63 of GPR(RT) using *mfspr RT,CSRR1*, setting bits 0:31 of GPR(RT) to zero. The contents of bits 32:63 of GPR(RS) can be written into the Critical Save/Restore Register 1 using *mtspr CSRR1,RS*.

7.2.5 Data Exception Address Register

The Data Exception Address Register (DEAR) is a 64-bit register. Data Exception Address Register bits are numbered 0 (most-significant bit) to 63 (least-significant bit). The Data Exception Address Register contains the address that was referenced by a *Load*, *Store* or *Cache Management* instruction that caused an Alignment, Data TLB Miss, or Data Storage interrupt.

The contents of Data Exception Address Register can be read into GPR(RT) using *mfspr RT,DEAR*. The contents of GPR(RS) can be written into the Data Exception Address Register using *mtspr DEAR,RS*.

7.2.6 Interrupt Vector Prefix Register

The Interrupt Vector Prefix Register (IVPR) is a 64-bit register. Interrupt Vector Prefix Register bits are numbered 0 (most-significant bit) to 63 (least-significant bit). Bits 48:63 are reserved. Bits 0:47 of the Interrupt Vector Prefix Register provides the high-order 48 bits of the address of the exception processing routines. The 16-bit exception vector offsets (provided in Section 7.2.8) are concatenated to the right of bits 0:47 of the Interrupt Vector Prefix Register to form the 64-bit address of the exception processing routine.

The contents of Interrupt Vector Prefix Register can be read into GPR(RT) using *mfspr RT,IVPR*. The contents of GPR(RS) can be written into Interrupt Vector Prefix Register using *mtspr IVPR,RS*.

7.2.7 Exception Syndrome Register

The Exception Syndrome Register (ESR) is a 32-bit register. Exception Syndrome Register bits are numbered 32 (most-significant bit) to 63 (least-significant bit). The Exception Syndrome Register provides a *syndrome* to differentiate between the different kinds of exceptions that can generate the same interrupt type. Upon the generation of one of these types of interrupts, the bit or bits corresponding to the specific exception that generated the interrupt is set, and all other Exception Syndrome Register bits are cleared. Other interrupt types do not affect the contents of the Exception Syndrome Register. The Exception Syndrome Register does not need to be cleared by software. Table 7-1 shows the bit definitions for the Exception Syndrome Register.

Table 7-1. Exception Syndrome Register Definition

Bit(s)	Syndrome		Associated Interrupt Types
32:35	Allocated		
36	PIL	Illegal Instruction exception	Program
37	PPR	Privileged Instruction exception	Program
38	PTR	Trap exception	Program
39	FP	Floating-point operation	Alignment Data Storage Data TLB Program
40	ST	Store operation	Alignment Data Storage Data TLB Error
41	Reserved		
42 43	DLK_0 DLK_1	Cache Locking (implementation-dependent)	Data Storage
44	AP	Auxiliary Processor operation	Alignment Data Storage Data TLB Program
45	PUO	Unimplemented Operation exception	Program
46	BO	Byte Ordering exception	Data Storage Inst Storage
47	PIE	Imprecise exception	Program
48:55	Reserved		
56:63	Allocated for implementation-dependent use		

Programming Note
The information provided by the Exception Syndrome Register is not complete. System software may also need to identify the type of instruction that caused the interrupt, examine the TLB entry accessed by a data or instruction storage access, as well as examining the Exception Syndrome Register to fully determine what exception or exceptions caused the interrupt. For example, a Data Storage interrupt may be caused by both a Protection Violation exception as well as a Byte Ordering exception. System software would have to look beyond ESR_{BO}, such as the state of MSR_{PR} in Save/Restore Register 1 and the page protection bits in the TLB entry accessed by the storage access, to determine whether or not a Protection Violation also occurred.

Engineering Note
An implementation may choose to implement additional Exception Syndrome Register bits to identify implementation-specific exception types or provide additional information about architected interrupt types.

The contents of the Exception Syndrome Register can be read into bits 32:63 of GPR(RT) using *mfspr RT,ESR*, setting bits 0:31 of GPR(RT) to zero. The contents of bits 32:63 of GPR(RS) can be written into the Exception Syndrome Register using *mtspr ESR,RS*.

7.2.8 Interrupt Vector Offset Registers

The Interrupt Vector Offset Registers (IVORs) are 32-bit registers. Interrupt Vector Offset Register bits are numbered 32 (most-significant bit) to 63 (least-significant bit). Bits 32:47 and bits 60:63 are reserved. An Interrupt Vector Offset Register provides the quadword index from the base address provided by the IVPR (see Section 7.2.6) for its respective interrupt type. Interrupt Vector Offset Registers 0 through 15 are provided for the defined interrupt types. SPR numbers corresponding to Interrupt Vector Offset Registers 16 through 31 are reserved. SPR numbers corresponding to Interrupt Vector Offset Registers 32 through 63 are allocated for implementation-dependent use. Table 7-2 provides the assignments of specific Interrupt Vector Offset Registers to specific interrupt types.

Table 7-2. Interrupt Vector Offset Registers

IVORi	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
IVOR16 : IVOR31	Reserved for future architectural use
IVOR32 : IVOR63	Allocated for implementation-dependent use

Bits 48:59 of the contents of IVORi can be read into bits 48:59 of GPR(RT) using *mfspr RT,IVORi*, setting bits 0:47 and bits 60:63 of GPR(RT) to zero. Bits 48:59 of the contents of GPR(RS) can be written into bits 48:59 of IVORi using *mtspr IVORi,RS*.

7.3 Exceptions

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

Examples of exceptions that can be caused directly by the execution of an instruction include but are not limited to the following:

- an attempt to execute a reserved-illegal instruction (Illegal Instruction exception type Program interrupt)
- an attempt by an application program to execute a 'privileged' instruction (Privileged Instruction exception type Program interrupt)
- an attempt by an application program to access a 'privileged' Special Purpose Register (Privileged Instruction exception type Program interrupt)
- an attempt by an application program to access a Special Purpose Register that does not exist (Unimplemented Operation Instruction exception type Program interrupt)
- an attempt by a system program to access a Special Purpose Register that does not exist (boundedly undefined)
- the execution of a defined instruction using an invalid form (Illegal Instruction exception type Program interrupt, Unimplemented Operation exception type Program interrupt, or Privileged Instruction exception type Program interrupt)
- an attempt to access a storage location that is either unavailable (Instruction TLB Error interrupt or Data TLB Error interrupt) or not permitted (Instruction Storage interrupt or Data Storage interrupt)
- an attempt to access storage with an effective address alignment not supported by the implementation (Alignment interrupt)
- the execution of a *System Call* instruction (System Call interrupt)
- the execution of a *Trap* instruction whose trap condition is met (Trap type Program interrupt)
- the execution of a floating-point instruction when floating-point instructions are unavailable (Floating-point Unavailable interrupt)
- the execution of a floating-point instruction that causes a floating-point enabled exception to exist (Enabled exception type Program interrupt)
- the execution of a defined instruction that is not implemented by the implementation (Illegal Instruction exception or Unimplemented Operation exception type of Program interrupt)
- the execution of an allocated instruction that is not implemented by the implementation (Illegal Instruction exception or Unimplemented Operation exception type of Program interrupt)
- the execution of an allocated instruction when the auxiliary instruction is unavailable (Auxiliary Unavailable interrupt)

- the execution of an allocated instruction that causes an auxiliary enabled exception (Enabled exception type Program interrupt)

The invocation of an interrupt is precise, except that if one of the imprecise modes for invoking the Floating-point Enabled Exception type Program interrupt is in effect then the invocation of the Floating-point Enabled Exception type Program interrupt may be imprecise. When the interrupt is invoked imprecisely, the excepting instruction does not appear to complete before the next instruction starts (because one of the effects of the excepting instruction, namely the invocation of the interrupt, has not yet occurred).

7.4 Interrupt Classes

All interrupts, except for Machine Check, can be categorized according to two independent characteristics of the interrupt:

- Asynchronous/Synchronous
- Critical/Non-critical

7.4.1 Asynchronous Interrupts

Asynchronous interrupts are caused by events that are independent of instruction execution. For asynchronous interrupts, the address reported to the exception handling routine is the address of the instruction that would have executed next, had the asynchronous interrupt not occurred.

7.4.2 Synchronous Interrupts

Synchronous interrupts are those that are caused directly by the execution (or attempted execution) of instructions, and are further divided into two classes, *precise* and *imprecise*.

Synchronous, precise interrupts are those that *precisely* indicate the address of the instruction causing the exception that generated the interrupt; or, for certain synchronous, precise interrupt types, the address of the immediately following instruction.

Synchronous, imprecise interrupts are those that may indicate the address of the instruction causing the exception that generated the interrupt, or some instruction after the instruction causing the exception.

7.4.2.1 Synchronous, Precise Interrupts

When the execution or attempted execution of an instruction causes a synchronous, precise interrupt, the following conditions exist at the interrupt point.

- Save/Restore Register 0 or Critical Save/Restore Register 0 addresses either the instruction causing the exception or the instruction immediately following instruction causing the exception. Which instruction is addressed can be determined from the interrupt type and status bits.

- An interrupt is generated such that all instructions preceding the instruction causing the exception appear to have completed with respect to the executing processor. However, some storage accesses associated with these preceding instructions may not have been performed with respect to other processors and mechanisms.

- The instruction causing the exception may appear not to have begun execution (except for causing the exception), may have been partially executed, or may have completed, depending on the interrupt type. See Section 7.7 on page 173.

- Architecturally, no subsequent instruction has executed beyond the instruction causing the exception.

7.4.2.2 Synchronous, Imprecise Interrupts

When the execution or attempted execution of an instruction causes an imprecise interrupt, the following conditions exist at the interrupt point.

- Save/Restore Register 0 or Critical Save/Restore Register 0 addresses either the instruction causing the exception or some instruction following the instruction causing the exception that generated the interrupt.

- An interrupt is generated such that all instructions preceding the instruction addressed by Save/Restore Register 0 or Critical Save/Restore Register 0 appear to have completed with respect to the executing processor.

- If the imprecise interrupt is forced by the context synchronizing mechanism, due to an instruction that causes another exception that generates an interrupt (e.g., Alignment, Data Storage), then Save/Restore Register 0 addresses the interrupt-forcing instruction, and the interrupt-forcing instruction may have been partially executed (see Section 7.7 on page 173).

- If the imprecise interrupt is forced by the execution synchronizing mechanism, due to executing an execution synchronizing instruction other than ***msync*** or ***isync***, then Save/Restore Register 0 or Critical Save/Restore Register 0 addresses the interrupt-forcing instruction, and the interrupt-forcing instruction appears not to have begun execution (except for its forcing the imprecise interrupt). If the imprecise interrupt is forced by an ***msync*** or ***isync*** instruction, then Save/Restore Register 0 or Critical Save/Restore Register 0 may address either the ***msync*** or ***isync*** instruction, or the following instruction.

- If the imprecise interrupt is not forced by either the context synchronizing mechanism or the execution synchronizing mechanism, then the instruction addressed by Save/Restore Register 0 or Critical Save/Restore Register 0 may have been partially executed (see Section 7.7 on page 173).

- No instruction following the instruction addressed by Save/Restore Register 0 or Critical Save/Restore Register 0 has executed.

7.4.3 Critical/Non-Critical Interrupts

Interrupts can also be classified as critical or non-critical interrupts. Certain interrupt types demand immediate attention, even if other interrupt types are currently being processed and have not yet had the opportunity to save the state of the machine (i.e. return address and captured state of the Machine State Regis-

ter). To enable taking a critical interrupt immediately after a non-critical interrupt is taken (i.e. before the state of the machine has been saved), two sets of Save/Restore Register pairs are provided. Critical interrupts use the Save/Restore Register pair CSRR0/CSRR1. Non-Critical interrupts use Save/Restore Register pair SRR0/SRR1.

7.4.4 Machine Check Interrupts

Machine Check interrupts are a special case. They are typically caused by some kind of hardware or storage subsystem failure, or by an attempt to access an invalid address. A Machine Check may be caused indirectly by the execution of an instruction, but not be recognized and/or reported until long after the processor has executed past the instruction that caused the Machine Check. As such, Machine Check interrupts cannot properly be thought of as synchronous or asynchronous, nor as precise or imprecise. They are handled as critical class interrupts however. In the case of Machine Check, the following general rules apply:

1. No instruction after the one whose address is reported to the Machine Check interrupt handler in Critical Save/Restore Register 0 has begun execution.

2. The instruction whose address is reported to the Machine Check interrupt handler in Critical Save/Restore Register 0, and all prior instructions, may or may not have completed successfully. All those instructions that are ever going to complete appear to have done so already, and have done so within the context existing prior to the Machine Check interrupt. No further interrupt (other than possible additional Machine Check interrupts) will occur as a result of those instructions.

7.5 Interrupt Processing

Associated with each kind of interrupt is an *interrupt vector*, that is the address of the initial instruction that is executed when the corresponding interrupt occurs.

Interrupt processing consists of saving a small part of the processor's state in certain registers, identifying the cause of the interrupt in another register, and continuing execution at the corresponding interrupt vector location. When an exception exists that will cause an interrupt to be generated and it has been determined that the interrupt can be taken, the following actions are performed, in order:

1. Save/Restore Register 0 (for non-critical class interrupts) or Critical Save/Restore Register 0 (for critical class interrupts) is loaded with an instruction address that depends on the type of interrupt; see the specific interrupt description for details.

2. The Exception Syndrome Register 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 Exception Syndrome Register setting to indicate to the cause of the interrupt was.

3. Save/Restore Register 1 (for non-critical class interrupts) or Critical Save/Restore Register 1 (for critical class interrupts) is loaded with a copy of the contents of the Machine State Register.

4. The Machine State Register is updated as described below. The new values take effect beginning with the first instruction following the interrupt. Machine State Register bits of particular interest are the following.

   - $MSR_{WE,EE,PR,FP,FE0,FE1,IS,DS}$ are set to 0 by all interrupts.
   - $MSR_{CE,DE}$ are set to 0 by critical class interrupts and left unchanged by non-critical class interrupts.
   - MSR_{ME} is set to 0 by Machine Check interrupts and left unchanged by all other interrupts.
   - Other defined Machine State Register bits are left unchanged by all interrupts.

   See Section 2.1.1 on page 39 for more detail on the definition of the Machine State Register.

5. Instruction fetching and execution resumes, using the new Machine State Register value, at a location specific to the interrupt type. The location is

$$IVPR_{0:47} \,||\, IVORi_{48:59} \,||\, 0b0000$$

   where IVPR is the Interrupt Vector Prefix Register and IVORi is the Interrupt Vector Offset Register for that interrupt type (see Table 7-2 on page 147). The contents of the Interrupt Vector Prefix Register and Interrupt Vector Offset Registers are indeterminate upon reset, and must be initialized by system software using the ***mtspr*** instruction.

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 non-critical interrupt handling routine, execution of an ***rfi*** causes the Machine State Register to be restored from the contents of Save/Restore Register 1 and instruction execution to resume at the address contained in Save/Restore Register 0. Likewise, execution of an ***rfci*** performs the same function at the end of a critical interrupt handling routine, using Critical Save/Restore Register 0 instead of Save/Restore Register 0 and Critical Save/Restore Register 1 instead of Save/Restore Register 1.

Programming Note
In general, at process switch, due to possible process interlocks and possible data availability requirements, the operating system needs to consider executing the following.

- ***stwcx***[***e***]***.*** or ***stdcxe.***, to clear the reservation if one is outstanding, to ensure that a ***lwarx***[***e***] or ***ldarxe*** in the "old" process is not paired with a ***stwcx***[***e***]***.*** or ***stdcxe.*** in the "new" process.
- ***msync***, to ensure that all storage operations of an interrupted process are complete with respect to other processors before that process begins executing on another processor.
- ***isync***, ***rfi*** or ***rfci***, to ensure that the instructions in the "new" process execute in the "new" context.

7.6 Interrupt Definitions

Table 7-3 provides a summary of each interrupt type, the various exception types that may cause that interrupt type, the classification of the interrupt, which Exception Syndrome Register bits can be set, if any, which Machine State Register bits can mask the interrupt type and which Interrupt Vector Offset Register is used to specify that interrupt type's vector address.

Table 7-3. Interrupt and Exception Types

IVOR	Interrupt Type	Exception Type	Asynchronous	Synchronous, Precise	Synchronous, Imprecise	Critical	ESR (See Note 5)	MSR Mask Bit(s)	DBCR0/TCR Mask Bit	Notes (see page 154)	Page
IVOR0	Critical Input	Critical Input	x			x		CE		1	155
IVOR1	Machine Check	Machine Check				x		ME		2,4	156
IVOR2	Data Storage	Access		x			[ST],[FP,AP]			9	157
		Load and Reserve or *Store Conditional* to 'write-thru required' storage (W=1)		x			[ST]			9	
		Cache Locking		x			{DLK_0,DLK_1},[ST]			8	
		Byte Ordering		x			[ST],[FP,AP],BO				
IVOR3	Inst Storage	Access		x							159
		Byte Ordering		x			BO				
IVOR4	External Input	External Input	x					EE		1	160
IVOR5	Alignment	Alignment		x			[ST],[FP,AP]				161
IVOR6	Program	Illegal		x			PIL				163
		Privileged		x			PPR,[AP]				
		Trap		x			PTR				
		FP Enabled		x	x		FP,[PIE]	FE0,FE1		6,7	
		AP Enabled		x	x		AP				
		Unimplemented Op		x			PUO,[FP,AP]			7	
IVOR7	FP Unavailable	FP Unavailable		x							165
IVOR8	System Call	System Call		x							166
IVOR9	AP Unavailable	AP Unavailable		x							166
IVOR10	Decrementer		x					EE	DIE		167
IVOR11	FIT		x					EE	FIE		168
IVOR12	Watchdog		x			x		CE	WIE		169
IVOR13	Data TLB Error	Data TLB Miss		x			[ST],[FP,AP]				170
IVOR14	Inst TLB Error	Inst TLB Miss		x							171
IVOR15	Debug	Trap		x		x		DE	IDM		172
		Inst Addr Compare		x		x		DE	IDM		
		Data Addr Compare		x		x		DE	IDM		
		Instruction Complete		x		x		DE	IDM	3	
		Branch Taken		x		x		DE	IDM	3	
		Return From Interrupt		x		x		DE	IDM		
		Interrupt Taken	x			x		DE	IDM		
		Uncond Debug Event	x			x		DE	IDM		

Table 7-3 Notes

1. Although it is not specified as part of Book E, it is common for system implementations to provide, as part of the interrupt controller, independent mask and status bits for the various sources of Critical Input and External Input interrupts.

2. Machine Check interrupts are a special case and are not classified as asynchronous nor synchronous. See Section 7.4.4 on page 151.

3. The Instruction Complete and Branch Taken debug events are only defined for MSR_{DE}=1 when in Internal Debug Mode ($DBCR0_{IDM}$=1). In other words, when in Internal Debug Mode with MSR_{DE}=0, then Instruction Complete and Branch Taken debug events cannot occur, and no Debug Status Register status bits are set and no subsequent imprecise Debug interrupt will occur (see Section 9.3 on page 201).

4. Machine Check status information is commonly provided as part of the system implementation, but is not part of Book E. See the User's Manual.

5. In general, when an interrupt causes a particular Exception Syndrome Register bit or bits to be set (or cleared) as indicated in the table, it also causes all other Exception Syndrome Register bits to be cleared. There may be special rules regarding the handling of implementation-specific Exception Syndrome Register bits. See the User's Manual.

   Legend:
   [xxx] means ESR_{xxx} could be set
   [xxx,yyy] means either ESR_{xxx} or ESR_{yyy} may be set, but never both
   (xxx,yyy) means either ESR_{xxx} or ESR_{yyy} will be set, but never both
   {xxx,yyy} means either ESR_{xxx} or ESR_{yyy} will be set, or possibly both
   xxx means ESR_{xxx} is set

6. The precision of the Floating-point Enabled Exception type Program interrupt is controlled by the $MSR_{FE0,FE1}$ bits. When $MSR_{FE0,FE1}$=0b01 or 0b10, the interrupt may be imprecise. When such a Program interrupt is taken, if the address saved in SRR0 is not the address of the instruction that caused the exception (i.e. the instruction that caused $FPSCR_{FEX}$ to be set to 1), ESR_{PIE} is set to 1. When $MSR_{FE0,FE1}$=0b11, the interrupt is precise. When $MSR_{FE0,FE1}$=0b00, the interrupt is masked, and the interrupt will subsequently occur imprecisely if and when Floating-point Enabled Exception type Program interrupts are enabled by setting either or both of $MSR_{FE0,FE1}$, and will also cause ESR_{PIE} to be set to 1. See Section 7.6.7 on page 163. Also, exception status on the exact cause is available in the Floating-Point Status and Control Register (see Section 5.2.2 on page 69 and Section 5.4 on page 81).

   The precision of the Auxiliary Processor Enabled Exception type Program interrupt is implementation-dependent. See the User's Manual.

7. Auxiliary Processor exception status is commonly provided as part of the implementation and is not part of Book E.

8. Cache locking and cache locking exceptions are implementation-dependent. See the User's Manual.

9. Software must examine the instruction and the subject TLB entry to determine the exact cause of the interrupt.

7.6.1 Critical Input Interrupt

A Critical Input interrupt occurs when no higher priority exception exists (see Section 7.9 on page 178), a Critical Input exception is presented to the interrupt mechanism, and MSR_{CE}=1. While the specific definition of a Critical Input exception is implementation-dependent, it would typically be caused by the activation of an asynchronous signal that is part of the system. Also, implementations may provide an alternative means (in addition to MSR_{CE}) for masking the Critical Input interrupt. See the User's Manual.

Critical Save/Restore Register 0, Critical Save/Restore Register 1, and Machine State Register are updated as follows:

Critical Save/Restore Register 0
Set to the effective address of the next instruction to be executed.

Critical Save/Restore Register 1
Set to the contents of the Machine State Register at the time of the interrupt.

Machine State Register

ME Unchanged.

All other Machine State Register bits set to 0.

Instruction execution resumes at address $IVPR_{0:47}||IVOR0_{48:59}||0b0000$.

Programming Note
Software is responsible for taking any action(s) that are required by the implementation in order to clear any Critical Input exception status prior to re-enabling MSR_{CE} in order to avoid another, redundant Critical Input interrupt.

7.6.2 Machine Check Interrupt

A Machine Check interrupt occurs when no higher priority exception exists (see Section 7.9 on page 178), a Machine Check exception is presented to the interrupt mechanism, and MSR_{ME}=1. The specific cause or causes of Machine Check exceptions are implementation-dependent, as are the details of the actions taken on a Machine Check interrupt. See the User's Manual.

Critical Save/Restore Register 0, Critical Save/Restore Register 1, Machine State Register, and Exception Syndrome Register are updated as follows:

Critical Save/Restore Register 0

Set to an instruction address. As closely as possible, set to the effective address of an instruction that was executing or about to be executed when the Machine Check exception occurred.

Critical Save/Restore Register 1

Set to the contents of the Machine State Register at the time of the interrupt.

Machine State Register

All Machine State Register bits set to 0.

Exception Syndrome Register

Implementation-dependent (see User's Manual).

Instruction execution resumes at address $IVPR_{0:47}||IVOR1_{48:59}||0b0000$.

Programming Note
If a Machine Check interrupt is caused by an error in the storage subsystem, the storage subsystem may return incorrect data, that may be placed into registers and/or on-chip caches.

Programming Note
On implementations that a Machine Check interrupt can be caused by referring to an invalid real address, executing a ***dcbz***[***e***] or ***dcba***[***e***] instruction can cause a delayed Machine Check interrupt by establishing in the data cache a block that is associated with an invalid real address. See Section 6.3.2 on page 139. A Machine Check interrupt can eventually occur if and when a subsequent attempt is made to write that block to main storage, for example as the result of executing an instruction that causes a cache miss for which the block is the target for replacement or as the result of executing a ***dcbst***[***e***] or ***dcbf***[***e***] instruction.

7.6.3 Data Storage Interrupt

A Data Storage interrupt may occur when no higher priority exception exists (see Section 7.9 on page 178) and a Data Storage exception is presented to the interrupt mechanism. A Data Storage exception is caused when any of the following exceptions arises during execution of an instruction:

Read Access Control exception

A Read Access Control exception is caused when one of the following conditions exist.

– While in user mode (MSR_{PR}=1), a *Load* or 'load-class' *Cache Management* instruction attempts to access a location in storage that is not user mode read enabled (i.e. page access control bit UR=0).

– While in supervisor mode (MSR_{PR}=0), a *Load* or 'load-class' *Cache Management* instruction attempts to access a location in storage that is not supervisor mode read enabled (i.e. page access control bit SR=0).

Write Access Control exception

A Write Access Control exception is caused when one of the following conditions exist.

– While in user mode (MSR_{PR}=1), a *Store* or 'store-class' *Cache Management* instruction attempts to access a location in storage that is not user mode write enabled (i.e. page access control bit UW=0).

– While in supervisor mode (MSR_{PR}=0), a *Store* or 'store-class' *Cache Management* instruction attempts to access a location in storage that is not supervisor mode write enabled (i.e. page access control bit SW=0).

Byte Ordering exception

A Byte Ordering exception may occur when the implementation cannot perform the data storage access in the byte order specified by the Endian storage attribute of the page being accessed.

> **Architecture Note**
> The Byte Ordering exception is provided to assist implementations that cannot support dynamically switching byte ordering between consecutive storage accesses, cannot support the byte order for a class of storage accesses, or cannot support unaligned storage accesses using a specific byte order.

Cache Locking exception

A Cache Locking exception may occur when the locked state of one or more cache lines has the potential to be altered. This exception is implementation-dependent.

Storage Synchronization exception

A Storage Synchronization exception may occur when an attempt is made to execute a *Load and Reserve* or *Store Conditional* instruction from or to a location that is Write Through Required or Caching Inhibited (if the interrupt does not occur then the instruction executes correctly: see Section 6.1.6.2 on page 117).

If 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, then it is implementation-dependent whether a Data Storage interrupt occurs. See User's Manual for the implementation.

Instructions ***lswx*** or ***stswx*** with a length of zero, ***icbt***[***e***], ***dcbt***[***e***], ***dcbtst***[***e***], or ***dcba***[***e***] cannot cause a Data Storage interrupt, regardless of the effective address.

Programming Note
The ***icbi***[***e***] and ***icbt***[***e***] instructions are treated as *Loads* from the addressed byte with respect to address translation and protection. These *Instruction Cache Management* instructions use MSR_{DS}, not MSR_{IS}, to determine translation for their operands. Instruction Storage exceptions and Instruction TLB Miss exceptions are associated with the 'fetching' of instructions not with the 'execution' of instructions. Data Storage exceptions and Data TLB Miss exceptions are associated with the 'execution' of *Instruction Cache Management* instructions.

When a Data Storage interrupt occurs, the processor suppresses the execution of the instruction causing the Data Storage exception.

Save/Restore Register 0, Save/Restore Register 1, Machine State Register, Data Exception Address Register, and Exception Syndrome Register are updated as follows:

Save/Restore Register 0
Set to the effective address of the instruction causing the Data Storage interrupt.

Save/Restore Register 1
Set to the contents of the Machine State Register at the time of the interrupt.

Machine State Register

CE, ME, DE Unchanged.

All other Machine State Register bits set to 0.

Data Exception Address Register
Set to the effective address of a byte that is both within the range of the bytes being accessed by the *Storage Access* or *Cache Management* instruction, and within the page whose access caused the Data Storage exception.

Exception Syndrome Register

FP Set to 1 if the instruction causing the interrupt is a floating-point load or store; otherwise set to 0.

ST Set to 1 if the instruction causing the interrupt is a *Store* or 'store-class' *Cache Management* instruction; otherwise set to 0.

$DLK_{0:1}$ Set to an implementation-dependent value due to a Cache Locking exception causing the interrupt. See User's Manual.

AP Set to 1 if the instruction causing the interrupt is an Auxiliary Processor load or store; otherwise set to 0.

BO Set to 1 if the instruction caused a Byte Ordering exception; otherwise set to 0.

All other defined Exception Syndrome Register bits are set to 0.

Instruction execution resumes at address $IVPR_{0:47}||IVOR2_{48:59}||0b0000$.

7.6.4 Instruction Storage Interrupt

An Instruction Storage interrupt occurs when no higher priority exception exists (see Section 7.9 on page 178) and an Instruction Storage exception is presented to the interrupt mechanism. An Instruction Storage exception is caused when any of the following exceptions arises during execution of an instruction:

Execute Access Control exception

An Execute Access Control exception is caused when one of the following conditions exist.

– While in user mode (MSR_{PR}=1), an instruction fetch attempts to access a location in storage that is not user mode execute enabled (i.e. page access control bit UX=0).

– While in supervisor mode (MSR_{PR}=0), an instruction fetch attempts to access a location in storage that is not supervisor mode execute enabled (i.e. page access control bit SX=0).

Byte Ordering exception

A Byte Ordering exception may occur when the implementation cannot perform the instruction fetch in the byte order specified by the Endian storage attribute of the page being accessed.

> **Architecture Note**
> This exception is provided to assist implementations that cannot support dynamically switching byte ordering between consecutive storage accesses, cannot support the byte order for a class of storage accesses, or cannot support unaligned storage accesses using a specific byte order.

When an Instruction Storage interrupt occurs, the processor suppresses the execution of the instruction causing the Instruction Storage exception.

Save/Restore Register 0, Save/Restore Register 1, Machine State Register, and Exception Syndrome Register are updated as follows:

Save/Restore Register 0

Set to the effective address of the instruction causing the Instruction Storage interrupt.

Save/Restore Register 1

Set to the contents of the Machine State Register at the time of the interrupt.

Machine State Register

CE, ME, DE Unchanged.

All other Machine State Register bits set to 0.

Exception Syndrome Register

BO Set to 1 if the instruction fetch caused a Byte Ordering exception; otherwise set to 0.

All other defined Exception Syndrome Register bits are set to 0.

Programming Note
Protection Violation and Byte Ordering exceptions are not mutually exclusive. Even if ESR_{BO} is set, system software must also examine the TLB entry accessed by the instruction fetch to determine whether or not a Protection Violation may have also occurred.

Instruction execution resumes at address $IVPR_{0:47}||IVOR3_{48:59}||0b0000$.

7.6.5 External Input Interrupt

An External Input interrupt occurs when no higher priority exception exists (see Section 7.9 on page 178), an External Input exception is presented to the interrupt mechanism, and MSR_{EE}=1. While the specific definition of an External Input exception is implementation-dependent, it would typically be caused by the activation of an asynchronous signal that is part of the processing system. Also, implementations may provide an alternative means (in addition to MSR_{EE}) for masking the External Input interrupt. See the User's Manual.

Save/Restore Register 0, Save/Restore Register 1, and Machine State Register are updated as follows:

Save/Restore Register 0
Set to the effective address of the next instruction to be executed.

Save/Restore Register 1
Set to the contents of the Machine State Register at the time of the interrupt.

Machine State Register

CE, ME, DE Unchanged.

All other Machine State Register bits set to 0.

Instruction execution resumes at address $IVPR_{0:47}||IVOR4_{48:59}||0b0000$.

Programming Note
Software is responsible for taking whatever action(s) are required by the implementation in order to clear any External Input exception status prior to re-enabling MSR_{EE} in order to avoid another, redundant External Input interrupt.

7.6.6 Alignment Interrupt

An Alignment interrupt occurs when no higher priority exception exists (see Section 7.9 on page 178) and an Alignment exception is presented to the interrupt mechanism. An Alignment exception may be caused when the implementation cannot perform a data storage access for one of the following reasons:

- The operand of a *Load* or *Store* is not aligned.
- The instruction is a *Move Assist*, *Load Multiple* or *Store Multiple*.
- The operand of ***dcbz***[***e***] is in storage that is Write Through Required or Caching Inhibited, or ***dcbz***[***e***] is executed in an implementation that has either no data cache or a Write Through data cache.
- The operand of a *Store*, except *Store Conditional*, is in storage that is Write-Through Required.

For ***lmw*** and ***stmw*** with an operand that is not word-aligned, and for *Load and Reserve* and *Store Conditional* instructions with an operand that is not aligned, an implementation may yield boundedly undefined results instead of causing an Alignment interrupt. A *Store Conditional* to Write Through Required storage may either cause a Data Storage interrupt, cause an Alignment interrupt, or correctly execute the instruction. For all other cases listed above, an implementation may execute the instruction correctly instead of causing an Alignment interrupt. (For ***dcbz***[***e***], 'correct' execution means setting each byte of the block in main storage to 0x00.)

Programming Note
The architecture does not support the use of an unaligned effective address by *Load and Reserve* and *Store Conditional* instructions. If an Alignment interrupt occurs because one of these instructions specifies an unaligned effective address, the Alignment interrupt handler must not attempt to emulate the instruction, but instead should treat the instruction as a programming error.

When an Alignment interrupt occurs, the processor suppresses the execution of the instruction causing the Alignment exception.

Save/Restore Register 0, Save/Restore Register 1, Machine State Register, Data Exception Address Register, and Exception Syndrome Register are updated as follows:

Save/Restore Register 0
Set to the effective address of the instruction causing the Alignment interrupt.

Save/Restore Register 1
Set to the contents of the Machine State Register at the time of the interrupt.

Machine State Register

CE, ME, DE Unchanged.

All other Machine State Register bits set to 0.

Data Exception Address Register

Set to the effective address of a byte that is both within the range of the bytes being accessed by the *Storage Access* or *Cache Management* instruction, and within the page whose access caused the Alignment exception.

Exception Syndrome Register

FP Set to 1 if the instruction causing the interrupt is a floating-point load or store; otherwise set to 0.

ST Set to 1 if the instruction causing the interrupt is a *Store*; otherwise set to 0.

AP Set to 1 if the instruction causing the interrupt is an Auxiliary Processor load or store; otherwise set to 0.

All other defined Exception Syndrome Register bits are set to 0.

Instruction execution resumes at address $IVPR_{0:47} || IVOR5_{48:59} || 0b0000$.

7.6.7 Program Interrupt

A Program interrupt occurs when no higher priority exception exists (see Section 7.9 on page 178), a Program exception is presented to the interrupt mechanism, and, for Floating-point Enabled exception, $MSR_{FE0,FE1}$ are non-zero. A Program exception is caused when any of the following exceptions arises during execution of an instruction:

Floating-point Enabled exception

A Floating-point Enabled exception is caused when $FPSCR_{FEX}$ is set to 1 by the execution of a floating-point instruction that causes an enabled exception, including the case of a *Move To FPSCR* instruction that causes an exception bit and the corresponding enable bit both to be 1. Note that in this context, the term 'enabled exception' refers to the enabling provided by control bits in the Floating-Point Status and Control Register. See Section 5.2.2 on page 69.

Auxiliary processor Enabled exception

The cause of an Auxiliary Processor Enabled exception is implementation-dependent.

Illegal Instruction exception

An Illegal Instruction exception *does* occur when execution is attempted of any of the following kinds of instructions.

– a reserved-illegal instruction

– when MSR_{PR}=1 (user mode), an ***mtspr*** or ***mfspr*** that specifies an SPRN value with $SPRN_5$=0 (user-mode accessible) that represents an unimplemented Special Purpose Register

An Illegal Instruction exception *may* occur when execution is attempted of any of the following kinds of instructions. If the exception does not occur, the alternative is shown in parentheses. See User's Manual for the implementation.

– an instruction that is in invalid form (boundedly undefined results)

– an ***lswx*** instruction for which GPR(RA) or GPR(RB) is in the range of registers to be loaded (boundedly undefined results)

– a reserved-nop instruction (no-operation performed is preferred)

– a defined or allocated instruction that is not implemented by the implementation (Unimplemented Operation exception)

Privileged Instruction exception

A Privileged Instruction exception occurs when MSR_{PR}=1 and execution is attempted of any of the following kinds of instructions.

– a privileged instruction

– an ***mtspr*** or ***mfspr*** instruction that specifies an SPRN value with $SPRN_5$=1

Trap exception

A Trap exception occurs when any of the conditions specified in a *Trap* instruction are met and the exception is not also enabled as a Debug interrupt. If enabled as a Debug interrupt (i.e. $DBCR0_{TRAP}$=1, $DBCR0_{IDM}$=1,

and MSR_{DE}=1), then a Debug interrupt will be taken instead of the Program interrupt.

Unimplemented Operation exception

An Unimplemented Operation exception *may* occur when a defined or allocated instruction is encountered that is not implemented by the implementation. Otherwise an Illegal Instruction exception occurs. See the User's Manual for the implementation.

Save/Restore Register 0, Save/Restore Register 1, Machine State Register, and Exception Syndrome Register are updated as follows:

Save/Restore Register 0

For all Program interrupts except an Enabled exception when in one of the imprecise modes (see Section 2.1.1 on page 39) or when a disabled exception is subsequently enabled, set to the effective address of the instruction that caused the Program interrupt.

For an imprecise Enabled exception, set to the effective address of the excepting instruction or to the effective address of some subsequent instruction. If it points to a subsequent instruction, that instruction has not been executed, and ESR_{PIE} is set to 1. If a subsequent instruction is an ***msync*** or ***isync***, Save/Restore Register 0 will point at the ***msync*** or ***isync*** instruction, or at the following instruction.

If $FPSCR_{FEX}$=1 but both MSR_{FE0}=0 and MSR_{FE1}=0, an Enabled exception type Program interrupt will occur imprecisely prior to or at the next synchronizing event if these Machine State Register bits are altered by any instruction that can set the Machine State Register so that the expression

$$(MSR_{FE0} \mid MSR_{FE1}) \,\&\, FPSCR_{FEX}$$

is 1. When this occurs, Save/Restore Register 0 is loaded with the address of the instruction that would have executed next, not with the address of the instruction that modified the Machine State Register causing the interrupt, and ESR_{PIE} is set to 1.

Save/Restore Register 1

Set to the contents of the Machine State Register at the time of the interrupt.

Machine State Register

CE, ME, DE Unchanged.

All other Machine State Register bits set to 0.

Exception Syndrome Register

PIL Set to 1 if an Illegal Instruction exception type Program interrupt; otherwise set to 0

PPR Set to 1 if a Privileged Instruction exception type Program interrupt; otherwise set to 0

PTR Set to 1 if a Trap exception type Program interrupt; otherwise set to 0

PUO Set to 1 if an Unimplemented Operation exception type Program interrupt; otherwise set to 0

FP Set to 1 if the instruction causing the interrupt is a floating-point instruction; otherwise set to 0.

PIE Set to 1 if a Floating-point Enabled exception type Program interrupt, and the address saved in SRR0 is not the address of the instruction causing the exception (i.e. the instruction that caused $FPSCR_{FEX}$ to be set); otherwise set to 0.

AP Set to 1 if the instruction causing the interrupt is an Auxiliary Processor instruction; otherwise set to 0.

All other defined Exception Syndrome Register bits are set to 0.

Instruction execution resumes at address $IVPR_{0:47}||IVOR6_{48:59}||0b0000$.

Engineering Note
Supporting the Imprecise Recoverable Mode Floating-Point Enabled exception type Program interrupt as a precise interrupt may be convenient for some implementations. However, if the Imprecise Recoverable Mode Floating-Point Enabled exception type Program interrupt is implemented as an imprecise interrupt, the hardware must provide, at the minimum, the address at which to resume the interrupted process (this is given in Save/Restore Register 0), the excepting instruction's opcode, extended opcode, and record bit, the source values or registers, and the target register. This information can be provided directly in registers or by means of a pointer to the excepting instruction. See the User's Manual for the implementation.

7.6.8 Floating-Point Unavailable Interrupt

A Floating-Point Unavailable interrupt occurs when no higher priority exception exists (see Section 7.9 on page 178), an attempt is made to execute a floating-point instruction (i.e. any instruction listed in Section 5.6 on page 98), and MSR_{FP}=0.

When a Floating-Point Unavailable interrupt occurs, the processor suppresses the execution of the instruction causing the Floating-Point Unavailable interrupt.

Save/Restore Register 0, Save/Restore Register 1, and Machine State Register are updated as follows:

Save/Restore Register 0
Set to the effective address of the instruction that caused the interrupt.

Save/Restore Register 1
Set to the contents of the Machine State Register at the time of the interrupt.

Machine State Register

CE, ME, DE Unchanged.

All other Machine State Register bits set to 0.

Instruction execution resumes at address $IVPR_{0:47}||IVOR7_{48:59}||0b0000$.

7.6.9 System Call Interrupt

A System Call interrupt occurs when no higher priority exception exists (see Section 7.9 on page 178) and a *System Call* (***sc***) instruction is executed.

Save/Restore Register 0, Save/Restore Register 1, and Machine State Register are updated as follows:

Save/Restore Register 0
Set to the effective address of the instruction *after* the ***sc*** instruction.

Save/Restore Register 1
Set to the contents of the Machine State Register at the time of the interrupt.

Machine State Register

CE, ME, DE Unchanged.

All other Machine State Register bits set to 0.

Instruction execution resumes at address $IVPR_{0:47}||IVOR8_{48:59}||0b0000$.

7.6.10 Auxiliary Processor Unavailable Interrupt

An Auxiliary Processor Unavailable interrupt occurs when no higher priority exception exists (see Section 7.9 on page 178), an attempt is made to execute an Auxiliary Processor instruction (including Auxiliary Processor loads, stores, and moves), the target Auxiliary Processor is present on the implementation, and the Auxiliary Processor is configured as unavailable. Details of the Auxiliary Processor, its instruction set, and its configuration are implementation-dependent. See User's Manual for the implementation.

When an Auxiliary Processor Unavailable interrupt occurs, the processor suppresses the execution of the instruction causing the Auxiliary Processor Unavailable interrupt.

Registers Save/Restore Register 0, Save/Restore Register 1, and Machine State Register are updated as follows:

Save/Restore Register 0
Set to the effective address of the instruction that caused the interrupt.

Save/Restore Register 1
Set to the contents of the Machine State Register at the time of the interrupt.

Machine State Register

CE, ME, DE Unchanged.

All other Machine State Register bits set to 0.

Instruction execution resumes at address $IVPR_{0:47}||IVOR9_{48:59}||0b0000$.

7.6.11 Decrementer Interrupt

A Decrementer interrupt occurs when no higher priority exception exists (see Section 7.9 on page 178), a Decrementer exception exists (TSR_{DIS}=1), and the interrupt is enabled (TCR_{DIE}=1 and MSR_{EE}=1). See Section 8.5 on page 194.

Note
MSR_{EE} also enables the External Input and Fixed-Interval Timer interrupts.

Save/Restore Register 0, Save/Restore Register 1, Machine State Register, and TSR are updated as follows:

Save/Restore Register 0
Set to the effective address of the next instruction to be executed.

Save/Restore Register 1
Set to the contents of the Machine State Register at the time of the interrupt.

Machine State Register

CE, ME, DE Unchanged.

All other Machine State Register bits set to 0.

Timer Status Register (See Section 8.3 on page 188.)

DIS Set to 1.

Instruction execution resumes at address $IVPR_{0:47}||IVOR10_{48:59}||0b0000$.

Programming Note
Software is responsible for clearing the Decrementer exception status prior to re-enabling the MSR_{EE} bit in order to avoid another redundant Decrementer interrupt. To clear the Decrementer exception, the interrupt handling routine must clear TSR_{DIS}. Clearing is done by writing a word to TSR using ***mtspr*** with a 1 in any bit position that is to be cleared and 0 in all other bit positions. The write-data to the TSR is not direct data, but a mask. A 1 causes the bit to be cleared, and a 0 has no effect.

7.6.12 Fixed-Interval Timer Interrupt

A Fixed-Interval Timer interrupt occurs when no higher priority exception exists (see Section 7.9 on page 178), a Fixed-Interval Timer exception exists (TSR_{FIS}=1), and the interrupt is enabled (TCR_{FIE}=1 and MSR_{EE}=1). See Section 8.6 on page 195.

Note
MSR_{EE} also enables the External Input and Decrementer interrupts.

Save/Restore Register 0, Save/Restore Register 1, Machine State Register, and TSR are updated as follows:

Save/Restore Register 0
Set to the effective address of the next instruction to be executed.

Save/Restore Register 1
Set to the contents of the Machine State Register at the time of the interrupt.

Machine State Register

CE, ME, DE Unchanged.

All other Machine State Register bits set to 0.

Timer Status Register (See Section 8.3 on page 188.)

FIS Set to 1

Instruction execution resumes at address $IVPR_{0:47}||IVOR11_{48:59}||0b0000$.

Programming Note
Software is responsible for clearing the Fixed-Interval Timer exception status prior to re-enabling the MSR_{EE} bit in order to avoid another redundant Fixed-Interval Timer interrupt. To clear the Fixed-Interval Timer exception, the interrupt handling routine must clear TSR_{FIS}. Clearing is done by writing a word to TSR using ***mtspr*** with a 1 in any bit position that is to be cleared and 0 in all other bit positions. The write-data to the TSR is not direct data, but a mask. A 1 causes the bit to be cleared, and a 0 has no effect.

7.6.13 Watchdog Timer Interrupt

A Watchdog Timer interrupt occurs when no higher priority exception exists (see Section 7.9 on page 178), a Watchdog Timer exception exists (TSR_{WIS}=1), and the interrupt is enabled (i.e. TCR_{WIE}=1 and MSR_{CE}=1). See Section 8.7 on page 196.

Note
MSR_{CE} also enables the Critical Input interrupt.

Critical Save/Restore Register 0, Critical Save/Restore Register 1, Machine State Register, and TSR are updated as follows:

Critical Save/Restore Register 0
Set to the effective address of the next instruction to be executed.

Critical Save/Restore Register 1
Set to the contents of the Machine State Register at the time of the interrupt.

Machine State Register

ME Unchanged.

All other Machine State Register bits set to 0.

Timer Status Register (See Section 8.3 on page 188.)

WIS Set to 1.

Instruction execution resumes at address $IVPR_{0:47}||IVOR12_{48:59}||0b0000$.

Programming Note
Software is responsible for clearing the Watchdog Timer exception status prior to re-enabling the MSR_{CE} bit in order to avoid another redundant Watchdog Timer interrupt. To clear the Watchdog Timer exception, the interrupt handling routine must clear TSR_{WIS}. Clearing is done by writing a word to TSR using ***mtspr*** with a 1 in any bit position that is to be cleared and 0 in all other bit positions. The write-data to the TSR is not direct data, but a mask. A 1 causes the bit to be cleared, and a 0 has no effect.

7.6.14 Data TLB Error Interrupt

A Data TLB Error interrupt occurs when no higher priority exception exists (see Section 7.9 on page 178) and any of the following Data TLB Error exceptions is presented to the interrupt mechanism.

TLB Miss exception
Caused when the virtual address associated with a data storage access does not match any valid entry in the TLB as specified in Section 6.2.2 on page 125.

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, then it is implementation-dependent whether a Data TLB Error interrupt occurs. See User's Manual for the implementation.

When a Data TLB Error interrupt occurs, the processor suppresses the execution of the instruction causing the Data TLB Error interrupt.

Save/Restore Register 0, Save/Restore Register 1, Machine State Register, Data Exception Address Register and Exception Syndrome Register are updated as follows:

Save/Restore Register 0
Set to the effective address of the instruction causing the Data TLB Error interrupt

Save/Restore Register 1
Set to the contents of the Machine State Register at the time of the interrupt.

Machine State Register

CE, ME, DE Unchanged.

All other Machine State Register bits set to 0.

Data Exception Address Register
Set to the effective address of a byte that is both within the range of the bytes being accessed by the *Storage Access* or *Cache Management* instruction, and within the page whose access caused the Data TLB Error exception.

Exception Syndrome Register

ST Set to 1 if the instruction causing the interrupt is a *Store*, ***dcbi***[***e***], or ***dcbz***[***e***] instruction; otherwise set to 0.

FP Set to 1 if the instruction causing the interrupt is a floating-point load or store; otherwise set to 0.

AP Set to 1 if the instruction causing the interrupt is an Auxiliary Processor load or store; otherwise set to 0.

All other defined Exception Syndrome Register bits are set to 0.

Instruction execution resumes at address $IVPR_{0:47}||IVOR13_{48:59}||0b0000$.

7.6.15 Instruction TLB Error Interrupt

An Instruction TLB Error interrupt occurs when no higher priority exception exists (see Section 7.9 on page 178) and any of the following Instruction TLB Error exceptions is presented to the interrupt mechanism.

TLB Miss exception
Caused when the virtual address associated with an instruction fetch do not match any valid entry in the TLB as specified in Section 6.2.2 on page 125.

When an Instruction TLB Error interrupt occurs, the processor suppresses the execution of the instruction causing the Instruction TLB Miss exception.

Save/Restore Register 0, Save/Restore Register 1, and Machine State Register are updated as follows:

Save/Restore Register 0
Set to the effective address of the instruction causing the Instruction TLB Error interrupt.

Save/Restore Register 1
Set to the contents of the Machine State Register at the time of the interrupt.

Machine State Register

CE, ME, DE Unchanged.

All other Machine State Register bits set to 0.

Instruction execution resumes at address $IVPR_{0:47}||IVOR14_{48:59}||0b0000$.

7.6.16 Debug Interrupt

A Debug interrupt occurs when no higher priority exception exists (see Section 7.9 on page 178), a Debug exception exists in the Debug Status Register, and Debug interrupts are enabled ($DBCR0_{IDM}$=1 and MSR_{DE}=1). A Debug exception occurs when a Debug Event causes a corresponding bit in the Debug Status Register to be set. See Chapter 9 on page 199.

Critical Save/Restore Register 0, Critical Save/Restore Register 1, Machine State Register, and Debug Status Register are updated as follows.

Critical Save/Restore Register 0

For Debug exceptions that occur while Debug interrupts are enabled ($DBCR0_{IDM}$=1 and MSR_{DE}=1), Critical Save/Restore Register 0 is set as follows:

- For Instruction Address Compare (IAC1, IAC2, IAC3, IAC4), Data Address Compare (DAC1R, DAC1W, DAC2R, DAC2W), Data Value Compare (DVC1, DVC2), Trap (TRAP), or Branch Taken (BRT) debug exceptions, set to the address of the instruction causing the Debug interrupt.

- For Instruction Complete (ICMP) debug exceptions, set to the address of the instruction that would have executed after the one that caused the Debug interrupt.

- For Unconditional Debug Event (UDE) debug exceptions, set to the address of the instruction that would have executed next if the Debug interrupt had not occurred.

- For Interrupt Taken (IRPT) debug exceptions, set to the interrupt vector value of the interrupt that caused the Interrupt Taken debug event.

- For Return From Interrupt (RET) debug exceptions, set to the address of the ***rfi*** or ***rfci*** instruction 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 will occur at the next synchronizing event if $DBCR0_{IDM}$ and MSR_{DE} are modified such that they are both 1 and if the Debug exception Status is still set in the Debug Status Register. When this occurs, Critical Save/Restore Register 0 is set to the address of the instruction that would have executed next, not with the address of the instruction that modified the Debug Control Register 0 or Machine State Register and thus caused the interrupt.

Critical Save/Restore Register 1

Set to the contents of the Machine State Register at the time of the interrupt.

Machine State Register

ME Unchanged.

All other Machine State Register bits set to 0.

Debug Status Register

Set to indicate type of Debug Event (see Chapter 9 on page 199)

Instruction execution resumes at address $IVPR_{0:47}||IVOR15_{48:59}||0b0000$.

7.7 Partially Executed Instructions

In general, the architecture permits load and store instructions to be partially executed, interrupted, and then to be restarted from the beginning upon return from the interrupt. Unaligned *Load* and *Store* instructions, or *Load Multiple*, *Store Multiple*, *Load String*, and *Store String* instructions may be broken up into multiple, smaller accesses, and these accesses may be performed in any order. In order to guarantee that a particular load or store instruction will complete without being interrupted and restarted, software must mark the storage being referred to as Guarded, and must use an elementary (non-string or non-multiple) load or store that is aligned on an operand-sized boundary.

In order to guarantee that *Load* and *Store* instructions can, in general, be restarted and completed correctly without software intervention, the following rules apply when an execution is partially executed and then interrupted:

- For an elementary *Load*, no part of the target register GPR(RT), or FPR(FRT), will have been altered.

- For 'with update' forms of *Load* or *Store*, the update register, GPR(RA), will not have been altered.

On the other hand, the following effects are permissible when certain instructions are partially executed and then restarted:

- For any *Store*, some of the bytes at the target storage location may have been altered (if write access to that page in which bytes were altered is permitted by the access control mechanism). In addition, for *Store Conditional* instructions, CR0 has been set to an undefined value, and it is undefined whether the reservation has been cleared.

- For any *Load*, some of the bytes at the addressed storage location may have been accessed (if read access to that page in which bytes were accessed is permitted by the access control mechanism).

- For *Load Multiple* or *Load String*, some of the registers in the range to be loaded may have been altered. Including the addressing registers (GPR(RA), and possibly GPR(RB)) in the range to be loaded is a programming error, and thus the rules for partial execution do not protect against overwriting of these registers.

In no case will access control be violated.

As previously stated, the only load or store instructions that are guaranteed to not be interrupted after being partially executed are elementary, aligned, guarded loads and stores. All others may be interrupted after being partially executed. The following list identifies the specific instruction types for which interruption after partial execution may occur, as well as the specific interrupt types that could cause the interruption:

1. Any *Load* or *Store* (except elementary, aligned, guarded):

   Any asynchronous interrupt
   Machine Check
   Program (Imprecise Mode Floating-Point Enabled)
   Program (Imprecise Mode Auxiliary Processor Enabled)

2. Unaligned elementary *Load* or *Store*, or any multiple or string:

   All of the above listed under item 1, plus the following:
   Data Storage (if the access crosses a protection boundary)
   Debug (Data Address Compare, Data Value Compare)

3. ***mtcrf*** may also be partially executed due to the occurrence of any of the interrupts listed under item 1 at the time the ***mtcrf*** was executing.

   - All instructions prior to the ***mtcrf*** have completed execution. (Some storage accesses generated by these preceding instructions may not have completed.)
   - No subsequent instruction has begun execution.
   - The ***mtcrf*** instruction (the address of which was saved in SRR0/CSRR0 at the occurrence of the interrupt), may appear not to have begun or may have partially executed.

7.8 Interrupt Ordering and Masking

It is possible for multiple exceptions to exist simultaneously, each of which could cause the generation of an interrupt. Furthermore, the architecture does not provide for reporting more than one interrupt of the same class (critical or non-critical) at a time. Therefore, the architecture defines that interrupts are ordered with respect to each other, and provides a masking mechanism for certain persistent interrupt types.

When an interrupt 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). However, 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 will then finally be generated.

All asynchronous interrupt types can be masked. In addition, certain synchronous interrupt types can be masked. An example of such an interrupt type is the Floating-Point Enabled exception type Program interrupt. The execution of a floating-point instruction that causes the $FPSCR_{FEX}$ bit to be set to 1 is considered an exception event, regardless of the setting of $MSR_{FE0,FE1}$. If $MSR_{FE0,FE1}$ are both 0, then the Floating-Point Enabled exception type of Program interrupt is masked, but the exception persists in the $FPSCR_{FEX}$ bit. Later, if the $MSR_{FE0,FE1}$ bits are enabled, the interrupt will finally be generated.

Architectural Note
As is the case with this example, when an otherwise synchronous, *precise* interrupt type is “delayed” in this fashion via masking, and the interrupt type is later enabled, the interrupt that is then generated due to the exception event that occurred while the interrupt type was disabled is then considered a synchronous, *imprecise* class of interrupt. However, this particular category of synchronous, imprecise interrupt is not generally discussed in other sections of this document. Rather, the discussion of synchronous, imprecise interrupts is generally limited to those specific interrupt types that are defined to be imprecise to begin with, and not those that are delayed versions of otherwise synchronous, precise interrupts.

The architecture enables implementations to avoid situations in which an interrupt would cause the state information (saved in Save/Restore Registers) from a previous interrupt to be overwritten and lost. As a first step, upon any non-critical class interrupt, hardware automatically disables any further asynchronous, non-critical class interrupts (External Input) by clearing MSR_{EE}. Likewise, upon any critical class interrupt, hardware automatically disables any further asynchronous interrupts of either class (critical and non-critical) by clearing MSR_{CE} in addition to MSR_{EE}. The additional interrupt types that are disabled by the clearing of $MSR_{CE,DE}$ are the Critical Input, Watchdog Timer and Debug interrupts. Note that Machine Check interrupts, while not considered asynchronous nor synchronous, are not maskable by either MSR_{CE}, MSR_{DE}, or MSR_{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 any subsequent asynchronous interrupts from overwriting the Save/Restore Registers (SRR0/SRR1 or CSRR0/CSRR1), prior to software being able to save their contents. Hardware also automatically clears, on any interrupt, $MSR_{WE,PR,FP,FE0,FE1,IS,DS}$. The clearing of these bits assists in the avoidance of subsequent interrupts of certain other types. However, *guaranteeing* that these interrupt types do not occur and thus do not overwrite the Save/Restore Registers (SRR0/SRR1 or CSRR0/CSRR1) also requires the cooperation of system software. Specifically, system software must avoid the execution of instructions that could cause (or enable) a subsequent interrupt, if the contents of the Save/Restore Registers (SRR0/SRR1 or CSRR0/CSRR1) have not yet been saved.

7.8.1 Guidelines for System Software

The following list identifies the actions that system software must avoid, prior to having saved the Save/Restore Registers' contents:

- Re-enabling of MSR_{EE} (or $MSR_{CE,DE}$ in critical class interrupt handlers)

  This prevents any asynchronous interrupts, as well as (in the case of MSR_{DE}) any Debug interrupts (which include both synchronous and asynchronous types).

- Branching (or sequential execution) to addresses not mapped by the TLB, or mapped without UX=1 or SX=1 permission.

  This prevents Instruction Storage and Instruction TLB Error interrupts.

- *Load*, *Store* or *Cache Management* instructions to addresses not mapped by the TLB or not having required access permissions.

  This prevents Data Storage and Data TLB Error interrupts.

- Execution of *System Call* (***sc***) or *Trap* (***tw***, ***twi***, ***td***, ***tdi***) instructions

  This prevents System Call and Trap exception type Program interrupts.

- Execution of any floating-point instruction

  This prevents Floating-Point Unavailable interrupts. Note that this interrupt would occur upon the execution of any floating-point instruction, due to the automatic clearing of MSR_{FP}. However, even if software were to re-enable MSR_{FP}, floating-point instructions must still be avoided in order to prevent Program interrupts due to various possible Program interrupt exceptions

(Floating-Point Enabled, Unimplemented Operation).

- Re-enabling of MSR_{PR}

  This prevents Privileged Instruction exception type Program interrupts. Alternatively, software could re-enable MSR_{PR}, but avoid the execution of any privileged instructions.

- Execution of any Auxiliary Processor instruction

  This prevents Auxiliary Processor Unavailable interrupts, and Auxiliary Processor Enabled type and Unimplemented Operation type Program interrupts.

- Execution of any Illegal instructions

  This prevents Illegal Instruction exception type Program interrupts.

- Execution of any instruction that could cause an Alignment interrupt

  This prevents Alignment interrupts. Included in this category are any string or multiple instructions, and any unaligned elementary load or store instructions. See Section 7.6.6 on page 161 for a complete list of instructions that may cause Alignment interrupts.

Machine Check interrupts are a special case. Machine Checks are critical class interrupts, but *normal* critical class interrupts (Critical Input, Watchdog Timer and Debug) do not automatically disable Machine Checks. Machine Checks are disabled by clearing the MSR_{ME} bit, 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, prior to the Save/Restore Registers' contents having been saved. In such a case, the interrupt may not be recoverable.

It is not necessary for hardware or software to avoid critical class interrupts from within non-critical class interrupt handlers (and hence hardware does not automatically clear $MSR_{CE,ME,DE}$ upon a non-critical interrupt), since the two classes of interrupts use different pairs of Save/Restore Registers to save the instruction address and Machine State Register (SRR0/SRR1 for non-critical, and CSRR0/CSRR1 for critical). The converse, however, is not true. That is, hardware and software must cooperate in the avoidance of both critical *and* non-critical class interrupts from within critical class interrupt handlers, even though the two classes of interrupts use different Save/Restore Register pairs. This is because the critical class interrupt may have occurred from within a non-critical handler, prior to the non-critical handler having saved the non-critical pair of Save/Restore Registers. Therefore, within the critical class interrupt handler, both pairs of Save/Restore Registers may contain data that is necessary to the system software.

7.8.2 Interrupt Order

The following is a prioritized listing of the various enabled interrupt types for which exceptions might exist simultaneously:

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 types of synchronous interrupts may have an existing exception generating it at any given time. This is guaranteed by the exception priority mechanism (see Section 7.9 on page 178) and the requirements of the Sequential Execution Model.

2. Machine Check
3. Debug
4. Critical Input
5. Watchdog Timer
6. External Input
7. Fixed-Interval Timer
8. Decrementer

Even though, as indicated above, the non-critical, synchronous exception types listed under item 1 are generated with higher priority than the critical interrupt types listed in items 2-5, the fact is that these non-critical interrupts will immediately be followed by the highest priority existing critical interrupt type, without executing any instructions at the non-critical interrupt handler. This is because the non-critical interrupt types do not automatically disable the Machine State Register mask bits for the critical interrupt types (CE and ME). In all other cases, a particular interrupt type from the above list will automatically disable any subsequent interrupts of the same type, as well as all other interrupt types that are listed below it in the priority order.

7.9 Exception Priorities

Book E requires all synchronous (precise and imprecise) interrupts 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 interrupts. Upon a synchronizing event, all previously executed instructions are required to report any synchronous imprecise interrupt-generating exceptions, and the interrupt will then be generated with all of those exception types reported cumulatively, in both the Exception Syndrome Register, and any status registers associated with the particular exception type (e.g. the Floating-Point Status and Control Register).

For any single instruction attempting to cause multiple exceptions for which the corresponding synchronous interrupt types are enabled, this section defines the priority order by which the instruction will be permitted to cause a *single* enabled exception, thus generating a particular synchronous interrupt. Note that it is this exception priority mechanism, along with the requirement that synchronous interrupts be generated in program order, that guarantees that at any given time, there exists for consideration only one of the synchronous interrupt types listed in item 1 of Section 7.8.2 on page 176. The exception priority mechanism also prevents certain debug exceptions from existing in combination with certain other synchronous interrupt-generating exceptions.

Because unaligned *Load* and *Store* instructions, or *Load Multiple*, *Store Multiple*, *Load String*, and *Store Sting* instructions may be broken up into multiple, smaller accesses, and these accesses may be performed in any order. The exception priority mechanism applies to each of the multiple storage accesses in the order they are performed by the implementation.

This section does not define the permitted setting of multiple exceptions for which the corresponding interrupt types are disabled. The generation of exceptions for which the corresponding interrupt types are disabled will have no effect on the generation of other exceptions for which the corresponding interrupt types are enabled. Conversely, if a particular exception for which the corresponding interrupt type is enabled is shown in the following sections to be of a higher priority than another exception, it will prevent the setting of that other exception, independent of whether that other exception's corresponding interrupt type is enabled or disabled.

Except as specifically noted, only one of the exception types listed for a given instruction type will be permitted to be generated at any given time. The priority of the exception types are listed in the following sections ranging from highest to lowest, within each instruction type.

Note
Some exception types may even be mutually exclusive of each other and could otherwise be considered the same priority. In these cases, the exceptions are listed in the order suggested by the sequential execution model.

7.9.1 Exception Priorities for Defined Instructions

7.9.1.1 Exception Priorities for Defined Floating-Point Load and Store Instructions

The following prioritized list of exceptions may occur as a result of the attempted execution of any defined *Floating-Point Load* and *Store* instruction.

1. Debug (Instruction Address Compare)
2. Instruction TLB Error
3. Instruction Storage Interrupt (all types)
4. Program (Illegal Instruction)
5. Floating-Point Unavailable
6. Program (Unimplemented Operation)
7. Data TLB Error
8. Data Storage (all types)
9. Alignment
10. Debug (Data Address Compare, Data Value Compare)
11. Debug (Instruction Complete)

If the instruction is causing both a Debug (Instruction Address Compare) and a Debug (Data Address Compare) or Debug (Data Value Compare), and is not causing any of the exceptions listed in items 2-9, it is permissible for both exceptions to be generated and recorded in the Debug Status Register. A single Debug interrupt will result.

7.9.1.2 Exception Priorities for Other Defined Load and Store Instructions and Defined Cache Management Instructions

The following prioritized list of exceptions may occur as a result of the attempted execution of any other defined *Load* or *Store* instruction, or defined *Cache Management* instruction.

1. Debug (Instruction Address Compare)
2. Instruction TLB Error
3. Instruction Storage Interrupt (all types)
4. Program (Illegal Instruction)
5. Program (Privileged Instruction) (***dcbi*** and ***dcbie*** only)
6. Program (Unimplemented Operation)
7. Data TLB Error
8. Data Storage (all types)
9. Alignment
10. Debug (Data Address Compare, Data Value Compare)
11. Debug (Instruction Complete)

If the instruction is causing both a Debug (Instruction Address Compare) and a Debug (Data Address Compare) or Debug (Data Value Compare), and is not causing any of the exceptions listed in items 2-9, it is permissible for both exceptions to be generated and recorded in the Debug Status Register. A single Debug interrupt will result.

7.9.1.3 Exception Priorities for Other Defined Floating-Point Instructions

The following prioritized list of exceptions may occur as a result of the attempted execution of any defined floating-point instruction other than a load or store.

1. Debug (Instruction Address Compare)
2. Instruction TLB Error
3. Instruction Storage Interrupt (all types)
4. Program (Illegal Instruction)
5. Floating-Point Unavailable
6. Program (Unimplemented Operation)
7. Program (Floating-point Enabled)
8. Debug (Instruction Complete)

7.9.1.4 Exception Priorities for Defined Privileged Instructions

The following prioritized list of exceptions may occur as a result of the attempted execution of any defined privileged instruction, except ***dcbi***[***e***], ***rfi***, and ***rfci*** instructions.

1. Debug (Instruction Address Compare)
2. Instruction TLB Error
3. Instruction Storage Interrupt (all types)
4. Program (Illegal Instruction)
5. Program (Privileged Instruction)
6. Program (Unimplemented Operation)
7. Debug (Instruction Complete)

For ***mtmsr***, ***mtspr*** (DBCR0, DBCR1, DBCR2), ***mtspr*** (TCR), and ***mtspr*** (TSR), if they are not causing Debug (Instruction Address Compare) nor Program (Privileged Instruction) exceptions, it is possible that they are simultaneously enabling (via mask bits) multiple existing exceptions (and at the same time possibly causing a Debug (Instruction Complete) exception). When this occurs, the interrupts will be handled in the order defined by Section 7.8.2 on page 176.

7.9.1.5 Exception Priorities for Defined Trap Instructions

The following prioritized list of exceptions may occur as a result of the attempted execution of a defined *Trap* instruction.

1. Debug (Instruction Address Compare)
2. Instruction TLB Error
3. Instruction Storage Interrupt (all types)
4. Program (Illegal Instruction)
5. Program (Unimplemented Operation)
6. Debug (Trap)
7. Program (Trap)
8. Debug (Instruction Complete)

If the instruction is causing both a Debug (Instruction Address Compare) and a Debug (Trap), and is not causing any of the exceptions listed in items 2-5, it is permissible for both exceptions to be generated and recorded in the DBSR. A single Debug interrupt will result.

7.9.1.6 Exception Priorities for Defined System Call Instruction

The following prioritized list of exceptions may occur as a result of the attempted execution of a defined *System Call* instruction.

1. Debug (Instruction Address Compare)
2. Instruction TLB Error
3. Instruction Storage Interrupt (all types)
4. Program (Illegal Instruction)
5. Program (Unimplemented Operation)
6. System Call
7. Debug (Instruction Complete)

7.9.1.7 Exception Priorities for Defined Branch Instructions

The following prioritized list of exceptions may occur as a result of the attempted execution of any defined branch instruction.

1. Debug (Instruction Address Compare)
2. Instruction TLB Error
3. Instruction Storage Interrupt (all types)
4. Program (Illegal Instruction)
5. Program (Unimplemented Operation)
6. Debug (Branch Taken)
7. Debug (Instruction Complete)

If the instruction is causing both a Debug (Instruction Address Compare) and a Debug (Branch Taken), and is not causing any of the exceptions listed in items 2-5, it is permissible for both exceptions to be generated and recorded in the Debug Status Register. A single Debug interrupt will result.

7.9.1.8 Exception Priorities for Defined Return From Interrupt Instructions

The following prioritized list of exceptions may occur as a result of the attempted execution of a defined *Return From Interrupt* or *Return From Critical Interrupt* instruction.

1. Debug (Instruction Address Compare)
2. Instruction TLB Error
3. Instruction Storage Interrupt (all types)
4. Program (Illegal Instruction)
5. Program (Privileged Instruction)
6. Program (Unimplemented Operation)
7. Debug (Return From Interrupt)
8. Debug (Instruction Complete)

If the ***rfi*** or ***rfci*** instruction is causing both a Debug (Instruction Address Compare) and a Debug (Return From Interrupt), and is not causing any of the exceptions listed in items 2-5, it is permissible for both exceptions to be generated and recorded in the Debug Status Register. A single Debug interrupt will result.

7.9.1.9 Exception Priorities for Other Defined Instructions

The following prioritized list of exceptions may occur as a result of the attempted execution of all other instructions not listed above.

1. Debug (Instruction Address Compare)
2. Instruction TLB Error
3. Instruction Storage Interrupt (all types)
4. Program (Illegal Instruction)
5. Program (Unimplemented Operation)
6. Debug (Instruction Complete)

7.9.2 Exception Priorities for Allocated Instructions

7.9.2.1 Exception Priorities for Allocated Load and Store Instructions

The following prioritized list of exceptions may occur as a result of the attempted execution of any allocated load or store instruction.

1. Debug (Instruction Address Compare)
2. Instruction TLB Error
3. Instruction Storage Interrupt (all types)
4. Program (Illegal Instruction)
5. Floating-Point Unavailable
   Auxiliary Processor Unavailable
6. Program (Privileged Instruction)
7. Program (Unimplemented Operation)
8. Data TLB Error
9. Data Storage (all types)
10. Alignment
11. Debug (Data Address Compare, Data Value Compare)
12. Debug (Instruction Complete)

If the instruction is causing both a Debug (Instruction Address Compare) and a Debug (Data Address Compare) or Debug (Data Value Compare), and is not causing any of the exceptions listed in items 2-9, it is permissible for both exceptions to be generated and recorded in the Debug Status Register. A single Debug interrupt will result.

7.9.2.2 Exception Priorities for Other Allocated Instructions

The following prioritized list of exceptions may occur as a result of the attempted execution of any allocated instruction that is not a load or store.

1. Debug (Instruction Address Compare)
2. Instruction TLB Error
3. Instruction Storage Interrupt (all types)
4. Program (Illegal Instruction)
5. Floating-Point Unavailable
   Auxiliary Processor Unavailable
6. Program (Privileged Instruction)
7. Program (Unimplemented Operation)
8. Program (Floating-point Enabled)
   Program (Auxiliary Processor Enabled)
9. Debug (Instruction Complete)

7.9.3 Exception Priorities for Preserved Instructions

7.9.3.1 Exception Priorities for Preserved Load, Store, Cache Management, and TLB Management Instructions

The following prioritized list of exceptions may occur as a result of the attempted execution of any preserved load or store instruction.

1. Debug (Instruction Address Compare)
2. Instruction TLB Error
3. Instruction Storage Interrupt (all types)
4. Program (Illegal Instruction)
5. Program (Privileged Instruction)
6. Program (Unimplemented Operation)
7. Data TLB Error
8. Data Storage (all types)
9. Alignment
10. Debug (Data Address Compare, Data Value Compare)
11. Debug (Instruction Complete)

If the instruction is causing both a Debug (Instruction Address Compare) and a Debug (Data Address Compare) or Debug (Data Value Compare), and is not causing any of the exceptions listed in items 2-9, it is permissible for both exceptions to be generated and recorded in the Debug Status Register. A single Debug interrupt will result.

7.9.3.2 Exception Priorities for Other Preserved Instructions

The following prioritized list of exceptions may occur as a result of the attempted execution of any preserved instruction that is not a load or store.

1. Debug (Instruction Address Compare)
2. Instruction TLB Error
3. Instruction Storage Interrupt (all types)
4. Program (Illegal Instruction)
5. Program (Privileged Instruction)
6. Program (Unimplemented Operation)
7. Debug (Instruction Complete)

7.9.4 Exception Priorities for Reserved Instructions

The following prioritized list of exceptions may occur as a result of the attempted execution of any reserved instruction.

1. Debug (Instruction Address Compare)
2. Instruction TLB Error
3. Instruction Storage Interrupt (all types)
4. Program (Illegal Instruction)

Chapter 8 Timer Facilities

8.1 Overview

The Time Base (TB), Decrementer (DEC), Fixed-Interval Timer (FIT), and Watchdog Timer (WDT) provide timing functions for the system. All of these must be initialized during start-up.

- The Time Base provides a long-period counter driven by an implementation-dependent frequency.

