PowerPC® Microprocessor Family: The Programming Environments Manual for 64-bit Microprocessors

Version 3.0

July 15, 2005

IBM ®

IBM Systems and Technology Group
2070 Route 52, Bldg. 330
Hopewell Junction, NY 12533-6351

The IBM home page can be found at http://www.ibm.com

The IBM Microelectronics home page can be found at http://www.ibm.com/chips

pem_64bit_title.fm(3.0)
July 15, 2005

Contents

List of Tables

List of Figures

THIS PAGE INTENTIONALLY LEFT BLANK

About This Book

The primary objective of this manual is to help programmers provide software that is compatible across the family of PowerPC™ processors. Because the PowerPC Architecture is designed to be flexible to support a broad range of processors, this book provides a general description of features that are common to PowerPC processors and indicates those features that are optional or that may be implemented differently in the design of each processor.

This book describes the PowerPC Architecture from the perspective of the 64-bit architecture. For information that pertains only to the 32-bit architecture refer to the *PowerPC Microprocessor Family: The Programming Environments for 32-Bit Microprocessors*. To locate any published errata or updates for this manual, refer to the world-wide web at http://www.ibm.com/powerpc. For programmers working with a specific processor, this book should be used in conjunction with the user's manual for that processor.

This manual distinguishes between the three levels, or programming environments, of the PowerPC Architecture, which are as follows:

- PowerPC user instruction set architecture (UISA)—The UISA defines the level of the architecture to which user-level software should conform.
- PowerPC virtual environment architecture (VEA)—The VEA, which is the smallest component of the PowerPC Architecture, defines additional user-level functionality that falls outside typical user-level software requirements.

  Implementations that conform to the PowerPC VEA also adhere to the UISA, but may not necessarily adhere to the OEA.
- PowerPC operating environment architecture (OEA)—The OEA defines supervisor-level resources typically required by an operating system.

  Implementations that conform to the PowerPC OEA also conform to the PowerPC UISA and VEA.

Refer to *Section 1.1.2* on page 32 for additional information on the PowerPC Architecture levels.

Temporary 64-Bit Bridge

The OEA defines optional features to simplify the migration of 32-bit operating systems to a 64-bit implementations.

It is important to note that some resources are defined more generally at one level in the architecture and more specifically at another. For example, conditions that can cause a floating-point exception are defined by the UISA, while the exception mechanism itself is defined by the OEA.

This book does not attempt to replace the PowerPC Architecture specification (version 2.01), which defines the architecture from the perspective of the three programming environments and which remains the defining manual for the PowerPC Architecture.

For ease in reference, this book and the processor user's manuals have arranged the architecture information into topics that build upon one another, beginning with a description and complete summary of registers and instructions (for all three environments) and progressing to more specialized topics such as the cache, exception, and memory management models. As such, chapters may include information from multiple levels of the architecture; for example, the discussion of the cache model uses information from both the VEA and the OEA.

It is beyond the scope of this manual to describe individual PowerPC processors. It must be kept in mind that each PowerPC processor may be unique in its implementation of the PowerPC Architecture.

The information in this book is subject to change without notice, as described in the disclaimers on the title page of this book. As with any technical documentation, it is the readers' responsibility to be sure they are using the most recent version of the documentation. For more information contact your sales representative or visit our web site at: http://www.ibm.com/powerpc.

Audience

This manual is intended for system software and hardware developers and application programmers who want to develop 64-bit products using IBM's 64-bit PowerPC processors. It is assumed that the reader understands operating systems, microprocessor system design, and the basic principles of RISC processing.

Organization

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

- *Chapter 1, "Overview,"* is useful for those who want a general understanding of the features and functions of the PowerPC Architecture. This chapter describes the flexible nature of the PowerPC Architecture definition and provides an overview of how the PowerPC Architecture defines the register set, operand conventions, addressing modes, instruction set, cache model, exception model, and memory management model.
- *Chapter 2, "PowerPC Register Set,"* is useful for software engineers who need to understand the PowerPC programming model for the three programming environments and the functionality of the PowerPC registers.
- *Chapter 3, "Operand Conventions,"* describes PowerPC conventions for storing data in memory, including information regarding alignment, single and double-precision floating-point conventions, and big and little-endian byte ordering.
- *Chapter 4, "Addressing Modes and Instruction Set Summary,"* provides an overview of the PowerPC addressing modes and a description of the PowerPC instructions. Instructions are organized by function.
- *Chapter 5, "Cache Model and Memory Coherency,"* provides a discussion of the cache and memory model defined by the VEA and aspects of the cache model that are defined by the OEA.
- *Chapter 6, "Exceptions,"* describes the exception model defined in the OEA.
- *Chapter 7, "Memory Management,"* provides descriptions of the PowerPC address translation and memory protection mechanism as defined by the OEA.
- *Chapter 8, "Instruction Set,"* functions as a handbook for the PowerPC instruction set. Instructions are sorted by mnemonic. Each instruction description includes the instruction formats and an individualized legend that provides such information as the level(s) of the PowerPC Architecture in which the instruction may be found and the privilege level of the instruction.
- *Appendix A, "PowerPC Instruction Set Listings,"* lists all the PowerPC instructions. Instructions are grouped according to mnemonic, opcode, function, and form.
- *Appendix B, "Multiple-Precision Shifts,"* describes how multiple-precision shift operations can be programmed as defined by the UISA.

- *Appendix C, "Floating-Point Models,"* gives examples of how the floating-point conversion instructions can be used to perform various conversions as described in the UISA.
- *Appendix D, "Synchronization Programming Examples,"* gives examples showing how synchronization instructions can be used to emulate various synchronization primitives and how to provide more complex forms of synchronization.
- *Appendix E, "Simplified Mnemonics,"* provides a set of simplified mnemonic examples and symbols.
- This manual also includes a glossary.

Suggested Reading

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

General Information

The following documentation provides useful information about the PowerPC Architecture and computer architecture in general:

- The following books are available via many online bookstores.
  - *The PowerPC Architecture: A Specification for a New Family of RISC Processors*, Second Edition, by International Business Machines, Inc.1994.
    Note: This book has been superseded with the PowerPC Architecture Books I-III, Version 2.01 and is available at www.ibm.com/powerpc.
  - *PowerPC Microprocessor Common Hardware Reference Platform: A System Architecture*, by Apple Computer, Inc., International Business Machines, Inc., and Motorola, Inc.
  - *Macintosh Technology in the Common Hardware Reference Platform*, by Apple Computer, Inc.
  - *Computer Architecture: A Quantitative Approach*, Second Edition, by John L. Hennessy and David A. Patterson,
- Inside Macintosh: PowerPC System Software, Addison-Wesley Publishing Company, One Jacob Way, Reading, MA, 01867.
- PowerPC Programming for Intel Programmers, by Kip McClanahan; IDG Books Worldwide, Inc., 919 East Hillsdale Boulevard, Suite 400, Foster City, CA, 94404.

PowerPC Documentation

The PowerPC documentation is organized in the following types of documents:

- User's manuals—These books provide details about individual PowerPC implementations and are intended to be used in conjunction with *The Programming Environments Manual. Chapter 1, Overview* is equivalent to previously released Technical Summaries.
- Addenda/errata to user's manuals—Because some processors have follow-on parts, an addendum may be provided that describes the additional features and changes to functionality of the follow-on part. These addenda are intended for use with the corresponding user's manuals.
- Programming environments manuals (PEM)—These books provide information about resources defined by the PowerPC Architecture that are common to PowerPC processors. There are several PEM versions available, this version of the PEM which describes the 64-bit PowerPC Architecture; the *PowerPC Microprocessor Family: The Programming Environments for 32-Bit Microprocessors* that describes only the 32-bit model; and the *PowerPC Microprocessor Family: AltiVec^{TM*} Technology Programming Environments Manual* which describes the vector/SIMD architecture.
- Datasheets—Datasheets provide specific data regarding bus timing; signal behavior; and AC, DC, and thermal characteristics, as well as other design considerations for each PowerPC implementation.
- *PowerPC Microprocessor Family: The Programmer's Reference Guide*: *MPRPPCPRG-01* is a concise reference that includes the register summary, memory control model, exception vectors, and the PowerPC instruction set.
- PowerPC Quick Reference Guide: This brochure is a Quick Reference Guide to IBM's portfolio of industry-leading PowerPC technology. It includes highlights and specifications for the PowerPC 405, PowerPC 440, PowerPC 750, and PowerPC 970 based standard products.
- Book I: PowerPC User Instruction Set Architecture (Version 2.01)–This book defines the instructions, registers, etc., typically used by application programs (for example, Branch, Load, Store, and Arithmetic instructions; general purpose and floating-point registers). All Book I facilities and instructions are non-privileged (are available in problem state).
- Book II: PowerPC Virtual Environment Architecture (Version 2.01)–This book defines the storage model (caches, storage access ordering, etc.) and related instructions, such as the instructions used to manage caches and to synchronize storage accesses when storage is shared among programs running on different processors. All Book II facilities and instructions are non-privileged, but they are typically used via operating-system-provided library subroutines, which application programs call as needed.
- Book III: PowerPC Operating Environment Architecture (Version 2.01) –This book defines the privileged facilities and related instructions (address translation, storage protection, interruptions, etc.). Nearly all Book III facilities and instructions are privileged. (Those that are non-privileged are described also in Book I or II, but only at the level needed by application programmers.)
- Application notes—These short documents contain useful information about specific design issues useful to programmers and engineers working with PowerPC processors.
- Documentation for support chips.

For a current list of PowerPC documentation, refer to the world-wide web at http://wwwibm.com/chips. Additional literature on PowerPC implementations is being released to the web as new processors become available.

Conventions

This manual uses the following notational conventions:

mnemonics	Instruction mnemonics are shown in lowercase bold.
italics	Italics indicate variable command parameters, for example, **bcctr***x*. Book titles in text are set in italics.
0x0	Prefix to denote hexadecimal number
0b0	Prefix to denote binary number
rA, **r**B	Instruction syntax used to identify a source GPR
rD	Instruction syntax used to identify a destination GPR
frA, **fr**B, **fr**C	Instruction syntax used to identify a source FPR
frD	Instruction syntax used to identify a destination FPR
n	Used to express an undefined numerical value
REG[FIELD]	Abbreviations or acronyms for registers are shown in uppercase text. Specific bits, fields, or ranges appear in brackets. For example, MSR[LE] refers to the little-endian mode enable bit in the machine state register.
x	In certain contexts, such as a signal encoding, this indicates a don't care.
¬	NOT logical operator
&	AND logical operator
|	OR logical operator
$\boxed{0\,0\,0\,0}$	Indicates reserved bits or bit fields in a register. Although these bits may be written to as either ones or zeroes, they are always read as zeros.

Temporary 64-Bit Bridge

Text that pertains to the optional 64-bit bridge defined by the OEA is presented with a box, as shown here.

Additional conventions used with instruction encodings are described in *Table 8-2* on page 300. Conventions used for pseudocode examples are described in *Table 8-3* on page 302.

Acronyms and Abbreviations

Table i contains acronyms and abbreviations that are used in this manual. Note that the meanings for some acronyms (such as SDR1 and XER) are historical, and the words for which an acronym stands may not be intuitively obvious.

Table i. Acronyms and Abbreviated Terms

Term	Meaning
ALU	Arithmetic logic unit
ASR	Address space register
BIST	Built-in self test
BPU	Branch processing unit
BUID	Bus unit ID
CR	Condition register
CTR	Count register
DABR	Data address breakpoint register
DAR	Data address register
DEC	Decrementer register
DSISR	Register used for determining the source of a DSI exception
DTLB	Data translation lookaside buffer
EA	Effective address
EAR	External access register
ECC	Error checking and correction
FIFO	First-in-first-out
FPECR	Floating-point exception cause register
FPR	Floating-point register
FPSCR	Floating-point status and control register
FPU	Floating-point unit
GPR	General-purpose register
IEEE®	Institute of Electrical and Electronics Engineers
ITLB	Instruction translation lookaside buffer
IU	Integer unit
L2	Secondary cache
LIFO	Last-in-first-out
LR	Link register
LRU	Least recently used
LSB	Least-significant byte
lsb	Least-significant bit
lsq	Least-significant quad word
MERSI	Modified/exclusive/reserved/shared/invalid–cache coherency protocol

Table i. Acronyms and Abbreviated Terms (Continued)

Term	Meaning
MESI	Modified/exclusive/shared/invalid—cache coherency protocol
MMU	Memory management unit
MSB	Most-significant byte
msb	Most-significant bit
msq	Most-significant quad word
MSR	Machine state register
NaN	Not a number
NIA	Next instruction address
No-op	No operation
OEA	Operating environment architecture
PIR	Processor identification register
PTE	Page table entry
PTEG	Page table entry group
PVR	Processor version register
RISC	Reduced instruction set computing
RTL	Register transfer language
RWITM	Read with intent to modify
SDR1	Register that specifies the page table base address for virtual-to-physical address translation
SIMD	Single instruction stream, multiple data streams
SIMM	Signed immediate value
SLB	Segment lookaside buffer
SPR	Special-purpose register
SPRG*n*	Registers available for general purposes
SR	Segment register
SRR0	Machine status save/restore register 0
SRR1	Machine status save/restore register 1
STE	Segment table entry
TB	Time base register
TLB	Translation lookaside buffer
UIMM	Unsigned immediate value
UISA	User instruction set architecture
VA	Virtual address
VEA	Virtual environment architecture
WAR	Write-after-read
WAW	Write-after-write
WIMG	Write-through/caching-inhibited/memory-coherency enforced/guarded – memory attribute bits
XER	Register used primarily for indicating conditions such as carries and overflows for integer operations

Terminology Conventions

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

Table ii. Terminology Conventions

The Architecture Specification	This Manual
Data storage interrupt (DSI)	DSI exception
Extended mnemonics	Simplified mnemonics
Instruction storage interrupt (ISI)	ISI exception
Interrupt	Exception
Privileged mode (or privileged state)	Supervisor-level privilege
Problem mode (or problem state)	User-level privilege
Real address	Physical address
Relocation	Translation
Storage (locations)	Memory
Storage (the act of)	Access
Swizzling	Doubleword swap

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

Table iii. Instruction Field Conventions

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

1. Overview

The PowerPC Architecture provides a software model that ensures software compatibility among implementations of the PowerPC family of microprocessors. In this manual, and in other PowerPC documentation as well, the term ‘implementation’ refers to a hardware device (typically a microprocessor) that complies with the specifications defined by the architecture.

general defines the following:

- Instruction set—The instruction set specifies the families of instructions (such as load/store, integer arithmetic, and floating-point arithmetic instructions), the specific instructions, and the forms used for encoding the instructions. The instruction set definition also specifies the addressing modes used for accessing memory.
- Programming model—The programming model defines the register set and the memory conventions, including details regarding the bit and byte ordering, and the conventions for how data (such as integer and floating-point values) are stored.
- Memory model—The memory model defines the size of the address space and of the subdivisions of that address space. It also defines the ability to configure pages of memory with respect to caching, byte ordering (big or little-endian), coherency, and various types of memory protection.
- Exception model—The exception model defines the common set of exceptions and the conditions that can generate those exceptions. The exception model specifies characteristics of the exceptions, such as whether they are precise or imprecise, synchronous or asynchronous, and maskable or nonmaskable. The exception model defines the exception vectors and a set of registers used when exceptions are taken. The exception model also provides memory space for implementation-specific exceptions. (Note that exceptions are referred to as interrupts in the architecture specification.)
- Memory management model—The memory management model defines how memory is partitioned, configured, and protected. The memory management model also specifies how memory translation is performed, the real, virtual, and physical address spaces, special memory control instructions, and other characteristics. (Physical address is referred to as real address in the architecture specification.)
- Time-keeping model—The time-keeping model defines facilities that permit the time of day to be determined and the resources and mechanisms required for supporting time-related exceptions.

These aspects of the PowerPC Architecture are defined at different levels of the architecture, and this chapter provides an overview of those levels—the user instruction set architecture (UISA), the virtual environment architecture (VEA), and the operating environment architecture (OEA).

1.1 PowerPC Architecture Overview

The PowerPC Architecture takes advantage of recent technological advances in such areas as process technology, compiler design, and reduced instruction set computing (RISC) microprocessor design. It provides software compatibility across a diverse family of implementations, primarily single-chip microprocessors, intended for a wide range of systems, including battery-powered personal computers; embedded controllers; high-end scientific and graphics workstations; and multiprocessing, microprocessor-based mainframes. To provide a single architecture for such a broad assortment of processor environments, the PowerPC Architecture is both flexible and scalable.

The flexibility of the PowerPC Architecture offers many price/performance options. Designers can choose whether to implement architecturally-defined features in hardware or in software. For example, a processor designed for a high-end workstation has a greater need for the performance gained from implementing floating-point normalization and denormalization in hardware than a battery-powered, general-purpose computer might.

The PowerPC Architecture is scalable to take advantage of continuing technological advances—for example, the continued miniaturization of transistors makes it more feasible to implement more execution units and a richer set of optimizing features without being constrained by the architecture.

The PowerPC Architecture defines the following features:

- Separate 32-entry register files for integer and floating-point instructions. The general-purpose registers (GPRs) hold source data for integer arithmetic instructions, and the floating-point registers (FPRs) hold source and target data for floating-point arithmetic instructions.
- Instructions for loading and storing data between the memory system and either the FPRs or GPRs.
- Uniform-length instructions to allow simplified instruction pipelining and parallel processing instruction dispatch mechanisms.
- Nondestructive use of registers for arithmetic instructions in which the second, third, and sometimes the fourth operand, typically specify source registers for calculations whose results are typically stored in the target register specified by the first operand.
- A precise exception model (with the option of treating floating-point exceptions imprecisely).
- Floating-point support that includes IEEE-754 floating-point operations.
- A flexible architecture definition that allows certain features to be performed in either hardware or with assistance from implementation-specific software depending on the needs of the processor design.
- The ability to perform both single and double-precision floating-point operations.
- User-level instructions for explicitly storing, flushing, and invalidating data in the on-chip caches. The architecture also defines special instructions (cache block touch instructions) for speculatively loading data before it is needed, reducing the effect of memory latency.
- Definition of a memory model that allows weakly-ordered memory accesses. This allows bus operations to be reordered dynamically, which improves overall performance and in particular reduces the effect of memory latency on instruction throughput.
- Support for separate instruction and data caches (Harvard architecture) and for unified caches.
- Support for both big and little-endian addressing modes.
- Support for 64-bit addressing.

This chapter provides an overview of the major characteristics of the PowerPC Architecture in the order in which they are addressed in this book:

- Register set and programming model
- Instruction set and addressing modes
- Cache implementations
- Exception model
- Memory management

1.1.1 64-Bit PowerPC Architecture and the 32-Bit Subset

The PowerPC Architecture is a 64-bit architecture with a 32-bit subset. It is important to distinguish the following modes of operations:

- 64-bit implementations/64-bit mode—The PowerPC Architecture provides 64-bit addressing, 64-bit integer data types, and instructions that perform arithmetic operations on those data types, as well as other features to support the wider addressing range. The processor is configured to operate in 64-bit mode by setting the MSR[SF] bit.
- 64-bit implementations/32-bit mode—For compatibility with 32-bit implementations, 64-bit implementations can be configured to operate in 32-bit mode by clearing the MSR[SF] bit. In 32-bit mode, the effective address is treated as a 32-bit address, condition bits, such as overflow and carry bits, are set based on 32-bit arithmetic (for example, integer overflow occurs when the result exceeds 32 bits), and the count register (CTR) is tested by branch conditional instructions following conventions for 32-bit implementations. All applications written for 32-bit implementations will run without modification on 64-bit processors running in 32-bit mode.

1.1.1.1 Temporary 64-Bit Bridge

The OEA defines an additional, optional bridge that may make it easier to migrate a 32-bit operating system to the 64-bit architecture. This bridge allows 64-bit implementations to use a simpler memory management model to access 32-bit effective address space. Processors that implement this bridge may implement resources, such as instructions, that are not supported, and in some cases not permitted by the 64-bit architecture.

For processors that implement the address translation portion of the bridge, segment descriptors take the form of the STEs defined for 64-bit MMUs; however, only 16 STEs are required to define the entire 4-Gbyte address space. Like 32-bit implementations, the effective address space is entirely defined by 16 contiguous 256-Mbyte segment descriptors. Rather than using the set of 16, 32-bit segment registers as is defined for the 32-bit MMU, the 16 STEs are implemented and are maintained in 16 SLB entries.

These resources are described more fully in *Section 7.6 Migration of Operating Systems from 32-Bit Implementations to 64-Bit Implementations*. These resources are not to be considered a permanent part of the PowerPC Architecture.

1.1.2 Levels of the PowerPC Architecture

The PowerPC Architecture is defined in three levels that correspond to three programming environments, roughly described from the most general, user-level instruction set environment, to the more specific, operating environment. This layering of the architecture provides flexibility, allowing degrees of software compatibility across a wide range of implementations. For example, an implementation such as an embedded controller will support the user instruction set, whereas it may be impractical for it to adhere to the memory management, exception, and cache models.

The three levels of the PowerPC Architecture are defined as follows:

- PowerPC user instruction set architecture (UISA)—The UISA defines the level of the architecture to which user-level (referred to as problem state in the architecture specification) software should conform. The UISA defines the base user-level instruction set, user-level registers, data types, floating-point memory conventions and exception model as seen by user programs, and the memory and programming models. The icon shown in the margin identifies text that is relevant with respect to the UISA.
- PowerPC virtual environment architecture (VEA)—The VEA defines additional user-level functionality that falls outside typical user-level software requirements. The VEA describes the memory model for an environment in which multiple devices can access memory, defines aspects of the cache model, defines cache control instructions, and defines the time base facility from a user-level perspective. The icon shown in the margin identifies text that is relevant with respect to the VEA.
- PowerPC operating environment architecture (OEA)—The OEA defines supervisor-level (referred to as privileged state in the architecture specification) resources typically required by an operating system. The OEA defines the PowerPC memory management model, supervisor-level registers, synchronization requirements, and the exception model. The OEA also defines the time base feature from a supervisor-level perspective. The icon shown in the margin identifies text that is relevant with respect to the OEA.

Implementations that adhere to the VEA level are guaranteed to adhere to the UISA level, but may not necessarily adhere to the OEA level; likewise, implementations that conform to the OEA level are also guaranteed to conform to the UISA and the VEA levels.

All PowerPC devices adhere to the UISA, offering compatibility among all PowerPC application programs. However, there may be different versions of the VEA and OEA than those described here. For example, some devices, such as embedded controllers, may not require some of the features as defined by this VEA and OEA, and may implement a simpler or modified version of those features.

The general-purpose PowerPC microprocessors comply both with the UISA and with the VEA and OEA discussed here. In this book, these three levels of the architecture are referred to collectively as the PowerPC Architecture. The distinctions between the levels of the PowerPC Architecture are maintained clearly throughout this manual, using the conventions described in the *Section Conventions* on page 25.

1.1.3 Latitude Within the Levels of the PowerPC Architecture

The PowerPC Architecture defines those parameters necessary to ensure compatibility among PowerPC processors, but also allows a wide range of options for individual implementations. These are as follows:

- The PowerPC Architecture defines some facilities (such as registers, bits within registers, instructions, and exceptions) as optional.
- The PowerPC Architecture allows implementations to define additional privileged special-purpose registers (SPRs), exceptions, and instructions for special system requirements (such as power management in processors designed for very low-power operation).

- There are many other parameters that the PowerPC Architecture allows implementations to define. For example, the PowerPC Architecture may define conditions for which an exception may be taken, such as alignment conditions. A particular implementation may choose to solve the alignment problem without taking the exception.
- Processors may implement any architectural facility or instruction with assistance from software (that is, they may trap and emulate) as long as the results (aside from performance) are identical to that specified by the architecture.
- Some parameters are defined at one level of the architecture and defined more specifically at another. For example, the UISA defines conditions that may cause an alignment exception, and the OEA specifies the exception itself.

1.1.4 Features Not Defined by the PowerPC Architecture

Because flexibility is an important design goal of the PowerPC Architecture, there are many aspects of the processor design, typically relating to the hardware implementation, that the PowerPC Architecture does not define, such as the following:

- System bus interface signals—Although numerous implementations may have similar interfaces, the PowerPC Architecture does not define individual signals or the bus protocol. For example, the OEA allows each implementation to determine the signal or signals that trigger the machine check exception.
- Cache design—The PowerPC Architecture does not define the size, structure, the replacement algorithm, or the mechanism used for maintaining cache coherency. The PowerPC Architecture supports, but does not require, the use of separate instruction and data caches. Likewise, the PowerPC Architecture does not specify the method by which cache coherency is ensured.
- The number and the nature of execution units—The PowerPC Architecture is a reduced instruction set computing (RISC) architecture, and as such has been designed to facilitate the design of processors that use pipelining and parallel execution units to maximize instruction throughput. However, the PowerPC Architecture does not define the internal hardware details of implementations. For example, one processor may execute load and store operations in the integer unit, while another may execute these instructions in a dedicated load/store unit.
- Other internal microarchitecture issues—The PowerPC Architecture does not prescribe which execution unit is responsible for executing a particular instruction; it also does not define details regarding the instruction fetching mechanism, how instructions are decoded and dispatched, and how results are written back. Dispatch and write-back may occur in-order or out-of-order. Also while the architecture specifies certain registers, such as the GPRs and FPRs, implementations can implement register renaming or other schemes to reduce the impact of data dependencies and register contention.

1.2 The PowerPC Architectural Models

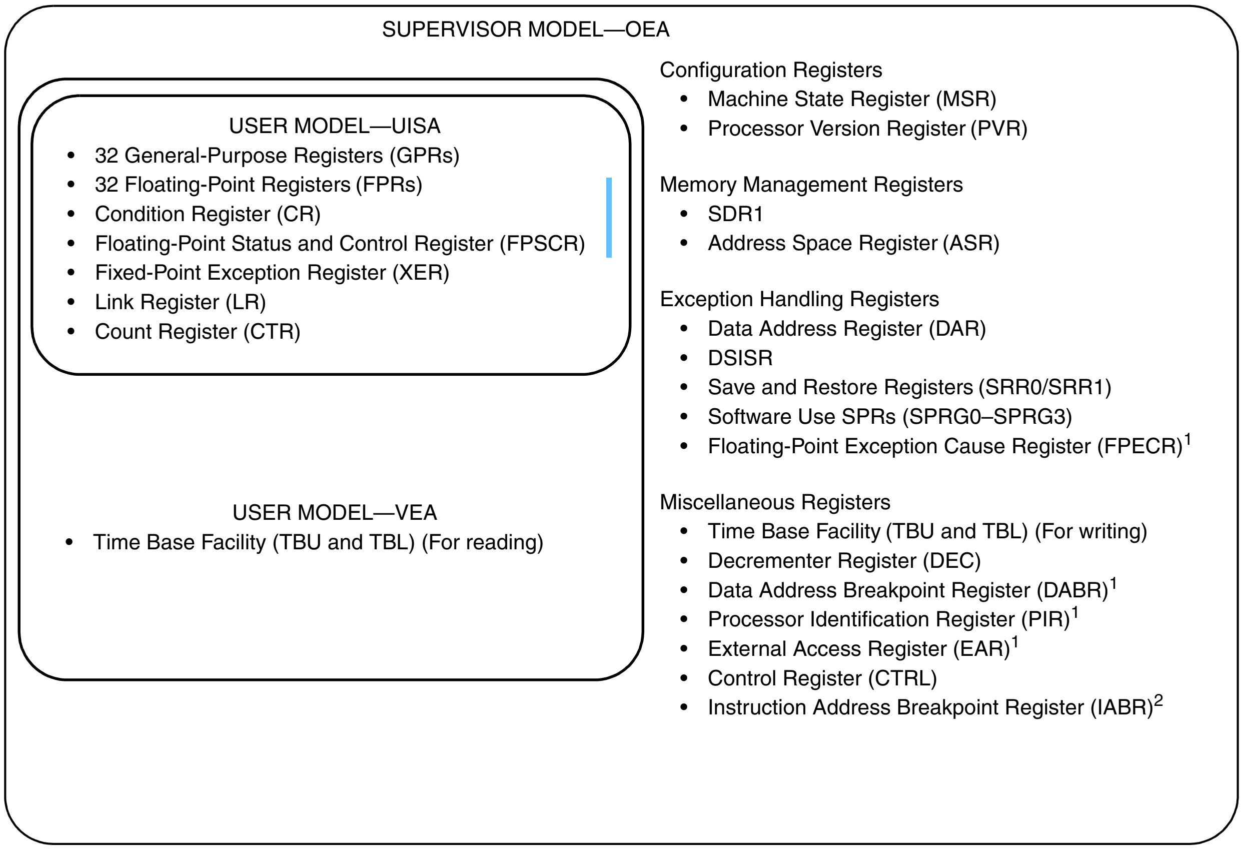
This section provides overviews of aspects defined by the PowerPC Architecture, following the same order as the rest of this book. The topics include the following:

- PowerPC registers and programming model
- PowerPC operand conventions
- PowerPC instruction set and addressing modes
- PowerPC cache model
- PowerPC exception model
- PowerPC memory management model

1.2.1 PowerPC Registers and Programming Model

The PowerPC Architecture defines register-to-register operations for computational instructions. Source operands for these instructions are accessed from the architected registers or are provided as immediate values embedded in the instruction. The three-register instruction format allows specification of a target register distinct from two source operand registers. This scheme allows efficient code scheduling in a highly parallel processor. Load and store instructions are the only instructions that transfer data between registers and memory. The PowerPC registers are shown in *Figure 1-1*.

Figure 1-1. Programming Model—PowerPC Registers


SUPERVISOR MODEL—OEA
USER MODEL—UISA
• 32 General-Purpose Registers (GPRs)
• 32 Floating-Point Registers (FPRs)
• Condition Register (CR)
• Floating-Point Status and Control Register (FPSCR)
• Fixed-Point Exception Register (XER)
• Link Register (LR)
• Count Register (CTR)
USER MODEL—VEA
• Time Base Facility (TBU and TBL) (For reading)
Configuration Registers
• Machine State Register (MSR)
• Processor Version Register (PVR)
Memory Management Registers
• SDR1
• Address Space Register (ASR)
Exception Handling Registers
• Data Address Register (DAR)
• DSISR
• Save and Restore Registers (SRR0/SRR1)
• Software Use SPRs (SPRG0–SPRG3)
• Floating-Point Exception Cause Register (FPECR)1
Miscellaneous Registers
• Time Base Facility (TBU and TBL) (For writing)
• Decrementer Register (DEC)
• Data Address Breakpoint Register (DABR)1
• Processor Identification Register (PIR)1
• External Access Register (EAR)1
• Control Register (CTRL)
• Instruction Address Breakpoint Register (IABR)2

1. Optional
2. Implementation specific register

The programming model incorporates 32 GPRs, 32 FPRs, special-purpose registers (SPRs), and several miscellaneous registers. Each implementation may have its own unique set of hardware implementation (HID) registers that are not defined by the architecture.

PowerPC processors have two levels of privilege:

- Supervisor mode—used exclusively by the operating system. Resources defined by the OEA can be accessed only by supervisor-level software.
- User mode—used by the application software and operating system software. (Only resources defined by the UISA and VEA can be accessed by user-level software.)

These two levels govern the access to registers, as shown in *Figure 1-1*. The division of privilege allows the operating system to control the application environment (providing virtual memory and protecting operating system and critical machine resources). Instructions that control the state of the processor, the address translation mechanism, and supervisor registers can be executed only when the processor is operating in supervisor mode.

- User Instruction Set Architecture Registers—All UISA registers can be accessed by all software with either user or supervisor privileges. These registers include the 32 general-purpose registers (GPRs) and the 32 floating-point registers (FPRs), and other registers used for integer, floating-point, and branch instructions.
- Virtual Environment Architecture Registers—The VEA defines the user-level portion of the time base facility, which consists of the two 32-bit time base registers. These registers can be read by user-level software, but can be written to only by supervisor-level software.
- Operating Environment Architecture Registers—SPRs defined by the OEA are used for system-level operations such as memory management, exception handling, and time-keeping.

The PowerPC Architecture also provides room in the SPR space for implementation-specific registers, typically referred to as HID registers. Individual HIDs are not discussed in this manual.

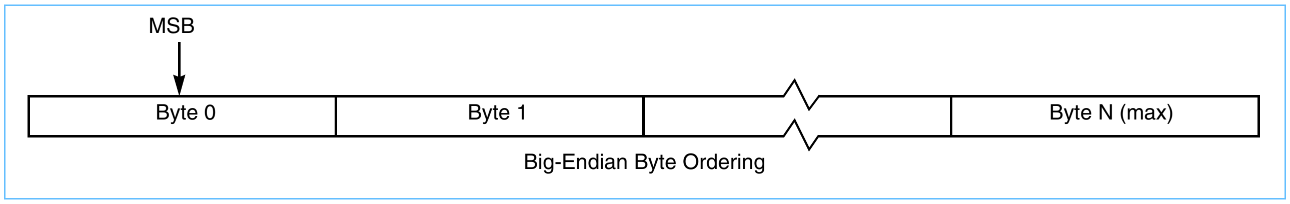
1.2.2 Operand Conventions

Operand conventions are defined in two levels of the PowerPC Architecture—user instruction set architecture (UISA) and virtual environment architecture (VEA). These conventions define how data is stored in registers and memory.

1.2.2.1 Byte Ordering

The default mapping for PowerPC processors is big-endian, but the UISA provides the option of operating in either big or little-endian mode. Big-endian byte ordering is shown in *Figure 1-2*.

Figure 1-2. Big-Endian Byte and Bit Ordering



The OEA defines two bits in the MSR for specifying byte ordering—LE (little-endian mode) and ILE (exception little-endian mode). The LE bit specifies whether the processor is configured for big-endian or little-endian mode; the ILE bit specifies the mode when an exception is taken by being copied into the LE bit of the MSR. A value of ’0’ specifies big-endian mode and a value of 1 specifies little-endian mode.

Note: Little endian mode is optional. If the processor does not support little endian mode, then MSR[LE] and MSR[ILE] are treated as reserved.

Refer to *Section 3.1.2 Byte Ordering* for details on big-endian and little-endian modes.

1.2.2.2 Data Organization in Memory and Data Transfers

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

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

The operand of a single-register memory access instruction has a natural alignment boundary equal to the operand length. In other words, the natural address of an operand is an integral multiple of the operand length. A memory operand is said to be aligned if it is aligned at its natural boundary; otherwise it is misaligned.

1.2.2.3 Floating-Point Conventions

The PowerPC Architecture adheres to the IEEE-754 standard for floating-point arithmetic:

- Double-precision arithmetic instructions may have single or double-precision operands but always produce double-precision results.
- Single-precision arithmetic instructions require all operands to be single-precision values and always produce single-precision results. Single-precision values are stored in double-precision format in the FPRs—these values are rounded such that they can be represented in 32-bit, single-precision format (as they are in memory).

1.2.3 PowerPC Instruction Set and Addressing Modes

All PowerPC instructions are encoded as single-word (32-bit) instructions. Instruction formats are consistent among all instruction types, permitting decoding to occur in parallel with operand accesses. This fixed instruction length and consistent format greatly simplifies instruction pipelining.

1.2.3.1 PowerPC Instruction Set

Although these categories are not defined by the PowerPC Architecture, the PowerPC instructions can be grouped as follows:

- Integer instructions—These instructions are defined by the UISA. They include computational and logical instructions. For example, integer arithmetic instructions, integer compare instructions, logical instructions, and integer rotate and shift instructions.
- Floating-point instructions—These instructions, defined by the UISA, include floating-point computational instructions, as well as instructions that manipulate the floating-point status and control register (FPSCR). For example, floating-point arithmetic instructions, floating-point multiply/add instructions, floating-point compare instructions, floating-point status and control instructions, floating-point move instructions, and optional floating-point instructions.
- Load/store instructions—These instructions, defined by the UISA, include integer and floating-point load and store instructions. For example, integer load and store instructions, integer load and store with byte reverse instructions, integer load and store multiple instructions, integer load and store string instructions, and floating-point load and store instructions.

- The UISA also provides a set of load/store with reservation instructions (**lwarx**/**ldarx** and **stwcx.**/**stdcx.**) that can be used as primitives for constructing atomic memory operations in multiprocessing environments. These are grouped under synchronization instructions.
- Synchronization instructions—The UISA and VEA define instructions for memory synchronizing, especially useful for multiprocessing. For example, load and store with reservation instructions (these UISA-defined instructions provide primitives for synchronization operations such as test and set, compare and swap, and compare memory). The synchronization instruction (**sync**) is useful for synchronizing load and store operations on a memory bus that is shared by multiple devices. The Enforce In-Order Execution of I/O (**eieio**) instruction provides an ordering function for the effects of load and store operations executed by a processor.
- Flow control instructions—These include branching instructions, condition register logical instructions, trap instructions, and other instructions that affect the instruction flow. The UISA defines numerous instructions that control the program flow, including branch, trap, and system call instructions, as well as instructions that read, write, or manipulate bits in the condition register. The OEA defines two flow control instructions that provide system linkage (**sc**, **rfid**). These instructions are used for entering and returning from supervisor level.
- Processor control instructions—These instructions are used for synchronizing memory accesses and managing caches and translation lookaside buffers (TLBs). These instructions include move to/from special-purpose register instructions (**mtspr** and **mfspr**).
- Memory/cache control instructions—These instructions provide control of caches, SLBs, and TLBs. The VEA defines several cache control instructions. The OEA defines several memory control instructions.
- External control instructions—The VEA defines two optional instructions (**eciwx**, **ecowx**) for use with special input/output devices.

Temporary 64-Bit Bridge

- The 64-bit bridge allows several instructions to be used in 64-bit implementations that are otherwise defined for use in 32-bit implementations only. These include the following:
  – Move to Segment Register (**mtsr**) and Move to Segment Register Indirect (**mtsrin**)
  – Move from Segment Register (**mfsr**) and Move from Segment Register Indirect (**mfsrin**)

  All four of these instructions are implemented as a group and are never implemented individually. Attempting to execute one of these instructions on a 64-bit implementation on which these instructions are not supported causes program exception.

Note: This grouping of the instructions does not indicate which execution unit executes a particular instruction or group of instructions. This is not defined by the PowerPC Architecture.

1.2.3.2 Calculating Effective Addresses

The effective address (EA), also called the logical address, is the address computed by the processor when executing a memory access or branch instruction or when fetching the next sequential instruction. Unless address translation is disabled, this address is converted by the MMU to the appropriate physical address.

Note: The architecture specification uses only the term effective address and not logical address.

The PowerPC Architecture supports the following simple addressing modes for memory access instructions:

- EA = (**r**A|0) (register indirect)
- EA = (**r**A|0) + offset (including offset = 0) (register indirect with immediate index)
- EA = (**r**A|0) + **r**B (register indirect with index)

These simple addressing modes allow efficient address generation for memory accesses.

1.2.4 PowerPC Cache Model

The VEA and OEA portions of the architecture define aspects of cache implementations for PowerPC processors. The PowerPC Architecture does not define hardware aspects of cache implementations. For example, some PowerPC processors may have separate instruction and data caches (Harvard architecture), while others have a unified cache.

The PowerPC Architecture allows implementations to control the following memory access modes on a page basis:

- Write-back/write-through mode
- Caching-inhibited mode
- Memory coherency
- Guarded/not guarded against speculative accesses

Coherency is maintained on a cache block basis, and cache control instructions perform operations on a cache block basis. The size of the cache block is implementation-dependent. The term cache block should not be confused with the notion of a block in memory, which is described in *Section 1.2.6 PowerPC Memory Management Model*.

The VEA portion of the PowerPC Architecture defines several instructions for cache management. These can be used by user-level software to perform such operations as touch operations (which cause the cache block to be speculatively loaded), and operations to store, flush, or clear the contents of a cache block.

1.2.5 PowerPC Exception Model

The PowerPC exception mechanism, defined by the OEA, allows the processor to change to supervisor state as a result of external signals, errors, or unusual conditions arising in the execution of instructions. When exceptions occur, information about the state of the processor is saved to various registers and the processor begins execution at an address (exception vector) predetermined for each type of exception. Exception handler routines begin execution in supervisor mode. The PowerPC exception model is described in detail in *Chapter 6, Exceptions*.

Note: Some aspects regarding exception conditions are defined at other levels of the architecture. For example, floating-point exception conditions are defined by the UISA, whereas the exception mechanism is defined by the OEA.

The PowerPC Architecture requires that exceptions be handled in program order (excluding the optional floating-point imprecise modes and the reset and machine check exception); therefore, although a particular implementation may recognize exception conditions out of order, they are handled strictly in order. When an instruction-caused exception is recognized, any unexecuted instructions that appear earlier in the instruction stream, including any that have not yet begun to execute, are required to complete before the exception is taken. Any exceptions caused by those instructions must be handled first. Likewise, exceptions that are asynchronous and precise are recognized when they occur, but are not handled until all instructions currently executing successfully complete processing and report their results.

The OEA supports four types of exceptions:

- Synchronous, precise
- Synchronous, imprecise
- Asynchronous, maskable
- Asynchronous, nonmaskable

1.2.6 PowerPC Memory Management Model

The PowerPC memory management unit (MMU) specifications are provided by the PowerPC OEA. The primary functions of the MMU in a PowerPC processor are to translate logical (effective) addresses to physical addresses for memory accesses and I/O accesses (most I/O accesses are assumed to be memory-mapped), and to provide access protection on a block or page basis.

Note: Many aspects of memory management are implementation-dependent. The description in *Chapter 7, Memory Management* describes the conceptual model of a PowerPC MMU; however, PowerPC processors may differ in the specific hardware used to implement the MMU model of the OEA.

PowerPC processors require address translation for two types of transactions—instruction accesses and data accesses to memory (typically generated by load and store instructions).

The memory management specification of the PowerPC OEA includes models for both 32 and 64-bit implementations. The MMU of a 64-bit PowerPC processor provides 2^{64} bytes of effective address space accessible to supervisor and user programs with support for two page sizes; a 4-Kbyte page size (2^{12}) and a large page whose size is implementation dependent (2^p where $13 \leq p \leq 28$). The MMU of 64-bit PowerPC processors uses an interim virtual address (between 65 and 80 bits) and hashed page tables in the generation of physical addresses that are $\leq$ 62 bits in length. *Table 7-1 MMU Features Summary* summarizes the features of PowerPC MMUs for 64-bit implementations.

Two types of accesses generated by PowerPC processors require address translation: instruction accesses, and data accesses to memory generated by load and store instructions. The address translation mechanism is defined in terms of segment tables and page tables used by PowerPC processors to locate the logical-to-physical address mapping for instruction and data accesses. The segment information translates the logical address to an interim virtual address, and the page table information translates the virtual address to a physical address.

Translation lookaside buffers (TLBs) are commonly implemented in PowerPC processors to keep recently-used page table entries on-chip. Although their exact characteristics are not specified by the architecture, the general concepts that are pertinent to the system software are described. Similarly, 64-bit implementations contain segment lookaside buffers (SLBs) on-chip that contain recently-used segment table entries, however the PowerPC Architecture does not define the exact characteristics for SLBs.

Temporary 64-Bit Bridge

The 64-bit bridge provides resources that may make it easier for a 32-bit operating system to migrate to a 64-bit processor. The nature of these resources are largely determined by the fact that in a 32-bit address space, only 16 segment descriptors are required to define all 4 Gbytes of memory. That is, there are sixteen 256 Mbyte segments, as is the case in the 32-bit architecture description.

1.3 Changes to this Manual

This manual reflects changes made to the PowerPC Architecture, Version 2.01.

2. PowerPC Register Set

This chapter describes the register organization defined by the three levels of the PowerPC Architecture:

- User instruction set architecture (UISA)
- Virtual environment architecture (VEA), and
- Operating environment architecture (OEA).

The PowerPC Architecture defines register-to-register operations for all computational instructions. Source data for these instructions are accessed from the on-chip registers or are provided as immediate values embedded in the opcode. The three-register instruction format allows specification of a target register distinct from the two source registers, thus preserving the original data for use by other instructions and reducing the number of instructions required for certain operations. Data is transferred between memory and registers with explicit load and store instructions only.

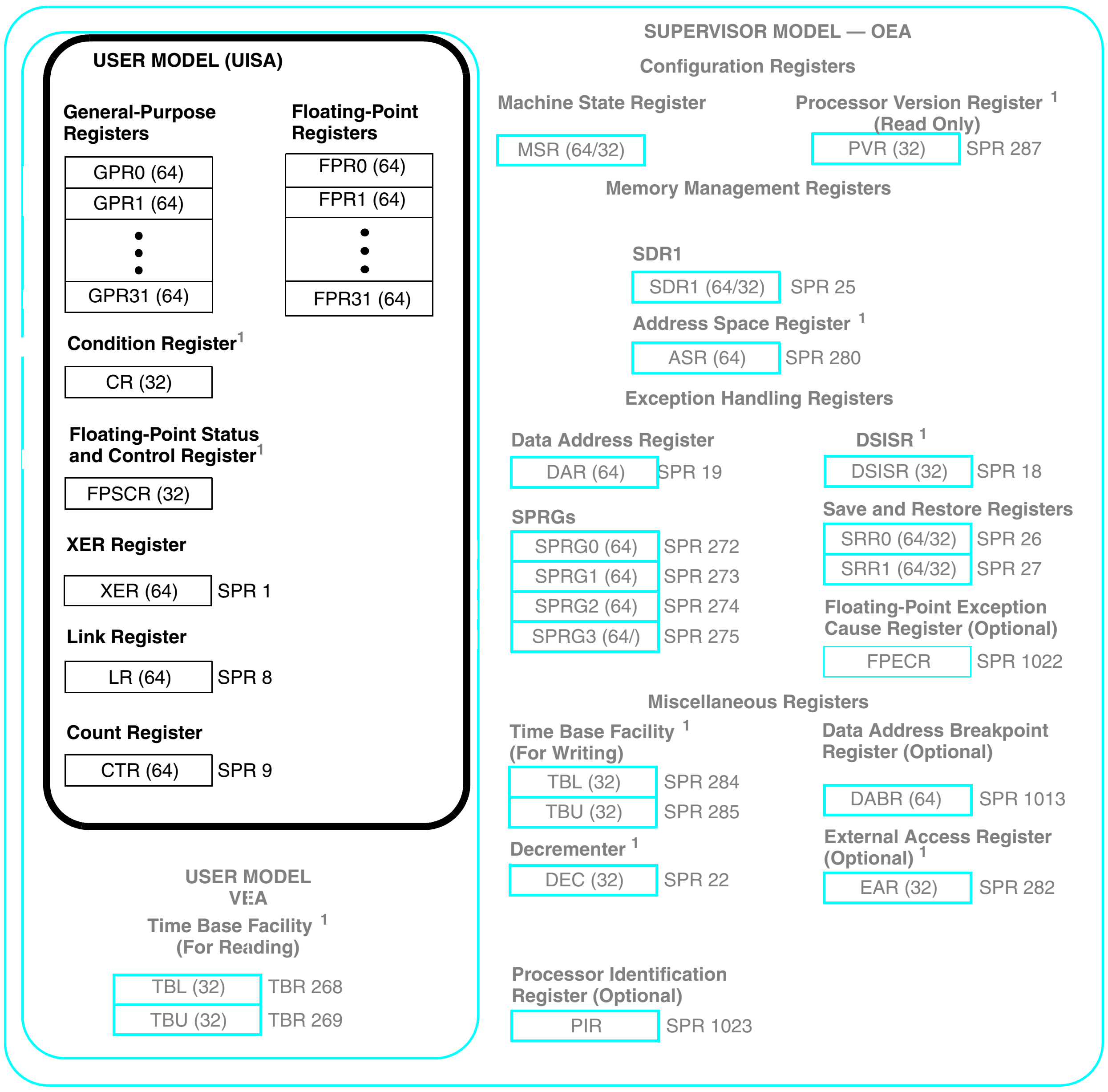
Note: The handling of reserved bits in any register is implementation-dependent. Software is permitted to write any value to a reserved bit in a register. However, a subsequent reading of the reserved bit returns '0' if the value last written to the bit was '0' and returns an undefined value (may be '0' or '1') otherwise. This means that even if the last value written to a reserved bit was '1', reading that bit may return '0'.

2.1 Overview of the PowerPC UISA Registers

The PowerPC UISA registers, shown in *Figure 2-1*, can be accessed by either user or supervisor-level instructions (the architecture specification refers to user-level and supervisor-level as problem state and privileged state respectively). The general-purpose registers (GPRs) and floating-point registers (FPRs) are accessed as instruction operands. Access to registers can be explicit (that is, through the use of specific instructions for that purpose such as Move to Special-Purpose Register (**mtspr**) and Move from Special-Purpose Register (**mfspr**) instructions) or implicit as part of the execution of an instruction. Some registers are accessed both explicitly and implicitly.

The number to the right of the register name indicates the number that is used in the syntax of the instruction operand to access the register (for example, the number used to access the XER is SPR 1).

Figure 2-1. UISA Programming Model—User-Level Registers



1. These registers are on 64-bit implementations only.
2. These registers are implementation dependent.
3. 64-bit registers operating in 32-bit mode clear the high order 32-bits.

The user-level registers can be accessed by all software with either user or supervisor privileges. The user-level registers are:

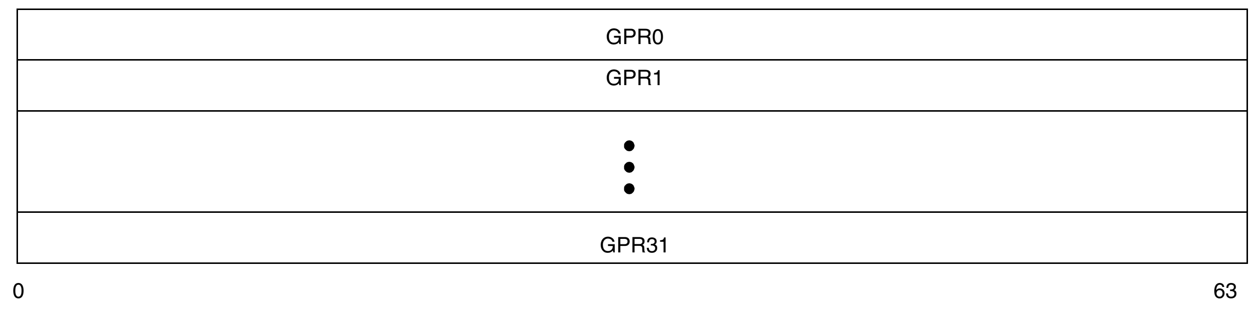
- General-purpose registers (GPRs). The general-purpose register file consists of 32 GPRs designated as GPR0–GPR31. The GPRs serve as either the data source or the destination registers for all integer instructions and provide data for generating addresses. For more information see *Section 2.1.1 General-Purpose Registers (GPRs)* on page 44.
- Floating-point registers (FPRs). The floating-point register file consists of 32 FPRs designated as FPR0–FPR31; these registers serve as either the data source or the destination for all floating-point instructions. While the floating-point model includes data objects of either single or double-precision floating-point format, the FPRs only contain data in double-precision format. For more information, see *Section 2.1.2 Floating-Point Registers (FPRs)* on page 44.
- Condition register (CR). The condition register is a 32-bit register that is divided into eight 4-bit fields, CR0–CR7. This register stores the results of certain arithmetic operations and provides a mechanism for testing and branching. For more information, see *Section 2.1.3 Condition Register (CR)* on page 45.
- Floating-point status and control register (FPSCR). The floating-point status and control register contains all floating-point exception signal bits, exception summary bits, exception enable bits, and rounding control bits needed for compliance with the IEEE 754 standard. For more information, see *Section 2.1.4 Floating-Point Status and Control Register (FPSCR)* on page 47.

  Note: The architecture specification refers to exceptions as interrupts.
- Fixed point exception register (XER). The fixed point exception register indicates overflows and carry conditions for integer operations and the number of bytes to be transferred by the load/store string indexed instructions. For more information, see *Section 2.1.5 XER Register (XER)* on page 50.
- Link register (LR). The link register provides the branch target address for the Branch Conditional to Link Register (**bclr***x*) instructions, and can optionally be used to hold the effective address of the instruction that follows a branch with link update instruction in the instruction stream, typically used for loading the return pointer for a subroutine. For more information, see *Section 2.1.6 Link Register (LR)* on page 51.
- Count register (CTR). The count register holds a loop count that can be decremented during execution of appropriately coded branch instructions. The CTR can also provide the branch target address for the Branch Conditional to Count Register (**bcctr***x*) instructions. For more information, see *Section 2.1.7 Count Register (CTR)* on page 52.

2.1.1 General-Purpose Registers (GPRs)

Integer data is manipulated in the processor's 32 GPRs shown in *Figure 2-2*. These registers are 64-bit registers. The GPRs are accessed as either source or destination registers in the instruction syntax.

Figure 2-2. General-Purpose Registers (GPRs)



2.1.2 Floating-Point Registers (FPRs)

The PowerPC Architecture provides thirty-two 64-bit FPRs as shown in *Figure 2-3*. These registers are accessed as either source or destination registers for floating-point instructions. Each FPR supports the double-precision floating-point format. Every instruction that interprets the contents of an FPR as a floating-point value uses the double-precision floating-point format for this interpretation.

Instructions for all floating-point arithmetic operations use the data located in the FPRs and, with the exception of compare instructions, place the result into a FPR. Information about the status of floating-point operations is placed into the FPSCR and in some cases, into the CR after the completion of instruction execution. For information on how the CR is affected for floating-point operations, see *Section 2.1.3 Condition Register (CR)*.

Instructions to load and to store floating-point double precision values transfer 64 bits of data between memory and the FPRs with no conversion.

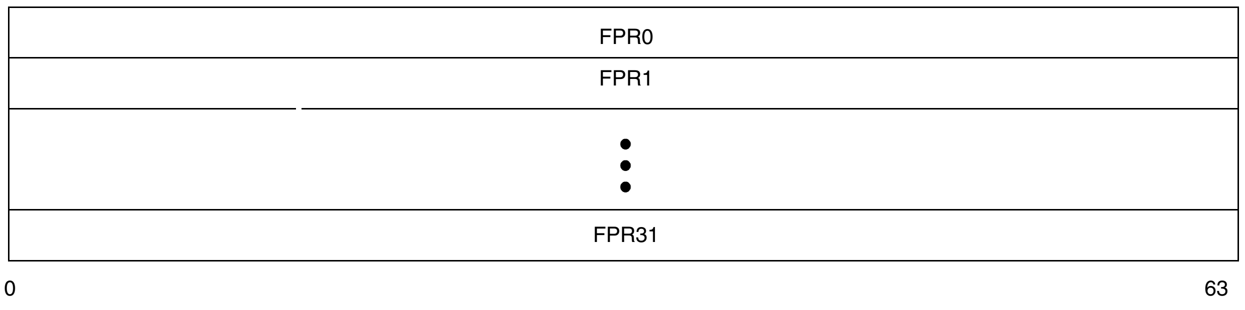
Instructions to load floating-point single precision values are provided to read single-precision floating-point values from memory, convert them to double-precision floating-point format, and place them in the target floating-point register.

Instructions to store single-precision values are provided to read double-precision floating-point values from a floating-point register, convert them to single-precision floating-point format, and place them in the target memory location.

Instructions for single and double-precision arithmetic operations accept values from the FPRs in double-precision format. For instructions of single-precision arithmetic and store operations, all input values must be representable in single-precision format; otherwise, the results placed into the target FPR (or the memory location) and the setting of status bits in the FPSCR and in the condition register (if the instruction's record bit, Rc, is set) are undefined.

The floating-point arithmetic instructions produce intermediate results that may be regarded as infinitely precise and with unbounded exponent range. This intermediate result is normalized or denormalized if required, and then rounded to the destination format. The final result is then placed into the target FPR in the double-precision format or in fixed-point format, depending on the instruction. Refer to *Section 3.3 Floating-Point Execution Models—UISA* on page 92 for more information.

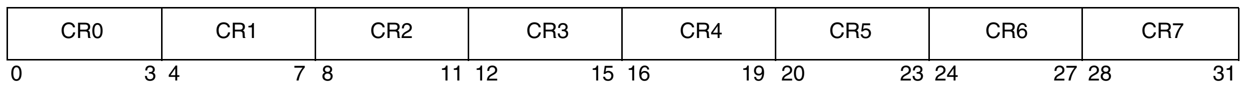
Figure 2-3. Floating-Point Registers (FPRs)



2.1.3 Condition Register (CR)

The condition register (CR) is a 32-bit register that reflects the result of certain operations and provides a mechanism for testing and branching. The bits in the CR are grouped into eight 4-bit fields, CR0–CR7, as shown in *Figure 2-4*.

Figure 2-4. Condition Register (CR)



The CR fields can be set in one of the following ways:

- Specified fields of the CR can be set from a GPR by using the **mtcrf** and **mtocrf** instruction.
- The contents of the XER[0–3] can be moved to another CR field by using the **mcrf** instruction.
- A specified field of the XER can be copied to a specified field of the CR by using the **mcrxr** instruction.
- A specified field of the FPSCR can be copied to a specified field of the CR by using the **mcrfs** instruction.
- Logical instructions of the condition register can be used to perform logical operations on specified bits in the condition register.
- CR0 can be the implicit result of an integer instruction.
- CR1 can be the implicit result of a floating-point instruction.
- A specified CR field can indicate the result of either an integer or floating-point compare instruction.

Note: Branch instructions are provided to test individual CR bits.

2.1.3.1 Condition Register CR0 Field Definition

For all integer instructions, when the CR is set to reflect the result of the operation (that is, when Rc = ’1’), and for **addic.**, **andi.**, and **andis.**, the first three bits of CR0 are set by an algebraic comparison of the result to zero; the fourth bit of CR0 is copied from XER[SO]. For integer instructions, CR bits [0–3] are set to reflect the result as a signed quantity.

The CR bits are interpreted as shown in *Table 2-1*. If any portion of the result is undefined, the value placed into the first three bits of CR0 is undefined. The **stwcx.** and **stdcx.** instructions also set the CR0 field.

Table 2-1. Bit Settings for CR0 Field of CR

CR0 Bit	Description
0	Negative (LT)—This bit is set when the result is negative.
1	Positive (GT)—This bit is set when the result is positive (and not zero).
2	Zero (EQ)—This bit is set when the result is zero or when a **stwcx.** or **stdcx.** successfully completes.
3	Summary overflow (SO)—This is a copy of the final state of XER[SO] at the completion of the instruction.

Note: If overflow occurs, CR0 may not reflect the true (infinitely precise) result. CR0 bits [0–2] are undefined if Rc = 1 for the **mulhw**, **mulhwu**, **divw**, and **divwu** instructions.

2.1.3.2 Condition Register CR1 Field Definition

In all floating-point instructions when the CR is set to reflect the result of the operation (Rc =1), CR1 (bits [4-7] of the CR) is copied from bits [0–3] of the FPSCR and indicates the floating-point exception status. For more information about the FPSCR, see *Section 2.1.4 Floating-Point Status and Control Register (FPSCR)*. The bit settings for the CR1 field are shown in *Table 2-2*.

Table 2-2. Bit Settings for CR1 Field of CR

CR1 Bit	Description
4	Floating-point exception summary (FX)—This is a copy of the final state of FPSCR[FX] at the completion of the instruction.
5	Floating-point enabled exception summary (FEX)—This is a copy of the final state of FPSCR[FEX] at the completion of the instruction.
6	Floating-point invalid operation exception summary (VX)—This is a copy of the final state of FPSCR[VX] at the completion of the instruction.
7	Floating-point overflow exception (OX)—This is a copy of the final state of FPSCR[OX] at the completion of the instruction.

2.1.3.3 Condition Register CRn Field—Compare Instruction

For a compare instruction, when a specified CR field is set to reflect the result of the comparison, the bits of the specified field are interpreted as shown in *Table 2-3*.

Table 2-3. CRn Field Bit Settings for Compare Instructions

CR*n* Bit [1]	Description [2]
0	Less than or floating-point less than (LT, FL). For integer compare instructions: **r**A < SIMM or **r**B (signed comparison) or **r**A < UIMM or **r**B (unsigned comparison). For floating-point compare instructions: **fr**A < **fr**B.
1	Greater than or floating-point greater than (GT, FG). For integer compare instructions: **r**A > SIMM or **r**B (signed comparison) or **r**A > UIMM or **r**B (unsigned comparison). For floating-point compare instructions: **fr**A > **fr**B.
2	Equal or floating-point equal (EQ, FE). For integer compare instructions: **r**A = SIMM, UIMM, or **r**B. For floating-point compare instructions: **fr**A = **fr**B.
3	Summary overflow or floating-point unordered (SO, FU). For integer compare instructions: This is a copy of the final state of XER[SO] at the completion of the instruction. For floating-point compare instructions: One or both of **fr**A and **fr**B is a Not a Number (NaN).

Notes:
1. Here, the bit indicates the bit number in any one of the 4-bit subfields, CR0–CR7.
2. For a complete description of instruction syntax conventions, refer to *Table 8-2* on page 300.

2.1.4 Floating-Point Status and Control Register (FPSCR)

The Floating-Point Status and Control Register (FPSCR), shown in *Figure 2-5*, is used for:

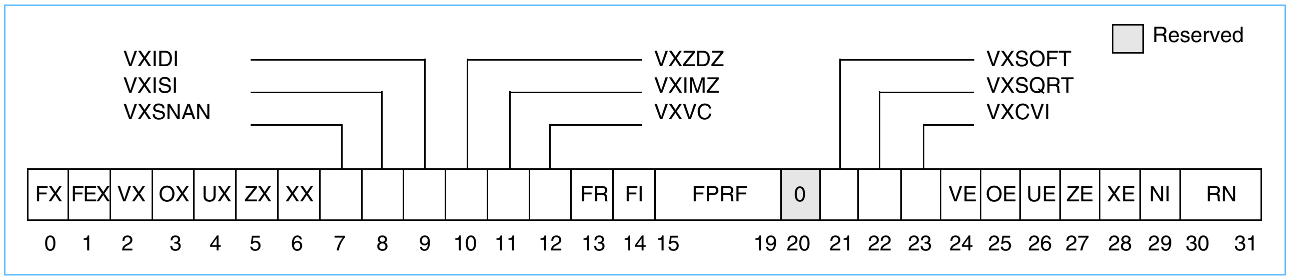
- Recording exceptions generated by floating-point operations
- Recording the type of the result produced by a floating-point operation
- Controlling the rounding mode used by floating-point operations
- Enabling or disabling the reporting of exceptions (that is, invoking the exception handler)

Bits [0–23] are status bits. Bits [24–31] are control bits. Status bits in the FPSCR are updated at the completion of the instruction execution.

Except for the floating-point enabled exception summary (FEX) and floating-point invalid operation exception summary (VX), the exception condition bits in the FPSCR (bits [3–12] and [21–23]) are sticky. Once set, sticky bits remain set until they are cleared by the relevant **mcrfs**, **mtfsfi**, **mtfsf**, or **mtfsb0** instruction.

FEX and VX are the logical ORs of other FPSCR bits. Therefore, these two bits are not listed among the FPSCR bits directly affected by the various instructions.