- The Decrementer, a counter that is updated at the same rate as the Time Base, provides a means of signaling an exception after a specified amount of time has elapsed unless:

  – the Decrementer is altered by software in the interim, or
  – the Time Base update frequency changes

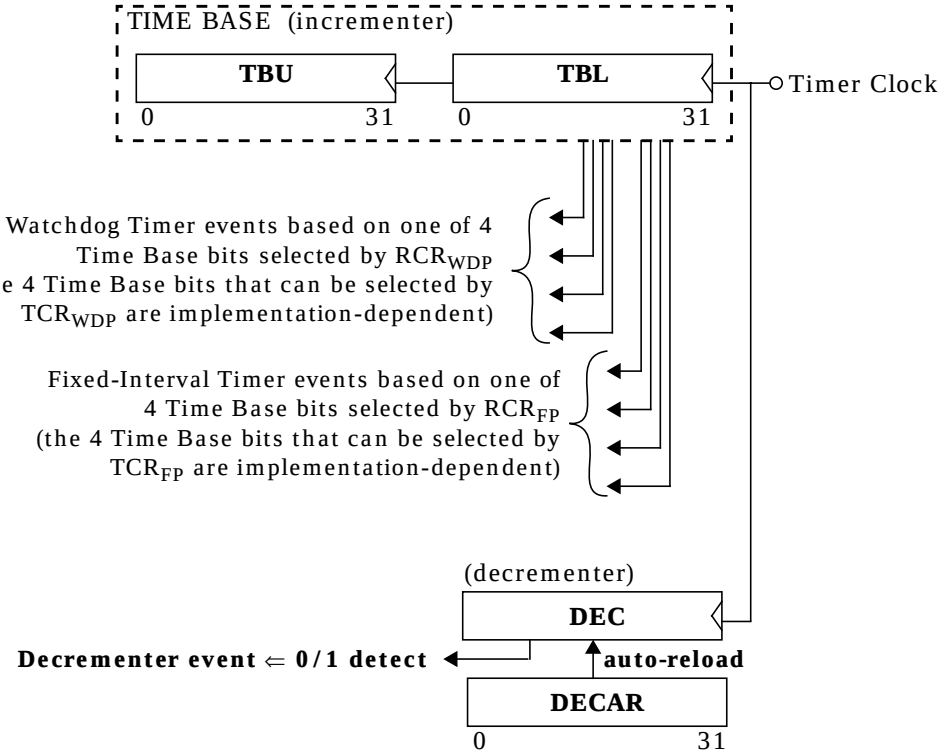
  The Decrementer is typically used as a general-purpose software timer.

- The Fixed-Interval Timer is really a selected bit of the Time Base, which provides a means of signalling an exception whenever the selected bit transitions from 0 to 1, in a repetitive fashion. The Fixed-Interval Timer is typically used to trigger periodic system maintenance functions. Software may select one of four bits in the Time Base to serve as the Fixed-Interval Timer. Which bits may be selected is implementation-dependent.

- The Watchdog Timer is also a selected bit of the Time Base, which provides a means of signalling a critical class exception whenever the selected bit transitions from 0 to 1. In addition, if software does not respond in time to the initial exception (by clearing the associated status in the Timer Status Register (TSR) prior to the next expiration of the Watchdog Timer interval), then a Watchdog Timer-generated processor reset may result, if so enabled. The Watchdog Timer is typically used to provide a system error recovery function.

The relationship of these Timer facilities to each other is illustrated in Figure 8-1 below.

Figure 8-1. Relationship of Timer Facilities to Time Base



8.2 Timer Control Register

The Timer Control Register (TCR) is a 32-bit register. Timer Control Register bits are numbered 32 (most-significant bit) to 63 (least-significant bit). The Timer Control Register controls Decrementer (see Section 8.5), Fixed-Interval Timer (see Section 8.6), and Watchdog Timer (see Section 8.7) options. Table 8-1 specifies the bit definitions of the Timer Control Register.

The contents of the Timer Control Register can be read into bits 32:63 of a GPR(RT) using *mfspr RT,TCR*, setting bits 0:31 of GPR(RT) to zero. The contents of bits 32:63 of GPR(RS) can be written to the Timer Control Register using *mtspr TCR,RS*.

Table 8-1. Timer Control Register Definition

Bit(s)	Description
32:33	***Watchdog Timer Period*** (WP) (See Section 8.7) Specifies one of 4 bit locations of the Time Base used to signal a Watchdog Timer exception on a transition from 0 to 1. The 4 Time Base bits that can be specified to serve as the Watchdog Timer period are implementation-dependent.

Bit(s)	Description	
34:35	***Watchdog Timer Reset Control*** (WRC) (See Section 8.7)	
	=00	No Watchdog Timer reset will occur TCR_{WRC} resets to 0b00. This field may be set by software, but cannot be cleared by software (except by a software-induced reset)
	=01 =10 =11	Force processor to be reset on second time-out of Watchdog Timer. The exact function of any of these settings is implementation-dependent. See the User's Manual for the implementation for further details.
	The Watchdog Timer Reset Control field is cleared to zero by processor reset. These bits are set only by software; however, hardware does not allow software to clear these bits once they have been set. Once software has written a 1 to one of these bits, that bit remains a 1 until a reset occurs. This is to prevent errant code from disabling the Watchdog reset function.	
36	***Watchdog Timer Interrupt Enable*** (WIE) (See Section 8.7)	
	=0	Disable Watchdog Timer interrupt
	=1	Enable Watchdog Timer interrupt
37	***Decrementer Interrupt Enable*** (DIE) (See Section 8.5)	
	=0	Disable Decrementer interrupt
	=1	Enable Decrementer interrupt
38:39	***Fixed-Interval Timer Period*** (FP) (See Section 8.6) Specifies one of 4 bit locations of the Time Base used to signal a Fixed-Interval Timer exception on a transition from 0 to 1. The 4 Time Base bits that can be specified to serve as the Fixed-Interval Timer period are implementation-dependent.	
40	***Fixed-Interval Timer Interrupt Enable*** (FIE) (See Section 8.6)	
	=0	Disable Fixed-Interval Timer interrupt
	=1	Enable Fixed-Interval Timer interrupt
41	***Auto-Reload Enable*** (ARE)	
	=0	Disable auto-reload of the Decrementer Decrementer exception is presented (i.e. TSR_{DIS} is set to 1) when the Decrementer is decremented from a value of 0x0000_0001. The next value placed in the Decrementer is the value 0x0000_0000. The Decrementer then stops decrementing. If MSR_{EE}=1, TCR_{DIE}=1, and TSR_{DIS}=1, a Decrementer interrupt is taken. Software must reset TSR_{DIS}.
	=1	Enable auto-reload of the Decrementer Decrementer exception is presented (i.e. TSR_{DIS} is set to 1) when the Decrementer is decremented from a value of 0x0000_0001. The contents of the Decrementer Auto-Reload Register is placed in the Decrementer. The Decrementer resumes decrementing. If MSR_{EE}=1, TCR_{DIE}=1, and TSR_{DIS}=1, a Decrementer interrupt is taken. Software must reset TSR_{DIS}.
42	Allocated for implementation-dependent use	
	Implementation Note This bit is allocated for Book E implementations that support a mode that causes the Decrementer to emulate the PowerPC Decrementer.	
43:63	Reserved	

8.3 Timer Status Register

The Timer Status Register (TSR) is a 32-bit register. Timer Status Register bits are numbered 32 (most-significant bit) to 63 (least-significant bit). The Timer Status Register contains status on timer events and the most recent Watchdog Timer-initiated processor reset.

The Timer Status Register is set via hardware, and read and cleared via software. The contents of the Timer Status Register can be read into bits 32:63 of a GPR(RT) using *mfspr RT,TSR*, setting bits 0:31 of GPR(RT) to zero. Bits in the Timer Status Register can be cleared using *mtspr TSR,RS*. Clearing is done by writing bits 32:63 of a General Purpose Register to the Timer Status Register with a 1 in any bit position that is to be cleared and 0 in all other bit positions. The write-data to the Timer Status Register is not direct data, but a mask. A 1 causes the bit to be cleared, and a 0 has no effect.

Table 8-2. Timer Status Register Definition

Bit(s)	Description	
32	***Enable Next Watchdog Timer*** (ENW) (See Section 8.7)	
	=0	Action on next Watchdog Timer time-out is to set TSR_0
	=1	Action on next Watchdog Timer time-out is governed by TSR_{WIS}
33	***Watchdog Timer Interrupt Status*** (WIS) (See Section 8.7)	
	=0	A Watchdog Timer event has not occurred.
	=1	A Watchdog Timer event has occurred. When MSR_{CE}=1 and TCR_{WIE}=1, a Watchdog Timer interrupt is taken.
34:35	***Watchdog Timer Reset Status*** (WRS) (See Section 8.7) These two bits are set to one of three values when a reset is caused by the Watchdog Timer. These bits are undefined at power-up.	
	=00	No Watchdog Timer reset has occurred.
	=01	Implementation-dependent reset information.
	=10	Implementation-dependent reset information.
	=11	Implementation-dependent reset information.
36	***Decrementer Interrupt Status*** (DIS) (See Section 8.5)	
	=0	A Decrementer event has not occurred.
	=1	A Decrementer event has occurred. When MSR_{EE}=1 and TCR_{DIE}=1, a Decrementer interrupt is taken.
37	***Fixed-Interval Timer Interrupt Status*** (FIS) (See Section 8.6)	
	=0	A Fixed-Interval Timer event has not occurred.
	=1	A Fixed-Interval Timer event has occurred. When MSR_{EE}=1 and TCR_{FIE}=1, a Fixed-Interval Timer interrupt is taken.
38:63	Reserved	

8.4 Time Base

8.4.1 Overview

The Time Base (TB) is composed of two 32-bit registers, the Time Base Upper (TBU) concatenated on the right with the Time Base Lower (TBL). Time Base Upper bits are numbered 32 (most-significant bit) to 63 (least-significant bit). Time Base Lower bits are numbered 32 (most-significant bit) to 63 (least-significant bit). The Time Base 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.

The Time Base provides timing functions for the system. The Time Base is a volatile resource and must be initialized during start-up.

The Time Base provides a long-period counter driven by an implementation-dependent frequency.

The contents of the Time Base Upper can be read into bits 32:63 of a General Purpose Register using *mfspr RT,TBU*, setting bits 0:31 of GPR(RT) to an undefined value. The contents of bits 32:63 of GPR(RS) can be written to the Time Base Upper using *mtspr TBU,RS*.

The contents of the Time Base Lower can be read into bits 32:63 of a General Purpose Register using *mfspr RT,TBL*, setting bits 0:31 of GPR(RT) to an undefined value. The contents of bits 32:63 of GPR(RS) can be written to the Time Base Lower using *mtspr TBL,RS*.

There is no automatic initialization of the Time Base; system software must perform this initialization.

The Time Base Lower increments until its value becomes 0xFFFF_FFFF (2^{32}–1). At the next increment, its value becomes 0x0000_0000 and Time Base Upper is incremented. This process continues until the value in the Time Base Upper becomes 0xFFFF_FFFF and the value in the Time Base Lower becomes 0xFFFF_FFFF, or the value in the Time Base is interpreted as 0xFFFF_FFFF_FFFF_FFFF (2^{64}–1). At the next increment, the value in the Time Base Upper becomes 0x0000_0000 and the value in the Time Base Lower becomes 0x0000_0000. There is no interrupt or other indication when this occurs.

The period of the Time Base depends on the driving frequency. As an order of magnitude example, suppose that the Time Base is driven by a frequency of 100MHz divided by 32. Then the period of the Time Base would be

$$T_{TB} = 2^{64} \times 32 / 100 \text{ MHz} = 5.90 \times 10^{12} \text{ seconds}$$

which is approximately 187,000 years.

The Time Base must be implemented such that the following requirements are satisfied.

- Loading a General Purpose Register from the Time Base shall have no effect on the accuracy of the Time Base.

- Storing a General Purpose Register to the Time Base shall replace the value in the Time Base with the value in the General Purpose Register.

Book E does not specify a relationship between the frequency at which the Time Base is updated and other frequencies, such as the CPU clock or bus clock in a Book E system. The Time Base update frequency is not required to be constant. What *is* required, so that system software can keep time of day and operate interval timers, is one of the following.

- The system provides an (implementation-dependent) interrupt to software whenever the update frequency of the Time Base changes, and a means to determine what the current update frequency is.

- The update frequency of the Time Base is under the control of the system software.

Implementations must provide a means for either preventing the Time Base from incrementing or preventing the Time Base from being read in problem state (MSR_{PR}=1). If the means is under software control, the Time Base must be accessible only in privileged state (MSR_{PR}=0).

Architecture Note
Disabling the Time Base or making reading the Time Base privileged prevents the Time Base from being used to implement a 'covert channel' in a secure system.

The requirements stated above for the Time Base apply also to any other SPRs that measure time and can be read in problem state (e.g., Performance Monitor registers).

Programming Note
If the operating system initializes the Time Base on power-on to some reasonable value and the update frequency of the Time Base is constant, the Time Base can be used as a source of values that increase at a constant rate, such as for time stamps in trace entries.

Even if the update frequency is not constant, values read from the Time Base are monotonically increasing (except when the Time Base wraps from 2^{64}–1 to 0). If a trace entry is recorded each time the update frequency changes, the sequence of Time Base values can be post-processed to become actual time values.

Successive readings of the Time Base may return identical values.

See Section 8.4.4 on page 191 for ways to compute time of day in POSIX format from the Time Base.

Architecture Note
It is intended that the Time Base be useful for timing reasonably short sequences of code (a few hundred instructions) and for low-overhead time stamps for tracing. The Time Base should not 'tick' faster than the CPU instruction clock. Driving the Time Base directly from the CPU instruction clock is probably finer granularity than necessary; the instruction clock divided by 8, 16, or 32 would be more appropriate.

8.4.2 Writing the Time Base

It is not possible to write the entire 64-bit Time Base using a single instruction. The Time Base can be written by a sequence such as:

```
lwz    Rx,upper   # load 64-bit value for
lwz    Ry,lower   #   TB into Rx and Ry
addi   Rz,R0,0
mtspr  TBL,Rz     # force TBL to 0
mtspr  TBU,Rx     # set TBU
mtspr  TBL,Ry     # set TBL
```

Provided that no interrupts occur while the last three instructions are being executed, loading 0 into Time Base Lower prevents the possibility of a carry from Time Base Lower to Time Base Upper while the Time Base is being initialized.

8.4.3 Reading the Time Base

It is not possible to read the entire 64-bit Time Base in a single instruction. ***mfspr RT,TBL*** moves from the lower half of the Time Base (TBL) to a GPR, and ***mfspr RT,TBU*** extended mnemonic moves from the upper half (TBU) to a GPR. Because of the possibility of a carry from Time Base Lower to Time Base Upper occurring between reads of Time Base Lower and Time Base Upper, a sequence such as the following is necessary to read the Time Base.

```
loop:
   mfspr  Rx,TBU        #load from TBU
   mfspr  Ry,TBL        #load from TBL
   mfspr  Rz,TBU        #load from TBU
   cmp    cr0,0,Rz,Rx   #see if 'old' = 'new'
   bc     4,2,loop      #loop if carry occurred
```

The comparison and loop are necessary to ensure that a consistent pair of values has been obtained.

8.4.4 Computing Time of Day from the Time Base

Since the update frequency of the Time Base is implementation-dependent, the algorithm for converting the current value in the Time Base to time of day is also implementation-dependent.

As an example, assume that the Time Base is incremented at a constant rate of once for every 32 cycles of a 100 MHz CPU instruction clock. What is wanted is the pair of 32-bit values comprising a POSIX standard clock:[1] the number of whole seconds that have passed since midnight January 1, 1970, and the remaining fraction of a second expressed as a number of nanoseconds.

Assume that:

1. Described in POSIX Draft Standard P1003.4/D12, *Draft Standard for Information Technology -- Portable Operating System Interface (POSIX) -- Part 1: System Application Program Interface (API) - Amendment 1: Real-time Extension [C Language].* Institute of Electrical and Electronics Engineers, Inc., Feb. 1992.

- The value 0 in the Time Base represents the start time of the POSIX clock (if this is not true, a simple 64-bit subtraction will make it so).

- The integer constant *ticks_per_sec* contains the value

  100MHz / 32 = 3,125,000

  which is the number of times the Time Base is updated each second.

- The integer constant *ns_adj* contains the value

  1,000,000,000 / 3,125,000 = 320

  which is the number of nanoseconds per tick of the Time Base.

The POSIX clock can be computed with an instruction sequence such as this:

```
# Read Time Base
loop:
    mfspr  Rx,TBU       #load from TBU into Rx
    mfspr  Ry,TBL       #load from TBL into Ry
    mfspr  Rz,TBU       #load from TBU into Rz
    cmp    CR0,0,Rz,Rx  #see if 'old TBU' = 'new TBU'
    bc     4,2,loop     #loop if carry occurred
    rldimi Ry,Rx,32,0   #splice TBU & TBL into Ry
#
# Compute POSIX clock
#
    lwz    Rx,ticks_per_sec
    divd   Rz,Ry,Rx     #Rz = whole seconds
    stw    Rz,posix_sec
    mulld  Rz,Rz,Rx     #Rz = quotient * divisor
    sub    Rz,Ry,Rz     #Rz = excess ticks
    lwz    Rx,ns_adj
    mulld  Rz,Rz,Rx     #Rz = excess nanoseconds
    stw    Rz,posix_ns
```

In the absence of a ***divd*** instruction (see Appendix A, “Guidelines for 32-bit Book E”, on page 371), direct implementation of the algorithm given above is awkward.[1]

Such division can be avoided entirely if a time of day clock in POSIX format is updated at least once each second.

Assume that:

- The operating system maintains the following variables:

  – *posix_tb* (64 bits)
  – *posix_sec* (32 bits)
  – *posix_ns* (32 bits)

  These variables hold the value of the Time Base and the computed POSIX second and nanosecond values from the last time the POSIX clock was computed.

1. See D. E. Knuth, *The Art of Computer Programming, Volume 2, Seminumerical Algorithms*, Section 4.3.1, Algorithm D. Addison-Wesley, 1981.

- The operating system arranges for an interrupt (see Chapter 8 on page 185) to occur at least once per second, at which time it recomputes the POSIX clock values.

- The integer constant *billion* contains the value 1,000,000,000.

The POSIX clock can be computed with an instruction sequence such as this:

```
loop:
   mfspr  Rx,TBU         #Rx = TBU
   mfspr  Ry,TBL         #Ry = TBL
   mfspr  Rz,TBU         #Rz = 'new' TBU value
   cmp    CR0,0,Rz,Rx    #see if 'old' = 'new'
   bc     4,2,loop       #loop if carry occurred
#      now have 64-bit TB in Rx and Ry
   lwz    Rz,posix_tb+4
   sub    Rz,Ry,Rz       #Rz = delta in ticks
   lwz    Rw,ns_adj
   mullw  Rz,Rz,Rw       #Rz = delta in ns
   lwz    Rw,posix_ns
   add    Rz,Rz,Rw       #Rz = new ns value
   lwz    Rw,billion
   cmp    CR0,0,Rz,Rw    #see if past 1 second
   bc     12,0,nochange  #branch if not
   sub    Rz,Rz,Rw       #adjust nanoseconds
   lwz    Rw,posix_sec
   addi   Rw,Rw,1        #adjust seconds
   stw    Rw,posix_sec   #store new seconds
nochange:
   stw    Rz,posix_ns    #store new ns
   stw    Rx,posix_tb    #store new time base
   stw    Ry,posix_tb+4
```

Note that the upper half of the Time Base does not participate in the calculation to determine the new POSIX time of day. This is correct as long as the time change does not exceed one second.

Non-constant update frequency

In a system in which the update frequency of the Time Base may change over time, it is not possible to convert an isolated Time Base value into time of day. Instead, a Time Base value has meaning only with respect to the current update frequency and the time of day that the update frequency was last changed. Each time the update frequency changes, either the system software is notified of the change via an interrupt (see Chapter 7 on page 143), or the change was instigated by the system software itself. At each such change, the system software must compute the current time of day using the old update frequency, compute a new value of *ticks_per_sec* for the new frequency, and save the time of day, Time Base value, and tick rate. Subsequent calls to compute time of day use the current Time Base value and the saved data.

8.5 Decrementer

The Decrementer (DEC) is a 32-bit register. Decrementer bits are numbered 32 (most-significant bit) to 63 (least-significant bit).

The contents of the Decrementer can be read into bits 32:63 of a General Purpose Register using *mfspr RT,DEC*, setting bits 0:31 of GPR(RT) to zero. The contents of bits 32:63 of GPR(RS) can be written to the Decrementer using *mtspr DEC,RS*.

The Decrementer Auto-Reload Register (DECAR) is a 32-bit register. Decrementer Auto-Reload Register bits are numbered 32 (most-significant bit) to 63 (least-significant bit). The Decrementer Auto-Reload Register is provided to support the auto-reload feature of the Decrementer.

The contents of the Decrementer Auto-Reload Register cannot be read. The contents of bits 32:63 of GPR(RS) can be written to the Decrementer Auto-Reload Register using *mtspr DECAR,RS*.

The Decrementer decrements at the same rate that the Time Base increments. A Decrementer event occurs when a decrement occurs on a Decrementer value of 0x0000_0001. Upon the occurrence of a Decrementer event, the Decrementer has the following basic modes of operation.

Decrement to one and stop on zero

If TCR_{ARE}=0, TSR_{DIS} is set to 1, the value 0x0000_0000 is then placed into the DEC, and the Decrementer stops decrementing.

If enabled by TCR_{DIE}=1 and MSR_{EE}=1, a Decrementer interrupt is taken. See Section 7.6.11 on page 167 for details of register behavior caused by the Decrementer interrupt.

Decrement to one and auto-reload

If TCR_{ARE}=1, TSR_{DIS} is set to 1, the contents of the Decrementer Auto-Reload Register is then placed into the DEC, and the Decrementer continues decrementing from the reloaded value.

If enabled by TCR_{DIE}=1 and MSR_{EE}=1, a Decrementer interrupt is taken. See Section 7.6.11 on page 167 for details of register behavior caused by the Decrementer interrupt.

The Decrementer interrupt handler must reset TSR_{DIS} in order to avoid taking another, redundant Decrementer interrupt once MSR_{EE} is re-enabled (assuming TCR_{DIE} is not cleared instead). This is done by writing a word to Timer Status Register using ***mtspr*** with a 1 in the bit corresponding to TSR_{DIS} (and any other bits that are to be cleared) and 0 in all other bits. The write-data to the Timer Status Register is not direct data, but a mask. A 1 causes the bit to be cleared, and a 0 has no effect.

Forcing the Decrementer to 0 using the ***mtspr*** instruction will not cause a Decrementer exception; however, decrementing which was in progress at the instant of the ***mtspr*** may cause the exception. To eliminate the Decrementer as a source of exceptions, set TCR_{DIE} to 0 (clear the Decrementer Interrupt Enable bit).

If it is desired to eliminate all Decrementer activity, the procedure is as follows:

1. Write 0 to TCR_{DIE}. This will prevent Decrementer activity from causing

exceptions.

2. Write 0 to TCR_{ARE} to disable the Decrementer auto-reload.

3. Write 0 to Decrementer. This will halt Decrementer decrementing. While this action will not cause a Decrementer exception to be set in TSR_{DIS}, a near simultaneous decrement may have done so.

4. Write 1 to TSR_{DIS}. This action will clear TSR_{DIS} to 0 (see Section 8.3 on page 188). This will clear any Decrementer exception which may be pending. Because the Decrementer is frozen at zero, no further Decrementer events are possible.

If the auto-reload feature is disabled (TCR_{ARE}=0), then once the Decrementer decrements to zero, it will stay there until software reloads it using the ***mtspr*** instruction.

On reset, TCR_{ARE} is set to 0. This disables the auto-reload feature.

Book E has changed the definition of the Decrementer from the PowerPC Architecture definition. However, Book E permits implementations of the architecture to provide implementation-dependent facilities to enable emulation of the PowerPC Architecture definition of the Decrementer for compatibility with legacy systems. There are a number of approaches to providing this compatibility support, and hence Book E allows implementation-dependent discretion. A new mode which causes the Decrementer to operate in the PowerPC-compatible mode would be required by implementations that support such compatibility. A mode bit that enables this new mode is recommended to be placed in the Timer Control Register. The PowerPC Decrementer operates continuously, wrapping from 0 to 0xFFFF_FFFF and generates an interrupt on every transition of bit 0 of the Decrementer from 0 to 1, including such a transition caused by a *mtspr DEC,RS*. A Book E implementation that supports this PowerPC Decrementer mode would be required to also provide facilities that enable the Book E Decrementer behavior.

8.6 Fixed-Interval Timer

The Fixed-Interval Timer (FIT) is a mechanism for providing timer interrupts with a repeatable period, to facilitate system maintenance. It is similar in function to an auto-reload Decrementer, except that there are fewer selections of interrupt period available. The Fixed-Interval Timer exception occurs on 0 to 1 transitions of a selected bit from the Time Base (see Section 8.2 on page 186).

The Fixed-Interval Timer exception is logged by TSR_{FIS}. A Fixed-Interval Timer interrupt will occur if TCR_{FIE} and MSR_{EE} are enabled. See Section 7.6.12 on page 168 for details of register behavior caused by the Fixed-Interval Timer interrupt.

The interrupt handler must reset TSR_{FIS} via software, in order to avoid another Fixed-Interval Timer interrupt once MSR_{EE} is re-enabled (assuming TCR_{FIE} is not cleared instead). This is done by writing a word to the Timer Status Register using ***mtspr*** with a 1 in the bit corresponding to TSR_{FIS} (and any other bits that are to be cleared) and 0 in all other bits. The write-data to the Timer Status Register is not direct data, but a mask. A 1 causes the bit to be cleared, and a “0” has no effect.

Note that a Fixed-Interval Timer exception will also occur if the selected Time Base bit transitions from 0 to 1 due to an *mtspr TBL,RS* that writes a 1 to the bit when its previous value was 0.

8.7 Watchdog Timer

The Watchdog Timer is a facility intended to aid system recovery from faulty software or hardware. Watchdog time-outs occur on 0 to 1 transitions of selected bits from the Time Base (see Section 8.2 on page 186).

When a Watchdog Timer time-out occurs while Watchdog Timer Interrupt Status is clear (TSR_{WIS} = 0) and the next Watchdog Time-out is enabled (TSR_{ENW} = 1), a Watchdog Timer exception is generated and logged by setting TSR_{WIS} to 1. This is referred to as a Watchdog Timer First Time Out. A Watchdog Timer interrupt will occur if enabled by TCR_{WIE} and MSR_{CE}. See Section 7.6.13 on page 169 for details of register behavior caused by the Watchdog Timer interrupt.

The interrupt handler must reset TSR_{WIS} via software, in order to avoid another Watchdog Timer interrupt once MSR_{CE} is re-enabled (assuming TCR_{WIE} is not cleared instead). This is done by writing a word to the Timer Status Register using ***mtspr*** with a 1 in the bit corresponding to TSR_{WIS} (and any other bits that are to be cleared) and a 0 in all other bits. The write-data to the Timer Status Register is not direct data, but a mask. A 1 causes the bit to be cleared, and a 0 has no effect.

Note that a Watchdog Timer exception will also occur if the selected Time Base bit transitions from 0 to 1 due to an *mtspr TBL,RS* that writes a 1 to the bit when its previous value was 0.

When a Watchdog Timer time-out occurs while TSR_{WIS} = 1 and TSR_{ENW} = 1, a processor reset occurs if it is enabled by a non-zero value of the Watchdog Reset Control field in the Timer Control Register (TCR_{WRC}). This is referred to as a Watchdog Timer Second Time Out. The assumption is that TSR_{WIS} was not cleared because the processor was unable to execute the Watchdog Timer interrupt handler, leaving reset as the only available means to restart the system. Note that once TCR_{WRC} has been set to a non-zero value, it cannot be reset by software; this feature prevents errant software from disabling the Watchdog Timer reset capability.

A more complete view of Watchdog Timer behavior is afforded by Figure 8-2 on page 197 and Table 8-3 on page 197, which describe the Watchdog Timer state machine and Watchdog Timer controls. The numbers in parentheses in the figure refer to the discussion of modes of operation which follow the table.

Figure 8-2. Watchdog State Machine

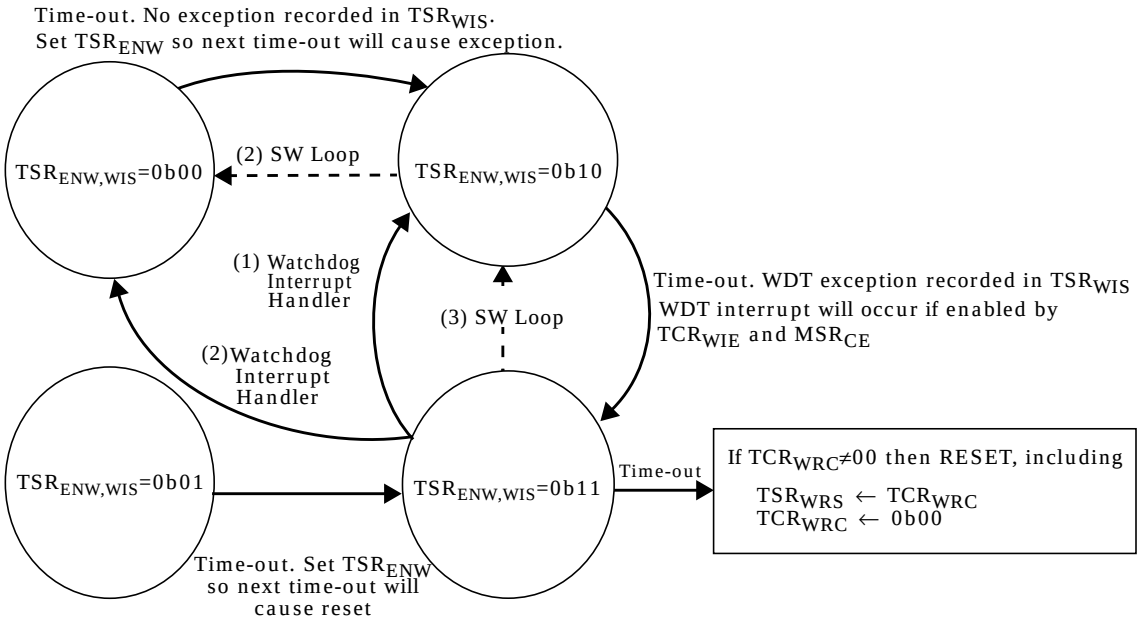


Table 8-3. Watchdog Timer Controls

Enable Next WDT (TSR$_{ENW}$)	WDT Status (TSR$_{WIS}$)	Action when timer interval expires
0	0	Set Enable Next Watchdog Timer (TSR$_{ENW}$=1).
0	1	Set Enable Next Watchdog Timer (TSR$_{ENW}$=1).
1	0	Set Watchdog Timer interrupt status bit (TSR$_{WIS}$=1). If Watchdog Timer interrupt is enabled (TCR$_{WIE}$=1 and MSR$_{CE}$=1), then interrupt.
1	1	Cause Watchdog Timer reset action specified by TCR$_{WRC}$. Reset will copy pre-reset TCR$_{WRC}$ into TSR$_{WRS}$, then clear TCR$_{WRC}$.

The controls described in the above table imply three different modes of operation that a programmer might select for the Watchdog Timer. Each of these modes assumes that TCR$_{WRC}$ has been set to allow processor reset by the Watchdog facility:

1. Always take the Watchdog Timer interrupt when pending, and never attempt to prevent its occurrence. In this mode, the Watchdog Timer interrupt caused by a first time-out is used to clear TSR$_{WIS}$ so a second time-out never occurs. TSR$_{ENW}$ is not cleared, thereby allowing the next time-out to cause another interrupt.

2. Always take the Watchdog Timer interrupt when pending, but avoid when possible. In this mode a recurring code loop of reliable duration (or perhaps a periodic interrupt handler such as the Fixed-Interval Timer interrupt handler) is used to repeatedly clear TSR$_{ENW}$ such that a first time-out exception is avoided, and thus no Watchdog Timer interrupt occurs. Once TSR$_{ENW}$ has been cleared, software has between one and two full Watchdog periods before a Watchdog exception will be posted in TSR$_{WIS}$. If this occurs before the soft-

ware is able to clear TSR_{ENW} again, a Watchdog Timer interrupt will occur. In this case, the Watchdog Timer interrupt handler will then clear both TSR_{ENW} and TSR_{WIS}, in order to (hopefully) avoid the next Watchdog Timer interrupt.

3. Never take the Watchdog Timer interrupt. In this mode, Watchdog Timer interrupts are disabled (via TCR_{WIE}=0), and the system depends upon a recurring code loop of reliable duration (or perhaps a periodic interrupt handler such as the Fixed-Interval Timer interrupt handler) to repeatedly clear TSR_{WIS} such that a second time-out is avoided, and thus no reset occurs. TSR_{ENW} is not cleared, thereby allowing the next time-out to set TSR_{WIS} again. The recurring code loop must have a period which is less than one Watchdog Timer period in order to guarantee that a Watchdog Timer reset will not occur.

8.8 Freezing the Timer Facilities

The debug mechanism provides a means of temporarily freezing the timers upon a debug event. Specifically, the Time Base and Decrementer can be frozen and prevented from incrementing/decrementing, respectively, whenever a debug event is set in the Debug Status Register. This allows a debugger to simulate the appearance of 'real time', even though the application has been temporarily 'halted' to service the debug event. See the description of bit 63 of the Debug Control Register 0 (Freeze Timers on Debug Event or $DBCR0_{FT}$) in Table 9-1 on page 210.

Chapter 9 Debug Facilities

9.1 Background

Book E provides debug facilities to enable hardware and software debug functions, such as instructions and data breakpoints and program single stepping. The debug facilities consist of a set of debug control registers (DBCR0, DBCR1, and DBCR2) described in Section 9.4.1, a set of address and data value compare registers (IAC1, IAC2, IAC3, IAC4, DAC1, DAC2, DVC1, and DVC2) described in Sections 9.4.3, 9.4.4, and 9.4.5, a Debug Status Register (DBSR) described is Section 9.4.2 for enabling and recording various kinds of debug events, and a special Debug interrupt type built into the interrupt mechanism (see Section 7.6.16 on page 172). The debug facilities also provide a mechanism for software-controlled processor reset, and for controlling the operation of the timers in a debug environment.

Access to the debug facilities using the ***mfspr*** and ***mtspr*** instructions, as well as the debug interrupt mechanism, are defined as part of Book E. In addition, implementations will typically include debug facilities, modes, and access mechanisms which are implementation-specific and defined as part of the User's Manual for the implementation. For example, implementations will typically provide access to the debug facilities via a dedicated interface such as the IEEE 1149.1 Test Access Port (JTAG).

Programming Note
There are two classes of debug exception types:

Type 1: exception before instruction
Type 2: exception after instruction

Almost all debug exceptions fall into the first category. That is, they all take the interrupt upon encountering an instruction having the exception without updating any architectural state (other than DBSR, CSRR0, CSRR1, MSR) for that instruction.

The CSRR0 for this type of exception points to the instruction that encountered the exception. This includes IAC, DAC, branch taken, etc.

The only exception which fall into the second category is the instruction complete debug exception. This exception is taken upon completing and updating one instruction and then pointing CSRR0 to the next instruction to execute.

To make forward progress for any Type 1 debug exception one does the following:

1. Software sets up Type 1 exceptions (e.g. branch taken debug exceptions) and then returns to normal program operation

2. Hardware takes critical debug interrupt upon the first branch taken debug exception, pointing to the branch with CSRR0.

3. Software, in the debug handler, sees the branch taken exception type, does whatever logging/analysis it wants to, then clears all debug event enables in the DBCR except for the instruction complete debug event enable.

4. Software does an ***rfci***.

5. Hardware would execute and complete one instruction (the branch taken in this case), and then take a critical debug interrupt with CSRR0 pointing to the target of the branch.

6. Software would see the instruction complete interrupt type. It clears the instruction complete event enable, then enables the branch taken interrupt event again.

7. Software does an ***rfci***.

8. Hardware resumes on the target of the taken branch and continues until another taken branch, in which case we end up at step 2 again.

This, at first, seems like a double tax (i.e. 2 debug interrupts for every instance of a Type 1 exception), but it doesn't seem like any other clean way to make forward progress on Type 1 debug exceptions. The only other way to avoid the double tax is to have the debug handler routine actually emulate the instruction pointed to for the Type 1 exceptions, determine the next instruction that would have been executed by the interrupted program flow and load the CSRR0 with that address and do an ***rfci***; this is probably not faster.

9.2 Internal Debug Mode

Debug events include such things as instruction and data breakpoints. These debug events cause status bits to be set in the Debug Status Register. The existence of a set bit in the Debug Status Register is considered a Debug exception. Debug exceptions, if enabled, will cause Debug interrupts.

There are two different mechanisms that control whether Debug interrupts are enabled. The first is the MSR_{DE} bit, and this bit must be set to 1 to enable Debug interrupts. The second mechanism is an enable bit in the Debug Control Register 0 (DBCR0). This bit is the Internal Debug Mode bit ($DBCR0_{IDM}$), and it must also be set to 1 to enable Debug interrupts.

When $DBCR0_{IDM}$=1, the processor is in Internal Debug Mode. In this mode, debug events will (if also enabled by MSR_{DE}) cause Debug interrupts. Software at the Debug interrupt vector location will thus be given control upon the occurrence of a debug event, and can access (via the normal instructions) all architected processor resources. In this fashion, debug monitor software can control the processor and gather status, and interact with debugging hardware connected to the processor.

When the processor is not in Internal Debug Mode ($DBCR0_{IDM}$=0), debug events may still occur and be recorded in the Debug Status Register. These exceptions may be monitored via software by reading the Debug Status Register (using ***mfspr***), or may eventually cause a Debug interrupt if later enabled by setting $DBCR0_{IDM}$=1 (and MSR_{DE}=1). Processor behavior when debug events occur while $DBCR0_{IDM}$=0 is implementation-dependent. The remainder of this chapter discusses processor behavior with the presumption that $DBCR0_{IDM}$=1.

9.3 Debug Events

Debug events are used to cause Debug exceptions to be recorded in the Debug Status Register (see Table 9-4 on page 217). In order for a debug event to be enabled to set a Debug Status Register bit and thereby cause a Debug exception, the specific event type must be enabled by a corresponding bit or bits in the Debug Control Registers (DBCR0 defined in Table 9-1, DBCR1 defined in Table 9-2, or DBCR2 defined in Table 9-3) (in most cases; the Unconditional Debug Event (UDE) is an exception to this rule). Once a Debug Status Register bit is set, if Debug interrupts are enabled by MSR_{DE}, a Debug interrupt will be generated.

Certain debug events are not allowed to occur when MSR_{DE}=0. In such situations, no Debug exception occurs and thus no Debug Status Register bit is set. Other debug events may cause Debug exceptions and set Debug Status Register bits regardless of the state of MSR_{DE}. The associated Debug interrupts that result from such Debug exceptions will be delayed until MSR_{DE}=1, provided the exceptions have not been cleared from the Debug Status Register in the meantime.

Any time that a Debug Status Register bit is allowed to be set while MSR_{DE}=0, a special Debug Status Register bit, Imprecise Debug Event ($DBSR_{IDE}$), will also be set. $DBSR_{IDE}$ indicates that the associated Debug exception bit in the Debug Status Register was set while Debug interrupts were disabled via the $MMSR_{DE}$ bit. Debug interrupt handler software can use this bit to determine whether the address recorded in the Critical Save/Restore Register 0 should be interpreted as the address associated with the instruction causing the Debug exception, or sim-

ply the address of the instruction after the one which set the MSR_{DE} bit, thereby enabling the delayed Debug interrupt.

Debug interrupts are ordered with respect to other interrupt types (see Section 7.8 on page 174). Debug exceptions are prioritized with respect to other exceptions (see Section 7.9 on page 178).

There are eight types of debug events defined:

1. Instruction Address Compare debug events
2. Data Address Compare debug events
3. Trap debug events
4. Branch Taken debug events
5. Instruction Complete debug events
6. Interrupt Taken debug events
7. Return debug events
8. Unconditional debug events

9.3.1 Instruction Address Compare Debug Event

One or more Instruction Address Compare debug events (IAC1, IAC2, IAC3 or IAC4) occur if they are enabled and execution is attempted of an instruction at an address that meets the criteria specified in the DBCR0, DBCR1, IAC1, IAC2, IAC3, and IAC4 Registers.

Instruction Address Compare User/Supervisor Mode

$DBCR1_{IAC1US}$ specifies whether IAC1 debug events can occur in user mode or supervisor mode, or both.

$DBCR1_{IAC2US}$ specifies whether IAC2 debug events can occur in user mode or supervisor mode, or both.

$DBCR1_{IAC3US}$ specifies whether IAC3 debug events can occur in user mode or supervisor mode, or both.

$DBCR1_{IAC4US}$ specifies whether IAC4 debug events can occur in user mode or supervisor mode, or both.

Effective/Real Address Mode

$DBCR1_{IAC1ER}$ specifies whether effective addresses, real addresses, effective addresses and MSR_{IS}=0, or effective addresses and MSR_{IS}=1 are used in determining an address match on IAC1 debug events.

$DBCR1_{IAC2ER}$ specifies whether effective addresses, real addresses, effective addresses and MSR_{IS}=0, or effective addresses and MSR_{IS}=1 are used in determining an address match on IAC2 debug events.

$DBCR1_{IAC3ER}$ specifies whether effective addresses, real addresses, effective addresses and MSR_{IS}=0, or effective addresses and MSR_{IS}=1 are used in determining an address match on IAC3 debug events.

$DBCR1_{IAC4ER}$ specifies whether effective addresses, real addresses, effective addresses and MSR_{IS}=0, or effective addresses and MSR_{IS}=1 are used in determining an address match on IAC4 debug events.

Instruction Address Compare Mode

$DBCR1_{IAC12M}$ specifies whether all or some of the bits of the address of the instruction fetch must match the contents of the Instruction Address Compare 1 Register or Instruction Address Compare 2 Register, whether the address must be inside a specific range specified by the Instruction Address Compare 1 Register and Instruction Address Compare 2 Register or outside a specific range specified by the Instruction Address Compare 1 Register and Instruction Address Compare 2 Register for an IAC1 or IAC2 debug event to occur.

$DBCR1_{IAC34M}$ specifies whether all or some of the bits of the address of the instruction fetch must match the contents of the IAC3 Register or IAC4 Register, whether the address must be inside a specific range specified by the IAC3 Register and IAC4 Register or outside a specific range specified by the IAC3 Register and IAC4 Register for an IAC3 or IAC4 debug event to occur.

There are four instruction address compare modes.

– *Exact address compare mode*
  If the 64-bit address of the instruction fetch is equal to the value in the enabled IAC register, an instruction address match occurs.

– *Address bit match mode*
  For IAC1 and IAC2 debug events, if the address of the instruction fetch access, ANDed with the contents of the Instruction Address Compare 2 Register, are equal to the contents of the Instruction Address Compare 1 Register, also ANDed with the contents of the Instruction Address Compare 2 Register, an instruction address match occurs.

  For IAC3 and IAC4 debug events, if the address of the instruction fetch, ANDed with the contents of the Instruction Address Compare 4 Register, are equal to the contents of the Instruction Address Compare 3 Register, also ANDed with the contents of the Instruction Address Compare 4 Register, an instruction address match occurs.

– *Inclusive address range compare mode*
  For IAC1 and IAC2 debug events, if the 64-bit address of the instruction fetch is greater than or equal to the contents of the Instruction Address Compare 1 Register and less than the contents of the Instruction Address Compare 2 Register, an instruction address match occurs.

  For IAC3 and IAC4 debug events, if the 64-bit address of the instruction fetch is greater than or equal to the contents of the Instruction Address Compare 3 Register and less than the contents of the Instruction Address Compare 4 Register, an instruction address match occurs.

– *Exclusive address range compare mode*
  For IAC1 and IAC2 debug events, if the 64-bit address of the instruction fetch is less than the contents of the Instruction Address Compare 1 Register or greater than or equal to the contents of the Instruction Address Compare 2 Register, an instruction address match occurs.

  For IAC3 and IAC4 debug events, if the 64-bit address of the instruction fetch is less than the contents of the Instruction Address Compare 3 Register or greater than or equal to the contents of the Instruction Address Compare 4 Register, an instruction address match occurs.

See Table 9-1 on page 210 and Table 9-2 on page 212 for a detailed description of DBCR0 and DBCR1 and the modes of for detecting IAC1, IAC2, IAC3 and IAC4

debug events. Instruction Address Compare debug events can occur regardless of the setting of MSR_{DE} or $DBCR0_{IDM}$.

When an Instruction Address Compare debug event occurs, the corresponding $DBSR_{IAC1}$, $DBSR_{IAC2}$, $DBSR_{IAC3}$, or $DBSR_{IAC4}$ bit or bits are set to record the debug exception. If MSR_{DE}=0, $DBSR_{IDE}$ is also set to 1 to record the imprecise debug event.

If MSR_{DE}=1 (i.e. Debug interrupts are enabled) at the time of the Instruction Address Compare debug exception, a Debug interrupt will occur immediately (provided there exists no higher priority exception which is enabled to cause an interrupt). The execution of the instruction causing the exception will be suppressed, and Critical Save/Restore Register 0 will be set to the address of the excepting instruction.

If MSR_{DE}=0 (i.e. Debug interrupts are disabled) at the time of the Instruction Address Compare debug exception, a Debug interrupt will not occur, and the instruction will complete execution (provided the instruction is not causing some other exception which will generate an enabled interrupt).

Later, if the debug exception has not been reset by clearing $DBSR_{IAC1}$, $DBSR_{IAC2}$, $DBSR_{IAC3}$, and $DBSR_{IAC4}$, and MSR_{DE} is set to 1, a delayed Debug interrupt will occur. In this case, Critical Save/Restore Register 0 will contain the address of the instruction after the one which enabled the Debug interrupt by setting MSR_{DE} to 1. Software in the Debug interrupt handler can observe $DBSR_{IDE}$ to determine how to interpret the value in Critical Save/Restore Register 0.

9.3.2 Data Address Compare Debug Event

One or more Data Address Compare debug events (DAC1R, DAC1W, DAC2R, DAC2W) occur if they are enabled, execution is attempted of a data storage access instruction, and the type, address, and possibly even the data value of the data storage access meet the criteria specified in the Debug Control Register 0, Debug Control Register 2, and the DAC1, DAC2, DVC1, and DVC2 registers.

Data Address Compare Read/Write Enable

$DBCR0_{DAC1}$ specifies whether DAC1R debug events can occur on read-type data storage accesses and whether DAC1W debug events can occur on write-type data storage accesses.

$DBCR0_{DAC2}$ specifies whether DAC2R debug events can occur on read-type data storage accesses and whether DAC2W debug events can occur on write-type data storage accesses.

All *Load* instructions are considered reads with respect to debug events, while all *Store* instructions are considered writes with respect to debug events. In addition, the *Cache Management* instructions, and certain special cases, are handled as follows.

– ***dcbt***[***e***], ***dcbtst***[***e***], ***icbt***[***e***], and ***icbi***[***e***] are all considered reads with respect to debug events. Note that ***dcbt***[***e***], ***dcbtst***[***e***], and ***icbt***[***e***] are treated as no-operations when they report Data Storage or Data TLB Miss exceptions, instead of being allowed to cause interrupts. However, these instructions are allowed to cause Debug interrupts, even when they would otherwise have been no-op'ed due to a Data Storage or Data TLB Miss exception.

– ***dcbz***[***e***], ***dcbi***[***e***], ***dcbf***[***e***], ***dcba***[***e***], and ***dcbst***[***e***] are all considered writes

with respect to debug events. Note that ***dcbf***[***e***] and ***dcbst***[***e***] are considered reads with respect to Data Storage exceptions, since they do not actually change the data at a given address. However, since the execution of these instructions may result in write activity on the processor's data bus, they are treated as writes with respect to debug events.

Engineering Note
dcba[***e***], ***dcbt***[***e***], ***dcbtst***[***e***], and ***icbt***[***e***] may cause a Data Address Compare debug event even when they are otherwise being no-op'ed due to causing a Data Storage or Data TLB Miss exception. However, signalling a Debug exception is not required in these cases.

Indexed-string instructions (***lswx***, ***stswx***) for which the XER field specifies zero bytes as the length of the string are treated as no-ops, and are not allowed to cause Data Address Compare debug events.

Data Address Compare User/Supervisor Mode

$DBCR2_{DAC1US}$ specifies whether DAC1R and DAC1W debug events can occur in user mode or supervisor mode, or both.

$DBCR2_{DAC2US}$ specifies whether DAC2R and DAC2W debug events can occur in user mode or supervisor mode, or both.

Effective/Real Address Mode

$DBCR2_{DAC1ER}$ specifies whether effective addresses, real addresses, effective addresses and MSR_{DS}=0, or effective addresses and MSR_{DS}=1 are used to in determining an address match on DAC1R and DAC1W debug events.

$DBCR2_{DAC2ER}$ specifies whether effective addresses, real addresses, effective addresses and MSR_{DS}=0, or effective addresses and MSR_{DS}=1 are used to in determining an address match on DAC2R and DAC2W debug events.

Data Address Compare Mode

$DBCR2_{DAC12M}$ specifies whether all or some of the bits of the address of the data storage access must match the contents of the Data Address Compare 1 Register or Data Address Compare 2 Register, whether the address must be inside a specific range specified by the Data Address Compare 1 Register and Data Address Compare 2 Register or outside a specific range specified by the Data Address Compare 1 Register and Data Address Compare 2 Register for a DAC1R, DAC1W, DAC2R or DAC2W debug event to occur.

There are four data address compare modes.

– *Exact address compare mode*
If the 64-bit address of the data storage access is equal to the value in the enabled Data Address Compare register, a data address match occurs.

– *Address bit match mode*
If the address of the data storage access, ANDed with the contents of the Data Address Compare 2 Register, are equal to the contents of the Data Address Compare 1 Register, also ANDed with the contents of the Data Address Compare 2 Register, a data address match occurs.

– *Inclusive address range compare mode*
If the 64-bit address of the data storage access is greater than or equal to the contents of the Data Address Compare 1 Register and less than the contents of the Data Address Compare 2 Register, a data address match occurs.

– *Exclusive address range compare mode*
If the 64-bit address of the data storage access is less than the contents of the Data Address Compare 1 Register or greater than or equal to the contents of the Data Address Compare 2 Register, a data address match occurs.

Data Value Compare Mode

$DBCR2_{DVC1M}$ and $DBCR2_{DVC1BE}$ specify whether and how the data value being accessed by the storage access must match the contents of the Data Value Compare 1 Register for a DAC1R or DAC1W debug event to occur.

$DBCR2_{DVC2M}$ and $DBCR2_{DVC2BE}$ specify whether and how the data value being accessed by the storage access must match the contents of the Data Value Compare 2 Register for a DAC2R or DAC2W debug event to occur.

See Table 9-1 on page 210 and Table 9-3 on page 215 for a detailed description of DBCR0 and DBCR2 and the modes for detecting Data Address Compare debug events. Data Address Compare debug events can occur regardless of the setting of MSR_{DE} or $DBCR0_{IDM}$.

When an Data Address Compare debug event occurs, the corresponding $DBSR_{DAC1R}$, $DBSR_{DAC1W}$, $DBSR_{DAC2R}$, or $DBSR_{DAC2W}$ bit or bits are set to 1 to record the debug exception. If MSR_{DE}=0, $DBSR_{IDE}$ is also set to 1 to record the imprecise debug event.

If MSR_{DE}=1 (i.e. Debug interrupts are enabled) at the time of the Data Address Compare debug exception, a Debug interrupt will occur immediately (provided there exists no higher priority exception which is enabled to cause an interrupt), the execution of the instruction causing the exception will be suppressed, and Critical Save/Restore Register 0 will be set to the address of the excepting instruction. Depending on the type of instruction and/or the alignment of the data access, the instruction causing the exception may have been partially executed (see Section 7.7 on page 173).

If MSR_{DE}=0 (i.e. Debug interrupts are disabled) at the time of the Data Address Compare debug exception, a Debug interrupt will not occur, and the instruction will complete execution (provided the instruction is not causing some other exception which will generate an enabled interrupt). Also, $DBSR_{IDE}$ is set to indicate that the debug exception occurred while Debug interrupts were disabled by MSR_{DE}=0.

Later, if the debug exception has not been reset by clearing $DBSR_{DAC1R}$, $DBSR_{DAC1W}$, $DBSR_{DAC2R}$, $DBSR_{DAC2W}$, and MSR_{DE} is set to 1, a delayed Debug interrupt will occur. In this case, Critical Save/Restore Register 0 will contain the address of the instruction after the one which enabled the Debug interrupt by setting MSR_{DE} to 1. Software in the Debug interrupt handler can observe $DBSR_{IDE}$ to determine how to interpret the value in Critical Save/Restore Register 0.

9.3.3 Trap Debug Event

A Trap debug event (TRAP) occurs if $DBCR0_{TRAP}$=1 (i.e. Trap debug events are enabled) and a Trap instruction (***tw***, ***twi***, ***td***, ***tdi***) is executed and the conditions specified by the instruction for the trap are met. The event can occur regardless of the setting of MSR_{DE} or $DBCR0_{IDM}$.

When a Trap debug event occurs, $DBSR_{TR}$ is set to 1 to record the debug exception. If MSR_{DE}=0, $DBSR_{IDE}$ is also set to 1 to record the imprecise debug event.

If MSR_{DE}=1 (i.e. Debug interrupts are enabled) at the time of the Trap debug exception, a Debug interrupt will occur immediately (provided there exists no higher priority exception which is enabled to cause an interrupt), and CSRR0 will be set to the address of the excepting instruction.

If MSR_{DE}=0 (i.e. Debug interrupts are disabled) at the time of the Trap debug exception, a Debug interrupt will not occur, and a Trap exception type Program interrupt will occur instead if the trap condition is met.

Later, if the debug exception has not been reset by clearing $DBSR_{TR}$, and MSR_{DE} is set to 1, a delayed Debug interrupt will occur. In this case, Critical Save/Restore Register 0 will contain the address of the instruction after the one which enabled the Debug interrupt by setting MSR_{DE} to 1. Software in the debug interrupt handler can observe $DBSR_{IDE}$ to determine how to interpret the value in Critical Save/Restore Register 0.

9.3.4 Branch Taken Debug Event

A Branch Taken debug event (BRT) occurs if $DBCR0_{BRT}$=1 (i.e. Branch Taken Debug events are enabled), execution is attempted of a branch instruction whose direction will be taken (that is, either an unconditional branch, or a conditional branch whose branch condition is met), and MSR_{DE}=1.

Branch Taken debug events are not recognized if MSR_{DE}=0 at the time of the execution of the branch instruction and thus $DBSR_{IDE}$ can not be set by a Branch Taken debug event. This is because branch instructions occur very frequently. Allowing these common events to be recorded as exceptions in the DBSR while debug interrupts are disabled via MSR_{DE} would result in an inordinate number of imprecise Debug interrupts.

When a Branch Taken debug event occurs, the $DBSR_{BRT}$ bit is set to 1 to record the debug exception and a Debug interrupt will occur immediately (provided there exists no higher priority exception which is enabled to cause an interrupt). The execution of the instruction causing the exception will be suppressed, and Critical Save/Restore Register 0 will be set to the address of the excepting instruction.

9.3.5 Instruction Complete Debug Event

An Instruction Complete debug event (ICMP) occurs if $DBCR0_{ICMP}$=1 (i.e. Instruction Complete debug events are enabled), execution of any instruction is completed, and MSR_{DE}=1. Note that if execution of an instruction is suppressed due to the instruction causing some other exception which is enabled to generate an interrupt, then the attempted execution of that instruction does not cause an Instruction Complete debug event. The ***sc*** instruction does not fall into the category of an instruction whose execution is suppressed, since the instruction actually completes execution and then generates a System Call interrupt. In this case, the Instruction Complete debug exception will also be set.

Instruction Complete debug events are not recognized if MSR_{DE}=0 at the time of the execution of the instruction, $DBSR_{IDE}$ can not be set by an ICMP debug event. This is because allowing the common event of Instruction Completion to be recorded as an exception in the DBSR while Debug interrupts are disabled via MSR_{DE} would mean that the Debug interrupt handler software would receive an

inordinate number of imprecise Debug interrupts every time Debug interrupts were re-enabled via MSR_{DE}.

When an Instruction Complete debug event occurs, $DBSR_{ICMP}$ is set to 1 to record the debug exception, a Debug interrupt will occur immediately (provided there exists no higher priority exception which is enabled to cause an interrupt), and Critical Save/Restore Register 0 will be set to the address of the instruction after the one causing the Instruction Complete debug exception.

9.3.6 Interrupt Taken Debug Event

An Interrupt Taken debug event (IRPT) occurs if $DBCR0_{IRPT}$=1 (i.e. Interrupt Taken debug events are enabled) and a non-critical interrupt occurs. Interrupt Taken debug events can occur regardless of the setting of MSR_{DE}.

Only non-critical class interrupts can cause an Interrupt Taken debug event because all critical interrupts automatically clear MSR_{DE}, and thus would always prevent the associated Debug interrupt from occurring precisely. Also, Debug interrupts themselves are critical class interrupts, and thus any Debug interrupt (for any other debug event) would always end up setting the additional exception of $DBSR_{IRPT}$ upon entry to the Debug interrupt handler. At this point, the Debug interrupt handler would be unable to determine whether or not the Interrupt Taken debug event was related to the original debug event.

When an Interrupt Taken debug event occurs, $DBSR_{IRPT}$ is set to 1 to record the debug exception. If MSR_{DE}=0, $DBSR_{IDE}$ is also set to 1 to record the imprecise debug event.

If MSR_{DE}=1 (i.e. Debug interrupts are enabled) at the time of the Interrupt Taken debug event, a Debug interrupt will occur immediately (provided there exists no higher priority exception which is enabled to cause an interrupt), and Critical Save/Restore Register 0 will be set to the address of the non-critical interrupt vector which caused the Interrupt Taken debug event. No instructions at the non-critical interrupt handler will have been executed.

If MSR_{DE}=0 (i.e. Debug interrupts are disabled) at the time of the Interrupt Taken debug event, a Debug interrupt will not occur, and the handler for the interrupt which caused the Interrupt Taken debug event will be allowed to execute.

Later, if the debug exception has not been reset by clearing $DBSR_{IRPT}$, and MSR_{DE} is set to 1, a delayed Debug interrupt will occur. In this case, CSRR0 will contain the address of the instruction after the one which enabled the Debug interrupt by setting MSR_{DE} to 1. Software in the Debug interrupt handler can observe the $DBSR_{IDE}$ bit to determine how to interpret the value in Critical Save/Restore Register 0.

9.3.7 Return Debug Event

A Return debug event (RET) occurs if $DBCR0_{RET}$=1 and an attempt is made to execute an ***rfi***. Return debug events can occur regardless of the setting of MSR_{DE}.

When a Return debug event occurs, $DBSR_{RET}$ is set to 1 to record the debug exception. If MSR_{DE}=0, $DBSR_{IDE}$ is also set to 1 to record the imprecise debug event.

If MSR_{DE}=1 at the time of the Return Debug event, a Debug interrupt will occur immediately, and Critical Save/Restore Register 0 will be set to the address of the ***rfi***.

If MSR_{DE}=0 at the time of the Return Debug event, a Debug interrupt will not occur.

Later, if the Debug exception has not been reset by clearing $DBSR_{RET}$, and MSR_{DE} is set to 1, a delayed imprecise Debug interrupt will occur. In this case, Critical Save/Restore Register 0 will contain the address of the instruction after the one which enabled the Debug interrupt by setting MSR_{DE} to 1. An imprecise Debug interrupt can be caused by executing an ***rfi*** when $DBCR0_{RET}$=1 and MSR_{DE}=0, and the execution of that ***rfi*** happens to cause MSR_{DE} to be set to 1. Software in the Debug interrupt handler can observe the $DBSR_{IDE}$ bit to determine how to interpret the value in Critical Save/Restore Register 0.

9.3.8 Unconditional Debug Event

An Unconditional debug event (UDE) occurs when the Unconditional Debug Event (UDE) signal is activated by the debug mechanism. The exact definition of the UDE signal and how it is activated is implementation-dependent (see the User's Manual for the implementation for more details). The Unconditional debug event is the only debug event which does not have a corresponding enable bit for the event in DBCR0 (hence the name of the event). The Unconditional debug event can occur regardless of the setting of MSR_{DE}.

When an Unconditional debug event occurs, the $DBSR_{UDE}$ bit is set to 1 to record the Debug exception. If MSR_{DE}=0, $DBSR_{IDE}$ is also set to 1 to record the imprecise debug event.

If MSR_{DE}=1 (i.e. Debug interrupts are enabled) at the time of the Unconditional Debug exception, a Debug interrupt will occur immediately (provided there exists no higher priority exception which is enabled to cause an interrupt), and Critical Save/Restore Register 0 will be set to the address of the instruction which would have executed next had the interrupt not occurred.

If MSR_{DE}=0 (i.e. Debug interrupts are disabled) at the time of the Unconditional Debug exception, a Debug interrupt will not occur.

Later, if the Unconditional Debug exception has not been reset by clearing $DBSR_{UDE}$, and MSR_{DE} is set to 1, a delayed Debug interrupt will occur. In this case, CSRR0 will contain the address of the instruction after the one which enabled the Debug interrupt by setting MSR_{DE} to 1. Software in the Debug interrupt handler can observe $DBSR_{IDE}$ to determine how to interpret the value in Critical Save/Restore Register 0.

9.4 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.

9.4.1 Debug Control Registers

Debug Control Register 0 (DBCR0), Debug Control Register 1 (DBCR1), and Debug Control Register 2 (DBCR2) are each 32-bit registers. Bits of Debug Control Register 0, Debug Control Register 1, and Debug Control Register 2 are numbered 32 (most-significant bit) to 63 (least-significant bit). Debug Control Register 0, Debug Control Register 1, and Debug Control Register 2 are used to enable debug events, reset the processor, control timer operation during debug events, and set the debug mode of the processor.

9.4.1.1 Debug Control Register 0

The contents of the Debug Control Register 0 can be read into bits 32:63 a General Purpose Register using *mfspr RT,DBCR0*, setting bits 0:31 of GPR(RT) to 0. The contents of bits 32:63 of a General Purpose Register can be written to the Debug Control Register 0 using *mtspr DBCR0,RS*. Table 9-1 provides bit definitions for Debug Control Register 0.

Table 9-1. Debug Control Register 0 Definition

Bit(s)	Description	
32	Allocated for implementation-dependent use. See the User's Manual for the implementation for details.	
33	***Internal Debug Mode*** (IDM)	
	=0	Debug interrupts are disabled.
	=1	If MSR_{DE}=1, then the occurrence of a debug event or the recording of an earlier debug event in the Debug Status Register when MSR_{DE}=0 or $DBCR0_{IDM}$=0 will cause a Debug interrupt.
	Programming Note Software must clear debug event status in the Debug Status Register in the Debug interrupt handler when a Debug interrupt is taken before re-enabling interrupts via MSR_{DE}. Otherwise, redundant Debug interrupts will be taken for the same debug event.	
34:35	***Reset*** (RST)	
	=00	No action
	=01	See the User's Manual for the implementation for details.
	=10	See the User's Manual for the implementation for details.
	=11	See the User's Manual for the implementation for details.
	Warning Writing 0b01, 0b10, or 0b11 to these bits may cause a processor reset to occur.	
36	***Instruction Completion Debug Event*** (ICMP)	
	=0	ICMP debug events are disabled
	=1	ICMP debug events are enabled
	Note Instruction Completion will not cause an ICMP debug event if MSR_{DE}=0.	

Bit(s)	Description	
37	***Branch Taken Debug Event Enable*** (BRT)	
	=0	BRT debug events are disabled
	=1	BRT debug events are enabled
	Note Taken branches will not cause a BRT debug event if MSR_{DE}=0.	
38	***Interrupt Taken Debug Event Enable*** (IRPT)	
	=0	IRPT debug events are disabled
	=1	IRPT debug events are enabled
	Note Critical interrupts will not cause an IRPT debug event if MSR_{DE}=0.	
39	***Trap Debug Event Enable*** (TRAP)	
	=0	TRAP debug events cannot occur
	=1	TRAP debug events can occur
40	***Instruction Address Compare 1 Debug Event Enable*** (IAC1)	
	=0	IAC1 debug events cannot occur
	=1	IAC1 debug events can occur
41	***Instruction Address Compare 2 Debug Event Enable*** (IAC2)	
	=0	IAC2 debug events cannot occur
	=1	IAC2 debug events can occur
42	***Instruction Address Compare 3 Debug Event Enable*** (IAC3)	
	=0	IAC3 debug events cannot occur
	=1	IAC3 debug events can occur
43	***Instruction Address Compare 4 Debug Event Enable*** (IAC4)	
	=0	IAC4 debug events cannot occur
	=1	IAC4 debug events can occur
44:45	***Data Address Compare 1 Debug Event Enable*** (DAC1)	
	=00	DAC1 debug events cannot occur
	=01	DAC1 debug events can occur only if a store-type data storage access
	=10	DAC1 debug events can occur only if a load-type data storage access
	=11	DAC1 debug events can occur on any data storage access
46:47	***Data Address Compare 2 Debug Event Enable*** (DAC2)	
	=00	DAC2 debug events cannot occur
	=01	DAC2 debug events can occur only if a store-type data storage access
	=10	DAC2 debug events can occur only if a load-type data storage access
	=11	DAC2 debug events can occur on any data storage access
48	***Return Debug Event Enable*** (RET)	
	=0	RET debug events cannot occur
	=1	RET debug events can occur
	Note *Return From Critical Interrupt* will not cause an RET debug event if MSR_{DE}=0.	
49:62	Reserved	
63	***Freeze Timers on Debug Event*** (FT)	
	=0	Enable clocking of timers
	=1	Disable clocking of timers if any DBSR bit is set (except MRR)

9.4.1.2 Debug Control Register 1

The contents of the Debug Control Register 1 can be read into bits 32:63 a General Purpose Register using *mfspr RT,DBCR1*, setting bits 0:31 of GPR(RT) to 0. The contents of bits 32:63 of a General Purpose Register can be written to the Debug Control Register 1 using *mtspr DBCR1,RS*. Table 9-2 provides bit definitions for the Debug Control Register 1.

Table 9-2. Debug Control Register 1 Definition

Bit(s)	Description	
32:33	***Instruction Address Compare 1 User/Supervisor Mode*** (IAC1US)	
	=00	IAC1 debug events can occur
	=01	Reserved
	=10	IAC1 debug events can occur only if MSR_{PR}=0
	=11	IAC1 debug events can occur only if MSR_{PR}=1
34:35	***Instruction Address Compare 1 Effective/Real Mode*** (IAC1ER)	
	=00	IAC1 debug events are based on effective addresses
	=01	IAC1 debug events are based on real addresses
	=10	IAC1 debug events are based on effective addresses and can occur only if MSR_{IS}=0
	=11	IAC1 debug events are based on effective addresses and can occur only if MSR_{IS}=1
36:37	***Instruction Address Compare 2 User/Supervisor Mode*** (IAC2US)	
	=00	IAC2 debug events can occur
	=01	Reserved
	=10	IAC2 debug events can occur only if MSR_{PR}=0
	=11	IAC2 debug events can occur only if MSR_{PR}=1
38:39	***Instruction Address Compare 2 Effective/Real Mode*** (IAC2ER)	
	=00	IAC2 debug events are based on effective addresses
	=01	IAC2 debug events are based on real addresses
	=10	IAC2 debug events are based on effective addresses and can occur only if MSR_{IS}=0
	=11	IAC2 debug events are based on effective addresses and can occur only if MSR_{IS}=1
40:41	***Instruction Address Compare 1/2 Mode*** (IAC12M)	
	=00	*Exact address compare* • IAC1 debug events can occur only if the address of the instruction fetch is equal to the value specified in IAC1. • IAC2 debug events can occur only if the address of the instruction fetch is equal to the value specified in IAC2.
	=01	*Address bit match* • IAC1 and IAC2 debug events can occur only if the address of the instruction fetch, ANDed with the contents of IAC2 are equal to the contents of IAC1, also ANDed with the contents of IAC2. • If IAC1US≠IAC2US or IAC1ER≠IAC2ER, results are boundedly undefined.
	=10	*Inclusive address range compare* • IAC1 and IAC2 debug events can occur only if the address of the instruction fetch is greater than or equal to the value specified in IAC1 and less than the value specified in IAC2. • If IAC1US≠IAC2US or IAC1ER≠IAC2ER, results are boundedly undefined.

Bit(s)	Description	
40:41	=11	*Exclusive address range compare* • IAC1 and IAC2 debug events can occur only if the address of the instruction fetch is less than the value specified in IAC1 or is greater than or equal to the value specified in IAC2. • If IAC1US≠IAC2US or IAC1ER≠IAC2ER, results are boundedly undefined.
42:47	Reserved	
48:49	***Instruction Address Compare 3 User/Supervisor Mode*** (IAC3US)	
	=00	IAC3 debug events can occur
	=01	Reserved
	=10	IAC3 debug events can occur only if MSR_{PR}=0
	=11	IAC3 debug events can occur only if MSR_{PR}=1
50:51	Instruction Address Compare 3 Effective/Real Mode (IAC3ER)	
	=00	IAC3 debug events are based on effective addresses
	=01	IAC3 debug events are based on real addresses
	=10	IAC3 debug events are based on effective addresses and can occur only if MSR_{IS}=0
	=11	IAC3 debug events are based on effective addresses and can occur only if MSR_{IS}=1
52:53	***Instruction Address Compare 4 User/Supervisor Mode*** (IAC4US)	
	=00	IAC4 debug events can occur
	=01	Reserved
	=10	IAC4 debug events can occur only if MSR_{PR}=0
	=11	IAC4 debug events can occur only if MSR_{PR}=1
54:55	***Instruction Address Compare 4 Effective/Real Mode*** (IAC4ER)	
	=00	IAC4 debug events are based on effective addresses
	=01	IAC4 debug events are based on real addresses
	=10	IAC4 debug events are based on effective addresses and can occur only if MSR_{IS}=0
	=11	IAC4 debug events are based on effective addresses and can occur only if MSR_{IS}=1

Bit(s)	Description	
56:57	***Instruction Address Compare 3/4 Mode*** (IAC34M)	
	=00	*Exact address compare* • IAC3 debug events can occur only if the address of the instruction fetch is equal to the value specified in IAC3. • IAC4 debug events can occur only if the address of the instruction fetch is equal to the value specified in IAC4.
	=01	*Address bit match* • IAC3 and IAC4 debug events can occur only if the address of the data storage access, ANDed with the contents of IAC4 are equal to the contents of IAC3, also ANDed with the contents of IAC4. • If IAC3US≠IAC4US or IAC3ER≠IAC4ER, results are boundedly undefined.
	=10	*Inclusive address range compare* • IAC3 and IAC4 debug events can occur only if the address of the instruction fetch is greater than or equal to the value specified in IAC3 and less than the value specified in IAC4. • If IAC3US≠IAC4US or IAC3ER≠IAC4ER, results are boundedly undefined.
	=11	*Exclusive address range compare* • IAC3 and IAC4 debug events can occur only if the address of the instruction fetch is less than the value specified in IAC3 or is greater than or equal to the value specified in IAC4. • If IAC3US≠IAC4US or IAC3ER≠IAC4ER, results are boundedly undefined.
58:63	Reserved	

9.4.1.3 Debug Control Register 2

The contents of the Debug Control Register 2 can be read into bits 32:63 a General Purpose Register using *mfspr RT,DBCR2*, setting bits 0:31 of GPR(RT) to 0. The contents of bits 32:63 of a General Purpose Register can be written to the Debug Control Register 2 using *mtspr DBCR2,RS*. Table 9-3 provides bit definitions for the Debug Control Register 2.

Table 9-3. Debug Control Register 2 Definition

Bit(s)	Description	
32:33	***Data Address Compare 1 User/Supervisor Mode*** (DAC1US)	
	=00	DAC1 debug events can occur
	=01	Reserved
	=10	DAC1 debug events can occur only if MSR_{PR}=0
	=11	DAC1 debug events can occur only if MSR_{PR}=1
34:35	***Data Address Compare 1 Effective/Real Mode*** (DAC1ER)	
	=00	DAC1 debug events are based on effective addresses
	=01	DAC1 debug events are based on real addresses
	=10	DAC1 debug events are based on effective addresses and can occur only if MSR_{DS}=0
	=11	DAC1 debug events are based on effective addresses and can occur only if MSR_{DS}=1
36:37	***Data Address Compare 2 User/Supervisor Mode*** (DAC2US)	
	=00	DAC2 debug events can occur
	=01	Reserved
	=10	DAC2 debug events can occur only if MSR_{PR}=0
	=11	DAC2 debug events can occur only if MSR_{PR}=1
38:39	***Data Address Compare 2 Effective/Real Mode*** (DAC2ER)	
	=00	DAC2 debug events are based on effective addresses
	=01	DAC2 debug events are based on real addresses
	=10	DAC2 debug events are based on effective addresses and can occur only if MSR_{DS}=0
	=11	DAC2 debug events are based on effective addresses and can occur only if MSR_{DS}=1
40:41	***Data Address Compare 1/2 Mode*** (DAC12M)	
	=00	Exact address compare • DAC1 debug events can occur only if the address of the data storage access is equal to the value specified in DAC1. • DAC2 debug events can occur only if the address of the data storage access is equal to the value specified in DAC2.
	=01	Address bit match • DAC1 and DAC2 debug events can occur only if the address of the data storage access, ANDed with the contents of DAC2 are equal to the contents of DAC1, also ANDed with the contents of DAC2. • If DAC1US≠DAC2US or DAC1ER≠DAC2ER, results are boundedly undefined.
	=10	Inclusive address range compare • DAC1 and DAC2 debug events can occur only if the address of the data storage access is greater than or equal to the value specified in DAC1 and less than the value specified in DAC2. • If DAC1US≠DAC2US or DAC1ER≠DAC2ER, results are boundedly undefined.

Bit(s)	Description	
40:41	=11	Exclusive address range compare • DAC1 and DAC2 debug events can occur only if the address of the data storage access is less than the value specified in DAC1 or is greater than or equal to the value specified in DAC2. • If DAC1US≠DAC2US or DAC1ER≠DAC2ER, results are boundedly undefined.
42:43	Reserved	
44:45	***Data Value Compare 1 Mode*** (DVC1M)	
	=00	DAC1 debug events can occur
	=01	DAC1 debug events can occur only when all bytes specified in $DBCR2_{DVC1BE}$ in the data value of the data storage access match their corresponding bytes in DVC1
	=10	DAC1 debug events can occur only when at least one of the bytes specified in $DBCR2_{DVC1BE}$ in the data value of the data storage access matches its corresponding byte in DVC1
	=11	DAC1 debug events can occur only when all bytes specified in $DBCR2_{DVC1BE}$ within at least one of the halfwords of the data value of the data storage access matches their corresponding bytes in DVC1
46:47	***Data Value Compare 2 Mode*** (DVC2M)	
	=00	DAC2 debug events can occur
	=01	DAC2 debug events can occur only when all bytes specified in $DBCR2_{DVC2BE}$ in the data value of the data storage access match their corresponding bytes in DVC2
	=10	DAC2 debug events can occur only when at least one of the bytes specified in $DBCR2_{DVC2BE}$ in the data value of the data storage access matches its corresponding byte in DVC2
	=11	DAC2 debug events can occur only when all bytes specified in $DBCR2_{DVC2BE}$ within at least one of the halfwords of the data value of the data storage access matches their corresponding bytes in DVC2
48:55	***Data Value Compare 1 Byte Enables*** (DVC1BE) Specifies which bytes in the aligned data value being read or written by the storage access are compared to the corresponding bytes in DVC1.	
56:63	***Data Value Compare 2 Byte Enables*** (DVC2BE) Specifies which bytes in the aligned data value being read or written by the storage access are compared to the corresponding bytes in DVC2	

9.4.2 Debug Status Register

The Debug Status Register (DBSR) is a 32-bit register and contains status on debug events and the most recent processor reset. Table 9-4 provides bit definitions for the Debug Status Register.

The Debug Status Register is set via hardware, and read and cleared via software. The contents of the Debug Status Register can be read into bits 32:63 of a General Purpose Register using *mfspr RT,DBSR*, setting bits 0:31 of GPR(RT) to zero. Bits in the Debug Status Register can be cleared using *mtspr DBSR,RS*. Clearing is done by writing bits 32:63 of a General Purpose Register to the Debug Status Register with a 1 in any bit position that is to be cleared and 0 in all other bit positions. The write-data to the Debug Status Register is not direct data, and DBCR0IAC1=1but a mask. A 1 causes the bit to be cleared, and a 0 has no effect.

Table 9-4. Debug Status Register Definition

Bit(s)	Description
32	***Imprecise Debug Event*** (IDE) Set to 1 if MSR_{DE}=0 and a debug event causes its respective Debug Status Register bit to be set to 1.
33	***Unconditional Debug Event*** (UDE) Set to 1 if an Unconditional debug event occurred. See Section 9.3.8 on page 209.
34:35	***Most Recent Reset*** (MRR) Set to one of three values when a reset occurs. These two bits are undefined at power-up. =00 — No reset occurred since these bits last cleared by software =01 — Implementation-dependent reset information =10 — Implementation-dependent reset information =11 — Implementation-dependent reset information **Note** See the User's Manual for the implementation for further details.
36	***Instruction Complete Debug Event*** (ICMP) Set to 1 if an Instruction Completion debug event occurred and $DBCR0_{ICMP}$=1. See Section 9.3.5 on page 207.
37	***Branch Taken Debug Event*** (BRT) Set to 1 if a Branch Taken debug event occurred and $DBCR0_{BRT}$=1. See Section 9.3.4 on page 207.
38	***Interrupt Taken Debug Event*** (IRPT) Set to 1 if an Interrupt Taken debug event occurred and $DBCR0_{IRPT}$=1. See Section 9.3.6 on page 208.
39	***Trap Instruction Debug Event*** (TRAP) Set to 1 if a Trap Instruction debug event occurred and $DBCR0_{TRAP}$=1. See Section 9.3.3 on page 206.
40	***Instruction Address Compare 1 Debug Event*** (IAC1) Set to 1 if an IAC1 debug event occurred and $DBCR0_{IAC1}$=1. See Section 9.3.1 on page 202.
41	***Instruction Address Compare 2 Debug Event*** (IAC2) Set to 1 if an IAC2 debug event occurred and $DBCR0_{IAC2}$=1. See Section 9.3.1 on page 202.
42	***Instruction Address Compare 3 Debug Event*** (IAC3) Set to 1 if an IAC3 debug event occurred and $DBCR0_{IAC3}$=1. See Section 9.3.1 on page 202.
43	***Instruction Address Compare 4 Debug Event*** (IAC4) Set to 1 if an IAC4 debug event occurred and $DBCR0_{IAC4}$=1. See Section 9.3.1 on page 202.
44	***Data Address Compare 1 Read Debug Event*** (DAC1R) Set to 1 if a read-type DAC1 debug event occurred and $DBCR0_{DAC1}$=0b10 or $DBCR0_{DAC1}$=0b11. See Section 9.3.2 on page 204.

Bit(s)	Description
45	***Data Address Compare 1 Write Debug Event*** (DAC1W) Set to 1 if a write-type DAC1 debug event occurred and $DBCR0_{DAC1}$=0b01 or $DBCR0_{DAC1}$=0b11. See Section 9.3.2 on page 204.
46	***Data Address Compare 2 Read Debug Event*** (DAC2R) Set to 1 if a read-type DAC2 debug event occurred and $DBCR0_{DAC2}$=0b10 or $DBCR0_{DAC2}$=0b11. See Section 9.3.2 on page 204.
47	***Data Address Compare 2 Write Debug Event*** (DAC2W) Set to 1 if a write-type DAC2 debug event occurred and $DBCR0_{DAC2}$=0b01 or $DBCR0_{DAC2}$=0b11. See Section 9.3.2 on page 204.
48	***Return Debug Event*** (RET) Set to 1 if a Return debug event occurred and $DBCR0_{RET}$=1. See Section 9.3.7 on page 208.
49:63	Reserved

9.4.3 Instruction Address Compare Registers

The Instruction Address Compare 1 Register (IAC1), Instruction Address Compare 2 Register (IAC2), Instruction Address Compare 3 Register (IAC3), and Instruction Address Compare 4 Register (IAC4) are each 64-bits, with bits 62:63 being reserved.

A debug event may be enabled to occur upon an attempt to execute an instruction from an address specified in either Instruction Address Compare 1 Register, Instruction Address Compare 2 Register, Instruction Address Compare 3 Register, or Instruction Address Compare 4 Register, inside or outside a range specified by Instruction Address Compare 1 Register and Instruction Address Compare 2 Register or, inside or outside a range specified by Instruction Address Compare 3 Register and Instruction Address Compare 4 Register, or to blocks of addresses specified by the combination of the Instruction Address Compare 1 Register and Instruction Address Compare 2 Register, or to blocks of addresses specified by the combination of the Instruction Address Compare 3 Register and Instruction Address Compare 4 Register. Since all instruction addresses are required to be word-aligned, the two low-order bits of the Instruction Address Compare Registers are reserved and do not participate in the comparison to the instruction address. See Section 9.3.1 on page 202.

The contents of the Instruction Address Compare *i* Register (where *i*={1,2,3, or 4}) can be read into a General Purpose Register using *mfspr RT,IACi*. The contents of a General Purpose Register can be written to the Instruction Address Compare *i* Register using *mtspr IACi,RS*.

9.4.4 Data Address Compare Registers

The Data Address Compare 1 Register (DAC1) and Data Address Compare 2 Register (DAC2) are each 64-bits.

A debug event may be enabled to occur upon loads, stores, or cache operations to an address specified in either the Data Address Compare 1 Register or Data Address Compare 2 Register, inside or outside a range specified by the Data Address Compare 1 Register and Data Address Compare 2 Register, or to blocks of addresses specified by the combination of the Data Address Compare 1 Register and Data Address Compare 1 Register. See Section 9.3.2 on page 204.

The contents of the Data Address Compare *i* Register (where *i*={1 or 2}) can be read into a General Purpose Register using *mfspr RT,DACi*. The contents of a General Purpose Register can be written to the Data Address Compare *i* Register using *mtspr DACi,RS*.

The contents of the Data Address Compare 1 Register or Data Address Compare 2 Register are compared to the address generated by a data storage access instruction.

9.4.5 Data Value Compare Registers

The Data Value Compare 1 Register (DVC1) and Data Value Compare 2 Register (DVC2) are each 64-bits.

A DAC1R, DAC1W, DAC2R, or DAC2W debug event may be enabled to occur upon loads or stores of a specific data value specified in either or both of the Data Value Compare 1 Register and Data Value Compare 2 Register. $DBCR2_{DVC1M}$ and $DBCR2_{DVC1BE}$ control how the contents of the Data Value Compare 1 Register is compared with the value and $DBCR2_{DVC2M}$ and $DBCR2_{DVC2BE}$ control how the contents of the Data Value Compare 2 Register is compared with the value. See Section 9.3.2 on page 204 and Table 9-3 on page 215 for a detailed description of the modes provided.

The contents of the Data Value Compare *i* Register (where *i*={1 or 2}) can be read into a General Purpose Register using *mfspr RT,DVCi*. The contents of a General Purpose Register can be written to the Data Value Compare *i* Register using *mtspr DVCi,RS*.

Chapter 10 Reset and Initialization

This chapter describes the requirements for Book E processor reset. This includes both the means of causing reset, and the specific initialization that is required to be performed automatically by the processor hardware. This chapter also provides an overview of the operations that should be performed by initialization software, in order to fully initialize the processor.

In general, the specific actions taken by a processor upon reset are implementation dependent, and are described in the User's Manual for the implementation. Also, it is the responsibility of system initialization software to initialize the majority of processor and system resources after reset. Implementations are required to provide a minimum processor initialization such that this system software may be fetched and executed, thereby accomplishing the rest of system initialization.

10.1 Reset Mechanisms

This specification defines two processor mechanisms for internally invoking a reset operation using either the Watchdog Timer (see Section 8.7 on page 196) or the Debug facilities using $DBCR0_{RST}$ (see Figure 9-1 on page 210). In addition, implementations will typically provide additional means for invoking a reset operation, via an external mechanism such as a signal pin which when activated will cause the processor to reset.

10.2 Processor State After Reset

The initial processor state is controlled by the register contents after reset. In general, the contents of most registers are undefined after reset.

The processor hardware is only guaranteed to initialize those registers (or specific bits in registers) which must be initialized in order for software to be able to reliably perform the rest of system initialization.

The Machine State Register and Processor Version Register and a TLB entry are updated as follows:

Machine State Register

Bit	Setting	Comments
WE	0	Wait State disabled
CE	0	Critical Input interrupts disabled
DE	0	Debug interrupts disabled
EE	0	External Input interrupts disabled
PR	0	User mode
FP	0	FP unavailable
ME	0	Machine Check interrupts disabled
FE0	0	FP exception type Program interrupts disabled
FE1	0	FP exception type Program interrupts disabled
IS	0	Instruction Address Space 0
DS	0	Data Address Space 0

Processor Version Register

Implementation-Dependent. (This register is read-only, and contains a value which identifies the specific implementation)

TLB entry

A TLB entry (which entry is implementation-dependent) is initialized in an implementation-dependent manner that maps the last 4KB page in the implemented storage address space, with the following field settings:

Field	Setting	Comments
EPN	2^N-4	N is the size of the implemented TLB EPN field
RPN	2^M-4	M is the size of the implemented TLB RPN field
TS	0	translation address space 0
SIZE	0b0001	4KB page size
W	?	implementation-dependent value
I	?	implementation-dependent value
M	?	implementation-dependent value
G	?	implementation-dependent value
E	?	implementation-dependent value
U0	?	implementation-dependent value
U1	?	implementation-dependent value
U2	?	implementation-dependent value
U3	?	implementation-dependent value
TID	?	implementation-dependent value, but page must be accessible
		Engineering Note A non-0 value may require the PID Register also be initialized to the same value to provide accessibility.
UX	?	implementation-dependent value
UR	?	implementation-dependent value
UW	?	implementation-dependent value
SX	1	page is execute accessible in supervisor mode
SR	1	page is read accessible in supervisor mode
SW	1	page is write accessible in supervisor mode

Instruction execution begins at address 2^{M+12} - 4, where M is the size of the implemented TLB RPN field and is implementation-dependent. Note that this address is different from the PowerPC Architecture System Reset interrupt vector.

10.3 Software Initialization Requirements

When reset occurs, the processor is initialized to a minimum configuration to start executing initialization code. Initialization code is necessary to complete the processor and system configuration. The initialization code described in this section is the minimum recommended for configuring the processor to run application code.

Initialization code should configure the following processor resources:

- Invalidate the instruction cache and data cache (implementation-dependent).
- Initialize system memory as required by the operating system or application code.
- Initialize processor registers as needed by the system.
- Initialize off-chip system facilities.
- Dispatch the operating system or application code.

Chapter 11 Synchronization Requirements

This section discusses synchronization requirements for special registers and translation lookaside buffers. 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 MSR_{IS}=1 has the side effect of changing address space. These side 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 a "context-altering instruction." This chapter covers all the context-altering instructions. The software synchronization required for each is shown in Table 11-1, "Data Access," on p. 11-4 and Table 11-2, "Instruction Fetch And/Or Execution," on p. 11-5.

The notation "CSI" in the tables means any context synchronizing instruction (i.e., ***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.

Programming 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 Table 11-2, because all instructions before the context-altering instruction are fetched and decoded before the context-altering instruction is executed (the processor must determine whether any of the preceding instructions are context synchronizing)

Table 11-1 below identifies the software synchronization requirements for data access for all context-altering instructions.

Table 11-1. Data Access

Context Altering Instruction or Event	Required Before	Required After	Notes
interrupt	none	none	
rfi	none	none	
rfci	none	none	
sc	none	none	
mtmsr (PR)	none	CSI	
mtmsr (ME)	none	CSI	1
mtmsr (DS)	none	CSI	
mtspr (DAC1, DAC2, DVC1, DVC2)	—	—	4
mtspr (DBCR0, DBCR2)	—	—	4
mtspr (DBSR)	—	—	4
mtspr (PID)	CSI	CSI	
tlbiva[***e***]	CSI	CSI or ***msync***	5,6
tlbwe	CSI	CSI or ***msync***	5,6

Table 11-2 below identifies the software synchronization requirements for instruction fetch and/or execution for all context-altering instructions.

Table 11-2. Instruction Fetch And/Or Execution

Context Altering Instruction or Event	Required Before	Required After	Notes
interrupt	none	none	
rfi	none	none	
rfci	none	none	
sc	none	none	
mtmsr (WE)	—	—	2
mtmsr (CE)	none	none	3
mtmsr (EE)	none	none	3
wrtee	none	none	3
wrteei	none	none	3
mtmsr (PR)	none	CSI	
mtmsr (FP)	none	CSI	
mtmsr (ME)	none	CSI	1
mtmsr (FE0)	none	CSI	
mtmsr (FE1)	none	CSI	
mtmsr (DE)	none	CSI	
mtspr (IVPR)	none	none	
mtspr (IVORi)	none	none	
mtmsr (IS)	none	CSI	7
mtspr (PID)	none	CSI	7
mtspr (DEC)	none	none	8
mtspr (TCR)	none	none	8
mtspr (TSR)	none	none	8
mtspr (IAC1, IAC2, IAC3, IAC4)	—	—	4
mtspr (DBCR0, DBCR1)	—	—	4
mtspr (DBSR)	—	—	4
tlbiva[***e***]	none	CSI or ***msync***	5,6
tlbwe	none	CSI or ***msync***	5,6

Notes for Tables 11-1 and 11-2

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 the Wait State Enable are implementation-dependent, and are specified in the User's Manual for the implementation.

3. The effect of changing MSR_{EE} or MSR_{CE} is immediate.

   If an ***mtmsr***, ***wrtee***, or ***wrteei*** instruction sets MSR_{EE} to '0', an External Input, DEC or FIT interrupt does not occur after the instruction is executed.

   If an ***mtmsr***, ***wrtee***, or ***wrteei*** instruction changes MSR_{EE} from '0' to '1' when an External Input, Decrementer, Fixed-Interval Timer, or higher priority enabled exception exists, the corresponding interrupt occurs immediately after the ***mtmsr***, ***wrtee***, or ***wrteei*** is executed, and before the next instruction is executed in the program that set MSR_{EE} to '1'.

   If an ***mtmsr*** instruction sets MSR_{CE} to '0', a Critical Input or Watchdog Timer interrupt does not occur after the instruction is executed.

If an ***mtmsr*** instruction changes MSR_{CE} from '0' to '1' when a Critical Input, Watchdog Timer or higher priority enabled exception exists, the corresponding interrupt occurs immediately after the ***mtmsr*** is executed, and before the next instruction is executed in the program that set MSR_{CE} to '1'.

4. Synchronization requirements for changing any of the Debug Facility registers are implementation-dependent, and are specified in the User's Manual for the implementation.

5. For data accesses, the context synchronizing instruction before the ***tlbwe*** or ***tlbiva***[***e***] instruction ensures that all storage accesses due to preceding instructions have completed to a point at which they have reported all exceptions they will cause.

   The context synchronizing instruction after the ***tlbwe*** or ***tlbiva***[***e***] ensures that subsequent storage accesses (data and instruction) will use the updated value in the TLB entry(s) being affected. It does not ensure that all storage accesses previously translated by the TLB entry(s) being updated have completed with respect to storage; if these completions must be ensured, the ***tlbwe*** or ***tlbiva***[***e***] must be followed by an ***msync*** instruction as well as by a context synchronizing instruction.

   Programming Note
   The following sequence illustrates why it is necessary, for data accesses, to ensure that all storage accesses due to instructions before the ***tlbwe*** or ***tlbiva***[***e***] 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 ***tlbiva***[***e***] 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.

6. Multiprocessor systems have other requirements to synchronize "TLB shoot down" (i.e., 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).

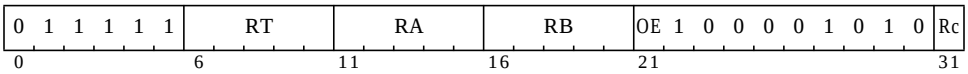
7. The alteration must not cause an implicit branch in real address space. Thus the real address of the context-altering instruction and of each subsequent instruction, up to and including the next context synchronizing instruction, must be independent of whether the alteration has taken effect.

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.

Chapter 12 Instruction Set

Add

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




```
carry0:63 ← Carry(GPR(RA) + GPR(RB))
sum0:63   ←       GPR(RA) + GPR(RB)
if OE=1 then do
   OV   ← carry32 ⊕ carry33
   SO   ← SO | (carry32 ⊕ carry33)
   OV64 ← carry0 ⊕ carry1
   SO64 ← SO64 | (carry0 ⊕ carry1)
if Rc=1 then do
   LT  ← sum32:63 < 0
   GT  ← sum32:63 > 0
   EQ  ← sum32:63 = 0
   CR0 ← LT || GT || EQ || SO
GPR(RT) ← sum
```

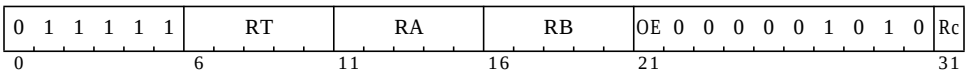
The sum of the contents of GPR(RA) and the contents of GPR(RB) is placed into GPR(RT).

Special Registers Altered:

CR0 . . . . . (if Rc=1)
SO OV SO64 OV64 . . . . . (if OE=1)

Add Carrying

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




```
carry0:63 ← Carry(GPR(RA) + GPR(RB))
sum0:63   ←       GPR(RA) + GPR(RB)
if OE=1 then do
   OV   ← carry32 ⊕ carry33
   SO   ← SO | (carry32 ⊕ carry33)
   OV64 ← carry0 ⊕ carry1
   SO64 ← SO64 | (carry0 ⊕ carry1)
if Rc=1 then do
   LT  ← sum32:63 < 0
   GT  ← sum32:63 > 0
   EQ  ← sum32:63 = 0
   CR0 ← LT || GT || EQ || SO
GPR(RT) ← sum
CA      ← carry32
CA64    ← carry0
```

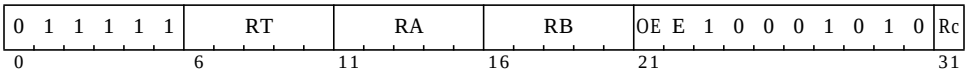
The sum of the contents of GPR(RA) and the contents of GPR(RB) is placed into GPR(RT).

Special Registers Altered:

CA CA64
CR0 . . . . . .(if Rc=1)
SO OV SO64 OV64 . . . . . (if OE=1)

Add Extended

adde RT,RA,RB . . . . . .(E=0, OE=0, Rc=0)
adde. RT,RA,RB . . . . . .(E=0, OE=0, Rc=1)
addeo RT,RA,RB . . . . . .(E=0, OE=1, Rc=0)
addeo. RT,RA,RB . . . . . .(E=0, OE=1, Rc=1)
adde64 RT,RA,RB . . . . . .(E=1, OE=0, Rc=0)
adde64o RT,RA,RB . . . . . .(E=1, OE=1, Rc=0)




```
if E=0 then Cin ← CA else Cin ← CA64
carry0:63 ← Carry(GPR(RA) + GPR(RB) + Cin)
sum0:63   ←       GPR(RA) + GPR(RB) + Cin
if OE=1 then do
   OV   ← carry32 ⊕ carry33
   SO   ← SO | (carry32 ⊕ carry33)
   OV64 ← carry0 ⊕ carry1
   SO64 ← SO64 | (carry0 ⊕ carry1)
if Rc=1 then do
   LT  ← sum32:63 < 0
   GT  ← sum32:63 > 0
   EQ  ← sum32:63 = 0
   CR0 ← LT || GT || EQ || SO
GPR(RT) ← sum
CA      ← carry32
CA64    ← carry0
```

For ***adde***[***o***][**.**], the sum of the contents of GPR(RA), the contents of GPR(RB), and CA is placed into GPR(RT).

For ***adde64***[***o***], the sum of the contents of GPR(RA), the contents of GPR(RB), and CA64 is placed into GPR(RT).

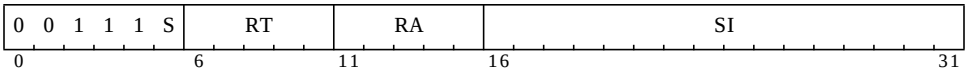
For ***adde64***[***o***], if Rc=1 the instruction form is invalid.

Special Registers Altered:

CA CA64
CR0 . . . . . .(if Rc=1)
SO OV SO64 OV64 . . . . . (if OE=1)

Add Immediate [Shifted]

addi RT,RA,SI . . . . . (S=0)
addis RT,RA,SI . . . . . (S=1)




```
if RA=0 then a ← 64 0 else a ← GPR(RA)
if s=0  then b ← EXTS(SI)
if s=1  then b ← EXTS(SI || 16 0)
GPR(RT) ← a + b
```

If ***addi*** and RA=0, the sign-extended value of the SI field is placed into GPR(RT).

If ***addi*** and RA≠0, the sum of the contents of GPR(RA) and the sign-extended value of field SI is placed into GPR(RT).

If ***addis*** and RA=0, the sign-extended value of the SI field, concatenated with 16 zeros, is placed into GPR(RT).

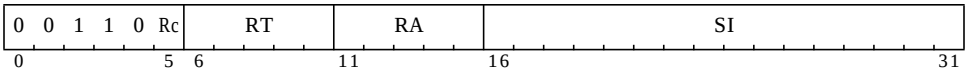
If ***addis*** and RA≠0, the sum of the contents of GPR(RA) and the sign-extended value of the SI field concatenated with 16 zeros, is placed into GPR(RT).

Special Registers Altered:

None

Add Immediate Carrying [and Record]

addic RT,RA,SI . . . . . (Rc=0)
addic. RT,RA,SI . . . . . (Rc=1)




```
carry0:63 ← Carry(GPR(RA) + EXTS(SI))
sum0:63   ←       GPR(RA) + EXTS(SI)
if Rc=1 then do
   LT   ← sum32:63 < 0
   GT   ← sum32:63 > 0
   EQ   ← sum32:63 = 0
   CR0  ← LT || GT || EQ || SO
GPR(RT) ← GPR(RA)+EXTS(SI)
CA      ← carry32
CA64    ← carry0
```

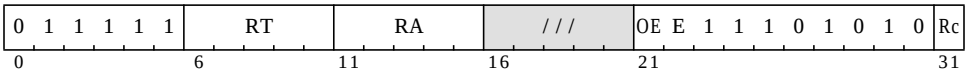
The sum of the contents of GPR(RA) and the sign-extended value of the SI field is placed into GPR(RT).

Special Registers Altered:

CA CA64
CR0 . . . . . .(if Rc=1)

Add to Minus One Extended

addme RT,RA . . . . . .(E=0, OE=0, Rc=0)
addme. RT,RA . . . . . .(E=0, OE=0, Rc=1)
addmeo RT,RA . . . . . .(E=0, OE=1, Rc=0)
addmeo. RT,RA . . . . . .(E=0, OE=1, Rc=1)
addme64 RT,RA . . . . . .(E=1, OE=0, Rc=0)
addme64o RT,RA . . . . . .(E=1, OE=1, Rc=0)




```
if E=0 then Cin ← CA else Cin ← CA64
carry0:63 ← Carry(GPR(RA) + Cin + 0xFFFF_FFFF_FFFF_FFFF)
sum0:63   ←       GPR(RA) + Cin + 0xFFFF_FFFF_FFFF_FFFF
if OE=1 then do
   OV   ← carry32 ⊕ carry33
   SO   ← SO | (carry32 ⊕ carry33)
   OV64 ← carry0 ⊕ carry1
   SO64 ← SO64 | (carry0 ⊕ carry1)
if Rc=1 then do
   LT  ← sum32:63 < 0
   GT  ← sum32:63 > 0
   EQ  ← sum32:63 = 0
   CR0 ← LT || GT || EQ || SO
GPR(RT) ← sum
CA      ← carry32
CA64    ← carry0
```

For ***addme***[***o***][**.**], the sum of the contents of GPR(RA), CA, and $^{64}1$ is placed into GPR(RT).

For ***addme64***[***o***], the sum of the contents of GPR(RA), CA64, and $^{64}1$ is placed into GPR(RT).

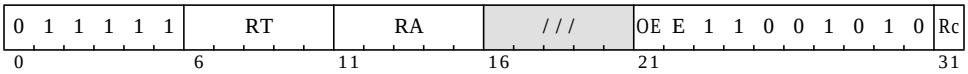
For ***addme64***[***o***], if Rc=1 the instruction form is invalid.

Special Registers Altered:

CA CA64
CR0 . . . . . .(if Rc=1)
SO OV SO64 OV64 . . . . . (if OE=1)

Add to Zero Extended

addze RT,RA . . . . . .(E=0, OE=0, Rc=0)
addze. RT,RA . . . . . .(E=0, OE=0, Rc=1)
addzeo RT,RA . . . . . .(E=0, OE=1, Rc=0)
addzeo. RT,RA . . . . . .(E=0, OE=1, Rc=1)
addze64 RT,RA . . . . . .(E=1, OE=0, Rc=0)
addze64o RT,RA . . . . . .(E=1, OE=1, Rc=0)




```
if E=0 then Cin ← CA else Cin ← CA64
carry0:63 ← Carry(GPR(RA) + Cin)
sum0:63   ←       GPR(RA) + Cin
if OE=1 then do
   OV   ← carry32 ⊕ carry33
   SO   ← SO | (carry32 ⊕ carry33)
   OV64 ← carry0 ⊕ carry1
   SO64 ← SO64 | (carry0 ⊕ carry1)
if Rc=1 then do
   LT  ← sum32:63 < 0
   GT  ← sum32:63 > 0
   EQ  ← sum32:63 = 0
   CR0 ← LT || GT || EQ || SO
GPR(RT) ← sum
CA      ← carry32
CA64    ← carry0
```

For ***addze***[***o***][**.**], the sum of the contents of GPR(RA) and CA is placed into GPR(RT).

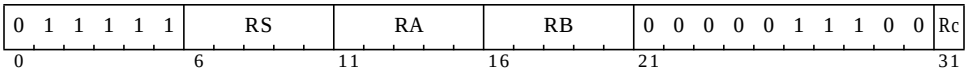
For ***addze64***[***o***], the sum of the contents of GPR(RA) and CA64 is placed into GPR(RT).

For ***addze64***[***o***], if Rc=1 the instruction form is invalid.

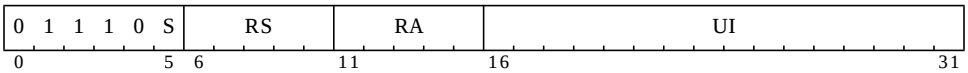
Special Registers Altered:

CA CA64
CR0 . . . . . .(if Rc=1)
SO OV SO64 OV64 . . . . . (if OE=1)

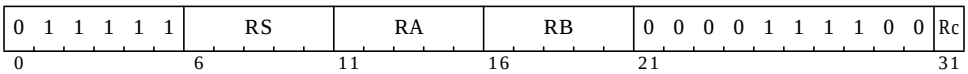
and RA,RS,RB . . . . . (Rc=0)
and. RA,RS,RB . . . . . (Rc=1)



andi. RA,RS,UI. . . . . (S=0, Rc=1)
andis. RA,RS,UI. . . . . (S=1, Rc=1)



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




```
if 'andi.'   then b ← 48 0 || UI
if 'andis.'  then b ← 32 0 || UI || 16 0
if 'and[.]'  then b ← GPR(RB)
if 'andc[.]' then b ← ¬GPR(RB)
result0:63 ← GPR(RS) & b
if Rc=1 then do
   LT  ← result32:63 < 0
   GT  ← result32:63 > 0
   EQ  ← result32:63 = 0
   CR0 ← LT || GT || EQ || SO
GPR(RA) ← result
```

For ***andi.***, the contents of GPR(RS) are ANDed with $^{48}0$ || UI.

For ***andis.***, the contents of GPR(RS) are ANDed with $^{32}0$ || UI || $^{16}0$.

For ***and***[**.**], the contents of GPR(RS) are ANDed with the contents of GPR(RB).

For ***andc***[**.**], the contents of GPR(RS) are ANDed with the one's complement of the contents of GPR(RB).

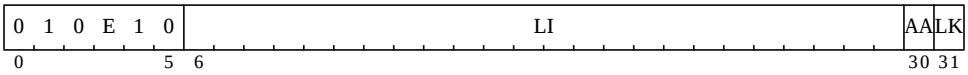
The result is placed into GPR(RA).

Special Registers Altered:

CR0 . . . . . .(if Rc=1)

Branch [Extended] [and Link] [Absolute]

b	LI	(E=0, AA=0, LK=0)
ba	LI	(E=0, AA=1, LK=0)
bl	LI	(E=0, AA=0, LK=1)
bla	LI	(E=0, AA=1, LK=1)
be	LI	(E=1, AA=0, LK=0)
bea	LI	(E=1, AA=1, LK=0)
bel	LI	(E=1, AA=0, LK=1)
bela	LI	(E=1, AA=1, LK=1)




```
if AA=1 then a ← 640 else a ← CIA
if E=0  then NIA ← 320 || (a + EXTS(LI||0b00))32:63
if E=1  then NIA ← a + EXTS(LI||0b00)
if LK=1 then LR ← CIA + 4
```

Let the branch target effective address (BTEA) be calculated as follows:

- For ***b***[***l***][***a***], let BTEA be 32 0s concatenated with bits 32:63 of the sum of the current instruction address (CIA), or 64 0s if AA=1, and the sign-extended value of the LI instruction field concatenated with 0b00.

- For ***be***[***l***][***a***], let BTEA be the sum of the current instruction address (CIA), or 64 0s if AA=1, and the sign-extended value of the LI instruction field concatenated with 0b00.

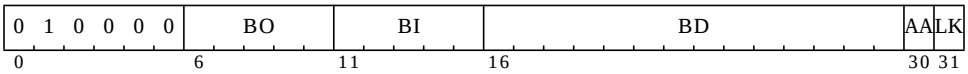
The BTEA is the address of the next instruction to be executed.

If LK=1, the sum CIA+4 is placed into the Link Register.

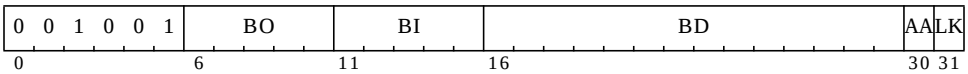
Special Registers Altered:

LR . . . . . (if LK=1)

bc BO,BI,BD . . . . . (E=0, AA=0, LK=0)
bca BO,BI,BD . . . . . (E=0, AA=1, LK=0)
bcl BO,BI,BD . . . . . (E=0, AA=0, LK=1)
bcla BO,BI,BD . . . . . (E=0, AA=1, LK=1)



bce BO,BI,BD . . . . . (E=1, AA=0, LK=0)
bcea BO,BI,BD . . . . . (E=1, AA=1, LK=0)
bcel BO,BI,BD . . . . . (E=1, AA=0, LK=1)
bcela BO,BI,BD . . . . . (E=1, AA=1, LK=1)




```
if ¬BO2 then CTR32:63 ← CTR32:63 – 1
ctr_ok  ← BO2 | ((CTR32:63 ≠ 0) ⊕ BO3)
cond_ok ← BO0 | (CRBI+32 ≡ BO1)
if ctr_ok & cond_ok then
   if AA=1 then a ← 640 else a ← CIA
   if E=0 then NIA ← 320 || (a + EXTS(BD||0b00))32:63
   if E=1 then NIA ← a + EXTS(BD||0b00)
else           NIA ← CIA + 4
if LK=1 then LR ← CIA + 4
```

Let the branch target effective address (BTEA) be calculated as follows:

- For ***bc***[***l***][***a***], let BTEA be 32 0s concatenated with bits 32:63 of the sum of the current instruction address (CIA), or 64 0s if AA=1, and the sign-extended value of the BD instruction field concatenated with 0b00.

- For ***bce***[***l***][***a***], let BTEA be the sum of the current instruction address (CIA), or 64 0s if AA=1, and the sign-extended value of the BD instruction field concatenated with 0b00.

The BO field of the instruction specifies the condition or conditions that must be met in order for the branch to be taken, as defined in Section 3.3 on page 49. The sum BI+32 specifies the bit of the Condition Register that is to be used.

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 Link Register.

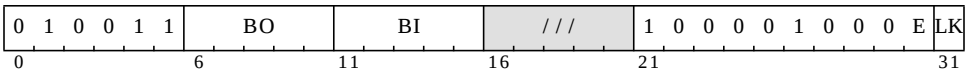
Special Registers Altered:

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

Branch Conditional to Count Register [Extended] [and Link]

bcctr BO,BI . . . . . (E=0, LK=0)
bcctrl BO,BI . . . . . (E=0, LK=1)

bcctre BO,BI . . . . . (E=1, LK=0)
bcctrel BO,BI . . . . . (E=1, LK=1)




```
cond_ok ← BO0 | (CRBI+32 ≡ BO1)
if  cond_ok & E=0 then NIA ← 320 || CTR32:61 || 0b00
if  cond_ok & E=1 then NIA ← CTR0:61 || 0b00
if ¬cond_ok       then NIA ← CIA + 4
if LK=1 then LR ← CIA + 4
```

Let the branch target effective address (BTEA) be calculated as follows:

- For ***bcctr***[***l***], let BTEA be 32 0s concatenated with the contents of bits 32:61 of the Count Register concatenated with 0b00.

- For ***bcctre***[***l***], let BTEA be the contents of bits 0:61 of the Count Register concatenated with 0b00.

The BO field of the instruction specifies the condition or conditions that must be met in order for the branch to be taken, as defined in Section 3.3 on page 49. The sum BI+32 specifies the bit of the Condition Register that is to be used.

If the branch 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 Link Register.

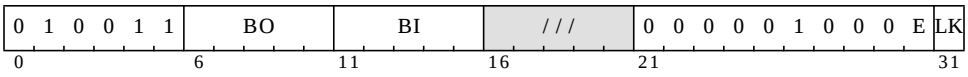
If the 'decrement and test CTR' option is specified (BO_2=0), the instruction form is invalid.

Special Registers Altered:

LR . . . . . (if LK=1)

bclr BO,BI . . . . . (E=0, LK=0)
bclrl BO,BI . . . . . (E=0, LK=1)

bclre BO,BI . . . . . (E=1, LK=0)
bclrel BO,BI . . . . . (E=1, LK=1)




```
if ¬BO2 then CTR32:63 ← CTR32:63 - 1
ctr_ok  ← BO2 | ((CTR32:63 ≠ 0) ⊕ BO3)
cond_ok ← BO0 | (CRBI+32 ≡ BO1)
if   ctr_ok & cond_ok & E=0 then NIA ← 320 || LR32:61 || 0b00
if   ctr_ok & cond_ok & E=1 then NIA ← LR0:61 || 0b00
if ¬(ctr_ok & cond_ok)      then NIA ← CIA + 4
if LK=1 then LR ← CIA + 4
```

Let the branch target effective address (BTEA) be calculated as follows:

- For ***bclr***[***l***], let BTEA be 32 0s concatenated with the contents of bits 32:61 of the Link Register concatenated with 0b00.

- For ***bclre***[***l***], let BTEA be the contents of bits 0:61 of the Link Register concatenated with 0b00.

The BO field of the instruction specifies the condition or conditions that must be met in order for the branch to be taken, as defined in Section 3.3 on page 49. The sum BI+32 specifies the bit of the Condition Register that is to be used.

If the branch condition is met, the BTEA is the address of the next instruction to be executed.

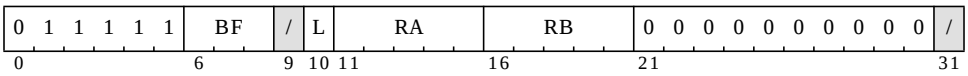
If LK=1, the sum CIA + 4 is placed into the Link Register.

Special Registers Altered:

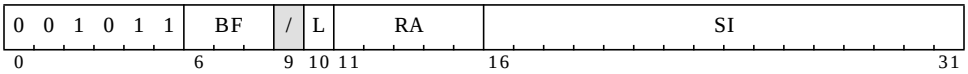
CTR . . . . . (if BO_2=0)
LR . . . . . (if LK=1)

Compare [Immediate]

cmp BF,L,RA,RB



cmpi BF,L,RA,SI




```
if L=0 then a ← EXTS(GPR(RA)32:63)
else        a ← GPR(RA)
if ‘cmpi’      then b ← EXTS(SI)
if ‘cmp’ & L=0 then b ← EXTS(GPR(RB)32:63)
if ‘cmp’ & L=1 then b ← GPR(RB)
if a < b then c ← 0b100
if a > b then c ← 0b010
if a = b then c ← 0b001
CR4×BF+32:4×BF+35 ← c || XERSO
```

If ***cmp*** and L=0, the contents of bits 32:63 of GPR(RA) are compared with the contents of bits 32:63 of GPR(RB), treating the operands as signed integers.

If ***cmp*** and L=1, the contents of GPR(RA) are compared with the contents of GPR(RB), treating the operands as signed integers.

If ***cmpi*** and L=0, the contents of bits 32:63 of GPR(RA) are compared with the sign-extended value of the SI field, treating the operands as signed integers.

If ***cmpi*** and L=1, the contents of GPR(RA) are compared with the sign-extended value of the SI field, treating the operands as signed integers.

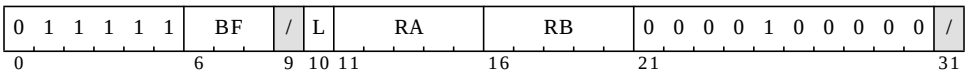
The result of the comparison is placed into CR field BF.

Special Registers Altered:

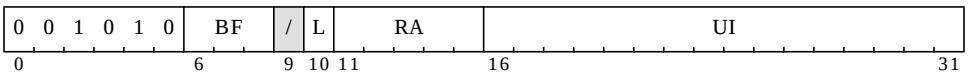
CR field BF

Compare Logical [Immediate]

cmpl BF,L,RA,RB



cmpli BF,L,RA,UI




```
if L=0 then a ← 320 || GPR(RA)32:63
else        a ← GPR(RA)
if ‘cmpli’      then b ← 480 || UI
if ‘cmpl’ & L=0 then b ← 320 || GPR(RB)32:63
if ‘cmpl’ & L=1 then b ← GPR(RB)
if a <u b then c ← 0b100
if a >u b then c ← 0b010
if a =  b then c ← 0b001
CR4×BF+32:4×BF+35 ← c || XERSO
```

If ***cmpl*** and L=0, the contents of bits 32:63 of GPR(RA) are compared with the contents of bits 32:63 of GPR(RB), treating the operands as unsigned integers.

If ***cmpl*** and L=1, the contents of GPR(RA) are compared with the contents of GPR(RB), treating the operands as unsigned integers.

If ***cmpli*** and L=0, the contents of bits 32:63 of GPR(RA) are compared with the zero-extended value of the UI field, treating the operands as unsigned integers.

If ***cmpli*** and L=1, the contents of GPR(RA) 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 BF.

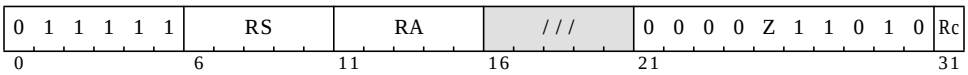
Special Registers Altered:

CR field BF

Count Leading Zeros (Word | Doubleword)

cntlzw RA,RS . . . . . (Z=0, Rc=0)
cntlzw. RA,RS . . . . . (Z=0, Rc=1)

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




```
if 'cntlzd' then n ← 0 else n ← 32
i ← 0
do while n < 64
   if GPR(RS)n = 1 then leave
   n ← n + 1
   i ← i + 1
GPR(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 bit 32 of the contents of GPR(RS) is placed into GPR(RA). This number ranges from 0 to 32, inclusive. If Rc=1, CR Field 0 is set to reflect the result.

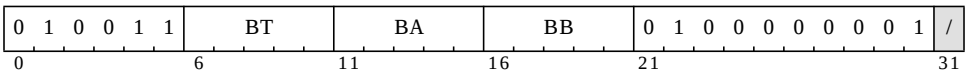
For ***cntlzd***, a count of the number of consecutive zero bits starting at bit 0 of the contents of GPR(RS) is placed into GPR(RA). This number ranges from 0 to 64, inclusive. If Rc=1, the instruction form is invalid.

Special Registers Altered:

CR0 . . . . . .(if Rc=1)

Condition Register AND

crand BT,BA,BB . . . . . 01000)



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

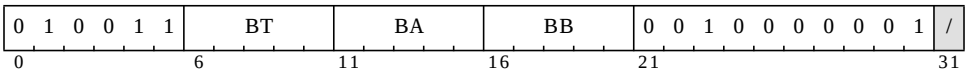
The content of bit BA+32 of the Condition Register is ANDed with the content of bit BB+32 of the Condition Register, and the result is placed into bit BT+32 of the Condition Register.

Special Registers Altered:

CR

Condition Register AND with Complement

crandc BT,BA,BB



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

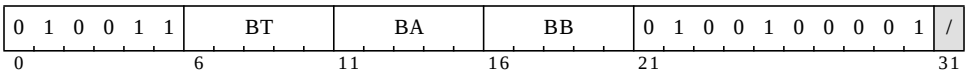
The content of bit BA+32 of the Condition Register is ANDed with the one's complement of the content of bit BB+32 of the Condition Register, and the result is placed into bit BT+32 of the Condition Register.

Special Registers Altered:

CR

Condition Register Equivalent

creqv BT,BA,BB



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

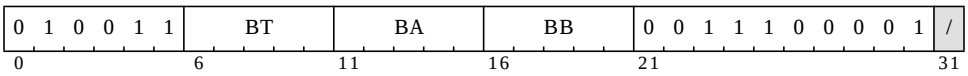
The content of bit BA+32 of the Condition Register is XORed with the content of bit BB+32 of the Condition Register, and the one's complement of result is placed into bit BT+32 of the Condition Register.

Special Registers Altered:

CR

Condition Register NAND

crnand BT,BA,BB



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

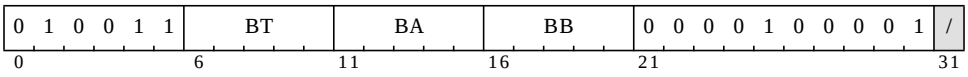
The content of bit BA+32 of the Condition Register is ANDed with the content of bit BB+32 of the Condition Register, and the one's complement of the result is placed into bit BT+32 of the Condition Register.

Special Registers Altered:

CR

Condition Register NOR

crnor BT,BA,BB



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

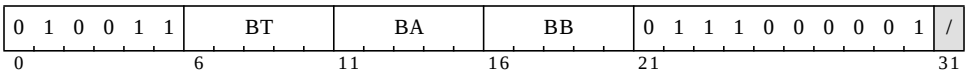
The content of bit BA+32 of the Condition Register is ORed with the content of bit BB+32 of the Condition Register, and the one's complement of the result is placed into bit BT+32 of the Condition Register.

Special Registers Altered:

CR

Condition Register OR

cror BT,BA,BB



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

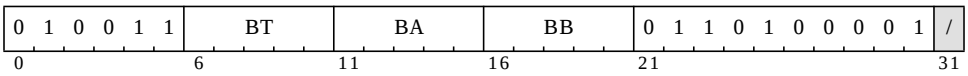
The content of bit BA+32 of the Condition Register is ORed with the content of bit BB+32 of the Condition Register, and the result is placed into bit BT+32 of the Condition Register.

Special Registers Altered:

CR

Condition Register OR with Complement

crorc BT,BA,BB



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

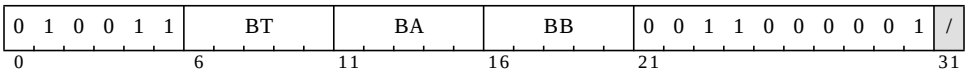
The content of bit BA+32 of the Condition Register is ORed with the one's complement of the content of bit BB+32 of the Condition Register, and the result is placed into bit BT+32 of the Condition Register.

Special Registers Altered:

CR

Condition Register XOR

crxor BT,BA,BB



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

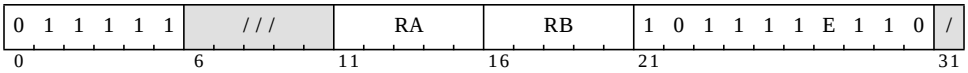
The content of bit BA+32 of the Condition Register is XORed with the content of bit BB+32 of the Condition Register, and the result is placed into bit BT+32 of the Condition Register.

Special Registers Altered:

CR

Data Cache Block Allocate

dcba RA,RB . . . . . (X-mode, E=0)
dcbae RA,RB . . . . . (XE-mode, E=1)




```
if RA=0 then a ← 64 0 else a ← GPR(RA)
if X-mode  then EA ← 32 0 || (a + GPR(RB))32:63
if XE-mode then EA ← a + GPR(RB)
AllocateDataCacheBlock( EA )
```

Let the effective address (EA) be calculated as follows:

Addressing Mode	EA for RA=0	EA for RA≠0
X-mode	$^{32}0$ || GPR(RB)$_{32:63}$	$^{32}0$ || (GPR(RA)+GPR(RB))$_{32:63}$
XE-mode	GPR(RB)	GPR(RA)+GPR(RB)

This instruction is a hint that performance will probably be improved if the block containing the byte addressed by EA is established in the data cache without fetching the block from main storage, because the program will probably soon store into a portion of the block and the contents of the rest of the block are not meaningful to the program. 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 storage that is Memory Coherence Required and the block exists in the data cache(s) of any other processor(s), it is kept coherent in those caches.

This instruction is treated as a *Store* (see Section 6.2.4.4 and Section 6.3.2), except 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, as described in Section 7.4.4, “Machine Check Interrupts,” on page 151.

Special Registers Altered:

None

Engineering Note

If the target block is already in the data cache, leaving the contents of the block unmodified may provide the best performance, especially if the block is Write Through Required. However, setting the contents of the block to zero may be easier to implement, because of the similarity to ***dcbz***[***e***].

If the target block is not already in the data cache and the block is Write Through Required, ignoring the hint may provide the best performance.

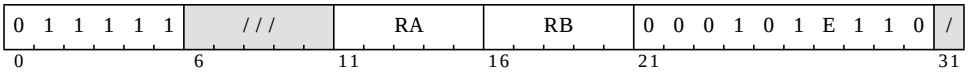
If a ***dcba***[***e***] causes the target block to be newly established in the data cache, processors must set all bytes of the block to zero, and *all* processors must treat the access as a *Store*. In particular, if the newly established block is Write Through Required, the contents of the cache block must be written to main storage.

Architecture Note

dcba[***e***] setting all bytes of newly established cache blocks to zero prevents a program executing the ***dcba***[***e***] from reading the preexisting contents of the block, which may include data that the program is not authorized to read. Such prevention is a requirement of secure systems.

Data Cache Block Flush

dcbf RA,RB . . . . . (X-mode, E=0)
dcbfe RA,RB . . . . . (XE-mode, E=1)




```
if RA=0 then a ← 64 0 else a ← GPR(RA)
if X-mode  then EA ← 32 0 || (a + GPR(RB))32:63
if XE-mode then EA ← a + GPR(RB)
FlushDataCacheBlock( EA )
```

Let the effective address (EA) be calculated as follows:

Addressing Mode	EA for RA=0	EA for RA≠0
X-mode	$^{32}0 \| \| GPR(RB)_{32:63}$	$^{32}0 \| \| (GPR(RA)+GPR(RB))_{32:63}$
XE-mode	GPR(RB)	GPR(RA)+GPR(RB)

If the block containing the byte addressed by EA is in storage that is Memory Coherence Required, a block containing the byte addressed by EA is in the data cache of any processor, and any locations in the block are considered to be modified there, then those locations are written to main storage. Additional locations in the block may also be written to main storage. The block is invalidated in the data caches of all processors.

If the block containing the byte addressed by EA is in storage that is not Memory Coherence Required, a block containing the byte addressed by EA is in the data cache of this processor, and any locations in the block are considered to be modified there, then those locations are written to main storage. Additional locations in the block may also be written to main storage. The block is invalidated in the data cache of this processor.

The function of this instruction is independent of whether the block containing the byte addressed by EA is in storage that is Write Through Required or Caching Inhibited.

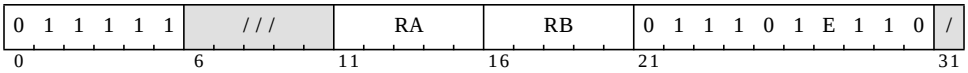
This instruction is treated as a *Load* (see Section 6.2.4.4 and Section 6.3.2).

Special Registers Altered:

None

Data Cache Block Invalidate

dcbi RA,RB . . . . . (X-mode, E=0)
dcbie RA,RB . . . . . (XE-mode, E=1)




```
if RA=0 then a ← 64 0 else a ← GPR(RA)
if X-mode  then EA ← 32 0 || (a + GPR(RB))32:63
if XE-mode then EA ← a + GPR(RB)
InvalidateDataCacheBlock( EA )
```

Let the effective address (EA) be calculated as follows:

Addressing Mode	EA for RA=0	EA for RA≠0
X-mode	$^{32}0 \| \| GPR(RB)_{32:63}$	$^{32}0 \| \| (GPR(RA)+GPR(RB))_{32:63}$
XE-mode	GPR(RB)	GPR(RA)+GPR(RB)

If the block containing the byte addressed by EA is in storage that is Memory Coherence Required and a block containing the byte addressed by EA is in the data cache of any processors, then the block is invalidated in those data caches. On some implementations, before the block is invalidated, if any locations in the block are considered to be modified in any such data cache, those locations are written to main storage and additional locations in the block may be written to main storage.

If the block containing the byte addressed by EA is in storage that is not Memory Coherence Required and a block containing the byte addressed by EA is in the data cache of this processor, then the block is invalidated in that data cache. On some implementations, before the block is invalidated, if any locations in the block are considered to be modified in that data cache, those locations are written to main storage and additional locations in the block may be written to main storage.

The function of this instruction is independent of whether the block containing the byte addressed by EA is in storage that is Write Through Required or Caching Inhibited.

This instruction is treated as a *Store* (see Section 6.2.4.4 and Section 6.3.2) on implementations that invalidate a block without first writing to main storage all locations in the block that are considered to be modified in the data cache, except that the invalidation is not ordered by ***mbar***. On other implementations this instruction is treated as a *Load* (see the section cited above).

Execution of this instruction is privileged and restricted to supervisor mode only.

Additional information about this instruction is as follows.

- The data cache block size for ***dcbi***[***e***] is the same as for ***dcbf***[***e***].
- If a processor holds a reservation and some other processor executes a ***dcbi***[***e***] to the same reservation granule, whether the reservation is lost is undefined.

dcbi[***e***] may cause a cache locking exception. See the User's Manual for the implementation.

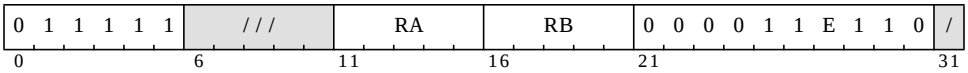
Special Registers Altered:

None

Engineering Note
It is permissible to implement ***dcbi***[***e***] as an instruction that performs the same operations as ***dcbf***[***e***] but is privileged.

Data Cache Block Store

dcbst RA,RB . . . . . (X-mode, E=0)
dcbste RA,RB . . . . . (XE-mode, E=1)




```
if RA=0 then a ← 64 0 else a ← GPR(RA)
if X-mode  then EA ← 32 0 || (a + GPR(RB))32:63
if XE-mode then EA ← a + GPR(RB)
StoreDataCacheBlock( EA )
```

Let the effective address (EA) be calculated as follows:

Addressing Mode	EA for RA=0	EA for RA≠0
X-mode	$^{32}0 \parallel GPR(RB)_{32:63}$	$^{32}0 \parallel (GPR(RA)+GPR(RB))_{32:63}$
XE-mode	GPR(RB)	GPR(RA)+GPR(RB)

If the block containing the byte addressed by EA is in storage that is Memory Coherence Required and a block containing the byte addressed by EA is in the data cache of any processor, and any locations in the block are considered to be modified there, those locations are written to main storage. Additional locations in the block may be written to main storage. The block ceases to be considered to be modified in that data cache.

If the block containing the byte addressed by EA is in storage that is not Memory Coherence Required and 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 storage. Additional locations in the block may be written to main storage. The block ceases to be considered to be modified in that data cache.

The function of this instruction is independent of whether the block containing the byte addressed by EA is in storage that is Write Through Required or Caching Inhibited.

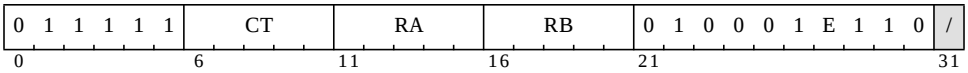
This instruction is treated as a *Load* (see Section 6.2.4.4 and Section 6.3.2).

Special Registers Altered:

None

Data Cache Block Touch

dcbt CT,RA,RB . . . . . (X-mode, E=0)
dcbte CT,RA,RB . . . . . (XE-mode, E=1)




```
if RA=0 then a ← 64 0 else a ← GPR(RA)
if X-mode  then EA ← 32 0 || (a + GPR(RB))32:63
if XE-mode then EA ← a + GPR(RB)
PrefetchDataCacheBlock( CT, EA )
```

Let the effective address (EA) be calculated as follows:

Addressing Mode	EA for RA=0	EA for RA≠0
X-mode	$^{32}0$ || GPR(RB)$_{32:63}$	$^{32}0$ || (GPR(RA)+GPR(RB))$_{32:63}$
XE-mode	GPR(RB)	GPR(RA)+GPR(RB)

If CT=0, this instruction is a hint that performance will probably be improved if the block containing the byte addressed by EA is fetched into the data cache, because the program will probably soon load from the addressed byte.

An implementation may use other values of CT to enable software to target specific, implementation-dependent 'portions of its cache hierarchy or structure that may better enhance performance. See the User's Manual for the implementation.

Implementations should perform no operation when CT specifies a value that is not supported by the implementation.

The hint is ignored if the block is Caching Inhibited.

This instruction is treated as a *Load* (see Section 6.2.4.4 and Section 6.3.2), except that an interrupt is not taken for a translation or protection violation.

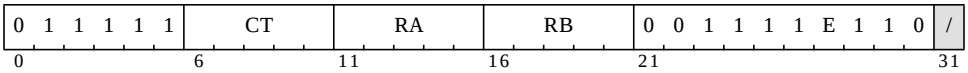
Special Registers Altered:

None

Engineering Note
Programs are likely to execute ***dcbt***[***e***] for several blocks before executing *Load* or *Store* instructions that refer to the first of these blocks. Implementations on which ***dcbt***[***e***] fetches the block into a separate buffer rather than directly into the data cache should provide buffer space sufficient for this use.

Data Cache Block Touch for Store

dcbtst CT,RA,RB . . . . . (X-mode, E=0)
dcbtste CT,RA,RB . . . . . (XE-mode, E=1)




```
if RA=0 then a ← 64 0 else a ← GPR(RA)
if X-mode  then EA ← 32 0 || (a + GPR(RB))32:63
if XE-mode then EA ← a + GPR(RB)
PrefetchForStoreDataCacheBlock( CT, EA )
```

Let the effective address (EA) be calculated as follows:

Addressing Mode	EA for RA=0	EA for RA≠0
X-mode	$^{32}0 \parallel GPR(RB)_{32:63}$	$^{32}0 \parallel (GPR(RA)+GPR(RB))_{32:63}$
XE-mode	GPR(RB)	GPR(RA)+GPR(RB)

If CT=0, this instruction is a hint that performance will probably 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.

An implementation may use other values of CT to enable software to target specific, implementation-dependent portions of its cache hierarchy or structure that may better enhance performance. See the User's Manual for the implementation.

Implementations should perform no operation when CT specifies a value that is not supported by the implementation.

The hint is ignored if the block is Caching Inhibited.

This instruction is treated as a *Load* (see Section 6.2.4.4 and Section 6.3.2), except that an interrupt is not taken for a translation or protection violation.

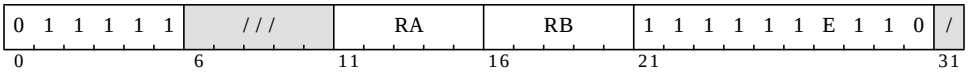
Special Registers Altered:

None

Engineering Note
Executing ***dcbtst***[***e***] does not cause the specified block to be considered to be modified in the data cache.

Data Cache Block set to Zero

dcbz RA,RB . . . . . (X-mode, E=0)
dcbze RA,RB . . . . . (XE-mode, E=1)




```
if RA=0 then a ← 64 0 else a ← GPR(RA)
if X-mode  then EA ← 32 0 || (a + GPR(RB))32:63
if XE-mode then EA ← a + GPR(RB)
ZeroDataCacheBlock( EA )
```

Let the effective address (EA) be calculated as follows:

Addressing Mode	EA for RA=0	EA for RA≠0
X-mode	$^{32}0 \| \| GPR(RB)_{32:63}$	$^{32}0 \| \| (GPR(RA)+GPR(RB))_{32:63}$
XE-mode	GPR(RB)	GPR(RA)+GPR(RB)

If the block containing the byte addressed by EA is in the data cache, all bytes of the block are set to zero.

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 set to zero.

If the block containing the byte addressed by EA is in storage that is Caching Inhibited or Write Through Required, then either (a) all bytes of the area of main storage that corresponds to the addressed block are set to zero, or (b) an Alignment interrupt is taken. See the User's Manual for the implementation.

If the block containing the byte addressed by EA is in storage that is Memory Coherence Required and the block exists in the data cache(s) of any other processor(s), it is kept coherent in those caches.

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, as described in Section 7.4.4, "Machine Check Interrupts," on page 151.

This instruction is treated as a *Store* (see Section 6.2.4.4 and Section 6.3.2).

This instruction may cause a cache locking exception. See the User's Manual for the implementation.

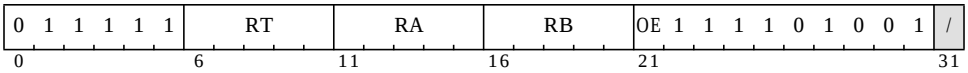
Special Registers Altered:

None

Programming Note
If the block containing the byte addressed by EA is in storage that is Caching Inhibited or Write Through Required, the Alignment interrupt handler should set to zero all bytes of the area of main storage that corresponds to the addressed block.

Divide Doubleword

divd RT,RA,RB . . . . . .(OE=0)
divdo RT,RA,RB . . . . . .(OE=1)




```
dividend0:63 ← GPR(RA)
divisor0:63  ← GPR(RB)
if OE=1 then do
   OV64 ← ( (GPR(RA)=-2^63) & (GPR(RB)=-1) ) | (GPR(RB)=0)
   SO64 ← SO64 | OV64
GPR(RT) ← dividend ÷ divisor
```

The 64-bit quotient of the contents of GPR(RA) divided by the contents of GPR(RB) is placed into GPR(RT). The remainder is not supplied as a result.

Both operands and the quotient are interpreted as signed integers. The quotient is the unique signed integer that satisfies

```
dividend = (quotient × divisor) + r
```

where $0 \le r < |\text{divisor}|$ if the dividend is nonnegative, and $-|\text{divisor}| < r \le 0$ if the dividend is negative.

If an attempt is made to perform any of the divisions

```
0x8000_0000_0000_0000 ÷ -1
<anything> ÷ 0
```

then the contents of GPR(RT) are undefined. In these cases, if OE=1 then OV is set to 1.

Special Registers Altered:

SO64 OV64 . . . . . (if OE=1)

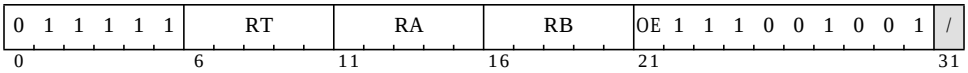
Programming Note

The 64-bit signed remainder of dividing GPR(RA) by GPR(RB) can be computed as follows, except in the case that GPR(RA) = -2^{63} and GPR(RB) = –1.

```
       divd   RT,RA,RB    # RT = quotient
       mulld  RT,RT,RB    # RT = quotient*divisor
       subf   RT,RT,RA    # RT = remainder
```

Divide Doubleword Unsigned

divdu RT,RA,RB . . . . . .(OE=0)
divduo RT,RA,RB . . . . . .(OE=1)




```
dividend0:63 ← GPR(RA)
divisor0:63  ← GPR(RB)
quotient0:63 ← dividend ÷ divisor
if OE=1 then do
   OV64 ← (GPR(RB)=0)
   SO64 ← SO64 | OV64
GPR(RT) ← quotient
```

The 64-bit quotient of the contents of GPR(RA) divided by the contents of GPR(RB) is placed into GPR(RT). The remainder is not supplied as a result.

Both operands and the quotient are interpreted as unsigned integers. The quotient is the unique unsigned integer that satisfies

```
dividend = (quotient × divisor) + r
```

where $0 \le r < \text{divisor}$.

If an attempt is made to perform the division

```
<anything> ÷ 0
```

then the contents of GPR(RT) are undefined. In this case, if OE=1 then OV is set to 1.

Special Registers Altered:

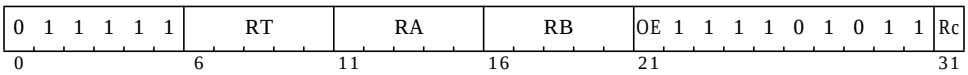
SO64 OV64 . . . . . (if OE=1)

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

```
divdu  RT,RA,RB    # RT = quotient
mulld  RT,RT,RB    # RT = quotient*divisor
subf   RT,RT,RA    # RT = remainder
```

Divide Word

divw RT,RA,RB . . . . . .(OE=0, Rc=0)
divw. RT,RA,RB . . . . . .(OE=0, Rc=1)
divwo RT,RA,RB . . . . . .(OE=1, Rc=0)
divwo. RT,RA,RB . . . . . .(OE=1, Rc=1)




```
dividend0:31 ← GPR(RA)32:63
divisor0:31  ← GPR(RB)32:63
quotient0:31 ← dividend ÷ divisor
if OE=1 then do
   OV ← ( (GPR(RA)32:63=-2^31) & (GPR(RB)32:63=-1) ) | (GPR(RB)32:63=0)
   SO ← SO | OV
if Rc=1 then do
   LT  ← quotient < 0
   GT  ← quotient > 0
   EQ  ← quotient = 0
   CR0 ← LT || GT || EQ || SO
GPR(RT)32:63 ← quotient
GPR(RT)0:31  ← undefined
```

The 32-bit quotient of the contents of bits 32:63 of GPR(RA) divided by the contents of bits 32:63 of GPR(RB) is placed into bits 32:63 of GPR(RT). Bits 0:31 of GPR(RT) are undefined. The remainder is not supplied as a result.

Both operands and the quotient are interpreted as signed integers. The quotient is the unique signed integer that satisfies

```
dividend = (quotient × divisor) + r
```

where $0 \le r < |\text{divisor}|$ if the dividend is nonnegative, and $-|\text{divisor}| < r \le 0$ if the dividend is negative.

If an attempt is made to perform any of the divisions

```
0x8000_0000 ÷ -1
<anything> ÷ 0
```

then the contents of GPR(RT) are undefined as are (if Rc=1) the contents of the LT, GT, and EQ bits of CR Field 0. In these cases, if OE=1 then OV is set to 1.

Special Registers Altered:

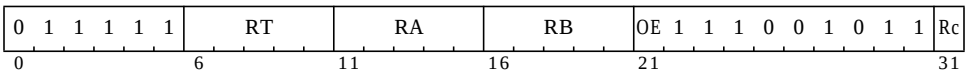
CR0 . . . . . .(if Rc=1)
SO OV. . . . . (if OE=1)

Programming Note
The 32-bit signed remainder of dividing $GPR(RA)_{32:63}$ by $GPR(RB)_{32:63}$ can be computed as follows, except in the case that $GPR(RA)_{32:63} = -2^{31}$ and $GPR(RB)_{32:63} = -1$.

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

Divide Word Unsigned

divwu RT,RA,RB . . . . . .(OE=0, Rc=0)
divwu. RT,RA,RB . . . . . .(OE=0, Rc=1)
divwuo RT,RA,RB . . . . . .(OE=1, Rc=0)
divwuo. RT,RA,RB . . . . . .(OE=1, Rc=1)




```
dividend0:31 ← GPR(RA)32:63
divisor0:31 ← GPR(RB)32:63
quotient0:31 ← dividend ÷ divisor
if OE=1 then do
   OV ← (GPR(RB)32:63=0)
   SO ← SO | OV
if Rc=1 then do
   LT  ← quotient < 0
   GT  ← quotient > 0
   EQ  ← quotient = 0
   CR0 ← LT || GT || EQ || SO
GPR(RT)32:63 ← quotient
GPR(RT)0:31  ← undefined
```

The 32-bit quotient of the contents of bits 32:63 of GPR(RA) divided by the contents of bits 32:63 of GPR(RB) is placed into bits 32:63 of GPR(RT). Bits 0:31 of GPR(RT) are undefined. The remainder is not supplied as a result.

Both operands and the quotient are interpreted as unsigned integers, except that if Rc=1 the first three bits of CR Field 0 are set by signed comparison of the result to zero. The quotient is the unique unsigned integer that satisfies

```
dividend = (quotient × divisor) + r
```

where $0 \le r < \text{divisor}$.

If an attempt is made to perform the division

```
<anything> ÷ 0
```

then the contents of GPR(RT) are undefined as are (if Rc=1) the contents of the LT, GT, and EQ bits of CR Field 0. In this case, if OE=1 then OV is set to 1.

Special Registers Altered:

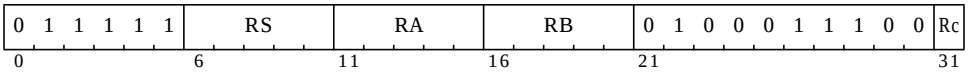
CR0 . . . . . .(if Rc=1)
SO OV. . . . . (if OE=1)

Programming Note
The 32-bit unsigned remainder of dividing $GPR(RA)_{32:63}$ by $GPR(RB)_{32:63}$ can be computed as follows.