Figure 2-5. Floating-Point Status and Control Register (FPSCR)



A listing of FPSCR bit settings is shown in *Table 2-4*.

Table 2-4. FPSCR Bit Settings

Bit(s)	Name	Description
0	FX	Floating-point exception summary. Every floating-point instruction, except **mtfsfi** and **mtfsf**, implicitly sets FPSCR[FX] if that instruction causes any of the floating-point exception bits in the FPSCR to transition from ‘0’ to ‘1’. The **mcrfs**, **mtfsfi**, **mtfsf**, **mtfsb0**, and **mtfsb1** instructions can alter FPSCR[FX] explicitly. This is a sticky bit.
1	FEX	Floating-point enabled exception summary. This bit signals the occurrence of any of the enabled exception conditions. It is the logical OR of all the floating-point exception bits masked by their respective enable bits (FEX = (VX & VE) ^ (OX & OE) ^ (UX & UE) ^ (ZX & ZE) ^ (XX & XE)). The **mcrfs**, **mtfsf**, **mtfsfi**, **mtfsb0**, and **mtfsb1** instructions cannot alter FPSCR[FEX] explicitly. This is not a sticky bit.
2	VX	Floating-point invalid operation exception summary. This bit signals the occurrence of any invalid operation exception. It is the logical OR of all of the invalid operation exceptions. The **mcrfs**, **mtfsf**, **mtfsfi**, **mtfsb0**, and **mtfsb1** instructions cannot alter FPSCR[VX] explicitly. This is not a sticky bit.
3	OX	Floating-point overflow exception. This is a sticky bit. See *Section 3.3.6.2 Overflow, Underflow, and Inexact Exception Conditions* on page 113.
4	UX	Floating-point underflow exception. This is a sticky bit. See *Underflow Exception Condition* on page 116.
5	ZX	Floating-point zero divide exception. This is a sticky bit. See *Zero Divide Exception Condition* on page 112.
6	XX	Floating-point inexact exception. This is a sticky bit. See *Inexact Exception Condition* on page 117. FPSCR[XX] is the sticky version of FPSCR[FI]. The following rules 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 logically 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.
7	VXSNAN	Floating-point invalid operation exception for SNaN. This is a sticky bit. See *Invalid Operation Exception Condition* on page 111.
8	VXISI	Floating-point invalid operation exception for $\infty - \infty$. This is a sticky bit. See *Invalid Operation Exception Condition* on page 111.
9	VXIDI	Floating-point invalid operation exception for $\infty \div \infty$. This is a sticky bit. See *Invalid Operation Exception Condition* on page 111.
10	VXZDZ	Floating-point invalid operation exception for $0 \div 0$. This is a sticky bit. See *Invalid Operation Exception Condition* on page 111.
11	VXIMZ	Floating-point invalid operation exception for $\infty * 0$. This is a sticky bit. See *Invalid Operation Exception Condition* on page 111.
12	VXVC	Floating-point invalid operation exception for invalid compare. This is a sticky bit. See *Invalid Operation Exception Condition* on page 111.
13	FR	Floating-point fraction rounded. The last arithmetic or rounding and conversion instruction that rounded the intermediate result incremented the fraction. See *Section 3.3.5 Rounding*. This bit is not sticky.

Table 2-4. FPSCR Bit Settings (Continued)

Bit(s)	Name	Description
14	FI	Floating-point fraction inexact. The last arithmetic or rounding and conversion instruction either rounded the intermediate result (producing an inexact fraction) or caused a disabled overflow exception. See *Section 3.3.5 Rounding*. This is not a sticky bit. For more information regarding the relationship between FPSCR[FI] and FPSCR[XX], see the description of the FPSCR[XX] bit.
15–19	FPRF	Floating-point result flags. For arithmetic, rounding, and conversion instructions, the field is based on the result placed into the target register, except that if any portion of the result is undefined, the value placed here is undefined. 15 Floating-point result class descriptor (C). Arithmetic, rounding, and conversion instructions may set this bit with the FPCC bits to indicate the class of the result as shown in *Table 2-5*. 16–19 Floating-point condition code (FPCC). Floating-point compare instructions always set one of the FPCC bits to one and the other three FPCC bits to zero. 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. 16 Floating-point less than or negative (FL or <) 17 Floating-point greater than or positive (FG or >) 18 Floating-point equal or zero (FE or =) 19 Floating-point unordered or NaN (FU or ?) **Note:** These are not sticky bits.
20	—	Reserved
21	VXSOFT	Floating-point invalid operation exception for software request. This is a sticky bit. This bit can be altered only by the **mcrfs**, **mtfsfi**, **mtfsf**, **mtfsb0**, or **mtfsb1** instructions. For more detailed information, refer to *Invalid Operation Exception Condition* on page 111.
22	VXSQRT	Floating-point invalid operation exception for invalid square root. This is a sticky bit. For more detailed information, refer to *Invalid Operation Exception Condition* on page 111. **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.
23	VXCVI	Floating-point invalid operation exception for invalid integer convert. This is a sticky bit. See *Invalid Operation Exception Condition* on page 111.
24	VE	Floating-point invalid operation exception enable. See *Invalid Operation Exception Condition* on page 111.
25	OE	IEEE floating-point overflow exception enable. See *Section 3.3.6.2 Overflow, Underflow, and Inexact Exception Conditions* on page 113.
26	UE	IEEE floating-point underflow exception enable. See *Underflow Exception Condition* on page 116.
27	ZE	IEEE floating-point zero divide exception enable. See *Zero Divide Exception Condition* on page 112.
28	XE	Floating-point inexact exception enable. See *Inexact Exception Condition* on page 117.
29	NI	Floating-point non-IEEE mode. If this bit is set, results need not conform with IEEE standards and the other FPSCR bits may have meanings other than those described here. If the bit is set and if all implementation-specific requirements are met and if an IEEE-conforming result of a floating-point operation would be a denormalized number, the result produced is zero (retaining the sign of the denormalized number). Any other effects associated with setting this bit are described in the user's manual for the implementation (the effects are implementation-dependent). **Note:** When the processor is in floating-point non-IEEE mode, the results of floating-point operations may be approximate, and performance for these operations may be better, more predictable, or less data-dependent than when the processor is not in non-IEEE mode. For example, in non-IEEE mode an implementation may return 0 instead of a denormalized number, and may return a large number instead of an infinity.
30–31	RN	Floating-point rounding control. See *Section 3.3.5 Rounding*. 00 Round to nearest 01 Round toward zero 10 Round toward +infinity 11 Round toward –infinity

Table 2-5 illustrates the floating-point result flags used by PowerPC processors. The result flags correspond to FPSCR bits [15–19].

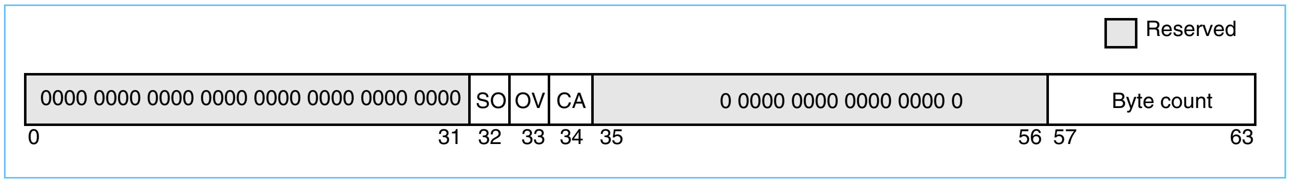
Table 2-5. Floating-Point Result Flags in FPSCR

Result Flags (Bits [15–19])					Result Value Class
C	<	>	=	?	
1	0	0	0	1	Quiet NaN
0	1	0	0	1	–Infinity
0	1	0	0	0	–Normalized number
1	1	0	0	0	–Denormalized number
1	0	0	1	0	–Zero
0	0	0	1	0	+Zero
1	0	1	0	0	+Denormalized number
0	0	1	0	0	+Normalized number
0	0	1	0	1	+Infinity

2.1.5 XER Register (XER)

The fixed-point exception register (XER) is a 64-bit, user-level register and is described in *Figure 2-6* and *Table 2-6*.

Figure 2-6. XER Register



The bit definitions for XER, shown in *Table 2-6*, are based on the operation of an instruction considered as a whole, not on intermediate results. For example, the result of the Subtract from Carrying (**subfc***x*) instruction is specified as the sum of three values. This instruction sets bits in the XER based on the entire operation, not on an intermediate sum.

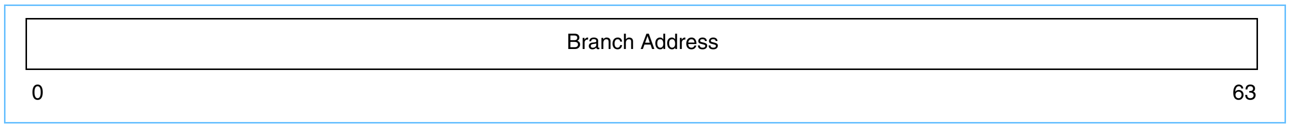
Table 2-6. XER Bit Definitions

Bit(s)	Name	Description
0-31	–	Reserved.
32	SO	Summary overflow. The summary overflow bit [SO] is set whenever an instruction (except **mtspr**) sets the overflow bit [OV]. Once set, the [SO] bit remains set until it is cleared by an **mtspr** instruction (specifying the XER) or an mcrxr instruction. It is not altered by compare instructions, nor by other instructions (except **mtspr** to the XER, and mcrxr) that cannot overflow. Executing an **mtspr** instruction to the XER, supplying the values zero for [SO] and one for [OV], causes [SO] to be cleared and [OV] to be set.
33	OV	Overflow. The overflow bit [OV] is set to indicate that an overflow has occurred during execution of an instruction. Add, subtract from, and negate instructions having OE = ’1’ set the [OV] bit if the carry out of the msb is not equal to the carry out of the msb + 1, and clear it otherwise. Multiply low and divide instructions having OE = ’1’ set the [OV] bit if the result cannot be represented in 64 bits (**mulld**, **divd**, **divdu**) or in 32 bits (**mullw**, **divw**, **divwu**), and clear it otherwise. The [OV] bit is not altered by compare instructions, nor by other instructions that cannot overflow (except **mtspr** to the XER, and **mcrxr**).
34	CA	Carry. The carry bit [CA] is set during execution of the following instructions: • Add carrying, subtract from carrying, add extended, and subtract from extended instructions set [CA] if there is a carry out of the msb, and clear it otherwise. • Shift right algebraic instructions set [CA] if any 1-bits have been shifted out of a negative operand, and clear it otherwise. The [CA] bit is not altered by compare instructions, nor by other instructions that cannot carry (except shift right algebraic, **mtspr** to the XER, and **mcrxr**).
35–56	—	Reserved
57–63		This field specifies the number of bytes to be transferred by a Load String Word Indexed (**lswx**) or Store String Word Indexed (**stswx**) instruction.

2.1.6 Link Register (LR)

The link register (LR) is a 64-bit register that supplies the branch target address for the Branch Conditional to Link Register (**bclr***x*) instructions, and in the case of a branch with link update instruction, can be used to hold the logical address of the instruction that follows the branch with link update instruction (for returning from a subroutine). The format of LR is shown in *Figure 2-7*.

Figure 2-7. Link Register (LR)



Note: Although the two least-significant bits can accept any values written to them, they are ignored when the LR is used as an address. Both conditional and unconditional branch instructions include the option of placing the logical address of the instruction following the branch instruction in the LR.

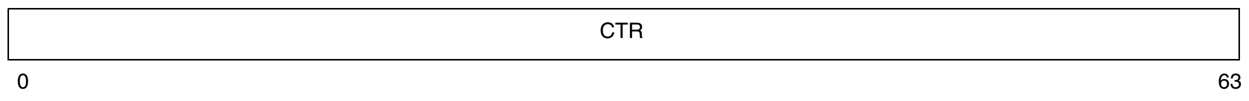
The link register can be also accessed by the **mtspr** and **mfspr** instructions using SPR 8. Prefetching instructions along the target path (loaded by an **mtspr** instruction) is possible provided the link register is loaded sufficiently ahead of the branch instruction (so that any branch prediction hardware can calculate the branch address). Additionally, PowerPC processors can prefetch along a target path loaded by a branch and link instruction.

Note: Some PowerPC processors may keep a stack of the LR values most recently set by branch with link update instructions. To benefit from these enhancements, use of the link register should be restricted to the manner described in *Section 4.2.4.2 Conditional Branch Control*.

2.1.7 Count Register (CTR)

The count register (CTR) is a 64-bit register that can hold a loop count that can be decremented during execution of branch instructions that contain an appropriately coded BO field. If the value in CTR is 0 before being decremented, it is -1 afterward; (0xFFFF_FFFF_FFFF_FFFF ($2^{64} – 1$). The CTR can also provide the branch target address for the Branch Conditional to Count Register (**bcctr***x*) instruction. The CTR is shown in *Figure 2-8*.

Figure 2-8. Count Register (CTR)



Prefetching instructions along the target path is also possible provided the count register is loaded sufficiently ahead of the branch instruction (so that any branch prediction hardware can calculate the correct value of the loop count).

The count register can also be accessed by the **mtspr** and **mfspr** instructions by specifying SPR 9. 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 CR and the CTR. The encoding for the BO field is shown in *Table 4-20 BO Operand Encodings*.

2.2 PowerPC VEA Register Set—Time Base

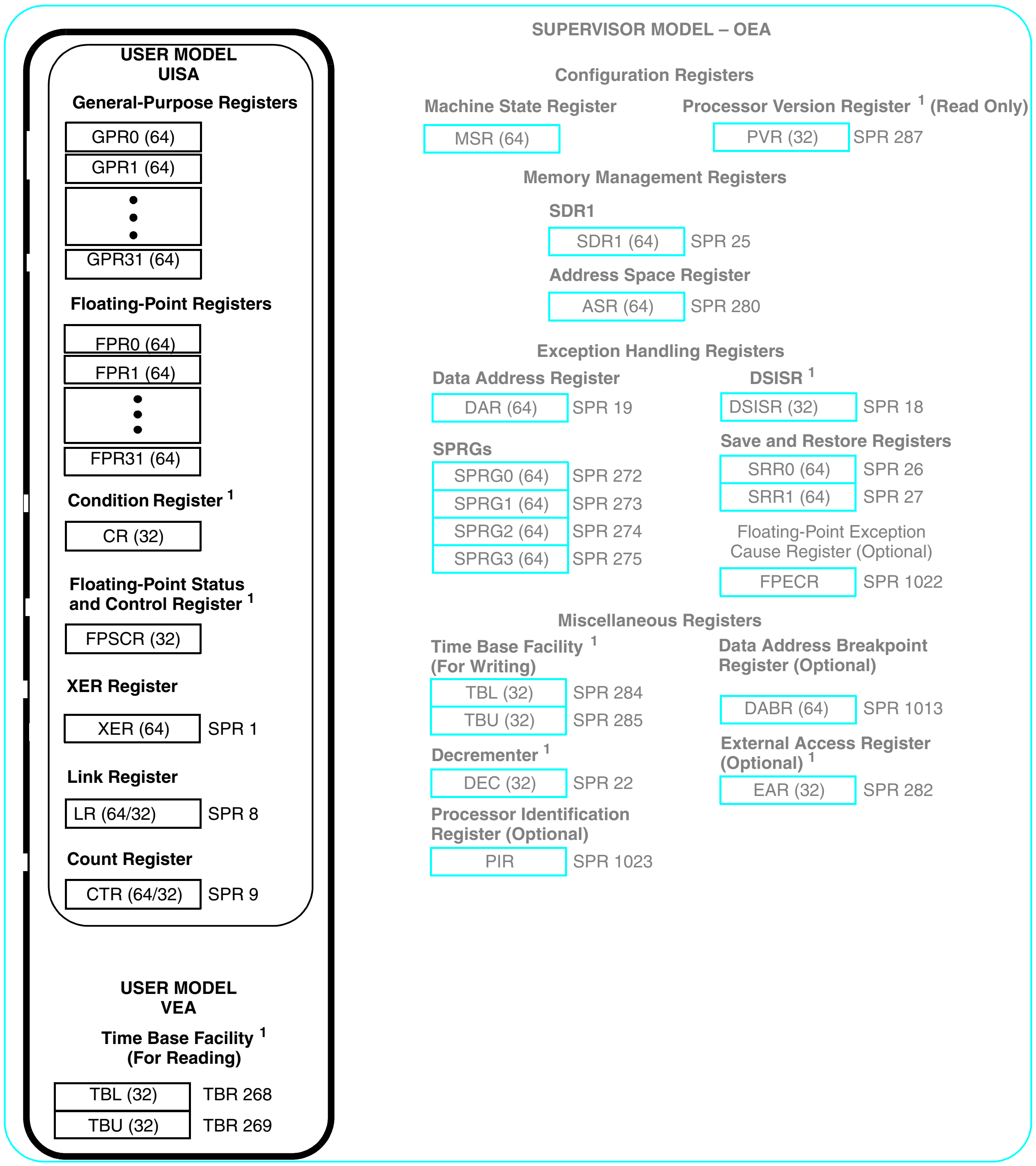
The PowerPC virtual environment architecture (VEA) defines registers in addition to those defined by the UISA. The PowerPC VEA register set can be accessed by all software with either user or supervisor-level privileges. *Figure 2-9* provides a graphic illustration of the PowerPC VEA register set. Note that the following programming model is similar to that found in *Figure 2-1*, with the additional PowerPC VEA registers.

The PowerPC VEA introduces the time base facility (TB), a 64-bit structure that consists of two 32-bit registers—time base upper (TBU) and time base lower (TBL).

Note: The time base registers can be accessed by both user and supervisor-level instructions. In the context of the VEA, user-level applications are permitted read-only access to the TB. The OEA defines supervisor-level access to the TB for writing values to the TB. See *Section 2.3.11 Time Base Facility (TB)—OEA* for more information.

In *Figure 2-9* the numbers to the right of the register name indicates the number that is used in the syntax of the instruction operands to access the register (for example, the number used to access the XER is SPR 1).

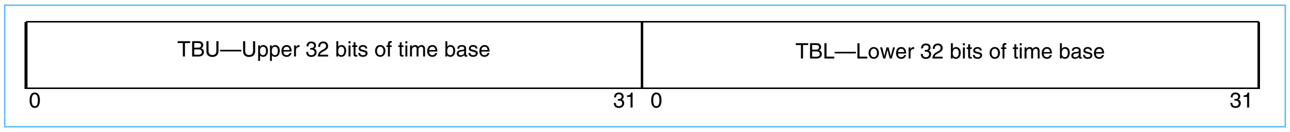
Figure 2-9. VEA Programming Model—User-Level Registers Plus Time Base



1. These registers are 32-bit registers only.
2. These registers are implementation dependent.
3. 64-bit registers operating in 32-bit mode clear the high order 32-bits.

The time base (TB), shown in *Figure 2-10*, is a 64-bit structure that contains a 64-bit unsigned integer that is incremented periodically. Each increment adds '1' to the low-order bit (bit[31] of TBL). The frequency at which the counter is incremented is implementation-dependent.

Figure 2-10. Time Base (TB)


TBU—Upper 32 bits of time base
TBL—Lower 32 bits of time base
0
31
0
31

Note: The TB increments until its value becomes 0xFFFF_FFFF_FFFF_FFFF ($2^{64} – 1$). At the next increment its value becomes 0x0000_0000_0000_0000. There is no exception or explicit indication when this occurs.

The period of the time base depends on the driving frequency. The TB is implemented such that the following requirements are satisfied:

1. Loading a GPR from the time base has no effect on the accuracy of the time base.
2. Storing a GPR to the time base replaces the value in the time base with the value in the GPR.

The PowerPC VEA does not specify a relationship between the frequency at which the time base is updated and other frequencies, such as the processor clock. The TB update frequency is not required to be constant; however, for the system software to maintain time of day and operate interval timers, one of two things is required:

- The system provides an implementation-dependent exception to software whenever the update frequency of the time base changes and a means to determine the current update frequency; or
- The system software controls the update frequency of the time base.

Note: If the operating system initializes the TB to some reasonable value and the update frequency of the TB is constant, the TB can be used as a source of values that increase at a constant rate, such as for time stamps in trace entries.

Even if the update frequency is not constant, values read from the TB are monotonically increasing (except when the TB wraps from $2^{64} – 1$ to 0). If a trace entry is recorded each time the update frequency changes, the sequence of TB values can be postprocessed to become actual time values.

However, successive readings of the time base may return identical values due to implementation-dependent factors such as a low update frequency or initialization.

2.2.1 Reading the Time Base

The **mftb** instruction is used to read the time base. The following sections discuss reading the time base in 64-bit modes. For specific details on using the **mftb** instruction, see *Chapter 8, Instruction Set*. For information on writing the time base, see *Section 2.3.11.1 Writing to the Time Base*.

2.2.1.1 Reading the Time Base

The contents of the time base may be read into a GPR by **mftb**. To read the contents of the TB into register **r**D, execute the following instruction:

```
        mftb    rD
```

The above example uses the simplified mnemonic (referred to as extended mnemonic in the architecture specification) form of the **mftb** instruction (equivalent to **mftb r**A,**268**). Using this instruction copies the entire time base (TBU || TBL) into **r**A. Reading the time base has no effect on the value it contains or the periodic incrementing of that value.

Note: If the simplified mnemonic form **mftbu r**A (equivalent to **mftb r**A**,269**) is used, the contents of TBU are copied to the low-order 32 bits of **r**A, and the high-order 32 bits of **r**A are cleared (0 || TBU).

2.2.2 Computing Time of Day from the Time Base

Since the update frequency of the time base is system-dependent, the algorithm for converting the current value in the time base to time of day is also implementation-dependent.

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 exception, or else 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-second 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.

A generalized service to compute time of day could take the following as input:

- Time of day at beginning of current epoch
- Time base value at beginning of current epoch
- Time base update frequency
- Time base value for which time of day is desired

For a PowerPC system in which the time base update frequency does not vary, the first three inputs would be constant.

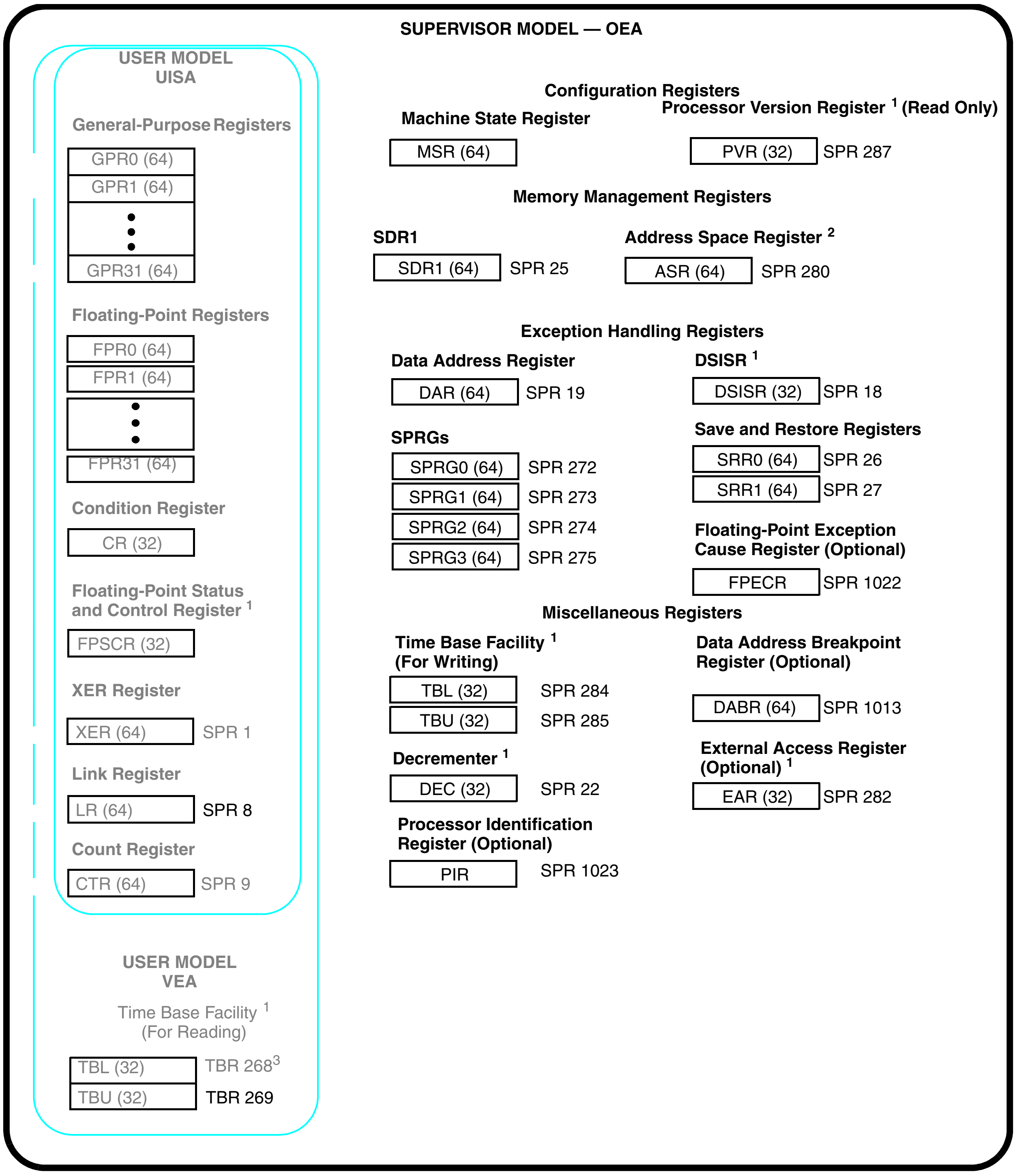
2.3 PowerPC OEA Register Set

The PowerPC operating environment architecture (OEA) completes the discussion of PowerPC registers. *Figure 2-11* shows a graphic representation of the entire PowerPC register set—UISA, VEA, and OEA. In *Figure 2-11* the numbers to the right of the register name indicates the number that is used in the syntax of the instruction operands to access the register (for example, the number used to access the XER is SPR 1).

All of the SPRs in the OEA can be accessed only by supervisor-level instructions; any attempt to access these SPRs with user-level instructions results in a supervisor-level exception. Some SPRs are implementation-specific. In some cases, not all of a register's bits are implemented in hardware.

If a PowerPC processor executes an **mtspr**/**mfspr** instruction with an undefined SPR encoding, it takes (depending on the implementation) an illegal instruction program exception, a privileged instruction program exception, or the results are boundedly undefined. See *Section 6.4.9 Program Exception (0x00700)* for more information.

Figure 2-11. OEA Programming Model—All Registers



1. These registers are 32-bit registers only.
2. These registers are on 64-bit implementations only.
3. TBR268 is read as a 64-bit value

The PowerPC OEA supervisor-level registers are:

- Configuration registers which include:
  - Machine state register (MSR). The MSR defines the state of the processor. The MSR can be modified by the Move to Machine State Register (**mtmsrd** [or **mtmsr**]), System Call (**sc**), and Return from Interrupt Doubleword (**rfid**) instructions. It can be read by the Move from Machine State Register (**mfmsr**) instruction. For more information, see *Section 2.3.1 Machine State Register (MSR)*.
  - Processor version register (PVR). The PVR is a read-only register that identifies the version (model) and revision level of the PowerPC processor. For more information, see *Section 2.3.2 Processor Version Register (PVR)*.
- Memory management registers which include:
  - SDR1. The SDR1 register specifies the page table base address used in virtual-to-physical address translation. For more information, see *Section 2.3.3 SDR1*. (Note that physical address is referred to as real address in the architecture specification.)
  - Address space register (ASR). The ASR holds the physical address of the segment table. It is found only on 64-bit implementations. For more information, see *Section 2.3.4 Address Space Register (ASR)*.
- Exception handling registers which include:
  - Data address register (DAR). A data address register (DAR) is set to the effective address generated by the a DSI or an alignment exception. For more information, see *Section 2.3.5 Data Address Register (DAR)*.
  - SPRG0–SPRG3. The SPRG0–SPRG3 registers are provided for operating system use. For more information, see *Section 2.3.6 Software Use SPRs (SPRG0–SPRG3)*.
  - DSISR. The DSISR defines the cause of DSI and alignment exceptions. For more information, refer to *Section 2.3.7 Data Storage Interrupt Status Register (DSISR)*.
  - Machine status save/restore register 0 (SRR0). The SRR0 register is used to save machine status on exceptions and to restore machine status when an **rfid** instruction is executed. For more information, see *Section 2.3.8 Machine Status Save/Restore Register 0 (SRR0)*.
  - Machine status save/restore register 1 (SRR1). The SRR1 register is used to save machine status on exceptions and to restore machine status when an **rfid** instruction is executed. For more information, see *Section 2.3.9 Machine Status Save/Restore Register 1 (SRR1)*.
  - Floating-point exception cause register (FPECR). This optional register is used to identify the cause of a floating-point exception.

- Miscellaneous registers which include:
  - Time base (TB). The TB is a 64-bit structure that maintains the time of day and operates interval timers. The TB consists of two 32-bit registers—time base upper (TBU) and time base lower (TBL). Note that the time base registers can be accessed by both user and supervisor-level instructions. For more information, see *Section 2.3.11 Time Base Facility (TB)—OEA* and *Section 2.2 PowerPC VEA Register Set—Time Base*."
  - Decrementer register (DEC). The DEC register is a 32-bit decrementing counter that provides a mechanism for causing a decrementer exception after a programmable delay; the frequency is a subdivision of the processor clock. For more information, see *Section 2.3.12 Decrementer Register (DEC)*.
  - External access register (EAR). This optional register is used in conjunction with the **eciwx** and **ecowx** instructions. Note that the EAR register and the **eciwx** and **ecowx** instructions are optional in the PowerPC Architecture and may not be supported in all PowerPC processors that implement the OEA. For more information about the external control facility, see *Section 4.3.4 External Control Instructions*.
  - Data address breakpoint register (DABR). This optional register is used to control the data address breakpoint facility. Note that the DABR is optional in the PowerPC Architecture and may not be supported in all PowerPC processors that implement the OEA. For more information about the data address breakpoint facility, see *Section 6.4.3 DSI Exception (0x00300)*.
  - Processor identification register (PIR). This optional register is used to hold a value that distinguishes an individual processor in a multiprocessor environment.

2.3.1 Machine State Register (MSR)

The machine state register (MSR) is a 64-bit register (see *Figure 2-12*) and defines the state of the processor. When an exception occurs, the contents of the MSR register are saved in SRR1. A new set of bits are loaded into the MSR as determined by the exception. The MSR can also be modified by the **mtmsrd** (or **mtmsr**), **sc**, and **rfid** instructions. It can be read by the **mfmsr** instruction.

Figure 2-12. Machine State Register (MSR)

▨ Reserved

SF	000 0000 ... 0000 0	POW	0	ILE	EE	PR	FP	ME	FE0	SE	BE	FE1	0 0	IR	DR	0	PMM	RI	LE
0	1 44	45	46	47	48	49	50	51	52	53	54	55	56 57	58	59	60	61	62	63

Table 2-7 shows the bit definitions for the MSR.

Table 2-7. MSR Bit Settings

Bit(s)	Name	Description
0	SF	Sixty-four bit mode 0 The 64-bit processor runs in 32-bit mode. 1 The 64-bit processor runs in 64-bit mode. Note that this is the default setting.
1	—	Reserved
2	ISF	Exception 64-bit mode (optional). When an exception occurs, this bit is copied into MSR[SF] to select 64 or 32-bit mode for the context established by the exception. **Note:** If the temporary bridge function is not implemented, this bit is treated as reserved.
3–44	—	Reserved
45	POW	Power management enable 0 Power management disabled (normal operation mode) 1 Power management enabled (reduced power mode) **Note:** Power management functions are implementation-dependent. If the function is not implemented, this bit is treated as reserved.
46	—	Reserved
47	ILE	This is part of the optional little-endian facility. If the little-endian facility is implemented, and an exception occurs, this bit is copied into MSR[LE] to select the endian mode for the context established by the exception.
48	EE	External interrupt enable 0 While the bit is cleared, the processor delays recognition of external interrupts and decrementer exception conditions. 1 The processor is enabled to take an external interrupt or the decrementer exception.
49	PR	Privilege level 0 The processor can execute both user and supervisor-level instructions. 1 The processor can only execute user-level instructions. **Note:** Any instruction or event that set MSR[PR] also sets MSR[EE], MSR[IR], and MSR[DR].
50	FP	Floating-point available 0 The processor prevents dispatch of floating-point instructions, including floating-point loads, stores, and moves. 1 The processor can execute floating-point instructions.
51	ME	Machine check enable 0 Machine check exceptions are disabled. 1 Machine check exceptions are enabled. **Note:** The only instruction that can alter MSR[ME] is the **rfid** instruction.
52	FE0	Floating-point exception mode 0 (see *Table 2-8*).
53	SE	Single-step trace enable (Optional) 0 The processor executes instructions normally. 1 The processor generates a single-step trace exception upon the successful execution of the next instruction (unless that instruction is **rfid**, which is never trace). Successful completion means that the instruction caused no other interrupt. **Note:** If the function is not implemented, this bit is treated as reserved.
54	BE	Branch trace enable (Optional) 0 The processor executes branch instructions normally. 1 The processor generates a branch trace exception after completing the execution of a branch instruction, regardless of whether the branch was taken. **Note:** If the function is not implemented, this bit is treated as reserved.
55	FE1	Floating-point exception mode 1 (See *Table 2-8*).

Table 2-7. MSR Bit Settings (Continued)

Bit(s)	Name	Description
56	—	Reserved
57	—	Reserved
58	IR	Instruction address translation 0 Instruction address translation is disabled. 1 Instruction address translation is enabled. For more information, see *Chapter 7, Memory Management*.
59	DR	Data address translation 0 Data address translation is disabled. 1 Data address translation is enabled. For more information, see *Chapter 7, Memory Management*.
60	—	Reserved
61	PMM	Performance monitor mark. This bit is part of the optional performance monitor facility. If the performance monitor facility is not implemented or does not use this bit, then this bit is treated as reserved.
62	RI	Recoverable exception (for system reset and machine check exceptions). 0 Exception is not recoverable. 1 Exception is recoverable. For more information, see *Chapter 6, Exceptions*.
63	LE	This is part of the optional little-endian facility. If the little-endian facility is implemented, then the bit has the following meaning: 0 The processor runs in big-endian mode. 1 The processor runs in little-endian mode. If the little-endian facility is not implemented or does not use this bit, then this bit is treated as reserved.

The floating-point exception mode bits [FE0–FE1] are interpreted as shown in *Table 2-8*.

Table 2-8. Floating-Point Exception Mode Bits

FE0	FE1	Mode
0	0	Floating-point exceptions disabled
0	1	Floating-point imprecise nonrecoverable
1	0	Floating-point imprecise recoverable
1	1	Floating-point precise mode

Table 2-9 indicates the initial state of the MSR at power up.

Table 2-9. State of MSR at Power Up

Bit	Name	Default Value
0	SF	1
1	—	Unspecified[1]
2 (Temporary 64-Bit Bridge)	ISF	Unspecified[1]
3–44	—	Unspecified[1]
45	POW	0
46	—	Unspecified[1]
47	ILE	0
48	EE	0
49	PR	0
50	FP	0
51	ME	0
52	FE0	0
53	SE	0
54	BE	0
55	FE1	0
56	—	Unspecified[1]
57	—	Unspecified[1]
58	IR	0
59	DR	0
60	—	Unspecified[1]
61	PMM	Unspecified[1]
62	RI	0
63	LE	0

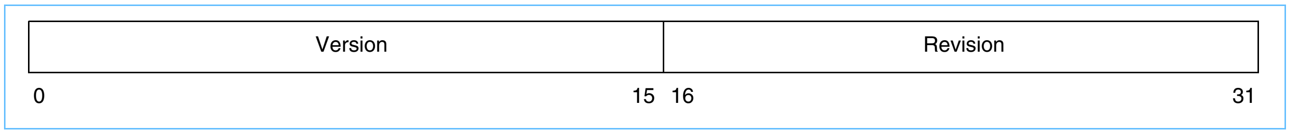
Note:

1. Unspecified can be either ‘0’ or ’1’
2. 1 is typical, but might be ‘0’

2.3.2 Processor Version Register (PVR)

The processor version register (PVR) is a 32-bit, read-only register which contains a value identifying the specific version (model) and revision level of the PowerPC processor (see *Figure 2-13*). The contents of the PVR can be copied to a GPR by the **mfspr** instruction. Read access to the PVR is supervisor-level only; write access is not provided.

Figure 2-13. Processor Version Register (PVR)



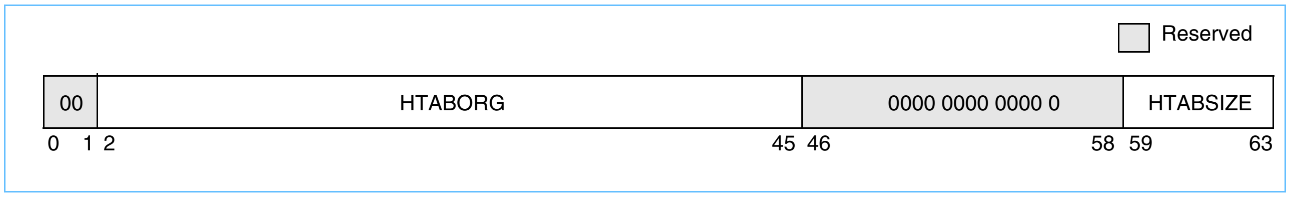
The PVR distinguishes between processors that differ in attributes that might affect software. It contains two 16-bit fields:

- Version (bits [0–15])—A 16-bit number that uniquely identifies a particular processor version. This number can be used to determine the version of a processor; it might not distinguish between different end product models if more than one model uses the same processor.
- Revision (bits [16–31])—A 16-bit number that distinguishes between various releases of a particular version (that is, an engineering change level). The value of the revision portion of the PVR is implementation-specific. The processor revision level is changed for each revision of the device.

2.3.3 SDR1

The SDR1 is a 64-bit register that is shown in *Figure 2-14*.

Figure 2-14. SDR1



The SDR1 bits are described in *Table 2-10*.

Table 2-10. SDR1 Bit Settings

Bits	Name	Description
0–1	—	Reserved
2-45	HTABORG	Physical base address of page table
46–58	—	Reserved
59–63	HTABSIZE	Encoded size of page table (used to generate mask)

The HTABORG field in SDR1 contains the high-order 46 bits of the 64-bit physical address of the page table. Therefore, the page table is constrained to lie on a 2^{18}-byte (256 Kbytes) boundary at a minimum. At least 11 bits from the hash function are used to index into the page table. The page table must consist of at least 256 Kbytes (2^{11} PTEGs of 128 bytes each).

The page table can be any size 2^n where $18 \leq n \leq 46$. As the table size is increased, more bits are used from the hash to index into the table and the value in HTABORG must have more of its low-order bits equal to 0. The HTABSIZE field in SDR1 contains an integer value that determines how many bits from the hash are used in the page table index. This number must not exceed 28. HTABSIZE is used to generate a mask of the form 0b00...011...1; that is, a string of 0 bits followed by a string of 1-bits. The 1-bits determine how many additional bits (beyond the minimum of 11) from the hash are used in the index. The HTABORG must have this same number of low-order bits equal to 0. See *Figure 7-17 Example Primary PTEG Address Generation* for an example of the primary PTEG address generation in a 64-bit implementation.

Example:
Suppose that the page table is 16,384 (2^{14}), 128-byte PTEGs, for a total size of 2^{21} bytes (2 Mbytes). Note that a 14-bit index is required. Eleven bits are provided from the hash initially, so three additional bits from the hash must be selected. The value in HTABSIZE must be 3 and the value in HTABORG must have its low-order three bits (bits [43-45] of SDR1) equal to 0. This means that the page table must begin on a $2^{3 + 11 + 7} = 2^{21}$ = 2 Mbytes boundary.

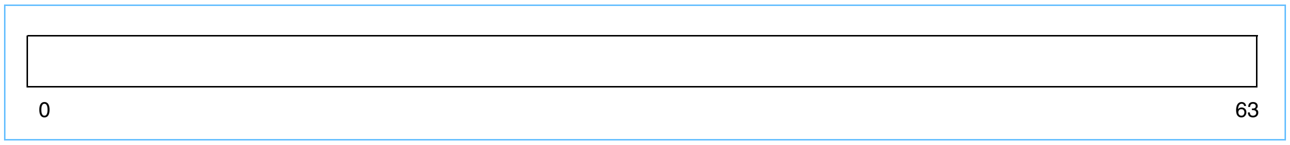
On implementations that support a virtual address size of only 64 bits, software should set the HTABSIZE field to a value that does not exceed 25. Because the high-order 16 bits of the VSID must be zeros for these implementations, the hash value used in the page table search will have the high-order three bits either all zeros (primary hash) or all ones (secondary hash). If HTABSIZE > 25, some of these hash value bits will be used to index into the page table, resulting in certain PTEGs never being searched.

For more information, refer to *Chapter 7, Memory Management*.

2.3.4 Address Space Register (ASR)

The ASR is a 64-bit special purpose register provided for operating system use and can be used to point to a segment register. On earlier PowerPC implementations and on 64-bit PowerPC implementations, bits[0-51] of the ASR contained the high-order 52 bits of the 64-bit real address of the segment table, and bit[63] of the ASR indicated whether the specified segment table should (bit[63] = ‘1’) or should not (bit[63] = ‘0’) be searched by the processor when doing address translation.

Figure 2-15. Address Space Register (ASR)



The bits of the ASR are described in *Table 2-11*.

Table 2-11. ASR Bit Settings

Bits	Name	Description
0–63	–	Reserved

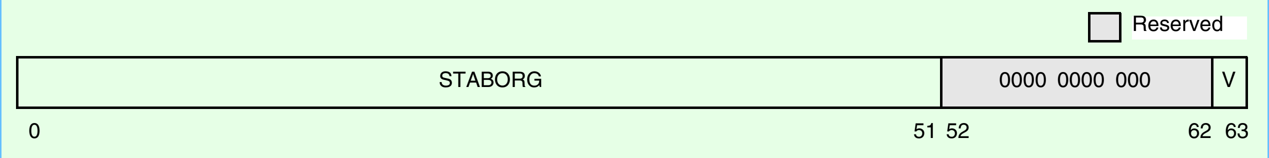
Temporary 64-Bit Bridge

Some 64-bit processors implement optional features that simplify the conversion of an operating system from the 32-bit to the 64-bit portion of the architecture. This architecturally-defined bridge allows the option of defining bit[63] as ASR[V], the STABORG field valid bit.

If the ASR[V] bit is implemented and is set, the ASR[STABORG] field is valid and functions are as described for the 64-bit architecture. However, if the ASR[V] bit is implemented and ASR[V] and MSR[SF] are cleared, an operating system can use 16 SLB entries similarly to the way 32-bit implementations use the segment registers, which are otherwise not supported in the 64-bit architecture. Note that if ASR[V] = 0, a reference to a nonexistent address in the STABORG field does not cause a machine check exception.

The ASR, with the optional V bit implemented, is shown in *Figure 2-16*.

Figure 2-16. Address Space Register (ASR)—64-Bit Bridge



The bits of the ASR, including the optional V bit, are described in *Table 2-12*.

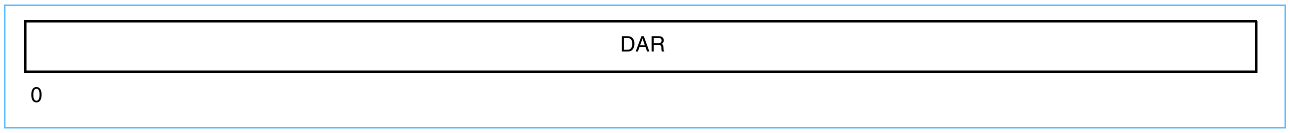
Table 2-12. ASR Bit Settings—64-Bit Bridge

Bits	Name	Description
0–51	STABORG	Physical address of segment table
52–62	—	Reserved
63	V	STABORG field valid (V = ’1’) or invalid (V = 0). Note that the [V] bit of the ASR is optional. If the function is not implemented, this bit is treated as reserved, except that it is assumed to be set for address translation.

2.3.5 Data Address Register (DAR)

The DAR is a 64-bit register and is shown in *Figure 2-17*.

Figure 2-17. Data Address Register (DAR)



The effective address generated by a memory access instruction is placed in the DAR if the access causes an exception (for example, an alignment exception). If the exception occurs in a 64-bit implementation operating in 32-bit mode, the high-order 32 bits of the DAR are cleared. For information, see *Chapter 6, Exceptions*.

2.3.6 Software Use SPRs (SPRG0–SPRG3)

SPRG0–SPRG3 are 64-bit registers which are provided for general operating system use, such as performing a fast state save or for supporting multiprocessor implementations. The formats of SPRG0–SPRG3 are shown in *Figure 2-18*.

Figure 2-18. SPRG0–SPRG3

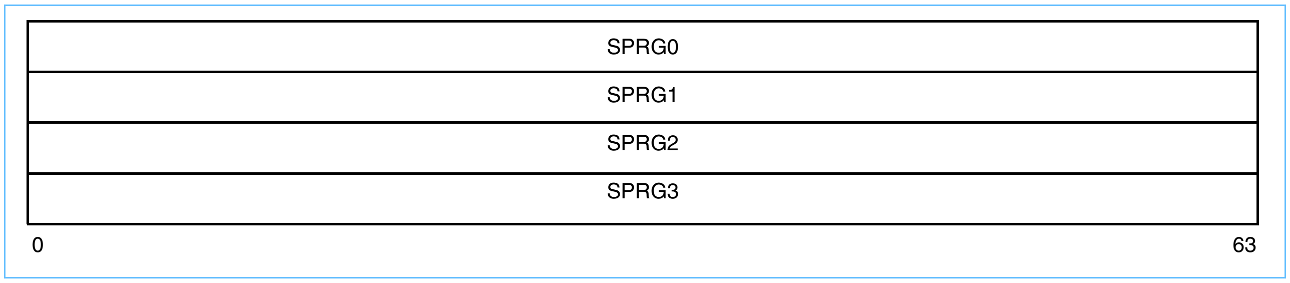


Table 2-13 provides a description of conventional uses of SPRG0 through SPRG3.

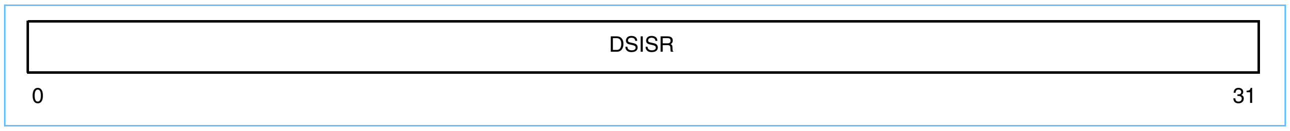
Table 2-13. Conventional Uses of SPRG0–SPRG3

Register	Description
SPRG0	Software may load a unique physical address in this register to identify an area of memory reserved for use by the first-level exception handler. This area must be unique for each processor in the system.
SPRG1	This register may be used as a scratch register by the first-level exception handler to save the content of a GPR. That GPR then can be loaded from SPRG0 and used as a base register to save other GPRs to memory.
SPRG2	This register may be used by the operating system as needed.
SPRG3	This register may be used by the operating system as needed. It is optional whether SPRG3 can be read in user mode. On implementations that provide this ability, SPRG3 may be used for information, such as a “thread-id”, that the operating system makes available to application programs. On implementations for which SPRG3 can be read in user mode, operating systems must ensure that no sensitive data are left in SPRG3 when a user mode program is dispatched, and operating systems for secure systems must ensure that SPRG3 cannot be used to implement a “covert channel” between user mode programs. These requirements can be satisfied by clearing SPRG3 before passing control to a program that will run in user mode. On such implementations, SPRG3 can be used “orthogonally” for both the purpose described for it above and the purpose described for SPRG1. If this is done, SPRG1 can be used for some other purpose.

2.3.7 Data Storage Interrupt Status Register (DSISR)

The 32-bit data storage interrupt status register (DSISR), shown in *Figure 2-19*, identifies the cause of the DSI, machine check, data segment, and alignment exceptions.

Figure 2-19. DSISR



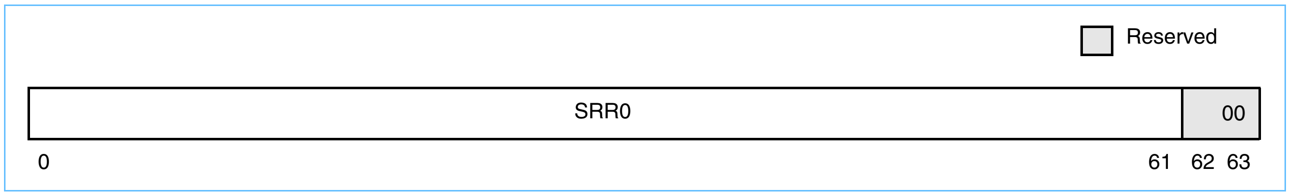
DSISR bits may be treated as reserved in a given implementation if they are specified as being set either to 0 or to an undefined value for all interrupts that set the DSISR (including implementation-dependent setting, for example, by the Machine Check interrupt or by implementation-specific interrupts).

For information about bit settings, see *Section 6.4.3 DSI Exception (0x00300)* and *Section 6.4.8 Alignment Exception (0x00600)*.

2.3.8 Machine Status Save/Restore Register 0 (SRR0)

The SRR0 is a 64-bit register that is used to save the effective address on exceptions (interrupts) and return to the interrupted program when an **rfid** instruction is executed. It also holds the EA for the instruction that follows the System Call (**sc**) instruction. The format of SRR0 is shown in *Figure 2-20*.

Figure 2-20. Machine Status Save/Restore Register 0 (SRR0)



When an exception occurs, SRR0 is set to point to an instruction such that all prior instructions have completed execution and no subsequent instruction has begun execution. In the case of an error exception the SRR0 register is pointing at the instruction that caused the error. When an **rfid** instruction is executed, the contents of SRR0 are copied to the next instruction address (NIA)—the 64 or 32-bit address of the next instruction to be executed. The instruction addressed by SRR0 may not have completed execution, depending on the exception type. SRR0 addresses either the instruction causing the exception or the immediately following instruction. The instruction addressed can be determined from the exception type and status bits.

If the exception occurs in 32-bit mode of a 64-bit implementation, the high-order 32 bits of the NIA are cleared, NIA[32–61] are set from SRR0[32–61], and the two least significant bits of NIA are cleared.

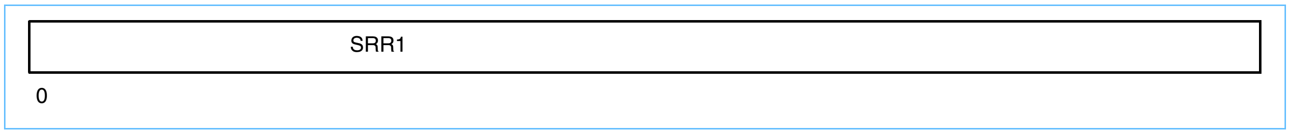
Note: In some implementations, every instruction fetch performed while MSR[IR] = ’1’ , and every instruction execution requiring address translation when MSR[DR] = ’1’ , may modify SRR0.

For information on how specific exceptions affect SRR0, refer to the descriptions of individual exceptions in *Chapter 6, Exceptions*.

2.3.9 Machine Status Save/Restore Register 1 (SRR1)

The SRR1 is a 64-bit which is used to save exception status and the machine status register when an **rfid** instruction is executed. The format of SRR1 is shown in *Figure 2-21*.

Figure 2-21. Machine Status Save/Restore Register 1 (SRR1)



When an exception occurs, bits [33–36] and [42–47] of SRR1 are loaded with exception-specific information and bits. The remaining bits of SRR1 are defined as reserved. An implementation may define one or more of these bits, and in this case, may also cause them to be saved from MSR on an exception and restored to MSR from SRR1 on a **rfid**.

Note: In some implementations, every instruction fetch when MSR[IR] = ’1’ , and every instruction execution requiring address translation when MSR[DR] = ’1’ , may modify SRR1.

For information on how specific exceptions affect SRR1, refer to the individual exceptions in *Chapter 6, “Exceptions*.”

2.3.10 Floating-Point Exception Cause Register (FPECR)

The FPECR register may be used to identify the cause of a floating-point exception.

Note: The FPECR is an optional register in the PowerPC Architecture and may be implemented differently (or not at all) in the design of each processor. The user’s manual of a specific processor will describe the functionality of the FPECR, if it is implemented in that processor.

2.3.11 Time Base Facility (TB)—OEA

As described in *Section 2.2 PowerPC VEA Register Set—Time Base*, the time base (TB) provides a long-period counter driven by an implementation-dependent frequency. The VEA defines user-level read-only access to the TB. Writing to the TB is reserved for supervisor-level applications such as operating systems and boot-strap routines. The OEA defines supervisor-level, write access to the TB.

The TB is a volatile resource and must be initialized during reset. Some implementations may initialize the TB with a known value; however, there is no guarantee of automatic initialization of the TB when the processor is reset. The TB runs continuously after start-up.

For more information on the user-level aspects of the time base, refer to *Section 2.2 PowerPC VEA Register Set—Time Base* on page 53.

2.3.11.1 Writing to the Time Base

Note: Writing to the TB is reserved for supervisor-level software.

The simplified mnemonics, **mttbl** and **mttbu**, write the lower and upper halves of the TB, respectively. The simplified mnemonics listed above are for the **mtspr** instruction; see *Appendix E Simplified Mnemonics* for more information. The **mtspr**, **mttbl**, and **mttbu** instructions treat TBL and TBU as separate 32-bit registers; setting one leaves the other unchanged. It is not possible to write the entire 64-bit time base in a single instruction.

The instructions for writing the time base are not dependent on the implementation or mode. Thus, code written to set the TB on a 32-bit implementation will work correctly on a 64-bit implementation running in either 32 or 64-bit mode.

The TB can be written by a sequence such as:

```
lwz     rx,upper               #load 64-bit value for
lwz     ry,lower               # TB into rx and ry
li      rz,0
mttbl   rz                     #force TBL to 0
mttbu   rx                     #set TBU
mttbl   ry                     #set TBL
```

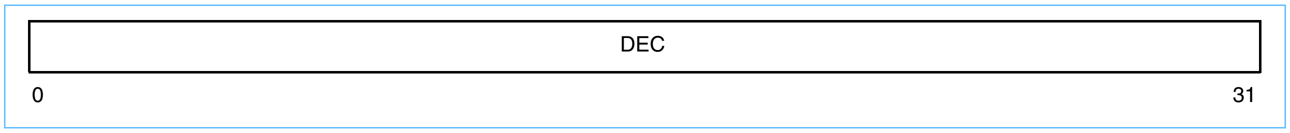
Provided that no exceptions occur while the last three instructions are being executed, loading 0 into TBL prevents the possibility of a carry from TBL to TBU while the time base is being initialized.

For information on reading the time base, refer to *Section 2.2.1 Reading the Time Base* on page 56.

2.3.12 Decrementer Register (DEC)

The decrementer register (DEC), shown in *Figure 2-22*, is a 32-bit decrementing counter that provides a mechanism for causing a decrementer exception after a programmable delay. The DEC frequency is based on the same implementation-dependent frequency that drives the time base.

Figure 2-22. Decrementer Register (DEC)



2.3.12.1 Decrementer Operation

The DEC counts down, causing an exception (unless masked by MSR[EE]) when it passes through zero. The DEC satisfies the following requirements:

- The operation of the time base and the DEC are coherent (that is, the counters are driven by the same fundamental time base).
- Loading a GPR from the DEC has no effect on the DEC.
- Storing the contents of a GPR to the DEC replaces the value in the DEC with the value in the GPR.
- Whenever bit[0] of the DEC changes from 0 to 1, a decrementer exception request is signaled. Multiple DEC exception requests may be received before the first exception occurs; however, any additional requests are canceled when the exception occurs for the first request.
- If the DEC is altered by software and the content of bit [0] is changed from 0 to 1, an exception request is signaled.

Note: In systems that change the Time Base update frequency for purposes such as power management, the Decrementer input frequency will also change. Software must be aware of this in order to set interval timers.

2.3.12.2 Writing and Reading the DEC

The content of the DEC can be read or written using the **mfspr** and **mtspr** instructions, both of which are supervisor-level when they refer to the DEC. Using a simplified mnemonic for the **mtspr** instruction, the DEC may be written from GPR **r**A with the following:

```
mtdec   rA
```

Using a simplified mnemonic for the **mfspr** instruction, the DEC may be read into GPR **r**A with the following:

```
mfdec   rA
```

2.3.12.3 Data Address Compare

The Data Address Compare mechanism provides a means of detecting load and store accesses to a virtual page. The Data Address Compare mechanism is controlled by the Address Compare Control Register (ACCR), and by a bit in each Page Table Entry (PTE[AC]).

Note: The Data Address Compare mechanism does not apply to instruction fetches, or to data accesses in real addressing mode (MSR[DR] = 0).

2.3.13 Data Address Breakpoint Register (DABR)

The optional data address breakpoint facility is controlled by an optional SPR, the DABR. The DABR is a 64-bit register. However, if the data address breakpoint facility is implemented, it is recommended, but not required, that it be implemented as described in this section.

The data address breakpoint facility provides a means to detect accesses to a designated doubleword. The address comparison is done on an effective address, and is done independent of whether address translation is enabled or disabled. The data address breakpoint mechanism applies to data accesses only. It does not apply to instruction fetches.

The DABR is shown in *Figure 2-23*.

Figure 2-23. Data Address Breakpoint Register (DABR)

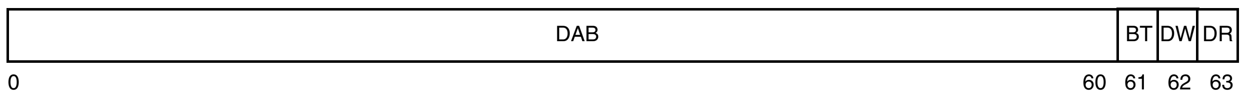


Table 2-14 describes the fields in the DABR.

Table 2-14. DABR—Bit Settings

Bits	Name	Description
64 Bit		
0–60	DAB	Data address breakpoint
61	BT	Breakpoint translation enable
62	DW	Data write enable
63	DR	Data read enable

A data address breakpoint match is detected for a load or store instruction if the three following conditions are met for any byte accessed:

- EA[0–60] = DABR[DAB]
- MSR[DR] = DABR[BT]
- Instruction is a store and DABR[DW] = ’1’ , or the instruction is a load and DABR[DR] = ’1’ .

Note: In 32-bit mode the high-order 32 bits of the effective address are treated as zeros for the purpose of detecting a match.

If the above conditions are satisfied, a match also occurs for **eciwx** and **ecowx**. For the purpose of determining whether a match occurs, **eciwx** is treated as a load, and **ecowx** is treated as a store.

Even if the above conditions are satisfied, it is undefined whether a match occurs in the following cases:

- A store string instruction (**stwcx.** or **stdcx.**) in which the store is not performed
- A load or store string instruction (**lswx** or **stswx**) with a zero length
- A **dcbz** instruction. For the purpose of determining whether a match occurs, **dcbz** is treated as a store.

The cache management instructions other than **dcbz** never cause a match. If **dcbz** causes a match, some or all of the target memory locations may have been updated.

A match generates a DSI exception. Refer to *Section 6.4.3 DSI Exception (0x00300)* for more information on the data address breakpoint facility.

If a match occurs, some or all of the bytes of the memory operand may have been accessed; however, if a store or **ecowx** instruction causes the match, the memory operand is not altered if the instruction is one of the following:

- any store instruction that causes an atomic access
- **ecowx**

Note: The data address breakpoint mechanism does not apply to instruction fetches. If a data address breakpoint match occurs for a load instruction for which any byte of the memory operand is in memory that is both caching inhibited and guarded, or for an **eciwx** instruction, it may not be safe for software to restart the instruction.

2.3.14 External Access Register (EAR)

The external access register (EAR) is an optional 32-bit SPR that controls access to the external control facility and identifies the target device for external control operations. The external control facility provides a means for user-level instructions to communicate with special external devices. The EAR is shown in *Figure 2-24*.

Figure 2-24. External Access Register (EAR)

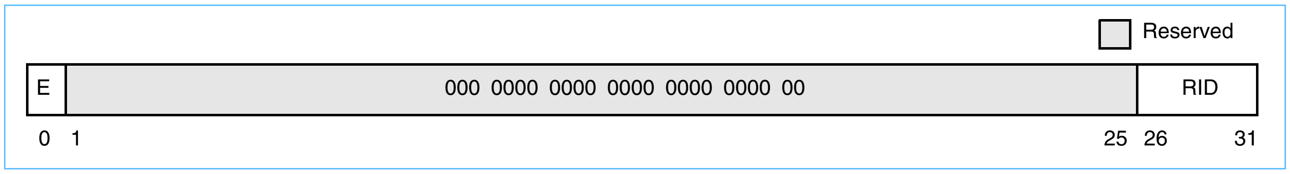


Table 2-15 describes the fields in the external access register.

Table 2-15. External Access Register (EAR)—Bit Settings

Bits	Name	Description
0	E	Enable bit 1 Enabled 0 Disabled If this bit is set, the **eciwx** and **ecowx** instructions can perform the specified external operation. If the bit is cleared, an **eciwx** or **ecowx** instruction causes a DSI exception.
1-25	–	Reserved
26-31	RID	Resource id

The high-order bits of the resource ID (RID) field beyond the width of the RID supported by a particular implementation are treated as reserved bits.

The EAR register is provided to support the External Control In Word Indexed (**eciwx**) and External Control Out Word Indexed (**ecowx**) instructions, which are described in *Chapter 8, Instruction Set*. Although access to the EAR is supervisor-level, the operating system can determine which tasks are allowed to issue external access instructions and when they are allowed to do so. The bit settings for the EAR are described in *Table 2-15*. Interpretation of the physical address transmitted by the **eciwx** and **ecowx** instructions and the 32-bit value transmitted by the **ecowx** instruction is not prescribed by the PowerPC OEA, but is determined by the target device. The data access of **eciwx** and **ecowx** is performed as though the memory access mode bits (WIMG) were 0101.

For example, if the external control facility is used to support a graphics adapter, the **ecowx** instruction could be used to send the translated physical address of a buffer containing graphics data to the graphics device. The **eciwx** instruction could be used to load status information from the graphics adapter.

This register can also be accessed by using the **mtspr** and **mfspr** instructions. Synchronization requirements for the EAR are shown in *Table 2-16 Data Access Synchronization* and *Table 2-17 Instruction Access Synchronization*.

2.3.15 Processor Identification Register (PIR)

The PIR register is used to differentiate between individual processors in a multiprocessor environment.

Note: The PIR is an optional register in the PowerPC Architecture and may be implemented differently (or not at all) in the design of each processor. The user's manual of a specific processor will describe the functionality of the PIR, if it is implemented in that processor.

2.3.16 Synchronization Requirements for Special Registers and for Lookaside Buffers

Changing the value in certain system registers, and invalidating SLB and TLB entries, can cause alteration of the context in which data addresses and instruction addresses are interpreted, and in which instructions are executed. 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. The context synchronization required for context-altering instructions is shown in *Table 2-16* for data access and *Table 2-17* for instruction fetch and execution.