```
divwu  RT,RA,RB    # RT = quotient
mullw  RT,RT,RB    # RT = quotient*divisor
subf   RT,RT,RA    # RT = remainder
```

Equivalent

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




```
result0:63 ← GPR(RS) ≡ GPR(RB)
if Rc=1 then do
   LT  ← result32:63 < 0
   GT  ← result32:63 > 0
   EQ  ← result32:63 = 0
   CR0 ← LT || GT || EQ || SO
GPR(RA) ← result
```

The contents of GPR(RS) are XORed with the contents of GPR(RB) and the one's complement of the result is placed into GPR(RA).

Special Registers Altered:

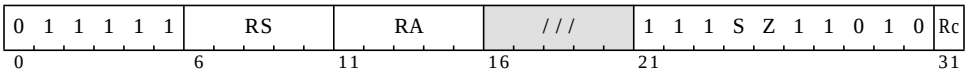
CR0 . . . . . .(if Rc=1)

Extend Sign (Byte | Halfword | Word)

extsb RA,RS . . . . . (SZ=0b01, Rc=0)
extsb. RA,RS . . . . . (SZ=0b01, Rc=1)

extsh RA,RS . . . . . (SZ=0b00, Rc=0)
extsh. RA,RS . . . . . (SZ=0b00, Rc=1)

extsw RA,RS . . . . . (SZ=0b10, Rc=0)




```
if 'extsb[.]' then n ← 56
if '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)n
GPR(RA) ← ns || GPR(RS)n:63
```

For ***extsb*[.]**, the contents of bits 56:63 of GPR(RS) are placed into bits 56:63 of GPR(RA). Bit 56 of the contents of GPR(RS) is copied into bits 0:55 of GPR(RA). If Rc=1, CR Field 0 is set to reflect the result.

For ***extsh*[.]**, the contents of bits 48:63 of GPR(RS) are placed into bits 48:63 of GPR(RA). Bit 48 of the contents of GPR(RS) is copied into bits 0:47 of GPR(RA). If Rc=1, CR Field 0 is set to reflect the result.

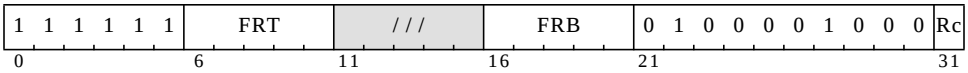
For ***extsw***, the contents of bits 32:63 of GPR(RS) are placed into the contents of bits 32:63 of GPR(RA). Bit 32 of the contents of GPR(RS) is copied into bits 0:31 of GPR(RA). If Rc=1, the instruction form is invalid.

Special Registers Altered:

CR0 . . . . . .(if Rc=1)

Floating Absolute Value

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



FPR(FRT) ← 0b0||FPR(FRB)$_{1:63}$

The contents of FPR(FRB) with bit 0 set to zero are placed into FPR(FRT).

An attempt to execute ***fabs***[**.**] while MSR_{FP}=0 will cause a Floating-Point Unavailable interrupt.

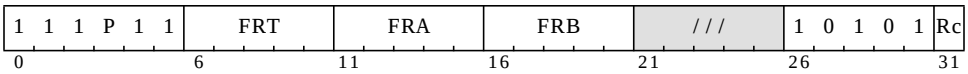
Special Registers Altered:

CR1 ← FX || FEX || VX || OX . . . . . .(if Rc=1)

Floating Add [Single]

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

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




```
if P=1 then FPR(FRT) ← FPR(FRA) +dp FPR(FRB)
else        FPR(FRT) ← FPR(FRA) +sp FPR(FRB)
```

The floating-point operand in FPR(FRA) is added to the floating-point operand in FPR(FRB).

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

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

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

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

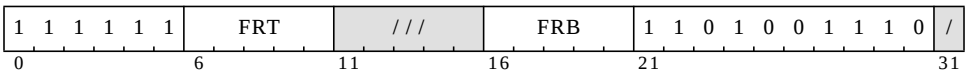
An attempt to execute ***fadd***[***s***][**.**] while MSR_{FP}=0 will cause a Floating-Point Unavailable interrupt.

Special Registers Altered:

FPRF FR FI FX OX UX XX VXSNAN VXISI
CR1 ← FX || FEX || VX || OX . . . . . .(if Rc=1)

Floating Convert From Integer Doubleword

fcfid FRT,FRB




```
sign ← FPR(FRB)0
exp ← 63
frac0:63 ← FPR(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 FPR(FRB) = max negative integer */
   frac0:63 ← frac1:63 || 0b0
   exp ← exp – 1
End

Round Float( sign, exp, frac0:63, FPSCRRN )

If sign = 0 then FPSCRFPRF ← '+normal number'
If sign = 1 then FPSCRFPRF ← '–normal number'
FPR(FRT)0 ← sign
FPR(FRT)1:11 ← exp + 1023   /* exp + bias */
FPR(FRT)12:63 ← frac1:52
Done
```

Zero Operand:

```
FPSCRFR FI ← 0b00
FPSCRFPRF ← '+zero'
FPR(FRT) ← 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
FPSCRFR ← inc
FPSCRFI ← gbit | rbit | xbit
FPSCRXX ← FPSCRXX | FPSCRFI
Return
```

The 64-bit signed operand in FPR(FRB) is converted to an infinitely precise floating-point integer. The result of the conversion is rounded to double-precision, using the rounding mode specified by $FPSCR_{RN}$, and placed into FPR(FRT).

$FPSCR_{FPRF}$ 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.

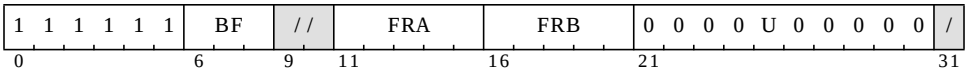
An attempt to execute ***fcfid*** while MSR_{FP}=0 will cause a Floating-Point Unavailable interrupt.

Special Registers Altered:

FPRF FR FI FX XX

Floating Compare

fcmpu BF,FRA,FRB . . . . . .(U=0)
fcmpo BF,FRA,FRB . . . . . .(U=1)




```
if FPR(FRA) is a NaN or
   FPR(FRB) is a NaN then        c ← 0b0001
else if FPR(FRA) < FPR(FRB) then c ← 0b1000
else if FPR(FRA) > FPR(FRB) then c ← 0b0100
else                             c ← 0b0010
FPCC ← c
CR4×BF:4×BF+3 ← c
if 'fcmpu' then do
   if FPR(FRA) is a SNaN or FPR(FRB) is a SNaN then
      VXSNAN ← 1
if 'fcmpo' then do
   if FPR(FRA) is a SNaN or FPR(FRB) is a SNaN then do
      VXSNAN ← 1
      if VE=0 then VXVC ← 1
   else if FPR(FRA) is a QNaN or FPR(FRB) is a QNaN then VXVC ← 1
```

The floating-point operand in FPR(FRA) is compared to the floating-point operand in FPR(FRB). The result of the compare is placed into CR field BF and the FPCC.

If either of the operands is a NaN, either quiet or signaling, then CR field BF and the FPCC are set to reflect unordered.

If ***fcmpu*** then if either of the operands is a Signaling NaN, then 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 and VXSNAN is set. If neither operand is a Signaling NaN but at least one operand is a Quiet NaN, then VXVC is set.

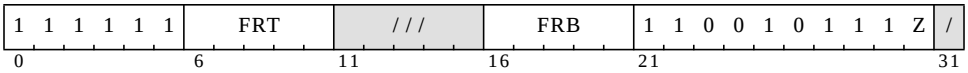
An attempt to execute ***fcmpo*** or ***fcmpu*** while MSR_{FP}=0 will cause a Floating-Point Unavailable interrupt.

Special Registers Altered:

CR field BF
FPCC FX VXSNAN
VXVC . . . . . (if ***fcmpo***)

Floating Convert To Integer Doubleword

fctid FRT,FRB . . . . . (Z=0)
fctidz FRT,FRB . . . . . (Z=1)




```
   if 'fctid[.]'  then round_mode ← FPSCR_RN
   if 'fctidz[.]' then round_mode ← 0b01
   sign ← FPR(FRB)_0
   If FPR(FRB)_1:11 = 2047 and FPR(FRB)_12:63 = 0 then goto Infinity Operand
   If FPR(FRB)_1:11 = 2047 and FPR(FRB)_12 = 0 then goto SNaN Operand
   If FPR(FRB)_1:11 = 2047 and FPR(FRB)_12 = 1 then goto QNaN Operand
   If FPR(FRB)_1:11 > 1086 then goto Large Operand
   If FPR(FRB)_1:11 > 0 then exp ← FPR(FRB)_1:11 – 1023   /* exp – bias */
   If FPR(FRB)_1:11 = 0 then exp ← –1022
/* normal; need leading 0 for later complement */
   If FPR(FRB)_1:11 > 0 then frac_0:64 ← 0b01 || FPR(FRB)_12:63 || [11]0
/* denormal */
   If FPR(FRB)_1:11 = 0 then frac_0:64 ← 0b00 || FPR(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 < FPR(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
   FPR(FRT) ← frac_1:64
   Done

Round Integer( sign,frac_0:64, 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 || frac_64 || gbit || rbit || xbit = 0b0uu1u then inc ← 1
         If sign || frac_64 || gbit || rbit || xbit = 0b0uuu1 then inc ← 1
      End
   If round_mode = 0b11 then    /* comparison ignores u bits */
      Do
         If sign || frac_64 || gbit || rbit || xbit = 0b1u1uu then inc ← 1
         If sign || frac_64 || gbit || rbit || xbit = 0b1uu1u then inc ← 1
         If sign || frac_64 || gbit || rbit || xbit = 0b1uuu1 then inc ← 1
      End

   frac_0:64 ← frac_0:64 + 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 FPR(FRT) ← 0x7FFF_FFFF_FFFF_FFFF
```

```
      If sign = 1 then FPR(FRT) ← 0x8000_0000_0000_0000
      FPSCR_FPRF ← undefined
   End
   Done

SNaN Operand:
   FPSCR_FR FI VXSNAN VXCVI ← 0b0011
   If FPSCR_VE = 0 then Do
      FPR(FRT) ← 0x8000_0000_0000_0000
      FPSCR_FPRF ← undefined
   End
   Done

QNaN Operand:
   FPSCR_FR FI VXCVI ← 0b001
   If FPSCR_VE = 0 then Do
      FPR(FRT) ← 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 FPR(FRT) ← 0x7FFF_FFFF_FFFF_FFFF
      If sign = 1 then FPR(FRT) ← 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 FPR(FRB) is converted to a 64-bit signed integer, using the rounding mode specified by the instruction, and placed into FPR(FRT).

If the floating-point operand in FPR(FRB) is greater than 2^{63}–1, then 0x7FFF_FFFF_FFFF_FFFF is placed into FPR(FRT). If the floating-point operand in FPR(FRB) is less than -2^{63}, then 0x8000_0000_0000_0000 is placed into FPR(FRT).

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.

An attempt to execute ***fctid***[***z***] while MSR_{FP}=0 will cause a Floating-Point Unavailable interrupt.

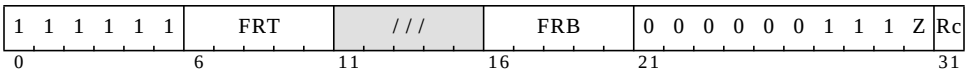
Special Registers Altered:

FPRF (undefined) FR FI FX XX VXSNAN VXCVI

Floating Convert To Integer Word

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

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



if 'fctiw[.]' then round_mode ← $FPSCR_{RN}$
if 'fctiwz[.]' then round_mode ← 0b01
sign ← $FPR(FRB)_0$
If $FPR(FRB)_{1:11}$ = 2047 and $FPR(FRB)_{12:63}$ = 0 then goto Infinity Operand
If $FPR(FRB)_{1:11}$ = 2047 and $FPR(FRB)_{12}$ = 0 then goto SNaN Operand
If $FPR(FRB)_{1:11}$ = 2047 and $FPR(FRB)_{12}$ = 1 then goto QNaN Operand
If $FPR(FRB)_{1:11}$ > 1086 then goto Large Operand
If $FPR(FRB)_{1:11}$ > 0 then exp ← $FPR(FRB)_{1:11}$ – 1023 /* exp – bias */
If $FPR(FRB)_{1:11}$ = 0 then exp ← –1022
/* normal; need leading 0 for later complement */
If $FPR(FRB)_{1:11}$ > 0 then $frac_{0:64}$ ← 0b01 || $FPR(FRB)_{12:63}$ || $^{11}0$
/* denormal */
If $FPR(FRB)_{1:11}$ = 0 then $frac_{0:64}$ ← 0b00 || $FPR(FRB)_{12:63}$ || $^{11}0$
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 $–2^{64}$ < FPR(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

$FPSCR_{XX}$ ← $FPSCR_{XX}$ | $FPSCR_{FI}$

FPR(FRT) ← 0xuuuu_uuuu || $frac_{33:64}$ /* u is undefined hex digit */
$FPSCR_{FPRF}$ ← undefined
Done

Round Integer(sign, $frac_{0:64}$, 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 || $frac_{64}$ || gbit || rbit || xbit = 0b0uu1u then inc ← 1
If sign || $frac_{64}$ || gbit || rbit || xbit = 0b0uuu1 then inc ← 1
End

If round_mode = 0b11 then /* comparison ignores u bits */
Do
If sign || $frac_{64}$ || gbit || rbit || xbit = 0b1u1uu then inc ← 1
If sign || $frac_{64}$ || gbit || rbit || xbit = 0b1uu1u then inc ← 1
If sign || $frac_{64}$ || gbit || rbit || xbit = 0b1uuu1 then inc ← 1
End

$frac_{0:64}$ ← $frac_{0:64}$ + 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 FPR(FRT) ← 0xuuuu_uuuu_7FFF_FFFF
   If sign = 1 then FPR(FRT) ← 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 */
   FPR(FRT) ← 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 */
   FPR(FRT) ← 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 FPR(FRT) ← 0xuuuu_uuuu_7FFF_FFFF
   If sign = 1 then FPR(FRT) ← 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 FPR(FRB) is converted to a 32-bit signed integer, using the rounding mode specified by the instruction, and placed into bits 32:63 of FPR(FRT). Bits 0:31 of FPR(FRT) are undefined.

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

An attempt to execute ***fctiw***[***z***][**.**] while MSR_{FP}=0 will cause a Floating-Point Unavailable interrupt.

Special Registers Altered:

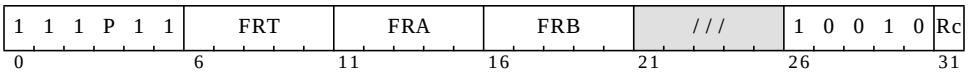
FPRF (undefined) FR FI FX XX VXSNAN VXCVI

CR1 ← FX || FEX || VX || OX . . . . . .(if Rc=1)

Floating Divide [Single]

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

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




```
if P=1 then FPR(FRT) ← FPR(FRA) ÷dp FPR(FRB)
else        FPR(FRT) ← FPR(FRA) ÷sp FPR(FRB)
```

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

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

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

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

An attempt to execute ***fdiv***[***s***][**.**] while MSR_{FP}=0 will cause a Floating-Point Unavailable interrupt.

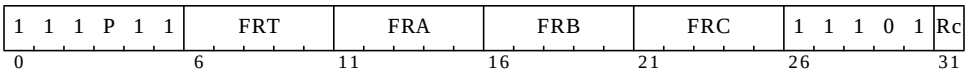
Special Registers Altered:

FPRF FR FI FX OX UX ZX XX VXSNAN VXIDI VXZDZ
CR1 ← FX || FEX || VX || OX . . . . . .(if Rc=1)

Floating Multiply-Add [Single]

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

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




```
if P=1 then FPR(FRT) ← [FPR(FRA) ×fp FPR(FRC)] +dp FPR(FRB)
else        FPR(FRT) ← [FPR(FRA) ×fp FPR(FRC)] +sp FPR(FRB)
```

The floating-point operand in FPR(FRA) is multiplied by the floating-point operand in FPR(FRC). The floating-point operand in FPR(FRB) is added to this intermediate result.

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

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

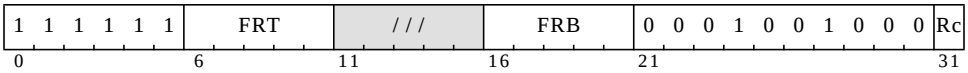
An attempt to execute ***fmadd***[***s***][**.**] while MSR_{FP}=0 will cause a Floating-Point Unavailable interrupt.

Special Registers Altered:

FPRF FR FI FX OX UX XX VXSNAN VXISI VXIMZ
CR1 ← FX || FEX || VX || OX . . . . . .(if Rc=1)

Floating Move Register

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




```
FPR(FRT) ← FPR(FRB)
```

The contents of FPR(FRB) are placed into FPR(FRT).

An attempt to execute ***fmr***[**.**] while MSR_{FP}=0 will cause a Floating-Point Unavailable interrupt.

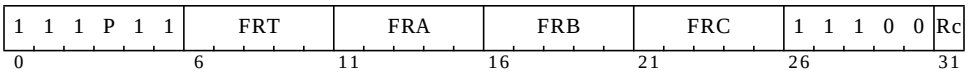
Special Registers Altered:

CR1 ← FX || FEX || VX || OX . . . . . .(if Rc=1)

Floating Multiply-Subtract [Single]

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

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




```
if P=1 then FPR(FRT) ← [FPR(FRA) ×fp FPR(FRC)] -dp FPR(FRB)
else        FPR(FRT) ← [FPR(FRA) ×fp FPR(FRC)] -sp FPR(FRB)
```

The floating-point operand in FPR(FRA) is multiplied by the floating-point operand in FPR(FRC). The floating-point operand in FPR(FRB) is subtracted from this intermediate result.

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

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

An attempt to execute ***fmsub***[***s***][**.**] while MSR_{FP}=0 will cause a Floating-Point Unavailable interrupt.

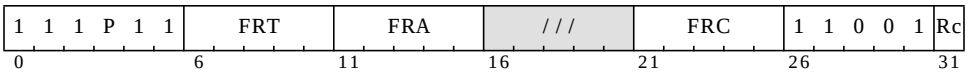
Special Registers Altered:

FPRF FR FI FX OX UX XX VXSNAN VXISI VXIMZ
CR1 ← FX || FEX || VX || OX . . . . . .(if Rc=1)

Floating Multiply [Single]

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

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




```
if P=1 then FPR(FRT) ← FPR(FRA) ×dp FPR(FRC)
else        FPR(FRT) ← FPR(FRA) ×sp FPR(FRC)
```

The floating-point operand in FPR(FRA) is multiplied by the floating-point operand in FPR(FRC).

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

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

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

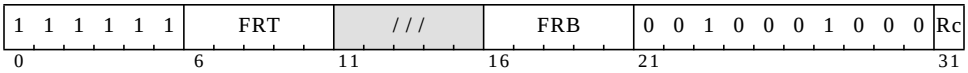
An attempt to execute ***fmul***[***s***][**.**] while MSR_{FP}=0 will cause a Floating-Point Unavailable interrupt.

Special Registers Altered:

FPRF FR FI FX OX UX XX VXSNAN VXIMZ
CR1 ← FX || FEX || VX || OX . . . . . .(if Rc=1)

Floating Negative Absolute Value

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



FPR(FRT) ← 0b1||FPR(FRB)$_{1:63}$

The contents of FPR(FRB) with bit 0 set to one are placed into FPR(FRT).

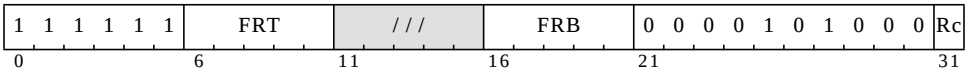
An attempt to execute ***fnabs***[**.**] while MSR_{FP}=0 will cause a Floating-Point Unavailable interrupt.

Special Registers Altered:

CR1 ← FX || FEX || VX || OX . . . . . .(if Rc=1)

Floating Negate

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



FPR(FRT) ← ¬FPR(FRB)$_0$||FPR(FRB)$_{1:63}$

The contents of FPR(FRB) with bit 0 inverted are placed into FPR(FRT).

An attempt to execute ***fneg***[**.**] while MSR_{FP}=0 will cause a Floating-Point Unavailable interrupt.

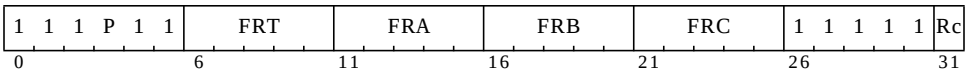
Special Registers Altered:

CR1 ← FX || FEX || VX || OX . . . . . .(if Rc=1)

Floating Negative Multiply-Add [Single]

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

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




```
if P=1 then FPR(FRT) ← -([FPR(FRA) ×fp FPR(FRC)] +dp FPR(FRB))
else        FPR(FRT) ← -([FPR(FRA) ×fp FPR(FRC)] +sp FPR(FRB))
```

The floating-point operand in FPR(FRA) is multiplied by the floating-point operand in FPR(FRC). The floating-point operand in FPR(FRB) is added to this intermediate result.

If the most significant bit of the resultant significand is not 1, the result is normalized. The result is rounded to the target precision under control of the Floating-Point Rounding Control field RN of the Floating-Point Status and Control Register, then negated and placed into FPR(FRT).

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

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

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

An attempt to execute ***fnmadd***[***s***][**.**] while MSR_{FP}=0 will cause a Floating-Point Unavailable interrupt.

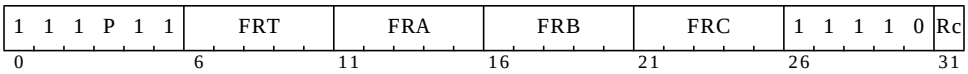
Special Registers Altered:

FPRF FR FI FX OX UX XX VXSNAN VXISI VXIMZ
CR1 ← FX || FEX || VX || OX . . . . . .(if Rc=1)

Floating Negative Multiply-Subtract [Single]

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

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




```
if P=1 then FPR(FRT) ← -([FPR(FRA) ×fp FPR(FRC)] −dp FPR(FRB))
else        FPR(FRT) ← -([FPR(FRA) ×fp FPR(FRC)] −sp FPR(FRB))
```

The floating-point operand in FPR(FRA) is multiplied by the floating-point operand in FPR(FRC). The floating-point operand in FPR(FRB) is subtracted from this intermediate result.

If the most significant bit of the resultant significand is not 1, the result is normalized. The result is rounded to the target precision under control of the Floating-Point Rounding Control field RN of the Floating-Point Status and Control Register, then negated and placed into FPR(FRT).

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

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

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

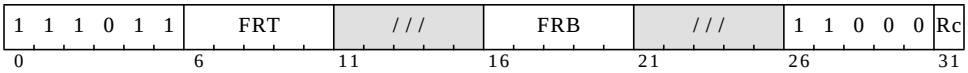
An attempt to execute ***fnmsub***[***s***][**.**] while MSR_{FP}=0 will cause a Floating-Point Unavailable interrupt.

Special Registers Altered:

FPRF FR FI FX OX UX XX VXSNAN VXISI VXIMZ
CR1 ← FX || FEX || VX || OX . . . . . .(if Rc=1)

Floating Reciprocal Estimate Single

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




```
FPR(FRT) ← FPReciprocalEstimate( FPR(FRB) )
```

A single-precision estimate of the reciprocal of the floating-point operand in FPR(FRB) is placed into FPR(FRT). The estimate placed into FPR(FRT) is correct to a precision of one part in 256 of the reciprocal of (FRB), i.e.,

$$\left| \frac{\text{estimate} - \frac{1}{x}}{\frac{1}{x}} \right| \leq \frac{1}{256}$$

where *x* is the initial value in FPR(FRB). Note that the value placed into FPR(FRT) may vary between implementations, and between different executions on the same implementation.

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

Operand	Result	Exception
-∞	-0	None
-0	-∞[1]	ZX
+0	+∞[1]	ZX
+∞	+0	None
SNaN	QNaN[2]	VXSNAN
QNaN	QNaN	None

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

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

An attempt to execute ***fres***[**.**] while MSR_{FP}=0 will cause a Floating-Point Unavailable interrupt.

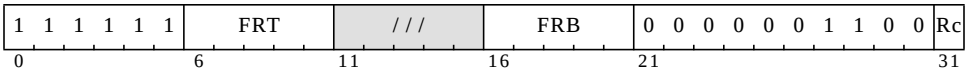
Special Registers Altered:

FPRF FR (undefined) FI (undefined)
FX OX UX ZX VXSNAN
CR1 ← FX || FEX || VX || OX . . . . . .(if Rc=1)

Architecture Note
No double-precision version of this instruction is provided because graphics applications are expected to need only the single-precision version, and no other important performance-critical applications are expected to need a double-precision version.

Floating Round to Single-Precision

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



If $FPR(FRB)_{1:11} < 897$ and $FPR(FRB)_{1:63} > 0$ then Do
  If $FPSCR_{UE} = 0$ then goto Disabled Exponent Underflow
  If $FPSCR_{UE} = 1$ then goto Enabled Exponent Underflow
If $FPR(FRB)_{1:11} > 1150$ and $FPR(FRB)_{1:11} < 2047$ then Do
  If $FPSCR_{OE} = 0$ then goto Disabled Exponent Overflow
  If $FPSCR_{OE} = 1$ then goto Enabled Exponent Overflow
If $FPR(FRB)_{1:11} > 896$ and $FPR(FRB)_{1:11} < 1151$ then goto Normal Operand
If $FPR(FRB)_{1:63} = 0$ then goto Zero Operand
If $FPR(FRB)_{1:11} = 2047$ then Do
  If $FPR(FRB)_{12:63} = 0$ then goto Infinity Operand
  If $FPR(FRB)_{12} = 1$ then goto QNaN Operand
  If $FPR(FRB)_{12} = 0$ and $FPR(FRB)_{13:63} > 0$ then goto SNaN Operand

Disabled Exponent Underflow:
  sign ← $FPR(FRB)_0$
  If $FPR(FRB)_{1:11} = 0$ then Do
    exp ← –1022
    $frac_{0:52}$ ← 0b0 || $FPR(FRB)_{12:63}$
  If $FPR(FRB)_{1:11} > 0$ then Do
    exp ← $FPR(FRB)_{1:11}$ – 1023
    $frac_{0:52}$ ← 0b1 || $FPR(FRB)_{12:63}$

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
  $FPR(FRT)_0$ ← sign
  $FPR(FRT)_{1:63}$ ← 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 $frac_0 = 0$
      exp ← exp–1
      $frac_{0:52}$ ← $frac_{1:52}$ || 0b0
  $FPR(FRT)_0$ ← sign
  $FPR(FRT)_{1:11}$ ← exp + 1023
  $FPR(FRT)_{12:63}$ ← $frac_{1:52}$
Done

Enabled Exponent Underflow:
  $FPSCR_{UX}$ ← 1
  sign ← $FPR(FRB)_0$
  If $FPR(FRB)_{1:11} = 0$ then Do
    exp ← –1022
    $frac_{0:52}$ ← 0b0 || $FPR(FRB)_{12:63}$

```
    If FPR(FRB)1:11 > 0 then Do
       exp ← FPR(FRB)1:11 – 1023
       frac0:52 ← 0b1 || FPR(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)
    FPSCRXX ← FPSCRXX | FPSCRFI
    exp ← exp + 192
    FPR(FRT)0 ← sign
    FPR(FRT)1:11 ← exp + 1023
    FPR(FRT)12:63 ← frac1:52
    If sign = 0 then FPSCRFPRF ← '+normal number'
    If sign = 1 then FPSCRFPRF ← '–normal number'
    Done

Disabled Exponent Overflow
    FPSCROX ← 1
    If FPSCRRN = 0b00 then Do          /* Round to Nearest */
       If FPR(FRB)0 = 0 then FPR(FRT) ← 0x7FF0_0000_0000_0000
       If FPR(FRB)0 = 1 then FPR(FRT) ← 0xFFF0_0000_0000_0000
       If FPR(FRB)0 = 0 then FPSCRFPRF ← '+infinity'
       If FPR(FRB)0 = 1 then FPSCRFPRF ← '–infinity'
    If FPSCRRN = 0b01 then Do          /* Round toward Zero */
       If FPR(FRB)0 = 0 then FPR(FRT) ← 0x47EF_FFFF_E000_0000
       If FPR(FRB)0 = 1 then FPR(FRT) ← 0xC7EF_FFFF_E000_0000
       If FPR(FRB)0 = 0 then FPSCRFPRF ← '+normal number'
       If FPR(FRB)0 = 1 then FPSCRFPRF ← '–normal number'
    If FPSCRRN = 0b10 then Do          /* Round toward +Infinity */
       If FPR(FRB)0 = 0 then FPR(FRT) ← 0x7FF0_0000_0000_0000
       If FPR(FRB)0 = 1 then FPR(FRT) ← 0xC7EF_FFFF_E000_0000
       If FPR(FRB)0 = 0 then FPSCRFPRF ← '+infinity'
       If FPR(FRB)0 = 1 then FPSCRFPRF ← '–normal number'
    If FPSCRRN = 0b11 then Do          /* Round toward –Infinity */
       If FPR(FRB)0 = 0 then FPR(FRT) ← 0x47EF_FFFF_E000_0000
       If FPR(FRB)0 = 1 then FPR(FRT) ← 0xFFF0_0000_0000_0000
       If FPR(FRB)0 = 0 then FPSCRFPRF ← '+normal number'
       If FPR(FRB)0 = 1 then FPSCRFPRF ← '–infinity'
    FPSCRFR ← undefined
    FPSCRFI ← 1
    FPSCRXX ← 1
    Done

Enabled Exponent Overflow:
    sign ← FPR(FRB)0
    exp ← FPR(FRB)1:11 – 1023
    frac0:52 ← 0b1 || FPR(FRB)12:63
    Round Single(sign,exp,frac0:52,0,0)
    FPSCRXX ← FPSCRXX | FPSCRFI
Enabled Overflow:
    FPSCROX ← 1
    exp ← exp – 192
    FPR(FRT)0 ← sign
    FPR(FRT)1:11 ← exp + 1023
    FPR(FRT)12:63 ← frac1:52
    If sign = 0 then FPSCRFPRF ← '+normal number'
    If sign = 1 then FPSCRFPRF ← '–normal number'
    Done

Zero Operand:
    FPR(FRT) ← FPR(FRB)
    If FPR(FRB)0 = 0 then FPSCRFPRF ← '+zero'
    If FPR(FRB)0 = 1 then FPSCRFPRF ← '–zero'
    FPSCRFR FI ← 0b00
    Done

Infinity Operand:
    FPR(FRT) ← FPR(FRB)
    If FPR(FRB)0 = 0 then FPSCRFPRF ← '+infinity'
    If FPR(FRB)0 = 1 then FPSCRFPRF ← '–infinity'
    FPSCRFR FI ← 0b00
    Done
```

```
QNaN Operand:
   FPR(FRT) ← FPR(FRB)0:34 || 29 0
   FPSCRFPRF ← 'QNaN'
   FPSCRFR FI ← 0b00
   Done

SNaN Operand:
   FPSCRVXSNAN ← 1
   If FPSCRVE = 0 then Do
      FPR(FRT)0:11 ← FPR(FRB)0:11
      FPR(FRT)12 ← 1
      FPR(FRT)13:63 ← FPR(FRB)13:34 || 29 0
      FPSCRFPRF ← 'QNaN'
   FPSCRFR FI ← 0b00
   Done

Normal Operand:
   sign ← FPR(FRB)0
   exp ← FPR(FRB)1:11 – 1023
   frac0:52 ← 0b1 || FPR(FRB)12:63
   Round Single(sign,exp,frac0:52,0,0,0)
   FPSCRXX ← FPSCRXX | FPSCRFI
   If exp > 127 and FPSCROE = 0 then go to Disabled Exponent Overflow
   If exp > 127 and FPSCROE = 1 then go to Enabled Overflow
   FPR(FRT)0 ← sign
   FPR(FRT)1:11 ← exp + 1023
   FPR(FRT)12:63 ← frac1:52
   If sign = 0 then FPSCRFPRF ← '+normal number'
   If sign = 1 then FPSCRFPRF ← '–normal number'
   Done

Round Single(sign,exp,frac0:52,G,R,X):
   inc ← 0
   lsb ← frac23
   gbit ← frac24
   rbit ← frac25
   xbit ← (frac26:52||G||R||X)≠0
   If FPSCRRN = 0b00 then 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 FPSCRRN = 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 FPSCRRN = 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
   FPSCRFR ← inc
   FPSCRFI ← gbit | rbit | xbit
   Return
```

The floating-point operand in FPR(FRB) is rounded to single-precision, using the rounding mode specified by $FPSCR_{RN}$, and placed into FPR(FRT).

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

An attempt to execute ***frsp***[**.**] while MSR_{FP}=0 will cause a Floating-Point Unavailable interrupt.

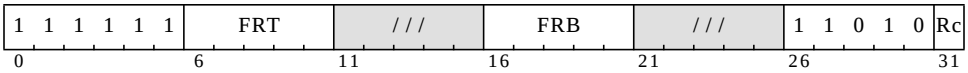
Special Registers Altered:

FPRF FR FI FX OX UX XX VXSNAN

CR1 ← FX || FEX || VX || OX . . . . . .(if Rc=1)

Floating Reciprocal Square Root Estimate

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




```
FPR(FRT) ← FPReciprocalSquareRootEstimate( FPR(FRB) )
```

A double-precision estimate of the reciprocal of the square root of the floating-point operand in FPR(FRB) is placed into FPR(FRT). The estimate placed into FPR(FRT) is correct to a precision of one part in 32 of the reciprocal of the square root of (FRB), i.e.,

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

where *x* is the initial value in FPR(FRB). Note that the value placed into FPR(FRT) may vary between implementations, and between different executions on the same implementation.

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

Operand	Result	Exception
-∞	QNaN[2]	VXSQRT
< 0	QNaN[2]	VXSQRT
-0	-∞[1]	ZX
+0	+∞[1]	ZX
+∞	+0	None
SNaN	QNaN[2]	VXSNAN
QNaN	QNaN	None

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

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

An attempt to execute ***frsqrte***[**.**] while MSR_{FP}=0 will cause a Floating-Point Unavailable interrupt.

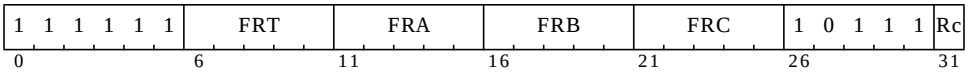
Special Registers Altered:

FPRF FR (undefined) FI (undefined)
FX ZX VXSNAN VXSQRT
CR1 ← FX || FEX || VX || OX . . . . . .(if Rc=1)

Architecture Note
No single-precision version of this instruction is provided because it would be superflous: if (FRB) is representable in single format, then so is (FRT).

Floating Select

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




```
if FPR(FRA) ≥ 0.0 then FPR(FRT) ← FPR(FRC)
else                   FPR(FRT) ← FPR(FRB)
```

The floating-point operand in FPR(FRA) is compared to the value zero. If the operand is greater than or equal to zero, FPR(FRT) is set to the contents of FPR(FRC). If the operand is less than zero or is a NaN, FPR(FRT) is set to the contents of FPR(FRB). The comparison ignores the sign of zero (i.e., regards +0 as equal to –0).

An attempt to execute ***fsel***[**.**] while MSR_{FP}=0 will cause a Floating-Point Unavailable interrupt.

Special Registers Altered:

CR1 ← FX || FEX || VX || OX . . . . . .(if Rc=1)

Architecture Note
The *Select* instruction is similar to a *Move* instruction, and therefore does not alter the Floating-Point Status and Control Register.

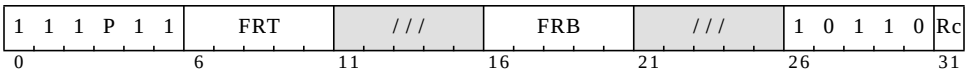
Programming Note
Examples of uses of this instruction can be found in Appendix C.4.

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.4.4 on page 396.

Floating Square Root [Single]

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

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




```
if P=1 then FPR(FRT) ← FPSquareRootDouble( FPR(FRB) )
else        FPR(FRT) ← FPSquareRootSingle( FPR(FRB) )
```

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

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

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

Operand	Result	Exception
–∞	QNaN[1]	VXSQRT
< 0	QNaN[1]	VXSQRT
–0	–0	None
+∞	+∞	None
SNaN	QNaN[1]	VXSNAN
QNaN	QNaN	None

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

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

An attempt to execute ***fsqrt***[***s***][**.**] while MSR_{FP}=0 will cause a Floating-Point Unavailable interrupt.

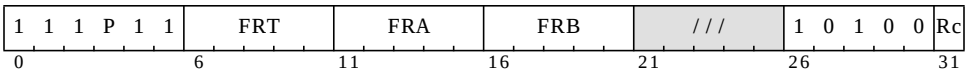
Special Registers Altered:

FPRF FR FI FX XX VXSNAN VXSQRT
CR1 ← FX || FEX || VX || OX . . . . . .(if Rc=1)

Floating Subtract [Single]

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

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




```
if P=1 then FPR(FRT) ← FPR(FRA) -dp FPR(FRB)
else        FPR(FRT) ← FPR(FRA) -sp FPR(FRB)
```

The floating-point operand in FPR(FRB) is subtracted from the floating-point operand in FPR(FRA).

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

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

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

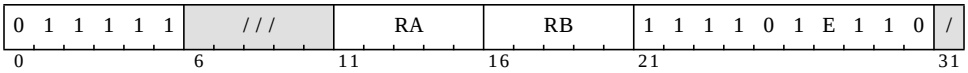
An attempt to execute ***fsub***[***s***][**.**] while MSR_{FP}=0 will cause a Floating-Point Unavailable interrupt.

Special Registers Altered:

FPRF FR FI FX OX UX XX VXSNAN VXISI
CR1 ← FX || FEX || VX || OX . . . . . .(if Rc=1)

Instruction Cache Block Invalidate

icbi RA,RB . . . . . (xop=982, X-mode)
icbie RA,RB . . . . . (xop=990, XE-mode)




```
if RA=0 then a ← 64 0 else a ← GPR(RA)
if X-mode  then EA ← 32 0 || (a + GPR(RB))32:63
if XE-mode then EA ← a + GPR(RB)
InvalidateInstructionCacheBlock( EA )
```

Let the effective address (EA) be calculated as follows:

Addressing Mode	EA for RA=0	EA for RA≠0
X-mode	$^{32}0 \| \| \text{GPR(RB)}_{32:63}$	$^{32}0 \| \| (\text{GPR(RA)}+\text{GPR(RB)})_{32:63}$
XE-mode	GPR(RB)	GPR(RA)+GPR(RB)

If the block containing the byte addressed by EA is in storage that is Memory Coherence Required and a block containing the byte addressed by EA is in the instruction cache of any processors, the block is invalidated in those instruction caches, so that subsequent references cause the block to be fetched from main storage.

If the block containing the byte addressed by EA is in storage that is not Memory Coherence Required and a block containing the byte addressed by EA is in the 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 storage.

The function of this instruction is independent of whether the block containing the byte addressed by EA is in storage that is Write Through Required or Caching Inhibited.

This instruction treated as a *Load* (see Section 6.2.4.4).

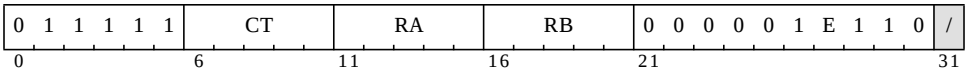
This instruction may cause a cache locking exception. See the User's Manual for the implementation.

Special Registers Altered:

None

Instruction Cache Block Touch

icbt CT,RA,RB . . . . . (X-mode, E=0)
icbte CT,RA,RB . . . . . (XE-mode, E=1)




```
if RA=0 then a ← 64 0 else a ← GPR(RA)
if X-mode  then EA ← 32 0 || (a + GPR(RB))32:63
if XE-mode then EA ← a + GPR(RB)
PrefetchInstructionCacheBlock( CT, EA )
```

Let the effective address (EA) be calculated as follows:

Addressing Mode	**EA for RA=0**	**EA for RA≠0**
X-mode	$^{32}0 \,\|\|\, GPR(RB)_{32:63}$	$^{32}0 \,\|\|\, (GPR(RA)+GPR(RB))_{32:63}$
XE-mode	GPR(RB)	GPR(RA)+GPR(RB)

If CT=0, this instruction is a hint that performance will probably 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.

An implementation may use other values of CT to enable software to target specific, implementation-dependent portions of its cache hierarchy or structure that may better enhance performance. See the User's Manual for the implementation.

Implementations should perform no operation when CT specifies a value that is not supported by the implementation.

The hint is ignored if the block is Caching Inhibited.

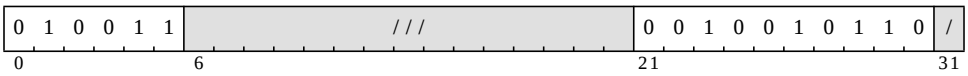
This instruction treated as a *Load* (see Section 6.2.4.4), except that an interrupt is not taken for a translation or protection violation.

Special Registers Altered:

None

Instruction Synchronize

isync



The ***isync*** instruction provides an ordering function for the effects of all instructions executed by the processor executing the ***isync*** instruction. Executing an ***isync*** instruction ensures that all instructions preceding the ***isync*** instruction have completed before the ***isync*** instruction completes, and that no subsequent instructions are initiated until after the ***isync*** instruction completes. It also causes any prefetched instructions to be discarded, with the effect that subsequent instructions will be fetched and executed in the context established by the instructions preceding the ***isync*** instruction.

The ***isync*** instruction may complete before storage accesses associated with instructions preceding the ***isync*** instruction have been performed.

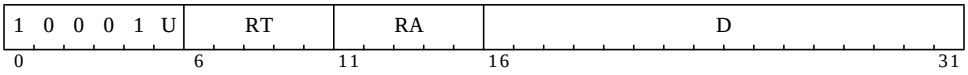
This instruction is context synchronizing (see Section 1.12.1 on page 38).

Special Registers Altered:

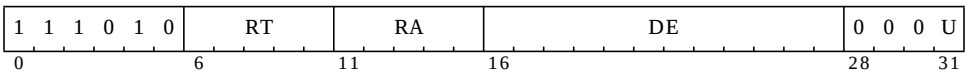
None

Load Byte and Zero [with Update] [Indexed] [Extended]

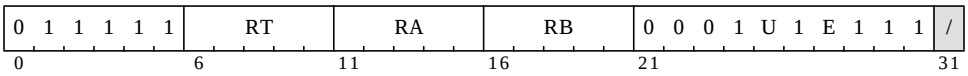
lbz RT,D(RA) . . . . . .(D-mode, U=0)
lbzu RT,D(RA) . . . . . .(D-mode, U=1)



lbze RT,DE(RA). . . . . .(DE-mode, U=0)
lbzue RT,DE(RA). . . . . .(DE-mode, U=1)



lbzx RT,RA,RB . . . . . .(X-mode, E=0, U=0)
lbzux RT,RA,RB . . . . . .(X-mode, E=0, U=1)
lbzxe RT,RA,RB . . . . . .(XE-mode, E=1, U=0)
lbzuxe RT,RA,RB . . . . . .(XE-mode, E=1, U=1)




```
if RA=0 then a ← 64 0 else a ← GPR(RA)
if D-mode  then EA ← 32 0 || (a + EXTS(D))32:63
if X-mode  then EA ← 32 0 || (a + GPR(RB))32:63
if DE-mode then EA ← a + EXTS(DE)
if XE-mode then EA ← a + GPR(RB)
GPR(RT) ← 56 0 || MEM(EA,1)
if U=1 then GPR(RA) ← EA
```

Let the effective address (EA) be calculated as follows:

- For ***lbz*** and ***lbzu***, let EA be 32 0s concatenated with bits 32:63 of the sum of the contents of GPR(RA), or 64 0s if RA=0, and the sign-extended value of the D instruction field.

- For ***lbzx*** and ***lbzux***, let EA be 32 0s concatenated with bits 32:63 of the sum of the contents of GPR(RA), or 64 0s if RA=0, and the contents of GPR(RB).

- For ***lbze*** and ***lbzue***, let EA be the sum of the contents of GPR(RA), or 64 0s if RA=0, and the sign-extended value of the DE instruction field.

- For ***lbzxe*** and ***lbzuxe***, let EA be the sum of the contents of GPR(RA), or 64 0s if RA=0, and the contents of GPR(RB).

The byte in storage addressed by EA is loaded into bits 56:63 of GPR(RT). Bits 0:55 of GPR(RT) are set to 0.

If U=1 (‘with update’), EA is placed into GPR(RA).

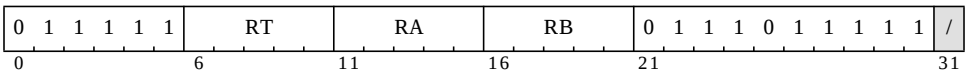
If U=1 (‘with update’), and RA=0 or RA=RT, the instruction form is invalid.

Special Registers Altered:

None

Load Doubleword And Reserve Indexed Extended

ldarxe RT,RA,RB




```
if RA=0 then EA ← GPR(RB)
else         EA ← GPR(RA) + GPR(RB)
RESERVE ← 1
RESERVE_ADDR ← real_addr(EA)
GPR(RT) ← MEM(EA,8)
```

Let the effective address (EA) be the sum of the contents of GPR(RA), or 64 0s if RA=0, and the contents of GPR(RB).

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

This instruction creates a reservation for use by a *Store Doubleword Conditional* instruction. An address computed from the EA is associated with the reservation and replaces any address previously associated with the reservation: the manner in which the address to be associated with the reservation is computed from the EA is described in Section 6.1.6.2 on page 117.

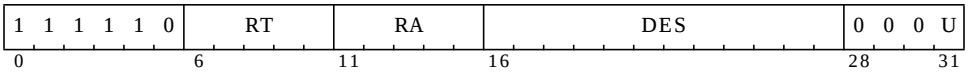
EA must be a multiple of 8. If it is not, either an Alignment interrupt is invoked or the results are boundedly undefined.

Special Registers Altered:

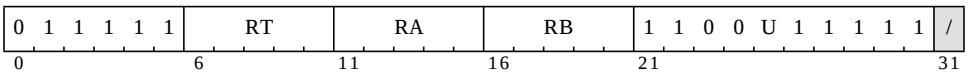
None

Load Doubleword [with Update] [Indexed] Extended

lde RT,DES(RA). . . . . .(DES-mode, U=0)
ldue RT,DES(RA). . . . . .(DES-mode, U=1)



ldxe RT,RA,RB . . . . . .(XE-mode, U=0)
lduxe RT,RA,RB . . . . . .(XE-mode, U=1)




```
if RA=0 then a ← 64'0 else a ← GPR(RA)
if DES-mode then EA ← a + EXTS(DES||0b00)
if XE-mode  then EA ← a + GPR(RB)
GPR(RT) ← MEM(EA,8)
if U=1 then GPR(RA) ← EA
```

Let the effective address (EA) be calculated as follows:

- For ***lde*** and ***ldue***, let EA be the sum of the contents of GPR(RA), or 64 0s if RA=0, and the sign-extended value of the DES instruction field concatenated with 0b00.

- For ***ldxe*** and ***lduxe***, let EA be the sum of the contents of GPR(RA), or 64 0s if RA=0, and the contents of GPR(RB).

The doubleword in storage addressed by the EA is loaded into GPR(RT).

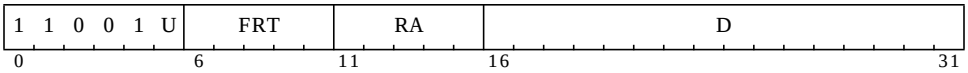
If U=1 ('with update'), EA is placed into GPR(RA).

If U=1 ('with update'), and RA=0 or RA=RT, the instruction form is invalid.

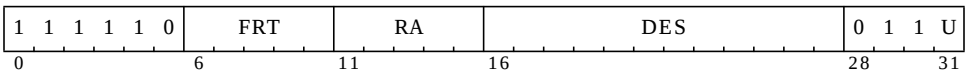
Special Registers Altered:
None

Load Floating-Point Double

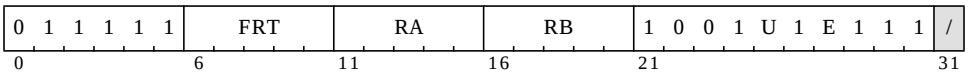
lfd FRT,D(RA) . . . . . .(D-mode, U=0)
lfdu FRT,D(RA) . . . . . .(D-mode, U=1)



lfde FRT,DES(RA). . . . . .(DES-mode, U=0)
lfdue FRT,DES(RA). . . . . .(DES-mode, U=1)



lfdx FRT,RA,RB . . . . . .(X-mode, E=0, U=0)
lfdux FRT,RA,RB . . . . . .(X-mode, E=0, U=1)
lfdxe FRT,RA,RB . . . . . .(XE-mode, E=1, U=0)
lfduxe FRT,RA,RB . . . . . .(XE-mode, E=1, U=1)




```
if RA=0 then a ← 640 else a ← GPR(RA)
if D-mode   then EA ← 320 || (a + EXTS(D))32:63
if X-mode   then EA ← 320 || (a + GPR(RB))32:63
if DES-mode then EA ← a + EXTS(DES||0b00)
if XE-mode  then EA ← a + GPR(RB)
FPR(FRT) ← MEM(EA,8)
if U=1 then GPR(RA) ← EA
```

Let the effective address (EA) be calculated as follows:

- For ***lfd*** and ***lfdu***, let EA be 32 0s concatenated with bits 32:63 of the sum of the contents of GPR(RA), or 64 0s if RA=0, and the sign-extended value of the D instruction field.

- For ***lfdx*** and ***lfdux***, let EA be 32 0s concatenated with bits 32:63 of the sum of the contents of GPR(RA), or 64 0s if RA=0, and the contents of GPR(RB).

- For ***lfde*** and ***lfdue***, let EA be the sum of the contents of GPR(RA), or 64 0s if RA=0, and the sign-extended value of the DES instruction fielfd concatenated with 0b00.

- For ***lfdxe*** and ***lfduxe***, let EA be the sum of the contents of GPR(RA), or 64 0s if RA=0, and the contents of GPR(RB).

The doubleword in storage addressed by EA is placed into FPR(FRT).

If U=1 (‘with update’), EA is placed into register RA.

If U=1 (‘with update’) and RA=0, the instruction form is invalid.

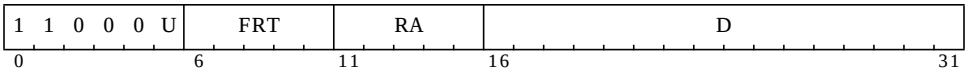
An attempt to execute ***lfd***[***u***][***x***][***e***] while MSR_{FP}=0 will cause a Floating-Point Unavailable interrupt.

Special Registers Altered:

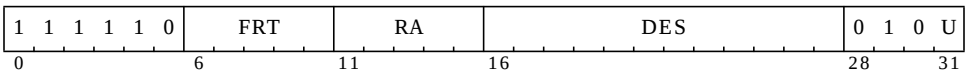
None

Load Floating-Point Single

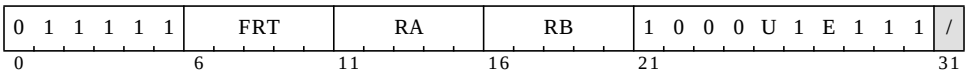
lfs FRT,D(RA) . . . . . .(D-mode, U=0)
lfsu FRT,D(RA) . . . . . .(D-mode, U=1)



lfse FRT,DES(RA). . . . . .(DES-mode, U=0)
lfsue FRT,DES(RA). . . . . .(DES-mode, U=1)



lfsx FRT,RA,RB . . . . . .(X-mode, E=0, U=0)
lfsux FRT,RA,RB . . . . . .(X-mode, E=0, U=1)
lfsxe FRT,RA,RB . . . . . .(XE-mode, E=1, U=0)
lfsuxe FRT,RA,RB . . . . . .(XE-mode, E=1, U=1)




```
if RA=0 then a ← 64 0 else a ← GPR(RA)
if D-mode   then EA ← 32 0 || (a + EXTS(D))32:63
if X-mode   then EA ← 32 0 || (a + GPR(RB))32:63
if DES-mode then EA ← a + EXTS(DES||0b00)
if XE-mode  then EA ← a + GPR(RB)
FPR(FRT) ← DOUBLE(MEM(EA,4))
if U=1 then GPR(RA) ← EA
```

Let the effective address (EA) be calculated as follows:

- For ***lfs*** and ***lfsu***, let EA be 32 0s concatenated with bits 32:63 of the sum of the contents of GPR(RA), or 64 0s if RA=0, and the sign-extended value of the D instruction field.

- For ***lfsx*** and ***lfsux***, let EA be 32 0s concatenated with bits 32:63 of the sum of the contents of GPR(RA), or 64 0s if RA=0, and the contents of GPR(RB).

- For ***lfse*** and ***lfsue***, let EA be the sum of the contents of GPR(RA), or 64 0s if RA=0, and the sign-extended value of the DES instruction fielfs concatenated with 0b00.

- For ***lfsxe*** and ***lfsuxe***, let EA be the sum of the contents of GPR(RA), or 64 0s if RA=0, and the contents of GPR(RB).

The word in storage addressed by EA is interpreted as a floating-point single-precision operand. This word is converted to floating-point double format (see page 98) and placed into FPR(FRT).

If U=1 (‘with update’), EA is placed into register RA.

If U=1 (‘with update’) and RA=0, the instruction form is invalid.

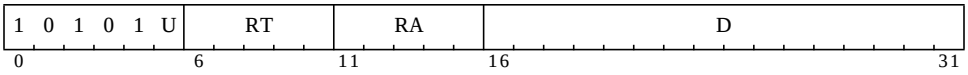
An attempt to execute ***lfs***[***u***][***x***][***e***] while MSR_{FP}=0 will cause a Floating-Point Unavailable interrupt.

Special Registers Altered:

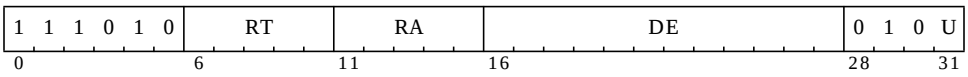
None

Load Halfword Algebraic [with Update] [Indexed] [Extended]

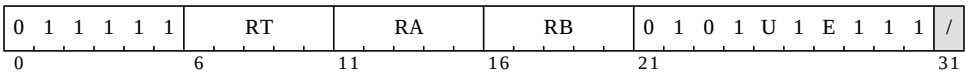
lha RT,D(RA) . . . . . .(D-mode, U=0)
lhau RT,D(RA) . . . . . .(D-mode, U=1)



lhae RT,DE(RA) . . . . . .(DE-mode, U=0)
lhaue RT,DE(RA) . . . . . .(DE-mode, U=1)



lhax RT,RA,RB . . . . . .(X-mode, E=0, U=0)
lhaux RT,RA,RB . . . . . .(X-mode, E=0, U=1)
lhaxe RT,RA,RB . . . . . .(XE-mode, E=1, U=0)
lhauxe RT,RA,RB . . . . . .(XE-mode, E=1, U=1)




```
if RA=0 then a ← 640 else a ← GPR(RA)
if D-mode  then EA ← 320 || (a + EXTS(D))32:63
if X-mode  then EA ← 320 || (a + GPR(RB))32:63
if DE-mode then EA ← a + EXTS(DE)
if XE-mode then EA ← a + GPR(RB)
GPR(RT) ← 320 || EXTS(MEM(EA,2))32:63
if U=1 then GPR(RA) ← EA
```

Let the effective address (EA) be calculated as follows:

- For ***lha*** and ***lhau***, let EA be 32 0s concatenated with bits 32:63 of the sum of the contents of GPR(RA), or 64 0s if RA=0, and the sign-extended value of the D instruction field.

- For ***lhax*** and ***lhaux***, let EA be 32 0s concatenated with bits 32:63 of the sum of the contents of GPR(RA), or 64 0s if RA=0, and the contents of GPR(RB).

- For ***lhae*** and ***lhaue***, let EA be the sum of the contents of GPR(RA), or 64 0s if RA=0, and the sign-extended value of the DE instruction field.

- For ***lhaxe*** and ***lhauxe***, let EA be the sum of the contents of GPR(RA), or 64 0s if RA=0, and the contents of GPR(RB).

The halfword in storage addressed by EA is loaded into bits 48:63 of GPR(RT). Bits 32:47 of GPR(RT) are filled with a copy of bit 0 of the loaded halfword. Bits 0:31 of GPR(RT) are set to 0.

If U=1 (‘with update’), EA is placed into GPR(RA).

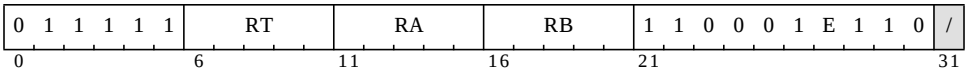
If U=1 (‘with update’), and RA=0 or RA=RT, the instruction form is invalid.

Special Registers Altered:

None

Load Halfword Byte-Reverse Indexed [Extended]

lhbrx RT,RA,RB . . . . . (X-mode, E=0)
lhbrxe RT,RA,RB . . . . . (XE-mode, E=1)




```
if RA=0 then a ← 64 0 else a ← GPR(RA)
if X-mode  then EA ← 32 0 || (a + GPR(RB))32:63
if XE-mode then EA ← a + GPR(RB)
data0:15 ← MEM(EA,2)
GPR(RT) ← 48 0 || data8:15 || data0:7
```

Let the effective address (EA) be calculated as follows:

- For ***lhbrx***, let EA be 32 0s concatenated with bits 32:63 of the sum of the contents of GPR(RA), or 64 0s if RA=0, and the contents of GPR(RB).

- For ***lhbrxe***, let EA be the sum of the contents of GPR(RA), or 64 0s if RA=0, and the contents of GPR(RB).

Bits 0:7 of the halfword in storage addressed by EA are loaded into bits 56:63 of GPR(RT). Bits 8:15 of the halfword in storage addressed by EA are loaded into bits 48:55 of GPR(RT). Bits 0:47 of GPR(RT) are set to 0.

Special Registers Altered:

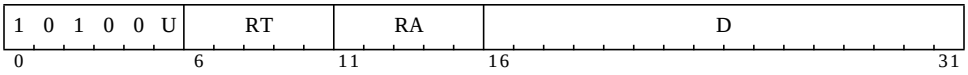
None

Programming Note
When EA references Big-Endian storage, these instructions have the effect of loading data in Little-Endian byte order. Likewise, when EA references Little-Endian storage, these instructions have the effect of loading data in Big-Endian byte order.

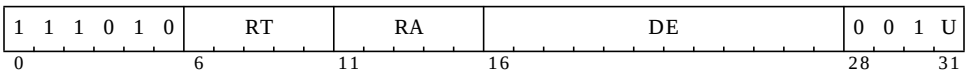
Programming Note
In some implementations, the *Load Halfword Byte-Reverse Indexed* instructions may have greater latency than other *Load* instructions.

Load Halfword and Zero [with Update] [Indexed] [Extended]

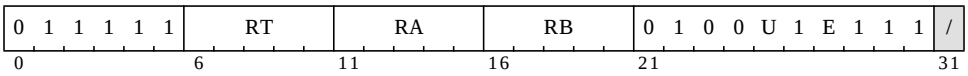
lhz RT,D(RA) . . . . . .(D-mode, U=0)
lhzu RT,D(RA) . . . . . .(D-mode, U=1)



lhze RT,DE(RA) . . . . . .(DE-mode, U=0)
lhzue RT,DE(RA) . . . . . .(DE-mode, U=1)



lhzx RT,RA,RB . . . . . .(X-mode, E=0, U=0)
lhzux RT,RA,RB . . . . . .(X-mode, E=0, U=1)
lhzxe RT,RA,RB . . . . . .(XE-mode, E=1, U=0)
lhzuxe RT,RA,RB . . . . . .(XE-mode, E=1, U=1)




```
if RA=0 then a ← 64 0 else a ← GPR(RA)
if D-mode  then EA ← 32 0 || (a + EXTS(D))32:63
if X-mode  then EA ← 32 0 || (a + GPR(RB))32:63
if DE-mode then EA ← a + EXTS(DE)
if XE-mode then EA ← a + GPR(RB)
GPR(RT) ← 48 0 || MEM(EA,2)
if U=1 then GPR(RA) ← EA
```

Let the effective address (EA) be calculated as follows:

- For ***lhz*** and ***lhzu***, let EA be 32 0s concatenated with bits 32:63 of the sum of the contents of GPR(RA), or 64 0s if RA=0, and the sign-extended value of the D instruction field.

- For ***lhzx*** and ***lhzux***, let EA be 32 0s concatenated with bits 32:63 of the sum of the contents of GPR(RA), or 64 0s if RA=0, and the contents of GPR(RB).

- For ***lhze*** and ***lhzue***, let EA be the sum of the contents of GPR(RA), or 64 0s if RA=0, and the sign-extended value of the DE instruction field.

- For ***lhzxe*** and ***lhzuxe***, let EA be the sum of the contents of GPR(RA), or 64 0s if RA=0, and the contents of GPR(RB).

The halfword in storage addressed by EA is loaded into bits 48:63 of GPR(RT). Bits 0:47 of GPR(RT) are set to 0.

If U=1 (‘with update’), EA is placed into GPR(RA).

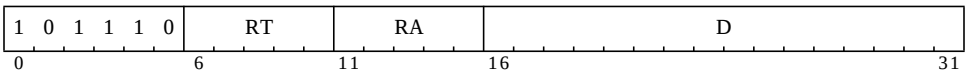
If U=1 (‘with update’), and RA=0 or RA=RT, the instruction form is invalid.

Special Registers Altered:

None

Load Multiple Word

lmw RT,D(RA)




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

Let the effective address (EA) be 32 0s concatenated with bits 32:63 of the sum of the contents of GPR(RA), or 64 0s if RA=0, and the sign-extended value of the D instruction field.

Let n=(32-RT). n consecutive words in storage starting at address EA are loaded into bits 32:63 of registers GPR(RT) through GPR(31). Bits 0:31 of these GPRs are set to zero.

EA must be a multiple of 4. If it is not, either an Alignment interrupt is invoked or the results are boundedly undefined. If RA is in the range of registers to be loaded, including the case in which RA=0, the instruction form is invalid.

Special Registers Altered:

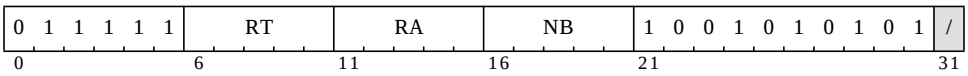
None

Engineering Note
Causing an Alignment interrupt if an attempt is made to execute a *Load Multiple* instruction having an incorrectly aligned effective address facilitates the debugging of software.

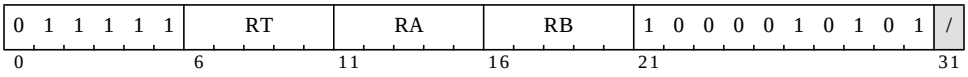
Architecture Note
Extended addressing modes are not defined for *Load Multiple*. Doubleword forms of *Load Multiple* are not defined.

Load String Word (Immediate | Indexed)

lswi RT,RA,NB



lswx RT,RA,RB




```
if RA=0 then a ← 64 0 else a ← GPR(RA)
if 'lswi' then EA ← 32 0 ||  a32:63
if 'lswx' then EA ← 32 0 || (a + GPR(RB))32:63
if 'lswi' & NB=0 then n ← 32
if 'lswi' & NB≠0 then n ← NB
if 'lswx'        then n ← XER57:63
r ← RT - 1
i ← 32
GPR(RT) ← undefined
do while n > 0
   if i = 32 then
      r ← r + 1 (mod 32)
      GPR(r) ← 0
   GPR(r)i:i+7 ← MEM(EA,1)
   i ← i + 8
   if i = 64 then i ← 32
   EA ← 32 0 || (EA+1)32:63
   n  ← n - 1
```

Let the effective address (EA) be calculated as follows:

- For ***lswi***, let EA be 32 0s concatenated with the contents of bits 32:63 of GPR(RA), or 32 0s if RA=0.

- For ***lswx***, let EA be 32 0s concatenated with bits 32:63 of the sum of the contents of GPR(RA), or 64 0s if RA=0, and the contents of GPR(RB).

If ***lswi*** then n=NB if NB≠0, n=32 if NB=0. If ***lswx*** then n=$XER_{57:63}$. n is the number of bytes to load. Let nr=CEIL(n÷4): nr is the number of registers to receive data.

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

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

If ***lswx*** and n=0, the contents of GPR(RT) are undefined.

If RA, or RB for ***lswx***, is in the range of registers to be loaded, including the case in which RA=0, either an Illegal Instruction type Program interrupt is invoked or the results are boundedly undefined. If RT=RA, or RT=RB for ***lswx***, the instruction form is invalid.

Special Registers Altered:

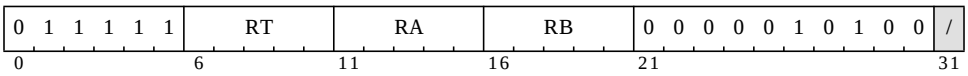
None

Programming Note
The *Load String Word* instructions, in combination with the *Store String Word* instructions allow movement of data from storage to registers or from registers to storage without concern for alignment. These instructions can be used for a short move between arbitrary storage locations or to initiate a long move between unaligned storage fields.

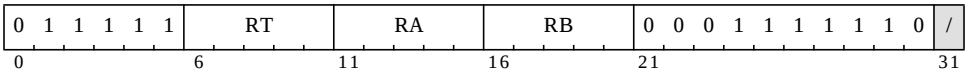
Architecture Note
Extended addressing modes are not defined for the *Load String Word* instructions. Doubleword forms of the *Load String Word* instructions are not defined.

Load Word And Reserve Indexed [Extended]

lwarx RT,RA,RB . . . . . (X-mode)



lwarxe RT,RA,RB . . . . . (XE-mode)




```
if RA=0 then a ← 64 0 else a ← GPR(RA)
if X-mode  then EA ← 32 0 || (a + GPR(RB))32:63
if XE-mode then EA ← a + GPR(RB)
RESERVE ← 1
RESERVE_ADDR ← real_addr(EA)
GPR(RT) ← 32 0 || MEM(EA,4)
```

Let the effective address (EA) be calculated as follows:

- For ***lwarx***, let EA be 32 0s concatenated with bits 32:63 of the sum of the contents of GPR(RA), or 64 0s if RA=0, and the contents of GPR(RB).

- For ***lwarxe***, let EA be the sum of the contents of GPR(RA), or 64 0s if RA=0, and the contents of GPR(RB).

The word in storage addressed by EA is loaded into $GPR(RT)_{32:63}$. $GPR(RT)_{0:31}$ are set to 0.

This instruction creates a reservation for use by a *Store Word Conditional* instruction. An address computed from the EA is associated with the reservation and replaces any address previously associated with the reservation: the manner in which the address to be associated with the reservation is computed from the EA is described in Section 6.1.6.2 on page 117.

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

Programming Note

lwarx, ***lwarxe***, and ***ldarxe***, in combination with ***stwcx.***, ***stwcxe.***, and ***stdcxe.***, permit the programmer to write a sequence of instructions that appear to perform an atomic update operation on a storage location. This operation depends upon 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 doublewords.

Programming Note

Because ***lwarx***, ***lwarxe***, and ***ldarxe*** have implementation dependencies (e.g., the granularity at which reservations are managed), they must be used with care. The operating system should provide system library programs that use these instructions to implement the high-level synchronization functions (Test and Set, Compare and Swap, etc.) needed by application programs. Application programs should use these library programs, rather than use ***lwarx***, ***lwarxe***, and ***ldarxe*** directly.

Architecture Note

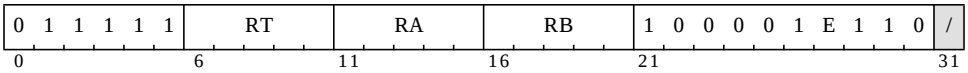
lwarx, ***lwarxe***, and ***ldarxe*** require the EA to be aligned. Software should not attempt to emulate an unaligned ***lwarx***, ***lwarxe***, or ***ldarxe***, because there is no correct way to define the address associated with the reservation.

Engineering Note
Causing an Alignment interrupt to be invoked if an attempt is made to execute a ***lwarx***, ***lwarxe***, or ***ldarxe*** having an incorrectly aligned effective address facilitates the debugging of software by signalling the exception when and where the exception occurs.

Programming Note
The granularity with which reservations are managed is implementation-dependent. Therefore the storage to be accessed by ***lwarx***, ***lwarxe***, or ***ldarxe*** should be allocated by a system library program. Additional information can be found in Section 6.1.6.2 on page 117.

Load Word Byte-Reverse Indexed [Extended]

lwbrx RT,RA,RB . . . . . (X-mode, E=0)
lwbrxe RT,RA,RB . . . . . (XE-mode, E=1)




```
if RA=0 then a ← 64 0 else a ← GPR(RA)
if X-mode  then EA ← 32 0 || (a + GPR(RB))32:63
if XE-mode then EA ← a + GPR(RB)
data0:31 ← MEM(EA,4)
GPR(RT) ← 32 0 || data24:31 || data16:23 || data8:15 || data0:7
```

Let the effective address (EA) be calculated as follows:

- For ***lwbrx***, let EA be 32 0s concatenated with bits 32:63 of the sum of the contents of GPR(RA), or 64 0s if RA=0, and the contents of GPR(RB).

- For ***lwbrxe***, let EA be the sum of the contents of GPR(RA), or 64 0s if RA=0, and the contents of GPR(RB).

Bits 0:7 of the word in storage addressed by EA are loaded into bits 56:63 of GPR(RT). Bits 8:15 of the word in storage addressed by EA are loaded into bits 48:55 of GPR(RT). Bits 16:23 of the word in storage addressed by EA are loaded into bits 40:47 of GPR(RT). Bits 24:31 of the word in storage addressed by EA are loaded into bits 32:39 of GPR(RT). Bits 0:31 of GPR(RT) are set to 0.

Special Registers Altered:

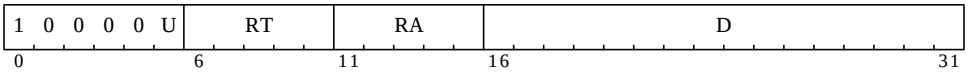
None

Programming Note
When EA references Big-Endian storage, these instructions have the effect of loading data in Little-Endian byte order. Likewise, when EA references Little-Endian storage, these instructions have the effect of loading data in Big-Endian byte order.

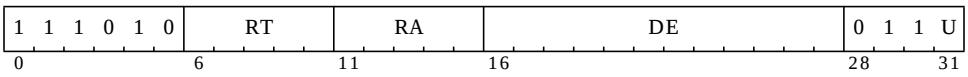
Programming Note
In some implementations, the *Load Word Byte-Reverse* instructions may have greater latency than other *Load* instructions.

Load Word and Zero [with Update] [Indexed] [Extended]

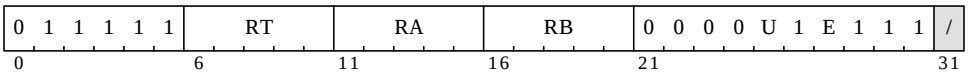
lwz RT,D(RA) . . . . . .(D-mode, U=0)
lwzu RT,D(RA) . . . . . .(D-mode, U=1)



lwze RT,DE(RA). . . . . .(DE-mode, U=0)
lwzue RT,DE(RA). . . . . .(DE-mode, U=1)



lwzx RT,RA,RB . . . . . .(X-mode, E=0, U=0)
lwzux RT,RA,RB . . . . . .(X-mode, E=0, U=1)
lwzxe RT,RA,RB . . . . . .(XE-mode, E=1, U=0)
lwzuxe RT,RA,RB . . . . . .(XE-mode, E=1, U=1)




```
if RA=0 then a ← 64 0 else a ← GPR(RA)
if D-mode  then EA ← 32 0 || (a + EXTS(D))32:63
if X-mode  then EA ← 32 0 || (a + GPR(RB))32:63
if DE-mode then EA ← a + EXTS(DE)
if XE-mode then EA ← a + GPR(RB)
GPR(RT) ← 32 0 || MEM(EA,4)
if U=1 then GPR(RA) ← EA
```

Let the effective address (EA) be calculated as follows:

- For ***lwz*** and ***lwzu***, let EA be 32 0s concatenated with bits 32:63 of the sum of the contents of GPR(RA), or 64 0s if RA=0, and the sign-extended value of the D instruction field.

- For ***lwzx*** and ***lwzux***, let EA be 32 0s concatenated with bits 32:63 of the sum of the contents of GPR(RA), or 64 0s if RA=0, and the contents of GPR(RB).

- For ***lwze*** and ***lwzue***, let EA be the sum of the contents of GPR(RA), or 64 0s if RA=0, and the sign-extended value of the DE instruction field.

- For ***lwzxe*** and ***lwzuxe***, let EA be the sum of the contents of GPR(RA), or 64 0s if RA=0, and the contents of GPR(RB).

The word in storage addressed by the EA is loaded into bits 32:63 of GPR(RT). Bits 0:31 of GPR(RT) are set to 0.

If U=1 (‘with update’), EA is placed into GPR(RA).

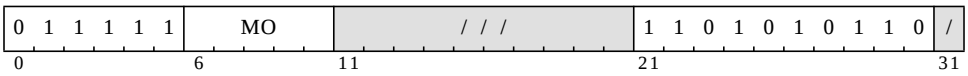
If U=1 (‘with update’), and RA=0 or RA=RT, the instruction form is invalid.

Special Registers Altered:

None

Memory Barrier

mbar MO



When MO=0, the ***mbar*** instruction provides a storage ordering function for all storage access instructions executed by the processor executing the ***mbar*** instruction. Executing an ***mbar*** instruction ensures that all data storage accesses caused by instructions preceding the ***mbar*** instruction have completed before any data storage accesses caused by any instructions after the ***mbar*** instruction. This order is seen by all mechanisms.

When MO≠0, an implementation may support the ***mbar*** instruction ordering a particular subset of storage accesses. An implementation may also support multiple, non-zero values of MO that each specify a different subset of storage accesses that are ordered by the ***mbar*** instruction. Which subsets of storage accesses that are ordered and which values of MO that specify these subsets is implementation-dependent. See the User's Manual for the implementation.

Special Registers Altered:

None

Programming Note

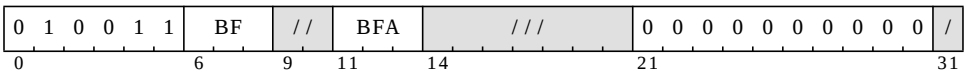
mbar is provided to implement a pipelined storage barrier. The following sequence illustrates one use of ***mbar*** in supporting shared data, ensuring the action is completed prior to releasing the lock.

P1	**P2**
lock	. . .
read & write	. . .
mbar	. . .
free lock	. . .
. . .	lock
. . .	read & write
. . .	mbar
. . .	free lock

See also Appendix D on page 397.

Move Condition Register Field

mcrf BF,BFA



$CR_{4xBF+32:4xBF+35} \leftarrow CR_{4xBFA+32:4xBFA+35}$

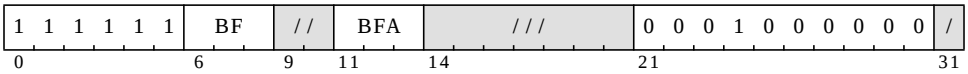
The contents of field BFA (bits 4×BFA+32:4×BFA+35) of the Condition Register are copied to field BF (bits 4×BF+32:4×BF+35) of the Condition Register.

Special Registers Altered:

CR

Move to Condition Register from Floating-Point Status and Control Register

mcrfs BF,BFA



$CR_{BF\times4:BF\times4+3} \leftarrow FPSCR_{BFA\times4:BFA\times4+3}$
$FPSCR_{BFA\times4:BFA\times4+3} \leftarrow 0b0000$

The contents of Floating-Point Status and Control Register field BFA are copied to Condition Register field BF. All exception bits copied are set to 0 in the Floating-Point Status and Control Register. If the FX bit is copied, it is set to 0 in the Floating-Point Status and Control Register.

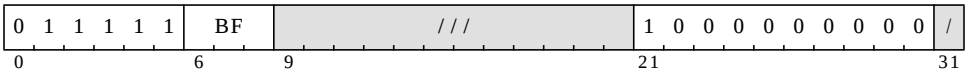
An attempt to execute ***mcrfs*** while MSR_{FP}=0 will cause a Floating-Point Unavailable interrupt.

Special Registers Altered:

CR field BF
FX OX . . . . . (if BFA=0)
UX ZX XX VXSNAN. . . . . (if BFA=1)
VXISI VXIDI VXZDZ VXIMZ. . . . . (if BFA=2)
VXVC . . . . . (if BFA=3)
VXSOFT VXSQRT VXCVI. . . . . (if BFA=5)

Move to Condition Register from Integer Exception Register

mcrxr BF



$CR_{4\times BF+32:4\times BF+35} \leftarrow XER_{32:35}$
$XER_{32:35} \leftarrow 0b0000$

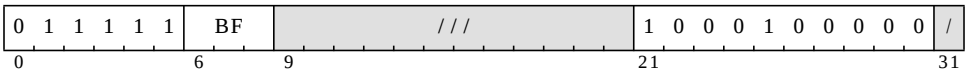
The contents of bits 32:35 of the Integer Exception Register are copied to Condition Register field BF. Bits 32:35 of the Integer Exception Register are set to zero.

Special Registers Altered:

CR $XER_{32:35}$

Move to Condition Register from Integer Exception Register 64

mcrxr64 BF



$CR_{4\times BF+32:4\times BF+35} \leftarrow XER_{0:3}$
$XER_{0:3} \leftarrow 0b0000$

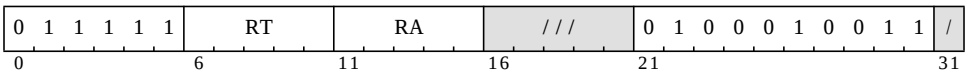
The contents of bits 0:3 of the Integer Exception Register are copied to Condition Register field BF. Bits 0:3 of the Integer Exception Register are set to zero.

Special Registers Altered:

CR $XER_{0:3}$

Move From APID Indirect

mfapidi RT,RA




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

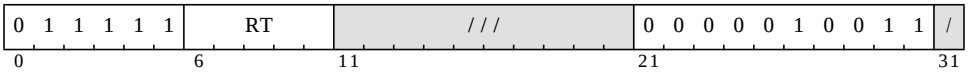
The contents of GPR(RA) are provided to any auxiliary processing extensions that may be present. A value, that is implementation-dependent and extension-dependent, is placed in GPR(RT).

Special Registers Altered:
None

Programming Note
This instruction is provided as a mechanism for software to query the presence and configuration of one or more auxiliary processing extensions. See User's Manual for the implementation for details on the behavior of this instruction.

Move From Condition Register

mfcr RT



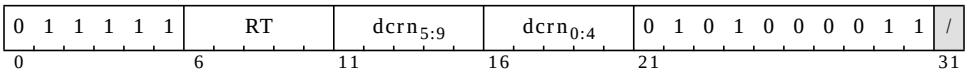

```
GPR(RT) ← 32 0 || CR
```

The contents of the Condition Register are placed into bits 32:63 of GPR(RT). Bits 0:31 of GPR(RT) are set to 0.

Special Registers Altered:
None

Move From Device Control Register

mfdcr RT,DCRN




```
DCRN    ← dcrn0:4 || dcrn5:9
GPR(RT) ← DCREG(DCRN)
```

Let DCRN denote a Device Control Register (see User's Manual for a list of the Device Control Registers supported by the implemention).

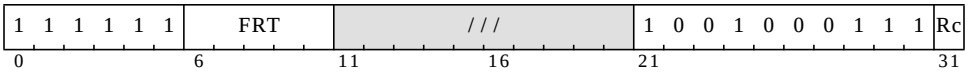
The contents of the designated Device Control Register are placed into GPR(RT). For 32-bit Device Control Registers, the contents of the Device Control Register are placed into bits 32:63 of GPR(RT). Bits 0:31 of GPR(RT) are set to 0.

Execution of this instruction is privileged and restricted to supervisor mode only.

Special Registers Altered:
None

Move From Floating-Point Status and Control Register

mffs FRT. . . . . (Rc=0)
mffs. FRT. . . . . (Rc=1)




```
FPR(FRT) ← FPSCR
```

The contents of the Floating-Point Status and Control Register are placed into bits 32:63 of FPR(FRT). Bits 0:31 of FPR(FRT) are undefined.

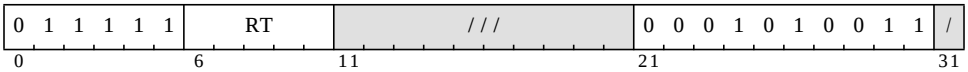
An attempt to execute ***mffs***[**.**] while MSR_{FP}=0 will cause a Floating-Point Unavailable interrupt.

Special Registers Altered:

CR1 ← FX || FEX || VX || OX . . . . . .(if Rc=1)

Move From Machine State Register

mfmsr RT




```
GPR(RT) ← 32 0 || MSR
```

The contents of the MSR are placed into bits 32:63 of GPR(RT). Bits 0:31 of GPR(RT) are set to 0.

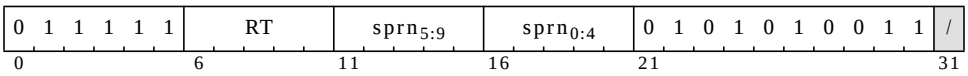
Execution of this instruction is privileged and restricted to supervisor mode only.

Special Registers Altered:

None

Move From Special Purpose Register

mfspr RT,SPRN




```
SPRN     ← sprn0:4 || sprn5:9
GPR(RT) ← SPREG(SPRN)
```

Let SPRN denote a Special Purpose Register (see Section B.1 for a list of Special Purpose Registers defined by Book E, Section B.3 for a list of SPRN values reserved by Book E, Section B.4 for a list of SPRN values allocated by Book E, and Section B.2 for a list of Special Purpose Registers preserved by Book E).

The contents of the designated Special Purpose Register are placed into GPR(RT). For 32-bit Special Purpose Registers, the contents of the Special Purpose Register are placed into bits 32:63 of GPR(RT). Bits 0:31 of GPR(RT) are set to 0.

$SPRN_5$	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

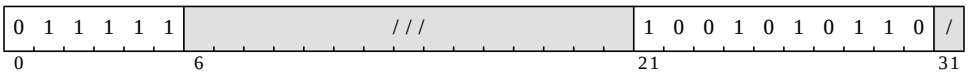
Execution of this instruction specifying a defined and privileged Special Purpose Register ($SPRN_5$=1) when MSR_{PR}=1 will result in a Privileged Instruction exception type Program interrupt.

Special Registers Altered:

None

Memory Synchronize

msync



The ***msync*** instruction provides an ordering function for the effects of all instructions executed by the processor executing the ***msync*** instruction. Executing a ***msync*** instruction ensures that all instructions preceding the ***msync*** instruction have completed before the ***msync*** instruction completes, and that no subsequent instructions are initiated until after the ***msync*** instruction completes. It also creates a memory barrier (see Section 6.1.6.1 on page 114), which orders the storage accesses associated with these instructions.

The ***msync*** instruction may not complete before storage accesses associated with instructions preceding the ***msync*** instruction have been performed.

This instruction is execution synchronizing (see Section 1.12.2 on page 38).

Special Registers Altered:

None

Programming Note
The ***msync*** instruction can be used to ensure that all stores into a data structure, caused by *Store* instructions executed in a 'critical section' of a program, will be performed with respect to another processor before the store that releases the lock is performed with respect to that processor.

The functions performed by the ***msync*** instruction may take a significant amount of time to complete, so indiscriminate use of this instruction may adversely affect performance. The *Memory Barrier* (***mbar***) instruction on page 304 may be more appropriate than ***msync*** for many cases.

Engineering Note
Unlike a context synchronizing operation, ***msync*** need not discard prefetched instructions.

Programming Note
msync replaces the PowerPC ***sync*** instruction. ***msync*** uses the same opcode as ***sync*** such that PowerPC applications calling for a ***sync*** instruction will get the Book E ***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 storage accesses complete. ***mbar*** is provided in the Book E for those occasions when only ordering of storage accesses is required without execution synchronization.

See also Appendix D on page 397.

Move To Condition Register Fields

mtcrf FXM,RS


```
i ← 0
do while i < 8
   if FXM_i=1 then CR_{4×i+32:4×i+35} ← GPR(RS)_{4×i+32:4×i+35}
   i ← i+1
```

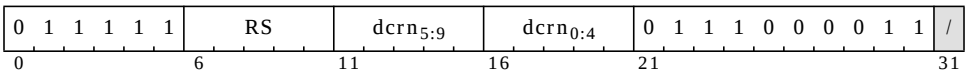
The contents of bits 32:63 of GPR(RS) are placed into the Condition Register under control of the field mask specified by FXM. The field mask identifies the 4-bit fields affected. Let i be an integer in the range 0-7. If FXM_i = 1 then CR field i (CR bits 4×i+32 through 4×i+35) is set to the contents of the corresponding field of bits 32:63 of GPR(RS).

Special Registers Altered:

CR fields selected by mask

Move To Device Control Register

mtdcr DCRN,RS




```
DCRN         ← dcrn_{0:4} || dcrn_{5:9}
DCREG(DCRN) ← GPR(RS)
```

Let DCRN denote a Device Control Register (see User's Manual for a list of the Device Control Registers supported by the implemention).

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

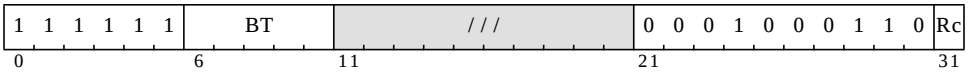
Execution of this instruction is privileged and restricted to supervisor mode only.

Special Registers Altered:

See the User's Manual for the implementation

Move To Floating-Point Status and Control Register Bit 0

mtfsb0 BT. . . . . (Rc=0)
mtfsb0. BT. . . . . (Rc=1)



$FPSCR_{BT+32}$ ← 0b0

Bit BT+32 of the Floating-Point Status and Control Register is set to 0.

An attempt to execute ***mtfsb0***[**.**] while MSR_{FP}=0 will cause a Floating-Point Unavailable interrupt.

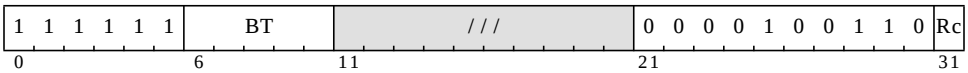
Special Registers Altered:

FPSCR bit BT
CR1 ← FX || FEX || VX || OX . . . . . .(if Rc=1)

Programming Note
Bits 33 and 34 (FEX and VX) cannot be explicitly reset.

Move To Floating-Point Status and Control Register Bit 1

mtfsb1 BT. . . . . (Rc=0)
mtfsb1. BT. . . . . (Rc=1)



$FPSCR_{BT+32}$ ← 0b1

Bit BT+32 of the Floating-Point Status and Control Register is set to 1.

An attempt to execute ***mtfsb1***[**.**] while MSR_{FP}=0 will cause a Floating-Point Unavailable interrupt.

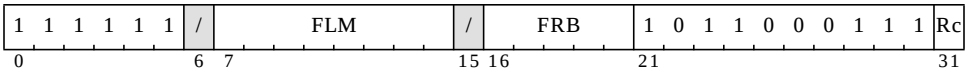
Special Registers Altered:

FPSCR bits BT and FX
CR1 ← FX || FEX || VX || OX . . . . . .(if Rc=1)

Programming Note
Bits 33 and 34 (FEX and VX) cannot be explicitly set.

Move To Floating-Point Status and Control Register Fields

mtfsf FLM,FRB . . . . . (Rc=0)
mtfsf. FLM,FRB . . . . . (Rc=1)




```
i ← 0
do while i<8
  if FLMi=1 then FPSCR4×i+32:4×i+35 ← FPR(FRB)4×i+32:4×i+35
  i ← i+1
```

The contents of bits 32:63 of FPR(FRB) are placed into the Floating-Point Status and Control Register under control of the field mask specified by FLM. The field mask identifies the 4-bit fields affected. Let i be an integer in the range 0-7. If FLM_i=1 then Floating-Point Status and Control Register field i (FPSCR bits 4×i+32 through 4×i+35) is set to the contents of the corresponding field of the low-order 32 bits of FPR(FRB).

$FPSCR_{FX}$ is altered only if FLM_0 = 1.

An attempt to execute ***mtfsf*[.]** while MSR_{FP}=0 will cause a Floating-Point Unavailable interrupt.

Special Registers Altered:

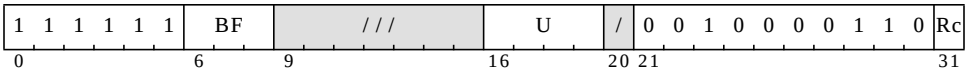
FPSCR fields selected by mask
CR1 ← FX || FEX || VX || OX . . . . . .(if Rc=1)

Programming Note
Updating fewer than all eight fields of the Floating-Point Status and Control Register may have substantially poorer performance on some implementations than updating all the fields.

Programming Note
When $FPSCR_{32:35}$ is specified, bits 32 (FX) and 35 (OX) are set to the values of $(FRB)_{32}$ and $(FRB)_{35}$ (i.e., even if this instruction causes OX to change from 0 to 1, FX is set from $(FRB)_{32}$ and not by the usual rule that FX is set to 1 when an exception bit changes from 0 to 1). Bits 33 and 34 (FEX and VX) are set according to the usual rule, given on page 70, and not from $(FRB)_{33:34}$.

Move To Floating-Point Status and Control Register Field Immediate

mtfsfi BF,U . . . . . (Rc=0)
mtfsfi. BF,U . . . . . (Rc=1)



$FPSCR_{BF\times4+32:BF\times4+35} \leftarrow U$

The value of the U field is placed into Floating-Point Status and Control Register field BF.

$FPSCR_{FX}$ is altered only if BF = 0.

An attempt to execute ***mtfsfi***[**.**] while MSR_{FP}=0 will cause a Floating-Point Unavailable interrupt.

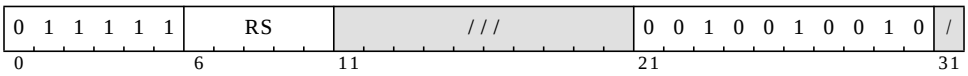
Special Registers Altered:

FPSCR field BF
CR1 ← FX || FEX || VX || OX . . . . . .(if Rc=1)

Programming Note
When $FPSCR_{32:35}$ is specified, bits 32 (FX) and 35 (OX) are set to the values of U_0 and U_3 (i.e., even if this instruction causes OX to change from 0 to 1, FX is set from U_0 and not by the usual rule that FX is set to 1 when an exception bit changes from 0 to 1). Bits 33 and 34 (FEX and VX) are set according to the usual rule, given on page 70, and not from $U_{1:2}$.

Move To Machine State Register

mtmsr RS



$MSR \leftarrow GPR(RS)_{32:63}$

The contents of bits 32:63 of GPR(RS) are placed into the MSR.

Execution of this instruction is privileged and restricted to supervisor mode only.

Execution of this instruction is execution synchronizing.

In addition, alterations to the EE or CE bits are effective as soon as the instruction completes. Thus if MSR_{EE}=0 and an External interrupt is pending, executing an ***mtmsr*** that sets MSR_{EE} to 1 will cause the External interrupt to be taken before the next instruction is executed, if no higher priority exception exists. Likewise, if MSR_{CE}=0 and a Critical Input interrupt is pending, executing an ***mtmsr*** that sets MSR_{CE} to 1 will cause the Critical Input interrupt to be taken before the next instruction is executed if no higher priority exception exists. (See Section 7.9 on page 178).

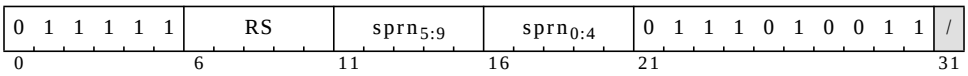
Special Registers Altered:

MSR

Programming Note
For a discussion of software synchronization requirements when altering certain MSR bits please refer to Chapter 11, “Synchronization Requirements”, on page 225.

Move To Special Purpose Register

mtspr SPRN,RS




```
SPRN        ← sprn0:4 || sprn5:9
SPREG(SPRN) ← GPR(RS)
```

Let SPRN denote a Special Purpose Register (see Section B.1 for a list of Special Purpose Registers defined by Book E, Section B.3 for a list of SPRN values reserved by Book E, Section B.4 for a list of SPRN values allocated by Book E, Section B.2 for a list of SPRN values preserved by Book E, and the User's Manual of the implementation for a list of all Special Purpose Registers that are implemented).

The contents of GPR(RS) are placed into the designated Special Purpose Register. For 32-bit Special Purpose Registers, the contents of bits 32:63 of GPR(RS) are placed into the Special Purpose Register.

When MSR_{PR}=1, specifying a Special Purpose Register 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 a Special Purpose Register that is privileged ($SPRN_5$=1) results in a Privileged Instruction exception type Program interrupt. When MSR_{PR}=0, specifying a Special Purpose Register that is not implemented is boundedly undefined.

Special Registers Altered:

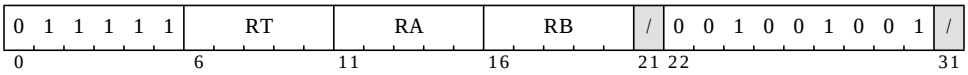
See Table B-1 on page 376 or the User's Manual for the implementation

Programming Note

For a discussion of software synchronization requirements when altering certain Special Purpose Registers, please refer to Chapter 11, "Synchronization Requirements", on page 225.

Multiply High Doubleword

mulhd RT,RA,RB



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

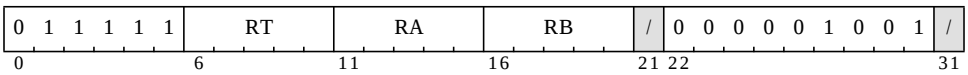
Bits 0:63 of the 128-bit product of the contents of GPR(RA) and the contents of GPR(RB) are placed into GPR(RT).

Both operands and the product are interpreted as signed integers.

Special Registers Altered:
None

Multiply High Doubleword Unsigned

mulhdu RT,RA,RB



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

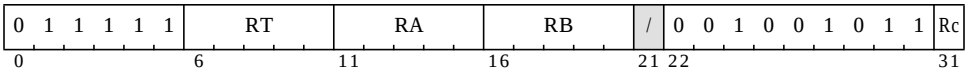
Bits 0:63 of the 128-bit product the contents of GPR(RA) and the contents of GPR(RB) are placed into GPR(RT).

Both operands and the product are interpreted as unsigned integers.

Special Registers Altered:
None

Multiply High Word

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




```
prod0:63  ← GPR(RA)32:63 × GPR(RB)32:63
if Rc=1 then do
   LT  ← prod0:31 < 0
   GT  ← prod0:31 > 0
   EQ  ← prod0:31 = 0
   CR0 ← LT || GT || EQ || SO
GPR(RT)32:63 ← prod0:31
GPR(RT)0:31  ← undefined
```

Bits 0:31 of the 64-bit product of the contents of bits 32:63 of GPR(RA) and the contents of bits 32:63 of GPR(RB) are placed into bits 32:63 of GPR(RT). Bits 0:31 of GPR(RT) are undefined.

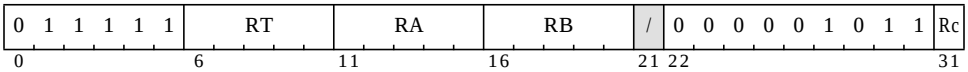
Both operands and the product are interpreted as signed integers.

Special Registers Altered:

CR0 . . . . . .(if Rc=1)

Multiply High Word Unsigned

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




```
prod0:63 ← GPR(RA)32:63 × GPR(RB)32:63
if Rc=1 then do
   LT  ← prod0:31 < 0
   GT  ← prod0:31 > 0
   EQ  ← prod0:31 = 0
   CR0 ← LT || GT || EQ || SO
GPR(RT)32:63 ← prod0:31
GPR(RT)0:31 ← undefined
```

Bits 0:31 of the 64-bit product the contents of bits 32:63 of GPR(RA) and the contents of bits 32:63 of GPR(RB) are placed into bits 32:63 of GPR(RT). Bits 0:31 of GPR(RT) are undefined.

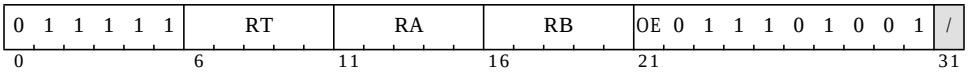
Both operands and the product are interpreted as unsigned integers, except that if Rc=1 the first three bits of CR Field 0 are set by signed comparison of the result to zero.

Special Registers Altered:

CR0 . . . . . .(if Rc=1)

Multiply Low Doubleword

mulld RT,RA,RB . . . . . .(OE=0)
mulldo RT,RA,RB . . . . . .(OE=1)




```
prod0:127 ← GPR(RA) × GPR(RB)
if OE=1 then do
   OV64 ← (prod0:64 ≠ 65 0) & (prod0:64 ≠ 65 1)
   SO64 ← SO64 | OV64
GPR(RT) ← prod64:127
```

Bits 64:127 of the 128-bit product of the contents of GPR(RA) and the contents of GPR(RB) are placed into GPR(RT).

If OE=1 then OV64 is set to 1 if the product cannot be represented in 64 bits.

Both operands and the product are interpreted as signed integers.

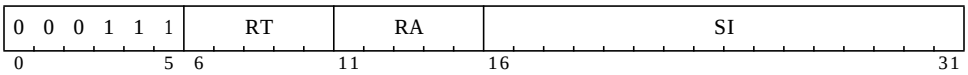
Special Registers Altered:

SO64 OV64 . . . . . (if OE=1)

> **Programming Note**
> The *Multiply* instructions that set the XER may execute faster on some implementations if GPR(RB) contains the operand having the smaller absolute value.

Multiply Low Immediate

mulli RT,RA,SI




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

Bits 64:127 of the 128-bit product of the contents of GPR(RA) and the sign-extended value of the SI field are placed into GPR(RT).

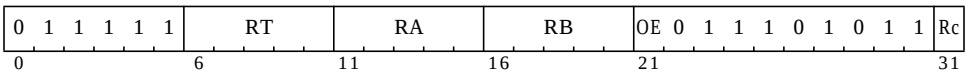
Both operands and the product are interpreted as signed integers.

Special Registers Altered:

None

Multiply Low Word

mullw RT,RA,RB . . . . . .(OE=0, Rc=0)
mullw. RT,RA,RB . . . . . .(OE=0, Rc=1)
mullwo RT,RA,RB . . . . . .(OE=1, Rc=0)
mullwo. RT,RA,RB . . . . . .(OE=1, Rc=1)




```
prod0:63 ← GPR(RA)32:63 × GPR(RB)32:63
if OE=1 then do
   OV ← (prod0:32 ≠ 330) & (prod0:32 ≠ 331)
   SO ← SO | OV
if Rc=1 then do
   LT  ← prod32:63 < 0
   GT  ← prod32:63 > 0
   EQ  ← prod32:63 = 0
   CR0 ← LT || GT || EQ || SO
GPR(RT) ← prod0:63
```

The 64-bit product of the contents of bits 32:63 of GPR(RA) and the contents of bits 32:63 of GPR(RB) is placed into GPR(RT).

If OE=1 then OV is set to 1 if the product cannot be represented in 32 bits.

Both operands and the product are interpreted as signed integers.

Special Registers Altered:

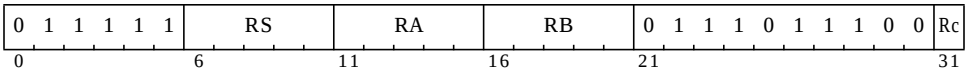
CR0 . . . . . .(if Rc=1)
SO OV. . . . . (if OE=1)

Programming Note
For ***mulli*** and ***mulld***, the low-order 64 bits of the product are independent of whether the operands are regarded as signed or unsigned 64-bit integers.

For ***mulli*** and ***mullw***, bits 32:63 of the product are independent of whether the operands are regarded as signed or unsigned 32-bit integers.

NAND

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




```
result0:63 ← ¬(GPR(RS) & GPR(RB))
if Rc=1 then do
   LT  ← result32:63 < 0
   GT  ← result32:63 > 0
   EQ  ← result32:63 = 0
   CR0 ← LT || GT || EQ || SO
GPR(RA) ← result
```

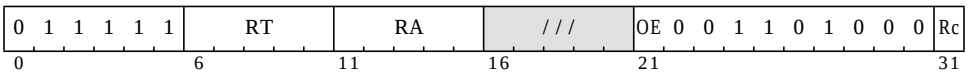
The contents of GPR(RS) are ANDed with the contents of GPR(RB) and the one's complement of the result is placed into GPR(RA).

Special Registers Altered:

CR0 . . . . . .(if Rc=1)

Negate

neg RT,RA . . . . . .(OE=0, Rc=0)
neg. RT,RA . . . . . .(OE=0, Rc=1)
nego RT,RA . . . . . .(OE=1, Rc=0)
nego. RT,RA . . . . . .(OE=1, Rc=1)




```
carry0:63 ← Carry(¬GPR(RA) + 1)
sum0:63   ←       ¬GPR(RA) + 1
if OE=1 then do
   OV   ← carry32 ⊕ carry33
   SO   ← SO | (carry32 ⊕ carry33)
   OV64 ← carry0 ⊕ carry1
   SO64 ← SO64 | (carry0 ⊕ carry1)
if Rc=1 then do
   LT  ← sum32:63 < 0
   GT  ← sum32:63 > 0
   EQ  ← sum32:63 = 0
   CR0 ← LT || GT || EQ || SO
GPR(RT) ← sum
```

The sum of the one's complement of the contents of GPR(RA) and 1 is placed into GPR(RT).

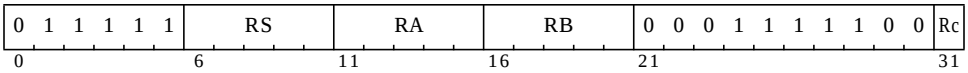
If GPR(RA) contains the most negative 64-bit number (0x8000_0000_0000_0000), the result is the most negative number and, if OE=1, OV64 is set to 1. Similarly, if bits 32:63 of GPR(RA) contain the most negative 32-bit number (0x8000_0000), bits 32:63 of the result contain the most negative 32-bit number and, if OE=1, OV is set to 1.

Special Registers Altered:

CR0 . . . . . .(if Rc=1)
SO OV SO64 OV64 . . . . . (if OE=1)

NOR

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




```
result0:63 ← ¬(GPR(RS) | GPR(RB))
if Rc=1 then do
   LT  ← result32:63 < 0
   GT  ← result32:63 > 0
   EQ  ← result32:63 = 0
   CR0 ← LT || GT || EQ || SO
GPR(RA) ← result
```

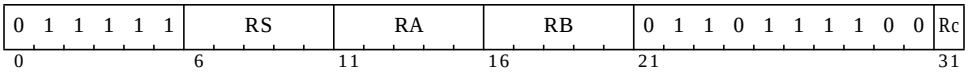
The contents of GPR(RS) are ORed with the contents of GPR(RB) and the one's complement of the result is placed into GPR(RA).

Special Registers Altered:

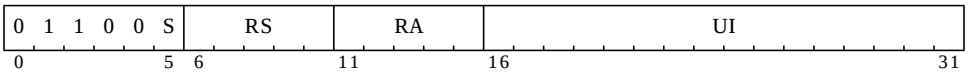
CR0 . . . . . .(if Rc=1)

OR [Immediate [Shifted] | with Complement]

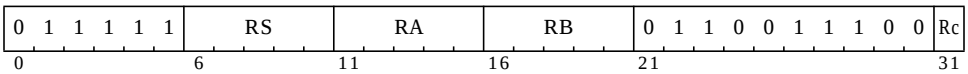
or RA,RS,RB . . . . . (Rc=0)
or. RA,RS,RB . . . . . (Rc=1)



ori RA,RS,UI. . . . . (S=0, Rc=0)
oris RA,RS,UI. . . . . (S=1, Rc=0)



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




```
if 'ori'    then b ← 48 0 || UI
if 'oris'   then b ← 32 0 || UI || 16 0
if 'or[.]'  then b ← GPR(RB)
if 'orc[.]' then b ← ¬GPR(RB)
result0:63 ← GPR(RS) | b
if Rc=1 then do
   LT  ← result32:63 < 0
   GT  ← result32:63 > 0
   EQ  ← result32:63 = 0
   CR0 ← LT || GT || EQ || SO
GPR(RA) ← result
```

For ***ori***, the contents of GPR(RS) are ORed with $^{48}0$ || UI.

For ***oris***, the contents of GPR(RS) are ORed with $^{32}0$ || UI || $^{16}0$.

For ***or*[.]**, the contents of GPR(RS) are ORed with the contents of GPR(RB).

For ***orc*[.]**, the contents of GPR(RS) are ORed with the one's complement of the contents of GPR(RB).

The result is placed into GPR(RA).

The preferred 'no-op' (an instruction that does nothing) is:

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

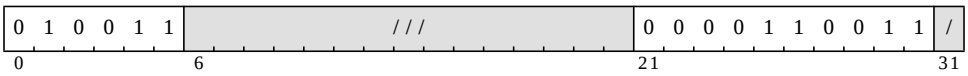
Special Registers Altered:

CR0 . . . . . .(if Rc=1)

Engineering Note
It is desirable for implementations to make the preferred form of no-op execute quickly, since this form should be used by compilers.

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 Critical Save/Restore Register 1 are placed into the Machine State Register. If the new Machine State Register value does not enable any pending exceptions, then the next instruction is fetched, under control of the new Machine State Register value, from the address $CSRR0_{0:61}$||0b00. If the new Machine State Register 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 Save/Restore Register 0 or Critical Save/Restore Register 0 by the interrupt processing mechanism (see Section 7.5 on page 151) is the address of the instruction that would have been executed next had the interrupt not occurred (i.e. the address in Critical Save/Restore Register 0 at the time of the execution of the ***rfci***).

Execution of this instruction is privileged and restricted to supervisor mode only.

Execution of this instruction is context synchronizing.

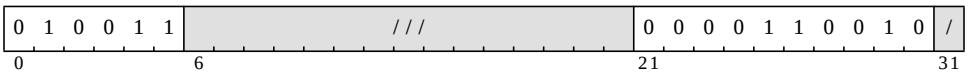
Special Registers Altered:

MSR

Programming Note
In addition to *Branch to Link Register* (***bclr***[***e***][***l***]) and *Branch to Count Register* (***bcctr***[***e***][***l***]) instructions, ***rfi*** and ***rfci*** allow software to branch to any valid 64-bit address by using the respective 64-bit Save/Restore Register 0 and Critical Save/Restore Register 0.

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 Save/Restore Register 1 are placed into the Machine State Register. If the new Machine State Register value does not enable any pending exceptions, then the next instruction is fetched, under control of the new Machine State Register value, from the address $SRR0_{0:61}$||0b00. If the new Machine State Register 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 Save/Restore Register 0 or Critical Save/Restore Register 0 by the interrupt processing mechanism (see Section 7.5 on page 151) is the address of the instruction that would have been executed next had the interrupt not occurred (i.e. the address in Save/Restore Register 0 at the time of the execution of the ***rfi***).

Execution of this instruction is privileged and restricted to supervisor mode only.

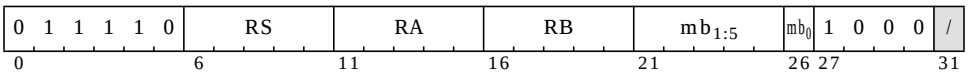
Execution of this instruction is context synchronizing.

Special Registers Altered:

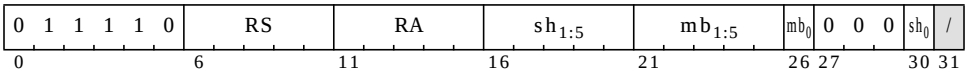
MSR

Rotate Left Doubleword [Immediate] then Clear Left

rldcl RA,RS,RB,mb



rldicl RA,RS,sh,mb




```
if 'rldcl' then n ← GPR(RB)58:63
else            n ← sh0 || sh1:5
b ← mb0 || mb1:5
r ← ROTL64(GPR(RS),n)
m ← MASK(b,63)
GPR(RA) ← r & m
```

If ***rldcl***, let the shift count *n* be the contents of bits 58:63 of GPR(RB).

If ***rldicl***, let the shift count *n* be the value *sh*.

The contents of GPR(RS) are rotated$_{64}$ left *n* bits. A mask is generated having '1' bits from bit *mb* through bit 63 and '0' bits elsewhere. The rotated data are ANDed with the generated mask and the result is placed into GPR(RA).

Special Registers Altered:

None

Programming Note

Uses for ***rldcl*[.]**:

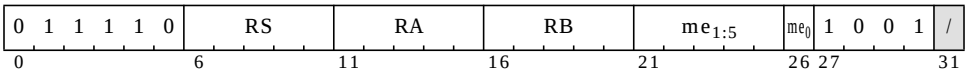
- Can be used to extract a *k*-bit field that starts at variable bit position *j* in GPR(RS), right-justified into GPR(RA) (clearing the remaining 64-*k* bits of GPR(RA)), by setting GPR(RB)$_{58:63}$=*j*+*k* and *mb*=64–*k*.
- Can be used to rotate the contents of a register left by variable *k* bits, by setting GPR(RB)$_{58:63}$=*k* and *mb*=0.
- Can be used to rotate the contents of a register right by variable *k* bits, by setting GPR(RB)$_{58:63}$=64–*k* and *mb*=0.

Uses for ***rldicl***:

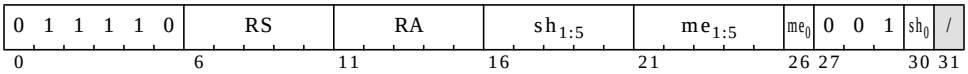
- Can be used to extract a *k*-bit field that starts at bit position *j* in GPR(RS), right-justified into GPR(RA) (clearing the remaining 64-*k* bits of GPR(RA)), by setting *sh*=*j*+*k* and *mb*=64-*k*.
- Can be used to rotate the contents of a register left by *k* bits, by setting *sh*=*k* and *mb*=0.
- Can be used to rotate the contents of a register right by *k* bits, by setting *sh*=64–*k* and *mb*=0.
- Can be used to shift the contents of a register right by *k* bits, by setting *sh*=64-*k* and *mb*=*k*.
- Can be used to clear the high-order *k* bits of a register, by setting *sh*=0 and *mb*=*k*.

Rotate Left Doubleword [Immediate] then Clear Right

rldcr RA,RS,RB,me



rldicr RA,RS,sh,me




```
if 'rldcr' then n ← GPR(RB)58:63
else            n ← sh0 || sh1:5
e ← me0 || me1:5
r ← ROTL64(GPR(RS),n)
m ← MASK(0,e)
GPR(RA) ← r & m
```

If ***rldcr***, let the shift count *n* be the contents of bits 58:63 of GPR(RB).

If ***rldicr***, let the shift count *n* be the value *sh*.

The contents of GPR(RS) are rotated$_{64}$ left *n* bits. A mask is generated having '1' bits from bit 0 through bit *me* and '0' bits elsewhere. The rotated data are ANDed with the generated mask and the result is placed into GPR(RA).

Special Registers Altered:

None

Programming Note

Uses for ***rldcr***:

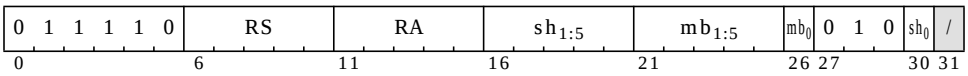
- Can be used to extract a *k*-bit field that starts at variable bit position *j* in GPR(RS), left-justified into GPR(RA) (clearing the remaining 64-*k* bits of GPR(RA)), by setting GPR(RB)$_{58:63}$=*j* and *me*=*k*-1.
- Can be used to rotate the contents of a register left by variable *k* bits, by setting GPR(RB)$_{58:63}$=*k* and *me*=63.
- Can be used to rotate the contents of a register right by variable *k* bits, by setting GPR(RB)$_{58:63}$=64-*k* and *me*=63.

Uses for ***rldicr***:

- Can be used to extract a *k*-bit field that starts at bit position *j* in GPR(RS), left-justified into GPR(RA) (clearing the remaining 64-*k* bits of GPR(RA)), by setting *sh*=*j* and *me*=*k*-1.
- Can be used to rotate the contents of a register left by *k* bits, by setting *sh*=*k* and *me*=63.
- Can be used to rotate the contents of a register right by *k* bits, by setting *sh*=64–*k* and *me*=63.
- Can be used to shift the contents of a register left by *k* bits, by setting *sh*=*k* and *me*=63-*k*.
- Can be used to clear the low-order *k* bits of a register, by setting *sh*=0 and *me*=63-*k*.

Rotate Left Doubleword Immediate then Clear

rldic RA,RS,sh,mb




```
n ← sh0 || sh1:5
b ← mb0 || mb1:5
r ← ROTL64(GPR(RS),n)
m ← MASK(b,¬n)
GPR(RA) ← r & m
```

Let the shift count *n* be the value *sh*.

The contents of GPR(RS) are rotated$_{64}$ left *n* bits. A mask is generated having '1' bits from bit *mb* through bit 63-*sh* and '0' bits elsewhere. The rotated data are ANDed with the generated mask and the result is placed into GPR(RA).

Special Registers Altered:

None

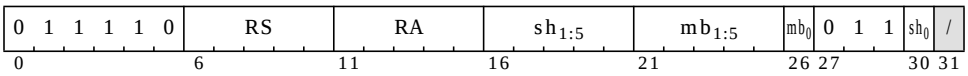
Programming Note

Uses for ***rldic***:

- Can be used to clear the high-order *j* bits of the contents of a register and then shift the result left by *k* bits, by setting *sh*=*k* and *mb*=*j*–*k*.
- Can be used to clear the high-order *k* bits of a register, by setting *sh*=0 and *mb*=*k*.

Rotate Left Doubleword Immediate then Mask Insert

rldimi RA,RS,sh,mb




```
n ← sh0 || sh1:5
b ← mb0 || mb1:5
r ← ROTL64(GPR(RS),n)
m ← MASK(b,¬n)
GPR(RA) ← r&m | GPR(RA)&¬m
```

Let the shift count *n* be the value *sh*.

The contents of GPR(RS) are rotated$_{64}$ left *n* bits. A mask is generated having '1' bits from bit *mb* through bit 63-*sh* and '0' bits elsewhere. The rotated data are inserted into GPR(RA) 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:

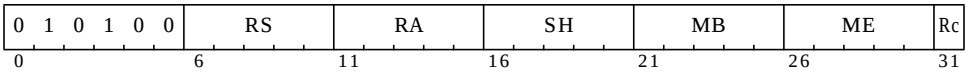
None

Programming Note

rldimi can be used to insert a *k*-bit field that is right-justified in GPR(RS), into GPR(RA) starting at bit position *j*, by setting *sh*=64-(*j*+*k*) and *mb*=*j*.

Rotate Left Word Immediate then Mask Insert

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




```
n ← SH
b ← MB+32
e ← ME+32
r ← ROTL32(GPR(RS)32:63,n)
m ← MASK(b,e)
result0:63 ← r&m | GPR(RA)&¬m
if Rc=1 then do
   LT  ← result32:63 < 0
   GT  ← result32:63 > 0
   EQ  ← result32:63 = 0
   CR0 ← LT || GT || EQ || SO
GPR(RA) ← result0:63
```

Let the shift count *n* be the value SH.

The contents of GPR(RS) 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(RA) 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:

CR0 . . . . . .(if Rc=1)

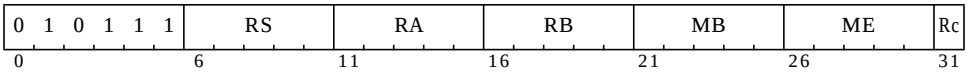
Programming Note

Uses for ***rlwimi*[.]**:

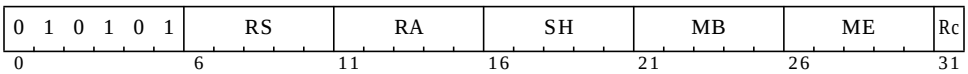
- Can be used to insert a *k*-bit field that is left-justified in bits 32:63 of GPR(RS), into bits 32:63 of GPR(RA) starting at bit position *j*, by setting SH=64-*j*, MB=*j-32*, and ME=(*j*+*k*)-33.
- Can be used to insert an *k*-bit field that is right-justified in bits 32:63 of GPR(RS), into bits 32:63 of GPR(RA) starting at bit position *j*, by setting SH=64-(*j*+*k*), MB=*j-32*, and ME=(*j*+*k*)-33.

Rotate Left Word [Immediate] then AND with Mask

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



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




```
if 'rlwnm[.]' then n ← GPR(RB)59:63
else               n ← SH
b ← MB+32
e ← ME+32
r ← ROTL32(GPR(RS)32:63,n)
m ← MASK(b,e)
result0:63 ← r & m
if Rc=1 then do
   LT  ← result32:63 < 0
   GT  ← result32:63 > 0
   EQ  ← result32:63 = 0
   CR0 ← LT || GT || EQ || SO
GPR(RA) ← result0:63
```

If ***rlwnm***[**.**], let the shift count *n* be the contents of bits 59:63 of GPR(RB).

If ***rlwinm***[**.**], let the shift count *n* be SH.

The contents of GPR(RS) 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(RA).

Special Registers Altered:

CR0 . . . . . .(if Rc=1)

Programming Note

Uses for ***rlwnm*[.]**:

- Can be used to extract a *k*-bit field that starts at variable bit position *j* in bits 32:63 of GPR(RS), right-justified into bits 32:63 of GPR(RA) (clearing the remaining 32–*k* bits of bits 32:63 of GPR(RA)), by setting $GPR(RB)_{59:63}$=*j*+*k*-*32*, MB=32–*k*, and ME=31.
- Can be used to extract a *k*-bit field that starts at variable bit position *j* in bits 32:63 of GPR(RS), left-justified into bits 32:63 of GPR(RA) (clearing the remaining 32–*k* bits of bits 32:63 of GPR(RA)), by setting $GPR(RB)_{59:63}$=*j*-32, MB=0, and ME=*k*–1.
- Can be used to rotate the contents of bits 32:63 of a register left by variable *k* bits, by setting $GPR(RB)_{59:63}$=*k*, MB=0, and ME=31.
- Can be used to rotate the contents of bits 32:63 of a register right by variable *k* bits, by setting $GPR(RB)_{59:63}$=32–*k*, MB=0, and ME=31.

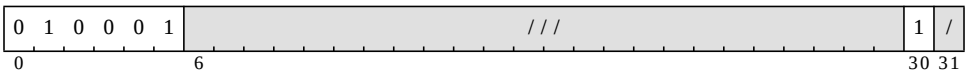
Uses for ***rlwinm*[.]**

- Can be used to extract a *k*-bit field that starts at bit position *j* in bits 32:63 of GPR(RS), right-justified into bits 32:63 of GPR(RA) (clearing the remaining 32–*k* bits of bits 32:63 of GPR(RA)), by setting SH=*j*+*k*-*32*, MB=32–*k*, and ME=31.
- Can be used to extract a *k*-bit field that starts at bit position *j* in bits 32:63 of GPR(RS), left-justified into bits 32:63 of GPR(RA) (clearing the remaining 32–*k* bits of bits 32:63 of GPR(RA)), by setting SH=*j*-32, MB=0, and ME=*k*–1.
- Can be used to rotate the contents of bits 32:63 of a register left by *k* bits, by setting SH=*k*, MB=0, and ME=31.
- Can be used to rotate the contents of bits 32:63 of a register right by *k* bits, by setting SH=32–*k*, MB=0, and ME=31.
- Can be used 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.
- Can be used 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*.
- Can be used 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 all the uses given above, bits 0:31 of GPR(RA) are cleared.

System Call

sc




```
SRR1 ← MSR
SRR0 ← CIA+4
NIA  ← EVPR0:47 || IVOR848:59 || 0b0000
MSRWE,EE,PR,IS,DS,FP,FE0,FE1 ← 0b0000_0000
```

sc is used to request a system service. A System Call interrupt is generated. The contents of the Machine State Register are copied into Save/Restore Register 1 and the address of the instruction after the ***sc*** instruction is placed into Save/Restore Register 0.

$MSR_{WE,EE,PR,IS,DS,FP,FE0,FE1}$ are set to 0.

The interrupt causes the next instruction to be fetched from the address

$IVPR_{0:47}||IVOR8_{48:59}||0b0000$

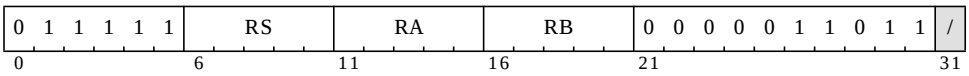
This instruction is context synchronizing.

Special Registers Altered:

SRR0 SRR1 $MSR_{WE,EE,PR,IS,DS,FP,FE0,FE1}$

Shift Left Doubleword

sld RA,RS,RB




```
n ← GPR(RB)58:63
r ← ROTL64(GPR(RS),n)
if GPR(RB)57=0 then m ← MASK(0,63-n)
else                m ← 640
GPR(RA) ← r & m
```

Let the shift count *n* be the value specified by the contents of bits 57:63 of GPR(RB).

The contents of GPR(RS) are shifted left *n* bits. Bits shifted out of position 0 are lost. Zeros are supplied to the vacated positions on the right. The result is placed into GPR(RA).

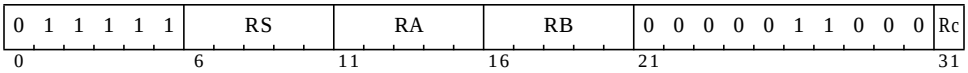
Shift amounts from 64 to 127 give a zero result.

Special Registers Altered:

None

Shift Left Word

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




```
n ← GPR(RB)59:63
r ← ROTL32(GPR(RS)32:63,n)
if GPR(RB)58=0 then m ← MASK(32,63-n)
else                m ← 640
result0:63 ← r & m
if Rc=1 then do
   LT  ← result32:63 < 0
   GT  ← result32:63 > 0
   EQ  ← result32:63 = 0
   CR0 ← LT || GT || EQ || SO
GPR(RA) ← result0:63
```

Let the shift count *n* be the value specified by the contents of bits 58:63 of GPR(RB).

The contents of bits 32:63 of GPR(RS) 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(RA). Bits 0:31 of GPR(RA) are set to zero.

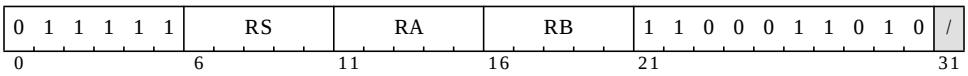
Shift amounts from 32 to 63 give a zero result.

Special Registers Altered:

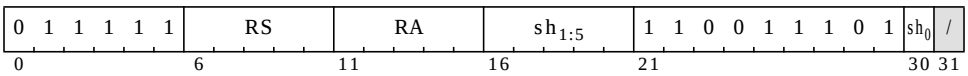
CR0 . . . . . .(if Rc=1)

Shift Right Algebraic Doubleword [Immediate]

srad RA,RS,RB



sradi RA,RS,sh




```
if 'srad' then n ← GPR(RB)58:63
else           n ← sh0 || sh1:5
r ← ROTL64(GPR(RS),64-n)
if 'srad' & GPR(RB)57=1 then m ← 640
else                         m ← MASK(n,63)
s ← GPR(RS)0
GPR(RA) ← r&m | (64s)&¬m
CA64 ← s & ((r&¬m)≠0)
```

If ***srad***, let the shift count *n* be the contents of bits 57:63 of GPR(RB).

If ***sradi***, let the shift count *n* be *sh*.

The contents of GPR(RS) are shifted right *n* bits. Bits shifted out of position 63 are lost. Bit 0 of the contents of GPR(RS) is replicated to fill the vacated positions on the left. The result is placed into GPR(RA).

CA64 is set to 1 if GPR(RS) contains a negative value and any '1' bits are shifted out of bit position 63; otherwise CA is set to 0.

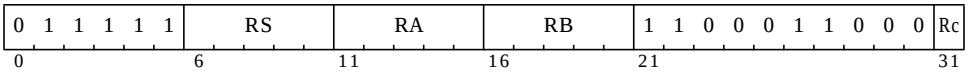
A shift amount of zero causes GPR(RA) to be set equal to the contents of GPR(RS), and CA64 to be set to 0. For ***srad*** shift amounts from 64 to 127 give a result of 64 sign bits in GPR(RA), and cause CA64 to receive bit 0 of the contents of GPR(RS) (i.e. the sign bit of GPR(RS)).

Special Registers Altered:

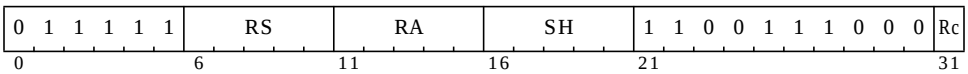
CA64

Shift Right Algebraic Word [Immediate]

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



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




```
if 'sraw[.]' then n ← GPR(RB)59:63
else              n ← SH
r ← ROTL32(GPR(RS)32:63,64-n)
if 'sraw[.]' & GPR(RB)58=1 then m ← 640
else                            m ← MASK(n+32,63)
s ← GPR(RS)32
result0:63 ← r&m | (64s)&¬m
if Rc=1 then do
   LT  ← result32:63 < 0
   GT  ← result32:63 > 0
   EQ  ← result32:63 = 0
   CR0 ← LT || GT || EQ || SO
GPR(RA) ← result0:63
CA      ← s & ((r&¬m)32:63≠0)
```

If ***sraw***[**.**], let the shift count *n* be the contents of bits 58:63 of GPR(RB).

If ***srawi***[**.**], let the shift count *n* be the value of the SH field.

The contents of bits 32:63 of GPR(RS) are shifted right *n* bits. Bits shifted out of position 63 are lost. Bit 32 of RS is replicated to fill the vacated positions on the left. The 32-bit result is placed into bits 32:63 of GPR(RA). Bit 32 of the contents of GPR(RS) is replicated to fill bits 0:31 of GPR(RA).

CA is set to 1 if bits 32:63 of GPR(RS) contain a negative value and any '1' bits are shifted out of bit position 63; otherwise CA is set to 0.

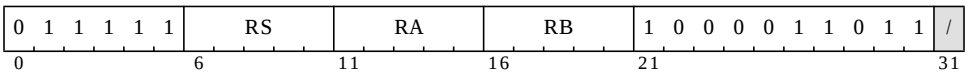
A shift amount of zero causes GPR(RA) to receive EXTS(GPR(RS)$_{32:63}$), and CA to be set to 0. For ***sraw***[**.**] shift amounts from 32 to 63 give a result of 64 sign bits, and cause CA to receive bit 32 of the contents of GPR(RS) (i.e. sign bit of GPR(RS)$_{32:63}$).

Special Registers Altered:

CA
CR0 . . . . . (if Rc=1)

Shift Right Doubleword

srd RA,RS,RB




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

Let the shift count *n* be the value specified by the contents of bits 57:63 of GPR(RB).

The contents of GPR(RS) are shifted right *n* bits. Bits shifted out of position 63 are lost. Zeros are supplied to the vacated positions on the left. The result is placed into GPR(RA).

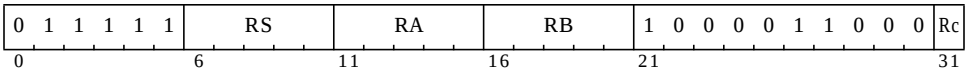
Shift amounts from 64 to 127 give a zero result.

Special Registers Altered:

None

Shift Right Word

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




```
n ← GPR(RB)59:63
r ← ROTL32(GPR(RS)32:63,64-n)
if GPR(RB)58=0 then m ← MASK(n+32,63)
else                m ← 640
result0:63 ← r & m
if Rc=1 then do
   LT  ← result32:63 < 0
   GT  ← result32:63 > 0
   EQ  ← result32:63 = 0
   CR0 ← LT || GT || EQ || SO
GPR(RA) ← result0:63
```

Let the shift count *n* be the value specified by the contents of bits 58:63 of GPR(RB).

The contents of bits 32:63 of GPR(RS) 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(RA). Bits 0:31 of GPR(RA) are set to zero.

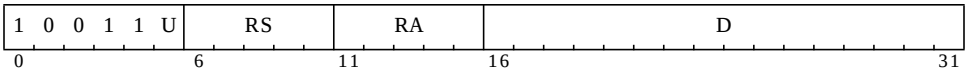
Shift amounts from 32 to 63 give a zero result.

Special Registers Altered:

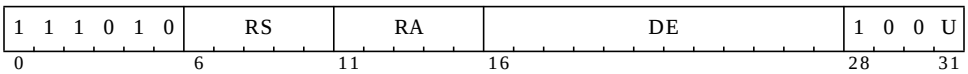
CR0 . . . . . (if Rc=1)

Store Byte [with Update] [Indexed] [Extended]

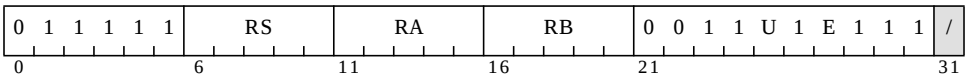
stb RS,D(RA). . . . . .(D-mode, U=0)
stbu RS,D(RA). . . . . .(D-mode, U=1)



stbe RS,DE(RA). . . . . .(DE-mode, U=0)
stbue RS,DE(RA). . . . . .(DE-mode, U=1)



stbx RS,RA,RB . . . . . .(X-mode, E=0, U=0)
stbux RS,RA,RB . . . . . .(X-mode, E=0, U=1)
stbxe RS,RA,RB . . . . . .(XE-mode, E=1, U=0)
stbuxe RS,RA,RB . . . . . .(XE-mode, E=1, U=1)




```
if RA=0 then a ← 64 0 else a ← GPR(RA)
if D-mode  then EA ← 32 0 || (a + EXTS(D))32:63
if X-mode  then EA ← 32 0 || (a + GPR(RB))32:63
if DE-mode then EA ← a + EXTS(DE)
if XE-mode then EA ← a + GPR(RB)
MEM(EA,1) ← GPR(RS)56:63
if U=1 then GPR(RA) ← EA
```

Let the effective address (EA) be calculated as follows:

- For ***stb*** and ***stbu***, let EA be 32 0s concatenated with bits 32:63 of the sum of the contents of GPR(RA), or 64 0s if RA=0, and the sign-extended value of the D instruction field.

- For ***stbx*** and ***stbux***, let EA be 32 0s concatenated with bits 32:63 of the sum of the contents of GPR(RA), or 64 0s if RA=0, and the contents of GPR(RB).

- For ***stbe*** and ***stbue***, let EA be the sum of the contents of GPR(RA), or 64 0s if RA=0, and the sign-extended value of the DE instruction field.

- For ***stbxe*** and ***stbuxe***, let EA be the sum of the contents of GPR(RA), or 64 0s if RA=0, and the contents of GPR(RB).

The contents of bits 56:63 of GPR(RS) are stored into the byte in storage addressed by EA.

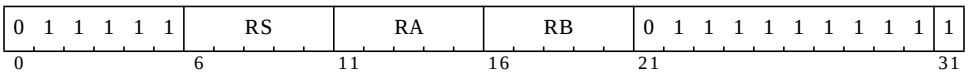
If U=1 (‘with update’), EA is placed into GPR(RA).

If U=1 (‘with update’) and RA=0, the instruction form is invalid.

Special Registers Altered:
None

Store Doubleword Conditional Indexed Extended

stdcxe. RS,RA,RB




```
if RA=0 then EA ← GPR(RB)
if RA≠0 then EA ← GPR(RA) + GPR(RB)
if RESERVE then do
    if RESERVE_ADDR = real_addr(EA) then
        MEM(EA,8) ← GPR(RS)
        CR0 ← 0b00 || 0b1 || XER[SO]
    else
        u ← undefined 1-bit value
        if u then MEM(EA,8) ← GPR(RS)
        CR0 ← 0b00 || u || XER[SO]
    RESERVE ← 0
else
    CR0 ← 0b00 || 0b0 || XER[SO]
```

Let the effective address (EA) be 32 0s concatenated with bits 32:63 of the sum of the contents of GPR(RA), or 64 0s if RA=0, and the contents of GPR(RB).

If a reservation exists and the storage address specified by the ***stdcxe.*** is the same as that specified by the ***ldarxe*** instruction that established the reservation, the contents of GPR(RS) is stored into the doubleword in storage addressed by EA and the reservation is cleared.

If a reservation exists but the storage address specified by the ***stdcxe.*** is not the same as that specified by the ***ldarxe*** instruction that established the reservation, the reservation is cleared, and it is undefined whether the instruction completes without altering storage.

If a reservation does not exist, the instruction completes without altering storage.

CR Field 0 is set to reflect whether the store operation was performed, as follows.

$CR0_{LT\ GT\ EQ\ SO} = 0b00 \| store\_performed \| XER_{SO}$

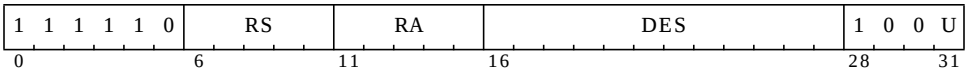
EA must be a multiple of 8. If it is not, either an Alignment interrupt is invoked or the results are boundedly undefined.

Special Registers Altered:

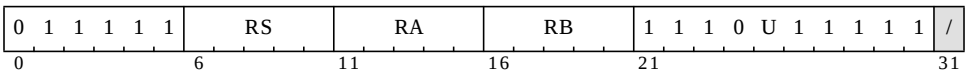
CR0

Store Doubleword [with Update] [Indexed] Extended

stde RS,DES(RA). . . . . .(DES-mode, U=0)
stdue RS,DES(RA). . . . . .(DES-mode, U=1)



stdxe RS,RA,RB . . . . . .(XE-mode, U=0)
stduxe RS,RA,RB . . . . . .(XE-mode, U=1)




```
if RA=0 then a ← 64'0 else a ← GPR(RA)
if DES-mode then EA ← a + EXTS(DES||0b00)
if XE-mode  then EA ← a + GPR(RB)
MEM(EA,8) ← GPR(RS)
if U=1 then GPR(RA) ← EA
```

Let the effective address (EA) be calculated as follows:

- For ***stde*** and ***stdue***, let EA be the sum of the contents of GPR(RA), or 64 0s if RA=0, and the sign-extended value of the DES instruction field concatenated with 0b00.

- For ***stdxe*** and ***stduxe***, let EA be the sum of the contents of GPR(RA), or 64 0s if RA=0, and the contents of GPR(RB).

The contents of GPR(RS) are stored into the doubleword in storage addressed by EA.

If U=1 ('with update'), EA is placed into GPR(RA).

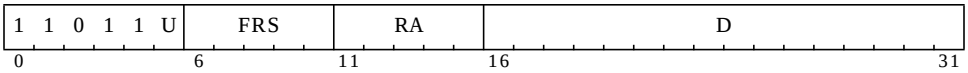
If U=1 ('with update') and RA=0, the instruction form is invalid.

Special Registers Altered:

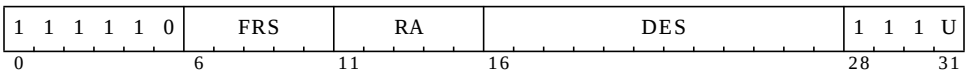
None

Store Floating-Point Double [with Update] [Indexed] [Extended]

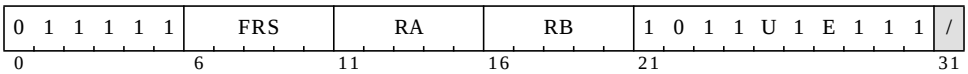
stfd FRS,D(RA) . . . . . .(D-mode, U=0)
stfdu FRS,D(RA) . . . . . .(D-mode, U=1)



stfde FRS,DES(RA). . . . . .(DES-mode, U=0)
stfdue FRS,DES(RA). . . . . .(DES-mode, U=1)



stfdx FRS,RA,RB . . . . . .(X-mode, E=0, U=0)
stfdux FRS,RA,RB . . . . . .(X-mode, E=0, U=1)
stfdxe FRS,RA,RB . . . . . .(XE-mode, E=1, U=0)
stfduxe FRS,RA,RB . . . . . .(XE-mode, E=1, U=1)




```
if RA=0 then a ← 640 else a ← GPR(RA)
if D-mode   then EA ← 320 || (a + EXTS(D))32:63
if X-mode   then EA ← 320 || (a + GPR(RB))32:63
if DES-mode then EA ← a + EXTS(DES||0b00)
if XE-mode  then EA ← a + GPR(RB)
MEM(EA,8) ← FPR(FRS)
if U=1 then GPR(RA) ← EA
```

Let the effective address (EA) be calculated as follows:

- For ***stfd*** and ***stfdu***, let EA be 32 0s concatenated with bits 32:63 of the sum of the contents of GPR(RA), or 64 0s if RA=0, and the sign-extended value of the D instruction field.

- For ***stfdx*** and ***stfdux***, let EA be 32 0s concatenated with bits 32:63 of the sum of the contents of GPR(RA), or 64 0s if RA=0, and the contents of GPR(RB).

- For ***stfde*** and ***stfdue***, let EA be the sum of the contents of GPR(RA), or 64 0s if RA=0, and the sign-extended value of the DES instruction field concatenated with 0b00.

- For ***stfdxe*** and ***stfduxe***, let EA be the sum of the contents of GPR(RA), or 64 0s if RA=0, and the contents of GPR(RB).

The contents of FPR(FRS) are stored into the doubleword in storage addressed by EA.

If U=1 (‘with update’), EA is placed into GPR(RA).

If U=1 (‘with update’) and RA=0, the instruction form is invalid.

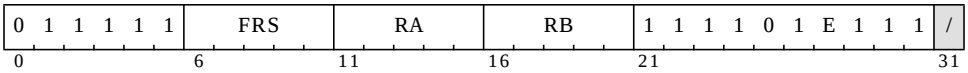
An attempt to execute ***stfd***[***u***][***x***][***e***] while MSR_{FP}=0 will cause a Floating-Point Unavailable interrupt.

Special Registers Altered:

None

Store Floating-Point as Integer Word Indexed [Extended]

stfiwx FRS,RA,RB . . . . . (X-mode, E=0)
stfiwxe FRS,RA,RB . . . . . (XE-mode, E=1)




```
if RA=0 then a ← 64 0 else a ← GPR(RA)
if X-mode   then EA ← 32 0 || (a + GPR(RB))32:63
if XE-mode  then EA ← a + GPR(RB)
MEM(EA,4) ← FPR(FRS)32:63
```

Let the effective address (EA) be calculated as follows:

- For ***stfiwx***, let EA be 32 0s concatenated with bits 32:63 of the sum of the contents of GPR(RA), or 64 0s if RA=0, and the contents of GPR(RB).

- For ***stfiwxe***, let EA be the sum of the contents of GPR(RA), or 64 0s if RA=0, and the contents of GPR(RB).

The contents of bits 32:63 of FPR(FRS) are stored, without conversion, into the word in storage addressed by EA.

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

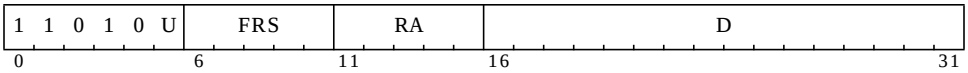
An attempt to execute ***stfiwx***[***e***] while MSR_{FP}=0 will cause a Floating-Point Unavailable interrupt.

Special Registers Altered:

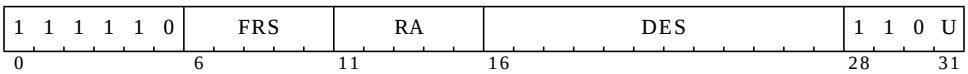
None

Store Floating-Point Single [with Update] [Indexed] [Extended]

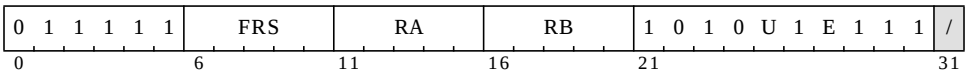
stfs FRS,D(RA) . . . . . .(D-mode, U=0)
stfsu FRS,D(RA) . . . . . .(D-mode, U=1)



stfse FRS,DES(RA). . . . . .(DES-mode, U=0)
stfsue FRS,DES(RA). . . . . .(DES-mode, U=1)



stfsx FRS,RA,RB . . . . . .(X-mode, E=0, U=0)
stfsux FRS,RA,RB . . . . . .(X-mode, E=0, U=1)
stfsxe FRS,RA,RB . . . . . .(XE-mode, E=1, U=0)
stfsuxe FRS,RA,RB . . . . . .(XE-mode, E=1, U=1)




```
if RA=0 then a ← 64 0 else a ← GPR(RA)
if D-mode   then EA ← 32 0 || (a + EXTS(D))32:63
if X-mode   then EA ← 32 0 || (a + GPR(RB))32:63
if DES-mode then EA ← a + EXTS(DES||0b00)
if XE-mode  then EA ← a + GPR(RB)
MEM(EA,4) ← SINGLE(FPR(FRS))
if U=1 then GPR(RA) ← EA
```

Let the effective address (EA) be calculated as follows:

- For ***stfs*** and ***stfsu***, let EA be 32 0s concatenated with bits 32:63 of the sum of the contents of GPR(RA), or 64 0s if RA=0, and the sign-extended value of the D instruction field.

- For ***stfsx*** and ***stfsux***, let EA be 32 0s concatenated with bits 32:63 of the sum of the contents of GPR(RA), or 64 0s if RA=0, and the contents of GPR(RB).

- For ***stfse*** and ***stfsue***, let EA be the sum of the contents of GPR(RA), or 64 0s if RA=0, and the sign-extended value of the DES instruction field concatenated with 0b00.

- For ***stfsxe*** and ***stfsuxe***, let EA be the sum of the contents of GPR(RA), or 64 0s if RA=0, and the contents of GPR(RB).

The contents of FPR(FRS) are converted to single format (see page 100) and stored into the word in storage addressed by EA.

If U=1 (‘with update’), EA is placed into GPR(RA).

If U=1 (‘with update’) and RA=0, the instruction form is invalid.

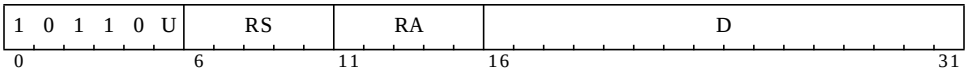
An attempt to execute ***stfs***[***u***][***x***][***e***] while MSR_{FP}=0 will cause a Floating-Point Unavailable interrupt.

Special Registers Altered:

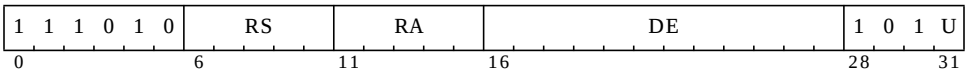
None

Store Halfword [with Update] [Indexed] [Extended]

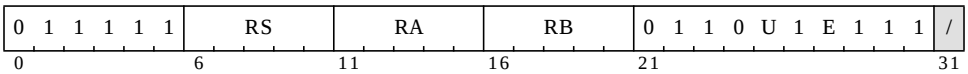
sth RS,D(RA). . . . . .(D-mode, U=0)
sthu RS,D(RA). . . . . .(D-mode, U=1)



sthe RS,DE(RA). . . . . .(DE-mode, U=0)
sthue RS,DE(RA). . . . . .(DE-mode, U=1)



sthx RS,RA,RB . . . . . .(X-mode, E=0, U=0)
sthux RS,RA,RB . . . . . .(X-mode, E=0, U=1)
sthxe RS,RA,RB . . . . . .(XE-mode, E=1, U=0)
sthuxe RS,RA,RB . . . . . .(XE-mode, E=1, U=1)




```
if RA=0 then a ← 64 0 else a ← GPR(RA)
if D-mode  then EA ← 32 0 || (a + EXTS(D))32:63
if X-mode  then EA ← 32 0 || (a + GPR(RB))32:63
if DE-mode then EA ← a + EXTS(DE)
if XE-mode then EA ← a + GPR(RB)
MEM(EA,2) ← GPR(RS)48:63
if U=1 then GPR(RA) ← EA
```

Let the effective address (EA) be calculated as follows:

- For ***sth*** and ***sthu***, let EA be 32 0s concatenated with bits 32:63 of the sum of the contents of GPR(RA), or 64 0s if RA=0, and the sign-extended value of the D instruction field.

- For ***sthx*** and ***sthux***, let EA be 32 0s concatenated with bits 32:63 of the sum of the contents of GPR(RA), or 64 0s if RA=0, and the contents of GPR(RB).

- For ***sthe*** and ***sthue***, let EA be the sum of the contents of GPR(RA), or 64 0s if RA=0, and the sign-extended value of the DE instruction field.

- For ***sthxe*** and ***sthuxe***, let EA be the sum of the contents of GPR(RA), or 64 0s if RA=0, and the contents of GPR(RB).

The contents of bits 48:63 of GPR(RS) are stored into the halfword in storage addressed by EA.

If U=1 (‘with update’), EA is placed into GPR(RA).

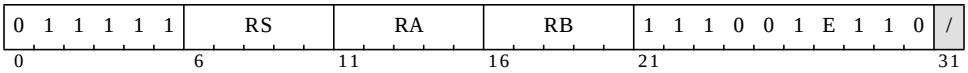
If U=1 (‘with update’) and RA=0, the instruction form is invalid.

Special Registers Altered:

None

Store Halfword Byte-Reverse [Extended]

sthbrx RS,RA,RB . . . . . (X-mode, E=0)
sthbrxe RS,RA,RB . . . . . (XE-mode, E=1)




```
if RA=0 then a ← 64 0 else a ← GPR(RA)
if X-mode  then EA ← 32 0 || (a + GPR(RB))32:63
if XE-mode then EA ← a + GPR(RB)
MEM(EA,2) ← GPR(RS)56:63 || GPR(RS)48:55
```

Let the effective address (EA) be calculated as follows:

- For ***sthbrx***, let EA be 32 0s concatenated with bits 32:63 of the sum of the contents of GPR(RA), or 64 0s if RA=0, and the contents of GPR(RB).

- For ***sthbrxe***, let EA be the sum of the contents of GPR(RA), or 64 0s if RA=0, and the contents of GPR(RB).

Bits 56:63 of GPR(RS) are stored into bits 0:7 of the halfword in storage addressed by EA. Bits 48:55 of GPR(RS) are stored into bits 8:15 of the halfword in storage addressed by EA.

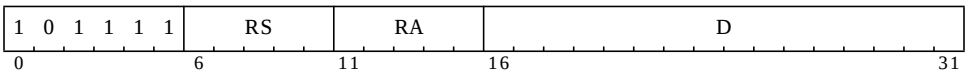
Special Registers Altered:

None

Programming Note
When EA references Big-Endian storage, these instructions have the effect of storing data in Little-Endian byte order. Likewise, when EA references Little-Endian storage, these instructions have the effect of storing data in Big-Endian byte order.

Store Multiple Word

stmw RS,D(RA)




```
if RA=0 then EA ← 32 0 || EXTS(D)32:63
else         EA ← 32 0 || (GPR(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
```

Let the effective address (EA) be 32 0s concatenated with bits 32:63 of the sum of the contents of GPR(RA), or 64 0s if RA=0, and the sign-extended value of the D instruction field.

Let n=(32-RT). Bits 32:63 of registers GPR(RS) through GPR(31) are stored into n consecutive words in storage 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:

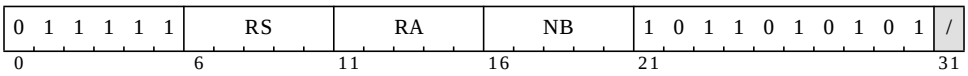
None

Engineering Note
Causing an Alignment interrupt if attempt is made to execute a *Store Multiple* instruction having an incorrectly aligned effective address facilitates the debugging of software.

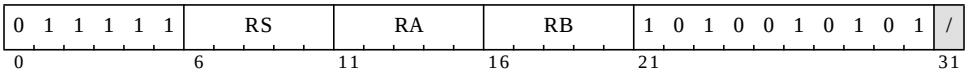
Architecture Note
Extended addressing modes are not defined for *Store Multiple*. Doubleword forms of *Store Multiple* are not defined.

Store String Word (Immediate | Indexed)

stswi RS,RA,NB



stswx RS,RA,RB




```
if RA=0 then a ← 64 0 else a ← GPR(RA)
if 'stswi' then EA ← 32 0 ||  a32:63
if 'stswx' then EA ← 32 0 || (a + GPR(RB))32:63
if 'stswi' & NB=0 then n ← 32
if 'stswi' & NB≠0 then n ← NB
if 'stswx'        then n ← XER57:63
r ← RS - 1
i ← 32
do while n > 0
   if i=32 then r ← r + 1 (mod 32)
   MEM(EA,1) ← GPR(r)i:i+7
   i ← i + 8
   if i = 64 then i ← 32
   EA ← 32 0 || (EA+1)32:63
   n  ← n - 1
```

Let the effective address (EA) be calculated as follows:

- For ***stswi***, let EA be 32 0s concatenated with the contents of bits 32:63 of GPR(RA), or 32 0s if RA=0.

- For ***stswx***, let EA be 32 0s concatenated with bits 32:63 of the sum of the contents of GPR(RA), or 64 0s if RA=0, and the contents of GPR(RB).

If ***stswi*** then let n=NB if NB≠0, n=32 if NB=0. If ***stswx*** then 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.

Registers GPR(RS) through GPR(RS+nr-1) are stored into n consecutive bytes in storage starting at address EA. Data are stored from the low-order four bytes of each GPR.

Bytes are stored left to right from each register. The sequence of registers wraps around to GPR(0) if required.

If ***stswx*** and n=0, no bytes are stored.

Special Registers Altered:

None

Programming Note

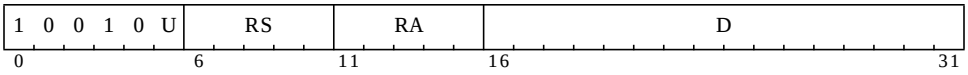
The *Store String Word* instructions, in combination with the *Load String Word* instructions allow movement of data from storage to registers or from registers to storage without concern for alignment. These instructions can be used for a short move between arbitrary storage locations or to initiate a long move between unaligned storage fields.

Architecture Note

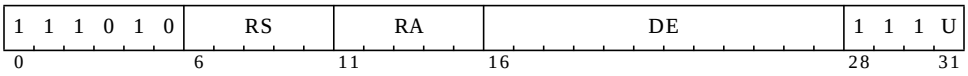
Extended addressing modes are not defined for the *Store String Word* instructions. Doubleword forms of the *Store String Word* instructions are not defined.