A context-synchronizing exception (that is, any exception except nonrecoverable system reset or nonrecoverable machine check) can be used instead of a context-synchronizing instruction. In the tables, 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.

A 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. A synchronizing instruction after the context-altering instruction ensures that all instructions after that synchronizing instruction are fetched and executed in the context established by the alteration. Instructions after the first synchronizing instruction, up to and including the second synchronizing instruction, may be fetched or executed in either context.

If a sequence of instructions contains context-altering instructions and contains no instructions that are affected by any of the context alterations, no software synchronization is required within the sequence.

Note: Some instructions that occur naturally in the program, such as the **rfid** at the end of an exception handler, provide the required synchronization.

No software synchronization is required before altering the MSR (except when altering the MSR[POW] or MSR[LE] bits; see *Table 2-16* and *Table 2-17*), because **mtmsrd** (or **mtmsr**) is execution synchronizing. No software synchronization is required before most of the other alterations shown in *Table 2-17*, 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 2-16. Data Access Synchronization

Instruction/Event	Required Prior	Required After	Notes
Exception	None	None	
rfid	None	None	
sc	None	None	
Trap	None	None	
mtmsrd (SF)	None	None	3
mtmsrd (or **mtmsr**) (ILE)	None	None	3
mtmsrd (or **mtmsr**) (PR)	None	None	3
mtmsrd (or **mtmsr**) (DR)	None	None	3
mtmsrd (or **mtmsr**) (LE)	—	—	1, 3
mtsr [or **mtsrin**]	Context-synchronizing instruction	Context-synchronizing instruction	
mtspr (ACCR)	Context-synchronizing instruction	Context-synchronizing instruction	
mtspr (SDR1)	**ptesync**	Context-synchronizing instruction	5, 6
mtspr (DABR)	—	—	4
mtspr (EAR)	Context-synchronizing instruction	Context-synchronizing instruction	
slbie	Context-synchronizing instruction	Context-synchronizing instruction	
slbia	Context-synchronizing instruction	Context-synchronizing instruction	
slbmte	Context-synchronizing instruction	Context-synchronizing instruction	13
tlbie	Context-synchronizing instruction	Context-synchronizing instruction	7, 9
tlbiel	Context-synchronizing instruction	**ptesync**	7, 9
tlbia	Context-synchronizing instruction	Context-synchronizing instruction	7
Store (PTE)	none	{**ptesync**, CSI}	8, 9

Note: Refer to *Section 2.3.16.1* on page 77 for explanation of notes.

For information on instruction access synchronization requirements, see *Table 2-17*.

Table 2-17. Instruction Access Synchronization

Instruction/Event	Required Prior	Required After	Notes
Exception	None	None	
rfid	None	None	
sc	None	None	
Trap	None	None	
mtmsrd (SF)	None	None	3, 10
mtmsrd (or **mtmsr**) (ILE)	None	None	3
mtmsrd (or **mtmsr**) (EE)	None	None	2, 3
mtmsrd (or **mtmsr**) (PR)	None	None	3, 11
mtmsrd (or **mtmsr**) (FP)	None	None	3
mtmsrd (or **mtmsr**) (FE0, FE1)	None	None	3
mtmsrd (or **mtmsr**) (SE, BE)	None	None	3
mtmsrd (or **mtmsr**) (IR)	None	None	3, 11
mtmsrd (or **mtmsr**) (RI)	None	None	3
mtmsrd (or **mtmsr**) (LE)	—	—	1, 3
mtsr [or **mtsrin**]	None	Context-synchronizing instruction	11
mtspr (SDR1)	**ptesync**	Context-synchronizing instruction	5, 6
mtspr (DEC)	None	None	12
mtspr (CTRL)	None	None	
slbie	None	Context-synchronizing instruction	
slbia	None	Context-synchronizing instruction	
slbmte	None	Context-synchronizing instruction	11, 13
tlbie	None	Context-synchronizing instruction	7, 9
tlbiel	None	Context-synchronizing instruction	7, 9
tlbia	None	Context-synchronizing instruction	7
Store (PTE)	none	{**ptesync**, CSI]	8, 9
Note: Refer to *Section 2.3.16.1* on page 77 for explanation of notes.			

2.3.16.1 Notes for Table 2-16 and Table 2-17

1. Synchronization requirements for changing from one endian mode to the other using the **mtmsr**[**d**] instruction are implementation-dependent.
2. The effect of changing the EE bit is immediate, even if the mtmsr[d] instruction is not context synchronizing (i.e., even if L='1').
   - If an **mtmsr**[**d**] instruction sets the [EE] bit to '0', neither an External interrupt nor a Decrementer interrupt occurs after the **mtmsr**[**d**] is executed.
   - If an **mtmsr**[**d**] instruction changes the [EE] bit from 0 to 1 when an External, Decrementer, or higher priority exception exists, the corresponding interrupt occurs immediately after the **mtmsr**[**d**] is executed, and before the next instruction is executed in the program that set [EE] to '1'.
3. For software that will run on processors that comply with earlier versions of the architecture, a context synchronizing instruction is required after the **mtmsr**[**d**] instruction.
4. Synchronization requirements for changing the Data Address Breakpoint Register are implementation-dependent.
5. SDR1 must not be altered when MSR[DR] = '1' or MSR[IR] = '1' ; if it is, the results are undefined.
6. A **ptesync** instruction is required before the **mtspr** instruction because (a) SDR1 identifies the Page Table and thereby the location of Reference and Change bits, and (b) on some implementations, use of SDR1 to update Reference and Change bits may be independent of translating the virtual address. (For example, an implementation might identify the PTE in which to update the Reference and Change bits in terms of its offset in the Page Table, instead of its real address, and then add the Page Table address from SDR1 to the offset to determine the real address at which to update the bits.) To ensure that Reference and Change bits are updated in the correct Page Table, SDR1 must not be altered until all Reference and Change bit updates associated with address translations that were performed, by the processor executing the **mtspr** instruction, before the **mtspr** instruction is executed have been performed with respect to that processor. A **ptesync** instruction guarantees this synchronization of Reference and Change bit updates, while neither a context synchronizing operation nor the instruction fetching mechanism does so.
7. For data accesses, the context synchronizing instruction before the **tlbie**, **tlbiel**, or **tlbia** instruction ensures that all preceding instructions that access data storage have completed to a point at which they have reported all exceptions they will cause. The context synchronizing instruction after the **tlbie**, **tlbiel**, or **tlbia** instruction ensures that storage accesses associated with instructions following the context synchronizing instruction will not use the TLB entry(s) being invalidated. (If it is necessary to order storage accesses associated with preceding instructions, or Reference and Change bit updates associated with preceding address translations, with respect to subsequent data accesses, a **ptesync** instruction must also be used, either before or after the **tlbie**, **tlbiel**, or **tlbia** instruction.
8. The notation "{ **ptesync**,CSI}" denotes an instruction sequence. Other instructions may be interleaved with this sequence, but these instructions must appear in the order shown.

   No software synchronization is required before the Store instruction because (a) stores are not performed out-of-order and (b) address translations associated with instructions preceding the Store instruction are not performed again after the store has been performed). These properties ensure that all address translations associated with instructions preceding the Store instruction will be performed using the old contents of the PTE.

   The **ptesync** instruction after the Store instruction ensures that all searches of the Page Table that are performed after the **ptesync** instruction completes will use the value stored (or a value stored subsequently). The context synchronizing instruction after the **ptesync** instruction ensures that any address translations associated with instructions following the context synchronizing instruction that were performed using the old contents of the PTE will be discarded, with the result that these address translations will be performed again and, if there is no corresponding TLB entry, will use the value stored (or a value stored subsequently).

9. There are additional software synchronization requirements for the **tlbie** instruction in multiprocessor environments. In a multiprocessor system, if software locking is used to help ensure that the requirements are satisfied, the **isync** instruction near the end of the lock acquisition sequence may naturally provide the context synchronization that is required before the alteration.
10. The alteration must not cause an implicit branch in effective address space. Thus, when changing MSR[SF] from 1 to 0, the **mtmsrd** instruction must have an effective address that is less than 2^{32} - 4. Furthermore, when changing MSR[SF] from 0 to 1, the **mtmsrd** instruction must not be at effective address 2^{32} - 4.
11. 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.
12. The elapsed time between the contents of the Decrementer becoming negative and the signaling of the corresponding exception is not defined.
13. If an **slbmte** instruction alters the mapping, or associated attributes, of a currently mapped ESID, the **slbmte** must be preceded by an **slbie** (or **slbia**) instruction that invalidates the existing translation. This applies even if the corresponding entry is no longer in the SLB (the translation may still be in implementation-specific address translation lookaside information). No software synchronization is needed between the **slbie** and the **slbmte**, regardless of whether the index of the SLB entry (if any) containing the current translation is the same as the SLB index specified by the **slbmte**.

    No **slbie** (or **slbia**) is needed if the **slbmte** instruction replaces a valid SLB entry with a mapping of a different ESID (for example, to satisfy an SLB miss). However, the **slbie** is needed later if and when the translation that was contained in the replaced SLB entry is to be invalidated.

3. Operand Conventions

This chapter describes the operand conventions as they are represented in two levels of the PowerPC Architecture—user instruction set architecture (UISA) and virtual environment architecture (VEA). Detailed descriptions are provided of conventions used for storing values in registers and memory, accessing PowerPC registers, and representing data in these registers in both big and little-endian modes. Additionally, the floating-point data formats and exception conditions are described. Refer to *Appendix C Floating-Point Models* for more information on the implementation of the IEEE floating-point execution models.

3.1 Data Organization in Memory and Data Transfers

In a PowerPC microprocessor-based system, bytes in memory are numbered consecutively starting with 0. Each number is the address of the corresponding byte. Memory operands may be bytes, halfwords, words, or doublewords, or, for the load and store multiple and the load and store string instructions, a sequence of bytes or words. The address of a memory operand is the address of its first byte (that is, of its lowest-numbered byte). Operand length is implicit for each instruction.

The following sections describe the concepts of alignment and byte ordering of data, and their significance to the PowerPC Architecture.

3.1.1 Aligned and Misaligned Accesses

The operand of a single-register memory access instruction has a natural alignment boundary equal to the operand length. In other words, the natural address of an operand is an integral multiple of the operand length. A memory operand is said to be aligned if it is aligned at its natural boundary; otherwise it is misaligned. Instructions are always four bytes long and word-aligned.

Operands for single-register memory access instructions have the characteristics shown in *Table 3-1*. (Although not permitted as memory operands, quad words are shown because quad-word alignment is desirable for certain memory operands.)

Table 3-1. Memory Operand Alignment

Operand	Length	Aligned Address [60–63] (if aligned)
Byte	8 bits	xxxx
Halfword	2 bytes	xxx0
Word	4 bytes	xx00
Doubleword	8 bytes	x000
Quadword	16 bytes	0000
Note: An x in an address bit position indicates that the bit can be 0 or 1 independent of the state of other bits in the address.		

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

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

3.1.2 Byte Ordering

If individual data items were indivisible, the concept of byte ordering would be unnecessary. The order of bits or groups of bits within the smallest addressable unit of memory is irrelevant, because nothing can be observed about such order. Order matters only when scalars, which the processor and programmer regard as indivisible quantities, can be made up of more than one addressable unit of memory.

For PowerPC processors, the smallest addressable memory unit is the byte (8 bits), and scalars are composed of one or more sequential bytes. Many scalars are halfwords, words, or doublewords, which consist of groups of bytes. When a word-length scalar (32-bit) is moved from a register to memory, the scalar occupies four consecutive byte addresses. It thus becomes meaningful to discuss the order of the byte addresses with respect to the value of the scalar: which byte contains the highest-order 8 bits of the scalar, which byte contains the next highest-order 8 bits, and so on.

Although the choice of byte ordering is arbitrary, only two orderings are practical—big-endian and little-endian. The PowerPC Architecture supports both big and little-endian byte ordering. The default byte ordering is big-endian.

3.1.2.1 Big-Endian Byte Ordering

For big-endian scalars, the most-significant byte (MSB) is stored at the lowest (or starting) address while the least-significant byte (LSB) is stored at the highest (or ending) address. This is called big-endian because the big end of the scalar comes first in memory.

3.1.2.2 Little-Endian Byte Ordering

For little-endian scalars, the least-significant byte is stored at the lowest (or starting) address while the most-significant byte is stored at the highest (or ending) address. This is called little-endian because the little end of the scalar comes first in memory.

3.1.3 Structure Mapping Examples

Figure 3-1 shows a C programming example that contains an assortment of scalars and one array of characters (a string). The value presumed to be in each structure element is shown in hexadecimal in the comments (except for the character array, which is represented by a sequence of characters, each enclosed in single quote marks).

Figure 3-1. C Program Example—Data Structure S

```
struct {
        int     a;      /* 0x1112_1314                  word            */
        double  b;      /* 0x2122_2324_2526_2728        doubleword      */
        char *  c;      /* 0x3132_3334                  word            */
        char    d[7];   /* 'L','M','N','O','P','Q','R'  array of bytes  */
        short   e;      /* 0x5152                       halfword        */
        int     f;      /* 0x6162_6364                  word            */
} S;
```

The data structure *S* is used throughout this section to demonstrate how the bytes that comprise each element (*a*, *b*, *c*, *d*, *e*, and *f*) are mapped into memory.

3.1.3.1 Big-Endian Mapping

The big-endian mapping of the structure, *S*, is shown in *Figure 3-2*. Addresses are shown in hexadecimal below each byte. The content of each byte, as shown in the preceding C programming example, is shown in hexadecimal and, for the character array, as characters enclosed in single quote marks.

Note: The most-significant byte of each scalar is at the lowest address.

Figure 3-2. Big-Endian Mapping of Structure S

Contents	11	12	13	14	(x)	(x)	(x)	(x)
Address	00	01	02	03	04	05	06	07
Contents	21	22	23	24	25	26	27	28
Address	08	09	0A	0B	0C	0D	0E	0F
Contents	31	32	33	34	‘L’	‘M’	‘N’	‘O’
Address	10	11	12	13	14	15	16	17
Contents	‘P’	‘Q’	‘R’	(x)	51	52	(x)	(x)
Address	18	19	1A	1B	1C	1D	1E	1F
Contents	61	62	63	64	(x)	(x)	(x)	(x)
Address	20	21	22	23	24	25	26	27

The structure mapping introduces padding (skipped bytes indicated by (x) in *Figure 3-2*) in the map in order to align the scalars on their proper boundaries—four bytes between elements *a* and *b*, one byte between elements *d* and *e*, and two bytes between elements *e* and *f*.

Note: The padding is dependent on the compiler; it is not a function of the architecture.

3.1.3.2 Little-Endian Mapping

Figure 3-3 shows the structure, *S*, using little-endian mapping.

Note: The least-significant byte of each scalar is at the lowest address.

Figure 3-3. Little-Endian Mapping of Structure S

Contents	14	13	12	11	(x)	(x)	(x)	(x)
Address	00	01	02	03	04	05	06	07
Contents	28	27	26	25	24	23	22	21
Address	08	09	0A	0B	0C	0D	0E	0F
Contents	34	33	32	31	‘L’	‘M’	‘N’	‘O’
Address	10	11	12	13	14	15	16	17
Contents	‘P’	‘Q’	‘R’	(x)	52	51	(x)	(x)
Address	18	19	1A	1B	1C	1D	1E	1F
Contents	64	63	62	61	(x)	(x)	(x)	(x)
Address	20	21	22	23	24	25	26	27

Figure 3-3 shows the sequence of doublewords laid out with addresses increasing from left to right. Programmers familiar with little-endian byte ordering may be more accustomed to viewing doublewords laid out with addresses increasing from right to left, as shown in *Figure 3-4*. This allows the little-endian programmer to view each scalar in its natural byte order of MSB to LSB. However, to demonstrate how the PowerPC Architecture provides both big and little-endian support, this section uses the convention of showing addresses increasing from left to right, as in *Figure 3-3*.

Figure 3-4. Little-Endian Mapping of Structure S —Alternate View

Contents	(x)	(x)	(x)	(x)	11	12	13	14
Address	07	06	05	04	03	02	01	00
Contents	21	22	23	24	25	26	27	28
Address	0F	0E	0D	0C	0B	0A	09	08
Contents	‘O’	‘N’	‘M’	‘L’	31	32	33	34
Address	17	16	15	14	13	12	11	10
Contents	(x)	(x)	51	52	(x)	‘R’	‘Q’	‘P’
Address	1F	1E	1D	1C	1B	1A	19	18
Contents	(x)	(x)	(x)	(x)	61	62	63	64
Address	27	26	25	24	23	22	21	20

3.1.4 PowerPC Byte Ordering

The PowerPC Architecture supports both big and little-endian byte ordering. The default byte ordering is big-endian. The code sequence used to switch from big to little-endian mode may differ among processors.

The PowerPC Architecture defines two bits in the MSR for specifying byte ordering—LE (little-endian mode) and ILE (interrupt little-endian mode). The LE bit specifies the endian mode in which the processor is currently operating and ILE specifies the mode to be used when an exception handler is invoked. That is, when an exception occurs, the ILE bit (as set for the interrupted process) is copied into MSR[LE] to select the endian mode for the context established by the exception. For both bits, a value of 0 specifies big-endian mode and a value of 1 specifies little-endian mode.

The PowerPC Architecture also provides load and store instructions that reverse byte ordering. These instructions have the effect of loading and storing data in the endian mode opposite from that which the processor is operating. See *Section 4.2.3.4 Integer Load and Store with Byte-Reverse Instructions* for more information on these instructions.

3.1.4.1 Aligned Scalars in Little-Endian Mode

Chapter 4, Addressing Modes and Instruction Set Summary describes the effective address calculation for the load and store instructions. For processors in little-endian mode, the effective address is modified before being used to access memory. The three low-order address bits of the effective address are exclusive-ORed (XOR) with a three-bit value that depends on the length of the operand (1, 2, 4, or 8 bytes), as shown in *Table 3-2*. This address modification is called ‘munging’.

Note: Although the process is described in the architecture, the actual term ‘munging’ is not defined or used in the specification. However, the term is commonly used to describe the effective address modifications necessary for converting big-endian addressed data to little-endian addressed data.

Table 3-2. Little Endian Effective Address Modifications for Individual Aligned Scalars

Data Width (Bytes)	Effective Address Modification
8	No change
4	XOR with 0b100
2	XOR with 0b110
1	XOR with 0b111

The munged physical address is passed to the cache or to main memory, and the specified width of the data is transferred (in big-endian order—that is, MSB at the lowest address, LSB at the highest address) between a GPR or FPR and the addressed memory locations (as modified).

Munging makes it appear to the processor that individual aligned scalars are stored as little-endian, when in fact they are stored in big-endian order, but at different byte addresses within doublewords. Only the address is modified, not the byte order.

Taking into account the preceding description of munging, in little-endian mode, structure *S* is placed in memory as shown in *Figure 3-5*.

Figure 3-5. Munged Little-Endian Structure S as Seen by the Memory Subsystem

Contents	(x)	(x)	(x)	(x)	11	12	13	14
Address	00	01	02	03	04	05	06	07
Contents	21	22	23	24	25	26	27	28
Address	08	09	0A	0B	0C	0D	0E	0F
Contents	‘O’	‘N’	‘M’	‘L’	31	32	33	34
Address	10	11	12	13	14	15	16	17
Contents	(x)	(x)	51	52	(x)	‘R’	‘Q’	‘P’
Address	18	19	1A	1B	1C	1D	1E	1F
Contents	(x)	(x)	(x)	(x)	61	62	63	64
Address	20	21	22	23	24	25	26	27

Note: The mapping shown in *Figure 3-5* is not a true little-endian mapping of the structure *S*. However, because the processor munges the address when accessing memory, the physical structure *S* shown in *Figure 3-5* appears to the processor as the structure *S* shown in *Figure 3-6*.

Figure 3-6. Munged Little-Endian Structure S as Seen by Processor

Contents	14	13	12	11				
Address	00	01	02	03	04	05	06	07
Contents	28	27	26	25	24	23	22	21
Address	08	09	0A	0B	0C	0D	0E	0F
Contents	34	33	32	31	‘L’	‘M’	‘N’	‘O’
Address	10	11	12	13	14	15	16	17
Contents	‘P’	‘Q’	‘R’		52	51		
Address	18	19	1A	1B	1C	1D	1E	1F
Contents	64	63	62	61				
Address	20	21	22	23	24	25	26	27

As seen by the program executing in the processor, the mapping for the structure *S* (*Figure 3-6*) is identical to the little-endian mapping shown in *Figure 3-3*. However, from outside of the processor, the addresses of the bytes making up the structure *S* are as shown in *Figure 3-5*. These addresses match neither the big-endian mapping of *Figure 3-2* nor the true little-endian mapping of *Figure 3-3*. This must be taken into account when performing I/O operations in little-endian mode; this is discussed in *Section 3.1.4.6 PowerPC Input/Output Data Transfer Addressing in Little-Endian Mode*.

3.1.4.2 Misaligned Scalars in Little-Endian Mode

Performing an XOR operation on the low-order bits of the address works only if the scalar is aligned on a boundary equal to a multiple of its length. *Figure 3-7* shows a true little-endian mapping of the four-byte word 0x1112_1314, stored at address 05.

Figure 3-7. True Little-Endian Mapping, Word Stored at Address 05

Contents						14	13	12
Address	00	01	02	03	04	05	06	07

Contents	11							
Address	08	09	0A	0B	0C	0D	0E	0F

For the true little-endian example in *Figure 3-7*, the least-significant byte (0x14) is stored at address 0x05, the next byte (0x13) is stored at address 0x06, the third byte (0x12) is stored at address 0x07, and the most-significant byte (0x11) is stored at address 0x08.

When a PowerPC processor, in little-endian mode, issues a single-register load or store instruction with a misaligned effective address, it may take an alignment exception. In this case, a single-register load or store instruction means any of the integer load/store, load/store with byte-reverse, floating-point load/store (including **stfiwx**) instructions, and Load And Reserve and Store Conditional.

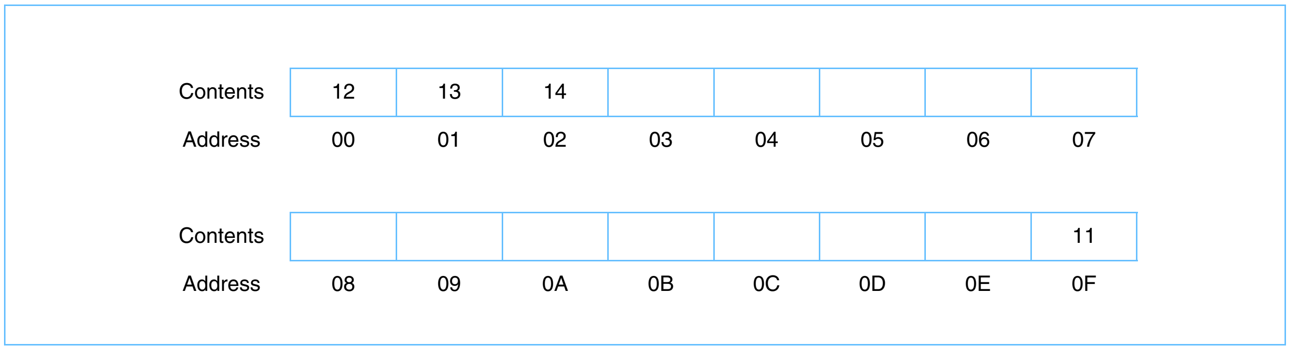
The Load and Store with Byte Reversal instructions have the effect of loading or storing data in the opposite endian mode from that in which the processor is running. Data is loaded or stored in little-endian order if the processor is running in big-endian mode, and in big-endian order if the processor is running in little-endian mode.

PowerPC processors in little-endian mode are not required to invoke an alignment exception when such a misaligned access is attempted. The processor may handle some or all such accesses without taking an alignment exception.

The PowerPC Architecture requires that halfwords, words, and doublewords be placed in memory such that the little-endian address of the lowest-order byte is the effective address computed by the load or store instruction; the little-endian address of the next-lowest-order byte is one greater, and so on. (Load And Reserve and Store Conditional differ somewhat from the rest of the instructions in that neither the implementation nor the system alignment error handler is expected to handle these four instructions correctly if their operands are not aligned.) However, because PowerPC processors in little-endian mode munge the effective address, the order of the bytes of a misaligned scalar must be as if they were accessed one at a time.

Using the same example as shown in *Figure 3-7*, when the least-significant byte (0x14) is stored to address 0x05, the address is XORed with 0b111 to become 0x02. When the next byte (0x13) is stored to address 0x06, the address is XORed with 0b111 to become 0x01. When the third byte (0x12) is stored to address 0x07, the address is XORed with 0b111 to become 0x00. Finally, when the most-significant byte (0x11) is stored to address 0x08, the address is XORed with 0b111 to become 0x0F. *Figure 3-8* shows the misaligned word, stored by a little-endian program, as seen by the memory subsystem.

Figure 3-8. Word Stored at Little-Endian Address 05 as Seen by the Memory Subsystem



Note: The misaligned word in this example spans two doublewords. The two parts of the misaligned word are not contiguous as seen by the memory system. An implementation may support some but not all misaligned little-endian accesses. For example, a misaligned little-endian access that is contained within a doubleword may be supported, while one that spans doublewords may cause an alignment exception.

3.1.4.3 Nonscalars

The PowerPC Architecture has two types of instructions that handle nonscalars (multiple instances of scalars):

- Load and store multiple instructions
- Load and store string instructions

Because these instructions typically operate on more than one word-length scalar, munging cannot be used. These types of instructions cause alignment exception conditions when the processor is executing in little-endian mode. Although string accesses are not supported, they are inherently byte-based operations, and can be broken into a series of word-aligned accesses.

3.1.4.4 Page Tables

The layout of the page table in memory is independent of endian mode. A given byte in the page table must be accessed using an effective address appropriate to the mode of the executing program (for example, the high-order byte of a Page Table Entry must be accessed with an effective address ending with 0b000 in big-endian mode, and with an effective address ending with 0b111 in little-endian mode).

3.1.4.5 PowerPC Instruction Addressing in Little-Endian Mode

Each PowerPC instruction occupies an aligned word of memory. PowerPC processors fetch and execute instructions as if the current instruction address is incremented by four for each sequential instruction. When operating in little-endian mode, the instruction address is munged as described in *Section 3.1.4.1 Aligned Scalars in Little-Endian Mode* for fetching word-length scalars; that is, the instruction address is XORed with 0b100. A program is thus an array of little-endian words with each word fetched and executed in order (not including branches).

All instruction addresses visible to an executing program are the effective addresses that are computed by that program, or, in the case of the exception handlers, effective addresses that were or could have been computed by the interrupted program. These effective addresses are independent of the endian mode. Examples for little-endian mode include the following:

- An instruction address placed in the link register by branch and link operation, or an instruction address saved in an SPR when an exception is taken, is the address that a program executing in little-endian mode would use to access the instruction as a word of data using a load instruction.
- An offset in a relative branch instruction reflects the difference between the addresses of the branch and target instructions, where the addresses used are those that a program executing in little-endian mode would use to access the instructions as data words using a load instruction.
- A target address in an absolute branch instruction is the address that a program executing in little-endian mode would use to access the target instruction as a word of data using a load instruction.
- The memory locations that contain the first set of instructions executed by each kind of exception handler must be set in a manner consistent with the endian mode in which the exception handler is invoked. Thus, if the exception handler is to be invoked in little-endian mode, the first set of instructions comprising each kind of exception handler must appear in memory with the instructions within each doubleword reversed from the order in which they are to be executed.

3.1.4.6 PowerPC Input/Output Data Transfer Addressing in Little-Endian Mode

For a PowerPC system running in big-endian mode, both the processor and the memory subsystem recognize the same byte as byte 0. However, this is not true for a PowerPC system running in little-endian mode because of the munged address bits when the processor accesses memory.

For I/O transfers in little-endian mode to transfer bytes properly, they must be performed as if the bytes transferred were accessed one at a time, using the little-endian address modification appropriate for the single-byte transfers (that is, the lowest order address bits must be XORed with 0b111). This does not mean that I/O operations in little-endian PowerPC systems must be performed using only one-byte-wide transfers. Data transfers can be as wide as desired, but the order of the bytes within doublewords must be as if they were fetched or stored one at a time. That is, for a true little-endian I/O device, the system must provide a mechanism to munge and unmunge the addresses and reverse the bytes within a doubleword (MSB to LSB).

However, not all I/O done on PowerPC systems is for large areas of storage as described above. I/O can be performed with certain devices merely by storing to or loading from addresses that are associated with the devices (the terms “memory-mapped I/O” and “programmed I/O” or “PIO” are used for this). For such PIO transfers, care must be taken when defining the addresses to be used, for these addresses are subject to the effective address modification shown in *Table 3-2 Little Endian Effective Address Modifications for Individual Aligned Scalars*. A *Load* or *Store* instruction that maps to a control register on a device may require that the value loaded or stored have its bytes reversed; if this is required, the *Load and Store with Byte Reversal* instructions can be used. Any requirement for such byte reversal for a particular I/O device register is independent of whether the PowerPC system is running in big-endian or little-endian mode.

Similarly, the address sent to an I/O device by an **eciwx** or **ecowx** instruction is subject to the effective address modification shown in *Table 3-2*.

3.2 Effect of Operand Placement on Performance—VEA

The PowerPC VEA states that the placement (location and alignment) of operands in memory affects the relative performance of memory accesses. The best performance is guaranteed if memory operands are aligned on natural boundaries. For more information on memory access ordering and atomicity, refer to *Section 5.1 The Virtual Environment*.

3.2.1 Summary of Performance Effects

To obtain the best performance across the widest range of PowerPC processor implementations, the programmer should assume the performance model described in *Table 3-3* and *Table 3-4* with respect to the placement of memory operands.

The performance of accesses varies depending on:

- Operand size
- Operand alignment
- Endian mode (big-endian or little-endian)
- Crossing no boundary
- Crossing a cache block boundary
- Crossing a virtual page boundary
- Crossing a segment boundary

Table 3-3 applies when the processor is in big-endian mode.

Table 3-3. Performance Effects of Memory Operand Placement, Big-Endian Mode

Operand		Boundary Crossing			
Size	Byte Alignment	None	Cache Block	Virtual Page[1]	Segment
Integer					
8 byte	8 4 <4	Optimal Good Good	— Good Good	— Good Good	— Poor Poor
4 byte	4 <4	Optimal Good	— Good	— Good	— Poor
2 byte	2 <2	Optimal Good	— Good	— Good	— Poor
1 byte	1	Optimal	—	—	—
lmw, **stmw**	4 <4	Good Poor	Good Poor	Good Poor	Poor Poor
String	—	Good	Good	Good	Poor
Floating Point					
8 byte	8 4 <4	Optimal Good Poor	— Good Poor	— Poor Poor	— Poor Poor
4 byte	4 <4	Optimal Poor	— Poor	— Poor	— Poor

Note:
1. If the memory operand spans two virtual pages that have different memory control attributes, performance is likely to be poor.
2. If an instruction causes an access that is not atomic and any portion of the operand is in memory that is write through required or caching inhibited, performance is likely to be poor.

Table 3-4 applies when the processor is in little-endian mode.

Table 3-4. Performance Effects of Memory Operand Placement, Little-Endian Mode

Operand		Boundary Crossing			
Size	Byte Alignment	None	Cache Block	Virtual Page[1]	Segment
Integer					
8 byte	8 4 <4	Optimal Good Poor	— Good Poor	— Poor Poor	— Poor Poor
4 byte	4 <4	Optimal Good	— Good	— Poor	— Poor
2 byte	2 <2	Optimal Good	— Good	— Poor	— Poor
1 byte	1	Optimal	—	—	—
Floating Point					
8 byte	8 4 <4	Optimal Good Poor	— Good Poor	— Poor Poor	— Poor Poor
4 byte	4 <4	Optimal Poor	— Poor	— Poor	— Poor

Note:
1. If the memory operand spans two virtual pages that have different memory control attributes, performance is likely to be poor.
2. If an instruction causes an access that is not atomic and any portion of the operand is in memory that is write through required or caching inhibited, performance is likely to be poor.

The load/store multiple and the load/store string instructions are supported only in big-endian mode. The load/store multiple instructions are defined by the PowerPC Architecture to operate only on aligned operands. The load/store string instructions have no alignment requirements.

3.2.2 Instruction Restart

In this section the "load instruction" includes the cache management and other instructions that are stated in the instruction descriptions to be "treated as a load," and similarly for "store instruction." The following instructions are never restarted after having accessed any portion of the memory operand (unless the instruction causes a "data address compare match" or a "data address breakpoint match").

1. Store instruction that causes an atomic access.
2. Load instruction that causes an atomic access to memory that is both caching inhibited and guarded.

Any other load or store instruction may be partially executed and then aborted after having accessed a portion of the memory operand, and then re-executed (i.e., restarted, by the processor or the operating system). If an instruction is partially executed, the contents of registers are preserved to the extent that the correct result will be produced when the instruction is re-executed.

There are many events which might cause a load or store instruction to be restarted. For example, a hardware error may cause execution of the instruction to be aborted after part of the access has been performed, and the recovery operation could then cause the aborted instruction to be re-executed.

When an instruction is aborted after being partially executed, the contents of the instruction pointer indicate that the instruction has not been executed, however the contents of some registers may have been altered and some bytes within the memory operand may have been accessed. The following are examples of an instruction being partially executed and altering the program state even though it appears that the instruction has not been executed.

1. Load multiple, load string: some registers in the range of registers to be loaded may have been altered.
2. Any store instruction, **dcbz**: some bytes of the memory operand may have been altered.
3. Any floating point load instruction: the target register (**fr**D) may have been altered.

3.3 Floating-Point Execution Models—UISA

There are two kinds of floating-point instructions defined for the PowerPC Architecture: computational and non computational. The computational instructions consist of those operations defined by the IEEE-754 standard for 32 and 64-bit arithmetic (those that perform addition, subtraction, multiplication, division, extracting the square root, rounding conversion, comparison, and combinations of these) and the multiply-add and reciprocal estimate instructions defined by the architecture. The non computational floating-point instructions consist of the floating-point load, store, and move instructions. While both the computational and non computational instructions are considered to be floating-point instructions governed by the MSR[FP] bit (that allows floating-point instructions to be executed), only the computational instructions are considered floating-point operations throughout this chapter.

The IEEE standard requires that single-precision arithmetic be provided for single-precision operands. The standard permits double-precision arithmetic instructions to have either (or both) single-precision or double-precision operands, but states that single-precision arithmetic instructions should not accept double-precision operands. The guidelines are as follows:

- Double-precision arithmetic instructions may have single-precision operands but always produce double-precision results.
- Single-precision arithmetic instructions require all operands to be single-precision and always produce single-precision results.

For arithmetic instructions, conversion from double to single-precision must be done explicitly by software, while conversion from single to double-precision is done implicitly by the processor.

All PowerPC implementations provide the equivalent of the following execution models to ensure that identical results are obtained. The definition of the arithmetic instructions for infinities, denormalized numbers, and NaNs follow conventions described in the following sections. *Appendix C Floating-Point Models* has additional detailed information on the execution models for IEEE operations, as well as the other floating-point instructions.

Although the double-precision format specifies an 11-bit exponent, exponent arithmetic uses two additional bit positions to avoid potential transient overflow conditions. An extra bit is required when denormalized double-precision numbers are prenormalized. A second bit is required to permit computation of the adjusted exponent value in the following examples when the corresponding exception enable bit is 1 (exceptions are referred to as interrupts in the architecture specification):

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

3.3.1 Floating-Point Data Format

The PowerPC UISA defines the representation of a floating-point value in two different binary, fixed-length formats. The format is a 32-bit format for a single-precision floating-point value or a 64-bit format for a double-precision floating-point value. The single-precision format may be used for data in memory. The double-precision format can be used for data in memory or in floating-point registers (FPRs).

The lengths of the exponent and the fraction fields differ between these two formats. The layout of the single-precision format is shown in *Figure 3-9*; the layout of the double-precision format is shown in *Figure 3-10*.

Figure 3-9. Floating-Point Single-Precision Format

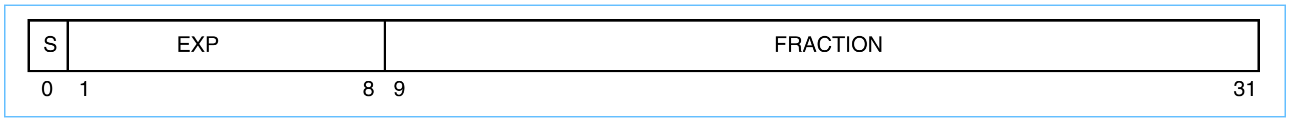
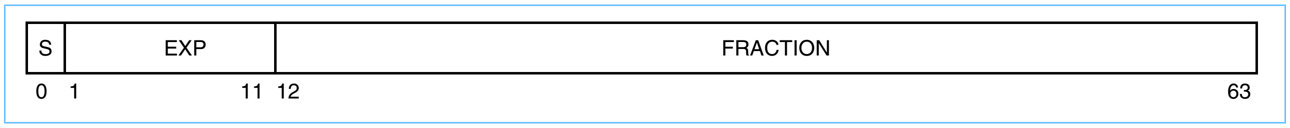


Figure 3-10. Floating-Point Double-Precision Format



Values in floating-point format consist of three fields:

- S (sign bit)
- EXP (exponent + bias)
- FRACTION (fraction)

If only a portion of a floating-point data item in memory is accessed, as with a load or store instruction for a byte or halfword (or word in the case of floating-point double-precision format), the value affected depends on whether the PowerPC system is using big or little-endian byte ordering, which is described in *Section 3.1.2 Byte Ordering*.

Note: Big-endian mode is the default.

For numeric values, the significand consists of a leading implied bit concatenated on the right with the FRACTION. This leading implied bit is a 1 for normalized numbers and a 0 for denormalized numbers and is 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 3-5*.

Table 3-5. IEEE Floating-Point Fields

Parameter	Single-Precision	Double-Precision
Exponent bias	+127	+1023
Maximum exponent (unbiased)	+127	+1023
Minimum exponent (unbiased)	–126	–1022
Format width	32 bits	64 bits
Sign width	1 bit	1 bit
Exponent width	8 bits	11 bits
Fraction width	23 bits	52 bits
Significand width	24 bits	53 bits

The true value of the exponent can be determined by subtracting 127 for single-precision numbers and 1023 for double-precision numbers. This is shown in *Table 3-6*.

Note: Two exponent values are reserved to represent special-case values. Setting all bits indicates that the value is an infinity or NaN and clearing all bits indicates that the number is either zero or denormalized.

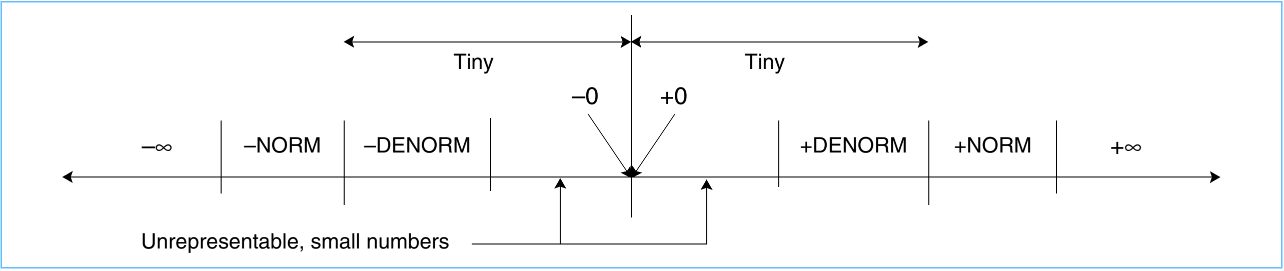
Table 3-6. Biased Exponent Format

Biased Exponent (Binary)	Single-Precision (Unbiased)	Double-Precision (Unbiased)
11. 11	Reserved for infinities and NaNs	
11. 10	+127	+1023
11. 01	+126	+1022
. . .	. . .	. . .
10. 00	1	1
01. 11	0	0
01. 10	–1	–1
. . .	. . .	. . .
00. 01	–126	–1022
00. 00	Reserved for zeros and denormalized numbers	

3.3.1.1 Value Representation

The PowerPC UISA defines numeric and nonnumeric values representable within single and double-precision formats. The numerical values are approximations to the real numbers and include the normalized numbers, denormalized numbers, and zero values. The nonnumeric values representable are the positive and negative infinities and the Not a Numbers (NaNs). The positive and negative infinities are adjoined to the real numbers, but are not numbers themselves, and the standard rules of arithmetic do not hold when they appear 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 numerical entities is shown in *Figure 3-11*. Tiny values include denormalized numbers and all numbers that are too small to be represented for a particular precision format; they do not include zero values.

Figure 3-11. Approximation to Real Numbers



The positive and negative NaNs are encodings that convey diagnostic information such as the representation of uninitialized variables and are not related to the numbers, ±∞, or each other by order or value.

Table 3-7 describes each of the floating-point formats.

Table 3-7. Recognized Floating-Point Numbers

Sign Bit	Biased Exponent	Implied Bit	Fraction	Value
0	Maximum	x	Nonzero	NaN
0	Maximum	x	Zero	+Infinity
0	0 < Exponent < Maximum	1	x	+Normalized
0	0	0	Nonzero	+Denormalized
0	0	x	Zero	+0
1	0	x	Zero	–0
1	0	0	Nonzero	–Denormalized
1	0 < Exponent < Maximum	1	x	–Normalized
1	Maximum	x	Zero	–Infinity
1	Maximum	x	Nonzero	NaN

The following sections describe floating-point values defined in the architecture.

3.3.1.2 Binary Floating-Point Numbers

Binary floating-point numbers are machine-representable values used to approximate real numbers. Three categories of numbers are supported—normalized numbers, denormalized numbers, and zero values.

3.3.1.3 Normalized Numbers (±NORM)

The values for normalized numbers have a biased exponent value in the range:

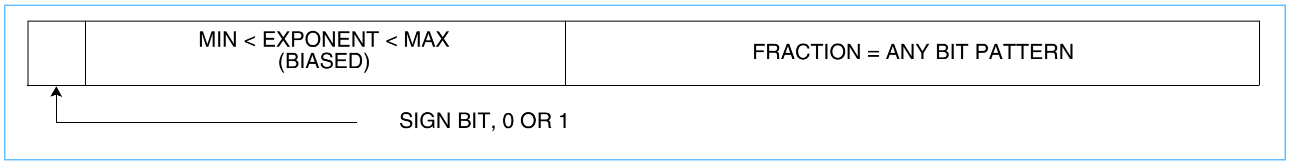
- 1 to 254 in single-precision format
- 1 to 2046 in double-precision format

The implied unit bit is one. Normalized numbers are interpreted as follows:

NORM = $(-1)^s \times 2^E \times (1.\text{fraction})$

The variable (s) is the sign, (E) is the unbiased exponent, and (1.fraction) is the significand composed of a leading unit bit (implied bit) and a fractional part. The format for normalized numbers is shown in *Figure 3-12*.

Figure 3-12. Format for Normalized Numbers



The ranges covered by the magnitude (M) of a normalized floating-point number are approximated in the following decimal representation:

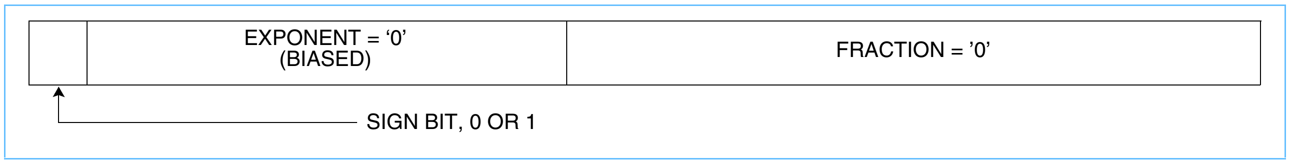
Single-precision format:
$1.2\text{x}10^{-38} \leq M \leq 3.4\text{x}10^{38}$

Double-precision format:
$2.2\text{x}10^{-308} \leq M \leq 1.8\text{x}10^{308}$

3.3.1.4 Zero Values (±0)

Zero values have a biased exponent value of zero and fraction of zero. This is shown in *Figure 3-13*. Zeros can have a positive or negative sign. The sign of zero is ignored by comparison operations (that is, comparison regards +0 as equal to –0). Arithmetic with zero results is always exact and does not signal any exception, except when an exception occurs due to the invalid operations as described in the *Invalid Operation Exception Condition* on page 111. Rounding a zero only affects the sign (±0).

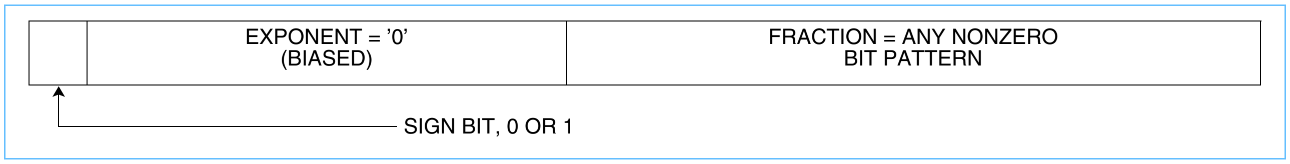
Figure 3-13. Format for Zero Numbers



3.3.1.5 Denormalized Numbers (±DENORM)

Denormalized numbers have a biased exponent value of zero and a nonzero fraction. The format for denormalized numbers is shown in *Figure 3-14*.

Figure 3-14. Format for Denormalized Numbers



Denormalized numbers are nonzero numbers smaller in magnitude than the normalized numbers. They are values in which the implied unit bit is zero. Denormalized numbers are interpreted as follows:

$$\text{DENORM} = (-1)^{s} \text{ x } 2^{\text{Emin}} \text{ x } (0.\text{fraction})$$

The value Emin is the minimum unbiased exponent value for a normalized number (–126 for single-precision, –1022 for double-precision).

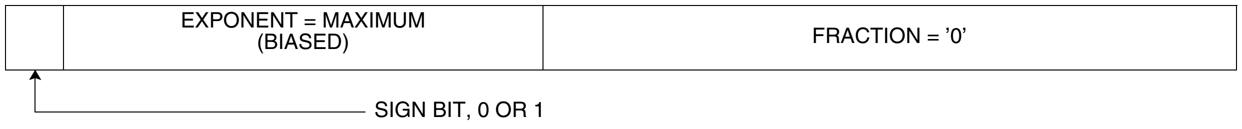
3.3.1.6 Infinities (±∞)

These are values that have the maximum biased exponent value of 255 in the single-precision format, 2047 in the double-precision format, and a zero fraction value. They are used to approximate values greater in magnitude than the maximum normalized value. Infinity arithmetic is defined as the limiting case of real arithmetic, with restricted operations defined among numbers and infinities. Infinities and the real numbers can be related by ordering in the affine sense:

$-\infty <$ every finite number $< +\infty$

The format for infinities is shown in *Figure 3-15*.

Figure 3-15. Format for Positive and Negative Infinities

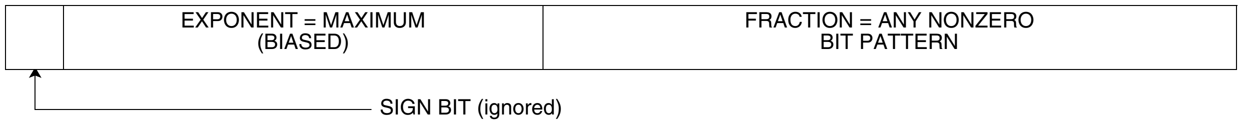


Arithmetic using infinite numbers is always exact and does not signal any exception, except when an exception occurs due to the invalid operations as described in *Invalid Operation Exception Condition* on page 111.

3.3.1.7 Not a Numbers (NaNs)

NaNs have the maximum biased exponent value and a nonzero fraction. The format for NaNs is shown in *Figure 3-16*. The sign bit of NaN does not show an algebraic sign; rather, it is simply another bit in the NaN. If the highest-order bit of the fraction field is a zero, the NaN is a signaling NaN; otherwise it is a quiet NaN (QNaN).

Figure 3-16. Format for NaNs



Signaling NaNs signal exceptions when they are specified as arithmetic operands.

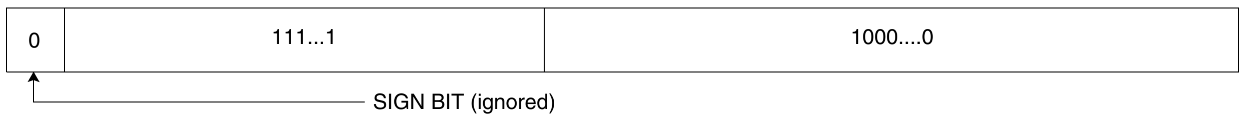
Quiet NaNs represent the results of certain invalid operations, such as attempts to perform arithmetic operations on infinities or NaNs, when the invalid operation exception is disabled (FPSCR[VE] = ’0’). Quiet NaNs propagate through all operations, except floating-point round to single-precision, ordered comparison, and conversion to integer operations, and signal exceptions only for ordered comparison and conversion to integer operations. Specific encodings in QNaNs can thus be preserved through a sequence of operations and used to convey diagnostic information to help identify results from invalid operations.

When a QNaN results from an operation because an operand is a NaN or because a QNaN is generated due to a disabled invalid operation exception, the following rule is applied to determine the QNaN to be stored as the result:

```
If (frA) is a NaN
  Then frD ← (frA)
  Else if (frB) is a NaN
    Then if instruction is frsp
      Then frD ← (frB)[0-34]||(29)0
      Else frD ← (frB)
    Else if (frC) is a NaN
      Then frD ← (frC)
      Else if generated QNaN
        Then frD ← generated QNaN
```

If the operand specified by **fr**A is a NaN, that NaN is stored as the result. Otherwise, if the operand specified by **fr**B is a NaN (if the instruction specifies an **fr**B operand), that NaN is stored as the result, with the low-order 29 bits cleared (if the instruction is **frsp***x*). Otherwise, if the operand specified by **fr**C is a NaN (if the instruction specifies an **fr**C operand), that NaN is stored as the result. Otherwise, if a QNaN is generated by a disabled invalid operation exception, that QNaN is stored as the result. If a QNaN is to be generated as a result, the QNaN generated has a sign bit of zero, an exponent field of all ones, and a highest-order fraction bit of one with all other fraction bits zero. An instruction that generates a QNaN as the result of a disabled invalid operation generates this QNaN (i.e., 0x7FF8_0000_0000_0000). This is shown in *Figure 3-17*.

Figure 3-17. Representation of Generated QNaN



A double-precision NaN is considered to be representable in single format if and only if the low-order 29 bits of the double-precision NaN's fraction are zero.

3.3.2 Sign of Result

The following rules govern the sign of the result of an arithmetic operation, when the operation does not yield an exception. These rules apply even when the operands or results are zero (0) or ±∞:

- The sign of the result of an addition operation is the sign of the source operand having the larger absolute value. If both operands have the same sign, the sign of the result of an addition operation is the same as the sign of the operands. The sign of the result of the subtraction operation, x – y, is the same as the sign of the result of the addition 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 negative infinity (–∞), in which case the sign is negative.
- The sign of the result of a multiplication or division operation is the XOR of the signs of the source operands.
- The sign of the result of a round to single-precision or convert to/from integer operation is the sign of the source operand.

- 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.

For multiply-add/subtract instructions, these rules are applied first to the multiplication operation and then to the addition/subtraction operation (one of the source operands to the addition/subtraction operation is the result of the multiplication operation).

3.3.3 Normalization and Denormalization

The intermediate result of an arithmetic or Floating Round to Single-Precision (**frsp***x*) instruction may require normalization and/or denormalization. When an intermediate result consists of a sign bit, an exponent, and a nonzero significand with a zero leading bit, the result must be normalized (and rounded) before being stored to the target.

A number is normalized by shifting its significand left and decrementing its exponent by one for each bit shifted until the leading significand bit becomes one. The guard and round bits are also shifted, with zeros shifted into the round bit; see *Appendix C.1 Execution Model for IEEE Operations* on page 597 for information about the guard and round bits. During normalization, the exponent is regarded as if its range was unlimited.

If an intermediate result has a nonzero significand and an exponent that is smaller than the minimum value that can be represented in the format specified for the result, this value is referred to as ‘tiny’ and the stored result is determined by the rules described in *Underflow Exception Condition* on page 116. These rules may involve denormalization. The sign of the number does not change.

An exponent can become tiny in either of the following circumstances:

- As the result of an arithmetic or Floating Round to Single-Precision (**frsp***x*) instruction or
- As the result of decrementing the exponent in the process of normalization.

Normalization is the process of coercing the leading significand bit to be a 1 while denormalization is the process of coercing the exponent into the target format's range.

In denormalization, the significand is shifted to the right while the exponent is incremented for each bit shifted until the exponent equals the format’s minimum value. The result is then rounded. If any significand bits are lost due to the rounding of the shifted value, the result is considered inexact. The sign of the number does not change and an Underflow Exception is signaled, see *Underflow Exception Condition* on page 116.

3.3.4 Data Handling and Precision

There are specific instructions for moving floating-point data between the FPRs and memory. For double-precision format data, the data is not altered during the move. For single-precision data, the format is converted to double-precision format when data is loaded from memory into an FPR. A format conversion from double to single-precision is performed when data from an FPR is stored as single-precision. These operations do not cause floating-point exceptions.

All floating-point arithmetic, move, and select instructions use floating-point double-precision format.

Floating-point single-precision formats are obtained by using the following four types of instructions:

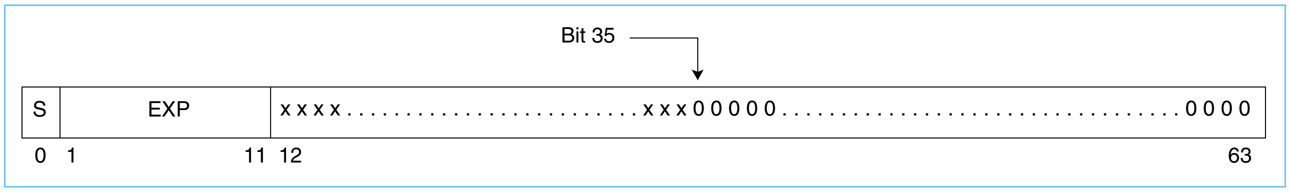
- Load floating-point single-precision instructions—These instructions access a single-precision operand in single-precision format in memory, convert it to double-precision, and load it into an FPR. Floating-point exceptions do not occur during the load operation.

- Floating Round to Single-Precision (**frsp***x*) instruction—The **frsp***x* 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 in the FPSCR. The instruction places that operand into an FPR as a double-precision operand. For results produced by single-precision arithmetic instructions and by single-precision loads, this operation does not alter the value.
- Single-precision arithmetic instructions—These instructions take operands from the FPRs in double-precision format, perform the operation as if it produced an intermediate result correct to infinite precision and with unbounded range, and then force this intermediate result to fit in single-precision format. Status bits in the FPSCR and in the condition register are set to reflect the single-precision result. The result is then converted to double-precision format and placed into an FPR. The result falls within the range supported by the single-precision format.

  Source operands for these instructions must be representable in single-precision format. Otherwise, the result placed into the target FPR and the setting of status bits in the FPSCR, and in the condition register if update mode is selected, are undefined.
- Store floating-point single-precision instructions—These instructions convert a double-precision operand to single-precision format and store that operand into memory. If the operand requires denormalization in order to fit in single-precision format, it is automatically denormalized prior to being stored. No exceptions are detected on the store operation (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 (**lfs**), Floating Round to Single-Precision (**frsp***x*), or single-precision arithmetic instruction is stored in an FPR, the low-order 29 fraction bits are zero. This is shown in *Figure 3-18*.

Figure 3-18. Single-Precision Representation in an FPR


Bit 35
S
EXP
x x x x . . . . x x x 0 0 0 0 0 . . . . 0 0 0 0
0
1
11
12
63

The **frsp***x* instruction allows conversion from double 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 before storing them into single-format memory elements or using them as operands for single-precision arithmetic instructions. Values produced by single-precision load and arithmetic instructions can be stored directly, or used directly as operands for single-precision arithmetic instructions, without being preceded by an **frsp***x* instruction.

A single-precision value can be used in double-precision arithmetic operations. The reverse is true only if the double-precision value can be represented in single-precision format. Some implementations may execute single-precision arithmetic instructions faster than double-precision arithmetic instructions. Therefore, if double-precision accuracy is not required, using single-precision data and instructions might speed operations in some implementations.

3.3.5 Rounding

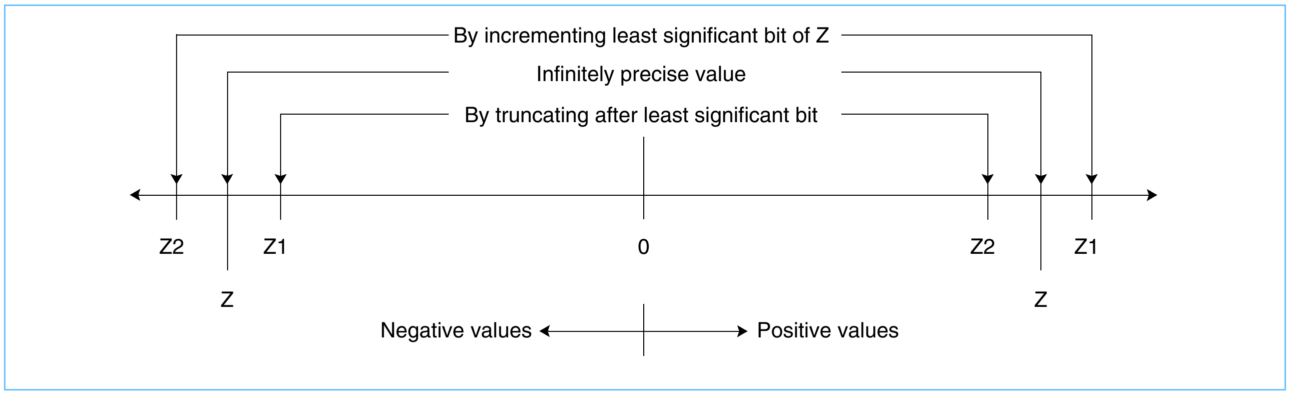
All arithmetic, rounding, and conversion instructions defined by the PowerPC Architecture (except the optional Floating Reciprocal Estimate Single (**fres***x*) and Floating Reciprocal Square Root Estimate (**frsqrte***x*) instructions) produce an intermediate result considered to be infinitely precise and with unbounded exponent range. This intermediate result is normalized or denormalized if required, and then rounded to the destination format. The final result is then placed into the target FPR in the double-precision format or in fixed-point format, depending on the instruction.

The IEEE-754 specification allows loss of accuracy to be defined as when the rounded result differs from the infinitely precise value with unbounded range (same as the definition of 'inexact'). In the PowerPC Architecture this is the way loss of accuracy is detected.

Let Z be the intermediate arithmetic result (with infinite precision and unbounded range) or the operand of a conversion operation. If Z can be represented exactly in the target format, then the result in all rounding modes is exactly Z. If Z cannot be represented exactly in the target format, let Z1 and Z2 be the next larger and next smaller numbers representable in the target format that bound Z; then Z1 or Z2 can be used to approximate the result in the target format.

Figure 3-19 shows a graphical representation of Z, Z1, and Z2 in this case.

Figure 3-19. Relation of Z1 and Z2


By incrementing least significant bit of Z
Infinitely precise value
By truncating after least significant bit
Z2
Z1
0
Z2
Z1
Z
Z
Negative values
Positive values

Four rounding modes are available through the floating-point rounding control field (RN) in the FPSCR. See *Section 2.1.4 Floating-Point Status and Control Register (FPSCR)*. These are encoded as shown in *Table 3-8*.

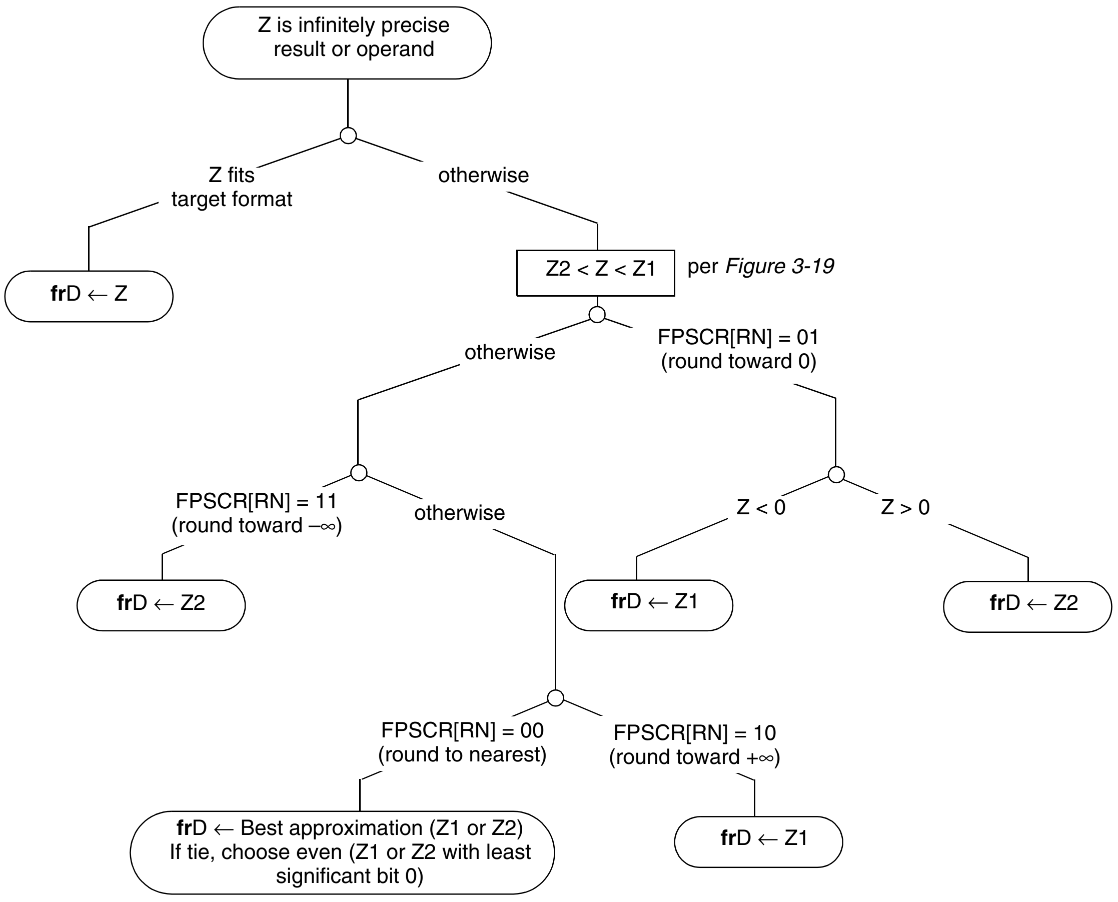
Table 3-8. FPSCR Bit Settings—RN Field

RN	Rounding Mode	Rules
00	Round to nearest	Choose the best approximation (Z1 or Z2). In case of a tie, choose the one that is even (least-significant bit 0).
01	Round toward zero	Choose the smaller in magnitude (Z1 or Z2).
10	Round toward +infinity	Choose Z1.
11	Round toward –infinity	Choose Z2.

See *Appendix C.1 Execution Model for IEEE Operations* on page 597 for a detailed explanation of rounding. Rounding occurs before an overflow condition is detected. This means that while an infinitely precise value with unbounded exponent range may be greater than the greatest representable value, the rounding mode may allow that value to be rounded to a representable value. In this case, no overflow condition occurs.

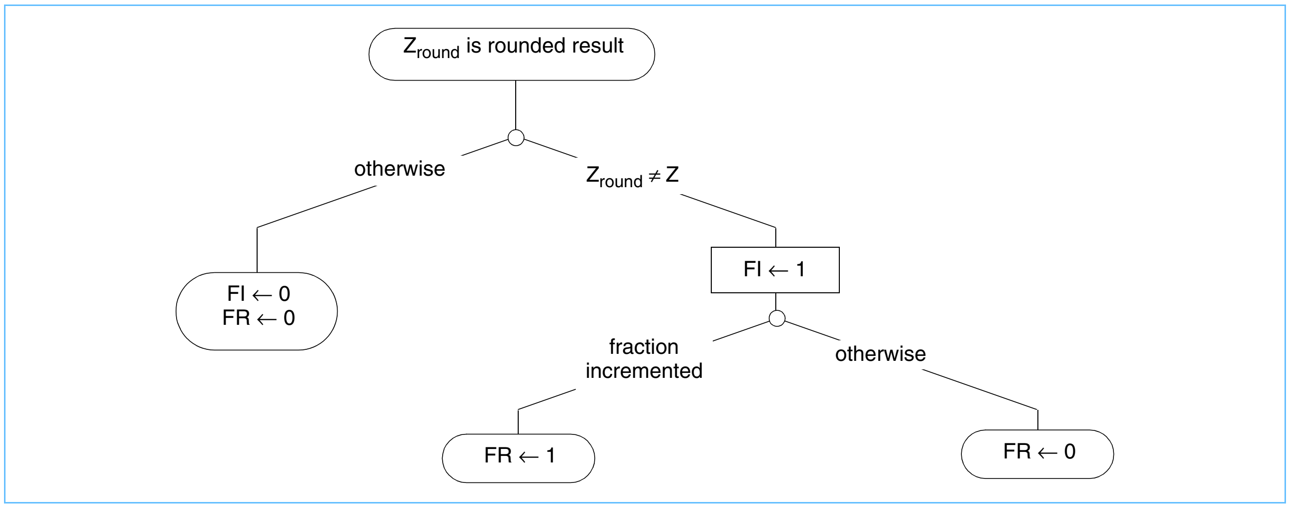
However, the underflow condition is tested before rounding. Therefore, if the value that is infinitely precise and with unbounded exponent range falls within the range of unrepresentable values, the underflow condition occurs. The results in these cases are defined in *Underflow Exception Condition* on page 116. *Figure 3-20* shows the selection of Z1 and Z2 for the four possible rounding modes that are provided by FPSCR[RN].

Figure 3-20. Selection of Z1 and Z2 for the Four Rounding Modes


Z is infinitely precise result or operand
Z fits target format
otherwise
frD ← Z
Z2 < Z < Z1
per *Figure 3-19*
otherwise
FPSCR[RN] = 01 (round toward 0)
FPSCR[RN] = 11 (round toward –∞)
otherwise
Z < 0
Z > 0
frD ← Z2
frD ← Z1
frD ← Z2
FPSCR[RN] = 00 (round to nearest)
FPSCR[RN] = 10 (round toward +∞)
frD ← Best approximation (Z1 or Z2) If tie, choose even (Z1 or Z2 with least significant bit 0)
frD ← Z1

All arithmetic, rounding, and conversion instructions affect FPSCR bits FR and FI, according to whether the rounded result is inexact (FI) and whether the fraction was incremented (FR) as shown in *Figure 3-21*. If the rounded result is inexact, FI is set and FR may be either set or cleared. If rounding does not change the result, both FR and FI are cleared. The optional **fres***x* and **frsqrte***x* instructions set FI and FR to undefined values; other floating-point instructions do not alter FR and FI.

Figure 3-21. Rounding Flags in FPSCR



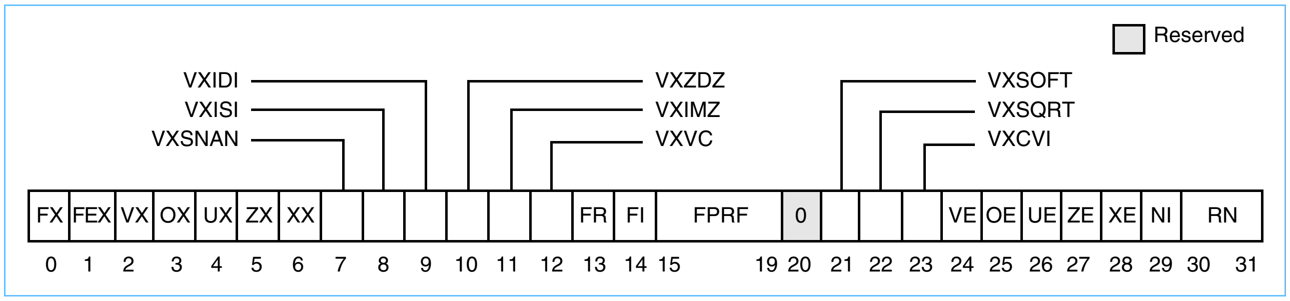
3.3.6 Floating-Point Program Exceptions

The computational instructions of the PowerPC Architecture are the only instructions that can cause floating-point enabled exceptions (subsets of the program exception).

In the processor, floating-point program exceptions are signaled by condition bits set in the floating-point status and control register (FPSCR) as described in this section and in *Chapter 2, “PowerPC Register Set*.” These bits correspond to those conditions identified as IEEE floating-point exceptions and can cause the system floating-point enabled exception error handler to be invoked. Handling for floating-point exceptions is described in *Section 6.4.9 Program Exception (0x00700)*.

The FPSCR is shown in *Figure 3-22*.

Figure 3-22. Floating-Point Status and Control Register (FPSCR)



A listing of FPSCR bit settings is shown in *Table 3-9*.

Table 3-9. FPSCR Bit Settings

Bit(s)	Name	Description
0	FX	Floating-point exception summary. Every floating-point instruction, except **mtfsfi** and **mtfsf**, implicitly sets FPSCR[FX] if that instruction causes any of the floating-point exception bits in the FPSCR to transition from 0 to 1. The **mcrfs**, **mtfsfi**, **mtfsf**, **mtfsb0**, and **mtfsb1** instructions can alter FPSCR[FX] explicitly. This is a sticky bit.
1	FEX	Floating-point enabled exception summary. This bit signals the occurrence of any of the enabled exception conditions. It is the logical OR of all the floating-point exception bits masked by their respective enable bits (FEX = (VX & VE) ^ (OX & OE) ^ (UX & UE) ^ (ZX & ZE) ^ (XX & XE)). The **mcrfs**, **mtfsf**, **mtfsfi**, **mtfsb0**, and **mtfsb1** instructions cannot alter FPSCR[FEX] explicitly. This is not a sticky bit.
2	VX	Floating-point invalid operation exception summary. This bit signals the occurrence of any invalid operation exception. It is the logical OR of all of the invalid operation exceptions. The **mcrfs**, **mtfsf**, **mtfsfi**, **mtfsb0**, and **mtfsb1** instructions cannot alter FPSCR[VX] explicitly. This is not a sticky bit.
3	OX	Floating-point overflow exception. This is a sticky bit. See *Section 3.3.6.2 Overflow, Underflow, and Inexact Exception Conditions* on page 113.
4	UX	Floating-point underflow exception. This is a sticky bit. See *Underflow Exception Condition* on page 116.
5	ZX	Floating-point zero divide exception. This is a sticky bit. See *Zero Divide Exception Condition* on page 112.
6	XX	Floating-point inexact exception. This is a sticky bit. See *Inexact Exception Condition* on page 117. FPSCR[XX] is the sticky version of FPSCR[FI]. The following rules 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 logically 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.
7	VXSNAN	Floating-point invalid operation exception for SNaN. This is a sticky bit. See *Invalid Operation Exception Condition* on page 111.
8	VXISI	Floating-point invalid operation exception for $\infty - \infty$. This is a sticky bit. See *Invalid Operation Exception Condition* on page 111.
9	VXIDI	Floating-point invalid operation exception for $\infty \div \infty$. This is a sticky bit. See *Invalid Operation Exception Condition* on page 111.
10	VXZDZ	Floating-point invalid operation exception for $0 \div 0$. This is a sticky bit. See *Invalid Operation Exception Condition* on page 111.
11	VXIMZ	Floating-point invalid operation exception for $\infty \times 0$. This is a sticky bit. See *Invalid Operation Exception Condition* on page 111.
12	VXVC	Floating-point invalid operation exception for invalid compare. This is a sticky bit. See *Invalid Operation Exception Condition* on page 111.
13	FR	Floating-point fraction rounded. The last arithmetic or rounding and conversion instruction that rounded the intermediate result incremented the fraction. See *Section 3.3.5 Rounding*. This bit is not sticky.
14	FI	Floating-point fraction inexact. The last arithmetic or rounding and conversion instruction either rounded the intermediate result (producing an inexact fraction) or caused a disabled overflow exception. See *Section 3.3.5 Rounding*. This is not a sticky bit. For more information regarding the relationship between FPSCR[FI] and FPSCR[XX], see the description of the FPSCR[XX] bit.

Table 3-9. FPSCR Bit Settings (Continued)

Bit(s)	Name	Description
15–19	FPRF	Floating-point result flags. For arithmetic, rounding, and conversion instructions, the field is based on the result placed into the target register, except that if any portion of the result is undefined, the value placed here is undefined. 15 Floating-point result class descriptor (C). Arithmetic, rounding, and conversion instructions may set this bit with the FPCC bits to indicate the class of the result as shown in *Table 3-10*. 16–19 Floating-point condition code (FPCC). Floating-point compare instructions always set one of the FPCC bits to one and the other three FPCC bits to zero. 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. 16 Floating-point less than or negative (FL or <) 17 Floating-point greater than or positive (FG or >) 18 Floating-point equal or zero (FE or =) 19 Floating-point unordered or NaN (FU or ?) **Note:** These are not sticky bits.
20	—	Reserved
21	VXSOFT	Floating-point invalid operation exception for software request. This is a sticky bit. This bit can be altered only by the **mcrfs**, **mtfsfi**, **mtfsf**, **mtfsb0**, or **mtfsb1** instructions. For more detailed information, refer to *Invalid Operation Exception Condition* on page 111.
22	VXSQRT	Floating-point invalid operation exception for invalid square root. This is a sticky bit. For more detailed information, refer to *Invalid Operation Exception Condition* on page 111. **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.
23	VXCVI	Floating-point invalid operation exception for invalid integer convert. This is a sticky bit. See *Invalid Operation Exception Condition* on page 111.
24	VE	Floating-point invalid operation exception enable. See *Invalid Operation Exception Condition* on page 111.
25	OE	IEEE floating-point overflow exception enable. See *Section 3.3.6.2 Overflow, Underflow, and Inexact Exception Conditions* on page 113.
26	UE	IEEE floating-point underflow exception enable. See *Underflow Exception Condition* on page 116.
27	ZE	IEEE floating-point zero divide exception enable. See *Zero Divide Exception Condition* on page 112.
28	XE	Floating-point inexact exception enable. See *Inexact Exception Condition* on page 117.
29	NI	Floating-point non-IEEE mode. If this bit is set, results need not conform with IEEE standards and the other FPSCR bits may have meanings other than those described here. If the bit is set and if all implementation-specific requirements are met and if an IEEE-conforming result of a floating-point operation would be a denormalized number, then the result produced is zero (retaining the sign of the denormalized number). Any other effects associated with setting this bit are described in the user's manual for the implementation (the effects are implementation-dependent). **Note:** When the processor is in floating-point non-IEEE mode, the results of floating-point operations may be approximate, and performance for these operations may be better, more predictable, or less data-dependent than when the processor is not in non-IEEE mode. For example, in non-IEEE mode an implementation may return 0 instead of a denormalized number, and may return a large number instead of an infinity.
30–31	RN	Floating-point rounding control. See *Section 3.3.5 Rounding*. 00 Round to nearest 01 Round toward zero 10 Round toward +infinity 11 Round toward –infinity

Table 3-10 illustrates the floating-point result flags used by PowerPC processors. The result flags correspond to FPSCR bits [15–19] (the FPRF field).

Table 3-10. Floating-Point Result Flags — FPSCR[FPRF]

Result Flags (Bits [15–19])					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

The following conditions that can cause program exceptions are detected by the processor. These conditions may occur during execution of computational floating-point instructions. The corresponding bits set in the FPSCR are indicated in parentheses:

- Invalid operation exception condition (VX)
  - SNaN condition (VXSNAN)
  - Infinity – infinity condition (VXISI)
  - Infinity ÷ infinity condition (VXIDI)
  - Zero ÷ zero condition (VXZDZ)
  - Infinity × zero condition (VXIMZ)
  - Invalid compare condition (VXVC)
  - Software request condition (VXSOFT)
  - Invalid integer convert condition (VXCVI)
  - Invalid square root condition (VXSQRT)

  These exception conditions are described in *Invalid Operation Exception Condition* on page 111.
- Zero divide exception condition (ZX). These exception conditions are described in *Zero Divide Exception Condition* on page 112.
- Overflow Exception Condition (OX). These exception conditions are described in *Overflow Exception Condition* on page 115.
- Underflow Exception Condition (UX). These exception conditions are described in *Underflow Exception Condition* on page 116.
- Inexact Exception Condition (XX). These exception conditions are described in *Inexact Exception Condition* on page 117.

Each floating-point exception condition and each category of invalid IEEE floating-point operation exception condition has a corresponding exception bit in the FPSCR which indicates the occurrence of that condition. Generally, the occurrence of an exception condition depends only on the instruction and its arguments (with one deviation, described below). When one or more exception conditions arise during the execution of an instruction, the way in which the instruction completes execution depends on the value of the IEEE floating-point enable bits in the FPSCR which govern those exception conditions. If no governing enable bit is set to 1, the instruction delivers a default result. Otherwise, specific condition bits and the FX bit in the FPSCR are set and instruction execution is completed by suppressing or delivering a result. Finally, after the instruction execution has completed, a nonzero FX bit in the FPSCR causes a program exception if either FE0 or FE1 is set in the MSR (invoking the system error handler). The values in the FPRs immediately after the occurrence of an enabled exception do not depend on the FE0 and FE1 bits.

The floating-point exception summary bit (FX) in the FPSCR is set by any floating-point instruction (except **mtfsfi** and **mtfsf**) that causes any of the exception bits in the FPSCR to change from 0 to 1, or by **mtfsfi**, **mtfsf**, and **mtfsb1** instructions that explicitly set one of these bits. FPSCR[FEX] is set when any of the exception condition bits is set and the exception is enabled (enable bit is one).

A single instruction may set more than one exception condition bit only in the following cases:

- The inexact exception condition bit (FPSCR[XX]) may be set with the overflow exception condition bit (FPSCR[OX]).
- The inexact exception condition bit (FPSCR[XX]) may be set with the underflow exception condition bit (FPSCR[UX]).
- The invalid IEEE floating-point operation exception condition bit (SNaN) may be set with invalid IEEE floating-point operation exception condition bit ($\infty \times 0$) (FPSCR[VXIMZ]) for multiply-add instructions.
- The invalid operation exception condition bit (SNaN) may be set with the invalid IEEE floating-point operation exception condition bit (invalid compare) (FPRSC[VXVC]) for compare ordered instructions.
- The invalid IEEE floating-point operation exception condition bit (SNaN) may be set with the invalid IEEE floating-point operation exception condition bit (invalid integer convert) (FPSCR[VXCVI]) for convert-to-integer instructions.

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

- Enabled invalid IEEE floating-point operation
- Enabled zero divide

For the remaining kinds of exception conditions, a result is generated and written to the destination specified by the instruction causing the exception condition. The result may depend on whether the condition is enabled or disabled. The kinds of exception conditions that deliver a result are the following:

- Disabled invalid IEEE floating-point operation
- Disabled zero divide
- Disabled overflow
- Disabled underflow
- Disabled inexact
- Enabled overflow
- Enabled underflow
- Enabled inexact

Subsequent sections define each of the floating-point exception conditions and specify the action taken when they are detected.

The IEEE standard specifies the handling of exception conditions in terms of traps and trap handlers. In the PowerPC Architecture, an FPSCR exception enable bit being set causes generation of the result value specified in the IEEE standard for the trap enabled case—the expectation is that the exception is detected by software, which will revise the result. An FPSCR exception enable bit of 0 causes generation of the default result value specified for the trap disabled (or no trap occurs or trap is not implemented) case—the expectation is that the exception will not be detected by software, which will simply use the default result. The result to be delivered in each case for each exception is described in the following sections.

The IEEE default behavior when an exception occurs, which is to generate a default value and not to notify software, is obtained by clearing all FPSCR exception enable bits and using ignore exceptions mode (see *Table 3-11*). In this case the system floating-point enabled exception error handler is not invoked, even if floating-point exceptions occur. If necessary, software can inspect the FPSCR exception bits to determine whether exceptions have occurred.

If the system error handler is to be invoked, the corresponding FPSCR exception enable bit must be set and a mode other than ignore exceptions mode must be used. In this case the system floating-point enabled exception error handler is invoked if an enabled floating-point exception condition occurs.

Whether and how the system floating-point enabled exception error handler is invoked if an enabled floating-point exception occurs is controlled by MSR bits [FE0] and [FE1] as shown in *Table 3-11*. (The system floating-point enabled exception error handler is never invoked if the appropriate floating-point exception is disabled.)

Table 3-11. MSR[FE0] and MSR[FE1] Bit Settings for FP Exceptions

FE0	FE1	Description
0	0	Ignore exceptions mode—Floating-point exceptions do not cause the program exception error handler to be invoked.
0	1	Imprecise nonrecoverable mode—When an exception occurs, the exception handler is invoked at some point at or beyond the instruction that caused the exception. It may not be possible to identify the offending instruction or the data that caused the exception. Results from the offending instruction may have been used by or affected subsequent instructions executed before the exception handler was invoked.
1	0	Imprecise recoverable mode— When an enabled exception occurs, the floating-point enabled exception handler is invoked at some point at or beyond the instruction that caused the exception. Sufficient information is provided to the exception handler that it can identify the offending instruction and correct any faulty results. In this mode, no results caused by the offending instruction have been used by or affected subsequent instructions that are executed before the exception handler is invoked. Running in this mode might cause a degradation in performance.
1	1	Precise mode—The system floating-point enabled exception error handler is invoked precisely at the instruction that caused the enabled exception. The architecture ensures that all instructions logically residing before the excepting instruction have completed and no instruction after the excepting instruction has been executed. Running in this mode might cause a degradation in performance.

In either of the imprecise modes, an FPSCR instruction can be used to force the occurrence of any invocations of the floating-point enabled exception handler, due to instructions initiated before the FPSCR instruction. This forcing has no effect in ignore exceptions mode and is superfluous for precise mode.

In all cases, the question of whether a floating-point result is stored, and what value is stored, is governed by the FPSCR exception enable bits, and is not affected by the value of the FE0 and FE1 bits. For the best performance across the widest range of implementations, the following guidelines should be considered:

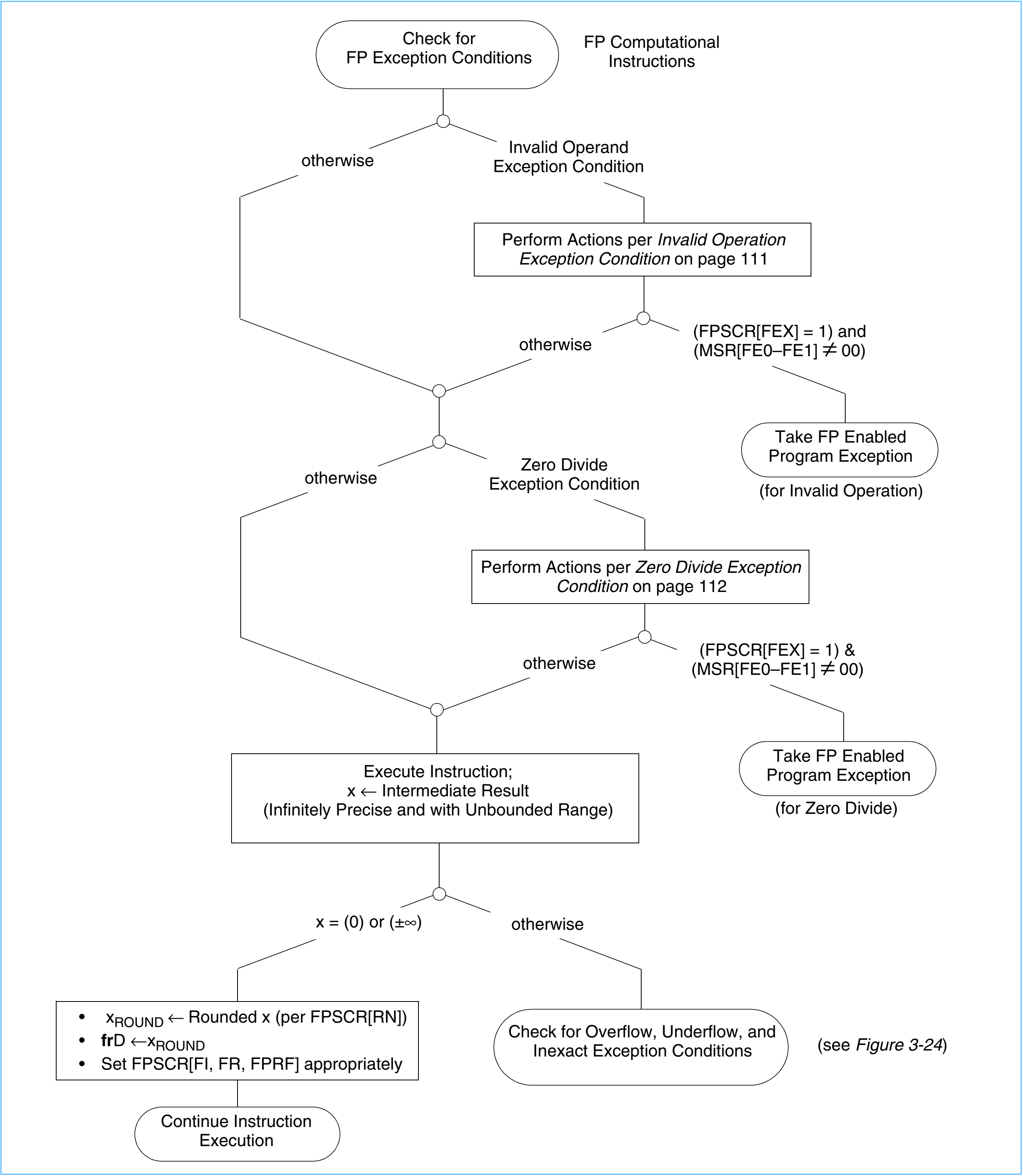
- If the IEEE default results are acceptable to the application, FE0 and FE1 should be cleared (ignore exceptions mode). All FPSCR exception enable bits should be cleared.
- If the IEEE default results are unacceptable to the application, an imprecise mode should be used with the FPSCR enable bits set as needed.

- Ignore exceptions mode should not, in general, be used when any FPSCR exception enable bits are set.
- Precise mode may degrade performance in some implementations, perhaps substantially, and therefore should be used only for debugging and other specialized applications.

3.3.6.1 Invalid Operation and Zero Divide Exception Conditions

The flow diagram in *Figure 3-23* shows the initial flow for checking floating-point exception conditions (invalid operation and divide by zero conditions). In any of these cases of floating-point exception conditions, if the FPSCR[FEX] bit is set (implicitly) and MSR[FE0–FE1] ≠ ‘00’, the processor takes a program exception (floating-point enabled exception type). Refer to *Chapter 6, Exceptions* for more information on exception processing. The actions performed for each floating-point exception condition are described in greater detail in the following sections.

Figure 3-23. Initial Flow for Floating-Point Exception Conditions


Check for FP Exception Conditions
FP Computational Instructions
otherwise
Invalid Operand Exception Condition
Perform Actions per *Invalid Operation Exception Condition* on page 111
otherwise
(FPSCR[FEX] = 1) and (MSR[FE0–FE1] ≠ 00)
Take FP Enabled Program Exception
(for Invalid Operation)
otherwise
Zero Divide Exception Condition
Perform Actions per *Zero Divide Exception Condition* on page 112
otherwise
(FPSCR[FEX] = 1) & (MSR[FE0–FE1] ≠ 00)
Take FP Enabled Program Exception
(for Zero Divide)
Execute Instruction; x ← Intermediate Result (Infinitely Precise and with Unbounded Range)
x = (0) or (±∞)
otherwise
• x_{ROUND} ← Rounded x (per FPSCR[RN])
• **fr**D ← x_{ROUND}
• Set FPSCR[FI, FR, FPRF] appropriately
Continue Instruction Execution
Check for Overflow, Underflow, and Inexact Exception Conditions
(see *Figure 3-24*)

Invalid Operation Exception Condition

An invalid operation exception occurs when an operand is invalid for the specified operation. The invalid operations are as follows:

- Any operation except load, store, move, select, or **mtfsf** on a signaling NaN (SNaN)
- For add or subtract operations, magnitude subtraction of infinities ($\infty - \infty$)
- Division of infinity by infinity ($\infty \div \infty$)
- Division of zero by zero ($0 \div 0$)
- Multiplication of infinity by zero ($\infty \times 0$)
- Ordered comparison involving a NaN (invalid compare)
- Square root or reciprocal square root of a negative, nonzero number (invalid square root)

  Note: If the implementation does not support the optional floating-point square root or floating-point reciprocal square root estimate instructions, software can simulate the instruction and set the FPSCR[VXSQRT] bit to reflect the exception.
- Integer convert involving a number that is too large in magnitude to be represented in the target format, or involving an infinity or a NaN (invalid integer convert)

FPSCR[VXSOFT] allows 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. This allows PowerPC instructions not implemented in hardware to be emulated.

Any time an invalid operation occurs or software explicitly requests the exception via FPSCR[VXSOFT], (regardless of the value of FPSCR[VE]), the following actions are taken:

- One or two invalid operation exception condition bits is set

FPSCR[VXSNAN]	(if SNaN)
FPSCR[VXISI]	(if $\infty - \infty$)
FPSCR[VXIDI]	(if $\infty \div \infty$)
FPSCR[VXZDZ]	(if $0 \div 0$)
FPSCR[VXIMZ]	(if $\infty \times 0$)
FPSCR[VXVC]	(if invalid comparison)
FPSCR[VXSOFT]	(if software request)
FPSCR[VXSQRT]	(if invalid square root)
FPSCR[VXCVI]	(if invalid integer convert)

- If the operation is a compare,
  FPSCR[FR, FI, C] are unchanged
  FPSCR[FPCC] is set to reflect unordered
- If software explicitly requests the exception,
  FPSCR[FR, FI, FPRF] are as set by the **mtfsfi**, **mtfsf**, or **mtfsb1** instruction.

There are additional actions performed that depend on the value of FPSCR[VE]. These are described in *Table 3-12*.

Table 3-12. Additional Actions Performed for Invalid FP Operations

Invalid Operation	Result Category	Action Performed	
		FPSCR[VE] = ’1’	FPSCR[VE] = ’0’
Arithmetic or floating-point round to single	**fr**D	Unchanged	QNaN
	FPSCR[FR, FI]	Cleared	Cleared
	FPSCR[FPRF]	Unchanged	Set for QNaN
Convert to 64-bit integer (positive number or +∞)	**fr**D[0–63]	Unchanged	Most positive 64-bit integer value
	FPSCR[FR, FI]	Cleared	Cleared
	FPSCR[FPRF]	Unchanged	Undefined
Convert to 64-bit integer (negative number, NaN, or –∞)	**fr**D[0–63]	Unchanged	Most negative 64-bit integer value
	FPSCR[FR, FI]	Cleared	Cleared
	FPSCR[FPRF]	Unchanged	Undefined
Convert to 32-bit integer (positive number or +∞)	frD[0–31]	Unchanged	Undefined
	frD[32–63]	Unchanged	Most positive 32-bit integer value
	FPSCR[FR, FI]	Cleared	Cleared
	FPSCR[FPRF]	Unchanged	Undefined
Convert to 32-bit integer (negative number, NaN, or –∞)	**fr**D[0–31]	Unchanged	Undefined
	frD[32–63]	Unchanged	Most negative 32-bit integer value
	FPSCR[FR, FI]	Cleared	Cleared
	FPSCR[FPRF]	Unchanged	Undefined
All cases	FPSCR[FEX]	Implicitly set (causes exception)	Unchanged

Zero Divide Exception Condition

A zero divide exception condition occurs when a divide instruction is executed with a zero divisor value and a finite, nonzero dividend value or when an floating reciprocal estimate single (**fres**) or a floating reciprocal square root estimate (**frsqrte**) instruction is executed with a zero operand value.

The corresponding result is infinity, where the sign is the sign of the source value, as follows:

- $1/+0.0 \rightarrow +\infty$
- $1/-0.0 \rightarrow -\infty$
- $1/(\sqrt{+0.0}) \rightarrow +\infty$
- $1/(\sqrt{-0.0}) \rightarrow -\infty$

When a zero divide condition occurs, the following actions are taken:

- Zero divide exception condition bit is set FPSCR[ZX] = ’1’ .
- FPSCR[FR, FI] are cleared.

Additional actions depend on the setting of the zero divide exception condition enable bit, FPSCR[ZE], as described in *Table 3-13*.

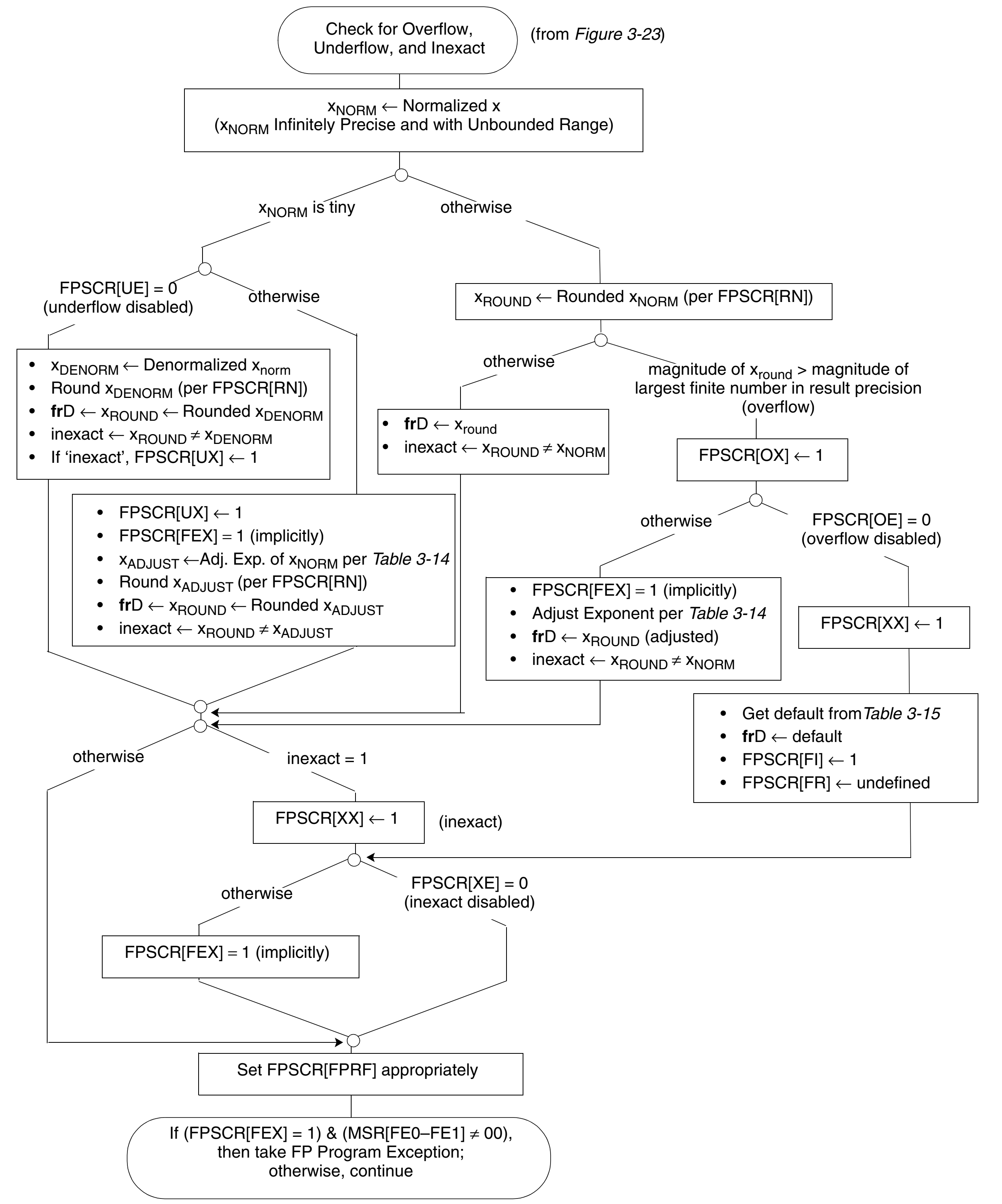
Table 3-13. Additional Actions Performed for Zero Divide

Result Category	Action Performed	
	FPSCR[ZE] = ’1’	FPSCR[ZE] = ’0’
frD	Unchanged	±∞ (sign determined by XOR of the signs of the operands)
FPSCR[FEX]	Implicitly set (causes exception)	Unchanged
FPSCR[FPRF]	Unchanged	Set to indicate ±∞

3.3.6.2 Overflow, Underflow, and Inexact Exception Conditions

As described earlier, the overflow, underflow, and inexact exception conditions are detected after the floating-point instruction has executed and an infinitely precise result with unbounded range has been computed. *Figure 3-24* shows the flow for the detection of these conditions and is a continuation of *Figure 3-23*. As in the cases of invalid operation, or zero divide conditions, if the FPSCR[FEX] bit is implicitly set as described in *Table 3-9* and MSR[FE0–FE1] ≠ 00, the processor takes a program exception (floating-point enabled exception type). Refer to *Chapter 6, Exceptions* for more information on exception processing. The actions performed for each of these floating-point exception conditions (including the generated result) are described in greater detail in the following sections.

Figure 3-24. Checking of Remaining Floating-Point Exception Conditions


Check for Overflow, Underflow, and Inexact
(from *Figure 3-23*)
$x_{NORM} \leftarrow$ Normalized x
(x_{NORM} Infinitely Precise and with Unbounded Range)
x_{NORM} is tiny
otherwise
FPSCR[UE] = 0 (underflow disabled)
otherwise
$x_{ROUND} \leftarrow$ Rounded x_{NORM} (per FPSCR[RN])
• $x_{DENORM} \leftarrow$ Denormalized x_{norm}
• Round x_{DENORM} (per FPSCR[RN])
• **fr**D $\leftarrow x_{ROUND} \leftarrow$ Rounded x_{DENORM}
• inexact $\leftarrow x_{ROUND} \neq x_{DENORM}$
• If 'inexact', FPSCR[UX] $\leftarrow$ 1
otherwise
magnitude of x_{round} > magnitude of largest finite number in result precision (overflow)
• **fr**D $\leftarrow x_{round}$
• inexact $\leftarrow x_{ROUND} \neq x_{NORM}$
FPSCR[OX] $\leftarrow$ 1
• FPSCR[UX] $\leftarrow$ 1
• FPSCR[FEX] = 1 (implicitly)
• $x_{ADJUST} \leftarrow$Adj. Exp. of x_{NORM} per *Table 3-14*
• Round x_{ADJUST} (per FPSCR[RN])
• **fr**D $\leftarrow x_{ROUND} \leftarrow$ Rounded x_{ADJUST}
• inexact $\leftarrow x_{ROUND} \neq x_{ADJUST}$
otherwise
FPSCR[OE] = 0 (overflow disabled)
• FPSCR[FEX] = 1 (implicitly)
• Adjust Exponent per *Table 3-14*
• **fr**D $\leftarrow x_{ROUND}$ (adjusted)
• inexact $\leftarrow x_{ROUND} \neq x_{NORM}$
FPSCR[XX] $\leftarrow$ 1
• Get default from *Table 3-15*
• **fr**D $\leftarrow$ default
• FPSCR[FI] $\leftarrow$ 1
• FPSCR[FR] $\leftarrow$ undefined
otherwise
inexact = 1
FPSCR[XX] $\leftarrow$ 1
(inexact)
otherwise
FPSCR[XE] = 0 (inexact disabled)
FPSCR[FEX] = 1 (implicitly)
Set FPSCR[FPRF] appropriately
If (FPSCR[FEX] = 1) & (MSR[FE0–FE1] $\neq$ 00), then take FP Program Exception; otherwise, continue

Overflow Exception Condition

Overflow occurs when the magnitude of what would have been the rounded result (had the exponent range been unbounded) is greater than the magnitude of the largest finite number of the specified result precision. Regardless of the setting of the overflow exception condition enable bit of the FPSCR, the overflow exception condition bit is set FPSCR[OX] = ’1’ .

Additional actions are taken that depend on the setting of the overflow exception condition enable bit of the FPSCR as described in *Table 3-14*.

Table 3-14. Additional Actions Performed for Overflow Exception Condition

Condition	Result Category	Action Performed	
		FPSCR[OE] = ’1’	FPSCR[OE] = ’0’
Double-precision arithmetic instructions	Exponent of normalized intermediate result	Adjusted by subtracting 1536	—
Single-precision arithmetic and **frsp***x* instruction	Exponent of normalized intermediate result	Adjusted by subtracting 192	—
All cases	**fr**D	Rounded result (with adjusted exponent)	Default result per *Table 3-15*
	FPSCR[XX]	Set if rounded result differs from intermediate result	Set
	FPSCR[FEX]	Implicitly set (causes exception)	Unchanged
	FPSCR[FPRF]	Set to indicate ±normal number	Set to indicate ±∞ or ±normal number
	FPSCR[FI]	Reflects rounding	Set
	FPSCR[FR]	Reflects rounding	Undefined

When the overflow exception condition is disabled (FPSCR[OE] = ’0’) and an overflow condition occurs, the default result is determined by the rounding mode bit (FPSCR[RN]) and the sign of the intermediate result as shown in *Table 3-15*.

Table 3-15. Target Result for Overflow Exception Disabled Case

FPSCR[RN]	Sign of Intermediate Result	frD
Round to nearest	Positive	+Infinity
	Negative	–Infinity
Round toward zero	Positive	Format’s largest finite positive number
	Negative	Format’s most negative finite number
Round toward +infinity	Positive	+Infinity
	Negative	Format’s most negative finite number
Round toward –infinity	Positive	Format’s largest finite positive number
	Negative	–Infinity

Underflow Exception Condition

The underflow exception condition 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 the rounded result is inexact.
  In this context, the term 'tiny' refers to a floating-point value that is too small to be represented for a particular precision format.

As shown in *Figure 3-24*, a tiny result is detected before rounding, when a nonzero intermediate result value computed as though it had infinite precision and unbounded exponent range is less in magnitude than the smallest normalized number.

If the intermediate result is tiny and the underflow exception condition enable bit is cleared (FPSCR[UE] = '0'), the intermediate result is denormalized (see *Section 3.3.3 Normalization and Denormalization*) and rounded (see *Section 3.3.5 Rounding*) before being stored in an FPR. In this case, if the rounding causes the delivered result value to differ from what would have been computed were both the exponent range and precision unbounded (the result is inexact), then underflow occurs and FPSCR[UX] is set.

The actions performed for underflow exception conditions are described in *Table 3-16*.

Table 3-16. Actions Performed for Underflow Conditions

Condition	Result Category	Action Performed	
		FPSCR[UE] = '1'	FPSCR[UE] = '0'
Double-precision arithmetic instructions	Exponent of normalized intermediate result	Adjusted by adding 1536	—
Single-precision arithmetic and **frsp***x* instructions	Exponent of normalized intermediate result	Adjusted by adding192	—
All cases	**fr**D	Rounded result (with adjusted exponent)	Denormalized and rounded result
	FPSCR[XX]	Set if rounded result differs from intermediate result	Set if rounded result differs from intermediate result
	FPSCR[UX]	Set	Set only if tiny and inexact after denormalization and rounding
	FPSCR[FPRF]	Set to indicate ±normalized number	Set to indicate ±denormalized number or ±zero
	FPSCR[FEX]	Implicitly set (causes exception)	Unchanged
	FPSCR[FI]	Reflects rounding	Reflects rounding
	FPSCR[FR]	Reflects rounding	Reflects rounding

Note: The FR and FI bits in the FPSCR allow the system floating-point enabled exception error handler, when invoked because of an underflow exception condition, to simulate a trap disabled environment. That is, the FR and FI bits allow the system floating-point enabled exception error handler to unround the result, thus allowing the result to be denormalized.

Inexact Exception Condition

The inexact exception condition occurs when one of two conditions occur during rounding:

- The rounded result differs from the intermediate result assuming the intermediate result exponent range and precision to be unbounded. (In the case of an enabled overflow or underflow condition, where the exponent of the rounded result is adjusted for those conditions, an inexact condition occurs only if the significand of the rounded result differs from that of the intermediate result.)
- The rounded result overflows and the overflow exception condition is disabled.

When an inexact exception condition occurs, the following actions are taken independent of the setting of the inexact exception condition enable bit of the FPSCR:

- Inexact exception condition bit in the FPSCR is set FPSCR[XX] = ’1’ .
- The rounded or overflowed result is placed into the target FPR.
- FPSCR[FPRF] is set to indicate the class and sign of the result.

In addition, if the inexact exception condition enable bit in the FPSCR (FPSCR[XE]) is set, and an inexact condition exists, then the FPSCR[FEX] bit is implicitly set, causing the processor to take a floating-point enabled program exception.

In PowerPC implementations, running with inexact exception conditions enabled may have greater latency than enabling other types of floating-point exception conditions.

THIS PAGE INTENTIONALLY LEFT BLANK

4. Addressing Modes and Instruction Set Summary

This chapter describes instructions and addressing modes defined by the three levels of the PowerPC Architecture—user instruction set architecture (UISA), virtual environment architecture (VEA), and operating environment architecture (OEA). These instructions are divided into the following functional categories:

- Integer instructions—These include arithmetic and logical instructions. For more information, see *Section 4.2.1 Integer Instructions*.
- Floating-point instructions—These include floating-point arithmetic instructions, as well as instructions that affect the floating-point status and control register (FPSCR). For more information, see *Section 4.2.2 Floating-Point Instructions*.
- Load and store instructions—These include integer and floating-point load and store instructions. For more information, see *Section 4.2.3 Load and Store Instructions*.
- Flow control instructions—These include branching instructions, condition register logical instructions, trap instructions, and other instructions that affect the instruction flow. For more information, see *Section 4.2.4 Branch and Flow Control Instructions*.
- Processor control instructions—These instructions are used for synchronizing memory accesses and managing of caches, TLBs, and the segment registers. For more information, see *Section 4.2.5 Processor Control Instructions—UISA*, *Section 4.3.1 Processor Control Instructions—VEA*, and *Section 4.4.2 Processor Control Instructions—OEA*.
- Memory synchronization instructions—These instructions control the order in which memory operations are completed with respect to asynchronous events, and the order in which memory operations are seen by other processors or memory access mechanisms. For more information, see *Section 4.2.6 Memory Synchronization Instructions—UISA*, and *Section 4.3.2 Memory Synchronization Instructions—VEA*.
- Memory control instructions—These include cache management instructions (user-level and supervisor-level), segment register manipulation instructions, and translation lookaside buffer management instructions. For more information, see *Section 4.3.3 Memory Control Instructions—VEA*, and *Section 4.4.3 Memory Control Instructions—OEA*.

  Note: User-level and supervisor-level are referred to as problem state and privileged state, respectively, in the architecture specification.
- External control instructions—These instructions allow a user-level program to communicate with a special-purpose device. For more information, see *Section 4.3.4 External Control Instructions*.

This grouping of instructions does not necessarily indicate the execution unit that processes a particular instruction or group of instructions within a processor implementation.

Integer instructions operate on byte, halfword, word, and doubleword operands. Floating-point instructions operate on single-precision and double-precision floating-point operands. The PowerPC Architecture uses instructions that are four bytes long and word-aligned. It provides for byte, halfword, word, and doubleword operand fetches and stores between memory and a set of 32 general-purpose registers (GPRs). It also provides for word and doubleword operand fetches and stores between memory and a set of 32 floating-point registers (FPRs). The FPRs and GPRs are 64 bits wide in all PowerPC implementations.

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

The description of each instruction includes the mnemonic and a formatted list of operands. PowerPC-compliant assemblers support the mnemonics and operand lists. To simplify assembly language programming, a set of simplified mnemonics (referred to as extended mnemonics in the architecture specification) and symbols is provided for some of the most frequently-used instructions; see *Appendix E Simplified Mnemonics*, for a complete list of simplified mnemonics.

The instructions are organized by functional categories while maintaining the delineation of the three levels of the PowerPC Architecture—UISA, VEA, and OEA; *Section 4.2 PowerPC UISA Instructions* discusses the UISA instructions, followed by *Section 4.3 PowerPC VEA Instructions* that discusses the VEA instructions and *Section 4.4 PowerPC OEA Instructions* that discusses the OEA instructions. See *Section 1.1.2 Levels of the PowerPC Architecture* for more information about the various levels defined by the PowerPC Architecture.

4.1 Conventions

This section describes conventions used for the PowerPC instruction set. Descriptions of computation modes, memory addressing, synchronization, and the PowerPC exception summary follow.

4.1.1 Sequential Execution Model

The PowerPC processors appear to execute instructions in program order, regardless of asynchronous events or program exceptions. The execution of a sequence of instructions may be interrupted by an exception caused by one of the instructions in the sequence, or by an asynchronous event.

Note: The architecture specification refers to exceptions as interrupts.

For exceptions to the sequential execution model, refer to *Chapter 6, Exceptions*. For information about the synchronization required when using store instructions to access instruction areas of memory, refer to *Section 4.2.3.3 Integer Store Instructions* and *Section 5.1.5.2 Instruction Cache Instructions*. For information regarding instruction fetching, and for information about guarded memory refer to *Section 5.2.1.5 Guarded Attribute (G)*.

4.1.2 Computation Modes

The general-purpose and floating-point registers, and some special-purpose registers (SPRs) are 64 bits long, with an effective address of 64 bits. All 64-bit implementations have two modes of operation: 64-bit mode (which is the default) and 32-bit mode. The mode controls how the effective address is interpreted, how condition bits are set, and how the count register (CTR) is tested by branch conditional instructions. All instructions provided for 64-bit implementations are available in both 64 and 32-bit modes.

The machine state register bit [0], MSR[SF], is used to choose between 64 and 32-bit modes. When MSR[SF = ‘0’, the processor runs in 32-bit mode, and when MSR[SF] = ’1’ the processor runs in the default 64-bit mode.

In both 64-bit mode (the default) and 32-bit mode of a 64-bit implementation, instructions that set a 64-bit register affect all 64 bits, and the value placed into the register is independent of mode. In both modes, effective address computations use all 64 bits of the relevant registers (GPRs, LR, CTR, etc.), and produce a 64-bit result; however, in 32-bit mode (MSR[SF] = ’0’), only the low-order 32 bits of the computed effective address are used to address memory.

4.1.3 Classes of Instructions

PowerPC instructions belong to one of the following three classes:

- Defined
- Illegal
- Reserved

Note: While the definitions of these terms are consistent among the PowerPC processors, the assignment of these classifications is not. For example, an instruction that is specific to 64-bit implementations is considered defined for 64-bit implementations, but illegal for 32-bit implementations.

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

In future versions of the PowerPC Architecture, instruction codings that are now illegal may become defined (by being added to the architecture) or reserved (by being assigned to one of the special purposes). Likewise, reserved instructions may become defined.

4.1.3.1 Definition of Boundedly Undefined

The results of executing a given instruction are said to be boundedly undefined if they could have been achieved by executing an arbitrary 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.

4.1.3.2 Defined Instruction Class

Defined instructions contain all the instructions defined in the PowerPC UISA, VEA, and OEA. Defined instructions are guaranteed to be supported in all PowerPC implementations. The only exceptions are instructions that are defined only for 64-bit implementations, instructions that are defined only for 32-bit implementations, and optional instructions, as stated in the instruction descriptions in *Chapter 8, Instruction Set*. A PowerPC processor may invoke the illegal instruction error handler (part of the program exception handler) when an unimplemented PowerPC instruction is encountered so that it may be emulated in software, as required.

A defined instruction can have preferred and/or invalid forms, as described in the following sections.

Preferred Instruction Forms

A defined instruction may have an instruction form that is preferred (that is, the instruction will execute in an efficient manner). Any form other than the preferred form will take significantly longer to execute. The following instructions have preferred forms:

- Condition register logical instructions
- Load/store multiple instructions
- Load/store string instructions
- Or immediate instruction (preferred form of no-op)
- Move to condition register fields instruction

Invalid Instruction Forms

A defined instruction may have an instruction form that is invalid if one or more operands, excluding the opcodes and reserved fields, are coded incorrectly in a manner that can be deduced by examining only the instruction encoding (primary and extended opcodes). Attempting to execute an invalid form of an instruction either invokes the illegal instruction error handler (a program exception) or yields boundedly-undefined results. See *Chapter 8, Instruction Set*, for individual instruction descriptions.

Invalid forms result when a bit or operand is coded incorrectly, for example, or when a reserved bit (shown as ‘0’) is coded as ‘1’.

The following instructions have invalid forms identified in their individual instruction descriptions:

- Branch conditional instructions
- Load/store with update instructions
- Load multiple instructions
- Load string instructions
- Load/store floating-point with update instructions

Optional Instructions

A defined instruction may be optional. The optional instructions fall into the following categories:

- General-purpose instructions—**fsqrt** and **fsqrts**
- Graphics instructions—**fres**, **frsqrte**, and **fsel**
- External control instructions—**eciwx** and **ecowx**
- Lookaside buffer management instructions—**slbia**, **slbie**, **tlbia**, **tlbie**, **tlbiel**, and **tlbsync** (with conditions, see *Chapter 8, Instruction Set* for more information)

TEMPORARY 64-BIT BRIDGE

The optional 64-bit bridge facility has three other categories of optional instructions for 64-bit implementations. These are described in greater detail in *Section 7.6 Migration of Operating Systems from 32-Bit Implementations to 64-Bit Implementations* and summarized below:

- 32-bit segment register support instructions—**mtsr**, **mtsrin**, **mfsr**, and **mfsrin**
- 32-bit system linkage instructions—**mtmsr**

Any attempt to execute an optional instruction that is not provided by the implementation will cause the illegal instruction error handler to be invoked. Exceptions to this rule are stated in the instruction descriptions found in *Chapter 8, Instruction Set*.

4.1.3.3 Illegal Instruction Class

Illegal instructions can be grouped into the following categories:

- Instructions that are not implemented in the PowerPC Architecture. These opcodes are available for future extensions of the PowerPC Architecture; that is, future versions of the PowerPC Architecture may define any of these instructions to perform new functions. The following primary opcodes are defined as illegal but may be used in future extensions to the architecture:

  1, 4, 5, 6, 56, 57, 60, 61

  Note: Opcode 4 may be used by the vector instructions as described in the *PowerPC Microprocessor Family: AltiVec Technology Programming Environments Manual.*

- All unused extended opcodes are illegal. The unused extended opcodes can be determined from information in *Appendix A.2 Instructions Sorted by Opcode* and *Section 4.1.3.4 Reserved Instructions*. The following primary opcodes have unused extended opcodes.

  19, 30, 31, 56, 57, 58, 59, 60, 61, 62, 63

- An instruction consisting entirely of zeros is guaranteed to be an illegal instruction. This increases the probability that an attempt to execute data or uninitialized memory invokes the illegal instruction error handler (a program exception).

Note: If only the primary opcode consists of all zeros, the instruction is considered a reserved instruction, as described in *Section 4.1.3.4 Reserved Instructions*.

An attempt to execute an illegal instruction invokes the illegal instruction error handler (a program exception) but has no other effect. See *Section 6.4.9 Program Exception (0x00700)* for additional information about illegal instruction exception.

With the exception of the instruction consisting entirely of binary zeros, the illegal instructions are available for further additions to the PowerPC Architecture.

4.1.3.4 Reserved Instructions

Reserved instructions are allocated to specific implementation-dependent purposes not defined by the PowerPC Architecture. An attempt to execute an unimplemented reserved instruction invokes the illegal instruction error handler (a program exception). See *Section 6.4.9 Program Exception (0x00700)* for additional information about illegal instruction exception.

The following types of instructions are included in this class:

1. Instructions for the POWER architecture that have not been included in the PowerPC Architecture.
2. Implementation-specific instructions used to conform to the PowerPC Architecture specifications. For example, the implementation specific instruction **tlbiel**, (the processor local form of the TLB Invalidate) for the PowerPC 970FX microprocessor.
3. The instruction with primary opcode 0, when the instruction does not consist entirely of binary zeros and the extended opcode:
   256 Service Processor “Attention.”
4. Any other implementation-specific instructions that are not defined in the UISA, VEA, or OEA.

4.1.4 Memory Addressing

A program references memory using the effective (logical) address computed by the processor when it executes a load, store, branch, or cache instruction, and when it fetches the next sequential instruction.

4.1.4.1 Memory Operands

Bytes in memory are numbered consecutively starting with zero. Each number is the address of the corresponding byte. Within words bytes are numbered from left to right.

Memory operands may be bytes, halfwords, words, or doublewords, or, for the load/store multiple and load/store string instructions, a sequence of bytes or words. The address of a memory operand is the address of its first byte (that is, of its lowest-numbered byte). Operand length is implicit for each instruction. The PowerPC Architecture supports both big-endian and little-endian byte ordering. The default byte and bit ordering is big-endian; see *Section 3.1.2 Byte Ordering* for more information.

The operand of a single-register memory access instruction has a natural alignment boundary equal to the operand length. In other words, the “natural” address of an operand is an integral multiple of the operand length. A memory operand is said to be aligned if it is aligned at its natural boundary; otherwise it is misaligned. For a detailed discussion about memory operands, see *Chapter 3, “Operand Conventions*.”

4.1.4.2 Effective Address Calculation

An effective address (EA) is the 32 or 64-bit sum computed by the processor when executing a memory access or branch instruction or when fetching the next sequential instruction. For a memory access instruction, if the sum of the effective address and the operand length exceeds the maximum effective address, the memory operand is considered to wrap around from the maximum effective address through effective address 0, as described in the following paragraphs.

Effective address computations for both data and instruction accesses use 64 or 32-bit unsigned binary arithmetic. A carry from bit [0] is ignored. The 64-bit current instruction address and next instruction address are not affected by a change from 32-bit mode to the default 64-bit mode, but a change from the default 64-bit mode to 32-bit mode causes the high-order 32 bits to be cleared.

In the default 64-bit mode, the entire 64-bit result comprises the 64-bit effective address. The effective address arithmetic wraps around from the maximum address, $2^{64} – 1$, to address 0.

When a 64-bit implementation executes in 32-bit mode (MSR[SF] = ’0’), the low-order 32 bits of the 64-bit result comprise the effective address for the purpose of addressing memory. The high-order 32 bits of the 64-bit effective address are ignored for the purpose of accessing data, but are included whenever a 64-bit effective address is placed into a GPR by load with update and store with update instructions. The high-order 32 bits of the 64-bit effective address are cleared for the purpose of fetching instructions, and whenever a 64-bit effective address is placed into the LR by branch instructions having link register update option enabled (LK field, bit 31, in the instruction encoding = 1). The high-order 32 bits of the 64-bit effective address are cleared in SPRs when an exception error handler is invoked. In the context of addressing memory, the effective address arithmetic appears to wrap around from the maximum address, $2^{32} – 1$, to address 0.

Treating the high-order 32 bits of the effective address as zero effectively truncates the 64-bit effective address to a 32-bit effective address, such as would have been generated on a 32-bit implementation.

In 64-bit implementations (including 32-bit mode in 64-bit implementations), the three low-order bits of the calculated effective address may be modified by the processor before accessing memory if the PowerPC system is operating in little-endian mode. See *Section 3.1.2 Byte Ordering* for more information about little-endian mode.

Load and store operations have three categories of effective address generation that depend on the operands specified:

- Register indirect with immediate index mode
- Register indirect with index mode
- Register indirect mode

See *Section 4.2.3.1 Integer Load and Store Address Generation* for a detailed description of effective address generation for load and store operations.

Branch instructions have three categories of effective address generation:

- Immediate addressing
- Link register indirect
- Count register indirect

See *Section 4.2.4.1 Branch Instruction Address Calculation* for a detailed description of effective address generation for branch instructions.

Branch instructions can optionally load the link register (LR) with the next sequential instruction address (current instruction address + 4). This is used for subroutine call and return.

4.1.5 Synchronizing Instructions

The synchronization described in this section refers to the state of activities within the processor that is performing the synchronization. Refer to *Section 6.1.2 Synchronization* for more detailed information about other conditions that can cause context and execution synchronization.

4.1.5.1 Context Synchronizing Instructions

The System Call (**sc**), Return from Interrupt Doubleword (**rfid**), and Instruction Synchronize (**isync**) instructions perform context synchronization by allowing previously issued instructions to complete before continuing with program execution. These instructions will flush the instruction prefetch queue and start instruction fetching from memory in the context established after all preceding instructions have completed execution.

1. No higher priority exception exists (**sc**) and dispatching is halted.
2. All previous instructions have completed to a point where they can no longer cause an exception.
3. Previous instructions complete execution in the context (privilege, protection, and address translation) under which they were issued.
4. The instructions at the target of the branch of **sc**, **rfid** and those following the **isync** instruction execute in the context established by these instructions. For the **isync** instruction the instruction fetch queue must be flushed and instruction fetching restarted at the next sequential instruction. Both **sc**, and **rfid** execute like a branch and the flushing and refetching is automatic.
5. The operation ensures that the instructions that follow the operation will be fetched and executed in the context established by the operation.

4.1.5.2 Execution Synchronizing Instructions

An instruction is execution synchronizing if it satisfies the conditions of the first two items described above for context synchronization. The **sync** and **ptesync** instructions are treated like **isync** with respect to the second item described above (that is, the conditions described in the second item apply to the completion of **sync** and **ptesync**). The **sync**, **ptesync,** and **mtmsrd** instructions are examples of execution-synchronizing instructions.

The **isync** instruction is concerned mainly with the instruction stream in the processor on which it is executed, whereas, **sync** is looking outward towards the caches and memory and is concerned with data arriving at memory where it is visible to other processors in a multiprocessor environment. (e.g., cache block store, cache block flush, etc.)

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

4.1.6 Exception Summary

PowerPC processors have an exception mechanism for handling system functions and error conditions in an orderly way. The exception model is defined by the OEA. There are two kinds of exceptions—those caused directly by the execution of an instruction and those caused by an asynchronous event. Either may cause components of the system software to be invoked.

Exceptions can be caused directly by the execution of an instruction as follows:

- An attempt to execute an illegal instruction causes the illegal instruction (program exception) error handler to be invoked. An attempt by a user-level program to execute the supervisor-level instructions listed below causes the privileged instruction (program exception) handler to be invoked.

  The PowerPC Architecture provides the following supervisor-level instructions: **mfmsr**, **mfspr**, **mfsr**, **mfsrin**, **mtmsr**, **mtmsrd**, **mtspr**, **mtsr**, **mtsrin**, **rfid**, **slbia**, **slbie**, **slbmfee**, **slbmfev**, **slbmte**, **tlbia**, **tlbie**, **tlbiel**, and **tlbsync** (defined by OEA).

  Note: The privilege level of the **mfspr** and **mtspr** instructions depends on the SPR encoding.
- The execution of a defined instruction using an invalid form causes either the illegal instruction error handler or the privileged instruction handler to be invoked.
- The execution of an optional instruction that is not provided by the implementation causes the illegal instruction error handler to be invoked.
- An attempt to access memory in a manner that violates memory protection, or an attempt to access memory that is not available (page fault), causes the DSI exception handler or ISI exception handler to be invoked.
- An attempt to access memory with an effective address alignment that is invalid for the instruction causes the alignment exception handler to be invoked.
- The execution of an **sc** instruction permits a program to call on the system to perform a service, by causing a system call exception handler to be invoked.
- The execution of a trap instruction invokes the program exception trap handler.
- The execution of a floating-point instruction when floating-point instructions are disabled invokes the floating-point unavailable exception handler.

- The execution of an instruction that causes a floating-point exception that is enabled invokes the floating-point enabled exception handler.

Exceptions caused by asynchronous events are described in *Chapter 6, Exceptions*.

4.2 PowerPC UISA Instructions

The PowerPC user instruction set architecture (UISA) includes the base user-level instruction set (excluding a few user-level cache-control, synchronization, and time base instructions), user-level registers, programming model, data types, and addressing modes. This section discusses the instructions defined in the UISA.

4.2.1 Integer Instructions

The integer instructions consist of the following:

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

Integer instructions use the content of the GPRs as source operands and place results into GPRs. Integer arithmetic, shift, rotate, and string move instructions may update or read values from the XER, and the condition register (CR) fields may be updated if the Rc bit of the instruction is set.

These instructions treat the source operands as signed integers unless the instruction is explicitly identified as performing an unsigned operation. For example, Multiply High-Word Unsigned (**mulhwu**) and Divide Word Unsigned (**divwu**) instructions interpret both operands as unsigned integers.

The integer instructions that are coded to update the condition register, and the integer arithmetic instruction, **addic.**, set CR bits [0–3] (CR0) to characterize the result of the operation. In the default 64-bit mode, CR0 is set to reflect a signed comparison of the 64-bit result to zero. In 32-bit mode (of 64-bit implementations), CR0 is set to reflect a signed comparison of the low-order 32 bits of the result to zero.

The integer arithmetic instructions, **addic**, **addic.**, **subfic**, **addc**, **subfc**, **adde**, **subfe**, **addme**, **subfme**, **addze**, and **subfze**, always set the XER bit [CA], to reflect the carry out of bit [0] in the default 64-bit mode and out of bit [32] in 32-bit mode (of 64-bit implementations). Integer arithmetic instructions with the overflow enable (OE) bit set in the instruction encoding (instructions with o suffix) cause the XER[SO] and XER[OV] to reflect an overflow of the result. Except for the multiply low and divide instructions, these integer arithmetic instructions reflect the overflow of the 64-bit result in the default 64-bit mode and overflow of the low-order 32-bit result in 32-bit mode; however, the multiply low and divide instructions (**mulld**, **mullw**, **divd**, **divw**, **divdu**, and **divwu**) with o suffix cause XER[SO] and XER[OV] to reflect overflow of the 64-bit result (**mulld**, **divd**, and **divdu**) and overflow of the low-order 32-bit result (**mullw**, **divw**, and **divwu**).