Store Word [with Update] [Indexed] [Extended]

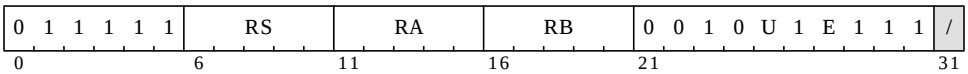
stw RS,D(RA). . . . . .(D-mode, U=0)
stwu RS,D(RA). . . . . .(D-mode, U=1)



stwe RS,DE(RA). . . . . .(DE-mode, U=0)
stwue RS,DE(RA). . . . . .(DE-mode, U=1)



stwx RS,RA,RB . . . . . .(X-mode, E=0, U=0)
stwux RS,RA,RB . . . . . .(X-mode, E=0, U=1)
stwxe RS,RA,RB . . . . . .(XE-mode, E=1, U=0)
stwuxe RS,RA,RB . . . . . .(XE-mode, E=1, U=1)




```
if RA=0 then a ← 64 0 else a ← GPR(RA)
if D-mode  then EA ← 32 0 || (a + EXTS(D))32:63
if X-mode  then EA ← 32 0 || (a + GPR(RB))32:63
if DE-mode then EA ← a + EXTS(DE)
if XE-mode then EA ← a + GPR(RB)
MEM(EA,4) ← GPR(RS)32:63
if U=1 then GPR(RA) ← EA
```

Let the effective address (EA) be calculated as follows:

- For ***stw*** and ***stwu***, let EA be 32 0s concatenated with bits 32:63 of the sum of the contents of GPR(RA), or 64 0s if RA=0, and the sign-extended value of the D instruction field.

- For ***stwx*** and ***stwux***, let EA be 32 0s concatenated with bits 32:63 of the sum of the contents of GPR(RA), or 64 0s if RA=0, and the contents of GPR(RB).

- For ***stwe*** and ***stwue***, let EA be the sum of the contents of GPR(RA), or 64 0s if RA=0, and the sign-extended value of the DE instruction field.

- For ***stwxe*** and ***stwuxe***, let EA be the sum of the contents of GPR(RA), or 64 0s if RA=0, and the contents of GPR(RB).

The contents of bits 32:63 of GPR(RS) are stored into the word in storage addressed by EA.

If U=1 (‘with update’), EA is placed into GPR(RA).

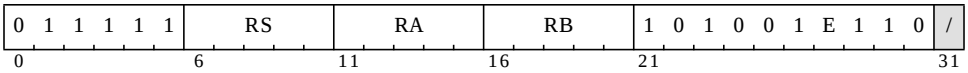
If U=1 (‘with update’) and RA=0, the instruction form is invalid.

Special Registers Altered:

None

Store Word Byte-Reverse [Extended]

stwbrx RS,RA,RB . . . . . (X-mode, E=0)
stwbrxe RS,RA,RB . . . . . (XE-mode, E=1)




```
if RA=0 then a ← 64 0 else a ← GPR(RA)
if X-mode  then EA ← 32 0 || (a + GPR(RB))32:63
if XE-mode then EA ← a + GPR(RB)
MEM(EA,4) ← GPR(RS)56:63 || GPR(RS)48:55 || GPR(RS)40:47 || GPR(RS)32:39
```

Let the effective address (EA) be calculated as follows:

- For ***stwbrx***, let EA be 32 0s concatenated with bits 32:63 of the sum of the contents of GPR(RA), or 64 0s if RA=0, and the contents of GPR(RB).

- For ***stwbrxe***, let EA be the sum of the contents of GPR(RA), or 64 0s if RA=0, and the contents of GPR(RB).

Bits 56:63 of GPR(RS) are stored into bits 0:7 of the word in storage addressed by EA. Bits 48:55 of GPR(RS) are stored into bits 8:15 of the word in storage addressed by EA. Bits 40:47 of GPR(RS) are stored into bits 16:23 of the word in storage addressed by EA. Bits 32:39 of GPR(RS) are stored into bits 24:31 of the word in storage addressed by EA.

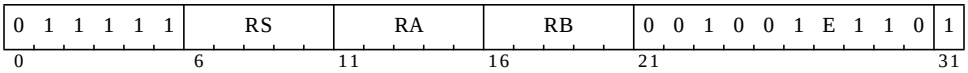
Special Registers Altered:

None

Programming Note
When EA references Big-Endian storage, these instructions have the effect of storing data in Little-Endian byte order. Likewise, when EA references Little-Endian storage, these instructions have the effect of storing data in Big-Endian byte order.

Store Word Conditional Indexed [Extended]

stwcx. RS,RA,RB . . . . . (X-mode, E=0)
stwcxe. RS,RA,RB . . . . . (XE-mode, E=1)




```
if RA=0 then a ← 64 0 else a ← GPR(RA)
if X-mode  then EA ← 32 0 || (a + GPR(RB))32:63
if XE-mode then EA ← a + GPR(RB)
if RESERVE then
   if RESERVE_ADDR = real_addr(EA) then
      MEM(EA,4) ← GPR(RS)32:63
      CR0 ← 0b00 || 0b1 || XERSO
   else
      u ← undefined 1-bit value
      if u then MEM(EA,4) ← GPR(RS)32:63
      CR0 ← 0b00 || u || XERSO
   RESERVE ← 0
else
   CR0 ← 0b00 || 0b0 || XERSO
```

Let the effective address (EA) be calculated as follows:

- For ***stwcx.***, let EA be 32 0s concatenated with bits 32:63 of the sum of the contents of GPR(RA), or 64 0s if RA=0, and the contents of GPR(RB).

- For ***stwcxe.***, let EA be the sum of the contents of GPR(RA), or 64 0s if RA=0, and the contents of GPR(RB).

If a reservation exists and the storage address specified by the ***stwcx.*** or ***stwcxe.*** is the same as that specified by the ***lwarx*** or ***lwarxe*** instruction that established the reservation, the contents of bits 32:63 of GPR(RS) are stored into the word in storage addressed by EA and the reservation is cleared.

If a reservation exists but the storage address specified by the ***stwcx.*** or ***stwcxe.*** 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 storage.

If a reservation does not exist, the instruction completes without altering storage.

CR Field 0 is set to reflect whether the store operation was performed, as follows.

```
CR0LT GT EQ SO = 0b00 || store_performed || XERSO
```

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:

CR0

Programming Note

stwcx., ***stwcxe.***, and ***stdcxe.***, in combination with ***lwarx***, ***lwarxe***, and ***ldarxe***, permit the programmer to write a sequence of instructions that appear to perform an atomic update operation on a storage location. This operation depends upon 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 doublewords.

Programming Note
Because ***stwcx.***, ***stwcxe.***, and ***stdcxe.*** have implementation dependencies (e.g., the granularity at which reservations are managed), they must be used with care. The operating system should provide system library programs that use these instructions to implement the high-level synchronization functions (Test and Set, Compare and Swap, etc.) needed by application programs. Application programs should use these library programs, rather than use ***stwcx.***, ***stwcxe.***, and ***stdcxe.*** directly.

Architecture Note
stwcx., ***stwcxe.***, and ***stdcxe.*** require the EA to be aligned. Software should not attempt to emulate an unaligned ***stwcx.***, ***stwcxe.***, or ***stdcxe.***, because there is no correct way to define the address associated with the reservation.

Engineering Note
Causing an Alignment interrupt to be invoked if an attempt is made to execute a ***stwcx.***, ***stwcxe.***, or ***stdcxe.*** having an incorrectly aligned effective address facilitates the debugging of software by signalling the exception when and where the exception occurs.

Engineering Note
If a *Store Conditional* instruction produces an effective address for which a normal *Store* would cause a Data Storage, Alignment, or Data TLB Error interrupt, but the processor does not have the reservation from a *Load and Reserve* instruction, then it is implementation-dependent whether a Data Storage, Alignment, or Data TLB Error interrupt occurs. See User's Manual for the implementation.

Programming Note
The granularity with which reservations are managed is implementation-dependent. Therefore the storage to be accessed by ***stwcx.***, ***stwcxe.***, or ***stdcxe.*** should be allocated by a system library program. Additional information can be found in Section 6.1.6.2 on page 117.

Programming Note
When correctly used, the *Load And Reserve* and *Store Conditional* instructions can provide an atomic update function for a single aligned word (*Load Word And Reserve* and *Store Word Conditional*) or doubleword (*Load Doubleword And Reserve* and *Store Doubleword Conditional*) of storage.

In general, correct use requires that *Load Word And Reserve* be paired with *Store Word Conditional*, and *Load Doubleword And Reserve* with *Store Doubleword Conditional*, with the same storage address specified by both instructions of the pair. The only exception is that an unpaired *Store Word Conditional* or *Store Doubleword Conditional* instruction 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 Section 11 on page 225.

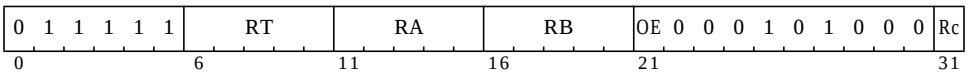
A reservation is cleared if any of the following events occurs.

- 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 Section 6.1.6.2 on page 117, for additional information.

Subtract From

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




```
carry0:63 ← Carry(¬GPR(RA) + GPR(RB) + 1)
sum0:63   ←       ¬GPR(RA) + GPR(RB) + 1
if OE=1 then do
   OV   ← carry32 ⊕ carry33
   SO   ← SO | (carry32 ⊕ carry33)
   OV64 ← carry0 ⊕ carry1
   SO64 ← SO64 | (carry0 ⊕ carry1)
if Rc=1 then do
   LT  ← sum32:63 < 0
   GT  ← sum32:63 > 0
   EQ  ← sum32:63 = 0
   CR0 ← LT || GT || EQ || SO
GPR(RT) ← sum
```

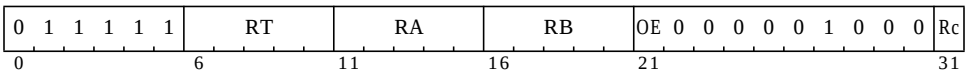
The sum of the one's complement of the contents of GPR(RA), the contents of GPR(RB), and 1 is placed into GPR(RT).

Special Registers Altered:

CR0 . . . . . .(if Rc=1)
SO OV SO64 OV64 . . . . . (if OE=1)

Subtract From Carrying

subfc RT,RA,RB . . . . . .(OE=0, Rc=0)
subfc. RT,RA,RB . . . . . .(OE=0, Rc=1)
subfco RT,RA,RB . . . . . .(OE=1, Rc=0)
subfco. RT,RA,RB . . . . . .(OE=1, Rc=1)




```
carry0:63 ← Carry(¬GPR(RA) + GPR(RB) + 1)
sum0:63   ←       ¬GPR(RA) + GPR(RB) + 1
if OE=1 then do
   OV   ← carry32 ⊕ carry33
   SO   ← SO | (carry32 ⊕ carry33)
   OV64 ← carry0 ⊕ carry1
   SO64 ← SO64 | (carry0 ⊕ carry1)
if Rc=1 then do
   LT  ← sum32:63 < 0
   GT  ← sum32:63 > 0
   EQ  ← sum32:63 = 0
   CR0 ← LT || GT || EQ || SO
GPR(RT) ← sum
CA      ← carry32
CA64    ← carry0
```

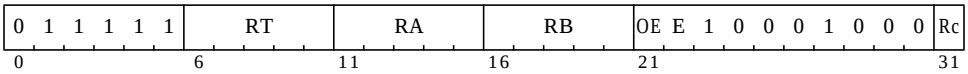
The sum of the one's complement of the contents of GPR(RA), the contents of GPR(RB), and 1 is placed into GPR(RT).

Special Registers Altered:

CA CA64
CR0 . . . . . .(if Rc=1)
SO OV SO64 OV64 . . . . . (if OE=1)

Subtract From Extended

subfe RT,RA,RB . . . . . .(E=0, OE=0, Rc=0)
subfe. RT,RA,RB . . . . . .(E=0, OE=0, Rc=1)
subfeo RT,RA,RB . . . . . .(E=0, OE=1, Rc=0)
subfeo. RT,RA,RB . . . . . .(E=0, OE=1, Rc=1)
subfe64 RT,RA,RB . . . . . .(E=1, OE=0, Rc=0)
subfe64o RT,RA,RB . . . . . .(E=1, OE=1, Rc=0)




```
if E=0 then Cin ← CA else Cin ← CA64
carry0:63 ← Carry(¬GPR(RA) + GPR(RB) + Cin)
sum0:63   ←       ¬GPR(RA) + GPR(RB) + Cin
if OE=1 then do
   OV   ← carry32 ⊕ carry33
   SO   ← SO | (carry32 ⊕ carry33)
   OV64 ← carry0 ⊕ carry1
   SO64 ← SO64 | (carry0 ⊕ carry1)
if Rc=1 then do
   LT  ← sum32:63 < 0
   GT  ← sum32:63 > 0
   EQ  ← sum32:63 = 0
   CR0 ← LT || GT || EQ || SO
GPR(RT) ← sum
CA      ← carry32
CA64    ← carry0
```

For ***subfe***[***o***][**.**], the sum of the one's complement of the contents of GPR(RA), the contents of GPR(RB), and CA is placed into GPR(RT).

For ***subfe64***[***o***], the sum of the one's complement of the contents of GPR(RA), the contents of GPR(RB), and CA64 is placed into GPR(RT).

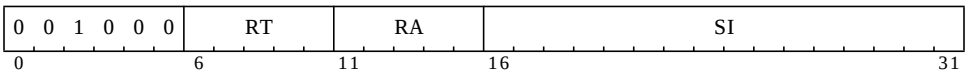
For ***subfe64***[***o***], if Rc=1 the instruction form is invalid.

Special Registers Altered:

CA CA64
CR0 . . . . . .(if Rc=1)
SO OV SO64 OV64 . . . . . (if OE=1)

Subtract From Immediate Carrying

subfic RT,RA,SI




```
carry0:63 ← Carry(¬GPR(RA) + EXTS(SI) + 1)
sum0:63   ←        ¬GPR(RA) + EXTS(SI) + 1
GPR(RT) ← sum
CA      ← carry32
CA64    ← carry0
```

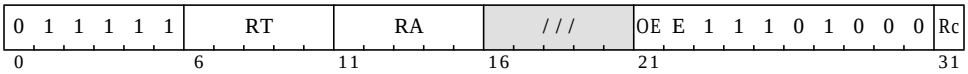
The sum of the one's complement of the contents of GPR(RA), the sign-extended value of the SI field, and 1 is placed into GPR(RT).

Special Registers Altered:

CA CA64

Subtract From Minus One Extended

subfme RT,RA . . . . . .(E=0, OE=0, Rc=0)
subfme. RT,RA . . . . . .(E=0, OE=0, Rc=1)
subfmeo RT,RA . . . . . .(E=0, OE=1, Rc=0)
subfmeo. RT,RA . . . . . .(E=0, OE=1, Rc=1)
subfme64 RT,RA . . . . . .(E=1, OE=0, Rc=0)
subfme64o RT,RA . . . . . .(E=1, OE=1, Rc=0)




```
if E=0 then Cin ← CA else Cin ← CA64
carry0:63 ← Carry(¬GPR(RA) + Cin + 0xFFFF_FFFF_FFFF_FFFF)
sum0:63   ←       ¬GPR(RA) + Cin + 0xFFFF_FFFF_FFFF_FFFF
if OE=1 then do
   OV   ← carry32 ⊕ carry33
   SO   ← SO | (carry32 ⊕ carry33)
   OV64 ← carry0 ⊕ carry1
   SO64 ← SO64 | (carry0 ⊕ carry1)
if Rc=1 then do
   LT  ← sum32:63 < 0
   GT  ← sum32:63 > 0
   EQ  ← sum32:63 = 0
   CR0 ← LT || GT || EQ || SO
GPR(RT) ← sum
CA      ← carry32
CA64    ← carry0
```

For ***subfme***[***o***][**.**], the sum of the one's complement of the contents of GPR(RA), CA, and $^{64}1$ is placed into GPR(RT).

For ***subfme64***[***o***], the sum of the one's complement of the contents of GPR(RA), CA64, and $^{64}1$ is placed into GPR(RT).

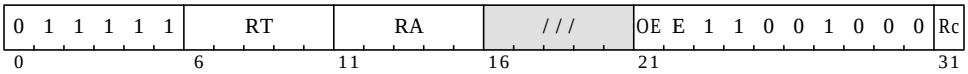
For ***subfme64***[***o***], if Rc=1 the instruction form is invalid.

Special Registers Altered:

CA CA64
CR0 . . . . . .(if Rc=1)
SO OV SO64 OV64 . . . . . (if OE=1)

Subtract From Zero Extended

subfze RT,RA . . . . . .(E=0, OE=0, Rc=0)
subfze. RT,RA . . . . . .(E=0, OE=0, Rc=1)
subfzeo RT,RA . . . . . .(E=0, OE=1, Rc=0)
subfzeo. RT,RA . . . . . .(E=0, OE=1, Rc=1)
subfze64 RT,RA . . . . . .(E=1, OE=0, Rc=0)
subfze64o RT,RA . . . . . .(E=1, OE=1, Rc=0)




```
if E=0 then Cin ← CA else Cin ← CA64
carry0:63 ← Carry(¬GPR(RA) + Cin)
sum0:63   ←       ¬GPR(RA) + Cin
if OE=1 then do
   OV   ← carry32 ⊕ carry33
   SO   ← SO | (carry32 ⊕ carry33)
   OV64 ← carry0 ⊕ carry1
   SO64 ← SO64 | (carry0 ⊕ carry1)
if Rc=1 then do
   LT  ← sum32:63 < 0
   GT  ← sum32:63 > 0
   EQ  ← sum32:63 = 0
   CR0 ← LT || GT || EQ || SO
GPR(RT) ← sum
CA      ← carry32
CA64    ← carry0
```

For ***subfze***[***o***][**.**], the sum of the one's complement of the contents of GPR(RA) and CA is placed into GPR(RT).

For ***subfze64***[***o***], the sum of the one's complement of the contents of GPR(RA) and CA64 is placed into GPR(RT).

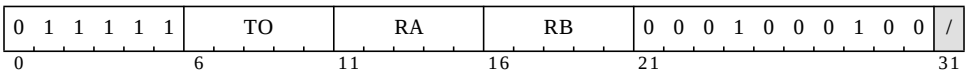
For ***subfze64***[***o***], if Rc=1 the instruction form is invalid.

Special Registers Altered:

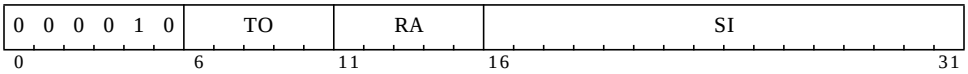
CA CA64
CR0 . . . . . .(if Rc=1)
SO OV SO64 OV64 . . . . . (if OE=1)

Trap Doubleword [Immediate]

td TO,RA,RB



tdi TO,RA,SI




```
a ← GPR(RA)
if ‘td’  then b ← GPR(RB)
if ‘tdi’ then b ← EXTS(SI)
if (a <  b) & TO0 then TRAP
if (a >  b) & TO1 then TRAP
if (a =  b) & TO2 then TRAP
if (a <u b) & TO3 then TRAP
if (a >u b) & TO4 then TRAP
```

If ***td***, the contents of GPR(RA) are compared with the contents of GPR(RB).

If ***tdi***, the contents of GPR(RA) are compared with the sign-extended value of the SI field.

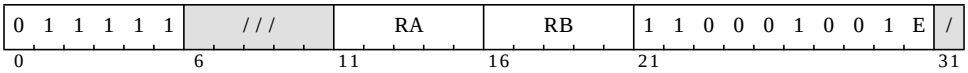
If any bit in the TO field is set to 1 and its corresponding condition is met by the result of the comparison, then the system trap handler is invoked.

Special Registers Altered:

None

TLB Invalidate Virtual Address Indexed [Extended]

tlbivax . . . . . (X-mode, E=0)
tlbivaxe . . . . . (XE-mode, E=1)




```
if RA=0 then a ← 64 0 else a ← GPR(RA)
if E=0 then EA ← 32 0 || (a + GPR(RB))32:63
if E=1 then EA ← a + GPR(RB)
AS        ← implementation-dependent value
ProcessID ← implementation-dependent value
VA        ← AS || ProcessID || EA
InvalidateTLB(VA)
```

Let the effective address (EA) be calculated as follows:

Addressing Mode	EA for RA=0	EA for RA≠0
X-mode	$^{32}0$ || $GPR(RB)_{32:63}$	$^{32}0$ || $(GPR(RA)+GPR(RB))_{32:63}$
XE-mode	GPR(RB)	GPR(RA)+GPR(RB)

Let address space (AS) be defined as implementation-dependent (e.g. could be MSR_{DS} or a bit from an implementation-dependent SPR).

Let the ProcessID be defined as implementation-dependent (e.g. could be from the PID Register or from an implementation-dependent SPR).

Let the virtual address (VA) be the value AS || ProcessID || EA. See Figure 6-2 on page 128.

If the Translation Lookaside Buffer (TLB) contains an entry corresponding to VA, that entry is made invalid (i.e. 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 the ***mbar*** (or ***msync***) instruction with respect to a subsequent ***tlbsync*** instruction executed by the processor executing the ***tlbivax***[***e***] instruction. The operations caused by ***tlbivax***[***e***] and ***tlbsync*** are ordered by ***mbar*** as a set of operations which is independent of the other sets that ***mbar*** orders.

Execution of this instruction is privileged and restricted to supervisor mode only.

Special Registers Altered:

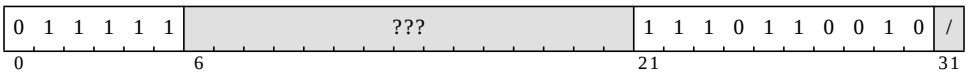
None

Programming Note
The effects of the invalidation are not guaranteed to be visible to the programming model until the completion of a context synchronizing operation (see Section 1.12.1 on page 38).

Programming & Engineering Note
Care must be taken not to invalidate any TLB entry that contains the mapping for any interrupt vector.

TLB Read Entry

tlbre



Bits 6:20 of the instruction 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 that the result is placed into.

The implementation-dependent-specified TLB entry is read, and the implementation-dependent-specified portion of the TLB entry is extracted and placed into an implementation-dependent target resource.

If the instruction specifies a TLB entry that does not exist, the results are undefined.

Execution of this instruction may cause other implementation-dependent effects. See User's Manual for the implementation.

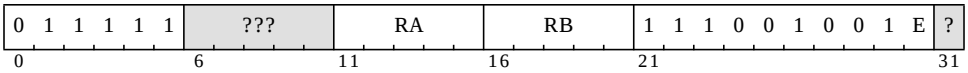
Execution of this instruction is privileged and restricted to supervisor mode only.

Special Registers Altered:

Implementation-dependent

TLB Search Indexed [Extended]

tlbsx RA,RB . . . . . (X-mode, E=0)
tlbsxe RA,RB . . . . . (XE-mode, E=1)




```
if RA=0 then a ← 64 0 else a ← GPR(RA)
if X-mode  then EA ← 32 0 || (a + GPR(RB))32:63
if XE-mode then EA ← a + GPR(RB)
ProcessID ← implementation-dependent value
AS        ← implementation-dependent value
VA        ← AS || ProcessID || EA
if Valid_TLB_matching_entry_exists( VA )
   then result ← Implementation-dependent value
   else result ← Undefined
implementation-dependent target resource ← result
```

Let the effective address (EA) be calculated as follows:

Addressing Mode	EA for RA=0	EA for RA≠0
X-mode	$^{32}0 \| \| GPR(RB)_{32:63}$	$^{32}0 \| \| (GPR(RA)+GPR(RB))_{32:63}$
XE-mode	GPR(RB)	GPR(RA)+GPR(RB)

Let the address space (AS) be defined as implementation-dependent (e.g. AS could be MSR_{DS} or a bit from an implementation-dependent SPR).

Let the ProcessID be defined as implementation-dependent (e.g. could be from the PID register or from an implementation-dependent SPR).

Let the virtual address (VA) be the value AS || ProcessID || EA. See Figure 6-2 on page 128.

Bits 6:10 of the instruction encoding are allocated for implementation-dependent use, and may be used to specify the target resource that the result of the instruction is placed into.

If the Translation Lookaside Buffer (TLB) contains an entry corresponding to VA, an implementation-dependent value is placed into an implementation-dependent-specified target. Otherwise the contents of the implementation-dependent-specified target are left undefined.

Bit 31 of the instruction encoding is allocated for implementation-dependent use. For example, bit 31 may be interpreted as an 'Rc' bit, used to enable recording the success or failure of the search operation.

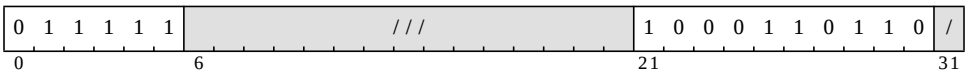
Execution of this instruction is privileged and restricted to supervisor mode only.

Special Registers Altered:

Implementation-dependent

TLB Synchronize

tlbsync



The ***tlbsync*** instruction provides an ordering function for the effects of all ***tlbivax***[***e***] 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 will occur.

- All TLB invalidations caused by ***tlbivax***[***e***] instructions preceding the ***tlbsync*** instruction will have completed on any other processor before any storage accesses associated with data accesses caused by instructions following the ***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.

The operation performed by this instruction is ordered by the ***mbar*** and ***msync*** instructions with respect to preceding ***tlbivax***[***e***] instructions executed by the processor executing the ***tlbsync*** instruction. The operations caused by ***tlbivax***[***e***] and ***tlbsync*** are ordered by ***mbar*** as a set of operations, which is independent of the other sets that ***mbar*** orders.

The ***tlbsync*** instruction may complete before operations caused by ***tlbivax***[***e***] instructions preceding the ***tlbsync*** instruction have been performed.

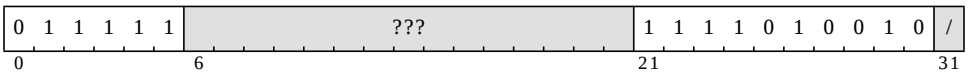
Execution of this instruction is privileged and restricted to supervisor mode only.

Special Registers Altered:

None

TLB Write Entry

tlbwe



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 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 User's Manual for the implementation.

Execution of this instruction is privileged and restricted to supervisor mode only.

Special Registers Altered:

None

Programming Notes

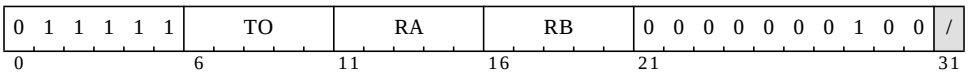
The effects of the update are not guaranteed to be visible to the programming model until the completion of a context synchronizing operation. See Section 1.12.1 on page 38.

Programming & Engineering Note

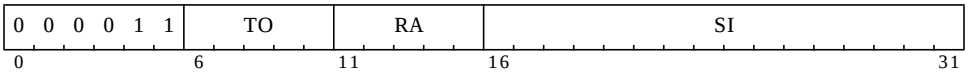
Care must be taken not to invalidate any TLB entry that contains the mapping for any interrupt vector.

Trap Word [Immediate]

tw TO,RA,RB



twi TO,RA,SI




```
a ← EXTS(GPR(RA)32:63)
if ‘tw’  then b ← EXTS(GPR(RB)32:63)
if ‘twi’ then b ← EXTS(SI)
if (a <  b) & TO0 then TRAP
if (a >  b) & TO1 then TRAP
if (a =  b) & TO2 then TRAP
if (a <u b) & TO3 then TRAP
if (a >u b) & TO4 then TRAP
```

For ***tw***, the contents of bits 32:63 of GPR(RA) are compared with the contents of bits 32:63 of GPR(RB).

For ***twi***, the contents of bits 32:63 of GPR(RA) are compared with the sign-extended value of the SI field.

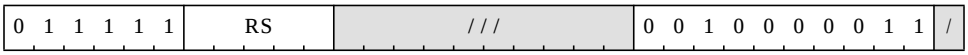
If any bit in the TO field is set to 1 and its corresponding condition is met by the result of the comparison, then the system trap handler is invoked.

Special Registers Altered:

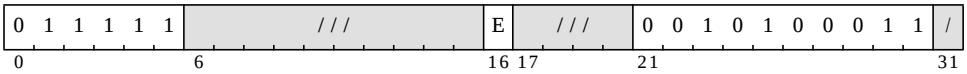
None

Write MSR External Enable [Immediate]

wrtee RS



wrteei E




```
if 'wrtee'  then MSR_EE ← GPR(RS)_48
if 'wrteei' then MSR_EE ← E
```

For ***wrtee***, bit 48 of the contents of GPR(RS) is placed into MSR_{EE}.

For ***wrteei***, the value specified in the E field is placed into MSR_{EE}.

Execution of this instruction is privileged and restricted to supervisor mode only.

In addition, alteration of the MSR_{EE} bit is effective as soon as the instruction completes. Thus if MSR_{EE}=0 and an External interrupt is pending, executing an ***wrtee*** or ***wrteei*** that sets MSR_{EE} to 1 will cause the External interrupt to be taken before the next instruction is executed, if no higher priority exception exists. (See Section 7.9 on page 178).

Special Registers Altered:

MSR

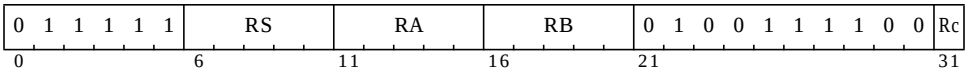
Programming Note

wrtee and ***wrteei*** are used to provide atomic update of MSR_{EE}. Typical usage is:

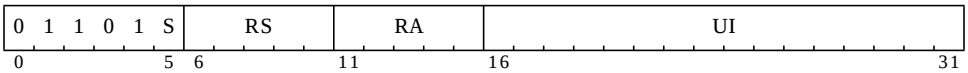
```
mfmsr   Rn          #save EE in GPR(Rn)_48
wrteei  0           #turn off EE
  :     :           :
  :     :           #code with EE disabled
  :     :           :
wrtee   Rn          #restore EE without altering other MSR bits that
                    #may have changed
```

XOR [Immediate [Shifted]]

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



xori RA,RS,UI. . . . . (S=0, Rc=0)
xoris RA,RS,UI. . . . . (S=1, Rc=0)




```
if 'xori'   then b ← 48 0 || UI
if 'xoris'  then b ← 32 0 || UI || 16 0
if 'xor[.]' then b ← GPR(RB)
result0:63 ← GPR(RS) ⊕ b
if Rc=1 then do
   LT  ← result32:63 < 0
   GT  ← result32:63 > 0
   EQ  ← result32:63 = 0
   CR0 ← LT || GT || EQ || SO
GPR(RA) ← result
```

For ***xori***, the contents of GPR(RS) are XORed with $^{48}0$ || UI.

For ***xoris***, the contents of GPR(RS) are XORed with $^{32}0$ || UI || $^{16}0$.

For ***xor*[.]**, the contents of GPR(RS) are XORed with the contents of GPR(RB).

The result is placed into GPR(RA).

Special Registers Altered:

CR0 . . . . . .(if Rc=1)

Appendix A Guidelines for 32-bit Book E

A.1 32-bit Book E Implementation Guidelines

A.1.1 64-bit-Specific Book E Instructions

There is a subset of Book E instructions that are considered restricted only to 64-bit Book E processing. A 32-bit Book E implementation need not implement any of the following instructions. Likewise, neither should 32-bit Book E applications utilize any of these instructions. All other Book E instructions shall either be supported directly by the implementation, or sufficient infrastructure will be provided to enable software emulation of the instructions.

64-bit integer arithmetic, compare, shift and rotate instructions
adde64[***o***], ***addme64***[***o***], ***addze64***[***o***],
subfe64[***o***], ***subfme64***[***o***], ***subfze64***[***o***],
mulhd, ***mulhdu***, ***mulld***[***o***], ***divd***, ***divdu***, ***extsw***,
cmp (L=1), ***cmpi*** (L=1), ***cmpl*** (L=1), ***cmpli*** (L=1),
rldcl, ***rldcr***, ***rldic***, ***rldicl***, ***rldicr***, ***rldimi***, ***sld***, ***srad***, ***sradi***, ***srd***,
cntlzd, ***td***, ***tdi***

64-bit extended addressing branch instructions
bcctre[***l***], ***bce***[***l***][***a***], ***bclre***[***l***], ***be***[***l***][***a***]

64-bit extended addressing cache management instructions
dcbae, ***dcbfe***, ***dcbie***, ***dcbste***, ***dcbte***, ***dcbtste***, ***dcbze***, ***icbie***, ***icbte***

64-bit extended addressing load instructions
lbze, ***lbzue***, ***lbzxe***, ***lbzuxe***, ***ldarxe***, ***lde***, ***ldue***, ***ldxe***, ***lduxe***, ***lfde***, ***lfdue***, ***lfdxe***, ***lfduxe***, ***lfse***, ***lfsue***, ***lfsxe***, ***lfsuxe***, ***lhae***, ***lhaue***, ***lhaxe***, ***lhauxe***, ***lhbrxe***, ***lhze***, ***lhzue***, ***lhzxe***, ***lhzuxe***, ***lwarxe***, ***lwbrxe***, ***lwze***, ***lwzue***, ***lwzxe***, ***lwzuxe***

64-bit extended addressing store instructions
stbe, ***stbue***, ***stbxe***, ***stbuxe***, ***stdcxe.***, ***stde***, ***stdue***, ***stdxe***, ***stduxe***, ***stfde***, ***stfdue***, ***stfdxe***, ***stfduxe***, ***stfiwxe***, ***stfse***, ***stfsue***, ***stfsxe***, ***stfsuxe***, ***sthbrxe***, ***sthe***, ***sthue***, ***sthxe***, ***sthuxe***, ***stwbrxe***, ***stwcxe.***, ***stwe***, ***stwue***, ***stwxe***, ***stwuxe***

A.1.2 Registers on 32-bit Book E Implementations

Book E provides 32-bit and 64-bits registers. All 32-bit registers shall be supported as defined in the Book E specification. 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 a 32-bit Book E implementation. Such 64-bit registers include the LR, the CTR, the 32 GPRs, SRR0 and CSRR0. Book E remains silent on implementing a subset of the 64-bit floating-point architecture.

Likewise, other than floating-point instructions, all instructions which are defined to return a 64-bit result shall return only bits 32:63 of the result on a 32-bit Book E implementation.

A.1.3 Addressing on 32-bit Book E Implementations

Only bits 32:63 of the 64-bit Book E instruction and data storage effective addresses need to be calculated and presented to main storage. Given the only branch and data storage access instructions that are not included in Section A.1 are defined to prepend 32 0s to bits 32:63 of the effective address computation, a 32-bit implementation can simply bypass the prepending of the 32 0s when implementing these instructions. For *Branch to Link Register* and *Branch to Count Register* instructions, given the LR and CTR are implemented only as 32-bit registers, only concatenating 2 0s to the right of bits 32:61 of these registers is necessary to form the 32-bit branch target address.

For next sequential instruction address computation, the simplest implementation would suggest allowing the effective address computations to wrap from 0xFFFF_FFFC to 0x0000_0000. This wrapping is the required behavior 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).

A.1.4 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.

A.2 32-bit Book E Software Guidelines

A.2.1 32-bit Instruction Selection

Any Book E software that uses any of the instructions listed in Section A.1.1 shall be considered 64-bit Book E software, and correct execution cannot be guaranteed on 32-bit Book E implementations. Generally speaking, 32-bit software should avoid using any instruction or instructions that depend on any particular setting of bits 0:31 of any 64-bit application-accessible system register, including General Purpose Registers, for producing the correct 32-bit results. Context

switching may or may not preserve the upper 32 bits of application-accessible 64-bit system registers and insertion of arbitrary settings of those upper 32 bits at arbitrary times during the execution of the 32-bit application must not affect the final result.

A.2.2 32-bit Addressing

Book E provides a complete set of data storage access instructions that perform a modulo 2^{32} on the computed effective address and then prepend 32 0s 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 0s 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 0s (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 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 will allow the application to execute on 32-bit and 64-bit Book E implementations.

Appendix B Special Purpose Registers Index

Special Purpose Registers (SPRs) are on-chip registers that are architecturally part of the processor core. They are accessed with the ***mtspr*** (page 316) and ***mfspr*** (page 309) instructions. Encodings not listed are reserved for future use or for use as implementation-specific registers.

In Table B-1, the column 'SPRN' (SPR number) lists register numbers, which are used in the instruction mnemonics.

Special purpose registers control the use of the debug facilities, the timers, the interrupts, the memory management unit, and other architected processor resources.

B.1 Defined Special Purpose Registers

Table B-1 provides a summary of all Special Purpose Registers defined in the Book E.

Table B-1. Defined Special Purpose Registers

Defined SPR	Defined SPR Name	Defined SPRN		Access	Privileged	Page
		Decimal	Binary			
CSRR0	Critical Save/Restore Register 0	58	00001 1 1010	Read/Write	Yes	144
CSRR1	Critical Save/Restore Register 1	59	00001 1 1011	Read/Write	Yes	145
CTR	Count Register	9	00000 0 1001	Read/Write	No	48
DAC1	Data Address Compare 1	316	01001 1 1100	Read/Write	Yes	218
DAC2	Data Address Compare 2	317	01001 1 1101			
DBCR0	Debug Control Register 0	308	01001 1 0100	Read/Write	Yes	210
DBCR1	Debug Control Register 1	309	01001 1 0101	Read/Write	Yes	212
DBCR2	Debug Control Register 2	310	01001 1 0110	Read/Write	Yes	215
DBSR	Debug Status Register	304	01001 1 0000	Read/Clear[1]	Yes	217
DEAR	Data Exception Address Register	61	00001 1 1101	Read/Write	Yes	145
DEC	Decrementer	22	00000 1 0110	Read/Write	Yes	194
DECAR	Decrementer Auto-Reload	54	00001 1 0110	Write-only		
DVC1	Data Value Compare 1	318	01001 1 1110	Read/Write	Yes	219
DVC2	Data Value Compare 2	319	01001 1 1111			
ESR	Exception Syndrome Register	62	00001 1 1110	Read/Write	Yes	146
IVPR	Interrupt Vector Prefix Register	63	00001 1 1111	Read/Write	Yes	145
IAC1	Instruction Address Compare 1	312	01001 1 1000	Read/Write	Yes	218
IAC2	Instruction Address Compare 2	313	01001 1 1001			
IAC3	Instruction Address Compare 3	314	01001 1 1010			
IAC4	Instruction Address Compare 4	315	01001 1 1011			
IVOR0	Interrupt Vector Offset Register 0	400	01100 1 0000	Read/Write	Yes	147
IVOR1	Interrupt Vector Offset Register 1	401	01100 1 0001			
IVOR2	Interrupt Vector Offset Register 2	402	01100 1 0010			
IVOR3	Interrupt Vector Offset Register 3	403	01100 1 0011			
IVOR4	Interrupt Vector Offset Register 4	404	01100 1 0100			
IVOR5	Interrupt Vector Offset Register 5	405	01100 1 0101			
IVOR6	Interrupt Vector Offset Register 6	406	01100 1 0110			
IVOR7	Interrupt Vector Offset Register 7	407	01100 1 0111			
IVOR8	Interrupt Vector Offset Register 8	408	01100 1 1000			
IVOR9	Interrupt Vector Offset Register 9	409	01100 1 1001			
IVOR10	Interrupt Vector Offset Register 10	410	01100 1 1010			
IVOR11	Interrupt Vector Offset Register 11	411	01100 1 1011			
IVOR12	Interrupt Vector Offset Register 12	412	01100 1 1100			
IVOR13	Interrupt Vector Offset Register 13	413	01100 1 1101			
IVOR14	Interrupt Vector Offset Register 14	414	01100 1 1110			
IVOR15	Interrupt Vector Offset Register 15	415	01100 1 1111			
LR	Link Register	8	00000 0 1000	Read/Write	No	48
PID	Process ID Register[2]	48	00001 1 0000	Read/Write	Yes	121
PIR	Processor ID Register	286	01000 1 1110	Read-only	Yes	41
PVR	Processor Version Register	287	01000 1 1111	Read-only	Yes	41

Defined SPR	Defined SPR Name	Defined SPRN		Access	Privileged	Page
		Decimal	Binary			
SPRG0	SPR General 0	272	01000 1 0000	Read/Write	Yes	42
SPRG1	SPR General 1	273	01000 1 0001	Read/Write	Yes	
SPRG2	SPR General 2	274	01000 1 0010	Read/Write	Yes	
SPRG3	SPR General 3	259	01000 0 0011	Read-only	No[3]	
		275	01000 1 0011	Read/Write	Yes	
SPRG4	SPR General 4	260	01000 0 0100	Read-only	No	
		276	01000 1 0100	Read/Write	Yes	
SPRG5	SPR General 5	261	01000 0 0101	Read-only	No	
		277	01000 1 0101	Read/Write	Yes	
SPRG6	SPR General 6	262	01000 0 0110	Read-only	No	
		278	01000 1 0110	Read/Write	Yes	
SPRG7	SPR General 7	263	01000 0 0111	Read-only	No	
		279	01000 1 0111	Read/Write	Yes	
USPRG0	User SPR General 0[4]	256	01000 0 0000	Read/Write	No	42
SRR0	Save/Restore Register 0	26	00000 1 1010	Read/Write	Yes	144
SRR1	Save/Restore Register 1	27	00000 1 1011	Read/Write	Yes	144
TBL	Time Base Lower	268	01000 0 1100	Read-only	No	189
		284	01000 1 1100	Write-only	Yes	
TBU	Time Base Upper	269	01000 0 1101	Read-only	No	
		285	01000 1 1101	Write-only	Yes	
TCR	Timer Control Register	340	01010 1 0100	Read/Write	Yes	186
TSR	Timer Status Register	336	01010 1 0000	Read/Clear[5]	Yes	188
XER	Integer Exception Register	1	00000 0 0001	Read/Write	No	53

1. The Debug Status Register can be read using *mfspr RT,DBSR*. The Debug Status Register cannot be directly written to. Instead, bits in the Debug Status Register corresponding to 1 bits in GPR(RS) can be cleared using *mtspr DBSR,RS*.
2. Implementations may support more than one Process ID register. SPR numbers 49-55 are reserved for implementations that support more than one Process ID register. See the User's Manual for the implementation.
3. User-mode read access to SPRG3 is implementation-dependent. See the User's Manual for the implementation.
4. USPRG0 is a separate physical register from SPRG0.
5. The Timer Status Register can be read using *mfspr RT,TSR*. The Timer Status Register cannot be directly written to. Instead, bits in the Timer Status Register corresponding to 1 bits in GPR(RS) can be cleared using *mtspr TSR,RS*.

B.2 Preserved Special Purpose Registers

Preserved SPRNs are SPRNs that otherwise would be classified as reserved, but have legacy use which requires their deployment in the Book E to be deferred as long as possible in order to allow legacy hardware and software to migrate these legacy Special Purpose Registers to Book E allocated SPRN space.

Table B-2. Preserved Special Purpose Registers

Preserved SPR	Preserved SPRN	
	Decimal	Binary
PowerPC DSISR	18	00000 1 0010
PowerPC DAR	19	00000 1 0011
PowerPC SDR1	25	00000 1 1001
8xx EIE	80	00010 1 0000
8xx EID	81	00010 1 0001
8xx NRE	82	00010 1 0010
5xx,8xx CMPA	144	00100 1 0000
5xx,8xx CMPB	145	00100 1 0001
5xx,8xx CMPC	146	00100 1 0010
5xx,8xx CMPD	147	00100 1 0011
5xx,8xx ICR	148	00100 1 0100
5xx,8xx DER	149	00100 1 0101
5xx,8xx COUNTA	150	00100 1 0110
5xx,8xx COUNTB	151	00100 1 0111
5xx,8xx CMPE	152	00100 1 1000
5xx,8xx CMPF	153	00100 1 1001
5xx,8xx CMPG	154	00100 1 1010
5xx,8xx CMPH	155	00100 1 1011
5xx,8xx LCTRL1	156	00100 1 1100
5xx,8xx LCTRL2	157	00100 1 1101
5xx,8xx ICTRL	158	00100 1 1110
5xx,8xx BAR	159	00100 1 1111
PowerPC ASR	280	01000 1 1000
PowerPC EAR	282	01000 1 1010

B.3 Reserved Special Purpose Registers

Any SPRN in the range 0x000-0x1FF (0-511) that is not Defined (see Table B-1) and is not Preserved (see Table B-2) is Reserved.

B.4 Allocated Special Purpose Registers

SPRNs that are allocated for implementation-dependent use are 0x200-0x3FF (512-1023).

Appendix C Programming Examples

C.1 Synchronization

This section gives examples of how the *Storage Synchronization* instructions can be used to emulate various synchronization primitives and to provide more complex forms of synchronization.

These examples have a common form. After possible initialization, there is a 'conditional sequence' that begins with a *Load And Reserve* instruction, which may be followed by memory accesses and/or computation that include neither a *Load And Reserve* nor a *Store Conditional*, and ends with a *Store Conditional* instruction 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 storage locations: see Note 4 in Section C.1.4, "Notes", on page 386 for a discussion of how the retry strategy can affect performance.

The *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. See Section 6.1.6.1, "Storage Access Ordering", on page 114 , for additional details.

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 storage (e.g., an I/O device), or the initial state of storage.

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.

The examples deal with words: they can be used for doublewords by changing all ***lwarx*** instructions to ***ldarxe***, all ***stwcx.*** instructions to ***stdcxe.***, all ***stw*** instructions to ***std***, and all ***cmp[i]*** instructions with L=0 to ***cmp[i]*** with L=1. ***lwarx-stwcx.*** pairs can also be substituted with ***lwarxe-stwcxe.*** pairs.

Programming Note
Because the *Storage Synchronization* instructions have implementation dependencies (e.g., the granularity at which reservations are managed), they must be used with care. The operating system should provide system library programs that use these instructions to implement the high-level synchronization functions (Test and Set, Compare and Swap, etc.) needed by application programs. Application programs should use these library programs, rather than use the *Storage 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, setting the EQ bit to 0, if the word loaded by ***lwarx*** has changed before the ***stwcx.*** is executed: see Section 6.1.6.2, “Atomic Update Primitives”, on page 117 for more detail.

Fetch and No-op

The ‘Fetch and No-op’ primitive atomically loads the current value in a word in storage.

In this example it is assumed that the address of the word to be loaded is in GPR(3) and the data loaded are returned in GPR(4).

```
loop: lwarx  r4,0,r3      #load and reserve
      stwcx. r4,0,r3      #store old value if
                          #  still reserved
      bc     4,2,loop     #loop if lost reservation
```

Note:

1. The ***stwcx.***, if it succeeds, stores to the target location the same value that was loaded by the preceding ***lwarx***. While the store is redundant with respect to the value in the location, its success ensures that the value loaded by the ***lwarx*** was the current value, i.e., 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 storage.

In this example it is assumed that the address of the word to be loaded and replaced is in GPR(3), the new value is in GPR(GPR(4), and the old value is returned in GPR(5).

```
loop: lwarx  r5,0,r3      #load and reserve
      stwcx. r4,0,r3      #store new value if
                          #  still reserved
      bc     4,2,loop     #loop if lost reservation
```

Fetch and Add

The 'Fetch and Add' primitive atomically increments a word in storage.

In this example it is assumed that the address of the word to be incremented is in GPR(3), the increment is in GPR(4), and the old value is returned in GPR(5).

```
loop: lwarx  r5,0,r3      #load and reserve
      add    r0,r4,r5     #increment word
      stwcx. r0,0,r3      #store new value if
                          #  still reserved
      bc     4,2,loop     #loop if lost reservation
```

Fetch and AND

The 'Fetch and AND' primitive atomically ANDs a value into a word in storage.

In this example it is assumed that the address of the word to be ANDed is in GPR(3), the value to AND into it is in GPR(4), and the old value is returned in GPR(5).

```
loop: lwarx  r5,0,r3      #load and reserve
      and    r0,r4,r5     #AND word
      stwcx. r0,0,r3      #store new value if
                          #  still reserved
      bc     4,2,loop     #loop if lost reservation
```

Note:

1. The sequence given above can be changed to perform another Boolean operation atomically on a word in storage, simply by changing the ***and*** instruction to the desired Boolean instruction (***or***, ***xor***, etc.).

Test and Set

This version of the 'Test and Set' primitive atomically loads a word from storage, sets the word in storage to a nonzero value if the value loaded is zero, and sets the EQ bit of CR Field 0 to indicate whether the value loaded is zero.

In this example it is assumed that the address of the word to be tested is in GPR(3), the new value (nonzero) is in GPR(4), and the old value is returned in GPR(5).

```
loop: lwarx  r5,0,r3      #load and reserve
      cmpwi  r5,0         #done if word
      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 storage, if they are equal stores the value from a second register into the word in storage, if they are unequal loads the word from storage into the first register, and sets the EQ bit of CR Field 0 to indicate the result of the comparison.

In this example it is assumed that the address of the word to be tested is in GPR(3), the comparand is in GPR(4) and the old value is returned there, and the new value is in GPR(5).

```
loop: lwarx  r6,0,r3      #load and reserve
      cmpw   r4,r6        #1st 2 operands equal?
      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 storage
```

Notes:

1. The semantics given for 'Compare and Swap' above are based on those of the IBM System/370 Compare and Swap instruction. Other architectures may define a Compare and Swap instruction differently.

2. 'Compare and Swap' is shown primarily for pedagogical reasons. It is useful on machines that lack the better synchronization facilities provided by ***lwarx*** and ***stwcx.***. A major weakness of a System/370-style Compare and Swap instruction is that, although the instruction itself is atomic, it checks only that the old and current values of the word being tested are equal, with the result that programs that use such a Compare and Swap to control a shared resource can err if the word has been modified and the old value subsequently restored. The sequence shown above has the same weakness.

3. In some applications the second ***bc*** instruction and/or the ***or*** instruction can be omitted. The ***bc*** is needed only if the application requires that if the EQ bit of CR Field 0 on exit indicates 'not equal' then GPR(r4) and GPR(r6) are in fact not equal. The ***or*** is needed only if the application requires that if the comparands are not equal then the word from storage is loaded into the register with which it was compared (rather than into a third register). If either or both of these instructions is omitted, the resulting Compare and Swap does not obey System/370 semantics.

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 storage location, the address of which is an argument of the 'lock' and 'unlock' procedures, given by GPR(3), 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 Section C.1.1, "Synchronization Primitives", on page 381, thereby atomically loading the old value of the lock, writing to the lock the new value (1) given in GPR(4), returning the old value in GPR(5) (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*** instruction after the ***bc*** that checks for the success of *test_and_set*. The ***isync*** instruction 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 inst'ns 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 a ***msync*** instruction before releasing the lock. The ***msync*** instruction ensures that the program's modifications will be 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 a ***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 the ***lwarx*** and ***stwcx.*** instructions can be used to implement simple insertion into a singly linked list. (Complicated list insertion, in which multiple values must be changed atomically, or in which the correct order of insertion depends on the contents of the elements, cannot be implemented in the manner shown below and requires a more complicated strategy such as using locks.)

The 'next element pointer' from the list element after which the new element is to be inserted, here called the 'parent element', is stored into the new element, so that the new element points to the next element in the list: this store is performed unconditionally. Then the address of the new element is conditionally stored into the parent element, thereby adding the new element to the list.

In this example it is assumed that the address of the parent element is in GPR(3), the address of the new element is in GPR(4), and the next element pointer is at offset 0 from the start of the element. It is also assumed that the next element pointer of each list element is in a 'reservation granule' separate from that of the next element pointer of all other list elements: see Section 6.1.6.2, "Atomic Update Primitives", on page 117.

```
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 it is not possible to allocate list elements such that each element's next element pointer is in a different reservation granule, then livelock can be avoided by using the following, more complicated, sequence.

```
       lwz    r2,0(r3)   #get next pointer
loop1: 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 Notes

1. In general, ***lwarx*** and ***stwcx.*** instructions 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*** instruction for which no ***stwcx.*** instruction 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 storage subsystem within a given processor (see Section 6.1.6.2, "Atomic Update Primitives", on page 117), is implementation-dependent. In some implementations performance may be improved by minimizing looping on a ***lwarx*** instruction that fails to return a desired value. For example, in the 'Test and Set' example shown above, if the programmer wishes to stay in the loop until the word loaded is zero, he could change the '*bne- $+12*' 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 storage area is in the reservation granule: see Section 6.1.6.2, "Atomic Update Primitives", on page 117. For example, the first code sequence shown in Section C.1.3, "List Insertion", on page 385 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-doubleword quantity (64-bit implementations) or 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 through 127 (64-bit implementations) or 0 through 63 (32-bit implementations), which are the maximum ranges supported by the *Shift* instructions used. However if N>2, the shift amount must be in the range 0 through 63 (64-bit implementations) or 0 through 31 (32-bit implementations), in order for the examples to yield the desired result. The specific instance shown for N>2 is N=3: extending those code sequences to larger N is straightforward, as is reducing them to the case N=2 when the more stringent restriction on shift amount is met. For shifts with immediate shift amounts only the case N=3 is shown, because the more stringent restriction on shift amount is always met.

In the examples it is assumed that GPRs 2 and 3 (and 4) contain the quantity to be shifted, and that the result is to be placed into the same registers, except for the immediate left shifts in 64-bit implementations, for which the result is placed into GPRs 3, 4, and 5. In all cases, for both input and result, the lowest-numbered register contains the highest-order part of the data and highest-numbered register contains the lowest-order part. For non-immediate shifts, the shift amount is assumed to be in GPR(6. For immediate shifts, the shift amount is assumed to be greater than 0. GPRs 0 and 31 are used as scratch registers.

For N>2, the number of instructions required is 2N–1 (immediate shifts) or 3N–1 (non-immediate shifts).

64-bit implementations	32-bit implementations
Shift Left Immediate, N=3 (shift amount < 64)	**Shift Left Immediate, N=3 (shift amount < 32)**
`rldicr r5,r4,sh,63-sh`	`rlwinm r2,r2,sh,0,31-sh`
`rldimi r4,r3,0,sh`	`rlwimi r2,r3,sh,32-sh,31`
`rldicl r4,r4,sh,0`	`rlwinm r3,r3,sh,0,31-sh`
`rldimi r3,r2,0,sh`	`rlwimi r3,r4,sh,32-sh,31`
`rldicl r3,r3,sh,0`	`rlwinm r4,r4,sh,0,31-sh`
Shift Left, N=2 (shift amount < 128)	**Shift Left, N=2 (shift amount < 64)**
`subfic r31,r6,64`	`subfic r31,r6,32`
`sld r2,r2,r6`	`slw r2,r2,r6`
`srd r0,r3,r31`	`srw r0,r3,r31`
`or r2,r2,r0`	`or r2,r2,r0`
`addi r31,r6,-64`	`addi r31,r6,-32`
`sld r0,r3,r31`	`slw r0,r3,r31`
`or r2,r2,r0`	`or r2,r2,r0`
`sld r3,r3,r6`	`slw r3,r3,r6`
Shift Left, N=3 (shift amount < 64)	**Shift Left, N=3 (shift amount < 32)**
`subfic r31,r6,64`	`subfic r31,r6,32`
`sld r2,r2,r6`	`slw r2,r2,r6`
`srd r0,r3,r31`	`srw r0,r3,r31`
`or r2,r2,r0`	`or r2,r2,r0`
`sld r3,r3,r6`	`slw r3,r3,r6`
`srd r0,r4,r31`	`srw r0,r4,r31`
`or r3,r3,r0`	`or r3,r3,r0`
`sld r4,r4,r6`	`slw r4,r4,r6`

64-bit implementations	32-bit implementations

Shift Right Immediate, N=3 (shift amount < 64)
`rldimi r4,r3,0,64-sh`
`rldicl r4,r4,64-sh,0`
`rldimi r3,r2,0,64-sh`
`rldicl r3,r3,64-sh,0`
`rldicl r2,r2,64-sh,sh`
	Shift Right 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 Right, N=2 (shift amount < 128)
`subfic r31,r6,64`
`srd r3,r3,r6`
`sld r0,r2,r31`
`or r3,r3,r0`
`addi r31,r6,-64`
`srd r0,r2,r31`
`or r3,r3,r0`
`srd r2,r2,r6`
	Shift Right, N=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 Right, N=3 (shift amount < 64)
`subfic r31,r6,64`
`srd r4,r4,r6`
`sld r0,r3,r31`
`or r4,r4,r0`
`srd r3,r3,r6`
`sld r0,r2,r31`
`or r3,r3,r0`
`srd r2,r2,r6`
	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 < 64)
`rldimi r4,r3,0,64-sh`
`rldicl r4,r4,64-sh,0`
`rldimi r3,r2,0,64-sh`
`rldicl r3,r3,64-sh,0`
`sradi r2,r2,sh`
	Shift Right Algebraic Immediate, N=3 (shift amnt < 32)
	`rlwinm r4,r4,32-sh,sh,31`
	`rlwimi r4,r3,32-sh,0,sh-1`
	`rlwinm r3,r3,32-sh,sh,31`
	`rlwimi r3,r2,32-sh,0,sh-1`
	`srawi r2,r2,sh`

Shift Right Algebraic, N=2 (shift amount < 128)
`subfic r31,r6,64`
`srd r3,r3,r6`
`sld r0,r2,r31`
`or r3,r3,r0`
`addic. r31,r6,-64`
`srad r0,r2,r31`
`bc 4,1,$+8`
`ori r3,r0,0`
`srad r2,r2,r6`
	Shift Right Algebraic, N=2 (shift amount < 64)
	`subfic r31,r6,32`
	`srw r3,r3,r6`
	`slw r0,r2,r31`
	`or r3,r3,r0`
	`addic. r31,r6,-32`
	`sraw r0,r2,r31`
	`bc 4,1,$+8`
	`ori r3,r0,0`
	`sraw r2,r2,r6`

Shift Right Algebraic, N=3 (shift amount < 64)
`subfic r31,r6,64`
`srd r4,r4,r6`
`sld r0,r3,r31`
`or r4,r4,r0`
`srd r3,r3,r6`
`sld r0,r2,r31`
`or r3,r3,r0`
`srad r2,r2,r6`
	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 the *Floating-Point Conversion* instructions can be used to perform various conversions.

Warning: 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: see Section C.4.4, “Notes”, on page 396.

C.3.1 Conversion from Floating-Point Number to Floating-Point Integer

The full *convert to floating-point integer* function can be implemented with the sequence shown below, assuming the floating-point value to be converted is in FPR(1) and the result is returned in FPR(3).

```
    mtfsb0   23            #clear VXCVI
    fctid[z] f3,f1         #convert to integer
    fcfid    f3,f3         #convert back again
    mcrfs    7,5           #VXCVI to CR
    bc       4,31,continue #skip if VXCVI was 0
    fmr      f3,f1         #input was fp integer
continue:
```

C.3.2 Conversion from Floating-Point Number to Signed Integer Doubleword

In a 64-bit implementation

The full *convert to signed integer doubleword* function can be implemented with the sequence shown below, assuming the floating-point value to be converted is in FPR(1), the result is returned in GPR(3), and a doubleword at displacement ‘disp’ from the address in GPR(1) can be used as scratch space.

```
    fctid[z] f2,f1         #convert to doubleword integer
    stfd     f2,disp(r1)   #store float
    ld       r3,disp(r1)   #load doubleword
```

In a 32-bit implementation

The full *convert to signed integer doubleword* function can be implemented with the sequence shown below, assuming the floating-point value to be converted is in FPR(1), bits 0:31 of the result are returned in GPR(3), bits 32:63 of the result are returned in GPR(4), and a doubleword at displacement 'disp' from the address in GPR(1) can be used as scratch space.

```
fctid[z] f2,f1         #convert to doubleword integer
stfd     f2,disp(r1)   #store float
lwz      r3,disp(r1)   #load upper half of doubleword
```

C.3.3 Conversion from Floating-Point Number to Unsigned Integer Doubleword

In a 64-bit implementation

The full *convert to unsigned integer doubleword* function can be implemented with the sequence shown below, assuming the floating-point value to be converted is in FPR(1), the value 0 is in FPR(0, the value 2^{64}–2048 is in FPR(3), the value 2^{63} is in FPR(4) and GPR(4), the result is returned in GPR(3), and a doubleword at displacement 'disp' from the address in GPR(1) can be used as scratch space.

```
fsel     f2,f1,f1,f0   #use 0 if < 0
fsub     f5,f3,f1      #use max if > max
fsel     f2,f5,f2,f3
fsub     f5,f2,f4      #subtract 2^63
fcmpu    cr2,f2,f4     #use diff if ≥ 2^63
fsel     f2,f5,f5,f2
fctid[z] f2,f2         #convert to integer
stfd     f2,disp(r1)   #store float
ld       r3,disp(r1)   #load doubleword
bc       12,8,$+8      #add 2^63 if input
add      r3,r3,r4      #  was ≥ 2^63
```

In a 32-bit implementation

Editors' Note
To be supplied.

C.3.4 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 FPR(1), the result is returned in GPR(3), and a doubleword at displacement 'disp' from the address in GPR(1) can be used as scratch space.

```
fctiw[z] f2,f1         #convert to integer
stfd     f2,disp(r1)   #store float
lwa      r3,disp+4(r1) #load word algebraic
                       #(use lwz on a 32-bit
                       #implementation)
```

C.3.5 Conversion from Floating-Point Number to Unsigned Integer Word

In a 64-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 FPR(1), the value 0 is in FPR(0, the value $2^{32}-1$ is in FPR(3), the result is returned in GPR(3), and a doubleword at displacement 'disp' from the address in GPR(1) can be used as scratch space.

```
fsel     f2,f1,f1,f0   #use 0 if < 0
fsub     f4,f3,f1      #use max if > max
fsel     f2,f4,f2,f3
fctid[z] f2,f2         #convert to integer
stfd     f2,disp(r1)   #store float
lwz      r3,disp+4(r1) #load word and zero
```

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 FPR(1), the value 0 is in FPR(0, the value $2^{32}-1$ is in FPR(3), the value 2^{31} is in FPR(4), the result is returned in GPR(3), and a doubleword at displacement 'disp' from the address in GPR(1) can be used as scratch space.

```
fsel     f2,f1,f1,f0   #use 0 if < 0
fsub     f5,f3,f1      #use max if > max
fsel     f2,f5,f2,f3
fsub     f5,f2,f4      #subtract 2^31
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.3.6 Conversion from Signed Integer Doubleword to Floating-Point Number

The full *convert from signed integer doubleword* function, using the rounding mode specified by $FPSCR_{RN}$, can be implemented with the sequence shown below, assuming the integer value to be converted is in GPR(3), the result is returned in FPR(1), and a doubleword at displacement 'disp' from the address in GPR(1) can be used as scratch space.

```
std      r3,disp(r1)   #store doubleword
lfd      f1,disp(r1)   #load float
fcfid    f1,f1         #convert to fp integer
```

C.3.7 Conversion from Unsigned Integer Doubleword to Floating-Point Number

In a 64-bit implementation

The full *convert from unsigned integer doubleword* function, using the rounding mode specified by $FPSCR_{RN}$, can be implemented with the sequence shown below, assuming the integer value to be converted is in GPR(3), the value 2^{32} is in FPR(4), the result is returned in FPR(1), and two doublewords at displacement 'disp' from the address in GPR(1) can be used as scratch space.

```
rldicl   r2,r3,32,32   #isolate high half
rldicl   r0,r3,0,32    #isolate low half
std      r2,disp(r1)   #store doubleword both
std      r0,disp+8(r1)
lfd      f2,disp(r1)   #load float both
lfd      f1,disp+8(r1)
fcfid    f2,f2         #convert each half to
fcfid    f1,f1         #  fp integer (exact result)
fmadd    f1,f4,f2,f1   #(2^32)*high + low
```

An alternative, shorter, sequence can be used if rounding according to $FSCPR_{RN}$ is desired and $FPSCR_{RN}$ specifies *Round toward +Infinity* or *Round toward -Infinity*, or if it is acceptable for the rounded answer to be either of the two representable floating-point integers nearest to the given integer integer. In this case the full *convert from unsigned integer doubleword* function can be implemented with the sequence shown below, assuming the value 2^{64} is in FPR(2).

```
std      r3,disp(r1)   #store doubleword
lfd      f1,disp(r1)   #load float
fcfid    f1,f1         #convert to fp integer
fadd     f4,f1,f2      #add 2^64
fsel     f1,f1,f1,f4   #  if r3 < 0
```

In a 32-bit implementation

The full *convert from unsigned integer doubleword* function, using the rounding mode specified by $FPSCR_{RN}$, can be implemented with the sequence shown below, assuming bits 0:31 of the doubleword integer value to be converted is in GPR(2), bits 32:63 of the doubleword integer value to be converted is in GPR(3), the value 0 is in GPR(0), the value 2^{32} is in FPR(4), the result is returned in FPR(1), and two doublewords at displacement 'disp' from the address in GPR(1) can be used as scratch space.

```
stw      r0,disp(r1)    #pad with 0s
stw      r2,disp+4(r1)  #store upper half of doubleword
stw      r0,disp+8(r1)  #pad with 0s
stw      r3,disp+12(r1) #store lower half of doubleword
lfd      f2,disp(r1)    #load float both
lfd      f1,disp+8(r1)
fcfid    f2,f2          #convert each half to
fcfid    f1,f1          #  fp integer (exact result)
fmadd    f1,f4,f2,f1    #(2^32)*high + low
```

An alternative, shorter, sequence can be used if rounding according to $FSCPR_{RN}$ is desired and $FPSCR_{RN}$ specifies *Round toward +Infinity* or *Round toward -Infin-*

ity, or if it is acceptable for the rounded answer to be either of the two representable floating-point integers nearest to the given integer integer. In this case the full *convert from unsigned integer doubleword* function can be implemented with the sequence shown below, assuming the value 2^{64} is in FPR(2).

```
stw      r2,disp(r1)    #store upper half of doubleword
stw      r3,disp+4(r1)  #store lower half of doubleword
lfd      f1,disp(r1)   #load float
fcfid    f1,f1         #convert to fp integer
fadd     f4,f1,f2      #add 2^64
fsel     f1,f1,f1,f4   #  if r3 < 0
```

C.3.8 Conversion from Signed Integer Word to Floating-Point Number

In a 64-bit implementation

The full *convert from signed integer word* function can be implemented with the sequence shown below, assuming the integer value to be converted is in GPR(3), the result is returned in FPR(1), and a doubleword at displacement 'disp' from the address in GPR(1) can be used as scratch space. (The result is exact.)

```
extsw    r3,r3         #extend sign
std      r3,disp(r1)   #store doubleword
lfd      f1,disp(r1)   #load float
fcfid    f1,f1         #convert to fp integer
```

In a 32-bit implementation

The full *convert from signed integer word* function can be implemented with the sequence shown below, assuming the integer value to be converted is in GPR(3), the result is returned in FPR(1), and a doubleword at displacement 'disp' from the address in GPR(1) can be used as scratch space. (The result is exact.)

```
sraw     r4,r3,32      #extract sign
stw     r4,disp(r1)   #store sign bits in upper half of dblword
stw      r3,disp+4(r1) #store lower half of doubleword
lfd      f1,disp(r1)   #load float
fcfid    f1,f1         #convert to fp integer
```

C.3.9 Conversion from Unsigned Integer Word to Floating-Point Number

In a 64-bit implementation

The full *convert from unsigned integer word* function can be implemented with the sequence shown below, assuming the integer value to be converted is in GPR(3), the result is returned in FPR(1), and a doubleword at displacement 'disp' from the address in GPR(1) can be used as scratch space. (The result is exact.)

```
rldicl   r0,r3,0,32    #zero-extend
std      r0,disp(r1)   #store doubleword
lfd      f1,disp(r1)   #load float
fcfid    f1,f1         #convert to fp integer
```

In a 32-bit implementation

The full *convert from unsigned integer word* function can be implemented with the sequence shown below, assuming the integer value to be converted is in GPR(3), a value of 0 is in GPR(0), the result is returned in FPR(1), and a doubleword at displacement 'disp' from the address in GPR(1) can be used as scratch space. (The result is exact.)

```
stw      r0,disp(r1)   #pad upper half of doubleword with 0s
stw      r2,disp(r1)   #store lower half of doubleword
lfd      f1,disp(r1)   #load float
fcfid    f1,f1         #convert to fp integer
```

C.4 Floating-Point Selection

This section gives examples of how the optional *Floating Select* instruction can be used to implement floating-point minimum and maximum functions, and certain simple forms of if-then-else constructions, without branching.

The examples show program fragments in an imaginary, C-like, high-level programming language, and the corresponding program fragment using ***fsel*** and other 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.

Additional examples can be found in Section C.3, “Floating-Point Conversions”, on page 389.

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.4.4, “Notes”, on page 396.

C.4.1 Comparison to Zero

High-level language:	Book E:	Notes
`if a ≥ 0.0 then x ← y` `else x ← z`	`fsel fx,fa,fy,fz`	(1)
`if a > 0.0 then x ← y` `else x ← z`	`fneg fs,fa` `fsel fx,fs,fz,fy`	(1,2)

`if a = 0.0 then x ← y`
`else x ← z`
	`fsel fx,fa,fy,fz`
	`fneg fs,fa`
	`fsel fx,fs,fx,fz`
		(1)

C.4.2 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)

C.4.3 Simple if-then-else Constructions

High-level language:	Book E:	Notes
`if a ≥ b then x ← y` `else x ← z`	`fsub fs,fa,fb` `fsel fx,fs,fy,fz`	(4,5)
`if a > b then x ← y` `else x ← z`	`fsub fs,fb,fa` `fsel fx,fs,fz,fy`	(3,4,5)

`if a = b then x ← y`
`else x ← z`
	`fsub fs,fa,fb`
	`fsel fx,fs,fy,fz`
	`fneg fs,fs`
	`fsel fx,fs,fx,fz`
		(4,5)

C.4.4 Notes

The following Notes apply to the preceding examples and to the corresponding cases using the other three arithmetic relations (<, ≤, and ≠). They should also be considered when any other use of ***fsel*** is contemplated.

In these Notes, the 'optimized program' is the 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 the VXSNAN bit of the FPSCR, and therefore may cause the system error handler to be invoked if the corresponding exception is enabled, while the optimized program does not affect this bit. This property of the optimized program is incompatible with the IEEE standard.

2. The optimized program gives the incorrect result if *a* is a NaN.

3. The optimized program gives the incorrect result if *a* and/or *b* is a NaN (except that it may give the correct result in some cases for the minimum and maximum functions, depending on how those functions are defined to operate on NaNs).

4. The optimized program gives the incorrect result if *a* and *b* are infinities of the same sign. (Here it is assumed that Invalid Operation Exceptions are disabled, in which case the result of the subtraction is a NaN. The analysis is more complicated if Invalid Operation Exceptions are enabled, because in that case the target register of the subtraction is unchanged.)

5. The optimized program affects the OX, UX, XX, and VXISI bits of the FPSCR, and therefore may cause the system error handler to be invoked if the corresponding exceptions are enabled, while the unoptimized program does not affect these bits. This property of the optimized program is incompatible with the IEEE standard.

Appendix D Controlling Storage Access Ordering

This appendix gives examples of how dependencies and the ***msync*** and ***mbar*** instructions can be used to control storage access ordering when storage is shared between programs.

D.1 Lock Acquisition and Import Barriers

An 'import barrier' is an instruction or sequence of instructions that prevents storage accesses caused by instructions following the barrier from being performed before storage accesses that acquire a lock have been performed. An import barrier can be used to ensure that a shared data structure protected by a lock is not accessed until the lock has been acquired. An ***msync*** instruction can always be used as an import barrier, but the approaches shown below will generally yield better performance because they order only the relevant storage accesses.

D.1.1 Acquire Lock and Import Shared Storage

If ***lwarx***[***e***] or ***ldarxe***, and ***stwcx***[***e***]**.** or ***stdcxe.*** instructions are used to obtain the lock, an import barrier can be constructed by placing an ***isync*** instruction immediately following the loop containing the ***lwarx***[***e***] or ***ldarxe***, and ***stwcx***[***e***]**.** or ***stdcxe.***. The following example uses the 'Compare and Swap' primitive (see the section entitled 'Synchronization Primitives' in Section I) to acquire the lock.

In this example it is assumed that the address of the lock is in GPR 3, the value indicating that the lock is free is in GPR 4, the value to which the lock should be set is in GPR 5, the old value of the lock is returned in GPR 6, and the address of the shared data structure is in GPR 9.

```
loop: lwarx  r6,0,r3        # load lock and reserve
      cmp    cr0,0,r4,r6    # skip ahead if
      bc     4,2,wait       #   lock not free
      stwcx. r5,0,r3        # try to set lock
      bc     4,2,loop       # loop if lost reservation
      isync                 # import barrier
      lwz    r7,data1(r9)   # load shared data
      .
      .
wait: ...                   # wait for lock to free
```

The second ***bc*** does not complete until CR0 has been set by the ***stwcx***[***e***]**.** or ***stdcxe.***. The ***stwcx***[***e***]**.** or ***stdcxe.*** does not set CR0 until it has completed (successfully or unsuccessfully). The lock is acquired when the ***stwcx***[***e***]**.** or ***stdcxe.*** completes successfully. Together, the second ***bc*** and the subsequent ***isync*** create an import barrier that prevents the load from 'data1' from being performed until the branch has been resolved not to be taken.