Instructions that select the overflow option (enable XER[OV]) or that set the XER carry bit [CA] might delay the execution of subsequent instructions.

Unless otherwise noted, when CR0 and the XER are set, they characterize the value placed in the target register.

4.2.1.1 Integer Arithmetic Instructions

Table 4-1 lists the integer arithmetic instructions for the PowerPC processors.

Table 4-1. Integer Arithmetic Instructions

Name	**Mnemonic**	Operand Syntax	Operation
Add Immediate	**addi**	**r**D,**r**A,SIMM	The sum (**r**A|0) + SIMM is placed into **r**D.
Add Immediate Shifted	**addis**	**r**D,**r**A,SIMM	The sum (**r**A|0) + (SIMM || 0x0000) is placed into **r**D.
Add	**add** **add.** **addo** **addo.**	**r**D,**r**A,**r**B	The sum (**r**A) + (**r**B) is placed into **r**D. **add** Add **add.** Add with CR Update. The dot suffix enables the update of the CR. **addo** Add with Overflow Enabled. The o suffix enables the overflow bit [OV] in the XER. **addo.** Add with Overflow and CR Update. The o. suffix enables the update of the CR and enables the overflow bit [OV] in the XER.
Subtract From	**subf** **subf.** **subfo** **subfo.**	**r**D,**r**A,**r**B	The sum ¬ (**r**A) + (**r**B) +1 is placed into **r**D. **subf** Subtract From **subf.** Subtract from with CR Update. The dot suffix enables the update of the CR. **subfo** Subtract from with Overflow Enabled. The o suffix enables the overflow bit [OV] in the XER. **subfo.** Subtract from with Overflow and CR Update. The o. suffix enables the update of the CR and enables the overflow bit [OV] in the XER.
Add Immediate Carrying	**addic** **addic.**	**r**D,**r**A,SIMM	The sum (**r**A) + SIMM is placed into **r**D. The dot suffix enables the update of the CR. XER bit [CA] is altered.
Subtract from Immediate Carrying	**subfic**	**r**D,**r**A,SIMM	The sum ¬ (**r**A) + SIMM + 1 is placed into **r**D. XER bit [CA] is altered.
Add Carrying	**addc** **addc.** **addco** **addco.**	**r**D,**r**A,**r**B	The sum (**r**A) + (**r**B) is placed into **r**D. XER bit [CA] is altered. **addc** Add Carrying **addc.** Add Carrying with CR Update. The dot suffix enables the update of the CR. **addco** Add Carrying with Overflow Enabled. The o suffix enables the overflow bit [OV] in the XER. **addco.** Add Carrying with Overflow and CR Update. The o. suffix enables the update of the CR and enables the overflow bit [OV] in the XER.
Subtract from Carrying	**subfc** **subfc.** **subfco** **subfco.**	**r**D,**r**A,**r**B	The sum ¬ (**r**A) + (**r**B) + 1 is placed into **r**D. XER bit [CA] is altered. **subfc** Subtract from Carrying **subfc.** Subtract from Carrying with CR Update. The dot suffix enables the update of the CR. **subfco** Subtract from Carrying with Overflow. The o suffix enables the overflow bit [OV] in the XER. **subfco.** Subtract from Carrying with Overflow and CR Update. The o. suffix enables the update of the CR and enables the overflow bit [OV] in the XER.

Table 4-1. Integer Arithmetic Instructions (Continued)

Name	**Mnemonic**	Operand Syntax	Operation
Add Extended	**adde** **adde.** **addeo** **addeo.**	**r**D,**r**A,**r**B	The sum (**r**A) + (**r**B) + XER[CA] is placed into **r**D. XER bit [CA] is altered. **adde** Add Extended **adde.** Add Extended with CR Update. The dot suffix enables the update of the CR. **addeo** Add Extended with Overflow. The o suffix enables the overflow bit [OV] in the XER. **addeo.** Add Extended with Overflow and CR Update. The o. suffix enables the update of the CR and enables the overflow bit [OV] in the XER.
Subtract from Extended	**subfe** **subfe.** **subfeo** **subfeo.**	**r**D,**r**A,**r**B	The sum ¬ (**r**A) + (**r**B) + XER[CA] is placed into **r**D. **subfe** Subtract from Extended **subfe.** Subtract from Extended with CR Update. The dot suffix enables the update of the CR. **subfeo** Subtract from Extended with Overflow. The o suffix enables the overflow bit [OV] in the XER. **subfeo.** Subtract from Extended with Overflow and CR Update. The o. suffix enables the update of the CR and enables the overflow [OV] bit in the XER.
Add to Minus One Extended	**addme** **addme.** **addmeo** **addmeo.**	**r**D,**r**A	The sum (**r**A) + XER[CA] added to 0xFFFF_FFFF_FFFF_FFFF is placed into **r**D. XER bit [CA] is altered. **addme** Add to Minus One Extended **addme.** Add to Minus One Extended with CR Update. The dot suffix enables the update of the CR. **addmeo**Add to Minus One Extended with Overflow. The o suffix enables the overflow bit [OV] in the XER. **addmeo.**Add to Minus One Extended with Overflow and CR Update. The o. suffix enables the update of the CR and enables the overflow [OV] bit in the XER.
Subtract from Minus One Extended	**subfme** **subfme.** **subfmeo** **subfmeo.**	**r**D,**r**A	The sum ¬ (**r**A) + XER[CA] added to 0xFFFF_FFFF_FFFF_FFFF is placed into **r**D. XER bit [CA] is altered. **subfme** Subtract from Minus One Extended **subfme.**Subtract from Minus One Extended with CR Update. The dot suffix enables the update of the CR. **subfmeo**Subtract from Minus One Extended with Overflow. The o suffix enables the overflow bit [OV] in the XER. **subfmeo.**Subtract from Minus One Extended with Overflow and CR Update. The o. suffix enables the update of the CR and enables the overflow bit [OV] in the XER.
Add to Zero Extended	**addze** **addze.** **addzeo** **addzeo.**	**r**D,**r**A	The sum (**r**A) + XER[CA] is placed into **r**D. XER bit [CA] is altered. **addze** Add to Zero Extended **addze.** Add to Zero Extended with CR Update. The dot suffix enables the update of the CR. **addzeo** Add to Zero Extended with Overflow. The o suffix enables the overflow bit [OV] in the XER. **addzeo.** Add to Zero Extended with Overflow and CR Update. The o. suffix enables the update of the CR and enables the overflow bit [OV] in the XER.

Table 4-1. Integer Arithmetic Instructions (Continued)

Name	**Mnemonic**	Operand Syntax	Operation
Subtract from Zero Extended	**subfze** **subfze.** **subfzeo** **subfzeo.**	**r**D,**r**A	The sum ¬ (**r**A) + XER[CA] is placed into **r**D. **subfze** Subtract from Zero Extended **subfze.** Subtract from Zero Extended with CR Update. The dot suffix enables the update of the CR. **subfzeo** Subtract from Zero Extended with Overflow. The o suffix enables the overflow bit [OV] in the XER. **subfzeo.** Subtract from Zero Extended with Overflow and CR Update. The o. suffix enables the update of the CR and enables the overflow bit [OV] in the XER.
Negate	**neg** **neg.** **nego** **nego.**	**r**D,**r**A	The sum ¬ (**r**A) + 1 is placed into **r**D. **neg** Negate **neg.** Negate with CR Update. The dot suffix enables the update of the CR. **nego** Negate with Overflow. The o suffix enables the overflow bit [OV] in the XER. **nego.** Negate with Overflow and CR Update. The o. suffix enables the update of the CR and enables the overflow bit [OV] in the XER.
Multiply Low Immediate	**mulli**	**r**D,**r**A,SIMM	The low-order bits of the 128-bit product (**r**A) x SIMM are placed into **r**D. This instruction can be used with **mulhd***x* or **mulhw***x* to calculate a full 128-bit (or 64-bit) product. The low-order 32 bits of the product are the correct 32-bit product for 32-bit mode in 64-bit implementations.
Multiply Low	**mullw** **mullw.** **mullwo** **mullwo.**	**r**D,**r**A,**r**B	The -bit product (**r**A) x (**r**B) is placed into register **r**D. The 32-bit operands are the contents of the low-order 32 bits of **r**A and of **r**B. This instruction can be used with **mulhw***x* to calculate a full 64-bit product. The low-order 32 bits of the product are the correct 32-bit product for 32-bit mode in 64-bit implementations. **mullw** Multiply Low **mullw.** Multiply Low with CR Update. The dot suffix enables the update of the CR. **mullwo** Multiply Low with Overflow. The o suffix enables the overflow bit (OV) in the XER. **mullwo.** Multiply Low with Overflow and CR Update. The o. suffix enables the update of the condition register and enables the overflow bit (OV) in the XER.
Multiply Low Doubleword	**mulld** **mulld.** **mulldo** **mulldo.**	**r**D,**r**A,**r**B	The low-order 64 bits of the 128-bit product (**r**A) x (**r**B) are placed into **r**D. **mulld** Multiply Low Doubleword **mulld.** Multiply Low Doubleword with CR Update. The dot suffix enables the update of the CR. **mulldo** Multiply Low Doubleword with Overflow. The o suffix enables the overflow bit [OV] in the XER. **mulldo.** Multiply Low Doubleword with Overflow and CR Update. The o. suffix enables the update of the CR and enables the overflow bit [OV] in the XER.
Multiply High Word	**mulhw** **mulhw.**	**r**D,**r**A,**r**B	The contents of **r**A and **r**B are interpreted as 32-bit signed integers. The 64-bit product is formed. The high-order 32 bits of the 64-bit product are placed into the low-order 32 bits of **r**D. The value in the high-order 32 bits of **r**D is undefined. **mulhw** Multiply High Word **mulhw.** Multiply High Word with CR Update. The dot suffix enables the update of the CR.

Table 4-1. Integer Arithmetic Instructions (Continued)

Name	**Mnemonic**	Operand Syntax	Operation
Multiply High Doubleword	**mulhd** **mulhd.**	**r**D,**r**A,**r**B	The high-order 64 bits of the 128-bit product (**r**A) x (**r**B) are placed into register **r**D. Both operands and the product are interpreted as signed integers. **mulld** Multiply High Doubleword **mulld.** Multiply High Doubleword with CR Update. The dot suffix enables the update of the CR.
Multiply High Word Unsigned	**mulhwu** **mulhwu.**	**r**D,**r**A,**r**B	The contents of **r**A and of **r**B are interpreted as 32-bit unsigned integers. The 64-bit product is formed. The high-order 32 bits of the 64-bit product are placed into the low-order 32 bits of **r**D. The value in the high-order 32 bits of **r**D is undefined. **mulhwu** Multiply High Word Unsigned **mulhwu.** Multiply High Word Unsigned with CR Update. The dot suffix enables the update of the CR.
Multiply High Doubleword Unsigned	**mulhdu** **mulhdu.**	**r**D,**r**A,**r**B	The high-order 64 bits of the 128-bit product (**r**A) × (**r**B) are placed into register **r**D. **mulhdu** Multiply High Word Unsigned **mulhdu.** Multiply High Word Unsigned with CR Update. The dot suffix enables the update of the CR.
Divide Word	**divw** **divw.** **divwo** **divwo.**	**r**D,**r**A,**r**B	The 64-bit dividend is the signed value of the low-order 32 bits of **r**A. The 64-bit divisor is the signed value of the low-order 32 bits of **r**B. The low-order 32 bits of the 64-bit quotient are placed into the low-order 32 bits of **r**D. The contents of the high-order 32 bits of **r**D are undefined. The remainder is not supplied as a result. **divw** Divide Word **divw.** Divide Word with CR Update. The dot suffix enables the update of the CR. **divwo** Divide Word with Overflow. The o suffix enables the overflow bit [OV] in the XER. **divwo.** Divide Word with Overflow and CR Update. The o. suffix enables the update of the CR and enables the overflow bit [OV] in the XER.

Table 4-1. Integer Arithmetic Instructions (Continued)

Name	**Mnemonic**	Operand Syntax	Operation
Divide Doubleword	**divd** **divd.** **divdo** **divdo.**	**r**D,**r**A,**r**B	The 64-bit dividend is (**r**A). The 64-bit divisor is (**r**B). The 64-bit quotient is placed into **r**D. The remainder is not supplied as a result. **divd** Divide Doubleword **divd.** Divide Doubleword with CR Update. The dot suffix enables the update of the CR. **divdo** Divide Doubleword with Overflow. The o suffix enables the overflow bit [OV] in the XER. **divdo.** Divide Doubleword with Overflow and CR Update. The o. suffix enables the update of the CR and enables the overflow bit [OV] in the XER.
Divide Word Unsigned	**divwu** **divwu.** **divwuo** **divwuo.**	**r**D,**r**A,**r**B	The 64-bit dividend is the zero-extended value in the low-order 32 bits of **r**A. The 64-bit divisor is the zero-extended value in the low-order 32 bits of **r**B. The low-order 32 bits of the 64-bit quotient are placed into the low-order 32 bits of **r**D. The contents of the high-order 32 bits of **r**D are undefined. The remainder is not supplied as a result. **divwu** Divide Word Unsigned **divwu.** Divide Word Unsigned with CR Update. The dot suffix enables the update of the CR. **divwuo** Divide Word Unsigned with Overflow. The o suffix enables the overflow bit [OV] in the XER. **divwuo.** Divide Word Unsigned with Overflow and CR Update. The o. suffix enables the update of the CR and enables the overflow bit [OV] in the XER.
Divide Doubleword Unsigned	**divdu** **divdu.** **divduo** **divduo.**	**r**D,**r**A,**r**B	The 64-bit dividend is (**r**A). The 64-bit divisor is (**r**B). The 64-bit quotient is placed into **r**D. The remainder is not supplied as a result. **divdu** Divide Word Unsigned **divdu.** Divide Word Unsigned with CR Update. The dot suffix enables the update of the CR. **divduo** Divide Word Unsigned with Overflow. The o suffix enables the overflow bit [OV] in the XER. **divduo.** Divide Word Unsigned with Overflow and CR Update. The o. suffix enables the update of the CR and enables the overflow bit [OV] in the XER.

Although there is no “Subtract Immediate” instruction, its effect can be achieved by using an **addi** instruction with the immediate operand negated. Simplified mnemonics are provided that include this negation. The **subf** instructions subtract the second operand (**r**A) from the third operand (**r**B). Simplified mnemonics are provided in which the third operand is subtracted from the second operand. See *Appendix E Simplified Mnemonics* for examples.

4.2.1.2 Integer Compare Instructions

The integer compare instructions algebraically or logically compare the contents of register **r**A with either the zero-extended value of the UIMM operand, the sign-extended value of the SIMM operand, or the contents of register **r**B. The comparison is signed for the **cmpi** and **cmp** instructions, and unsigned for the **cmpli** and **cmpl** instructions. *Table 4-2* summarizes the integer compare instructions.

The PowerPC UISA specifies that the value in the L field determines whether the operands are treated as 32 or 64-bit values. If the L field is ‘0’ the operand length is 32 bits, and if it is ‘1’ the operand length is 64 bits. The simplified mnemonics for integer compare instructions, as shown in *Appendix E Simplified Mnemonics* correctly set or clear the ‘L’ value in the instruction encoding rather than requiring it to be coded as a numeric operand. When operands are treated as 32-bit signed quantities, bit [32] of (**r**A) and (**r**B) is the sign bit.

The integer compare instructions (shown in *Table 4-2*) set one of the leftmost three bits of the designated CR field, and clear the other two. XER[SO] is copied into bit [3] of the CR field.

Table 4-2. Integer Compare Instructions

Name	Mnemonic	Operand Syntax	Operation
Compare Immediate	**cmpi**	**crf**D,L,**r**A,SIMM	The value in register **r**A (**r**A[32–63] sign-extended to 64 bits if L = ‘0’) is compared with the sign-extended value of the SIMM operand, treating the operands as signed integers. The result of the comparison is placed into the CR field specified by operand **crf**D.
Compare	**cmp**	**crf**D,L,**r**A,**r**B	The value in register **r**A (**r**A[32–63] if L = ’0’) is compared with the value in register **r**B (**r**B[32–63] if L = ’0’), treating the operands as signed integers. The result of the comparison is placed into the CR field specified by operand **crf**D.
Compare Logical Immediate	**cmpli**	**crf**D,L,**r**A,UIMM	The value in register **r**A (**r**A[32–63] zero-extended to 64 bits if L = ‘0’) is compared with 0x0000_0000_0000 || UIMM, treating the operands as unsigned integers. The result of the comparison is placed into the CR field specified by operand **crf**D.
Compare Logical	**cmpl**	**crf**D,L,**r**A,**r**B	The value in register **r**A (**r**A[32–63] if L = ’0’) is compared with the value in register **r**B (**r**B[32–63] if L = ’0’), treating the operands as unsigned integers. The result of the comparison is placed into the CR field specified by operand **crf**D.

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

For information on simplified mnemonics for the integer compare instructions see *Appendix E Simplified Mnemonics*.

4.2.1.3 Integer Logical Instructions

The logical instructions shown in *Table 4-3* perform bit-parallel operations on -bit operands. Logical instructions with the CR updating enabled (uses dot suffix) and instructions **andi.** and **andis.** set CR field CR0 (bits [0 to 2]) to characterize the result of the logical operation. In the default 64-bit mode, these fields are set as if the 64-bit result were compared algebraically to zero. In 32-bit mode of a 64-bit implementation, these fields are set as if the sign-extended low-order 32 bits of the result were algebraically compared to zero. Logical instructions without CR update and the remaining logical instructions do not modify the CR. Logical instructions do not affect the XER[SO], XER[OV], and XER[CA] bits.

See *Appendix E Simplified Mnemonics* for simplified mnemonic examples for integer logical operations.

Table 4-3. Integer Logical Instructions

Name	Mnemonic	Operand Syntax	Operation
AND Immediate	**andi.**	**r**A,**r**S,UIMM	The contents of **r**S are ANDed with 0x0000_0000_0000 || UIMM and the result is placed into **r**A. The CR is updated.
AND Immediate Shifted	**andis.**	**r**A,**r**S,UIMM	The content of **r**S are ANDed with 0x0000_0000 || UIMM || 0x0000 and the result is placed into **r**A. The CR is updated.
OR Immediate	**ori**	**r**A,**r**S,UIMM	The contents of **r**S are ORed with 0x0000_0000_0000 || UIMM and the result is placed into **r**A. The preferred no-op is **ori 0,0,0**

Table 4-3. Integer Logical Instructions (Continued)

Name	Mnemonic	Operand Syntax	Operation
OR Immediate Shifted	**oris**	**r**A,**r**S,UIMM	The contents of **r**S are ORed with 0x0000_0000 || UIMM || 0x0000 and the result is placed into **r**A.
XOR Immediate	**xori**	**r**A,**r**S,UIMM	The contents of **r**S are XORed with 0x0000_0000_0000 || UIMM and the result is placed into **r**A.
XOR Immediate Shifted	**xoris**	**r**A,**r**S,UIMM	The contents of **r**S are XORed with 0x0000_0000 || UIMM || 0x0000 and the result is placed into **r**A.
AND	**and** **and.**	**r**A,**r**S,**r**B	The contents of **r**S are ANDed with the contents of register **r**B and the result is placed into **r**A. **and** AND **and.** AND with CR Update**.** The dot suffix enables the update of the CR.
OR	**or** **or.**	**r**A,**r**S,**r**B	The contents of **r**S are ORed with the contents of **r**B and the result is placed into **r**A. **or** OR **or.** OR with CR Update**.** The dot suffix enables the update of the CR.
XOR	**xor** **xor.**	**r**A,**r**S,**r**B	The contents of **r**S are XORed with the contents of **r**B and the result is placed into **r**A. **xor** XOR **xor.** XOR with CR Update**.** The dot suffix enables the update of the CR.
NAND	**nand** **nand.**	**r**A,**r**S,**r**B	The contents of **r**S are ANDed with the contents of **r**B and the one's complement of the result is placed into **r**A. **nand** NAND **nand.** NAND with CR Update**.** The dot suffix enables the update of CR. **Note: nand***x*, with **r**S = **r**B, can be used to obtain the one's complement.
NOR	**nor** **nor.**	**r**A,**r**S,**r**B	The contents of **r**S are ORed with the contents of **r**B and the one's complement of the result is placed into **r**A. **nor** NOR **nor.** NOR with CR Update**.** The dot suffix enables the update of the CR. **Note: nor***x*, with **r**S = **r**B, can be used to obtain the one's complement.
Equivalent	**eqv** **eqv.**	**r**A,**r**S,**r**B	The contents of **r**S are XORed with the contents of **r**B and the complemented result is placed into **r**A. **eqv** Equivalent **eqv.** Equivalent with CR Update**.** The dot suffix enables the update of the CR.
AND with Complement	**andc** **andc.**	**r**A,**r**S,**r**B	The contents of **r**S are ANDed with the one's complement of the contents of **r**B and the result is placed into **r**A. **andc** AND with Complement **andc.** AND with Complement with CR Update**.** The dot suffix enables the update of the CR.
OR with Complement	**orc** **orc.**	**r**A,**r**S,**r**B	The contents of **r**S are ORed with the complement of the contents of **r**B and the result is placed into **r**A. **orc** OR with Complement **orc.** OR with Complement with CR Update**.** The dot suffix enables the update of the CR.
Extend Sign Byte	**extsb** **extsb.**	**r**A,**r**S	The contents of the low-order eight bits of **r**S are placed into the low-order eight bits of **r**A. Bit [56] of **r**S is placed into the remaining high-order bits of **r**A. **extsb** Extend Sign Byte **extsb.** Extend Sign Byte with CR Update**.** The dot suffix enables the update of the CR.

Table 4-3. Integer Logical Instructions (Continued)

Name	Mnemonic	Operand Syntax	Operation
Extend Sign Halfword	**extsh** **extsh.**	**r**A,**r**S	The contents of the low-order 16 bits of **r**S are placed into the low-order 16 bits of **r**A. Bit [48] of **r**S is placed into the remaining high-order bits of **r**A. **extsh** Extend Sign Half-word **extsh.** Extend Sign Half-word with CR Update**.** The dot suffix enables the update of the CR.
Extend Sign Word	**extsw** **extsw.**	**r**A,**r**S	The contents of the low-order 32 bits of **r**S are placed into the low-order 32 bits of **r**A. Bit [32] of **r**S is placed into the remaining high-order bits of **r**A. **extsw** Extend Sign Word **extsw.** Extend Sign Word with CR Update**.** The dot suffix enables the update of the CR.
Count Leading Zeros Word	**cntlzw** **cntlzw.**	**r**A,**r**S	A count of the number of consecutive zero bits starting at bit [32] of **r**S is placed into **r**A. This number ranges from 0 to 32, inclusive. If Rc = '1' (dot suffix), LT is cleared in CR0. **cntlzw** Count Leading Zeros Word **cntlzw.** Count Leading Zeros Word with CR Update**.** The dot suffix enables the update of the CR.
Count Leading Zeros Doubleword	**cntlzd** **cntlzd.**	**r**A,**r**S	A count of the number of consecutive zero bits starting at bit [0] of **r**S is placed into **r**A. This number ranges from 0 to 64, inclusive. If Rc = '1' (dot suffix), LT is cleared in CR0. **cntlzd** Count Leading Zeros Doubleword **cntlzd.** Count Leading Zeros Doubleword with CR Update**.** The dot suffix enables the update of the CR.

4.2.1.4 Integer Rotate and Shift Instructions

Rotation operations are performed on data from a GPR, and the result, or a portion of the result, is returned to a GPR. 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 .

Two types of rotation operation are supported:

1. ROTL64 or rotate64 – the value rotated is the given 64-bit value. The rotate64 operation is used to rotate a given 64-bit quantity.
2. ROTL32 or rotate32 – the value rotated consists of two copies of bits [32-63] of the given 64-bit value, one copy in bits [0-31] and the other in bits [32-63]. The rotate32 operation is used to rotate a given 32-bit quantity.