D.1.2 Obtain Pointer and Import Shared Storage

If ***lwarx***[***e***] or ***ldarxe*** and ***stwcx***[***e***]**.** or ***stdcxe.*** instructions are used to obtain a pointer into a shared data structure, an import barrier is not needed if all the accesses to the shared data structure depend on the value obtained for the pointer. The following example uses the 'Fetch and Add' primitive (see the section entitled 'Synchronization Primitives' in Section I) 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
      bc     4,2,loop       # loop if lost reservation
      lwz    r7,data1(r5)   # load shared data
```

The load from 'data1' cannot be performed until the pointer value has been loaded into GPR 5 by the ***lwarx***[***e***] or ***ldarxe***. The load from 'data1' may be performed out-of-order before the ***stwcx***[***e***]**.** or ***stdcxe.***. But if the ***stwcx***[***e***]**.** or ***stdcxe.*** fails, the branch is taken and the value returned by the load from 'data1' is discarded. If the ***stwcx***[***e***]**.** or ***stdcxe.*** 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***[***e***] or ***ldarxe*** that created the reservation used by the successful ***stwcx***[***e***]**.** or ***stdcxe.***.

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***[***e***] or ***ldarxe***.

D.2 Lock Release and Export Barriers

An 'export barrier' is an instruction or sequence of instructions that prevents the store that releases a lock from being performed before stores caused by instructions preceding the barrier have been performed. An export barrier can be used to ensure that all stores to a shared data structure protected by a lock will be performed with respect to any other processor (to the extent required by the associated Memory Coherence Required attributes) before the store that releases the lock is performed with respect to that processor.

D.2.1 Export Shared Storage and Release Lock

An ***msync*** instruction can always be used as an export barrier, independent of the storage control attributes (e.g., presence or absence of the Caching Inhibited attribute) of the storage containing the lock and the shared data structure. Unless both the lock and the shared data structure are in storage that is neither Caching Inhibited nor Write Through Required, an ***msync*** instruction *must* be used as the export barrier.

In this example it is assumed that the lock is in storage that is Caching Inhibited, the shared data structure is in storage that is not Caching Inhibited, the address of the lock is in GPR 3, the value indicating that the lock is free is in GPR 4, and the address of the shared data structure is in GPR 9.

```
stw   r7,data1(r9)   # store shared data (last)
msync                # export barrier
stw   r4,lock(r3)    # release lock
```

The ***msync*** ensures that the store that releases the lock will not be performed with respect to any other processor until all stores caused by instructions preceding the ***msync*** have been performed with respect to that processor.

D.2.2 Export Shared Storage and Release Lock using mbar

If both the lock and the shared data structure are in storage that is neither Caching Inhibited nor Write Through Required, an ***mbar*** instruction can be used as the export barrier. Using ***mbar*** rather than ***msync*** will yield better performance in most systems.

In this example it is assumed that both the lock and the shared data structure are in storage that is neither Caching Inhibited nor Write Through Required, the address of the lock is in GPR 3, the value indicating that the lock is free is in GPR 4, and the address of the shared data structure is in GPR 9.

```
stw   r7,data1(r9)   #store shared data (last)
mbar                 #export barrier
stw   r4,lock(r3)    #release lock
```

The ***mbar*** ensures that the store that releases the lock will not be performed with respect to any other processor until all stores caused by instructions preceding the ***mbar*** have been performed with respect to that processor.

Recall that, for storage that is neither Caching Inhibited nor Write Through Required, ***mbar*** orders only stores and has no effect on loads. If the portion of the program preceding the ***mbar*** contains loads from the shared data structure and the stores to the shared data structure do not depend on the values returned by those loads, the store that releases the lock could be performed before those loads. If it is necessary to ensure that those loads are performed before the store that releases the lock, the programmer can either use the ***msync*** instruction as in Section D.2.1 or use the technique described in Section D.3.

D.3 Safe Fetch

If a load must be performed before a subsequent store (e.g., the store that releases a lock protecting a shared data structure), a technique similar to the following can be used.

In this example it is assumed that the address of the storage operand to be loaded is in GPR 3, the contents of the storage operand are returned in GPR 4, and the address of the storage operand to be stored is in GPR 5.