The rotate and shift instructions employ a mask generator. The mask is 64 bits long and consists of '1' bits from a start bit, Mstart, through and including a stop bit, Mstop, and '0' bits elsewhere. The values of Mstart and Mstop range from 0 to . If Mstart > Mstop, the '1' bits wrap around from position 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–] = ones
    mask[0–mstop] = ones
    mask[all other bits] = zeros
```

It is not possible to specify an all-zero mask. The use of the mask is described in the following sections.

If CR updating is enabled, rotate and shift instructions set CR0[0–2] according to the contents of **r**A at the completion of the instruction. Rotate and shift instructions do not change the values of XER[OV] and XER[SO] bits. Rotate and shift instructions, except algebraic right shifts, do not change the XER[CA] bit.

See *Appendix E Simplified Mnemonics* for a complete list of simplified mnemonics that allows simpler coding of often-used functions such as clearing the leftmost or rightmost bits of a register, left justifying or right justifying an arbitrary field, and simple rotates and shifts.

Integer Rotate Instructions

Integer rotate instructions rotate the contents of a register. 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 is unchanged), or ANDed with a mask before being placed into the target register.

Rotate left instructions allow right-rotation of the contents of a register to be performed by a left-rotation of 64 - *n*, where *n* is the number of bits by which to rotate right. It also allows right-rotation of the contents of the low-order 32 bits of a register to be performed by a left-rotation of 32 - *n*, where *n* is the number of bits by which to rotate right.

The integer rotate instructions are summarized in *Table 4-4*

Table 4-4. Integer Rotate Instructions

Name	Mnemonic	Operand Syntax	Operation
Rotate Left Doubleword Immediate then Clear Left	**rldicl** **rldicl.**	**r**A,**r**S,SH,MB	The contents of **r**S are rotated left by the number of bits specified by operand SH. A mask is generated having ’1’ bits from the bit specified by operand MB through bit [63] and 0 bits elsewhere. The rotated data is ANDed with the generated mask and the result is placed into register **r**A. **rldicl** Rotate Left Doubleword Immediate then Clear Left **rldicl.** Rotate Left Doubleword Immediate then Clear Left with CR Update. The dot suffix enables the update of the CR.
Rotate Left Doubleword Immediate then Clear Right	**rldicr** **rldicr.**	**r**A,**r**S,SH,ME	The contents of **r**S are rotated left by the number of bits specified by operand SH. A mask is generated having ’1’ bits from bit [0] through the bit specified by operand ME and 0 bits elsewhere. The rotated data is ANDed with the generated mask and the result is placed into register **r**A. **rldicr** Rotate Left Doubleword Immediate then Clear Right **rldicl.** Rotate Left Doubleword Immediate then Clear Right with CR Update. The dot suffix enables the update of the CR.
Rotate Left Doubleword Immediate then Clear	**rldic** **rldic.**	**r**A,**r**S,SH,MB	The contents of register **r**S are rotated left by the number of bits specified by operand SH. A mask is generated having ’1’ bits from the bit specified by operand MB through bit [63 – SH], and 0 bits elsewhere. The rotated data is ANDed with the generated mask and the result is placed into register **r**A. **rldic** Rotate Left Doubleword Immediate then Clear **rldic.** Rotate Left Doubleword Immediate then Clear with CR Update. The dot suffix enables the update of the CR.

Table 4-4. Integer Rotate Instructions (Continued)

Name	Mnemonic	Operand Syntax	Operation
Rotate Left Word Immediate then AND with Mask	**rlwinm** **rlwinm.**	**r**A,**r**S,SH,MB,ME	The contents of register **r**S are rotated left by the number of bits specified by operand SH. A mask is generated having ’1’ bits from the bit specified by operand MB + 32 through the bit specified by operand ME + 32 and 0 bits elsewhere. The rotated data is ANDed with the generated mask and the result is placed into register **r**A. **rlwinm** Rotate Left Word Immediate then AND with Mask **rlwinm.** Rotate Left Word Immediate then AND with Mask with CR Update. The dot suffix enables the update of the CR.
Rotate Left Doubleword then Clear Left	**rldcl** **rldcl.**	**r**A,**r**S,**r**B,MB	The contents of register **r**S are rotated left by the number of bits specified by operand in the low-order six bits of **r**B. A mask is generated having ’1’ bits from the bit specified by operand MB through bit [63] and 0 bits elsewhere. The rotated data is ANDed with the generated mask and the result is placed into register **r**A. **rldcl** Rotate Left Doubleword then Clear Left **rldcl.** Rotate Left Doubleword then Clear Left with CR Update. The dot suffix enables the update of the CR.
Rotate Left Doubleword then Clear Right	**rldcr** **rldcr.**	**r**A,**r**S,**r**B,ME	The contents of register **r**S are rotated left by the number of bits specified by operand in the low-order six bits of **r**B. A mask is generated having ’1’ bits from bit [0] through the bit specified by operand ME and 0 bits elsewhere. The rotated data is ANDed with the generated mask and the result is placed into register **r**A. **rldcr** Rotate Left Doubleword then Clear Right **rldcr.** Rotate Left Doubleword then Clear Right with CR Update. The dot suffix enables the update of the CR.
Rotate Left Word then AND with Mask	**rlwnm** **rlwnm.**	**r**A,**r**S,**r**B,MB,ME	The contents of **r**S are rotated left by the number of bits specified by operand in the low-order five bits of **r**B. A mask is generated having ’1’ bits from the bit specified by operand MB + 32 through the bit specified by operand ME + 32 and 0 bits elsewhere. The rotated word is ANDed with the generated mask and the result is placed into **r**A. **rlwnm** Rotate Left Word then AND with Mask **rlwnm.** Rotate Left Word then AND with Mask with CR Update. The dot suffix enables the update of the CR.
Rotate Left Word Immediate then Mask Insert	**rlwimi** **rlwimi.**	**r**A,**r**S,SH,MB,ME	The contents of **r**S are rotated left by the number of bits specified by operand SH. A mask is generated having ’1’ bits from the bit specified by operand MB + 32 through the bit specified by operand ME + 32 and 0 bits elsewhere. The rotated word is inserted into **r**A under control of the generated mask. **rlwimi** Rotate Left Word Immediate then Mask **rlwimi.** Rotate Left Word Immediate then Mask Insert with CR Update. The dot suffix enables the update of the CR.
Rotate Left Doubleword Immediate then Mask Insert	**rldimi** **rldimi.**	**r**A,**r**S,SH,MB	The contents of **r**S are rotated left by the number of bits specified by operand SH. A mask is generated having ’1’ bits from the bit specified by operand MB through [63 – SH] (the bit specified by SH), and 0 bits elsewhere. The rotated data is inserted into **r**A under control of the generated mask. **rldimi** Rotate Left Word Immediate then Mask **rldimi.** Rotate Left Word Immediate then Mask Insert with CR Update. The dot suffix enables the update of the CR.

Integer Shift Instructions

The integer shift instructions perform left and right shifts. Immediate-form logical (unsigned) shift operations are obtained by specifying masks and shift values for certain rotate instructions. Simplified mnemonics (shown in *Appendix E Simplified Mnemonics*) are provided to make coding of such shifts simpler and easier to understand.

Any shift right algebraic instruction, followed by **addze**, can be used to divide quickly by 2^n. The setting of XER[CA] by the shift right algebraic instruction is independent of mode.

Multiple-precision shifts can be programmed as shown in *Appendix B Multiple-Precision Shifts*.

The integer shift instructions are summarized in *Table 4-5*.

Table 4-5. Integer Shift Instructions

Name	Mnemonic	Operand Syntax	Operation
Shift Left Doubleword	**sld** **sld.**	**r**A,**r**S,**r**B	The contents of **r**S are shifted left the number of bits specified by the low-order seven bits of **r**B. Bits shifted out of position 0 are lost. Zeros are supplied to the vacated positions on the right. The result is placed into **r**A. Shift amounts from 64 to 127 give a zero result. **sld** Shift Left Doubleword **sld.** Shift Left Doubleword with CR Update. The dot suffix enables the update of the CR.
Shift Left Word	**slw** **slw.**	**r**A,**r**S,**r**B	The contents of the low-order 32 bits of **r**S are shifted left the number of bits specified by operand in the low-order six bits of **r**B. 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 the low-order 32 bits of **r**A. The value in the high-order 32 bits of **r**A is cleared, and shift amounts from 32 to 63 give a zero result. **slw** Shift Left Word **slw.** Shift Left Word with CR Update. The dot suffix enables the update of the CR.
Shift Right Doubleword	**srd** **srd.**	**r**A,**r**S,**r**B	The contents of **r**S are shifted right the number of bits specified by the low-order seven bits of **r**B. Bits shifted out of position 63 are lost. Zeros are supplied to the vacated positions on the left. The result is placed into **r**A. Shift amounts from 64 to 127 give a zero result. **srd** Shift Right Doubleword **srd.** Shift Right Doubleword with CR Update. The dot suffix enables the update of the CR.
Shift Right Word	**srw** **srw.**	**r**A,**r**S,**r**B	The contents of the low-order 32 bits of **r**S are shifted right the number of bits specified by the low-order six bits of **r**B. 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 the low-order 32 bits of **r**A. The value in the high-order 32 bits of **r**A is cleared to zero, and shift amounts from 32 to 63 give a zero result. **srw** Shift Right Word **srw.** Shift Right Word with CR Update. The dot suffix enables the update of the CR.
Shift Right Algebraic Doubleword Immediate	**sradi** **sradi.**	**r**A,**r**S,SH	The contents of **r**S are shifted right the number of bits specified by operand SH. Bits shifted out of position 63 are lost. Bit [0] of **r**S is replicated to fill the vacated positions on the left. The result is placed into **r**A. XER[CA] is set if **r**S contains a negative number and any '1' bits are shifted out of position 63; otherwise XER[CA] is cleared. An operand SH of zero causes **r**A to be loaded with the contents of **r**S and XER[CA] to be cleared to zero. **sradi** Shift Right Algebraic Doubleword Immediate **sradi.** Shift Right Algebraic Doubleword Immediate with CR Update. The dot suffix enables the update of the CR.

Table 4-5. Integer Shift Instructions (Continued)

Name	Mnemonic	Operand Syntax	Operation
Shift Right Algebraic Word Immediate	**srawi** **srawi.**	**r**A,**r**S,SH	The contents of the low-order 32 bits of **r**S are shifted right the number of bits specified by operand SH. Bits shifted out of position 63 are lost. Bit [32] of **r**S is replicated to fill the vacated positions on the left. The 32-bit result is sign extended and placed into the low-order 32 bits of **r**A. **srawi** Shift Right Algebraic Word Immediate **srawi.** Shift Right Algebraic Word Immediate with CR Update. The dot suffix enables the update of the CR.
Shift Right Algebraic Doubleword	**srad** **srad.**	**r**A,**r**S,**r**B	The contents of **r**S are shifted right the number of bits specified by the low-order seven bits of **r**B. Bits shifted out of position 63 are lost. Bit [0] of **r**S is replicated to fill the vacated positions on the left. The result is placed into **r**A. **srad** Shift Right Algebraic Doubleword **srad.** Shift Right Algebraic Doubleword with CR Update. The dot suffix enables the update of the CR.
Shift Right Algebraic Word	**sraw** **sraw.**	**r**A,**r**S,**r**B	The contents of the low-order 32 bits of **r**S are shifted right the number of bits specified by the low-order six bits of **r**B. Bits shifted out of position 63 are lost. Bit [32] of **r**S is replicated to fill the vacated positions on the left. The 32-bit result is placed into the low-order 32 bits of **r**A. **sraw** Shift Right Algebraic Word **sraw.** Shift Right Algebraic Word with CR Update. The dot suffix enables the update of the CR.

4.2.2 Floating-Point Instructions

This section describes the floating-point instructions, which include the following:

- Floating-point arithmetic instructions
- Floating-point multiply-add instructions
- Floating-point rounding and conversion instructions
- Floating-point compare instructions
- Floating-point status and control register instructions
- Floating-point move instructions

Note: MSR[FP] must be set in order for any of these instructions (including the floating-point loads and stores) to be executed. If MSR[FP] = ’0’ when any floating-point instruction is attempted, the floating-point unavailable exception is taken (see *Section 6.4.10 Floating-Point Unavailable Exception (0x00800)*). See *Section 4.2.3 Load and Store Instructions* for information about floating-point loads and stores.

The PowerPC Architecture supports a floating-point system as defined in the IEEE-754 standard, but requires software support to conform with that standard. Floating-point operations conform to the IEEE-754 standard, with the exception of operations performed with the **fmadd**, **fres**, **fsel**, and **frsqrte** instructions, or if software sets the non-IEEE mode bit [NI] in the FPSCR. Refer to *Section 3.3 Floating-Point Execution Models—UISA*, for detailed information about the floating-point formats and exception conditions. Also, refer to *Appendix C Floating-Point Models* for more information on the floating-point execution models used by the PowerPC Architecture.

4.2.2.1 Floating-Point Arithmetic Instructions

The floating-point arithmetic instructions are summarized in *Table 4-6*.

Table 4-6. Floating-Point Arithmetic Instructions

Name	Mnemonic	Operand Syntax	Operation
Floating Add (Double-Precision)	**fadd** **fadd.**	**fr**D,**fr**A,**fr**B	The floating-point operand in register **fr**A is added to the floating-point operand in register **fr**B. If the most significant bit of the resultant significand is not a one the result is normalized. The result is rounded to the target precision under control of the floating-point rounding control field RN of the FPSCR and placed into register **fr**D. **fadd** Floating Add (Double-Precision) **fadd.** Floating Add (Double-Precision) with CR Update. The dot suffix enables the update of the CR.
Floating Add Single	**fadds** **fadds.**	**fr**D,**fr**A,**fr**B	The floating-point operand in register **fr**A is added to the floating-point operand in register **fr**B. If the most significant bit of the resultant significand is not a one, the result is normalized. The result is rounded to the target precision under control of the floating-point rounding control field RN of the FPSCR and placed into register **fr**D. **fadds** Floating Add Single **fadds.** Floating Add Single with CR Update. The dot suffix enables the update of the CR.
Floating Subtract (Double-Precision)	**fsub** **fsub.**	**fr**D,**fr**A,**fr**B	The floating-point operand in register **fr**B is subtracted from the floating-point operand in register **fr**A. If the most significant bit of the resultant significand is not 1, the result is normalized. The result is rounded to the target precision under control of the floating-point rounding control field RN of the FPSCR and placed into register **fr**D. **fsub** Floating Subtract (Double-Precision) **fsub.** Floating Subtract (Double-Precision) with CR Update. The dot suffix enables the update of the CR.
Floating Subtract Single	**fsubs** **fsubs.**	**fr**D,**fr**A,**fr**B	The floating-point operand in register **fr**B is subtracted from the floating-point operand in register **fr**A. If the most significant bit of the resultant significand is not 1, the result is normalized. The result is rounded to the target precision under control of the floating-point rounding control field RN of the FPSCR and placed into **fr**D. **fsubs** Floating Subtract Single **fsubs.** Floating Subtract Single with CR Update. The dot suffix enables the update of the CR.
Floating Multiply (Double-Precision)	**fmul** **fmul.**	**fr**D,**fr**A,**fr**C	The floating-point operand in register **fr**A is multiplied by the floating-point operand in register **fr**C. **fmul** Floating Multiply (Double-Precision) **fmul.** Floating Multiply (Double-Precision) with CR Update. The dot suffix enables the update of the CR.
Floating Multiply Single	**fmuls** **fmuls.**	**fr**D,**fr**A,**fr**C	The floating-point operand in register **fr**A is multiplied by the floating-point operand in register **fr**C. **fmuls** Floating Multiply Single **fmuls.** Floating Multiply Single with CR Update. The dot suffix enables the update of the CR.
Floating Divide (Double-Precision)	**fdiv** **fdiv.**	**fr**D,**fr**A,**fr**B	The floating-point operand in register **fr**A is divided by the floating-point operand in register **fr**B. No remainder is preserved. **fdiv** Floating Divide (Double-Precision) **fdiv.** Floating Divide (Double-Precision) with CR Update. The dot suffix enables the update of the CR.

Table 4-6. Floating-Point Arithmetic Instructions (Continued)

Name	Mnemonic	Operand Syntax	Operation
Floating Divide Single	**fdivs** **fdivs.**	**fr**D,**fr**A,**fr**B	The floating-point operand in register **fr**A is divided by the floating-point operand in register **fr**B. No remainder is preserved. **fdivs** Floating Divide Single **fdivs.** Floating Divide Single with CR Update. The dot suffix enables the update of the CR.
Floating Square Root (Double-Precision)	**fsqrt** **fsqrt.**	**fr**D,**fr**B	The square root of the floating-point operand in register **fr**B is placed into register **fr**D. **fsqrt** Floating Square Root (Double-Precision) **fsqrt.** Floating Square Root (Double-Precision) with CR Update. The dot suffix enables the update of the CR. **Note:** This instruction is optional.
Floating Square Root Single	**fsqrts** **fsqrts.**	**fr**D,**fr**B	The square root of the floating-point operand in register **fr**B is placed into register **fr**D. **fsqrts** Floating Square Root Single **fsqrts.** Floating Square Root Single with CR Update. The dot suffix enables the update of the CR. **Note:** This instruction is optional.
Floating Reciprocal Estimate Single	**fres** **fres.**	**fr**D,**fr**B	A single-precision estimate of the reciprocal of the floating-point operand in register **fr**B is placed into **fr**D. The estimate placed into **fr**D is correct to a precision of one part in 256 of the reciprocal of **fr**B. **fres** Floating Reciprocal Estimate Single **fres.** Floating Reciprocal Estimate Single with CR Update. The dot suffix enables the update of the CR. **Note:** This instruction is optional.
Floating Reciprocal Square Root Estimate	**frsqrte** **frsqrte.**	**fr**D,**fr**B	A double-precision estimate of the reciprocal of the square root of the floating-point operand in register **fr**B is placed into **fr**D. The estimate placed into **fr**D is correct to a precision of one part in 32 of the reciprocal of the square root of **fr**B. **frsqrte** Floating Reciprocal Square Root Estimate **frsqrte.** Floating Reciprocal Square Root estimate with CR Update. The dot suffix enables the update of the CR. **Note:** This instruction is optional.
Floating Select	**fsel**	**fr**D,**fr**A,**fr**C,**fr**B	The floating-point operand in **fr**A is compared to the value zero. If the operand is greater than or equal to zero, **fr**D is set to the contents of **fr**C. If the operand is less than zero or is a NaN, **fr**D is set to the contents of **fr**B. The comparison ignores the sign of zero (that is, regards '+0' as equal to '-0'). **fsel** Floating Select **fsel.** Floating Select with CR Update. The dot suffix enables the update of the CR. **Note:** This instruction is optional.

4.2.2.2 Floating-Point Multiply-Add Instructions

These instructions combine multiply and add operations without an intermediate rounding operation. The fractional part of the intermediate product is 106 bits wide, 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 (**fmuls**, followed by **fadds** or **fsubs**). That is, multiplication of infinity by zero or of anything by an SNaN, and/or addition of an SNaN, cause the corresponding exception bits to be set.

The floating-point multiply-add instructions are summarized in *Table 4-7*.

Table 4-7. Floating-Point Multiply-Add Instructions

Name	Mnemonic	Operand Syntax	Operation
Floating Multiply-Add (Double-Precision)	**fmadd** **fmadd.**	**fr**D,**fr**A,**fr**C,**fr**B	The floating-point operand in register **fr**A is multiplied by the floating-point operand in register **fr**C. The floating-point operand in register **fr**B is added to this intermediate result. **fmadd** Floating Multiply-Add (Double-Precision) **fmadd.** Floating Multiply-Add (Double-Precision) with CR Update. The dot suffix enables the update of the CR.
Floating Multiply-Add Single	**fmadds** **fmadds.**	**fr**D,**fr**A,**fr**C,**fr**B	The floating-point operand in register **fr**A is multiplied by the floating-point operand in register **fr**C. The floating-point operand in register **fr**B is added to this intermediate result. **fmadds** Floating Multiply-Add Single **fmadds.**Floating Multiply-Add Single with CR Update. The dot suffix enables the update of the CR.
Floating Multiply-Subtract (Double-Precision)	**fmsub** **fmsub.**	**fr**D,**fr**A,**fr**C,**fr**B	The floating-point operand in register **fr**A is multiplied by the floating-point operand in register **fr**C. The floating-point operand in register **fr**B is subtracted from this intermediate result. **fmsub** Floating Multiply-Subtract (Double-Precision) **fmsub.** Floating Multiply-Subtract (Double-Precision) with CR Update. The dot suffix enables the update of the CR.
Floating Multiply-Subtract Single	**fmsubs** **fmsubs.**	**fr**D,**fr**A,**fr**C,**fr**B	The floating-point operand in register **fr**A is multiplied by the floating-point operand in register **fr**C. The floating-point operand in register **fr**B is subtracted from this intermediate result. **fmsubs** Floating Multiply-Subtract Single **fmsubs.**Floating Multiply-Subtract Single with CR Update. The dot suffix enables the update of the CR.
Floating Negative Multiply- Add (Double-Precision)	**fnmadd** **fnmadd.**	**fr**D,**fr**A,**fr**C,**fr**B	The floating-point operand in register **fr**A is multiplied by the floating-point operand in register **fr**C. The floating-point operand in register **fr**B is added to this intermediate result. **fnmadd** Floating Negative Multiply-Add (Double-Precision) **fnmadd.**Floating Negative Multiply-Add (Double-Precision) with CR Update. The dot suffix enables update of the CR.

Table 4-7. Floating-Point Multiply-Add Instructions (Continued)

Name	Mnemonic	Operand Syntax	Operation
Floating Negative Multiply- Add Single	**fnmadds** **fnmadds.**	**fr**D,**fr**A,**fr**C,**fr**B	The floating-point operand in register **fr**A is multiplied by the floating-point operand in register **fr**C. The floating-point operand in register **fr**B is added to this intermediate result. **fnmadds**Floating Negative Multiply-Add Single **fnmadds.**Floating Negative Multiply-Add Single with CR Update. The dot suffix enables the update of the CR.
Floating Negative Multiply- Subtract (Double-Precision)	**fnmsub** **fnmsub.**	**fr**D,**fr**A,**fr**C,**fr**B	The floating-point operand in register **fr**A is multiplied by the floating-point operand in register **fr**C. The floating-point operand in register **fr**B is subtracted from this intermediate result. **fnmsub** Floating Negative Multiply-Subtract (Double-Precision) **fnmsub.**Floating Negative Multiply-Subtract (Double-Precision) with CR Update. The dot suffix enables the update of the CR.
Floating Negative Multiply- Subtract Single	**fnmsubs** **fnmsubs.**	**fr**D,**fr**A,**fr**C,**fr**B	The floating-point operand in register **fr**A is multiplied by the floating-point operand in register **fr**C. The floating-point operand in register **fr**B is subtracted from this intermediate result. **fnmsubs**Floating Negative Multiply-Subtract Single **fnmsubs.**Floating Negative Multiply-Subtract Single with CR Update. The dot suffix enables the update of the CR.

For more information on multiply-add instructions, refer to *Appendix C.2 Execution Model for Multiply-Add Type Instructions*.

4.2.2.3 Floating-Point Rounding and Conversion Instructions

The Floating Round to Single-Precision (**frsp**) instruction is used to truncate a 64-bit double-precision number to a 32-bit single-precision floating-point number. The floating-point convert instructions convert a 64-bit double-precision floating-point number to a 32-bit signed integer number.

The PowerPC Architecture defines bits [0–31] of floating-point register **fr**D as undefined when executing the Floating Convert to Integer Word (**fctiw**) and Floating Convert to Integer Word with Round toward Zero (**fctiwz**) instructions. The floating-point rounding instructions are shown in *Table 4-8*.

Examples of uses of these instructions to perform various conversions can be found in *Appendix C Floating-Point Models*.

Table 4-8. Floating-Point Rounding and Conversion Instructions

Name	Mnemonic	Operand Syntax	Operation
Floating Round to Single-Precision	**frsp** **frsp.**	**fr**D,**fr**B	The floating-point operand in **fr**B is rounded to single-precision using the rounding mode specified by FPSCR[RN] and placed into **fr**D. **frsp** Floating Round to Single-Precision **frsp.** Floating Round to Single-Precision with CR Update. The dot suffix enables the update of the CR.
Floating Convert from Integer Doubleword	**fcfid** **fcfid.**	**fr**D,**fr**B	The 64-bit signed integer operand in **fr**B is converted to an infinitely precise floating-point integer. The result of the conversion is rounded to double-precision using the rounding mode specified by FPSCR[RN] and placed into register **fr**D. **fcfid** Floating Convert from Integer Doubleword **fcfid.** Floating Convert from Integer Doubleword with CR Update. The dot suffix enables the update of the CR.
Floating Convert to Integer Doubleword	**fctid** **fctid.**	**fr**D,**fr**B	The floating-point operand in register **fr**B is converted to a 64-bit signed integer, using the rounding mode specified by FPSCR[RN], and placed in **fr**D. **fctiw** Floating Convert to Integer Doubleword **fctiw.** Floating Convert to Integer Doubleword with CR Update. The dot suffix enables the update of the CR.
Floating Convert to Integer Double-word with Round toward Zero	**fctidz** **fctidz.**	**fr**D,**fr**B	The floating-point operand in register **fr**B is converted to a 64-bit signed integer, using the rounding mode Round toward Zero and placed in **fr**D. **fctidz** Floating Convert to Integer Doubleword with Round toward Zero **fctidz.** Floating Convert to Integer Doubleword with Round toward Zero with CR Update. The dot suffix enables the update of the CR.
Floating Convert to Integer Word	**fctiw** **fctiw.**	**fr**D,**fr**B	The floating-point operand in register **fr**B is converted to a 32-bit signed integer, using the rounding mode specified by FPSCR[RN], and placed in the low-order 32 bits of **fr**D. Bits [0–31] of **fr**D are undefined. **fctiw** Floating Convert to Integer Word **fctiw.** Floating Convert to Integer Word with CR Update. The dot suffix enables the update of the CR.
Floating Convert to Integer Word with Round toward Zero	**fctiwz** **fctiwz.**	**fr**D,**fr**B	The floating-point operand in register **fr**B is converted to a 32-bit signed integer, using the rounding mode Round toward Zero, and placed in the low-order 32 bits of **fr**D. Bits [0–31] of **fr**D are undefined. **fctiwz** Floating Convert to Integer Word with Round toward Zero **fctiwz.** Floating Convert to Integer Word with Round toward Zero with CR Update. The dot suffix enables the update of the CR.

4.2.2.4 Floating-Point Compare Instructions

Floating-point compare instructions compare the contents of two floating-point registers and the comparison ignores the sign of zero (that is ‘+0’ = ‘-0’). The comparison can be ordered or unordered. The comparison sets one bit in the designated CR field and clears the other three bits. The floating-point condition code (FPCC) in bits [16–19] of the floating-point status and control register (FPSCR) is set in the same way.

The CR field and the FPCC are interpreted as shown in *Table 4-9*.

Table 4-9. CR Bit Settings

Bit	Name	Description
0	FL	(**fr**A) < (**fr**B)
1	FG	(**fr**A) > (**fr**B)
2	FE	(**fr**A) = (**fr**B)
3	FU	(**fr**A) ? (**fr**B) (unordered)
Note: A result of “unordered” indicates that at least one of operations of the comparison was a NaN.		

The floating-point compare instructions are summarized in *Table 4-10*.

Table 4-10. Floating-Point Compare Instructions

Name	Mnemonic	Operand Syntax	Operation
Floating Compare Unordered	fcmpu	**crf**D,**fr**A,**fr**B	The floating-point operand in **fr**A is compared to the floating-point operand in **fr**B. The result of the compare is placed into **crf**D and the FPCC.
Floating Compare Ordered	fcmpo	**crf**D,**fr**A,**fr**B	The floating-point operand in **fr**A is compared to the floating-point operand in **fr**B. The result of the compare is placed into **crf**D and the FPCC.

4.2.2.5 Floating-Point Status and Control Register Instructions

Every FPSCR instruction appears to synchronize the effects of all floating-point instructions executed by a given processor. Executing an FPSCR instruction ensures that all floating-point instructions previously initiated by the given processor appear to have completed before the FPSCR instruction is initiated and that no subsequent floating-point instructions appear to be initiated by the given processor until the FPSCR instruction has completed. In particular:

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

Floating-point memory access instructions are not affected by the execution of the FPSCR instructions.

The FPSCR instructions are summarized in *Table 4-11*.

Table 4-11. Floating-Point Status and Control Register Instructions

Name	Mnemonic	Operand Syntax	Operation
Move from FPSCR	**mffs** **mffs.**	**fr**D	The contents of the FPSCR are placed into bits [32–63] of **fr**D. Bits [0–31] of **fr**D are undefined. **mffs** Move from FPSCR **mffs.** Move from FPSCR with CR Update. The dot suffix enables the update of the CR.
Move to Condition Register from FPSCR	**mcrfs**	**crf**D,**crf**S	The contents of FPSCR field specified by operand **crf**S are copied to the CR field specified by operand **crf**D. All exception bits copied (except FEX and VX bits) are cleared in the FPSCR.
Move to FPSCR Field Immediate	**mtfsfi** **mtfsfi.**	**crf**D,IMM	The contents of the IMM field are placed into FPSCR field **crf**D. The contents of FPSCR[FX] are altered only if **crf**D = '0'. **mtfsfi** Move to FPSCR Field Immediate **mtfsfi.** Move to FPSCR Field Immediate with CR Update. The dot suffix enables the update of the CR.
Move to FPSCR Fields	**mtfsf** **mtfsf.**	FM,**fr**B	Bits [32–63] of **fr**B are placed into the FPSCR under control of the field mask specified by FM. The field mask identifies the 4-bit fields affected. Let *i* be an integer in the range 0–7. If FM[*i*] = '1', FPSCR field *i* (FPSCR bits 4×*i* through 4×*i*+3) is set to the contents of the corresponding field of the low-order 32 bits of **fr**B. The contents of FPSCR[FX] are altered only if FM[0] = '1'. **mtfsf** Move to FPSCR Fields **mtfsf.** Move to FPSCR Fields with CR Update. The dot suffix enables the update of the CR.
Move to FPSCR Bit 0	**mtfsb0** **mtfsb0.**	**crb**D	The FPSCR bit location specified by operand **crb**D is cleared. Bits [1, 2] (FEX and VX) cannot be reset explicitly. **mtfsb0** Move to FPSCR Bit [0] **mtfsb0.** Move to FPSCR Bit [0] with CR Update. The dot suffix enables the update of the CR.
Move to FPSCR Bit 1	**mtfsb1** **mtfsb1.**	**crb**D	The FPSCR bit location specified by operand **crb**D is set. Bits [1, 2] (FEX and VX) cannot be set explicitly. **mtfsb1** Move to FPSCR Bit [1] **mtfsb1.** Move to FPSCR Bit [1] with CR Update. The dot suffix enables the update of the CR.

4.2.2.6 Floating-Point Move Instructions

Floating-point move instructions copy data from one FPR to another, altering the sign bit (bit [0]) as described for the **fneg**, **fabs**, and **fnabs** instructions in *Table 4-12*. The **fneg**, **fabs**, and **fnabs** instructions may alter the sign bit of a NaN. The floating-point move instructions do not modify the FPSCR. The CR update option in these instructions controls the placing of result status into CR1. If the CR update option is enabled, CR1 is set; otherwise, CR1 is unchanged.

Table 4-12 provides a summary of the floating-point move instructions.

Table 4-12. Floating-Point Move Instructions

Name	Mnemonic	Operand Syntax	Operation
Floating Move Register	**fmr** **fmr.**	**fr**D,**fr**B	The contents of **fr**B are placed into **fr**D. **fmr** Floating Move Register **fmr.** Floating Move Register with CR Update. The dot suffix enables the update of the CR.
Floating Negate	**fneg** **fneg.**	**fr**D,**fr**B	The contents of **fr**B with bit [0] inverted are placed into **fr**D. **fneg** Floating Negate **fneg.** Floating Negate with CR Update. The dot suffix enables the update of the CR.
Floating Absolute Value	**fabs** **fabs.**	**fr**D,**fr**B	The contents of **fr**B with bit [0] cleared are placed into **fr**D. **fabs** Floating Absolute Value **fabs.** Floating Absolute Value with CR Update. The dot suffix enables the update of the CR.
Floating Negative Absolute Value	**fnabs** **fnabs.**	**fr**D,**fr**B	The contents of **fr**B with bit [0] set are placed into **fr**D. **fnabs** Floating Negative Absolute Value **fnabs.** Floating Negative Absolute Value with CR Update. The dot suffix enables the update of the CR.

4.2.3 Load and Store Instructions

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

- Integer load instructions
- Integer store instructions
- Integer load and store with byte-reverse instructions
- Integer load and store multiple instructions
- Floating-point load instructions
- Floating-point store instructions
- Memory synchronization instructions

4.2.3.1 Integer Load and Store Address Generation

Integer load and store operations generate effective addresses using register indirect with immediate index mode (register contents + immediate), register indirect with index mode (register contents + register contents), or register indirect mode (register contents only). See *Section 4.1.4.2 Effective Address Calculation* for information about calculating effective addresses.

Note: In some implementations, operations that are not naturally aligned may suffer performance degradation. Refer to *Section 6.4.8.1 Integer Alignment Exceptions* for additional information about load and store address alignment exceptions.

Register indirect addressing for integer loads and stores is discussed in the following sections:

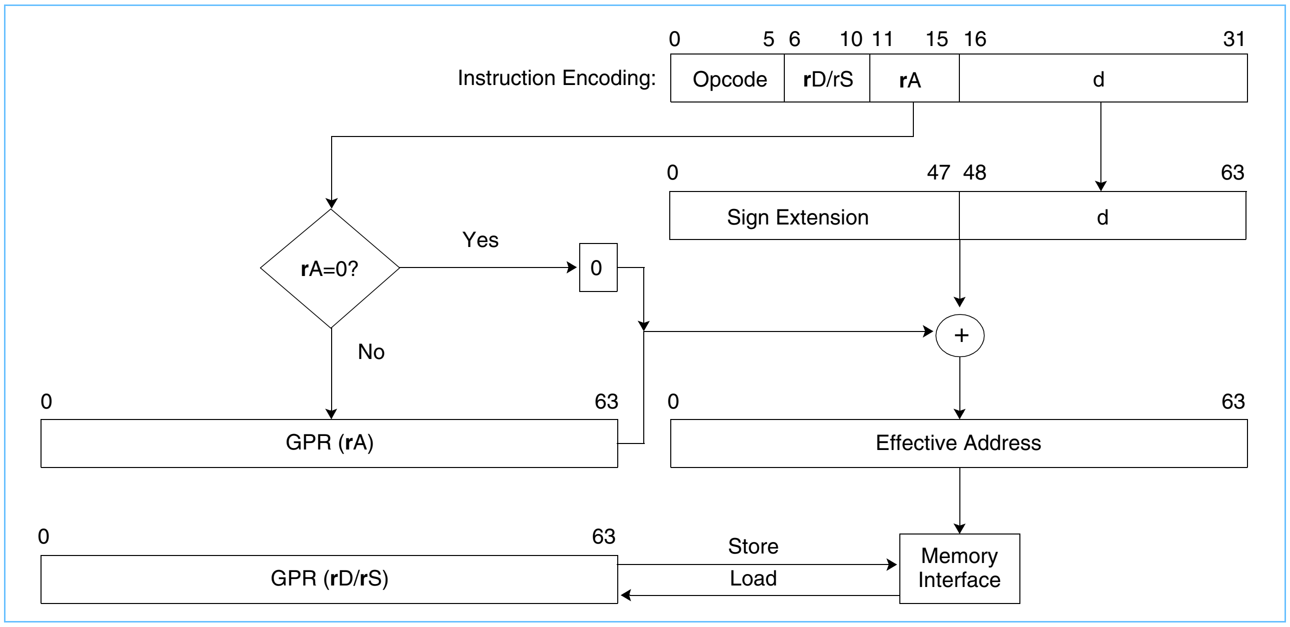
- Register Indirect with Immediate Index Addressing for Integer Loads and Stores
- Register Indirect with Index Addressing for Integer Loads and Stores
- Register Indirect Addressing for Integer Loads and Stores

Register Indirect with Immediate Index Addressing for Integer Loads and Stores

Instructions using this addressing mode contain a signed 16-bit immediate index (d operand) which is sign extended, and added to the contents of a general-purpose register specified in the instruction (**r**A operand) to generate the effective address. If the **r**A field of the instruction specifies **r0**, a value of zero is added to the immediate index (d operand) in place of the contents of **r0**. The option to specify **r**A or 0 is shown in the instruction descriptions as (**r**A|0).

Figure 4-1 shows how an effective address is generated when using register indirect with immediate index addressing.

Figure 4-1. Register Indirect with Immediate Index Addressing for Integer Loads/Stores

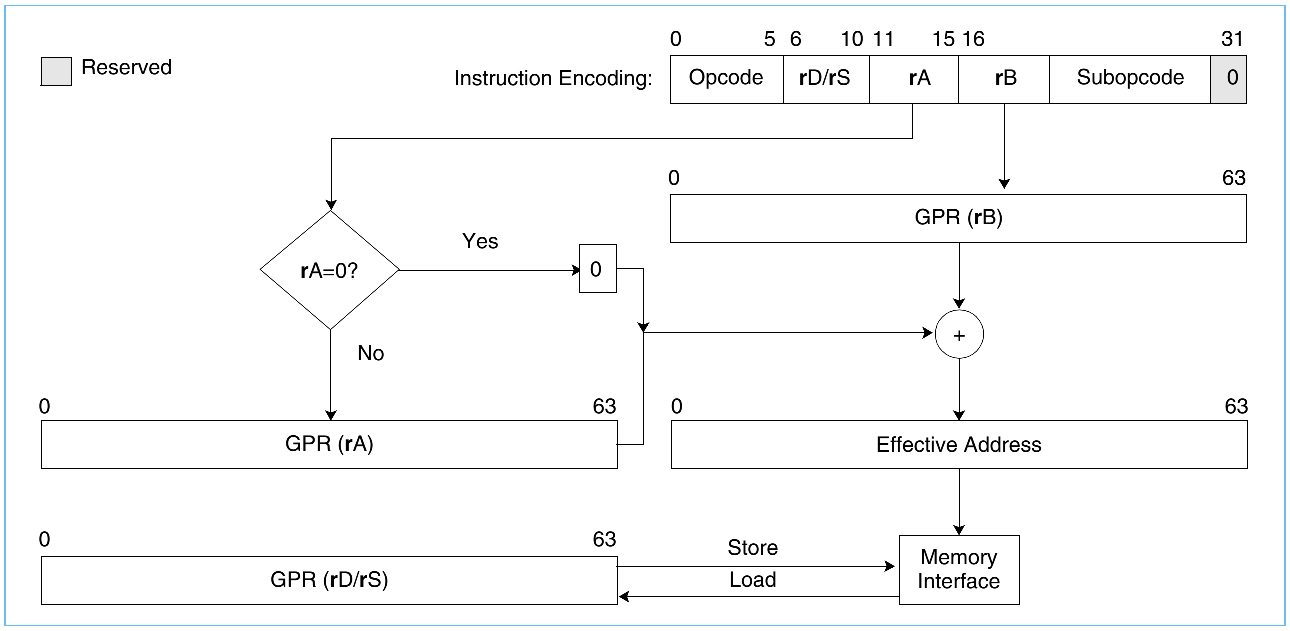


Register Indirect with Index Addressing for Integer Loads and Stores

Instructions using this addressing mode cause the contents of two general-purpose registers (specified as operands **r**A and **r**B) to be added in the generation of the effective address. A zero in place of the **r**A operand causes a zero to be added to the contents of the general-purpose register specified in operand **r**B (or the value zero for **lswi** and **stswi** instructions). The option to specify **r**A or 0 is shown in the instruction descriptions as (**r**A|0).

Figure 4-2 shows how an effective address is generated when using register indirect with index addressing.

Figure 4-2. Register Indirect with Index Addressing for Integer Loads/Stores

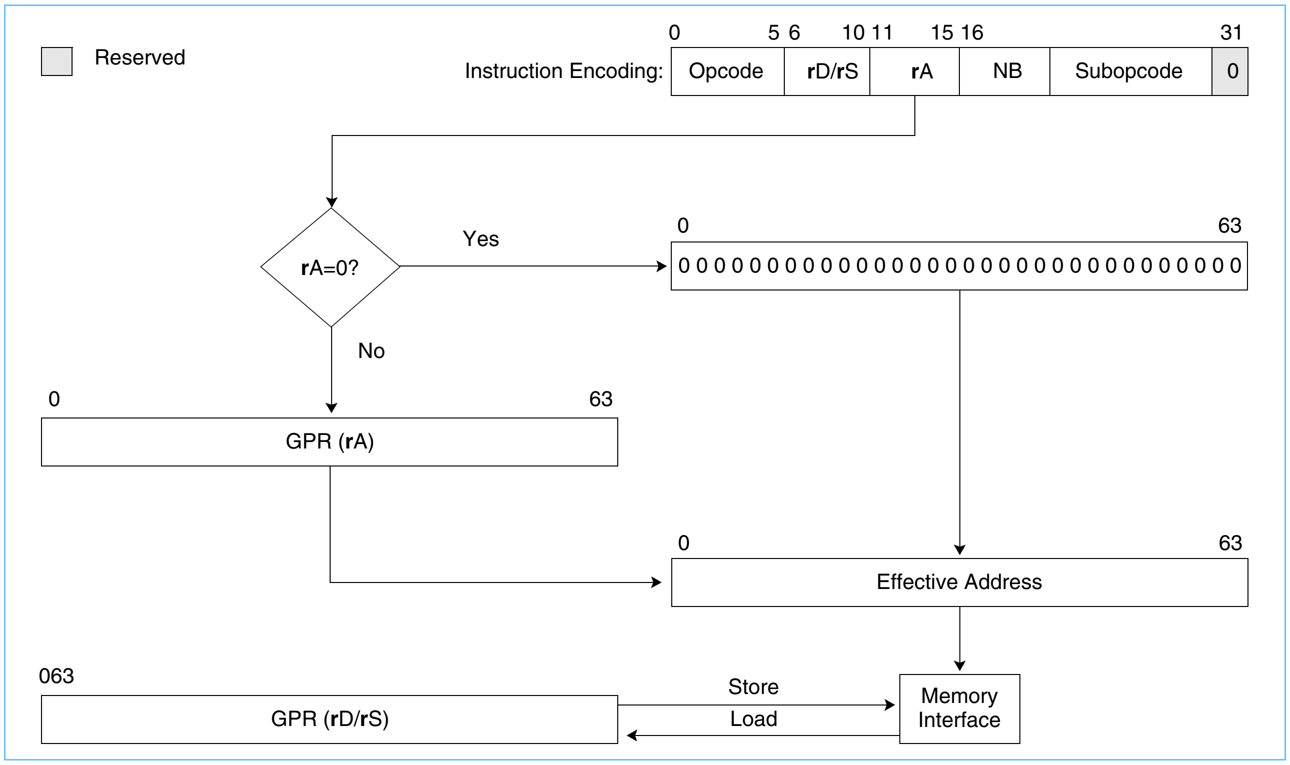


Register Indirect Addressing for Integer Loads and Stores

Instructions using this addressing mode use the contents of the general-purpose register specified by the **r**A operand as the effective address. A zero in the **r**A operand causes an effective address of zero to be generated. The option to specify **r**A or 0 is shown in the instruction descriptions as (**r**A|0).

Figure 4-3 shows how an effective address is generated when using register indirect addressing.

Figure 4-3. Register Indirect Addressing for Integer Loads/Stores



4.2.3.2 Integer Load Instructions

For integer load instructions, the byte, halfword, word, or doubleword addressed by the effective address (EA) is loaded into **r**D. Many integer load instructions have an update form, in which **r**A is updated with the generated effective address. For these forms, if **r**A ≠ 0 and **r**A ≠ **r**D (otherwise invalid), the EA is placed into **r**A and the memory element (byte, halfword, word, or doubleword) addressed by the EA is loaded into **r**D.

Note: The PowerPC Architecture defines load with update instructions with operand **r**A = 0 or **r**A = **r**D as invalid forms.

The default byte and bit ordering is big-endian in the PowerPC Architecture; see *Section 3.1.2 Byte Ordering* for information about little-endian byte ordering.

Note: In some implementations of the architecture, the load algebraic instructions (**lha**, **lhax**, **lwa**, **lwax**) and the load with update (**lbzu**, **lbzux**, **lhzu**, **lhzux**, **lhau**, **lhaux**, **lwaux**, **ldu**, **ldux**) instructions may execute with a greater latency than other types of load instructions. Moreover, the load with update instructions might take longer to execute in some implementations than the corresponding pair of a nonupdate load followed by an add instruction to update the register.

Table 4-13 summarizes the integer load instructions.

Table 4-13. Integer Load Instructions

Name	Mnemonic	Operand Syntax	Operation
Load Byte and Zero	**lbz**	**r**D,d(**r**A)	The EA is the sum (**r**A|0) + d. The byte in memory addressed by the EA is loaded into the low-order eight bits of **r**D. The remaining bits in **r**D are cleared.
Load Byte and Zero Indexed	**lbzx**	**r**D,**r**A,**r**B	The EA is the sum (**r**A|0) + (**r**B). The byte in memory addressed by the EA is loaded into the low-order eight bits of **r**D. The remaining bits in **r**D are cleared.
Load Byte and Zero with Update	**lbzu**	**r**D,d(**r**A)	The EA is the sum (**r**A) + d. The byte in memory addressed by the EA is loaded into the low-order eight bits of **r**D. The remaining bits in **r**D are cleared. The EA is placed into **r**A.
Load Byte and Zero with Update Indexed	**lbzux**	**r**D,**r**A,**r**B	The EA is the sum (**r**A) + (**r**B). The byte in memory addressed by the EA is loaded into the low-order eight bits of **r**D. The remaining bits in **r**D are cleared. The EA is placed into **r**A.
Load Halfword and Zero	**lhz**	**r**D,d(**r**A)	The EA is the sum (**r**A|0) + d. The halfword in memory addressed by the EA is loaded into the low-order 16 bits of **r**D. The remaining bits in **r**D are cleared.
Load Halfword and Zero Indexed	**lhzx**	**r**D,**r**A,**r**B	The EA is the sum (**r**A|0) + (**r**B). The halfword in memory addressed by the EA is loaded into the low-order 16 bits of **r**D. The remaining bits in **r**D are cleared.
Load Halfword and Zero with Update	**lhzu**	**r**D,d(**r**A)	The EA is the sum (**r**A) + d. The halfword in memory addressed by the EA is loaded into the low-order 16 bits of **r**D. The remaining bits in **r**D are cleared. The EA is placed into **r**A.
Load Halfword and Zero with Update Indexed	**lhzux**	**r**D,**r**A,**r**B	The EA is the sum (**r**A) + (**r**B). The halfword in memory addressed by the EA is loaded into the low-order 16 bits of **r**D. The remaining bits in **r**D are cleared. The EA is placed into **r**A.
Load Halfword Algebraic	**lha**	**r**D,d(**r**A)	The EA is the sum (**r**A|0) + d. The halfword in memory addressed by the EA is loaded into the low-order 16 bits of **r**D. The remaining bits in **r**D are filled with a copy of the most significant bit of the loaded halfword.
Load Halfword Algebraic Indexed	**lhax**	**r**D,**r**A,**r**B	The EA is the sum (**r**A|0) + (**r**B). The halfword in memory addressed by the EA is loaded into the low-order 16 bits of **r**D. The remaining bits in **r**D are filled with a copy of the most significant bit of the loaded halfword.

Table 4-13. Integer Load Instructions (Continued)

Name	Mnemonic	Operand Syntax	Operation
Load Halfword Algebraic with Update	**lhau**	rD,d(**r**A)	The EA is the sum (**r**A) + d. The halfword in memory addressed by the EA is loaded into the low-order 16 bits of **r**D. The remaining bits in **r**D are filled with a copy of the most significant bit of the loaded halfword. The EA is placed into **r**A.
Load Halfword Algebraic with Update Indexed	**lhaux**	rD,**r**A,**r**B	The EA is the sum (**r**A) + (**r**B). The halfword in memory addressed by the EA is loaded into the low-order 16 bits of **r**D. The remaining bits in **r**D are filled with a copy of the most significant bit of the loaded halfword. The EA is placed into **r**A.
Load Word and Zero	**lwz**	rD,d(**r**A)	The EA is the sum (**r**A|0) + d. The word in memory addressed by the EA is loaded into the low-order 32 bits of **r**D. The remaining bits in the high-order 32 bits of **r**D are cleared.
Load Word and Zero Indexed	**lwzx**	rD,**r**A,**r**B	The EA is the sum (**r**A|0) + (**r**B). The word in memory addressed by the EA is loaded into the low-order 32 bits of **r**D. The remaining bits in the high-order 32 bits of **r**D are cleared.
Load Word and Zero with Update	**lwzu**	rD,d(**r**A)	The EA is the sum (**r**A) + d. The word in memory addressed by the EA is loaded into the low-order 32 bits of **r**D. The remaining bits in the high-order 32 bits of **r**D are cleared. The EA is placed into **r**A.
Load Word and Zero with Update Indexed	**lwzux**	rD,**r**A,**r**B	The EA is the sum (**r**A) + (**r**B). The word in memory addressed by the EA is loaded into the low-order 32 bits of **r**D. The remaining bits in the high-order 32 bits of **r**D are cleared. The EA is placed into **r**A.
Load Word Algebraic	**lwa**	rD,ds(**r**A)	The EA is the sum (**r**A|0) + (ds||'00'). The word in memory addressed by the EA is loaded into the low-order 32 bits of **r**D. The remaining bits in the high-order 32 bits of **r**D are filled with a copy of the most significant bit of the loaded word.
Load Word Algebraic Indexed	**lwax**	rD,**r**A,**r**B	The EA is the sum (**r**A|0) + (**r**B). The word in memory addressed by the EA is loaded into the low-order 32 bits of **r**D. The remaining bits in the high-order 32 bits of **r**D are filled with a copy of the most significant bit of the loaded word.
Load Word Algebraic with Update Indexed	**lwaux**	rD,**r**A,**r**B	The EA is the sum (**r**A) + (**r**B). The word in memory addressed by the EA is loaded into the low-order 32 bits of **r**D. The remaining bits in the high-order 32 bits of **r**D are filled with a copy of the most significant bit of the loaded word. The EA is placed into **r**A.
Load Doubleword	**ld**	rD,ds(**r**A)	The EA is the sum (**r**A|0) + (ds||'00'). The doubleword in memory addressed by the EA is loaded into **r**D.
Load Doubleword Indexed	**ldx**	rD,**r**A,**r**B	The EA is the sum (**r**A|0) + (**r**B). The doubleword in memory addressed by the EA is loaded into **r**D.
Load Doubleword with Update	**ldu**	rD,ds(**r**A)	The EA is the sum (**r**A) + (ds||'00'). The doubleword in memory addressed by the EA is loaded into **r**D. The EA is placed into **r**A.
Load Doubleword with Update Indexed	**ldux**	rD,**r**A,**r**B	The EA is the sum (**r**A) + (**r**B). The doubleword in memory addressed by the EA is loaded into **r**D. The EA is placed into **r**A.

4.2.3.3 Integer Store Instructions

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

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

In general, the PowerPC Architecture defines a sequential execution model. However, when a store instruction modifies a memory location that contains an instruction, software synchronization (**isync**)is required to ensure that subsequent instruction fetches from that location obtain the modified version of the instruction.

If a program modifies the instructions it intends to execute, it should call the appropriate system library program before attempting to execute the modified instructions to ensure that the modifications have taken effect with respect to instruction fetching.

The PowerPC Architecture defines store with update instructions with **r**A = '0' as an invalid form. In addition, it defines integer store instructions with the CR update option enabled (Rc field, bit [31], in the instruction encoding = '1') to be an invalid form. *Table 4-14* provides a summary of the integer store instructions.

Table 4-14. Integer Store Instructions

Name	Mnemonic	Operand Syntax	Operation
Store Byte	**stb**	rS,d(**r**A)	The EA is the sum (**r**A|0) + d. The contents of the low-order eight bits of **r**S are stored into the byte in memory addressed by the EA.
Store Byte Indexed	**stbx**	rS,**r**A,**r**B	The EA is the sum (**r**A|0) + (**r**B). The contents of the low-order eight bits of **r**S are stored into the byte in memory addressed by the EA.
Store Byte with Update	**stbu**	rS,d(**r**A)	The EA is the sum (**r**A) + d. The contents of the low-order eight bits of **r**S are stored into the byte in memory addressed by the EA. The EA is placed into **r**A.
Store Byte with Update Indexed	**stbux**	rS,**r**A,**r**B	The EA is the sum (**r**A) + (**r**B). The contents of the low-order eight bits of **r**S are stored into the byte in memory addressed by the EA. The EA is placed into **r**A.
Store Halfword	**sth**	rS,d(**r**A)	The EA is the sum (**r**A|0) + d. The contents of the low-order 16 bits of **r**S are stored into the halfword in memory addressed by the EA.
Store Halfword Indexed	**sthx**	rS,**r**A,**r**B	The EA is the sum (**r**A|0) + (**r**B). The contents of the low-order 16 bits of **r**S are stored into the halfword in memory addressed by the EA.
Store Halfword with Update	**sthu**	rS,d(**r**A)	The EA is the sum (**r**A) + d. The contents of the low-order 16 bits of **r**S are stored into the halfword in memory addressed by the EA. The EA is placed into **r**A.
Store Halfword with Update Indexed	**sthux**	rS,**r**A,**r**B	The EA is the sum (**r**A) + (**r**B). The contents of the low-order 16 bits of **r**S are stored into the halfword in memory addressed by the EA. The EA is placed into **r**A.
Store Word	**stw**	rS,d(**r**A)	The EA is the sum (**r**A|0) + d. The contents of the low-order 32 bits of **r**S are stored into the word in memory addressed by the EA.
Store Word Indexed	**stwx**	rS,**r**A,**r**B	The EA is the sum (**r**A|0) + (**r**B). The contents of the low-order 32 bits of **r**S are stored into the word in memory addressed by the EA.
Store Word with Update	**stwu**	rS,d(**r**A)	The EA is the sum (**r**A) + d. The contents of the low-order 32 bits of **r**S are stored into the word in memory addressed by the EA. The EA is placed into **r**A.

Table 4-14. Integer Store Instructions (Continued)

Name	Mnemonic	Operand Syntax	Operation
Store Word with Update Indexed	**stwux**	**r**S,**r**A,**r**B	The EA is the sum (**r**A) + (**r**B). The contents of the low-order 32 bits of **r**S are stored into the word in memory addressed by the EA. The EA is placed into **r**A.
Store Doubleword	**std**	**r**S,ds(**r**A)	The EA is the sum (**r**A|0) + (ds||'00'). The contents of **r**S are stored into the doubleword in memory addressed by the EA.
Store Doubleword Indexed	**stdx**	**r**S,**r**A,**r**B	The EA is the sum (**r**A|0) + (**r**B). The contents of **r**S are stored into the doubleword in memory addressed by the EA.
Store Doubleword with Update	**stdu**	**r**S,ds(**r**A)	The EA is the sum (**r**A) + (ds||'00'). The contents of **r**S are stored into the doubleword in memory addressed by the EA. The EA is placed into **r**A.
Store Doubleword with Update Indexed	**stdux**	**r**S,**r**A,**r**B	The EA is the sum (**r**A) + (**r**B). The contents of **r**S are stored into the doubleword in memory addressed by the EA. The EA is placed into **r**A.

4.2.3.4 Integer Load and Store with Byte-Reverse Instructions

Table 4-15 describes integer load and store with byte-reverse instructions. Note that in some PowerPC implementations, load byte-reverse instructions might have a greater latency than other load instructions.

When used in a PowerPC system operating with the default big-endian byte order, these instructions have the effect of loading and storing data in little-endian order. Likewise, when used in a PowerPC system operating with little-endian byte order, these instructions have the effect of loading and storing data in big-endian order. For more information about big-endian and little-endian byte ordering, see *Section 3.1.2 Byte Ordering*.

Table 4-15. Integer Load and Store with Byte-Reverse Instructions

Name	Mnemonic	Operand Syntax	Operation
Load Halfword Byte-Reverse Indexed	**lhbrx**	**r**D,**r**A,**r**B	The EA is the sum (**r**A|0) + (**r**B). The high-order eight bits of the halfword addressed by the EA are loaded into the low-order eight bits of **r**D. The next eight higher-order bits of the halfword in memory addressed by the EA are loaded into the next eight lower-order bits of **r**D. The remaining **r**D bits are cleared.
Load Word Byte-Reverse Indexed	**lwbrx**	**r**D,**r**A,**r**B	The EA is the sum (**r**A|0) + (**r**B). Bits [0–7] of the word in memory addressed by the EA are loaded into the low-order eight bits of **r**D. Bits [8–15] of the word in memory addressed by the EA are loaded into bits [48–55] of **r**D. Bits [16-23] of the word in memory addressed by the EA are loaded into bits [40–47] of **r**D. Bits [24–31] of the word in memory addressed by the EA are loaded into bits [32–39] of **r**D. The remaining bits in **r**D are cleared.
Store Halfword Byte- Reverse Indexed	**sthbrx**	**r**S,**r**A,**r**B	The EA is the sum (**r**A|0) + (**r**B). The contents of the low-order eight bits of **r**S are stored into the high-order eight bits of the halfword in memory addressed by the EA. The contents of the next lower-order eight bits of **r**S are stored into the next eight higher-order bits of the halfword in memory addressed by the EA.
Store Word Byte-Reverse Indexed	**stwbrx**	**r**S,**r**A,**r**B	The effective address is the sum (**r**A|0) + (**r**B). The contents of the low-order eight bits of **r**S are stored into bits [0–7] of the word in memory addressed by EA. The contents of the next eight lower-order bits of **r**S are stored into bits [8–15] of the word in memory addressed by the EA. The contents of the next eight lower-order bits of **r**S are stored into bits [16-23] of the word in memory addressed by the EA. The contents of the next eight lower-order bits of **r**S are stored into bits [24–31] of the word addressed by the EA.

4.2.3.5 Integer Load and Store Multiple Instructions

The load/store multiple instructions are used to move blocks of data to and from the GPRs. The load multiple and store multiple instructions may have operands that require memory accesses crossing a 4-Kbyte page boundary. As a result, these instructions may be interrupted by a DSI exception associated with the address translation of the second page. *Table 4-16* summarizes the integer load and store multiple instructions.

In the load/store multiple instructions, the combination of the EA and **r**D (**r**S) is such that the low-order byte of GPR31 is loaded from or stored into the last byte of an aligned quad word in memory; if the effective address is not correctly aligned, it may take significantly longer to execute.

In some PowerPC implementations operating with little-endian byte order, execution of an **lmw** or **stmw** instruction causes the system alignment error handler to be invoked; see *Section 3.1.2 Byte Ordering* for more information.

The PowerPC Architecture defines the load multiple word (**lmw**) instruction with **r**A in the range of registers to be loaded, including the case in which **r**A = '0' as an invalid form.

Table 4-16. Integer Load and Store Multiple Instructions

Name	Mnemonic	Operand Syntax	Operation
Load Multiple Word	**lmw**	**r**D,d(**r**A)	The EA is the sum (**r**A|0) + d. *n* = (32 – **r**D).
Store Multiple Word	**stmw**	**r**S,d(**r**A)	The EA is the sum (**r**A|0) + d. *n* = (32 – **r**S).

4.2.3.6 Integer Load and Store String Instructions

The integer load and store string instructions allow movement of data from memory to registers or from registers to memory without concern for alignment. These instructions can be used for a short move between arbitrary memory locations or to initiate a long move between misaligned memory fields. However, in some implementations, these instructions are likely to have greater latency and take longer to execute, perhaps much longer, than a sequence of individual load or store instructions that produce the same results. *Table 4-17* summarizes the integer load and store string instructions.

Load and store string instructions execute more efficiently when **r**D or **r**S = '4' or '5', and the last register loaded or stored is less than or equal to '12'.

In some PowerPC implementations operating with little-endian byte order, execution of a load or string instruction causes the system alignment error handler to be invoked; see *Section 3.1.2 Byte Ordering* for more information.

Table 4-17. Integer Load and Store String Instructions

Name	Mnemonic	Operand Syntax	Operation
Load String Word Immediate	**lswi**	**r**D,**r**A,NB	The EA is (**r**A|0).
Load String Word Indexed	**lswx**	**r**D,**r**A,**r**B	The EA is the sum (**r**A|0) + (**r**B).
Store String Word Immediate	**stswi**	**r**S,**r**A,NB	The EA is (**r**A|0).
Store String Word Indexed	**stswx**	**r**S,**r**A,**r**B	The EA is the sum (**r**A|0) + (**r**B).

Load string and store string instructions may involve operands that are not word-aligned. As described in *Section 6.4.8 Alignment Exception (0x00600)*, a misaligned string operation suffers a performance penalty compared to an aligned operation of the same type. A nonword-aligned string operation that crosses a doubleword boundary is also slower than a word-aligned string operation.

4.2.3.7 Floating-Point Load and Store Address Generation

Floating-point load and store operations generate effective addresses using the register indirect with immediate index addressing mode and register indirect with index addressing mode.

The following sections discuss index addressing for floating-point loads and stores:

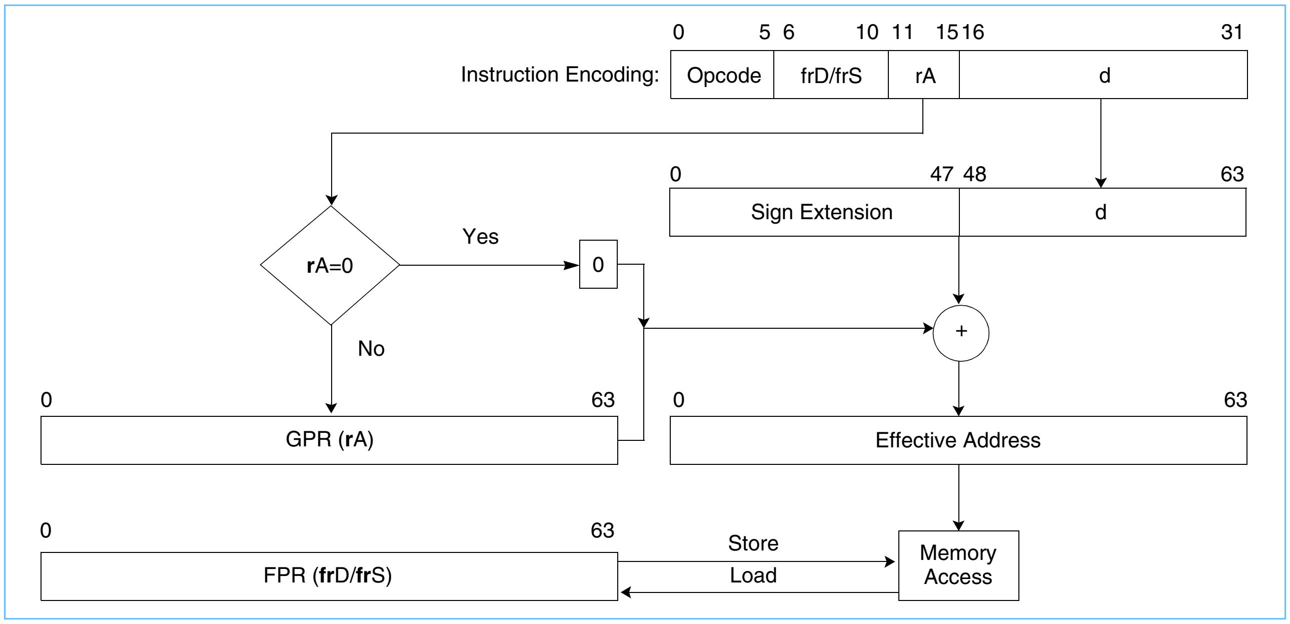
- Register Indirect with Immediate Index Addressing for Floating-point Loads and Stores
- Register Indirect with Index Addressing for Floating-point Loads and Stores

Register Indirect with Immediate Index Addressing for Floating-Point Loads and Stores

Instructions using this addressing mode contain a signed 16-bit immediate index (d operand) which is sign extended to 64 bits, and added to the contents of a GPR specified in the instruction (**r**A operand) to generate the effective address. If the **r**A field of the instruction specifies **r0**, a value of zero is added to the immediate index (d operand) in place of the contents of **r0**. The option to specify **r**A or ‘0’ is shown in the instruction descriptions as (**r**A|0).

Figure 4-4 shows how an effective address is generated when using register indirect with immediate index addressing for floating-point loads and stores.

Figure 4-4. Register Indirect (Contents) with Immediate Index Addressing for Floating-Point Loads/Stores

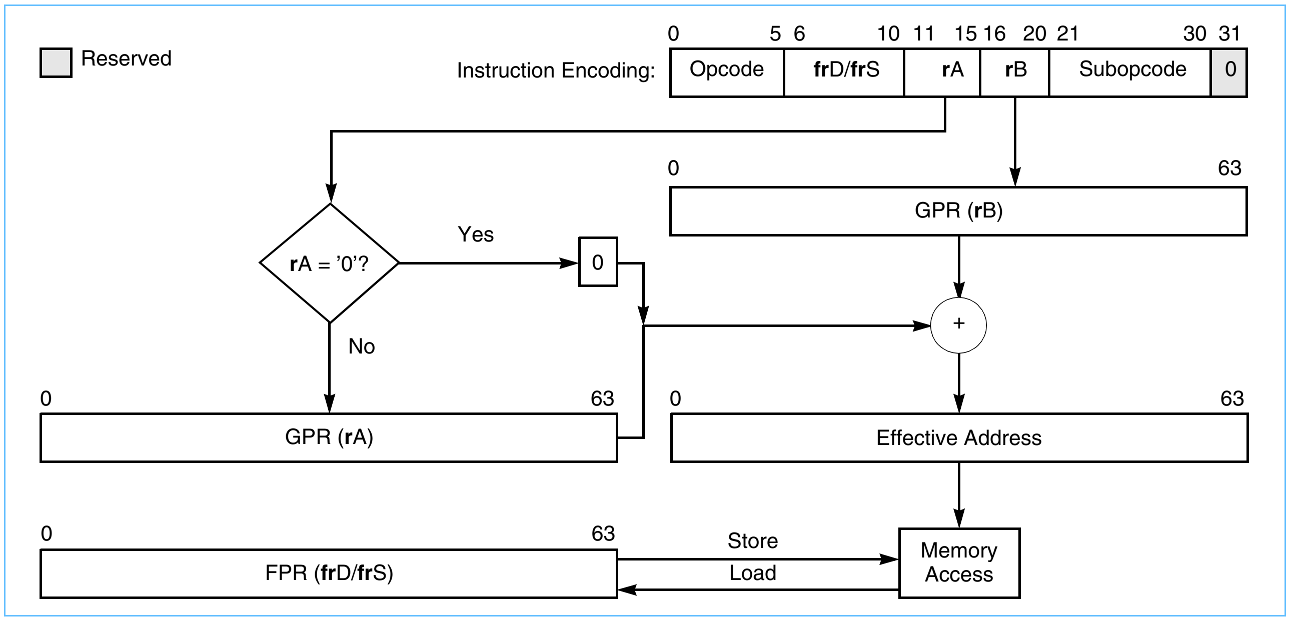


Register Indirect with Index Addressing for Floating-Point Loads and Stores

Instructions using this addressing mode add the contents of two GPRs (specified in operands **r**A and **r**B) to generate the effective address. A zero in the **r**A operand causes a zero to be added to the contents of the GPR specified in operand **r**B. This is shown in the instruction descriptions as (**r**A|0).

Figure 4-5 shows how an effective address is generated when using register indirect with index addressing.

Figure 4-5. Register Indirect with Index Addressing for Floating-Point Loads/Stores



The PowerPC Architecture defines floating-point load and store with update instructions (**lfsu**, **lfsux**, **lfdu**, **lfdux**, **stfsu**, **stfsux**, **stfdu**, **stfdux**) with operand **r**A = ’0’ as invalid forms of the instructions. In addition, it defines floating-point load and store instructions with the CR updating option enabled (Rc bit, bit [31] = ’1’) to be an invalid form.

The PowerPC Architecture defines that the FPSCR[UE] bit should not be used to determine whether denormalization should be performed on floating-point stores.

4.2.3.8 Floating-Point Load Instructions

There are two forms of the floating-point load instruction—single-precision and double-precision operand formats. Because the FPRs support only the floating-point double-precision format, single-precision floating-point load instructions convert single-precision data to double-precision format before loading the operands into the target FPR. This conversion is described fully in *Appendix C.6 Floating-Point Load Instructions*. *Table 4-18* provides a summary of the floating-point load instructions.

Note: The PowerPC Architecture defines load with update instructions with **r**A = ’0’ as an invalid form.

Table 4-18. Floating-Point Load Instructions

Name	Mnemonic	Operand Syntax	Operation
Load Floating-Point Single	**lfs**	**fr**D,d(**r**A)	The EA is the sum (**r**A|0) + d. The word in memory addressed by the EA is interpreted as a floating-point single-precision operand. This word is converted to floating-point double-precision format and placed into **fr**D.
Load Floating-Point Single Indexed	**lfsx**	**fr**D,**r**A,**r**B	The EA is the sum (**r**A|0) + (**r**B). The word in memory addressed by the EA is interpreted as a floating-point single-precision operand. This word is converted to floating-point double-precision format and placed into **fr**D.
Load Floating-Point Single with Update	**lfsu**	**fr**D,d(**r**A)	The EA is the sum (**r**A) + d. The word in memory addressed by the EA is interpreted as a floating-point single-precision operand. This word is converted to floating-point double-precision format and placed into **fr**D. The EA is placed into the register specified by **r**A.
Load Floating-Point Single with Update Indexed	**lfsux**	**fr**D,**r**A,**r**B	The EA is the sum (**r**A) + (**r**B). The word in memory addressed by the EA is interpreted as a floating-point single-precision operand. This word is converted to floating-point double-precision format and placed into **fr**D. The EA is placed into the register specified by **r**A.
Load Floating-Point Double	**lfd**	**fr**D,d(**r**A)	The EA is the sum (**r**A|0) + d. The doubleword in memory addressed by the EA is placed into register **fr**D.
Load Floating-Point Double Indexed	**lfdx**	**fr**D,**r**A,**r**B	The EA is the sum (**r**A|0) + (**r**B). The doubleword in memory addressed by the EA is placed into register **fr**D.
Load Floating-Point Double with Update	**lfdu**	**fr**D,d(**r**A)	The EA is the sum (**r**A) + d. The doubleword in memory addressed by the EA is placed into register **fr**D. The EA is placed into the register specified by **r**A.
Load Floating-Point Double with Update Indexed	**lfdux**	**fr**D,**r**A,**r**B	The EA is the sum (**r**A) + (**r**B). The doubleword in memory addressed by the EA is placed into register **fr**D. The EA is placed into the register specified by **r**A.

4.2.3.9 Floating-Point Store Instructions

This section describes floating-point store instructions. There are three basic forms of the store instruction—single-precision, double-precision, and integer. The integer form is supported by the **stfiwx** instruction.

Because the FPRs support only floating-point, double-precision format for floating-point data, single-precision floating-point store instructions convert double-precision data to single-precision format before storing the operands. The conversion steps are described fully in *Appendix C.7 Floating-Point Store Instructions*. *Table 4-19* provides a summary of the floating-point store instructions.

Note: The PowerPC Architecture defines store with update instructions with **r**A = ’0’ as an invalid form.

Table 4-19 provides the floating-point store instructions for the PowerPC processors.

Table 4-19. Floating-Point Store Instructions

Name	Mnemonic	Operand Syntax	Operation
Store Floating-Point Single	**stfs**	**fr**S,d(**r**A)	The EA is the sum (**r**A|0) + d. The contents of **fr**S are converted to single-precision and stored into the word in memory addressed by the EA.
Store Floating-Point Single Indexed	**stfsx**	**fr**S,**r**A,**r**B	The EA is the sum (**r**A|0) + (**r**B). The contents of **fr**S are converted to single-precision and stored into the word in memory addressed by the EA.
Store Floating-Point Single with Update	**stfsu**	**fr**S,d(**r**A)	The EA is the sum (**r**A) + d. The contents of **fr**S are converted to single-precision and stored into the word in memory addressed by the EA. The EA is placed into **r**A.
Store Floating-Point Single with Update Indexed	**stfsux**	**fr**S,**r**A,**r**B	The EA is the sum (**r**A) + (**r**B). The contents of **fr**S are converted to single-precision and stored into the word in memory addressed by the EA. The EA is placed into the **r**A.
Store Floating-Point Double	**stfd**	**fr**S,d(**r**A)	The EA is the sum (**r**A|0) + d. The contents of **fr**S are stored into the doubleword in memory addressed by the EA.
Store Floating-Point Double Indexed	**stfdx**	**fr**S,**r**A,**r**B	The EA is the sum (**r**A|0) + (**r**B). The contents of **fr**S are stored into the doubleword in memory addressed by the EA.
Store Floating-Point Double with Update	**stfdu**	**fr**S,d(**r**A)	The EA is the sum (**r**A) + d. The contents of **fr**S are stored into the doubleword in memory addressed by the EA. The EA is placed into **r**A.
Store Floating-Point Double with Update Indexed	**stfdux**	**fr**S,**r**A,**r**B	The EA is the sum (**r**A) + (**r**B). The contents of **fr**S are stored into the doubleword in memory addressed by EA. The EA is placed into register **r**A.
Store Floating-Point as Integer Word Indexed	**stfiwx**	**fr**S,**r**A,**r**B	The EA is the sum (**r**A|0) + (**r**B). The contents of the low-order 32 bits of **fr**S are stored, without conversion, into the word in memory addressed by the EA.

4.2.4 Branch and Flow Control Instructions

Some branch instructions can redirect instruction execution conditionally based on the value of bits in the CR. When the processor encounters one of these instructions, it can attempt to resolve the branch direction immediately, or predict the branch direction and defer its resolution.

When the branch cannot be resolved immediately, it may be predicted based on the 'at' bits (as described in *Table 4-20* and *Table 4-21*), or by using dynamic prediction. At some point before the branch instruction can complete, the branch direction will be resolved based on the value of the CR bit. If the prediction is correct, the branch is considered completed and instruction fetching continues along the predicted path. If the prediction is incorrect, the fetched instructions are purged, and instruction fetching continues along the alternate path.

4.2.4.1 Branch Instruction Address Calculation

Branch instructions can alter the sequence of instruction execution. Instruction addresses are always assumed to be word aligned; the PowerPC processors ignore the two low-order bits (bits [62, 63]) of the generated branch target address.

Branch instructions compute the effective address (EA) of the next instruction address using the following addressing modes:

- Branch relative
- Branch conditional to relative address
- Branch to absolute address
- Branch conditional to absolute address
- Branch conditional to link register
- Branch conditional to count register

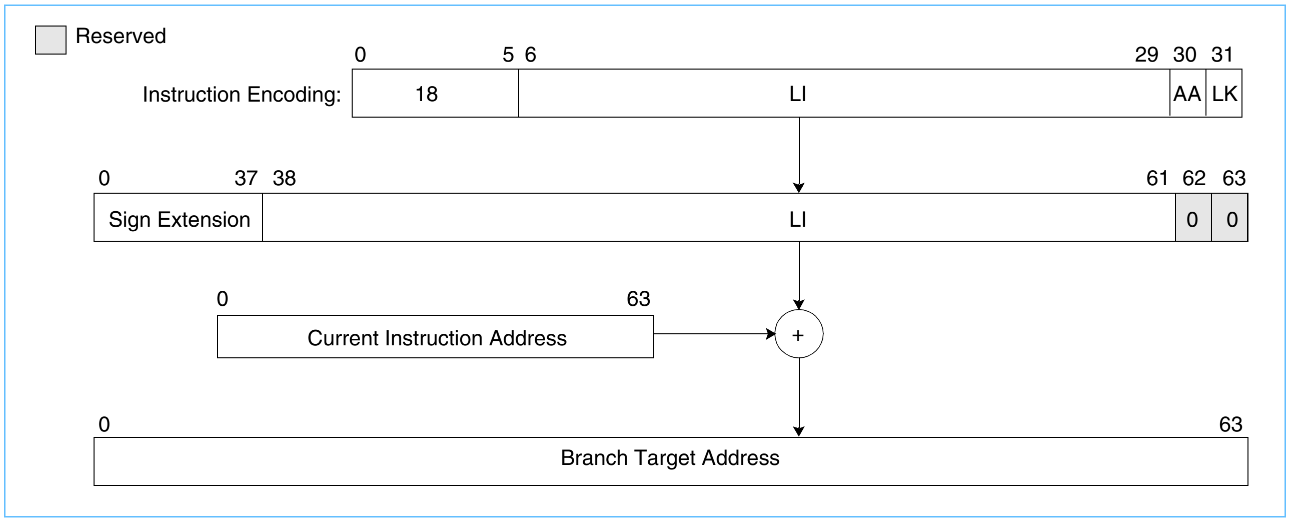
In the 32-bit mode of a 64-bit implementation, the final step in the address computation is clearing the high-order 32 bits of the target address.

Branch Relative Addressing Mode

Instructions that use branch relative addressing generate the next instruction address by sign extending and appending ‘00’ to the immediate displacement operand LI, and adding the resultant value to the current instruction address. Branches using this addressing mode have the absolute addressing option disabled (AA field, bit [30], in the instruction encoding = ‘0’). The link register (LR) update option can be enabled (LK field, bit [31], in the instruction encoding = ‘1’). This option causes the effective address of the instruction following the branch instruction to be placed in the LR.

Figure 4-6 shows how the branch target address is generated when using the branch relative addressing mode.

Figure 4-6. Branch Relative Addressing

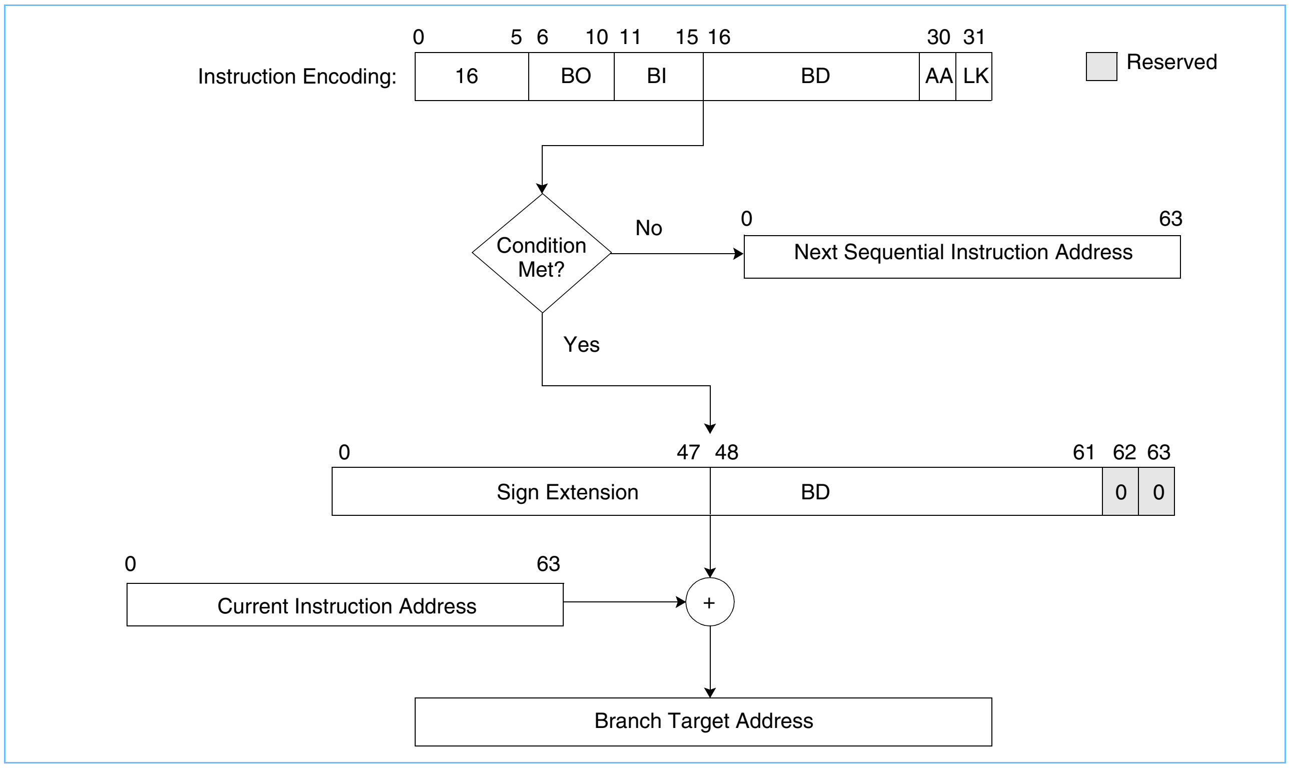


Branch Conditional to Relative Addressing Mode

If the branch conditions are met, instructions that use the branch conditional to relative addressing mode generate the next instruction address by sign extending and appending ‘00’ to the immediate displacement operand (BD) and adding the resultant value to the current instruction address. Branches using this addressing mode have the absolute addressing option disabled (AA field, bit[30], in the instruction encoding = ‘0’). The link register update option can be enabled (LK field, bit[31], in the instruction encoding = ‘1’). This option causes the effective address of the instruction following the branch instruction to be placed in the LR.

Figure 4-7 shows how the branch target address is generated when using the branch conditional relative addressing mode.

Figure 4-7. Branch Conditional Relative Addressing

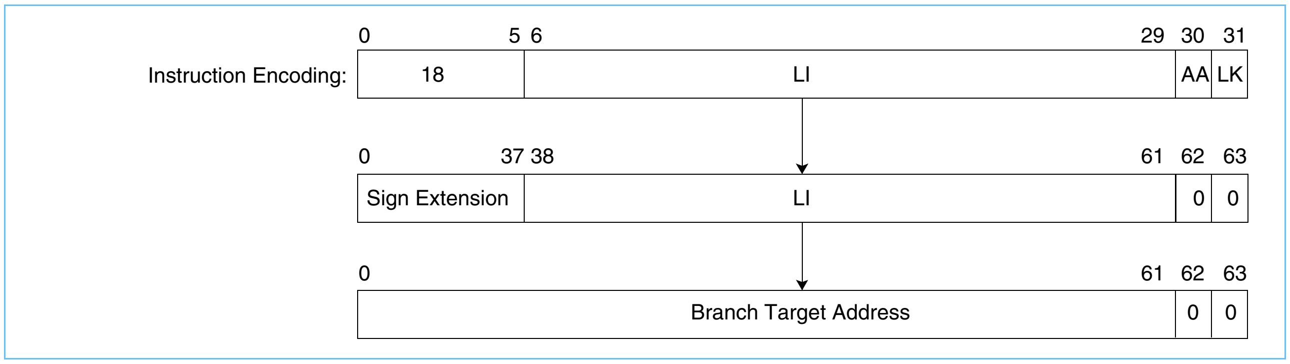


Branch to Absolute Addressing Mode

Instructions that use branch to absolute addressing mode generate the next instruction address by sign extending and appending ‘00’ to the LI operand. Branches using this addressing mode have the absolute addressing option enabled (AA field, bit[30], in the instruction encoding = ‘1’). The link register update option can be enabled (LK field, bit[31], in the instruction encoding = ’1’). This option causes the effective address of the instruction following the branch instruction to be placed in the LR.

Figure 4-8 shows how the branch target address is generated when using the branch to absolute addressing mode.

Figure 4-8. Branch to Absolute Addressing

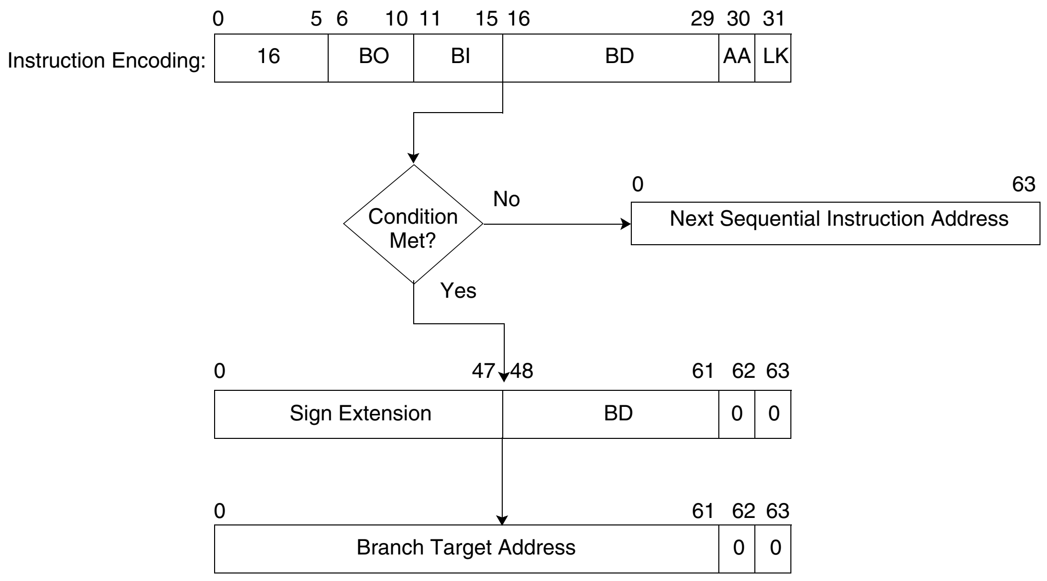


Branch Conditional to Absolute Addressing Mode

If the branch conditions are met, instructions that use the branch conditional to absolute addressing mode generate the next instruction address by sign extending and appending ‘00’ to the BD operand. Branches using this addressing mode have the absolute addressing option enabled (AA field, bit[30], in the instruction encoding = ’1’). The link register update option can be enabled (LK field, bit[31], in the instruction encoding = ‘1’). This option causes the effective address of the instruction following the branch instruction to be placed in the LR.

Figure 4-9 shows how the branch target address is generated when using the branch conditional to absolute addressing mode.

Figure 4-9. Branch Conditional to Absolute Addressing



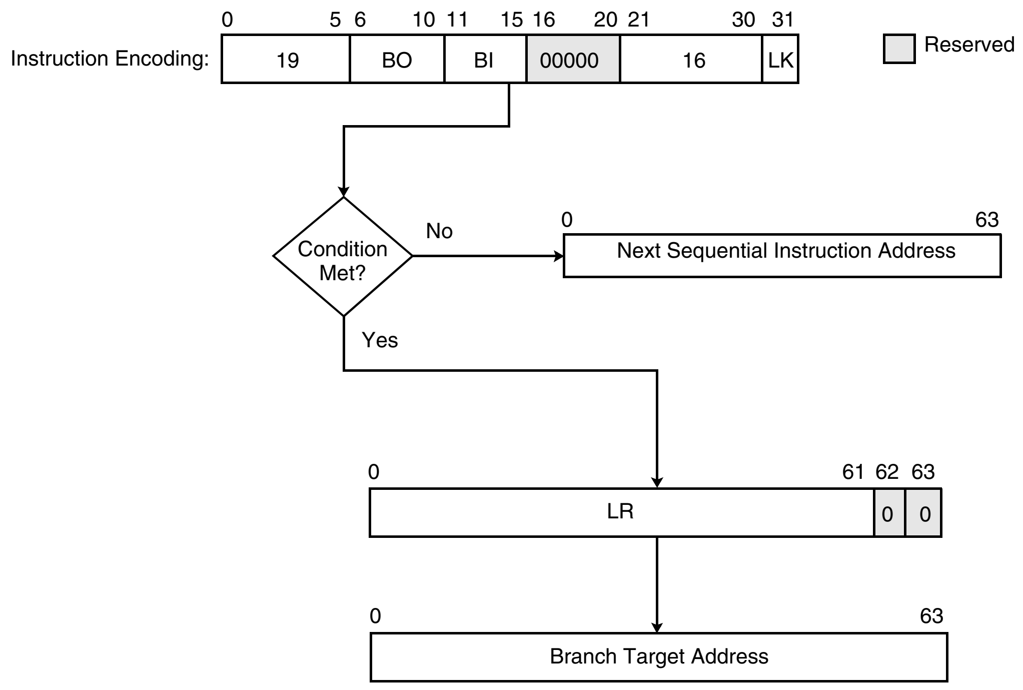
Branch Conditional to Link Register Addressing Mode

If the branch conditions are met, the branch conditional to link register instruction generates the next instruction address by using the contents of the LR and clearing the two low-order bits to zero. The result becomes the effective address from which the next instructions are fetched.

The link register update option can be enabled (LK field, bit[31], in the instruction encoding = ’1’). This option causes the effective address of the instruction following the branch instruction to be placed in the LR. This is done even if the branch is not taken.

Figure 4-10 shows how the branch target address is generated when using the branch conditional to link register addressing mode.

Figure 4-10. Branch Conditional to Link Register Addressing



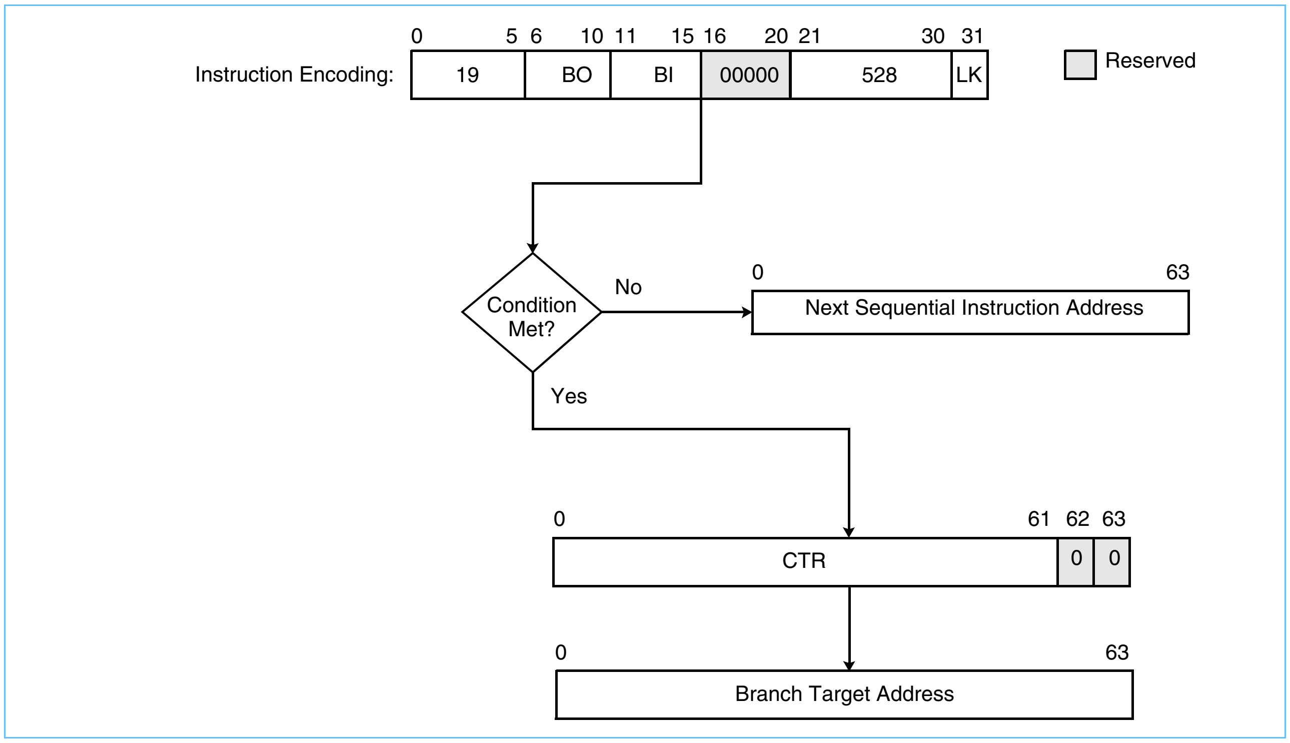
Branch Conditional to Count Register Addressing Mode

If the branch conditions are met, the branch conditional to count register instruction generates the next instruction address by using the contents of the count register (CTR) and clearing the two low-order bits to zero. The result becomes the effective address from which the next instructions are fetched.

The link register update option can be enabled (LK field, bit[31], in the instruction encoding = ‘1’). This option causes the effective address of the instruction following the branch instruction to be placed in the LR. This is done even if the branch is not taken.

Figure 4-11 shows how the branch target address is generated when using the branch conditional to count register addressing mode.

Figure 4-11. Branch Conditional to Count Register Addressing



4.2.4.2 Conditional Branch Control

For branch conditional instructions, the BO operand specifies the conditions under which the branch is taken. The encodings for the BO operands are shown in *Table 4-20*. M = ‘32’ in 32-bit mode (of a 64-bit implementation) and M = ‘0’ in the default 64-bit mode. If the BO field specifies that the CTR is to be decremented, the entire 64-bit CTR is decremented regardless of the 32-bit mode or the default 64-bit mode.

Table 4-20. BO Operand Encodings

BO	Description
0000*z*	Decrement the CTR, then branch if the decremented CTR[M–63] ≠ ‘0’ and CR[BI] = ‘0’ (condition is false).
0001*z*	Decrement the CTR, then branch if the decremented CTR[M–63] = ’0’ and CR[BI] = ‘0’ (condition is false).
001*at*	Branch if CR[BI] = 0 (false).
0100*z*	Decrement the CTR, then branch if the decremented CTR[M–63] ≠’0’ and CR[BI] = ’1’ (condition is true).
0101*z*	Decrement the CTR, then branch if the decremented CTR[M–63] = ’0’ and CR[BI] = ’1’ (condition is true).
011*at*	Branch if CR[BI] = ’1’ (condition is true).
1*a*00*t*	Decrement the CTR, then branch if the decremented CTR[M–63] ≠ ‘0’.
1*a*01*t*	Decrement the CTR, then branch if the decremented CTR[M–63] = ’0’.
1*z*1*zz*	Branch always.

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

The “a” and ”t” bits of the BO field can be used by software to provide a hint about whether the branch is likely to be taken or is likely not to be taken (see *Table 4-21*).

Table 4-21. “a” and “t” Bits of the BO Field

“a” “t”	Hint
00	No hint is given
01	Reserved
10	Branch is very likely not to be taken
11	Branch is very likely to be taken

Note: Many implementations have dynamic mechanisms for predicting whether a branch will be taken. Because the dynamic prediction is likely to be very accurate, and is likely to be overridden by any hint provided by the “at” bits, the “at” bits should be set to ‘00’ unless the static prediction implied by at=’10’ or at=’11’ is very likely to be correct.

For Branch Conditional to Link Register and Branch Conditional to Count Register instructions, the BH field provides a hint about the use of the instruction, as shown in *Table 4-22*.

Table 4-22. BH Field Encodings

BH	Hint
00	**bclr**[**l**]: The instruction is a subroutine return **bcctr**[**l**]: The instruction is not a subroutine return; the target address is likely to be the same as the target address used the preceding time the branch was taken.
01	**bclr**[**l**]: The instruction is not a subroutine return; the target address is likely to be the same as the target address used the preceding time the branch was taken. **bcctr**[**l**]: Reserved.
10	Reserved.
11	**bclr**[**l**] and **bcctr**[**l**]: The target address is not predictable.

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

The 5-bit BI operand in branch conditional instructions specifies which of the 32 bits in the CR represents the bit to test.

The 5-bit BO and BI fields control whether the branch is taken.

When the branch instructions contain immediate addressing operands, the branch target addresses can be computed sufficiently ahead of the branch execution and instructions can be fetched along the branch target path (if the branch is predicted to be taken or is an unconditional branch). If the branch instructions use the link or count register contents for the branch target address, instructions along the branch-taken path of a branch can be fetched if the link or count register is loaded sufficiently ahead of the branch instruction execution.

Branching can be conditional or unconditional. The branch target address is first calculated from the contents of the count or link register or from the branch immediate field. Optionally, a branch return address can be loaded into the LR register (this sets the return address for subroutine calls). When this option is selected (LK = ’1’) the LR is loaded with the effective address of the instruction following the branch instruction.

Some processors 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.

In the following examples, let A, B, and Glue represent subroutine labels:

- Obtaining the address of the next instruction–use the following form of branch and link:
  bcl 20,31,$+4
- Loop counts:
  Keep loop counts in the count register, and use one of the branch conditional instructions (LK = ’0’) to decrement the count and to control branching (for example, branching back to the start of a loop if the decremented counter value is nonzero).
- Computed GOTOs, case statements, etc.:
  Use the count register to hold the address to branch to, and use the **bcctr** instruction with the link register option disabled (LK = ‘0’ and BH = ‘11’ if appropriate) to branch to the selected address.
- Direct subroutine linkage—where A calls B and B returns to A. The two branches should be as follows:
  – A calls B: use a branch instruction (**bl**, **bcl**) that enables the link register (LK = ’1’).

  - B returns to A: use the **bclr** instruction with the link register option disabled (LK = ’0’) (the return address is in, or can be restored to, the link register).

- Indirect subroutine linkage:
  Where 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 (**bl**, **bcl**) that sets the link register with the link register option enabled (LK = ’1’).
  - Glue calls B: place the address of B in the count register, and use the **bcctr** instruction with the link register option disabled (LK = ‘0’).
  - B returns to A: use the **bclr** instruction with the link register option disabled (LK = ’0’) (the return address is in, or can be restored to, the link register).

- Function call:
  Here A calls a function, the identity of which may vary from one instance of the call to another, instead of calling a specific program B. This case should be handled using the conventions of the preceding two bullets, depending on whether the call is direct or indirect, with the following differences.
  - If the call is direct, place the address of the function into the count register, and use a **bcctrl** instruction (LK = ’1’) instead of a **bl** or **bcl** instruction.
  - For the **bcctr**[**l**] instruction that branches to the function, use BH = ’11’ if appropriate.

4.2.4.3 Branch Instructions

Table 4-23 describes the branch instructions provided by the PowerPC processors.

Table 4-23. Branch Instructions

Name	Mnemonic	Operand Syntax	Operation
Branch	**b** **ba** **bl** **bla**	target_addr	**b** Branch. Branch to the address computed as the sum of the immediate address and the address of the current instruction. **ba** Branch Absolute. Branch to the absolute address specified. **bl** Branch then Link. Branch to the address computed as the sum of the immediate address and the address of the current instruction. The instruction address following this instruction is placed into the link register (LR). **bla** Branch Absolute then Link. Branch to the absolute address specified. The instruction address following this instruction is placed into the LR.
Branch Conditional	**bc** **bca** **bcl** **bcla**	BO**,**BI**,**target_addr	The BI operand specifies the bit in the CR to be used as the condition of the branch. The BO operand is used as described in *Table 4-20*. **bc** Branch Conditional. Branch conditionally to the address computed as the sum of the immediate address and the address of the current instruction. **bca** Branch Conditional Absolute. Branch conditionally to the absolute address specified. **bcl** Branch Conditional then Link. Branch conditionally to the address computed as the sum of the immediate address and the address of the current instruction. The instruction address following this instruction is placed into the LR. **bcla** Branch Conditional Absolute then Link. Branch conditionally to the absolute address specified. The instruction address following this instruction is placed into the LR.
Branch Conditional to Link Register	**bclr** **bclrl**	BO,BI,BH	The BI operand specifies the bit in the CR to be used as the condition of the branch. The BO operand is used as described in *Table 4-20*. The BH field is used as described in *Table 4-22* and the branch target address is LR[0–61] || ’00’, with the high-order 32 bits of the branch target address cleared in the 32-bit mode of a 64-bit implementation. **bclr** Branch Conditional to Link Register. Branch conditionally to the address in the LR. **bclrl** Branch Conditional to Link Register then Link. Branch conditionally to the address specified in the LR. The instruction address following this instruction is then placed into the LR.