```
lwz     r4,0(r3)      #load shared data
cmp     cr0,0,r4,r4   #set CR0 to 'equal'
bc      4,2,$-8       #branch never taken
stw     r7,0(r5)      #store other shared data
```

Alternatively, a technique similar to that described in Section D.1.2 can be used, by causing the ***stw*** to depend on the value returned by the ***lwz*** and omitting the ***cmp*** and ***bc***. The dependency could be created by ANDing the value returned by the ***lwz*** with zero and then adding the result to the value to be stored by the ***stw***.

Appendix E Processor Simplifications for Uniprocessor Designs

Microprocessor designs that will not be used in symmetric multiprocessor (SMP) systems may adopt optimizations to avoid cost and design effort implementing functions that will never be used. Further optimizations may be adopted if the design will never be used in conjunction with an L2 cache.

The following list identifies the areas in which these optimizations can be made.

1. Receipt of TLB entry invalidate requests from other processors. Since the design will not be used in SMP systems, this function is not required.

2. Communication of ***msync*** to external mechanisms. The function provided by the ***msync*** instruction can be completed by the processor with no need to communicate with external mechanisms.

    a) Does the design of any storage subsystem require notification that an ***msync*** is being executed?

    b) Does the design of any graphics subsystem require notification that an ***msync*** is being executed?

    c) Does the design of any I/O subsystem require notification that an ***msync*** is being executed?

3. Communication of ***mbar*** to external mechanisms. The function provided by the ***mbar*** instruction can be completed by the processor with no need to communicate with external mechanisms. It is assumed that no L2 cache is used or that its operation is totally transparent, and that all other mechanisms perform storage operations in the order that they are received.

4. Communication of cache management operations to external caches. It is assumed that no L2 cache is used or that its operation is totally transparent. The function of these instructions can be completed in the processor with no need to communicate with external mechanisms.

5. Communication of TLB invalidates to external mechanisms. Graphics subsystem device drivers that use the move virtual storage instructions may

require notification of a TLB invalidation.

Architecture Note
There is a pending proposal for these functions, so this requirement is dependent on the resolution of that proposal.

Appendix F Reserved, Preserved, and Allocated Instructions

F.1 Preserved Instructions

Preserved instructions are provided by Book E to allow implementations to continue supporting PowerPC legacy software. Book E does not require these instructions be implemented but, instead, preserves the architectural resources that these instructions employed in the PowerPC Architecture. If they are not implemented, these instructions are treated the same as Reserved instructions. Opcodes for current preserved instructions will be among the last to be assigned a meaning in Book E. Preserved opcodes are listed in Table F-1.

Table F-1. 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

F.2 Allocated Instructions

Book E sets aside a set of Allocated opcodes for implementation-dependent use that is outside the scope of the architecture. The following blocks of opcodes are Allocated for implementation-dependent use. Some Allocated opcodes show example of current usage.

Table F-2. Allocated Instructions

Primary Opcode	Extended Opcodes
0	All instruction encodings (bits 6:31) except 0x0000_0000[a]
4	All instruction encodings (bits 6:31)
19	Extended opcodes (bits 21:30) --- 0buuuuu_0u11u
31	Extended opcodes (bits 21:30) --- 0buuuuu_0u11u 342 0b01010_10110 (VMX ***dst***) 374 0b01011_10110 (VMX ***dstst***) 822 0b11001_10110 (VMX ***dss***)
59	Extended opcodes (bits 21:30) --- 0buuuuu_0u10u
63	Extended opcodes (bits 21:30) --- 0buuuuu_0u10u (except 0x00000_01100 ***frsp***)

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

F.3 Reserved Instructions

With the exception of the instruction consisting entirely of binary 0s, the reserved instructions are available for future extensions to Book E: that is, some future version of Book E may define any of these instructions to perform new functions. There are two form of reserved instructions, reserved-nop and reserved-illegal instructions.

F.3.1 Reserved-Nop Instructions

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

When an attempt is made to execute a reserved-nop instruction, either no operation or effect occurs, or an Illegal Instruction exception type Program interrupt occurs. However, Book E strongly recommends the former rather than the latter implementation for reserved-nop instructions. Reserved-nop instructions include the following extended opcodes under primary opcode 31: 530, 562, 594, 626, 658, 690, 722, and 754.

F.3.2 Reserved-Illegal Instructions

Primary opcodes 1, 5, 6, 56, 57, 60, and 61 and extended opcodes under primary opcodes 19, 30, 31, 58, 59, 62, and 63 that are not classified as defined (listed in Section H, “Instruction Index”, on page 419), reserved-nop (listed in Section F.3.1), preserved (listed in Section F.1) nor allocated (listed in Section F.2) are considered reserved-illegal by the architecture.

Also, an instruction consisting entirely of binary 0s is reserved-illegal, and is guaranteed to be reserved-illegal in all future versions of this architecture. Attempt to execute this instruction or any other reserved-illegal instruction will cause an Illegal Instruction exception type Program interrupt.

Appendix G Opcode Maps

This section contains tables showing the Defined primary and extended opcodes in all members of the Book E family.

For all opcode tables, each cell is in the following format.

Opcode in Decimal	Opcode in Hexadecimal
Instruction Mnemonic	
Applicable Machines	Instruction Format

'Applicable Machines' identifies the Book E family members that recognize the opcode, encoded as follows:

E Book E only
P PowerPC Architecture 'classic' (retained in Book E[1])

When instruction names and/or mnemonics differ among the family members, Book E terminology is used.

Shaded boxes labeled as '<allocated>', '<preserved>', and '<nop>' represent allocated, preserved, and reserved-nop opcodes, respectively. Shaded boxes that are otherwise not labeled other than indicating the opcodes' numerical value represent reserved-illegal opcodes.

1. PowerPC Architecture "classic" instructions not retained in Book E are considered preserved.

Table G-1. Primary opcodes (instruction bits 0:5)

0 00 <allocated>	1 01	3 03 ***twi*** P D	2 02 ***tdi*** P D	<Allocated for implementation-dependent use> <Reserved> *Trap Word Immediate* *Trap Doubleword Immediate*
4 04 <allocated>	5 05	7 07 ***mulli*** P D	6 06	<Allocated for implementation-dependent use> <Reserved> *Multiply Low Immediate* <Reserved>
12 0C ***addic*** P D	13 0D ***addic.*** P D	15 0F ***addis*** P D	14 0E ***addi*** P D	*Add Immediate Carrying* *Add Immediate Carrying and Record* *Add Immediate Shifted* *Add Immediate*
8 08 ***subfic*** P D	9 09 ***bce***[***l***][***a***] E B	11 0B ***cmpi*** P D	10 0A ***cmpli*** P D	*Subtract From Immediate Carrying* *Branch Conditional Extended [& Link] [Absolute]* *Compare Immediate* *Compare Logical Immediate*
24 18 ***ori*** P D	25 19 ***oris*** P D	27 1B ***xoris*** P D	26 1A ***xori*** P	*OR Immediate* *OR Immediate Shifted* *XOR Immediate Shifted* *XOR Immediate*
28 1C ***andi*** P D	29 1D ***andis*** P D	31 1F Integer Extended P	30 1E Dwd Rotate Extended E	*AND Immediate* *AND Immediate Shifted* See Table G-6 on page 413 See Table G-2 on page 409
20 14 ***rlwimi*** P M	21 15 ***rlwinm*** P M	23 17 ***rlwnm*** P M	22 16 ***be***[***l***][***a***] E I	*Rotate Left Word Imm. then Mask Insert* *Rotate Left Word Imm. then AND with Mask* *Rotate Left Word then AND with Mask* *Branch Extended [& Link] [Absolute]*
16 10 ***bc***[***l***][***a***] P B	17 11 ***sc*** P SC	19 13 Branch Extended P XL	18 12 ***b***[***l***][***a***] P I	*Branch Conditional [& Link] [Absolute]* *System Call* See Table G-5 on page 410 *Branch [& Link] [Absolute]*
48 30 ***lfs*** P D	49 31 ***lfsu*** P D	51 33 ***lfdu*** P D	50 32 ***lfd*** P D	*Load Floating-Point Single* *Load Floating-Point Single with Update* *Load Floating-Point Double with Update* *Load Floating-Point Double*
52 34 ***stfs*** P D	53 35 ***stfsu*** P D	55 37 ***stfdu*** P D	54 36 ***stfd*** P D	*Store Floating-Point Single* *Store Floating-Point Single with Update* *Store Floating-Point Double with Update* *Store Floating-Point Double*
60 3C	61 3D	63 3F DP FP Extended P A/X	62 3E Load/Store Extended E DS	<Reserved> <Reserved> See Table G-8 on page 417 See Table G-4 on page 409
56 38	57 39	59 3B SP FP Extended P A/X	58 3A Load/Store Extended E DS	<Reserved> <Reserved> See Table G-7 on page 415 See Table G-3 on page 409
40 28 ***lhz*** P D	41 29 ***lhzu*** P D	43 2B ***lhau*** P D	42 2A ***lha*** P D	*Load Half and Zero* *Load Half and Zero with Update* *Load Half Algebraic with Update* *Load Half Algebraic*
44 2C ***sth*** P D	45 2D ***sthu*** P D	47 2F ***stmw*** P D	46 2E ***lmw*** P D	*Store Half* *Store Half with Update* *Store Multiple Word* *Load Multiple Word*
36 24 ***stw*** P D	37 25 ***stwu*** P D	39 27 ***stbu*** P D	38 26 ***stb*** P D	*Store Word* *Store Word with Update* *Store Byte with Update* *Store Byte*
32 20 ***lwz*** P D	33 21 ***lwzu*** P D	35 23 ***lbzu*** P D	34 22 ***lbz*** P D	*Load Word and Zero* *Load Word and Zero with Update* *Load Byte and Zero with Update* *Load Byte and Zero*

Table G-2. Extended opcodes for primary opcode 30 (instruction bits 27:30)

0-1 00-01 ***rldicl*** P MD		2-3 02-03 ***rldicr*** P MD		*Rotate Left Doubleword Immediate then Clear Left* <Reserved> *Rotate Left Doubleword Immediate then Clear Right* <Reserved>
4-5 04-05 ***rldic*** P		6-7 06-07 ***rldimi*** P		*Rotate Left Doubleword Immediate then Clear* <Reserved> *Rotate Left Doubleword Immediate then Mask Insert* <Reserved>
8 08 ***rldcl*** P MD	9 09 ***rldcr*** P MD	10 0A	11 0B	*Rotate Left Doubleword then Clear Left* *Rotate Left Doubleword then Clear Right* <Reserved> <Reserved>
12 0C	13 0D	14 0E	15 0F	<Reserved> <Reserved> <Reserved> <Reserved>

Table G-3. Extended opcodes for primary opcode 58 (instruction bits 28:31)

0 00 ***lbze*** E DS	1 01 ***lbzue*** E DS	3 03 ***lhzue*** E DS	2 02 ***lhze*** E DS	*Load Byte and Zero Extended* *Load Byte and Zero with Update Extended* *Load Halfword and Zero with Update Extended* *Load Halfword and Zero Extended*
4 04 ***lhae*** E DS	5 05 ***lhaue*** E DS	7 07 ***lwzue*** E DS	6 06 ***lwze*** E DS	*Load Halfword Algebraic Extended* *Load Halfword Algebraic with Update Extended* *Load Word and Zero with Update Extended* *Load Word and Zero Extended*
12 0C	13 0D	15 0F ***stwue*** E DS	14 0E ***stwe*** E DS	<Reserved> <Reserved> *Store Word with Update Extended* *Store Word Extended*
8 08 ***stbe*** E DS	9 09 ***stbue*** E DS	11 0B ***sthue*** E DS	10 0A ***sthe*** E DS	*Store Byte Extended* *Store Byte with Update Extended* *Store Halfword with Update Extended* *Store Halfword Extended*

Table G-4. Extended opcodes for primary opcode 62 (instruction bits 28:31)

0 00 ***lde*** E DS	1 01 ***ldue*** E DS	3 03	2 02	*Load Doubleword Extended* *Load Doubleword with Update Extended* <Reserved> <Reserved>
4 04 ***lfse*** E DS	5 05 ***lfsue*** E DS	7 07 ***lfdue*** E DS	6 06 ***lfde*** E DS	*Load Float Single-precision Extended* *Load Float Single-precision with Update Extended* *Load Float Double-precision with Update Extended* *Load Float Double-precision Extended*
12 0C ***stfse*** E DS	13 0D ***stfsue*** E DS	15 0F ***stfdue*** E DS	14 0E ***stfde*** E DS	*Store Float Single-precision Extended* *Store Float Single-precision with Update Extended* *Store Float Double-precision with Update Extended* *Store Float Double-precision Extended*
8 08 ***stde*** E DS	9 09 ***stdue*** E DS	11 0B	10 0A	*Store Doubleword Extended* *Store Doubleword Extended with Update* <Reserved> <Reserved>

Table G-5. Extended opcodes for primary opcode 19 (instruction bits 21:30) (part 1 of 2)

	00000	00001	00011	00010	00110	00111	00101	00100	01100	01101	01111	01110	01010	01011	01001	01000
00000	0 000 **mcrf** P XL	1 001	3 003	2 002	6 006 <allocated>	7 007 <allocated>	5 005	4 004	12 00C	13 00D	15 00F <allocated>	14 00E <allocated>	10 00A	11 00B	9 009	8 008
00001	32 020	33 021 **crnor** P XL	35 023	34 022	38 026 <allocated>	39 027 <allocated>	37 025	36 024	44 02C	45 02D	47 02F <allocated>	46 02E <allocated>	42 02A	43 02B	41 029	40 028
00011	96 060	97 061	99 063	98 062	102 066 <allocated>	103 067 <allocated>	101 065	100 064	108 06C	109 06D	111 06F <allocated>	110 06E <allocated>	106 06A	107 06B	105 069	104 068
00010	64 040	65 041	67 043	66 042	70 046 <allocated>	71 047 <allocated>	69 045	68 044	76 04C	77 04D	79 04F <allocated>	78 04E <allocated>	74 04A	75 04B	73 049	72 048
00110	192 0C0	193 0C1 **crxor** P XL	195 0C3	194 0C2	198 0C6 <allocated>	199 0C7 <allocated>	197 0C5	196 0C4	204 0CC	205 0CD	207 0CF <allocated>	206 0CE <allocated>	202 0CA	203 0CB	201 0C9	200 0C8
00111	224 0E0	225 0E1 **crnand** P XL	227 0E3	226 0E2	230 0E6 <allocated>	231 0E7 <allocated>	229 0E5	228 0E4	236 0EC	237 0ED	239 0EF <allocated>	238 0EE <allocated>	234 0EA	235 0EB	233 0E9	232 0E8
00101	160 0A0	161 0A1	163 0A3	162 0A2	166 0A6 <allocated>	167 0A7 <allocated>	165 0A5	164 0A4	172 0AC	173 0AD	175 0AF <allocated>	174 0AE <allocated>	170 0AA	171 0AB	169 0A9	168 0A8
00100	128 080	129 081 **crandc** P XL	131 083	130 082	134 086 <allocated>	135 087 <allocated>	133 085	132 084	140 08C	141 08D	143 08F <allocated>	142 08E <allocated>	138 08A	139 08B	137 089	136 088
01100	384 180	385 181	387 183	386 182	390 186 <allocated>	391 187 <allocated>	389 185	388 184	396 18C	397 18D	399 18F <allocated>	398 18E <allocated>	394 18A	395 18B	393 189	392 188
01101	416 1A0	417 1A1 **crorc** P XL	419 1A3	418 1A2	422 1A6 <allocated>	423 1A7 <allocated>	421 1A5	420 1A4	428 1AC	429 1AD	431 1AF <allocated>	430 1AE <allocated>	426 1AA	427 1AB	425 1A9	424 1A8
01111	480 1E0	481 1E1	483 1E3	482 1E2	486 1E6 <allocated>	487 1E7 <allocated>	485 1E5	484 1E4	492 1EC	493 1ED	495 1EF <allocated>	494 1EE <allocated>	490 1EA	491 1EB	489 1E9	488 1E8
01110	448 1C0	449 1C1 **cror** P XL	451 1C3	450 1C2	454 1C6 <allocated>	455 1C7 <allocated>	453 1C5	452 1C4	460 1CC	461 1CD	463 1CF <allocated>	462 1CE <allocated>	458 1CA	459 1CB	457 1C9	456 1C8
01010	320 140	321 141	323 143	322 142	326 146 <allocated>	327 147 <allocated>	325 145	324 144	332 14C	333 14D	335 14F <allocated>	334 14E <allocated>	330 14A	331 14B	329 149	328 148
01011	352 160	353 161	355 163	354 162	358 166 <allocated>	359 167 <allocated>	357 165	356 164	364 16C	365 16D	367 16F <allocated>	366 16E <allocated>	362 16A	363 16B	361 169	360 168
01001	288 120	289 121 **creqv** P XL	291 123	290 122	294 126 <allocated>	295 127 <allocated>	293 125	292 124	300 12C	301 12D	303 12F <allocated>	302 12E <allocated>	298 12A	299 12B	297 129	296 128
01000	256 100	257 101 **crand** P XL	259 103	258 102	262 106 <allocated>	263 107 <allocated>	261 105	260 104	268 10C	269 10D	271 10F <allocated>	270 10E <allocated>	266 10A	267 10B	265 109	264 108
11000	768 300	769 301	771 303	770 302	774 306 <allocated>	775 307 <allocated>	773 305	772 304	780 30C	781 30D	783 30F <allocated>	782 30E <allocated>	778 30A	779 30B	777 309	776 308
11001	800 320	801 321	803 323	802 322	806 326 <allocated>	807 327 <allocated>	805 325	804 324	812 32C	813 32D	815 32F <allocated>	814 32E <allocated>	810 32A	811 32B	809 329	808 328
11011	864 360	865 361	867 363	866 362	870 366 <allocated>	871 367 <allocated>	869 365	868 364	876 36C	877 36D	879 36F <allocated>	878 36E <allocated>	874 36A	875 36B	873 369	872 368
11010	832 340	833 341	835 343	834 342	838 346 <allocated>	839 347 <allocated>	837 345	836 344	844 34C	845 34D	847 34F <allocated>	846 34E <allocated>	842 34A	843 34B	841 349	840 348
11110	960 3C0	961 3C1	963 3C3	962 3C2	966 3C6 <allocated>	967 3C7 <allocated>	965 3C5	964 3C4	972 3CC	973 3CD	975 3CF <allocated>	974 3CE <allocated>	970 3CA	971 3CB	969 3C9	968 3C8
11111	992 3E0	993 3E1	995 3E3	994 3E2	998 3E6 <allocated>	999 3E7 <allocated>	997 3E5	996 3E4	1004 3EC	1005 3ED	1007 3EF <allocated>	1006 3EE <allocated>	1002 3EA	1003 3EB	1001 3E9	1000 3E8
11101	928 3A0	929 3A1	931 3A3	930 3A2	934 3A6 <allocated>	935 3A7 <allocated>	933 3A5	932 3A4	940 3AC	941 3AD	943 3AF <allocated>	942 3AE <allocated>	938 3AA	939 3AB	937 3A9	936 3A8
11100	896 380	897 381	899 383	798 382	902 386 <allocated>	903 387 <allocated>	901 385	900 384	908 38C	909 38D	911 38F <allocated>	910 38E <allocated>	906 38A	907 38B	905 389	904 388
10100	640 280	641 281	643 283	642 282	646 286 <allocated>	647 287 <allocated>	645 285	644 284	652 28C	653 28D	655 28F <allocated>	654 28E <allocated>	650 28A	651 28B	649 289	648 288
10101	672 2A0	673 2A1	675 2A3	674 2A2	678 2A6 <allocated>	679 2A7 <allocated>	677 2A5	676 2A4	684 2AC	685 2AD	687 2AF <allocated>	686 2AE <allocated>	682 2AA	683 2AB	681 2A9	680 2A8
10111	736 2E0	737 2E1	739 2E3	738 2E2	742 2E6 <allocated>	743 2E7 <allocated>	741 2E5	740 2E4	748 2EC	749 2ED	751 2EF <allocated>	750 2EE <allocated>	746 2EA	747 2EB	745 2E9	744 2E8
10110	704 2C0	705 2C1	707 2C3	706 2C2	710 2C6 <allocated>	711 2C7 <allocated>	709 2C5	708 2C4	716 2CC	717 2CD	719 2CF <allocated>	718 2CE <allocated>	714 2CA	715 2CB	713 2C9	712 2C8
10010	576 240	577 241	579 243	578 242	582 246 <allocated>	583 247 <allocated>	581 245	580 244	588 24C	589 24D	591 24F <allocated>	590 24E <allocated>	586 24A	587 24B	585 249	584 248
10011	608 260	609 261	611 263	610 262	614 266 <allocated>	615 267 <allocated>	613 265	612 264	620 26C	621 26D	623 26F <allocated>	622 26E <allocated>	618 26A	619 26B	617 269	616 268
10001	544 220	545 221	547 223	546 222	550 226 <allocated>	551 227 <allocated>	549 225	548 224	556 22C	557 22D	559 22F <allocated>	558 22E <allocated>	554 22A	555 22B	553 229	552 228
10000	512 200	513 201	515 203	514 202	518 206 <allocated>	519 207 <allocated>	517 205	516 204	524 20C	525 20D	527 20F <allocated>	526 20E <allocated>	522 20A	523 20B	521 209	520 208

Table G-5. Extended opcodes for primary opcode 19 (instruction bits 21:30) (part 2 of 2)

11000	11001	11011	11010	11110	11111	11101	11100	10100	10101	10111	10110	10010	10011	10001	10000	
24 018	25 019	27 01B	26 01A	30 01E	31 01F	29 01D	28 01C	20 014	21 015	23 017	22 016	18 012	19 013	17 011 **bclre** E XL	16 010 **bclr** P XL	00000
56 038	57 039	59 03B	58 03A	62 03E	63 03F	61 03D	60 03C	52 034	53 035	55 037	54 036	50 032 **rfi** P XL	51 033 **rfci** E XL	49 031	48 030	00001
120 078	121 079	123 07B	122 07A	126 07E	127 07F	125 07D	124 07C	116 074	117 075	119 077	118 076	114 072	115 073	113 071	112 070	00011
88 058	89 059	91 05B	90 05A	94 05E	95 05F	93 05D	92 05C	84 054	85 055	87 057	86 056	82 052	83 053	81 051	80 050	00010
216 0D8	217 0D9	219 0DB	218 0DA	222 0DE	223 0DF	221 0DD	220 0DC	212 0D4	213 0D5	215 0D7	214 0D6	210 0D2	211 0D3	209 0D1	208 0D0	00110
248 0F8	249 0F9	251 0FB	250 0FA	254 0FE	255 0FF	253 0FD	252 0FC	244 0F4	245 0F5	247 0F7	246 0F6	242 0F2	243 0F3	241 0F1	240 0F0	00111
184 0B8	185 0B9	187 0BB	186 0BA	190 0BE	191 0BF	189 0BD	188 0BC	180 0B4	181 0B5	183 0B7	182 0B6	178 0B2	179 0B3	177 0B1	176 0B0	00101
152 098	153 099	155 09B	154 09A	158 09E	159 09F	157 09D	156 09C	148 094	149 095	151 097	150 096 **isync** P XL	146 092	147 093	145 091	144 090	00100
408 198	409 199	411 19B	410 19A	414 19E	415 19F	413 19D	412 19C	404 194	405 195	407 197	406 196	402 192	403 193	401 191	400 190	01100
450 1B8	451 1B9	453 1BB	452 1BA	456 1BE	457 1BF	455 1BD	454 1BC	436 1B4	437 1B5	439 1B7	438 1B6	434 1B2	435 1B3	433 1B1	432 1B0	01101
504 1F8	505 1F9	507 1FB	506 1FA	510 1FE	511 1FF	509 1FD	508 1FC	500 1F4	501 1F5	503 1F7	502 1F6	498 1F2	499 1F3	497 1F1	496 1F0	01111
472 1D8	473 1D9	475 1DB	474 1DA	478 1DE	479 1DF	477 1DD	476 1DC	468 1D4	469 1D5	471 1D7	470 1D6	466 1D2	467 1D3	465 1D1	464 1D0	01110
344 158	345 159	347 15B	346 15A	350 15E	351 15F	349 15D	348 15C	340 154	341 155	343 157	342 156	338 152	339 153	337 151	336 150	01010
376 178	377 179	379 17B	378 17A	382 17E	383 17F	381 17D	380 17C	372 174	373 175	375 177	374 176	370 172	371 173	369 171	368 170	01011
312 138	313 139	315 13B	314 13A	318 13E	319 13F	317 13D	316 13C	308 134	309 135	311 137	310 136	306 132	307 133	305 131	304 130	01001
280 118	281 119	283 11B	282 11A	286 11E	287 11F	285 11D	284 11C	276 114	277 115	279 117	278 116	274 112	275 113	273 111	272 110	01000
792 318	793 319	795 31B	794 31A	798 31E	799 31F	797 31D	796 31C	788 314	789 315	791 317	790 316	786 312	787 313	785 311	784 310	11000
824 338	825 339	827 33B	826 33A	830 33E	831 33F	829 33D	828 33C	820 334	821 335	823 337	822 336	818 332	819 333	817 331	816 330	11001
888 378	889 379	891 37B	890 37A	894 37E	895 37F	893 37D	892 37C	884 374	885 375	887 377	886 376	882 372	883 373	881 371	880 370	11011
856 358	857 359	859 35B	858 35A	862 35E	863 35F	861 35D	860 35C	852 354	853 355	855 357	854 356	850 352	851 353	849 351	848 350	11010
984 3D8	985 3D9	987 3DB	986 3DA	990 3DE	991 3DF	989 3DD	988 3DC	980 3D4	981 3D5	983 3D7	982 3D6	978 3D2	979 3D3	977 3D1	976 3D0	11110
1016 3F8	1017 3F9	1019 3FB	1018 3FA	1022 3FE	1023 3FF	1021 3FD	1020 3FC	1012 3F4	1013 3F5	1015 3F7	1014 3F6	1010 3F2	1011 3F3	1009 3F1	1008 3F0	11111
952 3B8	953 3B9	955 3BB	954 3BA	958 3BE	959 3BF	957 3BD	956 3BC	948 3B4	949 3B5	951 3B7	950 3B6	946 3B2	947 3B3	945 3B1	944 3B0	11101
920 398	921 399	923 39B	922 39A	926 39E	927 39F	925 39D	924 39C	916 394	917 395	919 397	918 396	914 392	915 393	913 391	912 390	11100
664 298	665 299	667 29B	666 29A	670 29E	671 29F	669 29D	668 29C	660 294	661 295	663 297	662 296	658 292	659 293	657 291	656 290	10100
696 2B8	697 2B9	699 2BB	698 2BA	702 2BE	703 2BF P A	701 2BD	700 2BC	692 2B4	693 2B5	695 2B7	694 2B6	690 2B2	691 2B3	689 2B1	688 2B0	10101
760 2F8	761 2F9	763 2FB	762 2FA	766 2FE	767 2FF	765 2FD	764 2FC	756 2F4	757 2F5	759 2F7	758 2F6	754 2F2	755 2F3	753 2F1	752 2F0	10111
728 2D8	729 2D9	731 2DB	730 2DA	734 2DE	735 2DF	733 2DD	732 2DC	724 2D4	725 2D5	727 2D7	726 2D6	722 2D2	723 2D3	721 2D1	720 2D0	10110
600 258	601 259	603 25B	602 25A	606 25E	607 25F	605 25D	604 25C	596 254	597 255	599 257	598 256	594 252	595 253	593 251	592 250	10010
632 278	633 279	635 27B	634 27A	638 27E	639 27F	637 27D	636 27C	628 274	629 275	631 277	630 276	626 272	627 273	625 271	624 270	10011
568 238	569 239	571 23B	570 23A	574 23E	575 23F	573 23D	572 23C	564 234	565 235	567 237	566 236	562 232	563 233	561 231	560 230	10001
536 218	537 219	539 21B	538 21A	542 21E	543 21F	541 21D	540 21C	532 214	533 215	535 217	534 216	530 212	531 213	529 211 **bcctre** E XL	528 210 **bcctr** P XL	10000

Table G-6. Extended opcodes for primary opcode 31 (instruction bits 21:30) (part 1 of 2)

	00000	00001	00011	00010	00110	00111	00101	00100	01100	01101	01111	01110	01010	01011	01001	01000
00000	0 000 **cmp** P X	1 001	3 003	2 002	6 006 <allocated>	7 007 <allocated>	5 005	4 004 **tw** P X	12 00C	13 00D	15 00F <allocated>	14 00E <allocated>	10 00A **addc** P X	11 00B **mulhwu** P X	9 009 **mulhdu** P X	8 008 **subfc** P X
00001	32 020 **cmpl** P X	33 021	35 023	34 022	38 026 <allocated>	39 027 <allocated>	37 025	36 024	44 02C	45 02D	47 02F <allocated>	46 02E <allocated>	42 02A	43 02B	41 029	40 028 **subf** P X
00011	96 060	97 061	99 063	98 062	102 066 <allocated>	103 067 <allocated>	101 065	100 064	108 06C	109 06D	111 06F <allocated>	110 06E <allocated>	106 06A	107 06B	105 069	104 068 **neg** P X
00010	64 040	65 041	67 043	66 042	70 046 <allocated>	71 047 <allocated>	69 045	68 044 **td** P X	76 04C	77 04D	79 04F <allocated>	78 04E <allocated>	74 04A	75 04B **mulhw** P X	73 049 **mulhd** P X	72 048
00110	192 0C0	193 0C1	195 0C3	194 0C2	198 0C6 <allocated>	199 0C7 <allocated>	197 0C5	196 0C4	204 0CC	205 0CD	207 0CF <allocated>	206 0CE <allocated>	202 0CA **addze** P X	203 0CB	201 0C9	200 0C8 **subfze** P X
00111	224 0E0	225 0E1	227 0E3	226 0E2	230 0E6 <allocated>	231 0E7 <allocated>	229 0E5	228 0E4	236 0EC	237 0ED	239 0EF <allocated>	238 0EE <allocated>	234 0EA **addme** P X	235 0EB **mullw** P X	233 0E9 **mulld** P X	232 0E8 **subfme** P X
00101	160 0A0	161 0A1	163 0A3 **wrteei** E X	162 0A2	166 0A6 <allocated>	167 0A7 <allocated>	165 0A5	164 0A4	172 0AC	173 0AD	175 0AF <allocated>	174 0AE <allocated>	170 0AA	171 0AB	169 0A9	168 0A8
00100	128 080	129 081	131 083 **wrtee** E X	130 082	134 086 <allocated>	135 087 <allocated>	133 085	132 084	140 08C	141 08D	143 08F <allocated>	142 08E <allocated>	138 08A **adde** P X	139 08B	137 089	136 088 **subfe** P X
01100	384 180	385 181	387 183	386 182	390 186 <allocated>	391 187 <allocated>	389 185	388 184	396 18C	397 18D	399 18F <allocated>	398 18E <allocated>	394 18A **adde64** E X	395 18B	393 189	392 188 **subfe64** E X
01101	416 1A0	417 1A1	419 1A3	418 1A2	422 1A6 <allocated>	423 1A7 <allocated>	421 1A5	420 1A4	428 1AC	429 1AD	431 1AF <allocated>	430 1AE <allocated>	426 1AA	427 1AB	425 1A9	424 1A8
01111	480 1E0	481 1E1	483 1E3	482 1E2	486 1E6 <allocated>	487 1E7 <allocated>	485 1E5	484 1E4	492 1EC	493 1ED	495 1EF <allocated>	494 1EE <allocated>	490 1EA **addme64** E X	491 1EB **divw** P X	489 1E9 **divd** P X	488 1E8 **subfme64** E X
01110	448 1C0	449 1C1	451 1C3 **mtdcr** E X	450 1C2	454 1C6 <allocated>	455 1C7 <allocated>	453 1C5	452 1C4	460 1CC	461 1CD	463 1CF <allocated>	462 1CE <allocated>	458 1CA **addze64** E X	459 1CB **divwu** P X	457 1C9 **divdu** P X	456 1C8 **subfze64** E X
01010	320 140	321 141	323 143 **mfdcr** E X	322 142	326 146 <allocated>	327 147 <allocated>	325 145	324 144	332 14C	333 14D	335 14F <allocated>	334 14E <allocated>	330 14A	331 14B	329 149	328 148
01011	352 160	353 161	355 163	354 162	358 166 <allocated>	359 167 <allocated>	357 165	356 164	364 16C	365 16D	367 16F <allocated>	366 16E <allocated>	362 16A	363 16B	361 169	360 168
01001	288 120	289 121	291 123	290 122	294 126 <allocated>	295 127 <allocated>	293 125	292 124	300 12C	301 12D	303 12F <allocated>	302 12E <allocated>	298 12A	299 12B	297 129	296 128
01000	256 100	257 101	259 103	258 102	262 106 <allocated>	263 107 <allocated>	261 105	260 104	268 10C	269 10D	271 10F <allocated>	270 10E <allocated>	266 10A **add** P X	267 10B	265 109	264 108
11000	768 300	769 301	771 303	770 302	774 306 <allocated>	775 307 <allocated>	773 305	772 304	780 30C	781 30D	783 30F <allocated>	782 30E <allocated>	778 30A **addo** P X	779 30B	777 309	776 308
11001	800 320	801 321	803 323	802 322	806 326 <allocated>	807 327 <allocated>	805 325	804 324	812 32C	813 32D	815 32F <allocated>	814 32E <allocated>	810 32A	811 32B	809 329	808 328
11011	864 360	865 361	867 363	866 362	870 366 <allocated>	871 367 <allocated>	869 365	868 364	876 36C	877 36D	879 36F <allocated>	878 36E <allocated>	874 36A	875 36B	873 369	872 368
11010	832 340	833 341	835 343	834 342	838 346 <allocated>	839 347 <allocated>	837 345	836 344	844 34C	845 34D	847 34F <allocated>	846 34E <allocated>	842 34A	843 34B	841 349	840 348
11110	960 3C0	961 3C1	963 3C3	962 3C2	966 3C6 <allocated>	967 3C7 <allocated>	965 3C5	964 3C4	972 3CC	973 3CD	975 3CF <allocated>	974 3CE <allocated>	970 3CA **addze64o** E X	971 3CB **divwuo** P X	969 3C9 **divduo** P X	968 3C8 **subfze64o** E X
11111	992 3E0	993 3E1	995 3E3	994 3E2	998 3E6 <allocated>	999 3E7 <allocated>	997 3E5	996 3E4	1004 3EC	1005 3ED	1007 3EF <allocated>	1006 3EE <allocated>	1002 3EA **addme64o** E X	1003 3EB **divwo** P X	1001 3E9 **divdo** P X	1000 3E8 **subfme64o** E X
11101	928 3A0	929 3A1	931 3A3	930 3A2	934 3A6 <allocated>	935 3A7 <allocated>	933 3A5	932 3A4	940 3AC	941 3AD	943 3AF <allocated>	942 3AE <allocated>	938 3AA	939 3AB	937 3A9	936 3A8
11100	896 380	897 381	899 383	798 382	902 386 <allocated>	903 387 <allocated>	901 385	900 384	908 38C	909 38D	911 38F <allocated>	910 38E <allocated>	906 38A **adde64o** E X	907 38B	905 389	904 388 **subfe64o** E X
10100	640 280	641 281	643 283	642 282	646 286 <allocated>	647 287 <allocated>	645 285	644 284	652 28C	653 28D	655 28F <allocated>	654 28E <allocated>	650 28A **addeo** P X	651 28B	649 289	648 288 **subfeo** P X
10101	672 2A0	673 2A1	675 2A3	674 2A2	678 2A6 <allocated>	679 2A7 <allocated>	677 2A5	676 2A4	684 2AC	685 2AD	687 2AF <allocated>	686 2AE <allocated>	682 2AA	683 2AB	681 2A9	680 2A8
10111	736 2E0	737 2E1	739 2E3	738 2E2	742 2E6 <allocated>	743 2E7 <allocated>	741 2E5	740 2E4	748 2EC	749 2ED	751 2EF <allocated>	750 2EE <allocated>	746 2EA **addmeo** P X	747 2EB **mullwo** P X	745 2E9 **mulldo** P X	744 2E8 **subfmeo** P X
10110	704 2C0	705 2C1	707 2C3	706 2C2	710 2C6 <allocated>	711 2C7 <allocated>	709 2C5	708 2C4	716 2CC	717 2CD	719 2CF <allocated>	718 2CE <allocated>	714 2CA **addzeo** P X	715 2CB	713 2C9	712 2C8 **subfzeo** P X
10010	576 240	577 241	579 243	578 242	582 246 <allocated>	583 247 <allocated>	581 245	580 244	588 24C	589 24D	591 24F <allocated>	590 24E <allocated>	586 24A	587 24B **///** P X	585 249 **///** P X	584 248
10011	608 260	609 261	611 263	610 262	614 266 <allocated>	615 267 <allocated>	613 265	612 264	620 26C	621 26D	623 26F <allocated>	622 26E <allocated>	618 26A	619 26B	617 269	616 268 **nego** P X
10001	544 220 **mcrxr64** E X	545 221	547 223	546 222	550 226 <allocated>	551 227 <allocated>	549 225	548 224	556 22C	557 22D	559 22F <allocated>	558 22E <allocated>	554 22A	555 22B	553 229	552 228 **subfo** P X
10000	512 200 **mcrxr** P X	513 201	515 203	514 202	518 206 <allocated>	519 207 <allocated>	517 205	516 204	524 20C	525 20D	527 20F <allocated>	526 20E <allocated>	522 20A **addco** P X	523 20B **///** P X	521 209 **///** P X	520 208 **subfco** P X

Table G-6. Extended opcodes for primary opcode 31 (instruction bits 21:30) (part 2 of 2)

11000	11001	11011	11010	11110	11111	11101	11100	10100	10101	10111	10110	10010	10011	10001	10000	
24 018 **slw** P X	25 019	27 01B **sld** P X	26 01A **cntlzw** P X	30 01E **icbte** E X	31 01F **lwzxe** E X	29 01D	28 01C **and** P X	20 014 **lwarx** P X	21 015	23 017 **lwzx** P X	22 016 **icbt** E X	18 012	19 013 **mfcr** P X	17 011	16 010	00000
56 038	57 039	59 03B	58 03A **cntlzd** P X	62 03E **dcbste** E X	63 03F **lwzuxe** E X	61 03D	60 03C **andc** P X	52 034	53 035	55 037 **lwzux** P X	54 036 **dcbst** P X	50 032	51 033	49 031	48 030	00001
120 078	121 079	123 07B	122 07A	126 07E **lwarxe** E X	127 07F **lbzuxe** E X	125 07D	124 07C **nor** P X	116 074	117 075	119 077 **lbzux** P X	118 076	114 072	115 073	113 071	112 070	00011
88 058	89 059	91 05B	90 05A	94 05E **dcbfe** E X	95 05F **lbzxe** E X	93 05D	92 05C	84 054	85 055	87 057 **lbzx** P X	86 056 **dcbf** P X	82 052	83 053 **mfmsr** P X	81 051	80 050	00010
216 0D8	217 0D9	219 0DB	218 0DA	222 0DE	223 0DF **stbxe** E X	221 0DD	220 0DC	212 0D4	213 0D5	215 0D7 **stbx** P X	214 0D6	210 0D2 (preserved)	211 0D3	209 0D1	208 0D0	00110
248 0F8	249 0F9	251 0FB	250 0FA	254 0FE **dcbtste** E X	255 0FF **stbuxe** E X	253 0FD	252 0FC	244 0F4	245 0F5	247 0F7 **stbux** P X	246 0F6 **dcbtst** P X	242 0F2 (preserved)	243 0F3	241 0F1	240 0F0	00111
184 0B8	185 0B9	187 0BB	186 0BA	190 0BE	191 0BF **stwuxe** E X	189 0BD	188 0BC	180 0B4	181 0B5	183 0B7 **stwux** P X	182 0B6	178 0B2	179 0B3	177 0B1	176 0B0	00101
152 098	153 099	155 09B	154 09A	158 09E **stwcxe.** E X	159 09F **stwxe** E X	157 09D	156 09C	148 094	149 095	151 097 **stwx** P X	150 096 **stwcx.** P X	146 092 **mtmsr** P X	147 093	145 091	144 090 **mtcrf** P X	00100
408 198	409 199	411 19B	410 19A	414 19E	415 19F **sthxe** E X	413 19D	412 19C **orc** P X	404 194	405 195	407 197 **sthx** P X	406 196	402 192	403 193	401 191	400 190	01100
450 1B8	451 1B9	453 1BB	452 1BA	456 1BE	457 1BF **sthuxe** E X	455 1BD	454 1BC **or** P X	436 1B4	437 1B5	439 1B7 **sthux** P X	438 1B6 (preserved)	434 1B2	435 1B3	433 1B1	432 1B0	01101
504 1F8	505 1F9	507 1FB	506 1FA	510 1FE	511 1FF **stdcxe.** E X	509 1FD	508 1FC	500 1F4	501 1F5	503 1F7	502 1F6	498 1F2	499 1F3	497 1F1	496 1F0	01111
472 1D8	473 1D9	475 1DB	474 1DA	478 1DE **dcbie** E X	479 1DF **ldarxe** E X	477 1DD	476 1DC **nand** P X	468 1D4	469 1D5	471 1D7	470 1D6 **dcbi** X	466 1D2	467 1D3 **mtspr** P X	465 1D1	464 1D0	01110
344 158	345 159	347 15B	346 15A	350 15E	351 15F **lhaxe** E X	349 15D	348 15C	340 154	341 155	343 157 **lhax** P X	342 156 (allocated)	338 152	339 153 **mfspr** P X	337 151	336 150	01010
376 178	377 179	379 17B	378 17A	382 17E	383 17F **lhauxe** E X	381 17D	380 17C	372 174	373 175	375 177 **lhaux** P X	374 176 (allocated)	370 172 (preserved)	371 173 (preserved)	369 171	368 170	01011
312 138	313 139	315 13B	314 13A	318 13E	319 13F **lhzuxe** E X	317 13D	316 13C **xor** P X	308 134	309 135	311 137 **lhzux** P X	310 136 (preserved)	306 132 (preserved)	307 133	305 131	304 130	01001
280 118	281 119	283 11B	282 11A	286 11E **dcbte** E X	287 11F **lhzxe** E X	285 11D	284 11C **eqv** P X	276 114	277 115	279 117 **lhzx** P X	278 116 **dcbt** P X	274 112	275 113 **mfapidi** E X	273 111	272 110	01000
792 318 **sraw** P X	793 319	795 31B	794 31A **srad** P X	798 31E **lhbrxe** E X	799 31F **ldxe** E X	797 31D	796 31C	788 314	789 315	791 317	790 316 **lhbrx** P X	786 312 **tlbivax** E X	787 313 **tlbivaxe** E X	785 311	784 310	11000
824 338 **srawi** P X	825 339	827 33B	826 33A **sradi** P XS	830 33E	831 33F **lduxe** E X	829 33D	828 33C	820 334	821 335	823 337	822 336 (allocated)	818 332	819 333	817 331	816 330	11001
888 378	889 379	891 37B	890 37A	894 37E	895 37F	893 37D	892 37C	884 374	885 375	887 377	886 376	882 372	883 373	881 371	880 370	11011
856 358	857 359	859 35B	858 35A	862 35E	863 35F	861 35D	860 35C	852 354	853 355	855 357	854 356 **mbar** P X	850 352	851 353	849 351	848 350	11010
984 3D8	985 3D9	987 3DB	986 3DA **extsw** P X	990 3DE **icbie** E X	991 3DF **stfiwxe** E X	989 3DD	988 3DC	980 3D4	981 3D5	983 3D7 **stfiwx** P X	982 3D6 **icbi** P X	978 3D2 **tlbwe** E X	979 3D3	977 3D1	976 3D0	11110
1016 3F8	1017 3F9	1019 3FB	1018 3FA	1022 3FE **dcbze** E X	1023 3FF	1021 3FD	1020 3FC	1012 3F4	1013 3F5	1015 3F7	1014 3F6 **dcbz** P X	1010 3F2	1011 3F3	1009 3F1	1008 3F0	11111
952 3B8	953 3B9	955 3BB	954 3BA **extsb** P X	958 3BE	959 3BF **stduxe** E X	957 3BD	956 3BC	948 3B4	949 3B5	951 3B7	950 3B6	946 3B2 **tlbre** E X	947 3B3	945 3B1	944 3B0	11101
920 398	921 399	923 39B	922 39A **extsh** P X	926 39E **sthbrxe** E X	927 39F **stdxe** E X	925 39D	924 39C	916 394	917 395	919 397	918 396 **sthbrx** P X	914 392 **tlbsx** E X	915 393 **tlbsxe** E X	913 391	912 390	11100
664 298	665 299	667 29B	666 29A	670 29E **stwbrxe** E X	671 29F **stfsxe** E X	669 29D	668 29C	660 294	661 295 **stswx** P X	663 297 **stfsx** P X	662 296 **stwbrx** P X	658 292 (nop)	659 293 (preserved)	657 291	656 290	10100
696 2B8	697 2B9	699 2BB	698 2BA	702 2BE	703 2BF **stfsuxe** E X	701 2BD	700 2BC	692 2B4	693 2B5	695 2B7 **stfsux** P X	694 2B6	690 2B2 (nop)	691 2B3	689 2B1	688 2B0	10101
760 2F8	761 2F9	763 2FB	762 2FA	766 2FE **dcbae** E X	767 2FF **stfduxe** E X	765 2FD	764 2FC	756 2F4	757 2F5	759 2F7 **stfdux** P X	758 2F6 **dcba** P X	754 2F2 (nop)	755 2F3	753 2F1	752 2F0	10111
728 2D8	729 2D9	731 2DB	730 2DA	734 2DE	735 2DF **stfdxe** E X	733 2DD	732 2DC	724 2D4	725 2D5 **stswi** P X	727 2D7 **stfdx** P X	726 2D6	722 2D2 (nop)	723 2D3	721 2D1	720 2D0	10110
600 258	601 259	603 25B	602 25A	606 25E	607 25F **lfdxe** E X	605 25D	604 25C	596 254	597 255 **lswi** P X	599 257 **lfdx** P X	598 256 **msync** P X	594 252 (nop)	595 253 (preserved)	593 251	592 250	10010
632 278	633 279	635 27B	634 27A	638 27E	639 27F **lfduxe** E X	637 27D	636 27C	628 274	629 275	631 277 **lfdux** P X	630 276	626 272 (nop)	627 273	625 271	624 270	10011
568 238	569 239	571 23B	570 23A	574 23E	575 23F **lfsuxe** E X	573 23D	572 23C	564 234	565 235	567 237 **lfsux** P X	566 236 **tlbsync** P X	562 232 (nop)	563 233	561 231	560 230	10001
536 218 **srw** P X	537 219	539 21B **srd** P X	538 21A	542 21E **lwbrxe** E X	543 21F **lfsxe** E X	541 21D	540 21C	532 214	533 215 **lswx** P X	535 217 **lfsx** P X	534 216 **lwbrx** P X	530 212 (nop)	531 213	529 211	528 210	10000

Table G-7. Extended opcodes for primary opcode 59 (instruction bits 21:30) (part 1 of 2)

	00000	00001	00011	00010	00110	00111	00101	00100	01100	01101	01111	01110	01010	01011	01001	01000
00000	0 000	1 001	3 003	2 002	6 006	7 007	5 005 <allocated>	4 004 <allocated>	12 00C <allocated>	13 00D <allocated>	15 00F	14 00E	10 00A	11 00B	9 009	8 008
00001	32 020	33 021	35 023	34 022	38 026	39 027	37 025 <allocated>	36 024 <allocated>	44 02C <allocated>	45 02D <allocated>	47 02F	46 02E	42 02A	43 02B	41 029	40 028
00011	96 060	97 061	99 063	98 062	102 066	103 067	101 065 <allocated>	100 064 <allocated>	108 06C <allocated>	109 06D <allocated>	111 06F	110 06E	106 06A	107 06B	105 069	104 068
00010	64 040	65 041	67 043	66 042	70 046	71 047	69 045 <allocated>	68 044 <allocated>	76 04C <allocated>	77 04D <allocated>	79 04F	78 04E	74 04A	75 04B	73 049	72 048
00110	192 0C0	193 0C1	195 0C3	194 0C2	198 0C6	199 0C7	197 0C5 <allocated>	196 0C4 <allocated>	204 0CC <allocated>	205 0CD <allocated>	207 0CF	206 0CE	202 0CA	203 0CB	201 0C9	200 0C8
00111	224 0E0	225 0E1	227 0E3	226 0E2	230 0E6	231 0E7	229 0E5 <allocated>	228 0E4 <allocated>	236 0EC <allocated>	237 0ED <allocated>	239 0EF	238 0EE	234 0EA	235 0EB	233 0E9	232 0E8
00101	160 0A0	161 0A1	163 0A3	162 0A2	166 0A6	167 0A7	165 0A5 <allocated>	164 0A4 <allocated>	172 0AC <allocated>	173 0AD <allocated>	175 0AF	174 0AE	170 0AA	171 0AB	169 0A9	168 0A8
00100	128 080	129 081	131 083	130 082	134 086	135 087	133 085 <allocated>	132 084 <allocated>	140 08C <allocated>	141 08D <allocated>	143 08F	142 08E	138 08A	139 08B	137 089	136 088
01100	384 180	385 181	387 183	386 182	390 186	391 187	389 185 <allocated>	388 184 <allocated>	396 18C <allocated>	397 18D <allocated>	399 18F	398 18E	394 18A	395 18B	393 189	392 188
01101	416 1A0	417 1A1	419 1A3	418 1A2	422 1A6	423 1A7	421 1A5 <allocated>	420 1A4 <allocated>	428 1AC <allocated>	429 1AD <allocated>	431 1AF	430 1AE	426 1AA	427 1AB	425 1A9	424 1A8
01111	480 1E0	481 1E1	483 1E3	482 1E2	486 1E6	487 1E7	485 1E5 <allocated>	484 1E4 <allocated>	492 1EC <allocated>	493 1ED <allocated>	495 1EF	494 1EE	490 1EA	491 1EB	489 1E9	488 1E8
01110	448 1C0	449 1C1	451 1C3	450 1C2	454 1C6	455 1C7	453 1C5 <allocated>	452 1C4 <allocated>	460 1CC <allocated>	461 1CD <allocated>	463 1CF	462 1CE	458 1CA	459 1CB	457 1C9	456 1C8
01010	320 140	321 141	323 143	322 142	326 146	327 147	325 145 <allocated>	324 144 <allocated>	332 14C <allocated>	333 14D <allocated>	335 14F	334 14E	330 14A	331 14B	329 149	328 148
01011	352 160	353 161	355 163	354 162	358 166	359 167	357 165 <allocated>	356 164 <allocated>	364 16C <allocated>	365 16D <allocated>	367 16F	366 16E	362 16A	363 16B	361 169	360 168
01001	288 120	289 121	291 123	290 122	294 126	295 127	293 125 <allocated>	292 124 <allocated>	300 12C <allocated>	301 12D <allocated>	303 12F	302 12E	298 12A	299 12B	297 129	296 128
01000	256 100	257 101	259 103	258 102	262 106	263 107	261 105 <allocated>	260 104 <allocated>	268 10C <allocated>	269 10D <allocated>	271 10F	270 10E	266 10A	267 10B	265 109	264 108
11000	768 300	769 301	771 303	770 302	774 306	775 307	773 305 <allocated>	772 304 <allocated>	780 30C <allocated>	781 30D <allocated>	783 30F	782 30E	778 30A	779 30B	777 309	776 308
11001	800 320	801 321	803 323	802 322	806 326	807 327	805 325 <allocated>	804 324 <allocated>	812 32C <allocated>	813 32D <allocated>	815 32F	814 32E	810 32A	811 32B	809 329	808 328
11011	864 360	865 361	867 363	866 362	870 366	871 367	869 365 <allocated>	868 364 <allocated>	876 36C <allocated>	877 36D <allocated>	879 36F	878 36E	874 36A	875 36B	873 369	872 368
11010	832 340	833 341	835 343	834 342	838 346	839 347	837 345 <allocated>	836 344 <allocated>	844 34C <allocated>	845 34D <allocated>	847 34F	846 34E	842 34A	843 34B	841 349	840 348
11110	960 3C0	961 3C1	963 3C3	962 3C2	966 3C6	967 3C7	965 3C5 <allocated>	964 3C4 <allocated>	972 3CC <allocated>	973 3CD <allocated>	975 3CF	974 3CE	970 3CA	971 3CB	969 3C9	968 3C8
11111	992 3E0	993 3E1	995 3E3	994 3E2	998 3E6	999 3E7	997 3E5 <allocated>	996 3E4 <allocated>	1004 3EC <allocated>	1005 3ED <allocated>	1007 3EF	1006 3EE	1002 3EA	1003 3EB	1001 3E9	1000 3E8
11101	928 3A0	929 3A1	931 3A3	930 3A2	934 3A6	935 3A7	933 3A5 <allocated>	932 3A4 <allocated>	940 3AC <allocated>	941 3AD <allocated>	943 3AF	942 3AE	938 3AA	939 3AB	937 3A9	936 3A8
11100	896 380	897 381	899 383	798 382	902 386	903 387	901 385 <allocated>	900 384 <allocated>	908 38C <allocated>	909 38D <allocated>	911 38F	910 38E	906 38A	907 38B	905 389	904 388
10100	640 280	641 281	643 283	642 282	646 286	647 287	645 285 <allocated>	644 284 <allocated>	652 28C <allocated>	653 28D <allocated>	655 28F	654 28E	650 28A	651 28B	649 289	648 288
10101	672 2A0	673 2A1	675 2A3	674 2A2	678 2A6	679 2A7	677 2A5 <allocated>	676 2A4 <allocated>	684 2AC <allocated>	685 2AD <allocated>	687 2AF	686 2AE	682 2AA	683 2AB	681 2A9	680 2A8
10111	736 2E0	737 2E1	739 2E3	738 2E2	742 2E6	743 2E7	741 2E5 <allocated>	740 2E4 <allocated>	748 2EC <allocated>	749 2ED <allocated>	751 2EF	750 2EE	746 2EA	747 2EB	745 2E9	744 2E8
10110	704 2C0	705 2C1	707 2C3	706 2C2	710 2C6	711 2C7	709 2C5 <allocated>	708 2C4 <allocated>	716 2CC <allocated>	717 2CD <allocated>	719 2CF	718 2CE	714 2CA	715 2CB	713 2C9	712 2C8
10010	576 240	577 241	579 243	578 242	582 246	583 247	581 245 <allocated>	580 244 <allocated>	588 24C <allocated>	589 24D <allocated>	591 24F	590 24E	586 24A	587 24B	585 249	584 248
10011	608 260	609 261	611 263	610 262	614 266	615 267	613 265 <allocated>	612 264 <allocated>	620 26C <allocated>	621 26D <allocated>	623 26F	622 26E	618 26A	619 26B	617 269	616 268
10001	544 220	545 221	547 223	546 222	550 226	551 227	549 225 <allocated>	548 224 <allocated>	556 22C <allocated>	557 22D <allocated>	559 22F	558 22E	554 22A	555 22B	553 229	552 228
10000	512 200	513 201	515 203	514 202	518 206	519 207	517 205 <allocated>	516 204 <allocated>	524 20C <allocated>	525 20D <allocated>	527 20F	526 20E	522 20A	523 20B	521 209	520 208

Table G-7. Extended opcodes for primary opcode 59 (instruction bits 21:30) (part 2 of 2)

11000	11001	11011	11010	11110	11111	11101	11100	10100	10101	10111	10110	10010	10011	10001	10000	
24 018 **fres** P A	25 019 **fmuls** P A	27 01B	26 01A	30 01E **fnmsubs** P A	31 01F **fnmadds** P A	29 01D **fmadds** P A	28 01C **fmsubs** P A	20 014 **fsubs** P A	21 015 **fadds** P A	23 017	22 016 **fsqrts** P A	18 012 **fdivs** P A	19 013	17 011	16 010	00000
		59 03B	58 03A							55 037			51 033	49 031	48 030	00001
		123 07B	122 07A							119 077			115 073	113 071	112 070	00011
		91 05B	90 05A							87 057			83 053	81 051	80 050	00010
		219 0DB	218 0DA							215 0D7			211 0D3	209 0D1	208 0D0	00110
		251 0FB	250 0FA							247 0F7			243 0F3	241 0F1	240 0F0	00111
		187 0BB	186 0BA							183 0B7			179 0B3	177 0B1	176 0B0	00101
		155 09B	154 09A							151 097			147 093	145 091	144 090	00100
		411 19B	410 19A							407 197			403 193	401 191	400 190	01100
		453 1BB	452 1BA							439 1B7			435 1B3	433 1B1	432 1B0	01101
		507 1FB	506 1FA							503 1F7			499 1F3	497 1F1	496 1F0	01111
		475 1DB	474 1DA							471 1D7			467 1D3	465 1D1	464 1D0	01110
		347 15B	346 15A							343 157			339 153	337 151	336 150	01010
		379 17B	378 17A							375 177			371 173	369 171	368 170	01011
		315 13B	314 13A							311 137			307 133	305 131	304 130	01001
		283 11B	282 11A							279 117			275 113	273 111	272 110	01000
		795 31B	794 31A							791 317			787 313	785 311	784 310	11000
		827 33B	826 33A							823 337			819 333	817 331	816 330	11001
		891 37B	890 37A							887 377			883 373	881 371	880 370	11011
		859 35B	858 35A							855 357			851 353	849 351	848 350	11010
		987 3DB	986 3DA							983 3D7			979 3D3	977 3D1	976 3D0	11110
		1019 3FB	1018 3FA							1015 3F7			1011 3F3	1009 3F1	1008 3F0	11111
		955 3BB	954 3BA							951 3B7			947 3B3	945 3B1	944 3B0	11101
		923 39B	922 39A							919 397			915 393	913 391	912 390	11100
		667 29B	666 29A							663 297			659 293	657 291	656 290	10100
		699 2BB	698 2BA							695 2B7			691 2B3	689 2B1	688 2B0	10101
		763 2FB	762 2FA							759 2F7			755 2F3	753 2F1	752 2F0	10111
		731 2DB	730 2DA							727 2D7			723 2D3	721 2D1	720 2D0	10110
		603 25B	602 25A							599 257			595 253	593 251	592 250	10010
		635 27B	634 27A							631 277			627 273	625 271	624 270	10011
		571 23B	570 23A							567 237			563 233	561 231	560 230	10001
		539 21B	538 21A							535 217			531 213	529 211	528 210	10000

Table G-8. Extended opcodes for primary opcode 63 (instruction bits 21:30) (part 1 of 2)

	00000	00001	00011	00010	00110	00111	00101	00100	01100	01101	01111	01110	01010	01011	01001	01000
00000	0 000 **fcmpu** P X	1 001	3 003	2 002	6 006	7 007	5 005 <allocated>	4 004 <allocated>	12 00C **frsp** P X	13 00D <allocated>	15 00F **fctiwz** P X	14 00E **fctiw** P X	10 00A	11 00B	9 009	8 008
00001	32 020 **fcmpo** P X	33 021	35 023	34 022	38 026 **mtfsb1** P X	39 027	37 025 <allocated>	36 024 <allocated>	44 02C <allocated>	45 02D <allocated>	47 02F	46 02E	42 02A	43 02B	41 029	40 028 **fneg** P X
00011	96 060	97 061	99 063	98 062	102 066	103 067	101 065 <allocated>	100 064 <allocated>	108 06C <allocated>	109 06D <allocated>	111 06F	110 06E	106 06A	107 06B	105 069	104 068
00010	64 040 **mcrfs** P X	65 041	67 043	66 042	70 046 **mtfsb0** P X	71 047	69 045 <allocated>	68 044 <allocated>	76 04C <allocated>	77 04D <allocated>	79 04F	78 04E	74 04A	75 04B	73 049	72 048 **fmr** P X
00110	192 0C0	193 0C1	195 0C3	194 0C2	198 0C6	199 0C7	197 0C5 <allocated>	196 0C4 <allocated>	204 0CC <allocated>	205 0CD <allocated>	207 0CF	206 0CE	202 0CA	203 0CB	201 0C9	200 0C8
00111	224 0E0	225 0E1	227 0E3	226 0E2	230 0E6	231 0E7	229 0E5 <allocated>	228 0E4 <allocated>	236 0EC <allocated>	237 0ED <allocated>	239 0EF	238 0EE	234 0EA	235 0EB	233 0E9	232 0E8
00101	160 0A0	161 0A1	163 0A3	162 0A2	166 0A6	167 0A7	165 0A5 <allocated>	164 0A4 <allocated>	172 0AC <allocated>	173 0AD <allocated>	175 0AF	174 0AE	170 0AA	171 0AB	169 0A9	168 0A8
00100	128 080	129 081	131 083	130 082	134 086 **mtfsfi** P X	135 087	133 085 <allocated>	132 084 <allocated>	140 08C <allocated>	141 08D <allocated>	143 08F	142 08E	138 08A	139 08B	137 089	136 088 **fnabs** P X
01100	384 180	385 181	387 183	386 182	390 186	391 187	389 185 <allocated>	388 184 <allocated>	396 18C <allocated>	397 18D <allocated>	399 18F	398 18E	394 18A	395 18B	393 189	392 188
01101	416 1A0	417 1A1	419 1A3	418 1A2	422 1A6	423 1A7	421 1A5 <allocated>	420 1A4 <allocated>	428 1AC <allocated>	429 1AD <allocated>	431 1AF	430 1AE	426 1AA	427 1AB	425 1A9	424 1A8
01111	480 1E0	481 1E1	483 1E3	482 1E2	486 1E6	487 1E7	485 1E5 <allocated>	484 1E4 <allocated>	492 1EC <allocated>	493 1ED <allocated>	495 1EF	494 1EE	490 1EA	491 1EB	489 1E9	488 1E8
01110	448 1C0	449 1C1	451 1C3	450 1C2	454 1C6	455 1C7	453 1C5 <allocated>	452 1C4 <allocated>	460 1CC <allocated>	461 1CD <allocated>	463 1CF	462 1CE	458 1CA	459 1CB	457 1C9	456 1C8
01010	320 140	321 141	323 143	322 142	326 146	327 147	325 145 <allocated>	324 144 <allocated>	332 14C <allocated>	333 14D <allocated>	335 14F	334 14E	330 14A	331 14B	329 149	328 148
01011	352 160	353 161	355 163	354 162	358 166	359 167	357 165 <allocated>	356 164 <allocated>	364 16C <allocated>	365 16D <allocated>	367 16F	366 16E	362 16A	363 16B	361 169	360 168
01001	288 120	289 121	291 123	290 122	294 126	295 127	293 125 <allocated>	292 124 <allocated>	300 12C <allocated>	301 12D <allocated>	303 12F	302 12E	298 12A	299 12B	297 129	296 128
01000	256 100	257 101	259 103	258 102	262 106	263 107	261 105 <allocated>	260 104 <allocated>	268 10C <allocated>	269 10D <allocated>	271 10F	270 10E	266 10A	267 10B	265 109	264 108 **fabs** P X
11000	768 300	769 301	771 303	770 302	774 306	775 307	773 305 <allocated>	772 304 <allocated>	780 30C <allocated>	781 30D <allocated>	783 30F	782 30E	778 30A	779 30B	777 309	776 308
11001	800 320	801 321	803 323	802 322	806 326	807 327	805 325 <allocated>	804 324 <allocated>	812 32C <allocated>	813 32D <allocated>	815 32F **fctidz** P X	814 32E **fctid** P X	810 32A	811 32B	809 329	808 328
11011	864 360	865 361	867 363	866 362	870 366	871 367	869 365 <allocated>	868 364 <allocated>	876 36C <allocated>	877 36D <allocated>	879 36F	878 36E	874 36A	875 36B	873 369	872 368
11010	832 340	833 341	835 343	834 342	838 346	839 347	837 345 <allocated>	836 344 <allocated>	844 34C <allocated>	845 34D <allocated>	847 34F	846 34E **fcfid** P X	842 34A	843 34B	841 349	840 348
11110	960 3C0	961 3C1	963 3C3	962 3C2	966 3C6	967 3C7	965 3C5 <allocated>	964 3C4 <allocated>	972 3CC <allocated>	973 3CD <allocated>	975 3CF	974 3CE	970 3CA	971 3CB	969 3C9	968 3C8
11111	992 3E0	993 3E1	995 3E3	994 3E2	998 3E6	999 3E7	997 3E5 <allocated>	996 3E4 <allocated>	1004 3EC <allocated>	1005 3ED <allocated>	1007 3EF	1006 3EE	1002 3EA	1003 3EB	1001 3E9	1000 3E8
11101	928 3A0	929 3A1	931 3A3	930 3A2	934 3A6	935 3A7	933 3A5 <allocated>	932 3A4 <allocated>	940 3AC <allocated>	941 3AD <allocated>	943 3AF	942 3AE	938 3AA	939 3AB	937 3A9	936 3A8
11100	896 380	897 381	899 383	798 382	902 386	903 387	901 385 <allocated>	900 384 <allocated>	908 38C <allocated>	909 38D <allocated>	911 38F	910 38E	906 38A	907 38B	905 389	904 388
10100	640 280	641 281	643 283	642 282	646 286	647 287	645 285 <allocated>	644 284 <allocated>	652 28C <allocated>	653 28D <allocated>	655 28F	654 28E	650 28A	651 28B	649 289	648 288
10101	672 2A0	673 2A1	675 2A3	674 2A2	678 2A6	679 2A7	677 2A5 <allocated>	676 2A4 <allocated>	684 2AC <allocated>	685 2AD <allocated>	687 2AF	686 2AE	682 2AA	683 2AB	681 2A9	680 2A8
10111	736 2E0	737 2E1	739 2E3	738 2E2	742 2E6	743 2E7	741 2E5 <allocated>	740 2E4 <allocated>	748 2EC <allocated>	749 2ED <allocated>	751 2EF	750 2EE	746 2EA	747 2EB	745 2E9	744 2E8
10110	704 2C0	705 2C1	707 2C3	706 2C2	710 2C6	711 2C7 **mtfsf** P XFL	709 2C5 <allocated>	708 2C4 <allocated>	716 2CC <allocated>	717 2CD <allocated>	719 2CF	718 2CE	714 2CA	715 2CB	713 2C9	712 2C8
10010	576 240	577 241	579 243	578 242	582 246	583 247 **mffs** P X	581 245 <allocated>	580 244 <allocated>	588 24C <allocated>	589 24D <allocated>	591 24F	590 24E	586 24A	587 24B	585 249	584 248
10011	608 260	609 261	611 263	610 262	614 266	615 267	613 265 <allocated>	612 264 <allocated>	620 26C <allocated>	621 26D <allocated>	623 26F	622 26E	618 26A	619 26B	617 269	616 268
10001	544 220	545 221	547 223	546 222	550 226	551 227	549 225 <allocated>	548 224 <allocated>	556 22C <allocated>	557 22D <allocated>	559 22F	558 22E	554 22A	555 22B	553 229	552 228
10000	512 200	513 201	515 203	514 202	518 206	519 207	517 205 <allocated>	516 204 <allocated>	524 20C <allocated>	525 20D <allocated>	527 20F	526 20E	522 20A	523 20B	521 209	520 208

Table G-8. Extended opcodes for primary opcode 63 (instruction bits 21:30) (part 2 of 2)

11000	11001	11011	11010	11110	11111	11101	11100	10100	10101	10111	10110	10010	10011	10001	10000	
24 018	25 019 **fmul** P A	27 01B	26 01A **frsqrte** P A	30 01E **fnmsub** P A	31 01F **fnmadd** P A	29 01D **fmadd** P A	28 01C **fmsub** P A	20 014 **fsub** P A	21 015 **fadd** P A	23 017 **fsel** P A	22 016 **fsqrt** P A	18 012 **fdiv** P A	19 013	17 011	16 010	00000
56 038		59 03B											51 033	49 031	48 030	00001
120 078		123 07B											115 073	113 071	112 070	00011
88 058		91 05B											83 053	81 051	80 050	00010
216 0D8		219 0DB											211 0D3	209 0D1	208 0D0	00110
248 0F8		251 0FB											243 0F3	241 0F1	240 0F0	00111
184 0B8		187 0BB											179 0B3	177 0B1	176 0B0	00101
152 098		155 09B											147 093	145 091	144 090	00100
408 198		411 19B											403 193	401 191	400 190	01100
450 1B8		453 1BB											435 1B3	433 1B1	432 1B0	01101
504 1F8		507 1FB											499 1F3	497 1F1	496 1F0	01111
472 1D8		475 1DB											467 1D3	465 1D1	464 1D0	01110
344 158		347 15B											339 153	337 151	336 150	01010
376 178		379 17B											371 173	369 171	368 170	01011
312 138		315 13B											307 133	305 131	304 130	01001
280 118		283 11B											275 113	273 111	272 110	01000
792 318		795 31B											787 313	785 311	784 310	11000
824 338		827 33B											819 333	817 331	816 330	11001
888 378		891 37B											883 373	881 371	880 370	11011
856 358		859 35B											851 353	849 351	848 350	11010
984 3D8		987 3DB											979 3D3	977 3D1	976 3D0	11110
1016 3F8		1019 3FB											1011 3F3	1009 3F1	1008 3F0	11111
952 3B8		955 3BB											947 3B3	945 3B1	944 3B0	11101
920 398		923 39B											915 393	913 391	912 390	11100
664 298		667 29B											659 293	657 291	656 290	10100
696 2B8		699 2BB											691 2B3	689 2B1	688 2B0	10101
760 2F8		763 2FB											755 2F3	753 2F1	752 2F0	10111
728 2D8		731 2DB											723 2D3	721 2D1	720 2D0	10110
600 258		603 25B											595 253	593 251	592 250	10010
632 278		635 27B											627 273	625 271	624 270	10011
568 238		571 23B											563 233	561 231	560 230	10001
536 218		539 21B											531 213	529 211	528 210	10000

Appendix H Instruction Index

H.1 Instruction Index Sorted by Opcode

This appendix lists all the instructions in Book E, in order by opcode.

Format	Opcode		Mnemonic	Instruction	Page
	Primary ($Inst_{0:5}$)	Extended ($Inst_{21:31}$)			
D	000010	----- ----- -	tdi	Trap Doubleword Immediate	361
D	000011	----- ----- -	twi	Trap Word Immediate	367
D	000111	----- ----- -	mulli	Multiply Low Immediate	319
D	001000	----- ----- -	subfic	Subtract From Immediate Carrying	358
B	001001	----- ----0 0	bce	Branch Conditional Extended	238
B	001001	----- ----0 1	bcel	Branch Conditional Extended & Link	238
B	001001	----- ----1 0	bcea	Branch Conditional Extended Absolute	238
B	001001	----- ----1 1	bcela	Branch Conditional Extended & Link Absolute	238
D	001010	----- ----- -	cmpli	Compare Logical Immediate	242
D	001011	----- ----- -	cmpi	Compare Immediate	241
D	001100	----- ----- -	addic	Add Immediate Carrying	233
D	001101	----- ----- -	addic.	Add Immediate Carrying & record CR	233
D	001110	----- ----- -	addi	Add Immediate	232
D	001111	----- ----- -	addis	Add Immediate Shifted	232
B	010000	----- ----0 0	bc	Branch Conditional	238
B	010000	----- ----0 1	bcl	Branch Conditional & Link	238
B	010000	----- ----1 0	bca	Branch Conditional Absolute	238
B	010000	----- ----1 1	bcla	Branch Conditional & Link Absolute	238
SC	010001	///// ////1 /	sc	System Call	334
I	010010	----- ----0 0	b	Branch	237
I	010010	----- ----0 1	bl	Branch & Link	237
I	010010	----- ----1 0	ba	Branch Absolute	237

Legend:

- `-` Don't care, usually part of an operand field
- `/` Reserved bit, invalid instruction form if encoded as 1
- `?` Allocated for implementation-dependent use. See User' Manual for the implementation

Format	Opcode		Mnemonic	Instruction	Page
	Primary ($Inst_{0:5}$)	Extended ($Inst_{21:31}$)			
I	010010	----- ----1 1	bla	Branch & Link Absolute	237
XL	010011	00000 00000 /	mcrf	Move Condition Register Field	305
XL	010011	00000 10000 0	bclr	Branch Conditional to Link Register	240
XL	010011	00000 10000 1	bclrl	Branch Conditional to Link Register & Link	240
XL	010011	00000 10001 0	bclre	Branch Conditional to Link Register Extended	240
XL	010011	00000 10001 1	bclrel	Branch Conditional to Link Register Extended & Link	240
XL	010011	00001 00001 /	crnor	Condition Register NOR	245
XL	010011	00001 10010 /	rfi	Return From Interrupt	326
XL	010011	00001 10011 /	rfci	Return From Critical Interrupt	325
XL	010011	00100 00001 /	crandc	Condition Register AND with Complement	244
XL	010011	00100 10110 /	isync	Instruction Synchronize	288
XL	010011	00110 00001 /	crxor	Condition Register XOR	246
XL	010011	00111 00001 /	crnand	Condition Register NAND	245
XL	010011	01000 00001 /	crand	Condition Register AND	244
XL	010011	01001 00001 /	creqv	Condition Register Equivalent	244
XL	010011	01101 00001 /	crorc	Condition Register OR with Complement	246
XL	010011	01110 00001 /	cror	Condition Register OR	245
XL	010011	10000 10000 0	bcctr	Branch Conditional to Count Register	239
XL	010011	10000 10000 1	bcctrl	Branch Conditional to Count Register & Link	239
XL	010011	10000 10001 0	bcctre	Branch Conditional to Count Register Extended	239
XL	010011	10000 10001 1	bcctrel	Branch Conditional to Count Register Extended & Link	239
M	010100	----- ----- 0	rlwimi	Rotate Left Word Immed then Mask Insert	331
M	010100	----- ----- 1	rlwimi.	Rotate Left Word Immed then Mask Insert & record CR	331
M	010101	----- ----- 0	rlwinm	Rotate Left Word Immed then AND with Mask	332
M	010101	----- ----- 1	rlwinm.	Rotate Left Word Immed then AND with Mask & record CR	332
I	010110	----- ----0 0	be	Branch Extended	238
I	010110	----- ----0 1	bel	Branch Extended & Link	238
I	010110	----- ----1 0	bea	Branch Extended Absolute	238
I	010110	----- ----1 1	bela	Branch Extended & Link Absolute	238
M	010111	----- ----- 0	rlwnm	Rotate Left Word then AND with Mask	332
M	010111	----- ----- 1	rlwnm.	Rotate Left Word then AND with Mask & record CR	332
D	011000	----- ----- -	ori	OR Immediate	324
D	011001	----- ----- -	oris	OR Immediate Shifted	324
D	011010	----- ----- -	xori	XOR Immediate	369
D	011011	----- ----- -	xoris	XOR Immediate Shifted	369
D	011100	----- ----- -	andi.	AND Immediate & record CR	236
D	011101	----- ----- -	andis.	AND Immediate Shifted & record CR	236
MD	011110	----- -000- /	rldicl	Rotate Left Doubleword Immediate then Clear Left	327
MD	011110	----- -001- /	rldicr	Rotate Left Doubleword Immediate then Clear Right	328
MD	011110	----- -010- /	rldic	Rotate Left Doubleword Immediate then Clear	329
MD	011110	----- -011- /	rldimi	Rotate Left Doubleword Immediate then Mask Insert	330
MDS	011110	----- -1000 /	rldcl	Rotate Left Doubleword then Clear Left	327
MDS	011110	----- -1001 /	rldcr	Rotate Left Doubleword then Clear Right	328
X	011111	00000 00000 /	cmp	Compare	241
X	011111	00000 00100 /	tw	Trap Word	367
X	011111	00000 01000 0	subfc	Subtract From Carrying	356
X	011111	00000 01000 1	subfc.	Subtract From Carrying & record CR	356

Legend:

- Don't care, usually part of an operand field
/ Reserved bit, invalid instruction form if encoded as 1
? Allocated for implementation-dependent use. See User' Manual for the implementation

Format	Opcode		Mnemonic	Instruction	Page
	Primary ($Inst_{0:5}$)	Extended ($Inst_{21:31}$)			
X	011111	/0000 01001 /	mulhdu	Multiply High Doubleword Unsigned	317
X	011111	00000 01010 0	addc	Add Carrying	230
X	011111	00000 01010 1	addc.	Add Carrying & record CR	230
X	011111	/0000 01011 0	mulhwu	Multiply High Word Unsigned	318
X	011111	/0000 01011 1	mulhwu.	Multiply High Word Unsigned & record CR	318
X	011111	00000 10011 /	mfcr	Move From Condition Register	307
X	011111	00000 10100 /	lwarx	Load Word & Reserve Indexed	300
X	011111	00000 10110 /	icbt	Instruction Cache Block Touch Indexed	287
X	011111	00000 10111 /	lwzx	Load Word & Zero Indexed	303
X	011111	00000 11000 0	slw	Shift Left Word	336
X	011111	00000 11000 1	slw.	Shift Left Word & record CR	336
X	011111	00000 11010 0	cntlzw	Count Leading Zeros Word	243
X	011111	00000 11010 1	cntlzw.	Count Leading Zeros Word & record CR	243
X	011111	00000 11011 /	sld	Shift Left Doubleword	335
X	011111	00000 11100 0	and	AND	236
X	011111	00000 11100 1	and.	AND & record CR	236
X	011111	00000 11110 /	icbte	Instruction Cache Block Touch Indexed Extended	287
X	011111	00000 11111 /	lwzxe	Load Word & Zero Indexed Extended	303
X	011111	00001 00000 /	cmpl	Compare Logical	242
X	011111	00001 01000 0	subf	Subtract From	355
X	011111	00001 01000 1	subf.	Subtract From & record CR	355
X	011111	00001 10110 /	dcbst	Data Cache Block Store Indexed	251
X	011111	00001 10111 /	lwzux	Load Word & Zero with Update Indexed	303
X	011111	00001 11010 /	cntlzd	Count Leading Zeros Doubleword	243
X	011111	00001 11100 0	andc	AND with Complement	236
X	011111	00001 11100 1	andc.	AND with Complement & record CR	236
X	011111	00001 11110 /	dcbste	Data Cache Block Store Indexed Extended	251
X	011111	00001 11111 /	lwzuxe	Load Word & Zero with Update Indexed Extended	303
X	011111	00010 00100 /	td	Trap Doubleword	361
X	011111	/0010 01001 /	mulhd	Multiply High Doubleword	317
X	011111	/0010 01011 0	mulhw	Multiply High Word	318
X	011111	/0010 01011 1	mulhw.	Multiply High Word & record CR	318
X	011111	00010 10011 /	mfmsr	Move From Machine State Register	308
X	011111	00010 10110 /	dcbf	Data Cache Block Flush Indexed	248
X	011111	00010 10111 /	lbzx	Load Byte & Zero Indexed	289
X	011111	00010 11110 /	dcbfe	Data Cache Block Flush Indexed Extended	248
X	011111	00010 11111 /	lbzxe	Load Byte & Zero Indexed Extended	289
X	011111	00011 01000 0	neg	Negate	322
X	011111	00011 01000 1	neg.	Negate & record CR	322
X	011111	00011 10111 /	lbzux	Load Byte & Zero with Update Indexed	289
X	011111	00011 11100 0	nor	NOR	323
X	011111	00011 11100 1	nor.	NOR & record CR	323
X	011111	00011 11110 /	lwarxe	Load Word & Reserve Indexed Extended	300
X	011111	00011 11111 /	lbzuxe	Load Byte & Zero with Update Indexed Extended	289
X	011111	00100 00011 /	wrtee	Write External Enable	368
X	011111	00100 01000 0	subfe	Subtract From Extended with CA	357
X	011111	00100 01000 1	subfe.	Subtract From Extended with CA & record CR	357

Legend:

- Don't care, usually part of an operand field
/ Reserved bit, invalid instruction form if encoded as 1
? Allocated for implementation-dependent use. See User' Manual for the implementation

Format	Opcode		Mnemonic	Instruction	Page
	Primary ($Inst_{0:5}$)	Extended ($Inst_{21:31}$)			
X	011111	00100 01010 0	adde	Add Extended with CA	231
X	011111	00100 01010 1	adde.	Add Extended with CA & record CR	231
XFX	011111	00100 10000 /	mtcrf	Move To Condition Register Fields	311
X	011111	00100 10010 /	mtmsr	Move To Machine State Register	315
X	011111	00100 10110 1	stwcx.	Store Word Conditional Indexed & record CR	353
X	011111	00100 10111 /	stwx	Store Word Indexed	351
X	011111	00100 11110 1	stwcxe.	Store Word Conditional Indexed Extended & record CR	353
X	011111	00100 11111 /	stwxe	Store Word Indexed Extended	351
X	011111	00101 00011 /	wrteei	Write External Enable Immediate	368
X	011111	00101 10111 /	stwux	Store Word with Update Indexed	351
X	011111	00101 11111 /	stwuxe	Store Word with Update Indexed Extended	351
X	011111	00110 01000 0	subfze	Subtract From Zero Extended with CA	360
X	011111	00110 01000 1	subfze.	Subtract From Zero Extended with CA & record CR	360
X	011111	00110 01010 0	addze	Add to Zero Extended with CA	235
X	011111	00110 01010 1	addze.	Add to Zero Extended with CA & record CR	235
X	011111	00110 10111 /	stbx	Store Byte Indexed	341
X	011111	00110 11111 /	stbxe	Store Byte Indexed Extended	341
X	011111	00111 01000 0	subfme	Subtract From Minus One Extended with CA	359
X	011111	00111 01000 1	subfme.	Subtract From Minus One Extended with CA & record CR	359
X	011111	00111 01001 /	mulld	Multiply Low Doubleword	319
X	011111	00111 01010 0	addme	Add to Minus One Extended with CA	234
X	011111	00111 01010 1	addme.	Add to Minus One Extended with CA & record CR	234
X	011111	00111 01011 0	mullw	Multiply Low Word	320
X	011111	00111 01011 1	mullw.	Multiply Low Word & record CR	320
X	011111	00111 10110 /	dcbtst	Data Cache Block Touch for Store Indexed	253
X	011111	00111 10111 /	stbux	Store Byte with Update Indexed	341
X	011111	00111 11110 /	dcbtste	Data Cache Block Touch for Store Indexed Extended	253
X	011111	00111 11111 /	stbuxe	Store Byte with Update Indexed Extended	341
X	011111	01000 01010 0	add	Add	229
X	011111	01000 01010 1	add.	Add & record CR	229
X	011111	01000 10011 /	mfapidi	Move From APID Indirect	307
X	011111	01000 10110 /	dcbt	Data Cache Block Touch Indexed	252
X	011111	01000 10111 /	lhzx	Load Halfword & Zero Indexed	296
X	011111	01000 11100 0	eqv	Equivalent	259
X	011111	01000 11100 1	eqv.	Equivalent & record CR	259
X	011111	01000 11110 /	dcbte	Data Cache Block Touch Indexed Extended	252
X	011111	01000 11111 /	lhzxe	Load Halfword & Zero Indexed Extended	296
X	011111	01001 10111 /	lhzux	Load Halfword & Zero with Update Indexed	296
X	011111	01001 11100 0	xor	XOR	369
X	011111	01001 11100 1	xor.	XOR & record CR	369
X	011111	01001 11111 /	lhzuxe	Load Halfword & Zero with Update Indexed Extended	296
XFX	011111	01010 00011 /	mfdcr	Move From Device Control Register	307
XFX	011111	01010 10011 /	mfspr	Move From Special Purpose Register	309
X	011111	01010 10111 /	lhax	Load Halfword Algebraic Indexed	294
X	011111	01010 11111 /	lhaxe	Load Halfword Algebraic Indexed Extended	294
X	011111	01011 10111 /	lhaux	Load Halfword Algebraic with Update Indexed	294
X	011111	01011 11111 /	lhauxe	Load Halfword Algebraic with Update Indexed Extended	294

Legend:

- \- Don't care, usually part of an operand field
- / Reserved bit, invalid instruction form if encoded as 1
- ? Allocated for implementation-dependent use. See User' Manual for the implementation

Format	Opcode		Mnemonic	Instruction	Page
	Primary ($Inst_{0:5}$)	Extended ($Inst_{21:31}$)			
X	011111	01100 01000 /	subfe64	Subtract From Extended with CA64	357
X	011111	01100 01010 /	adde64	Add Extended with CA64	231
X	011111	01100 10111 /	sthx	Store Halfword Indexed	347
X	011111	01100 11100 0	orc	OR with Complement	324
X	011111	01100 11100 1	orc.	OR with Complement & record CR	324
X	011111	01100 11111 /	sthxe	Store Halfword Indexed Extended	347
X	011111	01101 10111 /	sthux	Store Halfword with Update Indexed	347
X	011111	01101 11100 0	or	OR	324
X	011111	01101 11100 1	or.	OR & record CR	324
X	011111	01101 11111 /	sthuxe	Store Halfword with Update Indexed Extended	347
XFX	011111	01110 00011 /	mtdcr	Move To Device Control Register	311
X	011111	01110 01000 /	subfze64	Subtract From Zero Extended with CA64	360
X	011111	01110 01001 /	divdu	Divide Doubleword Unsigned	256
X	011111	01110 01010 /	addze64	Add to Zero Extended with CA64	235
X	011111	01110 01011 0	divwu	Divide Word Unsigned	258
X	011111	01110 01011 1	divwu.	Divide Word Unsigned & record CR	258
XFX	011111	01110 10011 /	mtspr	Move To Special Purpose Register	316
X	011111	01110 10110 /	dcbi	Data Cache Block Invalidate Indexed	249
X	011111	01110 11100 0	nand	NAND	321
X	011111	01110 11100 1	nand.	NAND & record CR	321
X	011111	01110 11110 /	dcbie	Data Cache Block Invalidate Indexed Extended	249
X	011111	01110 11111 /	ldarxe	Load Doubleword & Reserve Indexed Extended	290
X	011111	01111 01000 /	subfme64	Subtract From Minus One Extended with CA64	359
X	011111	01111 01001 /	divd	Divide Doubleword	255
X	011111	01111 01010 /	addme64	Add to Minus One Extended with CA64	234
X	011111	01111 01011 0	divw	Divide Word	257
X	011111	01111 01011 1	divw.	Divide Word & record CR	257
X	011111	01111 11111 1	stdcxe.	Store Doubleword Conditional Indexed Extended	342
X	011111	10000 00000 /	mcrxr	Move to Condition Register from XER	306
X	011111	10000 01000 0	subfco	Subtract From Carrying & record OV	356
X	011111	10000 01000 1	subfco.	Subtract From Carrying & record OV & CR	356
X	011111	10000 01010 0	addco	Add Carrying & record OV	230
X	011111	10000 01010 1	addco.	Add Carrying & record OV & CR	230
X	011111	10000 10101 /	lswx	Load String Word Indexed	298
X	011111	10000 10110 /	lwbrx	Load Word Byte-Reverse Indexed	302
X	011111	10000 10111 /	lfsx	Load Floating-Point Single Indexed	293
X	011111	10000 11000 0	srw	Shift Right Word	340
X	011111	10000 11000 1	srw.	Shift Right Word & record CR	340
X	011111	10000 11011 /	srd	Shift Right Doubleword	339
X	011111	10000 11110 /	lwbrxe	Load Word Byte-Reverse Indexed Extended	302
X	011111	10000 11111 /	lfsxe	Load Floating-Point Single Indexed Extended	293
X	011111	10001 00000 /	mcrxr64	Move to Condition Register from XER64	306
X	011111	10001 01000 0	subfo	Subtract From & record OV	355
X	011111	10001 01000 1	subfo.	Subtract From & record OV & CR	355
X	011111	10001 10110 /	tlbsync	TLB Synchronize	365
X	011111	10001 10111 /	lfsux	Load Floating-Point Single with Update Indexed	293
X	011111	10001 11111 /	lfsuxe	Load Floating-Point Single with Update Indexed Extended	293

Legend:

- Don't care, usually part of an operand field
/ Reserved bit, invalid instruction form if encoded as 1
? Allocated for implementation-dependent use. See User' Manual for the implementation

Format	Opcode		Mnemonic	Instruction	Page
	Primary ($Inst_{0:5}$)	Extended ($Inst_{21:31}$)			
X	011111	10010 10101 /	lswi	Load String Word Immediate	298
X	011111	10010 10110 /	msync	Memory Synchronize	310
X	011111	10010 10111 /	lfdx	Load Floating-Point Double Indexed	292
X	011111	10010 11111 /	lfdxe	Load Floating-Point Double Indexed Extended	292
X	011111	10011 01000 0	nego	Negate & record OV	322
X	011111	10011 01000 1	nego.	Negate & record OV & record CR	322
X	011111	10011 10111 /	lfdux	Load Floating-Point Double with Update Indexed	292
X	011111	10011 11111 /	lfduxe	Load Floating-Point Double with Update Indexed Extended	292
X	011111	10100 01000 0	subfeo	Subtract From Extended with CA & record OV	357
X	011111	10100 01000 1	subfeo.	Subtract From Extended with CA & record OV & CR	357
X	011111	10100 01010 0	addeo	Add Extended with CA & record OV	231
X	011111	10100 01010 1	addeo.	Add Extended with CA & record OV & CR	231
X	011111	10100 10101 /	stswx	Store String Word Indexed	350
X	011111	10100 10110 /	stwbrx	Store Word Byte-Reverse Indexed	352
X	011111	10100 10111 /	stfsx	Store Floating-Point Single Indexed	346
X	011111	10100 11110 /	stwbrxe	Store Word Byte-Reverse Indexed Extended	352
X	011111	10100 11111 /	stfsxe	Store Floating-Point Single Indexed Extended	346
X	011111	10101 10111 /	stfsux	Store Floating-Point Single with Update Indexed	346
X	011111	10101 11111 /	stfsuxe	Store Floating-Point Single with Update Indexed Extended	346
X	011111	10110 01000 0	subfzeo	Subtract From Zero Extended with CA & record OV	360
X	011111	10110 01000 1	subfzeo.	Subtract From Zero Extended with CA & record OV & CR	360
X	011111	10110 01010 0	addzeo	Add to Zero Extended with CA & record OV	235
X	011111	10110 01010 1	addzeo.	Add to Zero Extended with CA & record OV & CR	235
X	011111	10110 10101 /	stswi	Store String Word Immediate	350
X	011111	10110 10111 /	stfdx	Store Floating-Point Double Indexed	344
X	011111	10110 11111 /	stfdxe	Store Floating-Point Double Indexed Extended	344
X	011111	10111 01000 0	subfmeo	Subtract From Minus One Extended with CA & record OV	359
X	011111	10111 01000 1	subfmeo.	Subtract From Minus One Extended with CA & record OV & CR	359
X	011111	10111 01001 /	mulldo	Multiply Low Doubleword & record OV	319
X	011111	10111 01010 0	addmeo	Add to Minus One Extended with CA & record OV	234
X	011111	10111 01010 1	addmeo.	Add to Minus One Extended with CA & record OV & CR	234
X	011111	10111 01011 0	mullwo	Multiply Low Word & record OV	320
X	011111	10111 01011 1	mullwo.	Multiply Low Word & record OV & CR	320
X	011111	10111 10110 /	dcba	Data Cache Block Allocate Indexed	247
X	011111	10111 10111 /	stfdux	Store Floating-Point Double with Update Indexed	344
X	011111	10111 11110 /	dcbae	Data Cache Block Allocate Indexed Extended	247
X	011111	10111 11111 /	stfduxe	Store Floating-Point Double with Update Indexed Extended	344
X	011111	11000 01010 0	addo	Add & record OV	229
X	011111	11000 01010 1	addo.	Add & record OV & CR	229
X	011111	11000 10010 /	tlbivax	TLB Invalidate Virtual Address Indexed	362
X	011111	11000 10011 /	tlbivaxe	TLB Invalidate Virtual Address Indexed Extended	362
X	011111	11000 10110 /	lhbrx	Load Halfword Byte-Reverse Indexed	295
X	011111	11000 11000 0	sraw	Shift Right Algebraic Word	338
X	011111	11000 11000 1	sraw.	Shift Right Algebraic Word & record CR	338
X	011111	11000 11010 /	srad	Shift Right Algebraic Doubleword	337
X	011111	11000 11110 /	lhbrxe	Load Halfword Byte-Reverse Indexed Extended	295
X	011111	11000 11111 /	ldxe	Load Doubleword Indexed Extended	291

Legend:

- Don't care, usually part of an operand field

/ Reserved bit, invalid instruction form if encoded as 1

? Allocated for implementation-dependent use. See User' Manual for the implementation

Format	Opcode		Mnemonic	Instruction	Page
	Primary ($Inst_{0:5}$)	Extended ($Inst_{21:31}$)			
X	011111	11001 11000 0	srawi	Shift Right Algebraic Word Immediate	338
X	011111	11001 11000 1	srawi.	Shift Right Algebraic Word Immediate & record CR	338
XS	011111	11001 1101- /	sradi	Shift Right Algebraic Doubleword Immediate	337
X	011111	11001 11111 /	lduxe	Load Doubleword with Update Indexed Extended	291
X	011111	11010 10110 /	mbar	Memory Barrier	304
X	011111	11100 01000 /	subfe64o	Subtract From Extended with CA64 & record OV	357
X	011111	11100 01010 /	adde64o	Add Extended with CA64 & record OV	231
X	011111	11100 10010 ?	tlbsx	TLB Search Indexed	364
X	011111	11100 10011 ?	tlbsxe	TLB Search Indexed Extended	364
X	011111	11100 10110 /	sthbrx	Store Halfword Byte-Reverse Indexed	348
X	011111	11100 11010 0	extsh	Extend Sign Halfword	260
X	011111	11100 11010 1	extsh.	Extend Sign Halfword & record CR	260
X	011111	11100 11110 /	sthbrxe	Store Halfword Byte-Reverse Indexed Extended	348
X	011111	11100 11111 /	stdxe	Store Doubleword Indexed Extended	343
X	011111	11101 10010 /	tlbre	TLB Read Entry	363
X	011111	11101 11010 0	extsb	Extend Sign Byte	260
X	011111	11101 11010 1	extsb.	Extend Sign Byte & record CR	260
X	011111	11101 11111 /	stduxe	Store Doubleword with Update Indexed Extended	343
X	011111	11110 01000 /	subfze64o	Subtract From Zero Extended with CA64 & record OV	360
X	011111	11110 01001 /	divduo	Divide Doubleword Unsigned & record OV	256
X	011111	11110 01010 /	addze64o	Add to Zero Extended with CA64 & record OV	235
X	011111	11110 01011 0	divwuo	Divide Word Unsigned & record OV	258
X	011111	11110 01011 1	divwuo.	Divide Word Unsigned & record OV & CR	258
X	011111	11110 10010 /	tlbwe	TLB Write Entry	366
X	011111	11110 10110 /	icbi	Instruction Cache Block Invalidate Indexed	286
X	011111	11110 10111 /	stfiwx	Store Floating-Point as Int Word Indexed	345
X	011111	11110 11010 /	extsw	Extend Sign Word	260
X	011111	11110 11110 /	icbie	Instruction Cache Block Invalidate Indexed Extended	286
X	011111	11110 11111 /	stfiwxe	Store Floating-Point as Int Word Indexed Extended	345
X	011111	11111 01000 /	subfme64o	Subtract From Minus One Extended with CA64 & record OV	359
X	011111	11111 01001 /	divdo	Divide Doubleword & record OV	255
X	011111	11111 01010 /	addme64o	Add to Minus One Extended with CA64 & record OV	234
X	011111	11111 01011 0	divwo	Divide Word & record OV	257
X	011111	11111 01011 1	divwo.	Divide Word & record OV & CR	257
X	011111	11111 10110 /	dcbz	Data Cache Block set to Zero Indexed	254
X	011111	11111 11110 /	dcbze	Data Cache Block set to Zero Indexed Extended	254
D	100000	----- ----- -	lwz	Load Word & Zero	303
D	100001	----- ----- -	lwzu	Load Word & Zero with Update	303
D	100010	----- ----- -	lbz	Load Byte & Zero	289
D	100011	----- ----- -	lbzu	Load Byte & Zero with Update	289
D	100100	----- ----- -	stw	Store Word	351
D	100101	----- ----- -	stwu	Store Word with Update	351
D	100110	----- ----- -	stb	Store Byte	341
D	100111	----- ----- -	stbu	Store Byte with Update	341
D	101000	----- ----- -	lhz	Load Halfword & Zero	296
D	101001	----- ----- -	lhzu	Load Halfword & Zero with Update	296
D	101010	----- ----- -	lha	Load Halfword Algebraic	294

Legend:

- `-` Don't care, usually part of an operand field
- `/` Reserved bit, invalid instruction form if encoded as 1
- `?` Allocated for implementation-dependent use. See User' Manual for the implementation

Format	Opcode		Mnemonic	Instruction	Page
	Primary ($Inst_{0:5}$)	Extended ($Inst_{21:31}$)			
D	101011	----- ----- -	lhau	Load Halfword Algebraic with Update	294
D	101100	----- ----- -	sth	Store Halfword	347
D	101101	----- ----- -	sthu	Store Halfword with Update	347
D	101110	----- ----- -	lmw	Load Multiple Word	297
D	101111	----- ----- -	stmw	Store Multiple Word	349
D	110000	----- ----- -	lfs	Load Floating-Point Single	293
D	110001	----- ----- -	lfsu	Load Floating-Point Single with Update	293
D	110010	----- ----- -	lfd	Load Floating-Point Double	292
D	110011	----- ----- -	lfdu	Load Floating-Point Double with Update	292
D	110100	----- ----- -	stfs	Store Floating-Point Single	346
D	110101	----- ----- -	stfsu	Store Floating-Point Single with Update	346
D	110110	----- ----- -	stfd	Store Floating-Point Double	344
D	110111	----- ----- -	stfdu	Store Floating-Point Double with Update	344
DE	111010	----- --000 0	lbze	Load Byte & Zero Extended	289
DE	111010	----- --000 1	lbzue	Load Byte & Zero with Update Extended	289
DE	111010	----- --001 0	lhze	Load Halfword & Zero Extended	296
DE	111010	----- --001 1	lhzue	Load Halfword & Zero with Update Extended	296
DE	111010	----- --010 0	lhae	Load Halfword Algebraic Extended	294
DE	111010	----- --010 1	lhaue	Load Halfword Algebraic with Update Extended	294
DE	111010	----- --011 0	lwze	Load Word & Zero Extended	303
DE	111010	----- --011 1	lwzue	Load Word & Zero with Update Extended	303
DE	111010	----- --100 0	stbe	Store Byte Extended	341
DE	111010	----- --100 1	stbue	Store Byte with Update Extended	341
DE	111010	----- --101 0	sthe	Store Halfword Extended	347
DE	111010	----- --101 1	sthue	Store Halfword with Update Extended	347
DE	111010	----- --111 0	stwe	Store Word Extended	351
DE	111010	----- --111 1	stwue	Store Word with Update Extended	351
A	111011	----- 10010 0	fdivs	Floating Divide Single	270
A	111011	----- 10010 1	fdivs.	Floating Divide Single & record CR	270
A	111011	----- 10100 0	fsubs	Floating Subtract Single	285
A	111011	----- 10100 1	fsubs.	Floating Subtract Single & record CR	285
A	111011	----- 10101 0	fadds	Floating Add Single	262
A	111011	----- 10101 1	fadds.	Floating Add Single & record CR	262
A	111011	----- 10110 0	fsqrts	Floating Square Root Single	284
A	111011	----- 10110 1	fsqrts.	Floating Square Root Single & record CR	284
A	111011	----- 11000 0	fres	Floating Reciprocal Estimate Single	278
A	111011	----- 11000 1	fres.	Floating Reciprocal Estimate Single & record CR	278
A	111011	----- 11001 0	fmuls	Floating Multiply Single	274
A	111011	----- 11001 1	fmuls.	Floating Multiply Single & record CR	274
A	111011	----- 11100 0	fmsubs	Floating Multiply-Subtract Single	273
A	111011	----- 11100 1	fmsubs.	Floating Multiply-Subtract Single & record CR	273
A	111011	----- 11101 0	fmadds	Floating Multiply-Add Single	271
A	111011	----- 11101 1	fmadds.	Floating Multiply-Add Single & record CR	271
A	111011	----- 11110 0	fnmsubs	Floating Negative Multiply-Subtract Single	277
A	111011	----- 11110 1	fnmsubs.	Floating Negative Multiply-Subtract Single & record CR	277
A	111011	----- 11111 0	fnmadds	Floating Negative Multiply-Add Single	276
A	111011	----- 11111 1	fnmadds.	Floating Negative Multiply-Add Single & record CR	276

Legend:

- Don't care, usually part of an operand field
/ Reserved bit, invalid instruction form if encoded as 1
? Allocated for implementation-dependent use. See User' Manual for the implementation

Format	Opcode		Mnemonic	Instruction	Page
	Primary ($Inst_{0:5}$)	Extended ($Inst_{21:31}$)			
DES	111110	----- --000 0	lde	Load Doubleword Extended	291
DES	111110	----- --000 1	ldue	Load Doubleword with Update Extended	291
DES	111110	----- --010 0	lfse	Load Floating-Point Single Extended	293
DES	111110	----- --010 1	lfsue	Load Floating-Point Single with Update Extended	293
DES	111110	----- --011 0	lfde	Load Floating-Point Double Extended	292
DES	111110	----- --011 1	lfdue	Load Floating-Point Double with Update Extended	292
DES	111110	----- --100 0	stde	Store Doubleword Extended	343
DES	111110	----- --100 1	stdue	Store Doubleword with Update Extended	343
DES	111110	----- --110 0	stfse	Store Floating-Point Single Extended	346
DES	111110	----- --110 1	stfsue	Store Floating-Point Single with Update Extended	346
DES	111110	----- --111 0	stfde	Store Floating-Point Double Extended	344
DES	111110	----- --111 1	stfdue	Store Floating-Point Double with Update Extended	344
A	111111	----- 10010 0	fdiv	Floating Divide	270
A	111111	----- 10010 1	fdiv.	Floating Divide & record CR	270
A	111111	----- 10100 0	fsub	Floating Subtract	285
A	111111	----- 10100 1	fsub.	Floating Subtract & record CR	285
A	111111	----- 10101 0	fadd	Floating Add	262
A	111111	----- 10101 1	fadd.	Floating Add & record CR	262
A	111111	----- 10110 0	fsqrt	Floating Square Root	284
A	111111	----- 10110 1	fsqrt.	Floating Square Root & record CR	284
A	111111	----- 10111 0	fsel	Floating Select	283
A	111111	----- 10111 1	fsel.	Floating Select & record CR	283
A	111111	----- 11001 0	fmul	Floating Multiply	274
A	111111	----- 11001 1	fmul.	Floating Multiply & record CR	274
A	111111	----- 11010 0	frsqrte	Floating Reciprocal Square Root Estimate	282
A	111111	----- 11010 1	frsqrte.	Floating Reciprocal Square Root Estimate & record CR	282
A	111111	----- 11100 0	fmsub	Floating Multiply-Subtract	273
A	111111	----- 11100 1	fmsub.	Floating Multiply-Subtract & record CR	273
A	111111	----- 11101 0	fmadd	Floating Multiply-Add	271
A	111111	----- 11101 1	fmadd.	Floating Multiply-Add & record CR	271
A	111111	----- 11110 0	fnmsub	Floating Negative Multiply-Subtract	277
A	111111	----- 11110 1	fnmsub.	Floating Negative Multiply-Subtract & record CR	277
A	111111	----- 11111 0	fnmadd	Floating Negative Multiply-Add	276
A	111111	----- 11111 1	fnmadd.	Floating Negative Multiply-Add & record CR	276
X	111111	00000 00000 /	fcmpu	Floating Compare Unordered	265
X	111111	00000 01100 0	frsp	Floating Round to Single-Precision	279
X	111111	00000 01100 1	frsp.	Floating Round to Single-Precision & record CR	279
X	111111	00000 01110 0	fctiw	Floating Convert To Int Word	268
X	111111	00000 01110 1	fctiw.	Floating Convert To Int Word & record CR	268
X	111111	00000 01111 0	fctiwz	Floating Convert To Int Word with round to Zero	268
X	111111	00000 01111 1	fctiwz.	Floating Convert To Int Word with round to Zero & record CR	268
X	111111	00001 00000 /	fcmpo	Floating Compare Ordered	265
X	111111	00001 00110 0	mtfsb1	Move To FPSCR Bit 1	312
X	111111	00001 00110 1	mtfsb1.	Move To FPSCR Bit 1 & record CR	312
X	111111	00001 01000 0	fneg	Floating Negate	275
X	111111	00001 01000 1	fneg.	Floating Negate & record CR	275
X	111111	00010 00000 /	mcrfs	Move to Condition Register from FPSCR	306

Legend:

- \- Don't care, usually part of an operand field
- / Reserved bit, invalid instruction form if encoded as 1
- ? Allocated for implementation-dependent use. See User' Manual for the implementation

Format	Opcode		Mnemonic	Instruction	Page
	Primary ($Inst_{0:5}$)	Extended ($Inst_{21:31}$)			
X	111111	00010 00110 0	mtfsb0	Move To FPSCR Bit 0	312
X	111111	00010 00110 1	mtfsb0.	Move To FPSCR Bit 0 & record CR	312
X	111111	00010 01000 0	fmr	Floating Move Register	272
X	111111	00010 01000 1	fmr.	Floating Move Register & record CR	272
X	111111	00100 00110 0	mtfsfi	Move To FPSCR Field Immediate	314
X	111111	00100 00110 1	mtfsfi.	Move To FPSCR Field Immediate & record CR	314
X	111111	00100 01000 0	fnabs	Floating Negative Absolute Value	275
X	111111	00100 01000 1	fnabs.	Floating Negative Absolute Value & record CR	275
X	111111	01000 01000 0	fabs	Floating Absolute Value	261
X	111111	01000 01000 1	fabs.	Floating Absolute Value & record CR	261
X	111111	10010 00111 0	mffs	Move From FPSCR	308
X	111111	10010 00111 1	mffs.	Move From FPSCR & record CR	308
XFL	111111	10110 00111 0	mtfsf	Move To FPSCR Fields	313
XFL	111111	10110 00111 1	mtfsf.	Move To FPSCR Fields & record CR	313
X	111111	11001 01110 /	fctid	Floating Convert To Int Doubleword	266
X	111111	11001 01111 /	fctidz	Floating Convert To Int Doubleword with round to Zero	266
X	111111	11010 01110 /	fcfid	Floating Convert From Int Doubleword	263

Legend:

- Don't care, usually part of an operand field

/ Reserved bit, invalid instruction form if encoded as 1

? Allocated for implementation-dependent use. See User' Manual for the implementation

H.2 Instruction Index Sorted by Mnemonic

This appendix lists all the instructions in the Book E, in order by mnemonic.

Format	Opcode		Mnemonic	Instruction	Page
	Primary ($Inst_{0:5}$)	Extended ($Inst_{21:31}$)			
X	011111	01000 01010 0	add	Add	229
X	011111	01000 01010 1	add.	Add & record CR	229
X	011111	00000 01010 0	addc	Add Carrying	230
X	011111	00000 01010 1	addc.	Add Carrying & record CR	230
X	011111	10000 01010 0	addco	Add Carrying & record OV	230
X	011111	10000 01010 1	addco.	Add Carrying & record OV & CR	230
X	011111	00100 01010 0	adde	Add Extended with CA	231
X	011111	00100 01010 1	adde.	Add Extended with CA & record CR	231
X	011111	01100 01010 /	adde64	Add Extended with CA64	231
X	011111	11100 01010 /	adde64o	Add Extended with CA64 & record OV	231
X	011111	10100 01010 0	addeo	Add Extended with CA & record OV	231
X	011111	10100 01010 1	addeo.	Add Extended with CA & record OV & CR	231
D	001110	----- ----- -	addi	Add Immediate	232
D	001100	----- ----- -	addic	Add Immediate Carrying	233
D	001101	----- ----- -	addic.	Add Immediate Carrying & record CR	233
D	001111	----- ----- -	addis	Add Immediate Shifted	232
X	011111	00111 01010 0	addme	Add to Minus One Extended with CA	234
X	011111	00111 01010 1	addme.	Add to Minus One Extended with CA & record CR	234
X	011111	01111 01010 /	addme64	Add to Minus One Extended with CA64	234
X	011111	11111 01010 /	addme64o	Add to Minus One Extended with CA64 & record OV	234
X	011111	10111 01010 0	addmeo	Add to Minus One Extended with CA & record OV	234
X	011111	10111 01010 1	addmeo.	Add to Minus One Extended with CA & record OV & CR	234
X	011111	11000 01010 0	addo	Add & record OV	229
X	011111	11000 01010 1	addo.	Add & record OV & CR	229
X	011111	00110 01010 0	addze	Add to Zero Extended with CA	235
X	011111	00110 01010 1	addze.	Add to Zero Extended with CA & record CR	235
X	011111	01110 01010 /	addze64	Add to Zero Extended with CA64	235
X	011111	11110 01010 /	addze64o	Add to Zero Extended with CA64 & record OV	235
X	011111	10110 01010 0	addzeo	Add to Zero Extended with CA & record OV	235
X	011111	10110 01010 1	addzeo.	Add to Zero Extended with CA & record OV & CR	235
X	011111	00000 11100 0	and	AND	236
X	011111	00000 11100 1	and.	AND & record CR	236
X	011111	00001 11100 0	andc	AND with Complement	236
X	011111	00001 11100 1	andc.	AND with Complement & record CR	236
D	011100	----- ----- -	andi.	AND Immediate & record CR	236
D	011101	----- ----- -	andis.	AND Immediate Shifted & record CR	236
I	010010	----- ----0 0	b	Branch	237
I	010010	----- ----1 0	ba	Branch Absolute	237
B	010000	----- ----0 0	bc	Branch Conditional	238
B	010000	----- ----1 0	bca	Branch Conditional Absolute	238
XL	010011	10000 10000 0	bcctr	Branch Conditional to Count Register	239
XL	010011	10000 10001 0	bcctre	Branch Conditional to Count Register Extended	239

Legend:

- Don't care, usually part of an operand field
/ Reserved bit, invalid instruction form if encoded as 1
? Allocated for implementation-dependent use. See User' Manual for the implementation

Format	Opcode		Mnemonic	Instruction	Page
	Primary ($Inst_{0:5}$)	Extended ($Inst_{21:31}$)			
XL	010011	10000 10001 1	bcctrel	Branch Conditional to Count Register Extended & Link	239
XL	010011	10000 10000 1	bcctrl	Branch Conditional to Count Register & Link	239
B	001001	----- ----0 0	bce	Branch Conditional Extended	238
B	001001	----- ----1 0	bcea	Branch Conditional Extended Absolute	238
B	001001	----- ----0 1	bcel	Branch Conditional Extended & Link	238
B	001001	----- ----1 1	bcela	Branch Conditional Extended & Link Absolute	238
B	010000	----- ----0 1	bcl	Branch Conditional & Link	238
B	010000	----- ----1 1	bcla	Branch Conditional & Link Absolute	238
XL	010011	00000 10000 0	bclr	Branch Conditional to Link Register	240
XL	010011	00000 10001 0	bclre	Branch Conditional to Link Register Extended	240
XL	010011	00000 10001 1	bclrel	Branch Conditional to Link Register Extended & Link	240
XL	010011	00000 10000 1	bclrl	Branch Conditional to Link Register & Link	240
I	010110	----- ----0 0	be	Branch Extended	238
I	010110	----- ----1 0	bea	Branch Extended Absolute	238
I	010110	----- ----0 1	bel	Branch Extended & Link	238
I	010110	----- ----1 1	bela	Branch Extended & Link Absolute	238
I	010010	----- ----0 1	bl	Branch & Link	237
I	010010	----- ----1 1	bla	Branch & Link Absolute	237
X	011111	00000 00000 /	cmp	Compare	241
D	001011	----- ----- -	cmpi	Compare Immediate	241
X	011111	00001 00000 /	cmpl	Compare Logical	242
D	001010	----- ----- -	cmpli	Compare Logical Immediate	242
X	011111	00001 11010 /	cntlzd	Count Leading Zeros Doubleword	243
X	011111	00000 11010 0	cntlzw	Count Leading Zeros Word	243
X	011111	00000 11010 1	cntlzw.	Count Leading Zeros Word & record CR	243
XL	010011	01000 00001 /	crand	Condition Register AND	244
XL	010011	00100 00001 /	crandc	Condition Register AND with Complement	244
XL	010011	01001 00001 /	creqv	Condition Register Equivalent	244
XL	010011	00111 00001 /	crnand	Condition Register NAND	245
XL	010011	00001 00001 /	crnor	Condition Register NOR	245
XL	010011	01110 00001 /	cror	Condition Register OR	245
XL	010011	01101 00001 /	crorc	Condition Register OR with Complement	246
XL	010011	00110 00001 /	crxor	Condition Register XOR	246
X	011111	10111 10110 /	dcba	Data Cache Block Allocate Indexed	247
X	011111	10111 11110 /	dcbae	Data Cache Block Allocate Indexed Extended	247
X	011111	00010 10110 /	dcbf	Data Cache Block Flush Indexed	248
X	011111	00010 11110 /	dcbfe	Data Cache Block Flush Indexed Extended	248
X	011111	01110 10110 /	dcbi	Data Cache Block Invalidate Indexed	249
X	011111	01110 11110 /	dcbie	Data Cache Block Invalidate Indexed Extended	249
X	011111	00001 10110 /	dcbst	Data Cache Block Store Indexed	251
X	011111	00001 11110 /	dcbste	Data Cache Block Store Indexed Extended	251
X	011111	01000 10110 /	dcbt	Data Cache Block Touch Indexed	252
X	011111	01000 11110 /	dcbte	Data Cache Block Touch Indexed Extended	252
X	011111	00111 10110 /	dcbtst	Data Cache Block Touch for Store Indexed	253
X	011111	00111 11110 /	dcbtste	Data Cache Block Touch for Store Indexed Extended	253
X	011111	11111 10110 /	dcbz	Data Cache Block set to Zero Indexed	254
X	011111	11111 11110 /	dcbze	Data Cache Block set to Zero Indexed Extended	254

Legend:

- Don't care, usually part of an operand field
/ Reserved bit, invalid instruction form if encoded as 1
? Allocated for implementation-dependent use. See User' Manual for the implementation

Format	Opcode		Mnemonic	Instruction	Page
	Primary ($Inst_{0:5}$)	Extended ($Inst_{21:31}$)			
X	011111	01111 01001 /	divd	Divide Doubleword	255
X	011111	11111 01001 /	divdo	Divide Doubleword & record OV	255
X	011111	01110 01001 /	divdu	Divide Doubleword Unsigned	256
X	011111	11110 01001 /	divduo	Divide Doubleword Unsigned & record OV	256
X	011111	01111 01011 0	divw	Divide Word	257
X	011111	01111 01011 1	divw.	Divide Word & record CR	257
X	011111	11111 01011 0	divwo	Divide Word & record OV	257
X	011111	11111 01011 1	divwo.	Divide Word & record OV & CR	257
X	011111	01110 01011 0	divwu	Divide Word Unsigned	258
X	011111	01110 01011 1	divwu.	Divide Word Unsigned & record CR	258
X	011111	11110 01011 0	divwuo	Divide Word Unsigned & record OV	258
X	011111	11110 01011 1	divwuo.	Divide Word Unsigned & record OV & CR	258
X	011111	01000 11100 0	eqv	Equivalent	259
X	011111	01000 11100 1	eqv.	Equivalent & record CR	259
X	011111	11101 11010 0	extsb	Extend Sign Byte	260
X	011111	11101 11010 1	extsb.	Extend Sign Byte & record CR	260
X	011111	11100 11010 0	extsh	Extend Sign Halfword	260
X	011111	11100 11010 1	extsh.	Extend Sign Halfword & record CR	260
X	011111	11110 11010 /	extsw	Extend Sign Word	260
X	111111	01000 01000 0	fabs	Floating Absolute Value	261
X	111111	01000 01000 1	fabs.	Floating Absolute Value & record CR	261
A	111111	----- 10101 0	fadd	Floating Add	262
A	111111	----- 10101 1	fadd.	Floating Add & record CR	262
A	111011	----- 10101 0	fadds	Floating Add Single	262
A	111011	----- 10101 1	fadds.	Floating Add Single & record CR	262
X	111111	11010 01110 /	fcfid	Floating Convert From Int Doubleword	263
X	111111	00001 00000 /	fcmpo	Floating Compare Ordered	265
X	111111	00000 00000 /	fcmpu	Floating Compare Unordered	265
X	111111	11001 01110 /	fctid	Floating Convert To Int Doubleword	266
X	111111	11001 01111 /	fctidz	Floating Convert To Int Doubleword with round to Zero	266
X	111111	00000 01110 0	fctiw	Floating Convert To Int Word	268
X	111111	00000 01110 1	fctiw.	Floating Convert To Int Word & record CR	268
X	111111	00000 01111 0	fctiwz	Floating Convert To Int Word with round to Zero	268
X	111111	00000 01111 1	fctiwz.	Floating Convert To Int Word with round to Zero & record CR	268
A	111111	----- 10010 0	fdiv	Floating Divide	270
A	111111	----- 10010 1	fdiv.	Floating Divide & record CR	270
A	111011	----- 10010 0	fdivs	Floating Divide Single	270
A	111011	----- 10010 1	fdivs.	Floating Divide Single & record CR	270
A	111111	----- 11101 0	fmadd	Floating Multiply-Add	271
A	111111	----- 11101 1	fmadd.	Floating Multiply-Add & record CR	271
A	111011	----- 11101 0	fmadds	Floating Multiply-Add Single	271
A	111011	----- 11101 1	fmadds.	Floating Multiply-Add Single & record CR	271
X	111111	00010 01000 0	fmr	Floating Move Register	272
X	111111	00010 01000 1	fmr.	Floating Move Register & record CR	272
A	111111	----- 11100 0	fmsub	Floating Multiply-Subtract	273
A	111111	----- 11100 1	fmsub.	Floating Multiply-Subtract & record CR	273
A	111011	----- 11100 0	fmsubs	Floating Multiply-Subtract Single	273

Legend:

- \- Don't care, usually part of an operand field
- / Reserved bit, invalid instruction form if encoded as 1
- ? Allocated for implementation-dependent use. See User' Manual for the implementation

Format	Opcode		Mnemonic	Instruction	Page
	Primary ($Inst_{0:5}$)	Extended ($Inst_{21:31}$)			
A	111011	----- 11100 1	fmsubs.	Floating Multiply-Subtract Single & record CR	273
A	111111	----- 11001 0	fmul	Floating Multiply	274
A	111111	----- 11001 1	fmul.	Floating Multiply & record CR	274
A	111011	----- 11001 0	fmuls	Floating Multiply Single	274
A	111011	----- 11001 1	fmuls.	Floating Multiply Single & record CR	274
X	111111	00100 01000 0	fnabs	Floating Negative Absolute Value	275
X	111111	00100 01000 1	fnabs.	Floating Negative Absolute Value & record CR	275
X	111111	00001 01000 0	fneg	Floating Negate	275
X	111111	00001 01000 1	fneg.	Floating Negate & record CR	275
A	111111	----- 11111 0	fnmadd	Floating Negative Multiply-Add	276
A	111111	----- 11111 1	fnmadd.	Floating Negative Multiply-Add & record CR	276
A	111011	----- 11111 0	fnmadds	Floating Negative Multiply-Add Single	276
A	111011	----- 11111 1	fnmadds.	Floating Negative Multiply-Add Single & record CR	276
A	111111	----- 11110 0	fnmsub	Floating Negative Multiply-Subtract	277
A	111111	----- 11110 1	fnmsub.	Floating Negative Multiply-Subtract & record CR	277
A	111011	----- 11110 0	fnmsubs	Floating Negative Multiply-Subtract Single	277
A	111011	----- 11110 1	fnmsubs.	Floating Negative Multiply-Subtract Single & record CR	277
A	111011	----- 11000 0	fres	Floating Reciprocal Estimate Single	278
A	111011	----- 11000 1	fres.	Floating Reciprocal Estimate Single & record CR	278
X	111111	00000 01100 0	frsp	Floating Round to Single-Precision	279
X	111111	00000 01100 1	frsp.	Floating Round to Single-Precision & record CR	279
A	111111	----- 11010 0	frsqrte	Floating Reciprocal Square Root Estimate	282
A	111111	----- 11010 1	frsqrte.	Floating Reciprocal Square Root Estimate & record CR	282
A	111111	----- 10111 0	fsel	Floating Select	283
A	111111	----- 10111 1	fsel.	Floating Select & record CR	283
A	111111	----- 10110 0	fsqrt	Floating Square Root	284
A	111111	----- 10110 1	fsqrt.	Floating Square Root & record CR	284
A	111011	----- 10110 0	fsqrts	Floating Square Root Single	284
A	111011	----- 10110 1	fsqrts.	Floating Square Root Single & record CR	284
A	111111	----- 10100 0	fsub	Floating Subtract	285
A	111111	----- 10100 1	fsub.	Floating Subtract & record CR	285
A	111011	----- 10100 0	fsubs	Floating Subtract Single	285
A	111011	----- 10100 1	fsubs.	Floating Subtract Single & record CR	285
X	011111	11110 10110 /	icbi	Instruction Cache Block Invalidate Indexed	286
X	011111	11110 11110 /	icbie	Instruction Cache Block Invalidate Indexed Extended	286
X	011111	00000 10110 /	icbt	Instruction Cache Block Touch Indexed	287
X	011111	00000 11110 /	icbte	Instruction Cache Block Touch Indexed Extended	287
XL	010011	00100 10110 /	isync	Instruction Synchronize	288
D	100010	----- ----- -	lbz	Load Byte & Zero	289
DE	111010	----- --000 0	lbze	Load Byte & Zero Extended	289
D	100011	----- ----- -	lbzu	Load Byte & Zero with Update	289
DE	111010	----- --000 1	lbzue	Load Byte & Zero with Update Extended	289
X	011111	00011 10111 /	lbzux	Load Byte & Zero with Update Indexed	289
X	011111	00011 11111 /	lbzuxe	Load Byte & Zero with Update Indexed Extended	289
X	011111	00010 10111 /	lbzx	Load Byte & Zero Indexed	289
X	011111	00010 11111 /	lbzxe	Load Byte & Zero Indexed Extended	289
X	011111	01110 11111 /	ldarxe	Load Doubleword & Reserve Indexed Extended	290

Legend:

- Don't care, usually part of an operand field
/ Reserved bit, invalid instruction form if encoded as 1
? Allocated for implementation-dependent use. See User' Manual for the implementation

Format	Opcode		Mnemonic	Instruction	Page
	Primary ($Inst_{0:5}$)	Extended ($Inst_{21:31}$)			
DES	111110	----- --000 0	lde	Load Doubleword Extended	291
DES	111110	----- --000 1	ldue	Load Doubleword with Update Extended	291
X	011111	11001 11111 /	lduxe	Load Doubleword with Update Indexed Extended	291
X	011111	11000 11111 /	ldxe	Load Doubleword Indexed Extended	291
D	110010	----- ----- -	lfd	Load Floating-Point Double	292
DES	111110	----- --011 0	lfde	Load Floating-Point Double Extended	292
D	110011	----- ----- -	lfdu	Load Floating-Point Double with Update	292
DES	111110	----- --011 1	lfdue	Load Floating-Point Double with Update Extended	292
X	011111	10011 10111 /	lfdux	Load Floating-Point Double with Update Indexed	292
X	011111	10011 11111 /	lfduxe	Load Floating-Point Double with Update Indexed Extended	292
X	011111	10010 10111 /	lfdx	Load Floating-Point Double Indexed	292
X	011111	10010 11111 /	lfdxe	Load Floating-Point Double Indexed Extended	292
D	110000	----- ----- -	lfs	Load Floating-Point Single	293
DES	111110	----- --010 0	lfse	Load Floating-Point Single Extended	293
D	110001	----- ----- -	lfsu	Load Floating-Point Single with Update	293
DES	111110	----- --010 1	lfsue	Load Floating-Point Single with Update Extended	293
X	011111	10001 10111 /	lfsux	Load Floating-Point Single with Update Indexed	293
X	011111	10001 11111 /	lfsuxe	Load Floating-Point Single with Update Indexed Extended	293
X	011111	10000 10111 /	lfsx	Load Floating-Point Single Indexed	293
X	011111	10000 11111 /	lfsxe	Load Floating-Point Single Indexed Extended	293
D	101010	----- ----- -	lha	Load Halfword Algebraic	294
DE	111010	----- --010 0	lhae	Load Halfword Algebraic Extended	294
D	101011	----- ----- -	lhau	Load Halfword Algebraic with Update	294
DE	111010	----- --010 1	lhaue	Load Halfword Algebraic with Update Extended	294
X	011111	01011 10111 /	lhaux	Load Halfword Algebraic with Update Indexed	294
X	011111	01011 11111 /	lhauxe	Load Halfword Algebraic with Update Indexed Extended	294
X	011111	01010 10111 /	lhax	Load Halfword Algebraic Indexed	294
X	011111	01010 11111 /	lhaxe	Load Halfword Algebraic Indexed Extended	294
X	011111	11000 10110 /	lhbrx	Load Halfword Byte-Reverse Indexed	295
X	011111	11000 11110 /	lhbrxe	Load Halfword Byte-Reverse Indexed Extended	295
D	101000	----- ----- -	lhz	Load Halfword & Zero	296
DE	111010	----- --001 0	lhze	Load Halfword & Zero Extended	296
D	101001	----- ----- -	lhzu	Load Halfword & Zero with Update	296
DE	111010	----- --001 1	lhzue	Load Halfword & Zero with Update Extended	296
X	011111	01001 10111 /	lhzux	Load Halfword & Zero with Update Indexed	296
X	011111	01001 11111 /	lhzuxe	Load Halfword & Zero with Update Indexed Extended	296
X	011111	01000 10111 /	lhzx	Load Halfword & Zero Indexed	296
X	011111	01000 11111 /	lhzxe	Load Halfword & Zero Indexed Extended	296
D	101110	----- ----- -	lmw	Load Multiple Word	297
X	011111	10010 10101 /	lswi	Load String Word Immediate	298
X	011111	10000 10101 /	lswx	Load String Word Indexed	298
X	011111	00000 10100 /	lwarx	Load Word & Reserve Indexed	300
X	011111	00011 11110 /	lwarxe	Load Word & Reserve Indexed Extended	300
X	011111	10000 10110 /	lwbrx	Load Word Byte-Reverse Indexed	302
X	011111	10000 11110 /	lwbrxe	Load Word Byte-Reverse Indexed Extended	302
D	100000	----- ----- -	lwz	Load Word & Zero	303
DE	111010	----- --011 0	lwze	Load Word & Zero Extended	303

Legend:

- Don't care, usually part of an operand field
/ Reserved bit, invalid instruction form if encoded as 1
? Allocated for implementation-dependent use. See User' Manual for the implementation

Format	Opcode		Mnemonic	Instruction	Page
	Primary ($Inst_{0:5}$)	Extended ($Inst_{21:31}$)			
D	100001	----- ----- -	lwzu	Load Word & Zero with Update	303
DE	111010	----- --011 1	lwzue	Load Word & Zero with Update Extended	303
X	011111	00001 10111 /	lwzux	Load Word & Zero with Update Indexed	303
X	011111	00001 11111 /	lwzuxe	Load Word & Zero with Update Indexed Extended	303
X	011111	00000 10111 /	lwzx	Load Word & Zero Indexed	303
X	011111	00000 11111 /	lwzxe	Load Word & Zero Indexed Extended	303
X	011111	11010 10110 /	mbar	Memory Barrier	304
XL	010011	00000 00000 /	mcrf	Move Condition Register Field	305
X	111111	00010 00000 /	mcrfs	Move to Condition Register from FPSCR	306
X	011111	10000 00000 /	mcrxr	Move to Condition Register from XER	306
X	011111	10001 00000 /	mcrxr64	Move to Condition Register from XER64	306
X	011111	01000 10011 /	mfapidi	Move From APID Indirect	307
X	011111	00000 10011 /	mfcr	Move From Condition Register	307
XFX	011111	01010 00011 /	mfdcr	Move From Device Control Register	307
X	111111	10010 00111 0	mffs	Move From FPSCR	308
X	111111	10010 00111 1	mffs.	Move From FPSCR & record CR	308
X	011111	00010 10011 /	mfmsr	Move From Machine State Register	308
XFX	011111	01010 10011 /	mfspr	Move From Special Purpose Register	309
X	011111	10010 10110 /	msync	Memory Synchronize	310
XFX	011111	00100 10000 /	mtcrf	Move To Condition Register Fields	311
XFX	011111	01110 00011 /	mtdcr	Move To Device Control Register	311
X	111111	00010 00110 0	mtfsb0	Move To FPSCR Bit 0	312
X	111111	00010 00110 1	mtfsb0.	Move To FPSCR Bit 0 & record CR	312
X	111111	00001 00110 0	mtfsb1	Move To FPSCR Bit 1	312
X	111111	00001 00110 1	mtfsb1.	Move To FPSCR Bit 1 & record CR	312
XFL	111111	10110 00111 0	mtfsf	Move To FPSCR Fields	313
XFL	111111	10110 00111 1	mtfsf.	Move To FPSCR Fields & record CR	313
X	111111	00100 00110 0	mtfsfi	Move To FPSCR Field Immediate	314
X	111111	00100 00110 1	mtfsfi.	Move To FPSCR Field Immediate & record CR	314
X	011111	00100 10010 /	mtmsr	Move To Machine State Register	315
XFX	011111	01110 10011 /	mtspr	Move To Special Purpose Register	316
X	011111	/0010 01001 /	mulhd	Multiply High Doubleword	317
X	011111	/0000 01001 /	mulhdu	Multiply High Doubleword Unsigned	317
X	011111	/0010 01011 0	mulhw	Multiply High Word	318
X	011111	/0010 01011 1	mulhw.	Multiply High Word & record CR	318
X	011111	/0000 01011 0	mulhwu	Multiply High Word Unsigned	318
X	011111	/0000 01011 1	mulhwu.	Multiply High Word Unsigned & record CR	318
X	011111	00111 01001 /	mulld	Multiply Low Doubleword	319
X	011111	10111 01001 /	mulldo	Multiply Low Doubleword & record OV	319
D	000111	----- ----- -	mulli	Multiply Low Immediate	319
X	011111	00111 01011 0	mullw	Multiply Low Word	320
X	011111	00111 01011 1	mullw.	Multiply Low Word & record CR	320
X	011111	10111 01011 0	mullwo	Multiply Low Word & record OV	320
X	011111	10111 01011 1	mullwo.	Multiply Low Word & record OV & CR	320
X	011111	01110 11100 0	nand	NAND	321
X	011111	01110 11100 1	nand.	NAND & record CR	321
X	011111	00011 01000 0	neg	Negate	322

Legend:

- Don't care, usually part of an operand field
/ Reserved bit, invalid instruction form if encoded as 1
? Allocated for implementation-dependent use. See User' Manual for the implementation

Format	Opcode		Mnemonic	Instruction	Page
	Primary ($Inst_{0:5}$)	Extended ($Inst_{21:31}$)			
X	011111	00011 01000 1	neg.	Negate & record CR	322
X	011111	10011 01000 0	nego	Negate & record OV	322
X	011111	10011 01000 1	nego.	Negate & record OV & record CR	322
X	011111	00011 11100 0	nor	NOR	323
X	011111	00011 11100 1	nor.	NOR & record CR	323
X	011111	01101 11100 0	or	OR	324
X	011111	01101 11100 1	or.	OR & record CR	324
X	011111	01100 11100 0	orc	OR with Complement	324
X	011111	01100 11100 1	orc.	OR with Complement & record CR	324
D	011000	----- ----- -	ori	OR Immediate	324
D	011001	----- ----- -	oris	OR Immediate Shifted	324
XL	010011	00001 10011 /	rfci	Return From Critical Interrupt	325
XL	010011	00001 10010 /	rfi	Return From Interrupt	326
MDS	011110	----- -1000 /	rldcl	Rotate Left Doubleword then Clear Left	327
MDS	011110	----- -1001 /	rldcr	Rotate Left Doubleword then Clear Right	328
MD	011110	----- -010- /	rldic	Rotate Left Doubleword Immediate then Clear	329
MD	011110	----- -000- /	rldicl	Rotate Left Doubleword Immediate then Clear Left	327
MD	011110	----- -001- /	rldicr	Rotate Left Doubleword Immediate then Clear Right	328
MD	011110	----- -011- /	rldimi	Rotate Left Doubleword Immediate then Mask Insert	330
M	010100	----- ----- 0	rlwimi	Rotate Left Word Immed then Mask Insert	331
M	010100	----- ----- 1	rlwimi.	Rotate Left Word Immed then Mask Insert & record CR	331
M	010101	----- ----- 0	rlwinm	Rotate Left Word Immed then AND with Mask	332
M	010101	----- ----- 1	rlwinm.	Rotate Left Word Immed then AND with Mask & record CR	332
M	010111	----- ----- 0	rlwnm	Rotate Left Word then AND with Mask	332
M	010111	----- ----- 1	rlwnm.	Rotate Left Word then AND with Mask & record CR	332
SC	010001	///// ////1 /	sc	System Call	334
X	011111	00000 11011 /	sld	Shift Left Doubleword	335
X	011111	00000 11000 0	slw	Shift Left Word	336
X	011111	00000 11000 1	slw.	Shift Left Word & record CR	336
X	011111	11000 11010 /	srad	Shift Right Algebraic Doubleword	337
XS	011111	11001 1101- /	sradi	Shift Right Algebraic Doubleword Immediate	337
X	011111	11000 11000 0	sraw	Shift Right Algebraic Word	338
X	011111	11000 11000 1	sraw.	Shift Right Algebraic Word & record CR	338
X	011111	11001 11000 0	srawi	Shift Right Algebraic Word Immediate	338
X	011111	11001 11000 1	srawi.	Shift Right Algebraic Word Immediate & record CR	338
X	011111	10000 11011 /	srd	Shift Right Doubleword	339
X	011111	10000 11000 0	srw	Shift Right Word	340
X	011111	10000 11000 1	srw.	Shift Right Word & record CR	340
D	100110	----- ----- -	stb	Store Byte	341
DE	111010	----- --100 0	stbe	Store Byte Extended	341
DE	111010	----- --100 1	stbue	Store Byte with Update Extended	341
D	100111	----- ----- -	stbu	Store Byte with Update	341
X	011111	00111 10111 /	stbux	Store Byte with Update Indexed	341
X	011111	00111 11111 /	stbuxe	Store Byte with Update Indexed Extended	341
X	011111	00110 10111 /	stbx	Store Byte Indexed	341
X	011111	00110 11111 /	stbxe	Store Byte Indexed Extended	341
X	011111	01111 11111 1	stdcxe.	Store Doubleword Conditional Indexed Extended	342

Legend:

- `-` Don't care, usually part of an operand field
- `/` Reserved bit, invalid instruction form if encoded as 1
- `?` Allocated for implementation-dependent use. See User' Manual for the implementation

Format	Opcode		Mnemonic	Instruction	Page
	Primary ($Inst_{0:5}$)	Extended ($Inst_{21:31}$)			
DES	111110	----- --100 0	stde	Store Doubleword Extended	343
DES	111110	----- --100 1	stdue	Store Doubleword with Update Extended	343
X	011111	11101 11111 /	stduxe	Store Doubleword with Update Indexed Extended	343
X	011111	11100 11111 /	stdxe	Store Doubleword Indexed Extended	343
D	110110	----- ----- -	stfd	Store Floating-Point Double	344
DES	111110	----- --111 0	stfde	Store Floating-Point Double Extended	344
D	110111	----- ----- -	stfdu	Store Floating-Point Double with Update	344
DES	111110	----- --111 1	stfdue	Store Floating-Point Double with Update Extended	344
X	011111	10111 10111 /	stfdux	Store Floating-Point Double with Update Indexed	344
X	011111	10111 11111 /	stfduxe	Store Floating-Point Double with Update Indexed Extended	344
X	011111	10110 10111 /	stfdx	Store Floating-Point Double Indexed	344
X	011111	10110 11111 /	stfdxe	Store Floating-Point Double Indexed Extended	344
X	011111	11110 10111 /	stfiwx	Store Floating-Point as Int Word Indexed	345
X	011111	11110 11111 /	stfiwxe	Store Floating-Point as Int Word Indexed Extended	345
D	110100	----- ----- -	stfs	Store Floating-Point Single	346
DES	111110	----- --110 0	stfse	Store Floating-Point Single Extended	346
D	110101	----- ----- -	stfsu	Store Floating-Point Single with Update	346
DES	111110	----- --110 1	stfsue	Store Floating-Point Single with Update Extended	346
X	011111	10101 10111 /	stfsux	Store Floating-Point Single with Update Indexed	346
X	011111	10101 11111 /	stfsuxe	Store Floating-Point Single with Update Indexed Extended	346
X	011111	10100 10111 /	stfsx	Store Floating-Point Single Indexed	346
X	011111	10100 11111 /	stfsxe	Store Floating-Point Single Indexed Extended	346
D	101100	----- ----- -	sth	Store Halfword	347
X	011111	11100 10110 /	sthbrx	Store Halfword Byte-Reverse Indexed	348
X	011111	11100 11110 /	sthbrxe	Store Halfword Byte-Reverse Indexed Extended	348
DE	111010	----- --101 0	sthe	Store Halfword Extended	347
D	101101	----- ----- -	sthu	Store Halfword with Update	347
DE	111010	----- --101 1	sthue	Store Halfword with Update Extended	347
X	011111	01101 10111 /	sthux	Store Halfword with Update Indexed	347
X	011111	01101 11111 /	sthuxe	Store Halfword with Update Indexed Extended	347
X	011111	01100 10111 /	sthx	Store Halfword Indexed	347
X	011111	01100 11111 /	sthxe	Store Halfword Indexed Extended	347
D	101111	----- ----- -	stmw	Store Multiple Word	349
X	011111	10110 10101 /	stswi	Store String Word Immediate	350
X	011111	10100 10101 /	stswx	Store String Word Indexed	350
D	100100	----- ----- -	stw	Store Word	351
X	011111	10100 10110 /	stwbrx	Store Word Byte-Reverse Indexed	352
X	011111	10100 11110 /	stwbrxe	Store Word Byte-Reverse Indexed Extended	352
X	011111	00100 10110 1	stwcx.	Store Word Conditional Indexed & record CR	353
X	011111	00100 11110 1	stwcxe.	Store Word Conditional Indexed Extended & record CR	353
DE	111010	----- --111 0	stwe	Store Word Extended	351
D	100101	----- ----- -	stwu	Store Word with Update	351
DE	111010	----- --111 1	stwue	Store Word with Update Extended	351
X	011111	00101 10111 /	stwux	Store Word with Update Indexed	351
X	011111	00101 11111 /	stwuxe	Store Word with Update Indexed Extended	351
X	011111	00100 10111 /	stwx	Store Word Indexed	351
X	011111	00100 11111 /	stwxe	Store Word Indexed Extended	351

Legend:

- \- Don't care, usually part of an operand field
- / Reserved bit, invalid instruction form if encoded as 1
- ? Allocated for implementation-dependent use. See User' Manual for the implementation

Format	Opcode		Mnemonic	Instruction	Page
	Primary ($Inst_{0:5}$)	Extended ($Inst_{21:31}$)			
X	011111	00001 01000 0	subf	Subtract From	355
X	011111	00001 01000 1	subf.	Subtract From & record CR	355
X	011111	00000 01000 0	subfc	Subtract From Carrying	356
X	011111	00000 01000 1	subfc.	Subtract From Carrying & record CR	356
X	011111	10000 01000 0	subfco	Subtract From Carrying & record OV	356
X	011111	10000 01000 1	subfco.	Subtract From Carrying & record OV & CR	356
X	011111	00100 01000 0	subfe	Subtract From Extended with CA	357
X	011111	00100 01000 1	subfe.	Subtract From Extended with CA & record CR	357
X	011111	01100 01000 /	subfe64	Subtract From Extended with CA64	357
X	011111	11100 01000 /	subfe64o	Subtract From Extended with CA64 & record OV	357
X	011111	10100 01000 0	subfeo	Subtract From Extended with CA & record OV	357
X	011111	10100 01000 1	subfeo.	Subtract From Extended with CA & record OV & CR	357
D	001000	----- ----- -	subfic	Subtract From Immediate Carrying	358
X	011111	00111 01000 0	subfme	Subtract From Minus One Extended with CA	359
X	011111	00111 01000 1	subfme.	Subtract From Minus One Extended with CA & record CR	359
X	011111	01111 01000 /	subfme64	Subtract From Minus One Extended with CA64	359
X	011111	11111 01000 /	subfme64o	Subtract From Minus One Extended with CA64 & record OV	359
X	011111	10111 01000 0	subfmeo	Subtract From Minus One Extended with CA & record OV	359
X	011111	10111 01000 1	subfmeo.	Subtract From Minus One Extended with CA & record OV & CR	359
X	011111	10001 01000 0	subfo	Subtract From & record OV	355
X	011111	10001 01000 1	subfo.	Subtract From & record OV & CR	355
X	011111	00110 01000 0	subfze	Subtract From Zero Extended with CA	360
X	011111	00110 01000 1	subfze.	Subtract From Zero Extended with CA & record CR	360
X	011111	01110 01000 /	subfze64	Subtract From Zero Extended with CA64	360
X	011111	11110 01000 /	subfze64o	Subtract From Zero Extended with CA64 & record OV	360
X	011111	10110 01000 0	subfzeo	Subtract From Zero Extended with CA & record OV	360
X	011111	10110 01000 1	subfzeo.	Subtract From Zero Extended with CA & record OV & CR	360
X	011111	00010 00100 /	td	Trap Doubleword	361
D	000010	----- ----- -	tdi	Trap Doubleword Immediate	361
X	011111	11000 10010 /	tlbivax	TLB Invalidate Virtual Address Indexed	362
X	011111	11000 10011 /	tlbivaxe	TLB Invalidate Virtual Address Indexed Extended	362
X	011111	11101 10010 /	tlbre	TLB Read Entry	363
X	011111	11100 10010 ?	tlbsx	TLB Search Indexed	364
X	011111	11100 10011 ?	tlbsxe	TLB Search Indexed Extended	364
X	011111	10001 10110 /	tlbsync	TLB Synchronize	365
X	011111	11110 10010 /	tlbwe	TLB Write Entry	366
X	011111	00000 00100 /	tw	Trap Word	367
D	000011	----- ----- -	twi	Trap Word Immediate	367
X	011111	00100 00011 /	wrtee	Write External Enable	368
X	011111	00101 00011 /	wrteei	Write External Enable Immediate	368
X	011111	01001 11100 0	xor	XOR	369
X	011111	01001 11100 1	xor.	XOR & record CR	369
D	011010	----- ----- -	xori	XOR Immediate	369
D	011011	----- ----- -	xoris	XOR Immediate Shifted	369

Legend:

- Don't care, usually part of an operand field
/ Reserved bit, invalid instruction form if encoded as 1
? Allocated for implementation-dependent use. See User' Manual for the implementation

Last Page of Document