Branch Conditional to Count Register	**bcctr** **bcctrl**	BO,BI,BH	The BI operand specifies the bit in the CR to be used as the condition of the branch. The BO operand is used as described in *Table 4-20*. The BH field is used as described in *Table 4-22* and the branch target address is CTR[0–61] || ‘00’, with the high-order 32 bits of the branch target address cleared in the 32-bit mode of a 64-bit implementation. **bcctr** Branch Conditional to Count Register. Branch conditionally to the address specified in the count register. **bcctrl** Branch Conditional to Count Register then Link. Branch conditionally to the address specified in the count register. The instruction address following this instruction is placed into the LR. **Note:** If the “decrement and test CTR” option is specified (BO[2] = ’0’), the instruction form is invalid.

4.2.4.4 Simplified Mnemonics for Branch Processor Instructions

To simplify assembly language programming, a set of simplified mnemonics and symbols is provided for the most frequently used forms of branch conditional, compare, trap, rotate and shift, and certain other instructions. See *Appendix E Simplified Mnemonics* for a list of simplified mnemonic examples.

4.2.4.5 Condition Register Logical Instructions

Condition register logical instructions, shown in *Table 4-24*, and the Move Condition Register Field (**mcrf**) instruction are also defined as flow control instructions.

Note: If the LR update option is enabled for any of these instructions, the PowerPC Architecture defines these forms of the instructions as invalid.

Table 4-24. Condition Register Logical Instructions

Name	Mnemonic	Operand Syntax	Operation
Condition Register AND	**crand**	**crb**D,**crb**A,**crb**B	The CR bit specified by **crb**A is ANDed with the CR bit specified by **crb**B. The result is placed into the CR bit specified by **crb**D.
Condition Register OR	**cror**	**crb**D,**crb**A,**crb**B	The CR bit specified by **crb**A is ORed with the CR bit specified by **crb**B. The result is placed into the CR bit specified by **crb**D.
Condition Register XOR	**crxor**	**crb**D,**crb**A,**crb**B	The CR bit specified by **crb**A is XORed with the CR bit specified by **crb**B. The result is placed into the CR bit specified by **crb**D.
Condition Register NAND	**crnand**	**crb**D,**crb**A,**crb**B	The CR bit specified by **crb**A is ANDed with the CR bit specified by **crb**B. The complemented result is placed into the CR bit specified by **crb**D.
Condition Register NOR	**crnor**	**crb**D,**crb**A,**crb**B	The CR bit specified by **crb**A is ORed with the CR bit specified by **crb**B. The complemented result is placed into the CR bit specified by **crb**D.
Condition Register Equivalent	**creqv**	**crb**D,**crb**A, **crb**B	The CR bit specified by **crb**A is XORed with the CR bit specified by **crb**B. The complemented result is placed into the CR bit specified by **crb**D.
Condition Register AND with Complement	**crandc**	**crb**D,**crb**A, **crb**B	The CR bit specified by **crb**A is ANDed with the complement of the CR bit specified by **crb**B and the result is placed into the CR bit specified by **crb**D.
Condition Register OR with Complement	**crorc**	**crb**D,**crb**A, **crb**B	The CR bit specified by **crb**A is ORed with the complement of the CR bit specified by **crb**B and the result is placed into the CR bit specified by **crb**D.
Move Condition Register Field	**mcrf**	**crf**D,**crf**S	The contents of **crf**S are copied into **crf**D. No other condition register fields are changed.

4.2.4.6 Trap Instructions

The trap instructions shown in *Table 4-25* are provided to test for a specified set of conditions. If any of the conditions tested by a trap instruction are met, the system trap handler is invoked. If the tested conditions are not met, instruction execution continues normally. See *Appendix E Simplified Mnemonics* for a complete set of simplified mnemonics.

Table 4-25. Trap Instructions

Name	Mnemonic	Operand Syntax	Operand Syntax
Trap Doubleword Immediate	**tdi**	TO,**r**A,SIMM	The contents of **r**A are compared with the sign-extended SIMM operand. If any bit in the TO operand is set and its corresponding condition is met by the result of the comparison, the system trap handler is invoked.
Trap Word Immediate	**twi**	TO,**r**A,SIMM	The contents of the low-order 32 bits of **r**A are compared with the sign-extended SIMM operand. If any bit in the TO operand is set and its corresponding condition is met by the result of the comparison, the system trap handler is invoked.
Trap Doubleword	**td**	TO,**r**A,**r**B	The contents of **r**A are compared with the contents of **r**B. If any bit in the TO operand is set and its corresponding condition is met by the result of the comparison, the system trap handler is invoked.
Trap Word	**tw**	TO,**r**A,**r**B	The contents of the low-order 32 bits of **r**A are compared with the contents of the low-order 32 bits of **r**B. If any bit in the TO operand is set and its corresponding condition is met by the result of the comparison, the system trap handler is invoked.

4.2.4.7 System Linkage Instruction—UISA

Table 4-26 describes the System Call (**sc**) instruction that permits a program to call on the system to perform a service. See *Section 4.4.1 System Linkage Instructions—OEA* for a complete description of the **sc** instruction.

Table 4-26. System Linkage Instruction—UISA

Name	Mnemonic	Operand Syntax	Operation
System Call	**sc**	—	This instruction calls the operating system to perform a service. When control is returned to the program that executed the system call, the content of the registers will depend on the register conventions used by the program providing the system service. This instruction is context synchronizing as described in *Section 4.1.5.1 Context Synchronizing Instructions*. See *Section 4.4.1 System Linkage Instructions—OEA* for a complete description of the **sc** instruction.

4.2.5 Processor Control Instructions—UISA

Processor control instructions are used to read from and write to the condition register (CR), machine state register (MSR), and special-purpose registers (SPRs). See *Section 4.3.1 Processor Control Instructions—VEA* for the **mftb** instruction and *Section 4.4.2 Processor Control Instructions—OEA* for information about the instructions used for reading from and writing to the MSR and SPRs.

4.2.5.1 Move to/from Condition Register Instructions

Table 4-27 summarizes the instructions for reading from or writing to the condition register.

Table 4-27. Move to/from Condition Register Instructions

Name	Mnemonic	Operand Syntax	Operation
Move to Condition Register Fields	**mtcrf**	CRM,**r**S	The contents of the low-order 32 bits of **r**S are placed into the CR under control of the field mask specified by operand CRM. The field mask identifies the 4-bit fields affected. Let *i* be an integer in the range 0–7. If CRM[*i*] = 1, CR field *i* (CR bits 4 × *i* through 4 × *i* + 3) is set to the contents of the corresponding field of the low-order 32 bits of **r**S.
Move to Condition One Register Fields	**mtocrf**	CRM,**r**S	This form of the **mtocrf** instruction is intended to replace the old form (**mtcrf**) of the instruction which will eventually be phased out of the architecture. The new form is backward compatible with most processors that comply with versions of the architecture that precede Version 2.01.
Move from Condition Register	**mfcr**	**r**D	The contents of the CR are placed into the low-order 32 bits of **r**D. The contents of the high-order 32 bits of **r**D are cleared.
Move from One Condition Register Field	**mfocrf**	**r**D,CRM	This form of the **mfocrf** instruction is intended to replace the old form (**mfcr**) of the instruction which is being phased out of the architecture. The new form is backward compatible with most processors that comply with versions of the architecture that precede Version 2.01. Refer to page 434 for details.

4.2.5.2 Move to/from Special-Purpose Register Instructions (UISA)

Figure 4-28 provides a brief description of the **mtspr** and **mfspr** instructions. For more detailed information refer to *Section 8 Instruction Set*.

Table 4-28. Move to/from Special-Purpose Register Instructions (UISA)

Name	Mnemonic	Operand Syntax	Operation
Move to Special-Purpose Register	**mtspr**	SPR,**r**S	The value specified by **r**S are placed in the specified SPR. For 32-bit SPRs, the low-order 32 bits of **r**S are placed into the SPR.
Move from Special-Purpose Register	**mfspr**	**r**D,SPR	The contents of the specified SPR are placed in **r**D. For 32-bit SPRs, the low-order 32 bits of **r**D receive the contents of the SPR. The high-order 32 bits of **r**D are cleared.

4.2.6 Memory Synchronization Instructions—UISA

Memory synchronization instructions control the order in which memory operations are completed with respect to asynchronous events, and the order in which memory operations are seen by other processors or memory access mechanisms.

The number of cycles required to complete a **sync** instruction depends on system parameters and on the processor's state when the instruction is issued. As a result, frequent use of this instruction may degrade performance slightly. The **eieio** instruction may be more appropriate than **sync** for many cases.

The PowerPC Architecture defines the **sync** instruction with CR update enabled (Rc field, bit [31] = ‘1’) to be an invalid form.

The concept behind the use of the **lwarx**, **ldarx**, **stwcx.**, and **stdcx.** instructions is that a processor may load a semaphore from memory, compute a result based on the value of the semaphore, and conditionally store it back to the same location. Examples of these semaphore operations can be found in *Appendix D Synchronization Programming Examples*. The **lwarx** instruction must be paired with an **stwcx.** instruction, and **ldarx** instruction with an **stdcx.** instruction, with the same effective address specified by both instructions of the pair. The only exception is that an unpaired **stwcx.** or **stdcx.** instruction to any (scratch) effective address can be used to clear any reservation held by the processor. The conditional store is performed based upon the existence of a reservation established by the preceding **lwarx** or **ldarx** instruction. If the reservation exists when the store is executed, the store is performed and a bit is set in the CR. If the reservation does not exist when the store is executed, the target memory location is not modified and a bit is cleared in the CR.

Note: The reservation granularity is implementation-dependent.

The **lwarx**, **ldarx**, **stwcx.**, and **stdcx.** primitives allow software to read a semaphore, compute a result based on the value of the semaphore, store the new value back into the semaphore location only if that location has not been modified since it was first read, and determine if the store was successful. If the store was successful, the sequence of instructions from the read of the semaphore to the store that updated the semaphore appear to have been executed atomically (that is, no other processor or mechanism modified the semaphore location between the read and the update), thus providing the equivalent of a real atomic operation. However, in reality, other processors may have read from the location during this operation.

The **lwarx**, **ldarx**,**stwcx.**, and **stdcx.** instructions require the effective address to be aligned.

In general, the **lwarx**, **ldarx**, **stwcx.**, and **stdcx.** instructions should be used only in system programs, which can be invoked by application programs as needed.

At most one reservation exists simultaneously on any processor. The address associated with the reservation can be changed by a subsequent **lwarx** or **ldarx** instruction. The conditional store is performed based upon the existence of a reservation established by the preceding **lwarx** or **ldarx.** instruction.

A reservation held by the processor is cleared (or may be cleared, in the case of the fourth and fifth bullet items) by one of the following:

- The processor holding the reservation executes another **lwarx** or **ldarx** instruction; this clears the first reservation and establishes a new one.
- The processor holding the reservation executes any **stwcx.** or **stdcx.** instruction regardless of whether its address matches that of the **lwarx**.
- Some other processor executes a store or **dcbz** to the same reservation granule, or modifies a referenced or changed bit in the same reservation granule.

- Some other processor executes a **dcbtst**, **dcbst**, or **dcbf** to the same reservation granule; whether the reservation is cleared is undefined.
- Some other mechanism modifies a memory location in the same reservation granule.

Note: Exceptions do not clear reservations; however, system software invoked by exceptions may clear reservations.

Table 4-29 summarizes the memory synchronization instructions as defined in the UISA. See *Section 4.3.2 Memory Synchronization Instructions—VEA* for details about additional memory synchronization (**eieio** and **isync**) instructions.

Table 4-29. Memory Synchronization Instructions—UISA

Name	Mnemonic	Operand Syntax	Operation
Load Doubleword and Reserve Indexed	**ldarx**	**r**D,**r**A,**r**B	The EA is the sum (**r**A|0) + (**r**B). The doubleword in memory addressed by the EA is loaded into **r**D and the reservation is established.
Load Word and Reserve Indexed	**lwarx**	**r**D,**r**A,**r**B	The EA is the sum (**r**A|0) + (**r**B). The word in memory addressed by the EA is loaded into the low-order 32 bits of **r**D. The contents of the high-order 32 bits of **r**D are cleared.
Store Doubleword Conditional Indexed	**stdcx.**	**r**S,**r**A,**r**B	The EA is the sum (**r**A|0) + (**r**B). If a reservation exists and the effective address specified by the **stdcx.** instruction is the same as that specified by the load and reserve instruction that established the reservation, the contents of **r**S are stored into the doubleword in memory addressed by the EA, and the reservation is cleared. If a reservation exists but the effective address specified by the **stdcx.** instruction 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 contents of **r**S are stored into the doubleword in memory addressed by the EA. If a reservation does not exist, the instruction completes without altering memory or the contents of the cache.
Store Word Conditional Indexed	**stwcx.**	**r**S,**r**A,**r**B	The EA is the sum (**r**A|0) + (**r**B). If a reservation exists and the effective address specified by the **stwcx.** instruction is the same as that specified by the load and reserve instruction that established the reservation, the low-order 32 bits of **r**S are stored into the word in memory addressed by the EA, and the reservation is cleared. If a reservation exists but the effective address specified by the **stwcx.** instruction 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 low-order 32 bits of **r**S are stored into the word in memory addressed by the EA. If a reservation does not exist, the instruction completes without altering memory or the contents of the cache.
Synchronize	**sync**	L	Executing a **sync** instruction ensures that all instructions preceding the **sync** instruction appear to have completed before the **sync** instruction completes, and that no subsequent instructions are initiated by the processor until after the **sync** instruction completes. When the **sync** instruction completes, all memory accesses caused by instructions preceding the **sync** instruction will have been performed with respect to all other mechanisms that access memory, on the L=0,1, and 2 variants of this instruction. See *Chapter 8, Instruction Set* for more information.

Note: The architecture is likely to be changed in the future to permit the reservation to be lost if a **dcbf** instruction is executed on the processor holding the reservation. Therefore **dcbf** instructions should not be placed between a load and reserve instruction and the subsequent store conditional instruction.

4.2.7 Recommended Simplified Mnemonics

To simplify assembly language programs, a set of simplified mnemonics is provided for some of the most frequently used operations (such as no-op, load immediate, load address, move register, and complement register). Assemblers should provide the simplified mnemonics listed in *Appendix E.9 Recommended Simplified Mnemonics*. Programs written to be portable across the various assemblers for the PowerPC Architecture should not assume the existence of mnemonics not described in this manual.

For a complete list of simplified mnemonics, see *Appendix E Simplified Mnemonics*.

4.3 PowerPC VEA Instructions

The PowerPC virtual environment architecture (VEA) describes the semantics of the memory model that can be assumed by software processes, and includes descriptions of the cache model, cache-control instructions, address aliasing, and other related issues. Implementations that conform to the VEA also adhere to the UISA, but may not necessarily adhere to the OEA.

This section describes additional instructions that are provided by the VEA.

4.3.1 Processor Control Instructions—VEA

The VEA defines the **mftb** instruction (user-level instruction) for reading the contents of the time base register; see *Chapter 5, Cache Model and Memory Coherency* for more information. *Table 4-30* describes the **mftb** instruction.

Simplified mnemonics are provided (See *Appendix E.8 Simplified Mnemonics for Special-Purpose Registers*) for the **mftb** instruction so it can be coded with the TBR name as part of the mnemonic rather than requiring it to be coded as an operand. The simplified mnemonics Move from Time Base (**mftb**) and Move from Time Base Upper (**mftbu**) are variants of the **mftb** instruction rather than of the **mfspr** instruction. The **mftb** instruction serves as both a basic and simplified mnemonic. Assemblers recognize an **mftb** mnemonic with two operands as the basic form, and an **mftb** mnemonic with one operand as the simplified form.

The **mftb** simplified mnemonic moves from the lower half of the time base register (TBL) to a GPR, and the **mftbu** simplified mnemonic moves from the upper half of the time base (TBU) to a GPR.

Table 4-30. Move from Time Base Instruction

Name	Mnemonic	Operand Syntax	Operation
Move from Time Base	**mftb**	**r**D, TBR	The TBR field denotes either time base lower or time base upper, encoded as shown in *Table 4-31* and *Table 4-32*. The contents of the designated register are copied to **r**D. When reading TBU the high-order 32 bits of **r**D are cleared. When reading TBL the 64 bits of the time base are copied to **r**D.

Table 4-31 summarizes the time base (TBL/TBU) register encodings to which user-level access (using **mftb**) is permitted (as specified by the VEA).

Table 4-31. User-Level TBR Encodings (VEA)

Decimal Value in TBR Field	TBR[0–4] TBR[5–9]	Register Name	Description
268	01100 01000	TBL	Time base lower (read-only)
269	01101 01000	TBU	Time base upper (read-only)

Table 4-32 summarizes the TBL and TBU register encodings to which supervisor-level access (using **mtspr**) is permitted.

Table 4-32. Supervisor-Level TBR Encodings (VEA)

Decimal Value in SPR Field	SPR[0–4] SPR[5–9]	Register Name	Description
284	11100 01000	TBL[1]	Time base lower (write only)
285	11101 01000	TBU[1]	Time base upper (write only)

Note:

1. Moving from the time base (TBL and TBU) can also be accomplished with the **mftb** instruction.

4.3.2 Memory Synchronization Instructions—VEA

Memory synchronization instructions control the order in which memory operations are completed with respect to asynchronous events, and the order in which memory operations are seen by other processors or memory access mechanisms. See *Chapter 5, Cache Model and Memory Coherency* for additional information about these instructions and about related aspects of memory synchronization.

System designs that use a second-level cache should take special care to recognize the hardware signaling caused by a **sync** operation and perform the appropriate actions to guarantee that memory references that may be queued internally to the second-level cache have been performed globally.

In addition to the **sync** instruction (specified by UISA), the VEA defines the Enforce In-Order Execution of I/O (**eieio**) and Instruction Synchronize (**isync**) instructions; see *Table 4-33*. The number of cycles required to complete an **eieio** instruction depends on system parameters and on the processor's state when the instruction is issued. As a result, frequent use of this instruction may degrade performance slightly.

The **isync** instruction causes the processor to wait for any preceding instructions to complete, discard all prefetched instructions, and then branch to the next sequential instruction after **isync** (which has the effect of clearing the pipeline of prefetched instructions).

Table 4-33. Memory Synchronization Instructions—VEA

Name	Mnemonic	Operand Syntax	Operation
Enforce In-Order Execution of I/O	**eieio**	—	The **eieio** instruction provides an ordering function for the effects of loads and stores executed by a processor.
Instruction Synchronize	**isync**	—	Executing an **isync** instruction ensures that all previous instructions complete before the **isync** instruction completes, although memory accesses caused by those instructions need not have been performed with respect to other processors and mechanisms. It also ensures that the processor initiates no subsequent instructions until the **isync** instruction completes. Finally, it causes the processor to discard any prefetched instructions, so subsequent instructions will be fetched and executed in the context established by the instructions preceding the **isync** instruction. This instruction does not affect other processors or their caches.

4.3.3 Memory Control Instructions—VEA

Memory control instructions include the following types:

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

This section describes the user-level cache management instructions defined by the VEA. See *Section 4.4.3 Memory Control Instructions—OEA* for more information about supervisor-level cache, segment register manipulation, and translation lookaside buffer management instructions.

4.3.3.1 User-Level Cache Instructions—VEA

The instructions summarized in this section provide user-level programs the ability to manage on-chip caches if they are implemented. See *Chapter 5, Cache Model and Memory Coherency* for more information about cache topics.

As with other memory-related instructions, the effect of the cache management instructions on memory are weakly ordered. If the programmer needs to ensure that cache or other instructions have been performed with respect to all other processors and system mechanisms, a **sync** instruction must be placed in the program following those instructions.

Note: When data address translation is disabled (MSR[DR] = ’0’), the Data Cache Block Clear to Zero (**dcbz**) instruction allocates a cache block in the cache and might not verify that the physical address (referred to as real address in the architecture specification) is valid. If a cache block is created for an invalid physical address, a machine check condition may result when an attempt is made to write that cache block back to memory. The cache block could be written back as a result of the execution of an instruction that causes a cache miss and the invalid addressed cache block is the target for replacement or a Data Cache Block Store (**dcbst**) instruction.

Table 4-34 summarizes the cache instructions defined by the VEA.

Note: These instructions are accessible to user-level programs.

Table 4-34. User-Level Cache Instructions

Name	Mnemonic	Operand Syntax	Operation
Data Cache Block Touch	**dcbt**	**r**A,**r**B	The EA is the sum (**r**A|0) + (**r**B). 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. The hint is ignored if the block is caching inhibited or guarded.
Data Cache Block Touch for Store	**dcbtst**	**r**A,**r**B	The EA is the sum (**r**A|0) + (**r**B). 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. The hint is ignored if the block is caching inhibited or guarded.
Data Cache Block Clear to Zero	**dcbz**	**r**A,**r**B	The EA is the sum (**r**A|0) + (**r**B). If the cache block containing the byte addressed by the EA is in the data cache, all bytes of the cache block are cleared to zero. If the page containing the byte addressed by the EA is not in the data cache and the corresponding page is marked caching allowed (I = ’0’), the cache block is established in the data cache without fetching the block from main memory, and all bytes of the cache block are cleared to zero. If the page containing the byte addressed by the EA is marked caching inhibited (WIM = ‘x1x’) or write-through (WIM = ‘1xx’), either all bytes of the area of main memory that corresponds to the addressed cache block are cleared to zero, or an alignment exception occurs. If the cache block addressed by the EA is located in a page marked as memory coherent (WIM = ‘xx1’) and the cache block exists in the caches of other processors, memory coherence is maintained in those caches. The **dcbz** instruction is treated as a store to the addressed byte with respect to address translation, memory protection, referenced and changed recording, and the ordering enforced by **eieio** or by the combination of caching-inhibited and guarded attributes for a page.

Table 4-34. User-Level Cache Instructions (Continued)

Name	Mnemonic	Operand Syntax	Operation
Data Cache Block Store	**dcbst**	**r**A,**r**B	The EA is the sum(**r**A|0) + (**r**B). If the cache block containing the byte addressed by the EA is located in a page marked memory coherent (WIM = ‘xx1’), and a cache block containing the byte addressed by EA is in the data cache of any processor and has been modified, the cache block is written to main memory. If the cache block containing the byte addressed by the EA is located in a page not marked memory coherent (WIM = ‘xx0’), and a cache block containing the byte addressed by EA is in the data cache of this processor and has been modified, the cache block is written to main memory. The function of this instruction is independent of the write-through/write-back and caching-inhibited/caching-allowed modes of the cache block containing the byte addressed by the EA. The **dcbst** instruction is treated as a load from the addressed byte with respect to address translation and memory protection. It may also be treated as a load for referenced and changed bit recording except that referenced and changed bit recording may not occur.
Data Cache Block Flush	**dcbf**	**r**A,**r**B	The EA is the sum (**r**A|0) + (**r**B). The action taken depends on the memory mode associated with the target, and on the state of the block. The following list describes the action taken for the various cases, regardless of whether the page or block containing the addressed byte is designated as write-through or if it is in the caching-inhibited or caching-allowed mode. • Coherency required (WIM = ‘xx1’) — Unmodified block—Invalidates copies of the block in the caches of all processors. — Modified block—Copies the block to memory. Invalidates copies of the block in the caches of all processors. — Absent block—If modified copies of the block are in the caches of other processors, causes them to be copied to memory and invalidated. If unmodified copies are in the caches of other processors, causes those copies to be invalidated. • Coherency not required (WIM = ‘xx0’) — Unmodified block—Invalidates the block in the processor’s cache. — Modified block—Copies the block to memory. Invalidates the block in the processor’s cache. — Absent block—Does nothing. The function of this instruction is independent of the write-through/write-back and caching-inhibited/caching-allowed modes of the cache block containing the byte addressed by the EA. The **dcbf** instruction is treated as a load from the addressed byte with respect to address translation and memory protection. It may also be treated as a load for referenced and changed bit recording except that referenced and changed bit recording may not occur.

Table 4-34. User-Level Cache Instructions (Continued)

Name	Mnemonic	Operand Syntax	Operation
Instruction Cache Block Invalidate	**icbi**	**r**A,**r**B	The EA is the sum (**r**A|0) + (**r**B). If the cache block containing the byte addressed by EA is located in a page marked memory coherent (WIM = ‘xx1’), and a cache block containing the byte addressed by EA is in the instruction cache of any processor, the cache block is made invalid in all such instruction caches, so that the next reference causes the cache block to be refetched. If the cache block containing the byte addressed by EA is located in a page not marked memory coherent (WIM = ‘xx0’), and a cache block containing the byte addressed by EA is in the instruction cache of this processor, the cache block is made invalid in that instruction cache, so that the next reference causes the cache block to be refetched. The function of this instruction is independent of the write-through/write-back and caching-inhibited/caching-allowed modes of the cache block containing the byte addressed by the EA. The **icbi** instruction is treated as a load from the addressed byte with respect to address translation and memory protection. It may also be treated as a load for referenced and changed bit recording except that referenced and changed bit recording may not occur. **Note:** The invalidation of the specified instruction cache block cannot be assumed to have been performed with respect to the processor executing the instruction until a subsequent **isync** instruction has been executed by the processor. No other instruction or event has the corresponding effect.

Note: In response to the hint provided by **dcbt** and **dcbtst**, the processor may prefetch the specified block into the data cache, or take other actions that reduce the latency of subsequent load or store instructions that refer to the block.

4.3.4 External Control Instructions

The external control instructions allow a user-level program to communicate with a special-purpose device. Two instructions are provided and are summarized in *Table 4-35*.

Table 4-35. External Control Instructions

Name	Mnemonic	Operand Syntax	Operation
External Control In Word Indexed	**eciwx**	rD,rA,rB	The EA is the sum (**r**A|0) + (**r**B). A load word request for the physical address corresponding to the EA is sent to the device identified by the EAR[RID] (bits [26–31]), bypassing the cache. The word returned by the device is placed into the low-order 32 bits of **r**D. The value in the high-order 32 bits of **r**D is cleared to zero. The EA sent to the device must be word-aligned. This instruction is treated as a load from the addressed byte with respect to address translation, memory protection, referenced and changed recording, and the ordering performed by **eieio**. This instruction is optional.
External Control Out Word Indexed	**ecowx**	**r**S,**r**A,**r**B	The EA is the sum (**r**A|0) + (**r**B). A store word request for the physical address corresponding to the EA and the contents of the low-order 32 bits of **r**S are sent to the device identified by EAR[RID] (bits [26–31]), bypassing the cache. The EA sent to the device must be word-aligned. This instruction is treated as a store to the addressed byte with respect to address translation, memory protection, referenced and changed recording, and the ordering performed by **eieio**. Software synchronization is required in order to ensure that the data access is performed in program order with respect to data accesses caused by other store or **ecowx** instructions, even though the addressed byte is assumed to be caching-inhibited and guarded. This instruction is optional.

4.4 PowerPC OEA Instructions

The PowerPC operating environment architecture (OEA) includes the structure of the memory management model, supervisor-level registers, and the exception model. Implementations that conform to the OEA also adhere to the UISA and the VEA. This section describes the instructions provided by the OEA.

4.4.1 System Linkage Instructions—OEA

This section describes the system linkage instructions (see *Table 4-36*). The **sc** instruction is a user-level instruction that permits a user program to call on the system to perform a service and causes the processor to take an exception. The **rfid** instruction is supervisor-level instructions that are useful for returning from an exception handler.

Table 4-36. System Linkage Instructions—OEA

Name	Mnemonic	Operand Syntax	Operation
System Call	**sc**	—	When executed, the effective address of the instruction following the **sc** instruction is placed into SRR0. Bits [33–36 and 42–47] of SRR1 are cleared. Additionally, bits [48–55, 57–59,and 62–63] of the MSR are placed into the corresponding bits of SRR1. Depending on the implementation, additional bits of MSR may also be saved in SRR1. Then a system call exception is generated. The exception causes the MSR to be altered as described in *Section 6.4 Exception Definitions*. The exception causes the next instruction to be fetched from offset 0x0000_0000_0000_0C00 from the physical base address determined by the value of HIOR. This instruction is context synchronizing.
Return from Interrupt Doubleword	**rfid**	—	Bits [0-2, 4-32, 37-41, 48-50, 52-57, 60-63] of SRR1 are placed into the corresponding bits of the MSR. Depending on the implementation, additional bits of MSR may also be restored from SRR1. If the new MSR value does not enable any pending exceptions, the next instruction is fetched, under control of the new MSR value, from the address SRR0[0–61] || ’00’ (default 64-bit mode) or (32)0 || the low-order 32 bits of SRR0 || ’00’ (32-bit mode of 64-bit implementations). If the new MSR value enables one or more pending exceptions, the exception associated with the highest priority pending exception is generated; in this case, the value placed into SRR0 (machine status save/restore 0) by the exception processing mechanism is the address of the instruction that would have been executed next had the exception not occurred. This is a supervisor-level instruction and is context-synchronizing.

4.4.2 Processor Control Instructions—OEA

This section describes the processor control instructions that are used to read from and write to the MSR and the SPRs.

4.4.2.1 Move to/from Machine State Register Instructions

Table 4-37 summarizes the instructions used for reading from and writing to the MSR.

Table 4-37. Move to/from Machine State Register Instructions

Name	Mnemonic	Operand Syntax	Operation
Move to Machine State Register	**mtmsr**	**r**S,L	The MSR is set based on the contents of register **r**S and the L field. L='0' The result of ORing bits [58] and [49] of register **r**S is placed into MSR[58]. The result of ORing bits [59] and [49] of register **r**S is placed into MSR[59]. Bits [32-47, 49-50, 52-57, 60-63] of register **r**S are placed into the corresponding bits of the MSR. The high order 32 bits of the MSR are unchanged. L="1 Bits [48, 62] of **r**S are placed into the corresponding bits of the MSR. The remaining bits of the MSR are unchanged. This instruction is a supervisor-level instruction. If L='0' this instruction is context synchronizing except with respect to alterations to the [LE] bit. If L='1' this instruction is execution synchronizing; in addition, the alterations of the [EE] and [RI] bits take effect as soon as the instruction completes.
Move to Machine State Register Doubleword	**mtmsrd**	**r**S,L	The MSR is set based on the contents of register **r**S and the L field. L='0' The result of ORing bits [0] and [1] of register **r**S is placed into MSR[0]. The result of ORing bits [59] and [49] of register **r**S is placed into MSR[59]. Bits [1-2, 4-47, 49, 50, 52-57, 60-63] of register **r**S are placed into the corresponding bits of the MSR. The high order 32 bits of the MSR are unchanged. L='1' Bits [48, 62] of **r**S are placed into the corresponding bits of the MSR. The remaining bits of the MSR are unchanged. This instruction is a supervisor-level instruction. If L = '0' this instruction is context synchronizing except with respect to alterations to the [LE] bit. If L = '1' this instruction is execution synchronizing; in addition, the alterations of the [EE] and [RI] bits take effect as soon as the instruction completes.
Move from Machine State Register	**mfmsr**	**r**D	The contents of the MSR are placed into **r**D. This is a supervisor-level instruction.

4.4.2.2 Move to/from Special-Purpose Register Instructions (OEA)

Provided is a brief description of the **mtspr** and **mfspr** instructions (see *Table 4-38*). For more detailed information, see *Chapter 8, Instruction Set*. Simplified mnemonics are provided for the **mtspr** and **mfspr** instructions in *Appendix E Simplified Mnemonics*. For a discussion of context synchronization requirements when altering certain SPRs, refer to *Appendix D Synchronization Programming Examples*.

Table 4-38. Move to/from Special-Purpose Register Instructions (OEA)

Name	Mnemonic	Operand Syntax	Operation
Move to Special-Purpose Register	**mtspr**	SPR,**r**S	The SPR field denotes a special-purpose register. The contents of **r**S are placed into the designated SPR. For SPRs that are 32 bits long, the contents of the low-order 32 bits of **r**S are placed into the SPR. For this instruction, SPRs TBL and TBU are treated as separate 32-bit registers; setting one leaves the other unaltered.
Move from Special- Purpose Register	**mfspr**	**r**D,SPR	The SPR field denotes a special-purpose register. The contents of the designated SPR are placed into **r**D.

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

For information on SPR encodings (both user and supervisor-level), see *Chapter 8, Instruction Set*.

Note: There are additional SPRs specific to each implementation; for implementation-specific SPRs, see the user's manual for your particular processor.

4.4.3 Memory Control Instructions—OEA

Memory control instructions include the following types of instructions:

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

This section describes supervisor-level memory control instructions. See *Section 4.3.3 Memory Control Instructions—VEA* for more information about user-level cache management instructions.

4.4.3.1 Segment Register Manipulation Instructions

The instructions listed in *Table 4-39* allow software to associate effective segments 0 through 15 with any of virtual segments 0 through 227- 1. SLB entries [0-15] serve as virtual Segment Registers, with SLB entry i used to emulate Segment Register i. The mtsr and mtsrin instructions move 32 bits from a selected GPR to a selected SLB entry. The mfsr and mfsrin instructions move 32 bits from a selected SLB entry to a selected GPR. These instructions operate completely independent of the MSR[IR] and MSR[DR] bit settings. Refer to *Section 2.3.16 Synchronization Requirements for Special Registers and for Lookaside Buffers* for serialization requirements and other recommended precautions to observe when manipulating the segment registers.

Table 4-39. Segment Register Manipulation Instructions

Name	Mnemonic	Operand Syntax	Operation
64-Bit Bridge Move to Segment Register	**mtsr**	SR,**r**S	The SLB entry specified by SR is loaded from register **r**S as described in *Section 8.2 PowerPC Instruction Set*. MSR[SF] must be ‘0’ when this instruction is executed, otherwise the results are boundedly undefined. This instruction is a supervisor-level instruction.
64-Bit Bridge Move to Segment Register Indirect	**mtsrin**	**r**S,**r**B	The SLB entry specified by **r**B[32-35] is loaded from **r**S as described in *Section 8.2 PowerPC Instruction Set*. MSR[SF] must be ‘0’ when this instruction is executed, otherwise the results are boundedly undefined. This is a supervisor-level instruction.
64-Bit Bridge Move from Segment Register	**mfsr**	**r**D,SR	This instruction must be used only to read an SLB entry that was, or could have been, created by **mtsr** or **mtsrin** and has not subsequently been invalidated. Otherwise the contents of register **r**D is undefined. Refer to *Section 8.2 PowerPC Instruction Set* for details. This instruction is a supervisor-level instruction.
64-Bit Bridge Move from Segment Register Indirect	**mfsrin**	**r**D,**r**B	This instruction must be used only to read an SLB entry that was, or could have been, created by **mtsr** or **mtsrin** and has not subsequently been invalidated. Otherwise the contents of register **r**D is undefined. Refer to *Section 8.2 PowerPC Instruction Set* for details. This instruction is a supervisor-level instruction.

4.4.3.2 Translation and Segment Lookaside Buffer Management Instructions

The address translation mechanism is defined in terms of segment descriptors and page table entries (PTEs) used by PowerPC processors to locate the logical-to-physical address mapping for a particular access. These segment descriptors and PTEs reside in segment tables and page tables in memory, respectively.

All implementations have a segment lookaside buffer (SLB) to cache a portion of the segment table. For performance reasons, most implementations have one or more translation lookaside buffers (TLB) to cache a portion of the page table. As changes are made to the segment and page tables, it is necessary to maintain coherence between these lookaside buffers and the translation tables.

This is done by invalidating SLB or TLB entries, or occasionally by invalidating the entire SLB or TLB, and allowing the translation caching mechanism to refetch from the segment and page tables. For this purpose, each implementation provides the SLB management instructions described in *Table 4-40*. Each implementation that has a TLB provides a means for invalidating a single TLB entry, and a means for invalidating the entire TLB. If a processor does not implement a TLB, it treats the TLB managment instructions (also described in *Table 4-40*) either as no-ops or as illegal instructions.

Refer to *Chapter 7, Memory Management* for more information about TLB operation. *Table 4-40* summarizes the operation of the SLB and TLB instructions.

Table 4-40. Lookaside Buffer Management Instructions

Name	Mnemonic	Operand Syntax	Operation
SLB Invalidate All	**slbia**	—	For all SLB entries, except SLB entry 0, the V-bit in the entry is set to 0, making the entry invalid, and all other fields undefined. SLB entry 0 is undefined. This is a supervisor-level instruction. **Note:** **slbia** does not affect SLBs on other processors.
SLB Invalidate Entry	**slbie**	**r**B	The Effective Segment ID (ESID) is **r**B[0-35]. The class is **r**B[36]. The class value must be the same as the class value in the SLB entry that translates the ESID, or the class value that was in the SLB entry that most recently translated the ESID if the translation is no longer in the SLB. If the class value is not the same, the results of translating effective addresses for which EA[0-35] = ESID are undefined. The only SLB entry that is invalidated is the entry that translates the specified ESID. **slbie** does not affect SLBs on other processors. If this instruction is executed in 32-bit mode, **r**B[0:31] must be zeros. This is a supervisor-level instruction. **Note:** If the optional “bridge” facility is implemented, the move to segment register instructions create SLB entries in which the class value = ‘0’.
SLB Move to Entry	**slbmte**	**r**S,**r**B	The SLB entry specified by **r**B[52-63] is loaded from register **r**S and from the remainder of register **r**B. This instruction cannot be used to invalidate an SLB entry. This is a supervisor-level instruction. For more information refer to *Section 8.2 PowerPC Instruction Set*.
SLB Move from Entry	**slbmfev**	**r**S,**r**B	If the SLB entry specified by bits [52-63] of register **r**B is valid (V = ’1’), the contents of the VSID, Ks, Kp, N, L, and C fields of the entry are placed into register **r**S. This is a supervisor-level instruction. For more information refer to *Section 8.2 PowerPC Instruction Set*.
SLB Move from Entry ESID	**slbmfee**	**r**S,**r**B	If the SLB entry specified by bits [52-63] of register **r**B is valid (V = ’1’), the contents of the ESID and V fields of the entry are placed into register **r**S. This is a supervisor-level instruction. For more information refer to *Section 8.2 PowerPC Instruction Set*.
TLB Invalidate Entry	**tlbie**	**r**B,L	The contents of **r**B specify the VPN of target TLB entries. See *Section 8.2 PowerPC Instruction Set* for further details. If L = ’0’, target entries are for 4KB pages, otherwise large pages. This instruction causes the target TLB entry to be invalidated in all processors. The operation performed by this instruction is treated as a caching inhibited and guarded data access with respect to the ordering performed by **eieio** (or **sync** or **ptesync**). When this instruction is executed MSR[SF] must be one, otherwise the results are boundedly undefined. This is a supervisor-level instruction and optional in the PowerPC Architecture.

Table 4-40. Lookaside Buffer Management Instructions (Continued)

Name	Mnemonic	Operand Syntax	Operation
TLB Invalidate Entry Local	**tlbiel**	**r**B,L	The contents of **r**B specify the VPN of target TLB entries. See *Section 8.2 PowerPC Instruction Set* for further details. If L = '0', target entries are for 4KB pages, otherwise large pages. Support of large pages for **tlbiel** is optional. To synchronize the completion of this processor local form of **tlbie**, only a **ptesync** is required. **r**B[52-63] must be zero. This is a supervisor-level instruction and optional.
TLB Invalidate All	**tlbia**	—	All TLB entries are made invalid. The TLB is invalidated regardless of the settings of MSR[IR] and MSR[DR]. This instruction does not cause the entries to be invalidated in other processors. This is a supervisor-level instruction and optional.
TLB Synchronize	**tlbsync**	—	Executing a **tlbsync** instruction ensures that all **tlbie** instructions previously executed by the processor executing the **tlbsync** instruction have completed on all processors. The operation performed by this instruction is treated as a caching inhibited and guarded data access with respect to the ordering performed by **eieio** (or **sync** or **ptesync**). **tlbsync** should not be used to synchronize the completion of **tlbiel**. This is a supervisor-level instruction and optional.

Because the presence and exact semantics of the translation lookaside buffer management instructions is implementation-dependent, system software should incorporate uses of the instruction into subroutines to minimize compatibility problems.

5. Cache Model and Memory Coherency

This chapter summarizes the cache model as defined by the virtual environment architecture (VEA), as well as the built-in architectural controls for maintaining memory coherency. This chapter describes the cache control instructions and special concerns for memory coherency in single-processor and multiprocessor systems. Aspects of the operating environment architecture (OEA) as they relate to the cache model and memory coherency are also covered.

The PowerPC Architecture provides for relaxed memory coherency. Features such as write-back caching and out-of-order execution allow software engineers to exploit the performance benefits of weakly-ordered memory access. The architecture also provides the means to control the order of accesses for order-critical operations.

In this chapter, the term multiprocessor is used in the context of maintaining cache coherency. In this context, a system could include other devices that access system memory, maintain independent caches, and function as bus masters.

Each cache management instruction operates on an aligned unit of memory. The VEA defines this cacheable unit as a block. Since the term ‘block’ is easily confused with the unit of memory addressed by the block address translation (BAT) mechanism, this chapter uses the term ‘cache block’ to indicate the cacheable unit. The size of the cache block can vary by instruction and by implementation. In addition, the unit of memory at which coherency is maintained is called the coherence block. The size of the coherence block is also implementation-specific. However, the coherence block is often the same size as the cache block.

5.1 The Virtual Environment

The User Instruction Set Architecture (UISA) relies upon a memory space of 2^{64} bytes for applications. The VEA expands upon the memory model by introducing virtual memory, caches, and shared memory multiprocessing. Although many applications will not need to access the features introduced by the VEA, it is important that programmers are aware that they are working in a virtual environment where the physical memory may be shared by multiple processes running on one or more processors.

This section describes load and store ordering, atomicity, the cache model, memory coherency, and the VEA cache management instructions. The features of the VEA are accessible to both user-level and supervisor-level applications (referred to as problem state and privileged state, respectively, in the architecture specification).

The mechanism for controlling the virtual memory space is defined by the OEA. The features of the OEA are accessible to supervisor-level applications only (typically operating systems). For more information on the address translation mechanism, refer to *Chapter 7, Memory Management*.

5.1.1 Memory Access Ordering

The VEA specifies a weakly consistent memory model for shared memory multiprocessor systems. This model provides an opportunity for significantly improved performance over a model that has stronger consistency rules, but places the responsibility for access ordering on the programmer. When a program requires strict access ordering for proper execution, the programmer must insert the appropriate ordering or synchronization instructions into the program.

The order in which the processor performs memory accesses, the order in which those accesses complete in memory, and the order in which those accesses are viewed as occurring by another processor may all be different. A means of enforcing memory access ordering is provided to allow programs (or instances of programs) to share memory. Similar means are needed to allow programs executing on a processor to share memory with some other mechanism, such as an I/O device, that can also access memory.

Various facilities are provided that enable programs to control the order in which memory accesses are performed by separate instructions. First, if separate store instructions access memory that is designated as both caching-inhibited and guarded, the accesses are performed in the order specified by the program. Refer to *Section 5.1.4 Memory Coherency* and *Section 5.2.1 Memory/Cache Access Attributes* for a complete description of the caching-inhibited and guarded attributes. Additionally, two instructions, **eieio** and **sync**, are provided that enable the program to control the order in which the memory accesses caused by separate instructions are performed.

No ordering should be assumed among the memory accesses caused by a single instruction (that is, by an instruction for which multiple accesses are not atomic), and no means are provided for controlling that order. *Chapter 4, Addressing Modes and Instruction Set Summary* contains additional information about the **sync** and **eieio** instructions.

5.1.1.1 Enforce In-Order Execution of I/O Instruction

The **eieio** instruction permits the program to control the order in which loads and stores are performed when the accessed memory has certain attributes, as described in *Chapter 8, Instruction Set*. For example, **eieio** can be used to ensure that a sequence of load and store operations to an I/O device's control registers updates those registers in the desired order. The **eieio** instruction can also be used to ensure that all stores to a shared data structure are visible to other processors before the store that releases the lock is visible to them.

The **eieio** instruction may complete before memory accesses caused by instructions preceding the **eieio** instruction have been performed with respect to system memory or coherent storage as appropriate.

If stronger ordering is desired, the **sync** instruction must be used.

5.1.1.2 Synchronize Instruction

When a portion of memory that requires coherency must be forced to a known state, it is necessary to synchronize memory with respect to other processors and mechanisms. This synchronization is accomplished by requiring programs to indicate explicitly in the instruction stream, by inserting a **sync** instruction, that synchronization is required. Only when **sync** completes are the effects of all coherent memory accesses previously executed by the program guaranteed to have been performed with respect to all other processors and mechanisms that access those locations coherently.

The **sync** instruction ensures that all the coherent memory accesses, initiated by a program, have been performed with respect to all other processors and mechanisms that access the target locations coherently, before its next instruction is executed. A program can use this instruction to ensure that all updates to a shared data structure, accessed coherently, are visible to all other processors that access the data structure coherently, before executing a store that will release a lock on that data structure. Execution of the **sync** instruction does the following:

- Performs the functions described for the **sync** instruction in *Section 4.2.6 Memory Synchronization Instructions—UISA*.
- Ensures that consistency operations, and the effects of **icbi**, **dcbz**, **dcbst**, and **dcbf** instructions previously executed by the processor executing **sync**, have completed on such other processors as the memory/cache access attributes of the target locations require.
- Ensures that TLB invalidate operations previously executed by the processor executing the **sync** have completed on that processor. The **sync** instruction does not wait for such invalidates to complete on other processors.
- Ensures that memory accesses due to instructions previously executed by the processor executing the **sync** are recorded in the R and C bits in the page table and that the new values of those bits are visible to all processors and mechanisms; refer to *Section 7.4.3 Page History Recording*.

The **sync** instruction is execution synchronizing. It is not context synchronizing, and therefore need not discard prefetched instructions.

For memory that does not require coherency, the **sync** instruction operates as described above except that its only effect on memory operations is to ensure that all previous memory operations have completed, with respect to the processor executing the **sync** instruction, to the level of memory specified by the memory/cache access attributes (including the updating of R and C bits).

See *Chapter 8, Instruction Set* for a description of the **sync** instruction, including the L=’1’ (**lwsync**) and L=’2’ (**ptesync**) variants.

5.1.2 Atomicity

An access is 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 neither specified in the program nor enforced between processors.

Only the following single-register accesses are guaranteed to be 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. In particular, the accesses caused by the following instructions are not guaranteed to be atomic:

- Load and store instructions with misaligned operands
- **lmw**, **stmw**, **lswi**, **lswx**, **stswi**, or **stswx** instructions
- Any cache management instructions

The **ldarx/stdcx.** and **lwarx**/**stwcx.** instruction combinations can be used to perform atomic memory references. The **ldarx** instruction is a load from a doubleword–aligned location that has two side effects:

1. A reservation for a subsequent instruction is created.
2. The memory coherence mechanism is notified that a reservation exists for the memory location accessed by the **ldarx**.

The **stdcx.** instruction is a store to a doubleword–aligned location that is conditioned on the existence of the reservation created by **ldarx** and on whether the same memory location is specified by both instructions and whether the instructions are issued by the same processor.

The **lwarx** and **stwcx.** instructions are the word-aligned forms of the **ldarx** and **stwcx.** instructions. To emulate an atomic operation with these instructions, it is necessary that both **ldarx** and **stdcx.** (or **lwarx** and **stwcx.**) access the same memory location.

Note: When a reservation is made to a word in memory by the **lwarx** or **ldarx** instruction, an address is saved and a reservation is set. Both of these are necessary for the memory coherence mechanism, however, some processors do not implement the address compare for the **stwcx.** instruction. Only the reservation needs to be established in order for the **stwcx./stdcx.** to be successful. This requires that exception handlers clear reservations if control is passed to another program. Programmers should read the specifications for each individual processor.

In a multiprocessor system, every processor (other than the one executing **ldarx/stdcx.** or **lwarx**/**stwcx.**) that might update the location must configure the addressed page as memory coherency required. The **ldarx/stdcx.** and **lwarx**/**stwcx.** instructions function in caching-inhibited, as well as in caching-allowed, memory. If the addressed memory is in write-through mode, it is implementation-dependent whether these instructions function correctly or cause the DSI exception handler to be invoked.

The **ldarx/stdcx.** and **lwarx**/**stwcx.** instruction combinations are described in *Section 4.2.6 Memory Synchronization Instructions—UISA* and *Chapter 8, Instruction Set*.

5.1.3 Cache Model

The PowerPC Architecture does not specify the type, organization, implementation, or even the existence of a cache. The standard cache model has separate instruction and data caches, also known as a Harvard cache model. However, the architecture allows for many different cache types. Some implementations will have a unified cache (where there is a single cache for both instructions and data). Other implementations may not have a cache at all.

The function of the cache management instructions depends on the implementation of the cache(s) and the setting of the memory/cache access modes. For a program to execute properly on all implementations, software should use the Harvard model. In cases where a processor is implemented without a cache, the architecture guarantees that instructions affecting the nonimplemented cache will not halt execution.

Note: **dcbz** may cause an alignment exception on some implementations. For example, a processor with no cache may treat a cache instruction as a no-op. Or, a processor with a unified cache may treat the **icbi** instruction as a no-op. In this manner, programs written for separate instruction and data caches will run on all compliant implementations.

5.1.4 Memory Coherency

The primary objective of a coherent memory system is to provide the same image of memory to all devices using the system. The VEA and OEA define coherency controls that facilitate synchronization, cooperative use of shared resources, and task migration among processors. These controls include the memory/cache access attributes, the **sync** and **eieio** instructions, and the **ldarx**/**stdcx.** and **lwarx**/**stwcx.** instruction pairs. Without these controls, the processor could not support a weakly-ordered memory access model.

A strongly-ordered memory access model hinders performance by requiring excessive overhead, particularly in multiprocessor environments. For example, a processor performing a store operation in a strongly-ordered system requires exclusive access to an address before making an update, to prevent another device from using stale data.

The VEA defines a page as a unit of memory for which protection and control attributes are independently specifiable. The OEA (supervisor level) specifies the size of a page as 4 Kbytes or a large page whose size is implementation dependent.

Note: The VEA (user level) does not specify the page size.

5.1.4.1 Memory/Cache Access Modes

The OEA defines the set of memory/cache access modes and the mechanism to implement these modes. Refer to *Section 5.2.1 Memory/Cache Access Attributes* for more information. However, the VEA specifies that at the user level, the operating system can be expected to provide the following attributes for each page of memory:

- Write-through or write-back
- Caching-inhibited or caching-allowed
- Memory coherency required or memory coherency not required
- Guarded or not guarded

User-level programs specify the memory/cache access attributes through an operating system service.

Pages Designated as Write-Through

When a page is designated as write-through, store operations update the data in the cache and also update the data in main memory. The processor writes to the cache and through to main memory. Load operations use the data in the cache, if it is present.

In write-back mode, the processor is only required to update data in the cache. The processor may (but is not required to) update main memory. Load and store operations use the data in the cache, if it is present. The data in main memory does not necessarily stay consistent with that same location’s data in the cache. Many implementations automatically update main memory in response to a memory access by another device (for example, a snoop hit). In addition, the **dcbst** and **dcbf** instructions can be used to explicitly force an update of main memory.

The write-through attribute is meaningless for locations designated as caching-inhibited.

Pages Designated as Caching-Inhibited

When a page is designated as caching-inhibited, the processor bypasses the cache and performs load and store operations to main memory. When a page is designated as caching-allowed, the processor uses the cache and performs load and store operations to the cache or main memory depending on the other memory/cache access attributes for the page.

It is important that all locations in a page are purged from the cache prior to changing the memory/cache access attribute for the page from caching-allowed to caching-inhibited. It is considered a programming error if a caching-inhibited memory location is found in the cache. 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.

Pages Designated as Memory Coherency Required

When a page is designated as memory coherency required, store operations to that location are serialized with all stores to that same location by all other processors that also access the location coherently. This can be implemented, for example, by an ownership protocol that allows at most one processor at a time to store to the location. Moreover, the current copy of a cache block that is in this mode may be copied to main storage any number of times, for example, by successive **dcbst** instructions.

Coherency does not ensure that the result of a store by one processor is visible immediately to all other processors and mechanisms. Only after a program has executed the **sync** instruction are the previous storage accesses it executed guaranteed to have been performed with respect to all other processors and mechanisms.

Pages Designated as Memory Coherency Not Required

For a memory area that is configured such that coherency is not required, software must ensure that the data cache is consistent with main storage before changing the mode or allowing another device to access the area.

Executing a **dcbst** or **dcbf** instruction specifying a cache block that is in this mode causes the block to be copied to main memory if and only if the processor modified the contents of a location in the block and the modified contents have not been written to main memory.

In a single-cache system, correct coherent execution may likely not require memory coherency; therefore, using memory coherency not required mode improves performance.

Pages Designated as Guarded

The guarded attribute pertains to out-of-order execution. Refer to *Out-of-Order Accesses to Guarded Memory* on page 203 for more information about out-of-order execution.

When a page is designated as guarded, instructions and data cannot be accessed out of order. Additionally, if separate store instructions access memory that is both caching-inhibited and guarded, the accesses are performed in the order specified by the program. When a page is designated as not guarded, out-of-order fetches and accesses are allowed.

Guarded pages are traditionally used for memory-mapped I/O devices.

5.1.4.2 Coherency Precautions

Mismatched memory/cache attributes cause coherency paradoxes in both single-processor and multi-processor systems. When the memory/cache access attributes are changed, it is critical that the cache contents reflect the new attribute settings. For example, if a page that had allowed caching becomes caching-inhibited, the appropriate cache blocks should be flushed to leave no indication that caching had previously been allowed.

Although coherency paradoxes are considered programming errors, specific implementations may attempt to handle the offending conditions and minimize the negative effects on memory coherency. Bus operations that are generated for specific instructions and state conditions are not defined by the architecture.

5.1.5 VEA Cache Management Instructions

The VEA defines instructions for controlling both the instruction and data caches. For implementations that have a unified instruction/data cache, instruction cache control instructions are valid instructions, but may function differently.

This section briefly describes the cache management instructions available to programs at the user privilege level. Additional descriptions of coding the VEA cache management instructions is provided in *Chapter 4, Addressing Modes and Instruction Set Summary* and *Chapter 8, Instruction Set*. In the following instruction descriptions, the target is the cache block containing the byte addressed by the effective address.

5.1.5.1 Data Cache Instructions

Data caches and unified caches must be consistent with other caches (data or unified), memory, and I/O data transfers. To ensure consistency, aliased effective addresses (two effective addresses that map to the same physical address) must have the same page offset.

Note: Physical address is referred to as real address in the architecture specification.

Data Cache Block Touch (dcbt) and Data Cache Block Touch for Store (dcbtst) Instructions

These instructions provide a method for improving performance through the use of software-initiated prefetch hints. However, these instructions do not guarantee that a cache block will be fetched.

A program uses the **dcbt** instruction to request a cache block fetch before it is needed by the program. The program can then use the data from the cache rather than fetching from main memory.

The **dcbtst** instruction behaves similarly to the **dcbt** instruction. A program uses **dcbtst** to request a cache block fetch to guarantee that a subsequent store will be to a cached location.

The processor does not invoke the exception handler for translation or protection violations caused by either of the touch instructions. Additionally, memory accesses caused by these instructions are not necessarily recorded in the page tables. If an access is recorded, then it is treated in a manner similar to that of a load from the addressed byte. Some implementations may not take any action based on the execution of these instructions, or they may prefetch the cache block corresponding to the effective address into their cache. For information about the R and C bits, see *Section 7.4.3 Page History Recording*.

Both **dcbt** and **dcbtst** are provided for performance optimization. These instructions do not affect the correct execution of a program, regardless of whether they succeed (fetch the cache block) or fail (do not fetch the cache block). If the target block is not accessible to the program for loads, then no operation occurs.

*Data Cache Block Set to Zero (**dcbz**) Instruction*

The **dcbz** instruction clears a single cache block as follows:

- If the target is in the data cache, all bytes of the cache block are cleared.
- If the target is not in the data cache and the corresponding page is caching-allowed, the cache block is established in the data cache (without fetching the cache block from main memory), and all bytes of the cache block are cleared.
- If the target is designated as either caching-inhibited or write-through, then either all bytes in main memory that correspond to the addressed cache block are cleared, or the alignment exception handler is invoked. The exception handler should clear all the bytes in main memory that correspond to the addressed cache block.
- If the target is designated as coherency required, and the cache block exists in the data cache(s) of any other processor(s), it is kept coherent in those caches.

The **dcbz** instruction is treated as a store to the addressed byte with respect to address translation, protection, referenced and changed recording, and the ordering enforced by **eieio** or by the combination of caching-inhibited and guarded attributes for a page.

Refer to *Chapter 6, Exceptions* for more information about a possible delayed machine check exception that can occur by using **dcbz** when the operating system has set up an incorrect memory mapping.

*Data Cache Block Store (**dcbst**) Instruction*

The **dcbst** instruction permits the program to ensure that the latest version of the target cache block is in main memory. The **dcbst** instruction executes as follows:

- Coherency required—If the target exists in the data cache of any processor and has been modified, the data is written to main memory. Only one processor in a multiprocessor system should have possession of a modified cache block.
- Coherency not required—If the target exists in the data cache of the executing processor and has been modified, the data is written to main memory.

The PowerPC Architecture does not specify whether the modified status of the cache block is left unchanged or is cleared (cleared implies valid-shared or valid-exclusive). That decision is left to the implementation of individual processors. Either state is logically correct.

The function of this instruction is independent of the write-through/write-back and caching-inhibited/caching-allowed attributes of the target.

The memory access caused by a **dcbst** instruction is not necessarily recorded in the page tables. If the access is recorded, then it is treated as a load operation (not as a store operation).

*Data Cache Block Flush (**dcbf**) Instruction*

The action taken depends on the memory/cache access mode associated with the target, and on the state of the cache block. The following list describes the action taken for the various cases:

- Coherency required
  - Unmodified cache block—Invalidates copies of the cache block in the data caches of all processors.
  - Modified cache block—Copies the cache block to memory. Invalidates the copy of the cache block in the data cache of any processor where it is found. There should only be one modified cache block in a coherency required multiprocessor system.
  - Target block not in cache—If a modified copy of the cache block is in the data cache(s) of another processor, **dcbf** causes the modified cache block to be copied to memory and then invalidated. If unmodified copies are in the data caches of other processors, **dcbf** causes those copies to be invalidated.
- Coherency not required
  - Unmodified cache block—Invalidates the cache block in the executing processor's data cache.
  - Modified cache block—Copies the data cache block to memory and then invalidates the cache block in the executing processor.
  - Target block not in cache—No action is taken.

The function of this instruction is independent of the write-through/write-back and caching-inhibited/caching-allowed attributes of the target.

The memory access caused by a **dcbf** instruction is not necessarily recorded in the page tables. If the access is recorded, then it is treated as a load operation (not as a store operation).

5.1.5.2 Instruction Cache Instructions

Instruction caches, if they exist, are not required to be consistent with data caches, memory, or I/O data transfers. Software must use the appropriate cache management instructions to ensure that instruction caches are kept coherent when instructions are modified by the processor or by input data transfer. When a processor alters a memory location that may be contained in an instruction cache, software must ensure that updates to memory are visible to the instruction fetching mechanism. Although the instructions to enforce consistency vary among implementations, the following sequence for a uniprocessor system is typical:

1. **dcbst** (update memory)
2. **sync** (wait for update)
3. **icbi** (invalidate copy in instruction cache)
4. **isync** (perform context synchronization)

Note: Most operating systems will provide a system service for this function. These operations are necessary because the memory may be designated as write-back. Since instruction fetching may bypass the data cache, changes made to items in the data cache may not otherwise be reflected in memory until after the instruction fetch completes.

For implementations used in multiprocessor systems, variations on this sequence may be recommended. For example, in a multiprocessor system with a unified instruction/data cache (at any level), if instructions are fetched without coherency being enforced, the preceding instruction sequence is inadequate. Because the **icbi** instruction does not invalidate blocks in a unified cache, a **dcbf** instruction should be used instead of a **dcbst** instruction for this case.

*Instruction Cache Block Invalidate Instruction (**icbi**)*

The **icbi** instruction executes as follows:

- Coherency required
  If the target is in the instruction cache of any processor, the cache block is made invalid in all such processors, so that the next reference causes the cache block to be refetched.
- Coherency not required
  If the target is in the instruction cache of the executing processor, the cache block is made invalid in the executing processor so that the next reference causes the cache block to be refetched.

The **icbi** instruction is provided for use in processors with separate instruction and data caches. The effective address is computed, translated, and checked for protection violations as defined in *Chapter 7, Memory Management*. If the target block is not accessible to the program for loads, then a DSI exception occurs.

The function of this instruction is independent of the write-through/write-back and caching-inhibited/caching-allowed attributes of the target.

The memory access caused by an **icbi** instruction is not necessarily recorded in the page tables. If the access is recorded, then it is treated as a load operation. Implementations that have a unified cache treat the **icbi** instruction as a no-op except that they may invalidate the target cache block in the instruction caches of other processors (in coherency required mode).

Note: The invalidation of the specified instruction cache block cannot be assumed to have been performed with respect to the processor executing the instruction until a subsequent **isync** instruction has been executed by the processor. No other instruction or event has the corresponding effect.

*Instruction Synchronize Instruction (**isync**)*

The **isync** instruction provides an ordering function for the effects of all instructions executed by a processor. Executing an **isync** instruction ensures that all instructions preceding the **isync** instruction have completed before the **isync** instruction completes, except that memory accesses caused by those instructions need not have been performed with respect to other processors and mechanisms. It also ensures that no subsequent instructions are initiated by the processor until after the **isync** instruction completes. Finally, it causes the processor to discard any prefetched instructions, 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 has no effect on other processors or on their caches.

5.2 The Operating Environment

The OEA defines the mechanism for controlling the memory/cache access modes introduced in *Section 5.1.4.1 Memory/Cache Access Modes*. This section describes the cache-related aspects of the OEA including the memory/cache access attributes, out-of-order execution, and the **dcbi** instruction. The features of the OEA are accessible to supervisor-level applications only. The mechanism for controlling the virtual memory space is described in *Chapter 7, Memory Management*.

The memory model of PowerPC processors provides the following features:

- Flexibility to allow performance benefits of weakly-ordered memory access
- A mechanism to maintain memory coherency among processors and between a processor and I/O devices controlled at the block and page level
- Instructions that can be used to ensure a consistent memory state
- Guaranteed processor access order

The memory implementations in PowerPC systems can take advantage of the performance benefits of weak ordering of memory accesses between processors or between processors and other external devices without any additional complications. Memory coherency can be enforced externally by a snooping bus design, a centralized cache directory design, or other designs that can take advantage of the coherency features of PowerPC processors.

Memory accesses performed 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 memory hierarchy. Order is guaranteed at each level of the memory hierarchy for accesses to the same address from the same processor. The **dcbf**, **dcbst, eieio**, **icbi**, **isync**, **ldarx**, **lwarx**, **stdcx.**, **stwcx.**, **sync**, and **tlbsync** instructions allow the programmer to ensure a consistent and ordered memory state.

5.2.1 Memory/Cache Access Attributes

All instruction and data accesses are performed under the control of the four memory/cache access attributes:

- Write-through (W attribute)
- Caching-inhibited (I attribute)
- Memory coherency (M attribute)
- Guarded (G attribute)

These attributes are maintained in the PTEs by the operating system for each page. The operating system stores the WIMG bits for each page into the PTEs in system memory as it sets up the page tables. The W and I attributes control how the processor performing an access uses its own cache. The M attribute ensures that coherency is maintained for all copies of the addressed memory location. When an access requires coherency, the processor performing the access must inform the coherency mechanisms throughout the system that the access requires memory coherency. The G attribute prevents out-of-order loading and prefetching from the addressed memory location.

Note: The memory/cache access attributes are relevant only when an effective address is translated by the processor performing the access. Also, not all combinations of settings of these bits are supported. The attributes are not saved along with data in the cache (for cacheable accesses), nor are they associated with subsequent accesses made by other processors.

Note: For data accesses performed in real addressing mode (MSR[DR] = ‘0’), the WIMG bits are assumed to be ‘0011’ (the data is write-back, caching is enabled, memory coherency is enforced, and memory is guarded). For instruction accesses performed in real addressing mode (MSR[IR] = ‘0’), the WIMG bits are assumed to be ‘0001’ (the data is write-back, caching is enabled, memory coherency is not enforced, and memory is guarded).

5.2.1.1 Write-Through Attribute (W)

When an access is designated as write-through (W = ‘1’), if the data is in the cache, a store operation updates the cached copy of the data. In addition, the update is written to the memory location. The definition of the memory location to be written to (in addition to the cache) depends on the implementation of the memory system but can be illustrated by the following examples:

- RAM—The store is sent to the RAM controller to be written into the target RAM.
- I/O device—The store is sent to the memory-mapped I/O controller to be written to the target register or memory location.

In systems with multilevel caching, the store must be written to at least a depth in the memory hierarchy that is seen by all processors and devices.

Multiple store instructions may be combined for write-through accesses except when the store instructions are separated by a **sync** or **eieio** instruction. A store operation to a memory location designated as write-through may cause any part of the cache block to be written back to main memory.

Accesses that correspond to W = ‘0’ are considered write-back. For this case, although the store operation is performed to the cache, the data is copied to memory only when a copy-back operation is required. Use of the write-back mode (W = ‘0’) can improve overall performance for areas of the memory space that are seldom referenced by other processors or devices in the system.

Accesses to the same memory location using two effective addresses for which the W-bit setting differs meet the memory-coherency requirements if the accesses are performed by a single processor. If the accesses are performed by two or more processors, coherence is enforced by the hardware only if the write-through attribute is the same for all the accesses.

5.2.1.2 Caching-Inhibited Attribute (I)

If I=’1’, the memory access is completed by referencing the location in main memory, bypassing the cache. During the access, the addressed location is not loaded into the cache nor is the location allocated in the cache.

It is considered a programming error if a copy of the target location of an access to caching-inhibited memory is resident in the cache. Software must ensure that the location has not been previously loaded into the cache, or if it has, that it has been flushed from the cache.

Data accesses from more than one instruction may be combined for cache-inhibited operations, except when the accesses are separated by a **sync** instruction, or by an **eieio** instruction when the page is also designated as guarded.

Instruction fetches, **dcbz** instructions, and load and store operations to the same memory location using two effective addresses for which the I-bit setting differs must meet the requirement that a copy of the target location of an access to caching-inhibited memory not be in the cache. Violation of this requirement is considered a programming error; software must ensure that the location has not previously been brought into the cache or, if it has, that it has been flushed from the cache. If the programming error occurs, the result of the access is boundedly undefined. It is not considered a programming error if the target location of any other cache management instruction to caching-inhibited memory is in the cache.

5.2.1.3 Memory Coherency Attribute (M)

This attribute is provided to allow improved performance in systems where hardware-enforced coherency is relatively slow, and software is able to enforce the required coherency. When M='0', there are no requirements to enforce data coherency. When M='1', the processor enforces data coherency.

When the M attribute is set, and the access is performed to memory, there is a hardware indication to the rest of the system that the access is global. Other processors affected by the access must then respond to this global access. For example, in a snooping bus design, the processor may assert some type of global access signal. Other processors affected by the access respond and signal whether the data is being shared. If the data in another processor is modified, then the location is updated and the access is retried.

Because instruction memory does not have to be coherent with data memory, some implementations may ignore the M attribute for instruction accesses. In a single-processor (or single-cache) system, performance might be improved by designating all pages as memory coherency not required.

Accesses to the same memory location using two effective addresses for which the M-bit settings differ 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, no software synchronization may be required for systems that use bus snooping. In some directory-based systems, software may be required to execute **dcbf** instructions on each processor to flush all storage locations accessed with M='0' before accessing those locations with M='1'.

5.2.1.4 W, I, and M Bit Combinations

Table 5-1 summarizes the six combinations of the WIM bits supported by the OEA. The combinations where WIM = '11x' are not supported.

Note: Either a '0' or '1' setting for the G-bit is allowed for each of these WIM bit combinations.

Table 5-1. Combinations of W, I, and M Bits

WIM Setting	Meaning
000	The processor may cache data (or instructions). A load or store operation whose target hits in the cache may use that entry in the cache. The processor does not need to enforce memory coherency for accesses it initiates.
001	Data (or instructions) may be cached. A load or store operation whose target hits in the cache may use that entry in the cache. The processor enforces memory coherency for accesses it initiates.
010	Caching is inhibited. The access is performed to memory, completely bypassing the cache. The processor does not need to enforce memory coherency for accesses it initiates.

Table 5-1. Combinations of W, I, and M Bits (Continued)

WIM Setting	Meaning
011	Caching is inhibited. The access is performed to memory, completely bypassing the cache. The processor enforces memory coherency for accesses it initiates.
100	Data (or instructions) may be cached. A load operation whose target hits in the cache may use that entry in the cache. Store operations are written to memory. The target location of the store may be cached and is updated on a hit. The processor does not need to enforce memory coherency for accesses it initiates.
101	Data (or instructions) may be cached. A load operation whose target hits in the cache may use that entry in the cache. Store operations are written to memory. The target location of the store may be cached and is updated on a hit. The processor enforces memory coherency for accesses it initiates.

5.2.1.5 Guarded Attribute (G)

When the guarded bit is set, the memory area (page) is designated as guarded. This setting can be used to protect certain pages from read accesses made by the processor that are not dictated directly by the program. If there are areas of physical memory that are not fully populated (in other words, there are holes in the physical memory map within this area), this setting can protect the system from undesired accesses caused by out-of-order load operations or instruction prefetches that could lead to the generation of the machine check exception. Also, the guarded bit can be used to prevent out-of-order (speculative) load operations or prefetches from occurring to certain peripheral devices that produce undesired results when accessed in this way.

Performing Operations Out of Order

An operation is said to be performed in-order if it is guaranteed to be required by the sequential execution model. Any other operation is said to be performed out of order.

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, and return from interrupt instructions, and exceptions, 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 exceptions 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 memory – The restrictions for this case are given in *Out-of-Order Accesses to Guarded Memory* on page 203.

No error of any kind other than a machine check exception may be reported due to an operation that is performed out of order, until such time as it is known that the operation is required by the sequential execution model. The only other permitted side effects (other than machine check) of performing an operation out of order are the following:

- Referenced and changed bits may be set as described in *Section 7.2.5 Page History Information*.
- Nonguarded memory locations that could be fetched into a cache by in-order execution may be fetched out of order into that cache.

Guarded Memory

Memory is said to be well behaved if the corresponding physical memory exists and is not defective, and if the effects of a single access to it are indistinguishable from the effects of multiple identical accesses to it. Data and instructions can be fetched out of order from well-behaved memory without causing undesired side effects.

Memory is said to be guarded if either:
(a) the G-bit is '1' in the relevant PTE or
(b) the processor is in real addressing mode (MSR[IR] = '0' or MSR[DR] = '0' for instruction fetches or data accesses respectively).

In case (b), all of memory is guarded for the corresponding accesses. In general, memory that is not well-behaved should be guarded. Because such memory may represent an I/O device or may include locations that do not exist, an out-of-order access to such memory may cause an I/O device to perform incorrect operations or may result in a machine check.

Note: If separate store instructions access memory that is both caching-inhibited and guarded, the accesses are performed in the order specified by the program. If an aligned, elementary load or store to caching-inhibited, guarded memory has accessed main memory and an external, decrementer, or imprecise-mode floating-point enabled exception is pending, the load or store is completed before the exception is taken.

Out-of-Order Accesses to Guarded Memory

The circumstances in which guarded memory may be accessed out of order are as follows:

- Load instruction – If a copy of the target location is in a cache, the location may be accessed in the cache or in main memory.
- Instruction fetch – In real addressing mode (MSR[IR] = '0'), an instruction may be fetched if any of the following conditions is met:
  - The instruction is in a cache. In this case, it may be fetched from that cache.
  - The instruction is in the same physical page as an instruction that is required by the sequential execution model or is in the physical page immediately following such a page.

  If MSR[IR] = '1', instructions may not be fetched from either no-execute segments or guarded memory. If the effective address of the current instruction is mapped to either of these kinds of memory when MSR[IR] = '1', an ISI exception is generated. However, it is permissible for an instruction from either of these kinds of memory to be in the instruction cache if it was fetched into that cache when its effective address was mapped to some other kind of memory. Thus, for example, the operating system can access an application's instruction segments as no-execute without having to invalidate them in the instruction cache.

Note: Software should ensure that only well-behaved memory is loaded into a cache, either by marking as caching-inhibited (and guarded) all memory that may not be well-behaved, or by marking such memory caching-allowed (and guarded) and referring only to cache blocks that are well-behaved.

If a physical page contains instructions that will be executed in real addressing mode (MSR[IR] = ‘0’), software should ensure that this physical page and the next physical page contain only well-behaved memory.

5.2.2 I/O Interface Considerations

Memory-mapped I/O interface operations are considered to address memory space and are therefore subject to the same coherency control as memory accesses. Depending on the specific I/O interface, the memory/cache access attributes (WIMG) and the degree of access ordering (requiring **eieio** or **sync** instructions) need to be considered. This is the recommended way of accessing I/O.

6. Exceptions

The operating environment architecture (OEA) portion of the PowerPC Architecture defines the mechanism by which PowerPC processors implement exceptions (referred to as interrupts in the architecture specification). Exception conditions may be defined at other levels of the architecture. For example, the user instruction set architecture (UISA) defines conditions that may cause floating-point exceptions; the OEA defines the mechanism by which the exception is taken.

The PowerPC exception mechanism allows the processor to change to supervisor state as a result of external signals, errors, or unusual conditions arising in the execution of instructions. When exceptions occur, information about the state of the processor is saved to certain registers and the processor begins execution at an address (exception vector) predetermined for each exception. Processing of exceptions begins in supervisor mode.

Although multiple exception conditions can map to a single exception vector, a more specific condition may be determined by examining a register associated with the exception—for example, the DSISR and the floating-point status and control register (FPSCR). Additionally, certain exception conditions can be explicitly enabled or disabled by software.

The PowerPC Architecture requires that exceptions be taken in program order; therefore, although a particular implementation may recognize exception conditions out of order, they are handled strictly in order with respect to the instruction stream. When an instruction-caused exception is recognized, any unexecuted instructions that appear earlier in the instruction stream, including any that have not yet entered the execute state, are required to complete before the exception is taken. For example, if a single instruction encounters multiple exception conditions, those exceptions are taken and handled sequentially. Likewise, exceptions that are asynchronous and precise are recognized when they occur, but are not handled until all instructions currently in the execute stage successfully complete execution and report their results.

Note: Exceptions can occur while an exception handler routine is executing, and multiple exceptions can become nested. It is up to the exception handler to save the appropriate machine state if it is desired to allow control to ultimately return to the excepting program.

In many cases, after the exception handler handles an exception, there is an attempt to execute the instruction that caused the exception. Instruction execution continues until the next exception condition is encountered. This method of recognizing and handling exception conditions sequentially guarantees that the machine state is recoverable and processing can resume without losing instruction results.

To prevent the loss of state information, exception handlers must save the information stored in SRR0 and SRR1 soon after the exception is taken to prevent this information from being lost due to another exception being taken.

In this chapter, the following terminology is used to describe the various stages of exception processing:

Recognition	Exception recognition occurs when the condition that can cause an exception is identified by the processor.
Taken	An exception is said to be taken when control of instruction execution is passed to the exception handler; that is, the context is saved and the instruction at the appropriate vector offset is fetched and the exception handler routine is begun in supervisor mode.
Handling	Exception handling is performed by the software linked to the appropriate vector offset. Exception handling is begun in supervisor mode (referred to as privileged state in the architecture specification).

6.1 Exception Classes

As specified by the PowerPC Architecture, all exceptions can be described as either precise or imprecise and either synchronous or asynchronous. Asynchronous exceptions are caused by events external to the processor's execution; synchronous exceptions are caused by instructions.

The PowerPC exception types are shown in *Table 6-1*.

Table 6-1. PowerPC Exception Classifications

Type	Exception
Asynchronous/nonmaskable	Machine Check System Reset
Asynchronous/maskable	External interrupt Decrementer
Synchronous/Precise	Instruction-caused exceptions, excluding floating-point imprecise exceptions
Synchronous/Imprecise	Instruction-caused imprecise exceptions (Floating-point imprecise exceptions)

Exceptions, their offsets, and conditions that cause them, are summarized in *Table 6-2*. The exception vectors described in the table correspond to physical address locations, relative to address 0. Refer to *Section 7.2.1.2 Predefined Physical Memory Locations* for a complete list of the predefined physical memory areas. Remaining sections in this chapter provide more complete descriptions of the exceptions and of the conditions that cause them.

Table 6-2. Exceptions and Conditions—Overview

Exception Type	Vector Offset (hex)	Causing Conditions
System reset	00100	The causes of system reset exceptions are implementation-dependent. If the conditions that cause the exception also cause the processor state to be corrupted such that the contents of SRR0 and SRR1 are no longer valid or such that other processor resources are so corrupted that the processor cannot reliably resume execution, the copy of the RI bit copied from the MSR to SRR1 is cleared.
Machine check	00200	The causes for machine check exceptions are implementation-dependent, but typically these causes are related to conditions such as bus parity errors or attempting to access an invalid physical address. Typically, these exceptions are triggered by an input signal to the processor. **Note:** Not all processors provide the same level of error checking. The machine check exception is disabled when MSR[ME] = ’0’. If a machine check exception condition exists and the ME bit is cleared, the processor goes into the checkstop state. If the conditions that cause the exception also cause the processor state to be corrupted such that the contents of SRR0 and SRR1 are no longer valid or such that other processor resources are so corrupted that the processor cannot reliably resume execution, the copy of the RI bit written from the MSR to SRR1 is cleared. **Note:** The physical address is referred to as real address in the architecture specification.)
DSI	00300	A DSI exception occurs when a data memory access cannot be performed for any of the reasons described in *Section 6.4.3 DSI Exception (0x00300)*. Such accesses can be generated by load/store instructions, certain memory control instructions, and certain cache control instructions.
Data Segment	00380	A Data Segment interrupt occurs if MSR[DR] = ’1’ and the translation of the effective address of any byte of the specified storage location is not found in the SLB. Refer to *Section 6.4.4 Data Segment Exception (0x00380)* for details.
ISI	00400	An ISI exception occurs when an instruction fetch cannot be performed for a variety of reasons described in *Section 6.4.5 ISI Exception (0x00400)*.

Table 6-2. Exceptions and Conditions—Overview (Continued)

Exception Type	Vector Offset (hex)	Causing Conditions
Instruction Segment	00480	An instruction segment exception occurs when no higher priority exception exists and next instruction to be executed cannot be fetched because instruction address translation is enabled (MSR[IR]=1) and the effective address cannot be translated to a virtual address.
External interrupt	00500	An external interrupt is generated only when an external interrupt is pending (typically signalled by a signal defined by the implementation) and the interrupt is enabled (MSR[EE] = ’1’).
Alignment	00600	An alignment exception may occur when the processor cannot perform a memory access for reasons described in *Section 6.4.8 Alignment Exception (0x00600)*. **Note:** An implementation is allowed to perform the operation correctly and not cause an alignment exception.
Program	00700	A program exception is caused by one of the following exception conditions, which correspond to bit settings in SRR1 and arise during execution of an instruction: • Floating-point enabled exception—A floating-point enabled exception condition is generated when MSR[FE0–FE1] ≠ ‘00’ and FPSCR[FEX] is set. The settings of FE0 and FE1 are described in *Table 6-3*. FPSCR[FEX] is set by the execution of a floating-point instruction that causes an enabled exception or by the execution of a Move to FPSCR instruction that sets both an exception condition bit and its corresponding enable bit in the FPSCR. These exceptions are described in *Section 3.3.6 Floating-Point Program Exceptions*.” • Illegal instruction—An illegal instruction program exception is generated when execution of an instruction is attempted with an illegal opcode or illegal combination of opcode and extended opcode fields or when execution of an optional instruction not provided in the specific implementation is attempted (these do not include those optional instructions that are treated as no-ops). The PowerPC instruction set is described in *Chapter 4, “Addressing Modes and Instruction Set Summary*.” See *Section 6.4.9 Program Exception (0x00700)* for a complete list of causes for an illegal instruction program exception. • Privileged instruction—A privileged instruction type program exception is generated when the execution of a privileged instruction is attempted and the MSR user privilege bit, MSR[PR], is set. This exception is also generated for **mtspr** or **mfspr** with an invalid SPR field if spr[0] = ’1’ and MSR[PR] = ’1’. • Trap—A trap type program exception is generated when any of the conditions specified in a trap instruction is met. For more information, refer to *Section 6.4.9 Program Exception (0x00700)*.”
Floating-point unavailable	00800	A floating-point unavailable exception is caused by an attempt to execute a floating-point instruction (including floating-point load, store, and move instructions) when the floating-point available bit is cleared, MSR[FP] = ’0’.
Decrementer	00900	The decrementer interrupt exception is taken if the exception is enabled (MSR[EE] = ’1’), and it is pending. The exception is created when the most-significant bit of the decrementer changes from 0 to 1. If it is not enabled, the exception remains pending until it is taken.
Reserved	00A00	This is reserved for implementation-specific exceptions.
Reserved	00B00	—
System call	00C00	A system call exception occurs when a System Call (**sc**) instruction is executed.
Trace	00D00	Implementation of the trace exception is optional. If implemented, it occurs if either the MSR[SE] = ’1’ and almost any instruction successfully completed or MSR[BE] = ’1’ and a branch instruction is completed. See *Section 6.4.13 Trace Exception (0x00D00)* for more information.
Reserved	00E00–00FFF	—
Performance monitor	00F00	The performance monitor exception is part of the optional performance monitor facility. If the performance monitor facility is not implemented or does not use this interrupt, the corresponding interrupt vector is treated as reserved.
Reserved	01000–02FFF	This is reserved for implementation-specific purposes. May be used for implementation-specific exception vectors or other uses.

6.1.1 Precise Exceptions

When any precise exception occurs, SRR0 points to either the instruction causing the exception or the instruction immediately following. The exception type and status bits determine which instruction is addressed. However, depending on the exception type, the instruction addressed by SRR0 and those following it might have started, but might not have completed execution.

When an exception occurs, instruction dispatch (the issuance of instructions by the instruction fetch unit to any instruction execution mechanism) is halted and the following synchronization is performed:

1. The exception mechanism waits for all previous instructions in the instruction stream to complete to a point where they will not report any exceptions.
2. The processor ensures that all previous instructions in the instruction stream complete in the context in which they began execution.
3. The exception mechanism implemented in hardware (the loading of registers SRR0 and SRR1) and the software handler (saving SRR0 and SRR1 in the stack and updating stack pointer, etc.) are responsible for saving and restoring the processor state.

The synchronization described conforms to the requirements for context synchronization. A complete description of context synchronization is described in *Section 6.1.2.1 Context Synchronization*.

6.1.2 Synchronization

The synchronization described in this section refers to the state of activities within the processor that performs the synchronization.

6.1.2.1 Context Synchronization

An instruction or event is context synchronizing if it satisfies all the requirements listed below. Such instructions and events are collectively called context-synchronizing operations. Examples of context-synchronizing operations include the **isync**, **sc,** and **rfid** instructions, the **mtmsr**[**d**] instruction if L = ‘0’, and most exceptions. A context-synchronizing operation has the following characteristics:

1. The operation causes instruction fetching and dispatching (the issuance of instructions by the instruction fetch mechanism to any instruction execution mechanism) to be halted.
2. The operation is not initiated or, in the case of **isync**, does not complete, until all instructions in execution have completed to a point at which they have reported all exceptions they will cause.
3. The operation ensures that the instructions that precede the operation will complete execution in the context (privilege, relocation, memory protection, etc.) in which they were initiated, except that the operation has no effect on the context in which the associated Reference and Change bit updates are performed.
4. If the operation either directly causes an exception (for example, the **sc** instruction causes a system call exception) or is an exception, then the operation is not initiated until there is no exception having a higher priority than the exception associated with the context-synchronizing operation.
5. The operation ensures that the instructions that follow the operation will be fetched and executed in the context established by the operation. (This requirement dictates that any prefetched instructions be discarded and that any effects and side effects of executing them out-of-order also be discarded, except as described in the *Section Out-of-Order Accesses to Guarded Memory*.)

A context-synchronizing operation is necessarily execution synchronizing. Unlike the **sync** instruction, a context-synchronizing operation need not wait for memory-related operations to complete on this or other processors, or for Referenced and Changed bits in the page table to be updated.

6.1.2.2 Execution Synchronization

An instruction is execution synchronizing if it satisfies the conditions of the first two items described above for context synchronization. The **sync** and **ptesync** instructions are treated like **isync** with respect to the second item described above (that is, the conditions described in the second item apply to the completion of **sync**). The **sync** and **mtmsr** instructions are examples of execution-synchronizing instructions.

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

6.1.2.3 Synchronous/Precise Exceptions

When instruction execution causes a precise exception, the following conditions exist at the exception point:

- SRR0 always points to the instruction causing the exception except for the **sc** instruction. In this case SRR0 points to the immediately following instruction. The instruction addressed can be determined from the exception type and status bits, which are defined in the description of each exception. In all cases SRR0 points to the first instruction that has not completed execution. The **sc** instruction always completes execution, updates the instruction pointer and reports the exception. Hence, SRR0 points to the instructions following **sc**.
- All instructions that precede the excepting instruction complete to a point where they will not report exceptions before the exception is processed. However, some memory accesses generated by these preceding instructions may not have been performed with respect to all other processors or system devices.
- The instruction causing the exception may not have begun execution, may have partially completed, or may have completed, depending on the exception type. Handling of partially executed instructions is described in *Section 6.1.4 Partially Executed Instructions*.
- Architecturally, no subsequent instruction has begun execution.

While instruction parallelism allows the possibility of multiple instructions reporting exceptions during the same cycle, they are handled one at a time in program order. Exception priorities are described in *Section 6.1.5 Exception Priorities*.

6.1.2.4 Asynchronous Exceptions

There are four asynchronous exceptions—system reset and machine check, which are nonmaskable and highest-priority exceptions, and external interrupt and decrementer exceptions which are maskable and low-priority. These two types of asynchronous exceptions are discussed separately.

System Reset and Machine Check Exceptions

System reset and machine check exceptions have the highest priority and can occur while other exceptions are being processed.

Note: Nonmaskable, asynchronous exceptions are never delayed; therefore, if two of these exceptions occur in immediate succession, the state information saved by the first exception may be overwritten when the subsequent exception occurs. Also, these exceptions are context-synchronizing if they are recoverable; the system uses the MSR[RI] to detect whether an exception is recoverable.

While a system is running the MSR[RI] bit is set. When an exception occurs a copy of the MSR register is stored in SRR1. Then most bits in the MSR are cleared including the RI bit with various exceptions (see the exceptions types for new setting of the MSR bits). The exception handler saves the state of the machine (saving SRR0 and SRR1 into the stack and updating the stack pointer) to a point that it can incur another exception. At this point the exception handler sets the MSR[RI] bit. Also the external interrupt can be re-enabled. Now you can clearly understand that if the exception handler ever sees in the SRR1 register a case where the MSR[RI] bit is not set, the exception is not recoverable (because the exception occurred while the machine state was being saved) and a system restart procedure should be initiated.

System reset and machine check exceptions cannot be masked by using the MSR[EE] bit. Furthermore, if the machine check enable bit, MSR[ME], is cleared and a machine check exception condition occurs, the processor goes directly into checkstop state as the result of the exception condition. Clearly, one never wants to run in this mode (MSR[ME] cleared) for extended periods of time. When one of these exceptions occur, the following conditions exist at the exception point:

- For system reset exceptions, SRR0 addresses the instruction that would have attempted to execute next if the exception had not occurred.
- For machine check exceptions, SRR0 holds either an instruction that would have completed or some instruction following it that would have completed if the exception had not occurred.
- An exception is generated such that all instructions preceding the instruction addressed by SRR0 appear to have completed with respect to the executing processor.

Note: MSR[RI] indicates whether enough of the machine state was saved to allow the processor to resume processing.

External Interrupt and Decrementer Exceptions

For the external interrupt and decrementer exceptions, the following conditions exist at the exception point (assuming these exceptions are enabled (MSR[EE] bit is set)):

- All instructions issued before the exception is taken and any instructions that precede those instructions in the instruction stream appear to have completed before the exception is processed.
- No subsequent instructions in the instruction stream have begun execution.
- SRR0 addresses the first instruction that has not completed execution.

That is, these exceptions are context-synchronizing. The external interrupt and decrementer exceptions are maskable. When the machine state register external interrupt enable bit is cleared (MSR[EE] = ’0’), these exception conditions are not recognized until the EE bit is set. MSR[EE] is cleared automatically when an exception is taken, to delay recognition of subsequent exception conditions. No two precise exceptions can be recognized simultaneously. Exception handling does not begin until all currently executing instructions complete and any synchronous, precise exceptions caused by those instructions have been handled. Exception priorities are described in *Section 6.1.5 Exception Priorities*.

6.1.3 Imprecise Exceptions

The PowerPC Architecture defines one imprecise exception, the imprecise mode floating-point enabled exception. This is implemented as one of the conditions that can cause a program exception.

6.1.3.1 Imprecise Exception Status Description

When the execution of an instruction causes an imprecise exception, SRR0 contains information related to the address of the excepting instruction as follows:

- SRR0 addresses either the instruction causing the exception or some instruction following the instruction causing the exception that generated the interrupt.
- The exception is generated such that all instructions preceding the instruction addressed by SRR0 have completed with respect to the processor.
- If the imprecise exception is caused by the context-synchronizing mechanism (due to an instruction that caused another exception—for example, an alignment or DSI exception), then SRR0 contains the address of the instruction that caused the exception, and that instruction may have been partially executed (refer to *Section 6.1.4 Partially Executed Instructions*).
- If the imprecise exception is caused by an execution-synchronizing instruction other than **sync**, **isync**, or **ptesync**, then SRR0 addresses the instruction causing the exception. Additionally, besides causing the exception, that instruction is considered not to have begun execution. If the exception is caused by the **sync**, **isync**, or **ptesync** instruction, SRR0 may address either the **sync**, **isync,** or **ptesync** instruction, or the following instruction.
- If the imprecise exception is not forced by either the context-synchronizing mechanism or the execution-synchronizing mechanism, then the instruction addressed by SRR0 is considered not to have begun execution if it is not the instruction that caused the exception.
- When an imprecise exception occurs, no instruction following the instruction addressed by SRR0 is considered to have begun execution.

6.1.3.2 Recoverability of Imprecise Floating-Point Exceptions

The enabled IEEE floating-point exception mode bits in the MSR (FE0 and FE1) together define whether IEEE floating-point exceptions are handled precisely, imprecisely, or whether they are taken at all. The possible settings are shown in *Table 6-3*. For further details, see *Section 3.3.6 Floating-Point Program Exceptions*.

Table 6-3. IEEE Floating-Point Program Exception Mode Bits

FE0	FE1	Mode
0	0	Floating-point exceptions ignored
0	1	Floating-point imprecise nonrecoverable
1	0	Floating-point imprecise recoverable
1	1	Floating-point precise mode

As shown in the table, the imprecise floating-point enabled exception has two modes—nonrecoverable and recoverable. These modes are specified by setting the MSR[FE0] and MSR[FE1] bits and are described as follows:

- *Imprecise nonrecoverable floating-point enabled mode*. MSR[FE0] = ’0’; MSR[FE1] = ‘1’. When an exception occurs, the exception handler is invoked at some point at or beyond the instruction that caused the exception. It may not be possible to identify the offending instruction or the data that caused the exception. Results from the offending instruction may have been used by or affected data of subsequent instructions executed before the exception handler was invoked.
- *Imprecise recoverable floating-point enabled mode*. MSR[FE0] = ’1’; MSR[FE1] = ’0’. When an exception occurs, the floating-point enabled exception handler is invoked at some point at or beyond the offending instruction that caused the exception. Sufficient information is provided to the exception handler that it can identify the offending instruction and correct any faulty data. In this mode, no incorrect data caused by the offending instruction have been used by or affected data of subsequent instructions that are executed before the exception handler is invoked.

Although these exceptions are maskable with these bits, they differ from other maskable exceptions in that the masking is usually controlled by the application program rather than by the operating system.

6.1.4 Partially Executed Instructions

The architecture permits certain instructions to be partially executed when an alignment exception or DSI exception occurs, or an imprecise floating-point exception is forced by an instruction that causes an alignment or DSI exception. They are as follows:

- Load multiple/string instructions that cause an alignment or DSI exception—Some registers in the range of registers to be loaded may have been loaded.
- Store multiple/string instructions that cause an alignment or DSI exception—Some bytes in the addressed memory range may have been updated.
- Non-multiple/string store instructions that cause an alignment or DSI exception—Some bytes just before the boundary may have been updated. If the instruction normally alters CR0 (**stwcx.** or **stdcx.**), CR0 is set to an undefined value. For instructions that perform register updates, the update register (**r**A) is not altered.
- Floating-point load instructions that cause an alignment or DSI exception—The target register may be altered. For update forms, the update register (**r**A) is not altered.

In the cases above, the number of registers and the amount of memory altered are implementation, instruction, and boundary-dependent. However, memory protection is not violated.

Note: An exception may result in the partial execution of a Load or Store instruction. For example, if the Page Table Entry that translates the address of the memory operand is altered, by a program running on another processor, such that the new contents of the Page Table Entry preclude performing the access, the alteration could cause the Load or Store instruction to be aborted after having been partially executed.

Partial execution is not allowed when integer load operations (except multiple/string operations) cause an alignment or DSI exception. The target register is not altered. For update forms of the integer load instructions, the update register (**r**A) is not altered.

6.1.5 Exception Priorities

Exceptions are roughly prioritized by exception class, as follows:

1. Nonmaskable, asynchronous exceptions have priority over all other exceptions—system reset and machine check exceptions (although the machine check exception condition can be disabled so that the condition causes the processor to go directly into the checkstop state). These two types of exceptions in this class cannot be delayed by exceptions in other classes, and do not wait for the completion of any precise exception handling.
2. Synchronous, precise exceptions are caused by instructions and are taken in strict program order.
3. If an imprecise exception exists (the instruction that caused the exception has been completed and is required by the sequential execution model), exceptions signaled by instructions subsequent to the instruction that caused the exception are not permitted to change the architectural state of the processor. The exception causes an imprecise program exception unless a machine check or system reset exception is pending.
4. Maskable asynchronous exceptions (external interrupt and decrementer exceptions) have lowest priority.

The exceptions are listed in *Table 6-4* in order of highest to lowest priority.

Table 6-4. Exception Priorities

Exception Class	Priority	Exception
Nonmaskable, asynchronous	1	System reset—The system reset exception has the highest priority of all exceptions. If this exception exists, the exception mechanism ignores all other exceptions and generates a system reset exception. When the system reset exception is generated, previously issued instructions can no longer generate exception conditions that cause a nonmaskable exception.
	2	Machine check—The machine check exception is the second-highest priority exception. If this exception occurs, the exception mechanism ignores all other exceptions (except reset) and generates a machine check exception. When the machine check exception is generated, previously issued instructions can no longer generate exception conditions that cause a nonmaskable exception.
Synchronous, precise	3	Instruction dependent— When an instruction causes an exception, the exception mechanism waits for any instructions prior to the excepting instruction in the instruction stream to complete. Any exceptions caused by these instructions are handled first. It then generates the appropriate exception if no higher priority exception exists when the exception is to be generated. **Note:** A single instruction can cause multiple exceptions. When this occurs, those exceptions are ordered in priority as indicated in the following: A. Integer loads and stores a. Program-illegal instruction b. DSI, Data Segment, or Alignment c. Trace (if implemented) B. Floating-point loads and stores a. Program-illegal instruction b. Floating-point unavailable c. DSI, Data Segment, or Alignment d. Trace (if implemented) C. Other floating-point instructions a. Floating-point unavailable b. Program—Precise-mode floating-point enabled exception c. Trace (if implemented) D. **rfid** and **mtmsrd** (or **mtmsr**) a. Program—Privileged Instruction b. Program—Precise-mode floating-point enabled exception c. Trace (if implemented), for **mtmsrd** (or **mtmsr**) only If precise-mode IEEE floating-point enabled exceptions are enabled and the FPSCR[FEX] bit is set, a program exception occurs no later than the next synchronizing event. E. Other instructions a. These exceptions are mutually exclusive and have the same priority: —Program: Trap — System call (**sc**) —Program: Privileged Instruction —Program: Illegal Instruction b. Trace (if implemented) F. ISI or Instruction Segment exception The ISI or Instruction Segment exception has the lowest priority in this category. It is only recognized when all instructions prior to the instruction causing this exception appear to have completed and that instruction is to be executed. The priority of this exception is specified for completeness and to ensure that it is not given more favorable treatment. An implementation can treat this exception as though it had a lower priority.

Table 6-4. Exception Priorities (Continued)

Exception Class	Priority	Exception
Imprecise	4	Program imprecise floating-point mode enabled exceptions—When this exception occurs, the exception handler is invoked at or beyond the floating-point instruction that caused the exception. The PowerPC Architecture supports recoverable and nonrecoverable imprecise modes, which are enabled by setting MSR[FE0-FE1] = ‘10’ or ‘01’, respectively. For more information see, *Section 6.1.3 Imprecise Exceptions*.
Maskable, asynchronous	5	External interrupt—The external interrupt mechanism waits for instructions currently or previously dispatched to complete execution. After all such instructions are completed, and any exceptions caused by those instructions have been handled, the exception mechanism generates this exception if no higher priority exception exists. This exception is enabled only if MSR[EE] is currently set. If EE is zero when the exception is detected, it is delayed until the bit is set.
	5	Decrementer—This exception is the lowest priority exception. When this exception is created, the exception mechanism waits for all other possible exceptions to be reported. It then generates this exception if no higher priority exception exists. This exception is enabled only if MSR[EE] is currently set. If EE is zero when the exception is detected, it is delayed until the bit is set.

Nonmaskable, asynchronous exceptions (namely, system reset or machine check exceptions) may occur at any time. That is, these exceptions are not delayed if another exception is being handled (although machine check exceptions can be delayed by system reset exceptions). As a result, state information for the interrupted exception handler may be lost.

All other exceptions have lower priority than system reset and machine check exceptions, and the exception might not be taken immediately when it is recognized. Only one synchronous, precise exception can be reported at a time. If a maskable, asynchronous or an imprecise exception condition occurs while instruction-caused exceptions are being processed, its handling is delayed until all exceptions caused by previous instructions in the program flow are handled and those instructions complete execution.

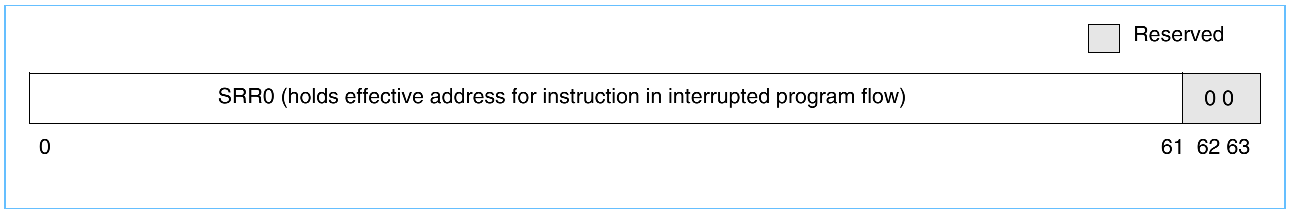
6.2 Exception Processing

Associated with each kind of exception is an exception vector, which contains the initial sequence of instructions that is executed when the corresponding exception occurs. Exception processing consists of saving a small part of the processor's state in certain registers, identifying the cause of the exception in other registers, and continuing execution at the corresponding exception vector location.

When an exception is taken, the processor uses the save/restore registers, SRR1 and SRR0, respectively, to save the contents of the MSR for the interrupted process and to help determine where instruction execution should resume after the exception is handled.

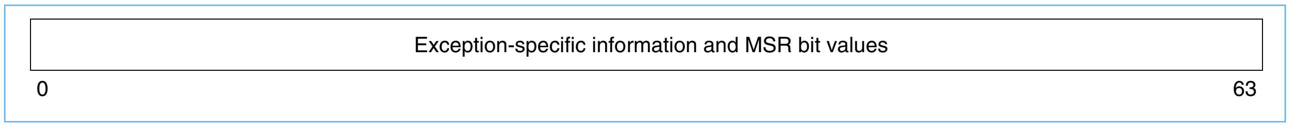
When an exception occurs, the address saved in SRR0 is used to help calculate where instruction processing should resume when the exception handler returns control to the interrupted process. Depending on the exception, this may be the address in SRR0 or at the next address in the program flow. All instructions in the program flow preceding this one will have completed execution and no subsequent instruction will have completed execution. This may be the address of the instruction that caused the exception or the next one (as in the case of a system call or trap exception). The SRR0 register is shown in *Figure 6-1*.

Figure 6-1. Machine Status Save/Restore Register 0


Reserved
SRR0 (holds effective address for instruction in interrupted program flow)
0 0
0
61 62 63

The save/restore register 1 (SRR1) is used to save machine status (selected bits from the MSR and other implementation-specific status bits as well) on exceptions and to restore those values when **rfid** is executed. SRR1 is shown in *Figure 6-2*.

Figure 6-2. Machine Status Save/Restore Register 1



When an exception occurs, SRR1 bits [33–36] and [42–47] are loaded with exception-specific information and MSR bits [0, 48–55, 57–59,62–63] are placed into the corresponding bit positions of SRR1. Depending on the implementation, additional bits of the MSR may be copied to SRR1.

Note: In some implementations, every instruction fetch when MSR[IR] = ’1’, and every data access requiring address translation when MSR[DR] = ’1’, can modify SRR0 and SRR1.

The MSR bits are shown in *Figure 6-3*.

Figure 6-3. Machine State Register (MSR)

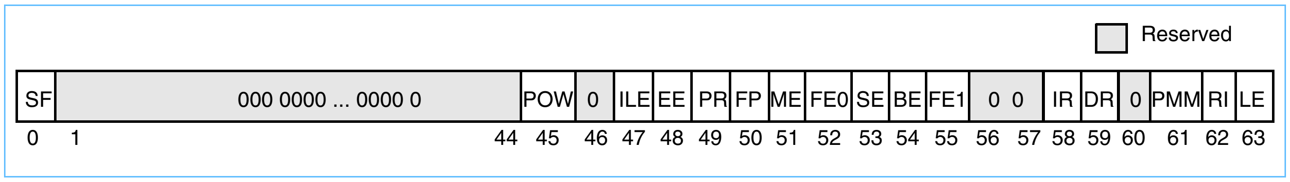


Table 6-5 shows the bit definitions for the MSR.

Table 6-5. MSR Bit Settings

Bit(s)	Name	Description
0	SF	Sixty-four bit mode 0 The 64-bit processor runs in 32-bit mode. 1 The 64-bit processor runs in 64-bit mode. Note that this is the default setting.
1	—	Reserved
64-BIT BRIDGE 2	ISF	Exception 64-bit mode (optional). When an exception occurs, this bit is copied into MSR[SF] to select 64 or 32-bit mode for the context established by the exception. **Note:** If the bridge function is not implemented, this bit is treated as reserved.
3–44	—	Reserved
45	POW	Power management enable 0 Power management disabled (normal operation mode) 1 Power management enabled (reduced power mode) **Note:** Power management functions are implementation-dependent. If the function is not implemented, this bit is treated as reserved.
46	—	Reserved
47	ILE	This is part of the optional little-endian facility. If the little-endian facility is implemented, and an exception occurs, this bit is copied into MSR[LE] to select the endian mode for the context established by the exception.
48	EE	External interrupt enable 0 While the bit is cleared, the processor delays recognition of external interrupts and decrementer exception conditions. 1 The processor is enabled to take an external interrupt or the decrementer exception.
49	PR	Privilege level 0 The processor can execute both user and supervisor-level instructions. 1 The processor can only execute user-level instructions. **Note:** Any instruction or event that set MSR[PR] also sets MSR[EE], MSR[IR], and MSR[DR].
50	FP	Floating-point available 0 The processor prevents dispatch of floating-point instructions, including floating-point loads, stores, and moves. 1 The processor can execute floating-point instructions.
51	ME	Machine check enable 0 Machine check exceptions are disabled. 1 Machine check exceptions are enabled. **Note:** The only instruction that can alter MSR[ME] is the **rfid** instruction.
52	FE0	Floating-point exception mode 0 (see *Table 2-8*).
53	SE	Single-step trace enable (Optional) 0 The processor executes instructions normally. 1 The processor generates a single-step trace exception upon the successful execution of the next instruction (unless that instruction is **rfid**, which is never trace). Successful completion means that the instruction caused no other interrupt. **Note:** If the function is not implemented, this bit is treated as reserved.
54	BE	Branch trace enable (Optional) 0 The processor executes branch instructions normally. 1 The processor generates a branch trace exception after completing the execution of a branch instruction, regardless of whether the branch was taken. **Note:** If the function is not implemented, this bit is treated as reserved.
55	FE1	Floating-point exception mode 1 (See *Table 2-8*).

Table 6-5. MSR Bit Settings (Continued)

Bit(s)	Name	Description
56	—	Reserved
57	—	Reserved
58	IR	Instruction address translation 0 Instruction address translation is disabled. 1 Instruction address translation is enabled. For more information, see *Chapter 7, Memory Management*.
59	DR	Data address translation 0 Data address translation is disabled. 1 Data address translation is enabled. For more information, see *Chapter 7, Memory Management*.
60	—	Reserved
61	PMM	Performance monitor mark. This bit is part of the optional performance monitor facility. If the performance monitor facility is not implemented or does not use this bit, then this bit is treated as reserved.
62	RI	Recoverable exception (for system reset and machine check exceptions). 0 Exception is not recoverable. 1 Exception is recoverable. For more information, see *Chapter 6, Exceptions*.
63	LE	This is part of the optional little-endian facility. If the little-endian facility is implemented, then the bit has the following meaning: 0 The processor runs in big-endian mode. 1 The processor runs in little-endian mode. If the little-endian facility is not implemented or does not use this bit, then this bit is treated as reserved.

TEMPORARY 64-BIT BRIDGE

Bit [2] of the MSR (MSR[ISF]) may optionally be used by a 64-bit implementation to control the mode (64-bit or 32-bit) that is entered when an exception is taken. If this bit is implemented, it has the following properties:

- When an exception is taken, the value of MSR[ISF] is copied to MSR[SF].
- When an exception is taken, MSR[ISF] is not altered.
- No software synchronization is required before or after altering MSR[ISF]. Refer to *Section 2.3.16 Synchronization Requirements for Special Registers and for Lookaside Buffers* for more information on synchronization requirements for altering other bits in the MSR.

If the MSR[ISF] bit is not implemented, it is treated as reserved except that the value is assumed to be ‘1’ for exception processing.

When an exception occurs instruction fetching, dispatching, decoding of instructions stops. The processor waits until all previous instructions have completed to a point where no other exceptions will be reported. SRR0 is loaded with the address where program execution will resume when the exception has been processed. SRR1 is loaded with the MSR register along with any status bits for this exception. A new value is loaded into the MSR and instruction execution resumes at the entry point for the exception handler under the influence of the new MSR.

The data address register (DAR) may be used by several exceptions (for example, DSI and alignment exceptions) to identify the address of a memory element.

6.2.1 Enabling and Disabling Exceptions

When a condition exists that may cause an exception to be generated, it must be determined whether the exception is enabled for that condition as follows:

- IEEE floating-point enabled exceptions (a type of program exception) are ignored when both MSR[FE0] and MSR[FE1] are cleared. If either of these bits is set, all IEEE enabled floating-point exceptions are taken and cause a program exception.
- Asynchronous, maskable exceptions (that is, the external and decrementer interrupts) are enabled by setting the MSR[EE] bit. When MSR[EE] = ’0’, recognition of these exception conditions is delayed. MSR[EE] is cleared automatically when an exception is taken, to delay recognition of conditions causing those exceptions.
- A machine check exception can only occur if the machine check enable bit, MSR[ME], is set. If MSR[ME] is cleared, the processor goes directly into a checkstop state when a machine check exception condition occurs.

6.2.2 Steps for Exception Processing

After it is determined that the exception can be taken (by confirming that any instruction-caused exceptions occurring earlier in the instruction stream have been handled, and by confirming that the exception is enabled for the exception condition), the processor does the following:

1. The machine status save/restore register 0 (SRR0) is loaded with an instruction address that depends on the type of exception. See the individual exception description for details about how this register is used for specific exceptions. Normally, SRR0 contains the address to the first instruction to execute if the exception handler resumes program execution.
2. SRR1 bits [33–36] and [42–47] are loaded with information specific to the exception type.
3. SRR1 bits [0-32, 37-41, 48–63] are loaded with a copy of the corresponding bits of the MSR. Depending on the implementation, additional bits from the MSR may be saved in SRR1.
4. The MSR is set as described in *Table 6-6*. The new values take effect beginning with the fetching of the first instruction of the exception-handler routine located at the exception vector address.

   Note: MSR[IR] and MSR[DR] are cleared for all exception types; therefore, address translation is disabled for both instruction fetches and data accesses beginning with the first instruction of the exception-handler routine. The MSR[ILE] bit setting at the time of the exception is copied to MSR[LE] when the exception is taken (as shown in *Table 6-6*).

TEMPORARY 64-BIT BRIDGE

Similar to MSR[ILE], the MSR[ISF] bit setting at the time of the exception is copied to MSR[SF] when the exception is taken (if the ISF bit is implemented).

5. The MSR[RI] bit is cleared. This indicates that the interrupt handler is operating in the “window-of-vulnerability” and cannot recover if another exception now occurs. After the machine state is saved (SRR0 and SRR1) and stack pointer has been updated, the exception handler sets this bit to indicate that it could now handle another exception. See *System Reset and Machine Check Exceptions* on page 210 for more details.
6. Instruction fetch and execution resumes, using the new MSR value, at the address specified by the exception’s vector offset. For a machine check exception that occurs when MSR[ME] = ’0’ (machine check exceptions are disabled), the checkstop state is entered (the machine stops executing instructions). See *Section 6.4.2 Machine Check Exception (0x00200)*.

In some implementations, any instruction fetch with MSR[IR] = ’1’ and any load or store with MSR[DR] = ’1’ might cause SRR0 and SRR1 to be modified.

Note: Exceptions do not clear reservations obtained with **lwarx** or **ldarx**.

6.2.3 Returning from an Exception Handler

The Return from Interrupt Doubleword (**rfid**) instruction performs context synchronization by allowing previously issued instructions to complete before returning to the interrupted process. Execution of the **rfid** instruction ensures the following:

- All previous instructions have completed to a point where they can no longer cause an exception.
- Previous instructions complete execution in the context (privilege, protection, and address translation) under which they were issued.
- The **rfid** instruction copies SRR1 bits back into the MSR.
- The instructions following this instruction execute in the context established by this instruction.

For a complete description of context synchronization, refer to *Section 6.1.2.1 Context Synchronization*.

6.3 Process Switching

The operating system should execute the following when processes are switched:

- The **sync** instruction, which orders the effects of instruction execution. All instructions previously initiated appear to have completed before the **sync** instruction completes, and no subsequent instructions appear to be initiated until the **sync** instruction completes.
- The **isync**/**rfid** instruction, which waits for all previous instructions to complete and then discards any fetched instructions, causing subsequent instructions to be fetched (or refetched) from memory and to execute in the context (privilege, translation, protection, etc.) established by the previous instructions.
- The **stwcx.**/**stdcx.** instruction, to clear any outstanding reservations, which ensures that an **lwarx**/**ldarx** instruction in the old process is not paired with an **stwcx./stdcx.** instruction in the new process. This is necessary because some implementations of the PowerPC Architecture do not do an address compare when the **stwcx./stdcx.** is executed. Only the reservation is required for the **stwcx./stdcx.** to be successful.

The operating system should handle MSR[RI] as follows:

- In machine check and system reset exception handlers—if the SRR1 bit corresponding to MSR[RI] is cleared, the exception is not recoverable.

- In each exception handler—when enough state information has been saved that a machine check or system reset exception can reconstruct the previous state, set MSR[RI].
- At the end of each exception handler—clear MSR[RI], set the SRR0 and SRR1 registers appropriately, update stack pointers, and then execute **rfid**.

Note: The [RI] bit being set indicates that, with respect to the processor, enough processor state data is valid for the processor to continue, but it does not guarantee that the interrupted process can resume.

6.4 Exception Definitions

Table 6-6 shows all the types of exceptions that can occur and certain MSR bit settings when the exception handler is invoked. Depending on the exception, certain of these bits are stored in SRR1 when an exception is taken. The following subsections describe each exception in detail.

Table 6-6. MSR Setting Due to Exception

Exception Type	MSR Bit																
	SF[1,2]	ISF[2]	POW	ILE	EE	PR	FP	ME	FE0	SE	BE	FE1	PMM	IR	DR	RI	LE
System reset	1	—	0	—	0	0	0	—	0	0	0	0	0	0	0	0	ILE
Machine check	1	—	0	—	0	0	0	0	0	0	0	0	0	0	0	0	ILE
DSI	1	—	0	—	0	0	0	—	0	0	0	0	0	0	0	0	ILE
Data segment	1	—	0	—	0	0	0	—	0	0	0	0	0	0	0	0	ILE
ISI	1	—	0	—	0	0	0	—	0	0	0	0	0	0	0	0	ILE
Instruction Segment	1	—	0	—	0	0	0	—	0	0	0	0	0	0	0	0	ILE
External	1	—	0	—	0	0	0	—	0	0	0	0	0	0	0	0	ILE
Alignment	1	—	0	—	0	0	0	—	0	0	0	0	0	0	0	0	ILE
Program	1	—	0	—	0	0	0	—	0	0	0	0	0	0	0	0	ILE
Floating-point unavailable	1	—	0	—	0	0	0	—	0	0	0	0	0	0	0	0	ILE
Decrementer	1	—	0	—	0	0	0	—	0	0	0	0	0	0	0	0	ILE
System call	1	—	0	—	0	0	0	—	0	0	0	0	0	0	0	0	ILE
Trace exception	1	—	0	—	0	0	0	—	0	0	0	0	0	0	0	0	ILE
Performance Monitor	1	—	0	—	0	0	0	—	0	0	0	0	0	0	0	0	ILE

0	Bit is cleared.
1	Bit is set.
ILE	Bit is copied from the ILE bit in the MSR.
—	Bit is not altered.

Reading of reserved bits may return 0, even if the value last written to it was 1.
[1]64-bit implementations only.

Temporary 64-Bit Bridge
[2] When the 64-bit bridge is implemented in a 64-bit processor and the MSR[ISF] bit is implemented, the value of the MSR[ISF] bit is copied to the MSR[SF] bit when an exception is taken.

6.4.1 System Reset Exception (0x00100)

The system reset exception is a nonmaskable, asynchronous exception signaled to the processor typically through the assertion of a system-defined signal; see *Table 6-7*.

Table 6-7. System Reset Exception—Register Settings

Register	Setting Description							
SRR0	Set to the effective address of the instruction that the processor would have attempted to execute next if no exception conditions were present.							
SRR1	0	Loaded with equivalent bit from the MSR						
	33–36	Cleared						
	42–47	Cleared						
	48–55	Loaded with equivalent bits from the MSR						
	57–59	Loaded with equivalent bits from the MSR						
	62	Loaded from the equivalent MSR bit, MSR[RI], if the exception is recoverable; otherwise cleared.						
	63	Loaded with equivalent bit from the MSR						
	Note: Depending on the implementation, additional bits in the MSR may be copied to SRR1. If the processor state is corrupted to the extent that execution cannot resume reliably, the bit corresponding to MSR[RI], (SRR1[62] in 64-bit implementations and SRR1[30] in 32-bit implementations), is cleared.							
MSR	SF	1	PR	0			DR	0
	POW	0	FP	0	BE	0	PMM	0
	ILE	—	ME	—	FE1	0	RI	0
	EE	0	FE0	0			LE	Set to value of ILE

When a system reset exception is taken, instruction execution continues at effective address 0x0000_0000_0000_0100.

If the exception is recoverable, the value of the MSR[RI] bit is copied to the corresponding SRR1 bit. The exception functions as a context-synchronizing operation. If a reset exception causes the loss of:

- A machine check exception,
- An external exception (interrupt or decrementer),
- Floating-point enabled type program exception,

then the exception is not recoverable. If the SRR1 bit corresponding to MSR[RI] is cleared, the exception is context-synchronizing only with respect to subsequent instructions.

Note: Each implementation provides a means for software to distinguish between power-on reset and other types of system resets (such as soft reset).

6.4.2 Machine Check Exception (0x00200)

If no higher-priority exception is pending (namely, a system reset exception), the processor initiates a machine check exception when the appropriate condition is detected.

Note: The causes of machine check exceptions are implementation and system-dependent, and are typically signalled to the processor by the assertion of a specified signal on the processor interface.

When a machine check condition occurs and MSR[ME] = ’1’, the exception is recognized and handled. If MSR[ME] = ’0’ and a machine check occurs, the processor generates an internal checkstop condition. When a processor is in checkstop state, instruction processing is suspended and generally cannot continue without resetting the processor. Some implementations may preserve some or all of the internal state of the processor when entering the checkstop state, so that the state can be analyzed as an aid in problem determination.

In general, it is expected that a bus error signal would be used by a memory controller to indicate a memory parity error or an uncorrectable memory ECC error.

Note: The resulting machine check exception has priority over any exceptions caused by the instruction that generated the bus operation.

If a machine check exception causes an exception that is not context-synchronizing, the exception is not recoverable. Also, a machine check exception is not recoverable if it causes the loss of one of the following:

- An external exception (interrupt or decrementer)
- Floating-point enabled type program exception

If the SRR1 bit corresponding to MSR[RI] is cleared, the exception is context-synchronizing only with respect to subsequent instructions. If the exception is recoverable, the SRR1 bit corresponding to MSR[RI] is set and the exception is context-synchronizing.

Note: If the error is caused by the memory subsystem, incorrect data could be loaded into the processor and register contents could be corrupted regardless of whether the exception is considered recoverable by the SRR1 bit corresponding to MSR[RI].

On some implementations, a machine check exception may be caused by referring to a nonexistent physical (real) address, either because translation is disabled (MSR[IR] or MSR[DR] = ’0’) or through an invalid translation. On such a system, execution of the **dcbz** instruction can cause a delayed machine check exception by introducing a block into the data cache that is associated with an invalid physical (real) address. A machine check exception could eventually occur when and if a subsequent attempt is made to store that block to memory (for example, as the block becomes the target for replacement, or as the result of executing a **dcbst** instruction).

When a machine check exception is taken, registers are updated as shown in *Table 6-8*.

Table 6-8. Machine Check Exception—Register Settings

Register	Setting Description							
SRR0	On a best-effort basis, implementations can set this to an EA of some instruction that was executing or about to be executing when the machine check condition occurred.							
SRR1	Bit [62] is loaded from MSR[RI] if the processor is in a recoverable state. Otherwise cleared. The setting of all other SRR1 bits is implementation-dependent.							
MSR	SF	1	PR	0			DR	0
	POW	0	FP	0	BE	0	PMM	0
	ILE	—	ME [1]	—	FE1	0	RI	0
	EE	0	FE0	0			LE	Set to value of ILE
DSISR	Implementation dependent.							
DAR	Implementation dependent.							

1. When a machine check exception is taken, the exception handler should set MSR[ME] as soon as it is practical to handle another machine check exception. Otherwise, subsequent machine check exceptions cause the processor to automatically enter the checkstop state.

If MSR[RI] is set, the machine check exception may still be unrecoverable in the sense that execution can resume in the same context that existed before the exception.

When a machine check exception is taken, instruction execution continues at effective address 0x0000_0000_0000_0200.

6.4.3 DSI Exception (0x00300)

A DSI (data storage interrupt) exception occurs when no higher priority exception exists and a data memory access cannot be performed. The condition that caused the DSI exception can be determined by reading the DSISR, a supervisor-level SPR (SPR18) register that can be read by using the **mfspr** instruction. *Table 6-9* lists bit settings and indicates which memory element is pointed to by the DAR. DSI exceptions can be generated by load/store instructions, cache-control instructions (**icbi**, **dcbi**, **dcbz**, **dcbst**, and **dcbf**), or the **eciwx**/**ecowx** instructions for any of the following reasons:

- The effective address cannot be translated. That is, there is a page fault for this portion of the translation, so a DSI exception must be taken to retrieve the page and update the translation tables. For example read a page from a storage device such as a hard disk drive.
- The instruction is not supported for the type of memory addressed. For **lwarx**/**stwcx.** and **ldarx**/**stdcx.** instructions that reference a memory location that is write-through required. If the exception is not taken, the instructions execute correctly.
- The access violates memory protection.
- The execution of an **eciwx** or **ecowx** instruction is disallowed because the external access register enable bit (EAR[E]) is cleared.
- A data address compare match occurs.

- A data address breakpoint register (DABR) match occurs. The DABR facility is optional to the PowerPC Architecture, but if one is implemented, it is recommended, but not required, that it be implemented as follows. A data address breakpoint match is detected for a load or store instruction if the three following conditions are met for any byte accessed:
  - EA[0–60] = DABR[DAB]
  - MSR[DR] = DABR[BT]
  - The instruction is a store and DABR[DW] = ’1’, or the instruction is a load and DABR[DR] = ’1’.

  The DABR is described in *Section 2.3.13 Data Address Breakpoint Register (DABR)*. In 32-bit mode of 64-bit implementations, the high-order 32 bits of the EA are treated as zero for the purpose of detecting a match; the DAR settings are described in *Table 6-9*. If the above conditions are satisfied, it is undefined whether a match occurs in the following cases:
  - The instruction is store conditional but the store is not performed.
  - The instruction is a load/store string of zero length.
  - The instruction is **dcbz**, **eciwx**, or **ecowx.**

  The cache management instructions other than **dcbz** never cause a match. If **dcbz** causes a match, some or all of the target memory locations may have been updated. For the purpose of determining whether a match occurs, **eciwx** is treated as a load, and **ecowx** and **dcbz** are treated as stores.

If an **stwcx.**/**stdcx.** instruction has an effective address for which a normal store operation would cause a DSI exception but the processor does not have the reservation from **lwarx**/**ldarx**, whether a DSI exception is taken is then implementation-dependent.

If the value in XER[25–31] indicates that a load or store string instruction has a length of zero, a DSI exception does not occur, regardless of the effective address.

The condition that caused the exception is defined in the DSISR. As shown in *Table 6-9*, this exception also sets the data address register (DAR).

Table 6-9. DSI Exception—Register Settings

Register	Setting Description							
SRR0	Set to the effective address of the instruction that caused the exception.							
SRR1	33–36	Cleared						
	42–47	Cleared						
	Others	Loaded with equivalent bits from the MSR						
	Note: Depending on the implementation, additional bits in the MSR may be copied to SRR1.							
MSR	SF	1	PR	0	SE	0	DR	0
	POW	0	FP	0	BE	0	PMM	0
	ILE	—	ME	—	FE1	0	RI	0
	EE	0	FE0	0			LE	Set to value of ILE
DSISR	0	Set to ‘0’.						
	1	Set if MSR[DR] = ’1’ and the translation for an attempted access is not found in the primary page table entry group (PTEG), or in the secondary PTEG (page fault condition); otherwise cleared.						
	2–3	Cleared						
	4	Set if a memory access is not permitted by the memory protection mechanism; otherwise cleared.						
	5	Set if the access is due to a **lwarx**, **ldarx**, **stwcx.,** or **stdcx.** instruction that addresses memory that is Write Through Required or Caching Inhibited; otherwise cleared.						
	6	Set for a store, **dcbz**, or **ecowx** instruction otherwise cleared.						
	7–8	Cleared						
	9	Set if a data address compare match or a DABR match occurs. Otherwise cleared.						
	10	Cleared						
	11	Set if the instruction is an **eciwx** or **ecowx** and EAR[E] = ’0’; otherwise cleared.						
	12–14	Cleared						
	15	Set if MSR[DR] = ’1’, the translation for an attempted access is found in the SLB, the translation is not found in the primary PTEG or in the secondary PTEG, and LSLBE[L] = ’1’; otherwise cleared.						
	16–31	Cleared						
	If multiple Data Storage exceptions occur for a given effective address, any one or more of the bits corresponding to these exceptions may be set in the DSISR.							
DAR	Set to the effective address of a memory element as described in the following list: • A Data Storage exception occurs for reasons other than DABR match or, for **eciwx** and **ecowx**, EAR[E] = ’0’ – A byte in the block that caused the exception, for a cache management instruction – A byte in the first aligned doubleword for which access was attempted in the page that caused the exception, for a Load, Store, **eciwx**, or **ecowx** instruction **Note:** If the exception occurs when a 64-bit processor is running in 32-bit mode, the 32 high-order bits are cleared.							

When a DSI exception is taken, instruction execution resumes at effective address 0x0000_0000_0000_0300.

6.4.4 Data Segment Exception (0x00380)

A data segment interrupt occurs when no higher priority exception exists and a data access cannot be performed because data address translation is enabled (MSR[DR] = ’1’) and the effective address of any byte of the memory location specified by a Load, Store, **icbi**, **dcbz**, **dcbst**, **dcbf**, **eciwx**, or **ecowx** instruction cannot be translated to a virtual address.

If a **stwcx.** or **stdcx.** would not perform its store in the absence of a data segment interrupt, and a nonconditional Store to the specified effective address would cause a data segment interrupt, it is implementation-dependent whether a data segment interrupt occurs.

If a Move Assist instruction has a length of zero (in the XER), a data segment interrupt does not occur, regardless of the effective address.

Table 6-10 describes the registers affected by the data segment exception.

Table 6-10. Data Segment Exception—Register Settings

Register	Setting Description							
SRR0	Set to the effective address of the instruction that caused the exception.							
SRR1	33–36 42–47 48–55 57–59 62–63	Cleared Cleared Loaded with equivalent bits from the MSR Loaded with equivalent bits from the MSR Loaded with equivalent bits from the MSR						
MSR	SF POW ILE EE	1 0 — 0	PR FP ME FE0	0 0 — 0	SE BE FE1	0 0 0	DR PMM RI LE	0 0 0 Set to value of ILE
DSISR	Set to an undefined value							
DAR	Set to the effective address of a memory element as described in the following list: • A Data Storage exception occurs for reasons other than DABR match or, for **eciwx** and **ecowx**, EAR[E] = ’0’ – A byte in the block that caused the exception, for a cache management instruction – A byte in the first aligned doubleword for which access was attempted in the page that caused the exception, for a Load, Store, **eciwx**, or **ecowx** instruction **Note:** If the exception occurs when a 64-bit processor is running in 32-bit mode, the 32 high-order bits are cleared.							

Execution resumes at effective address 0x0000_0000_0000_0380.

6.4.5 ISI Exception (0x00400)

An instruction storage interrupt (ISI) exception occurs when no higher priority exception exists and an attempt to fetch the next instruction to be executed fails for any of the following reasons:

- Instruction address translation is enabled (MSR[IR] = ’1’) and the virtual address cannot be translated to a real address.
- The fetch access violates memory protection.

Register settings for ISI exceptions are shown in *Table 6-11*.

Table 6-11. ISI Exception—Register Settings

Register	Setting Description			
SRR0	Set to the effective address of the instruction that the processor would have attempted to execute next if no exception conditions were present (if the exception occurs on attempting to fetch a branch target, SRR0 is set to the branch target address).			
SRR1	0-32	Loaded with equivalent bit from the MSR		
	33	Set if MSR[IR] = ’1’ and the translation of an attempted access is not found in the primary page table entry group (PTEG), or in the secondary PTEG; otherwise cleared		
	34	Cleared		
	35	Set if the fetch access occurs when MSR[IR] = ’1’ and is to No-execute storage, to Guarded storage, or to a segment for which bit [57] of the Segment Table Entry is set to 1;. Otherwise, cleared.		
	36	Set if a memory access is not permitted by the page protection mechanism, described in *Chapter 7, Memory Management*”; otherwise cleared.		
	42–47	Cleared		
	Others	Loaded with equivalent bits from the MSR		
	Note: Only one of the bits [33, 35, 36, and 42] can be set. Also, note that depending on the implementation, additional bits in the MSR may be copied to SRR1.			
MSR	SF 1	PR 0	SE 0	DR 0
	POW 0	FP 0	BE 0	PMM 0
	ILE —	ME —	FE1 0	RI 0
	EE 0	FE0 0		LE Set to value of ILE

If multiple instruction storage exceptions occur due to attempting to fetch a single instruction, any one or more of the bits corresponding to these exceptions may be set to ‘1’ in SRR1. More than one bit may be set to ‘1’ in SRR1 in the following combinations.

33, 35
33, 47
33, 35, 47
35, 36

When an ISI exception is taken, instruction execution resumes at effective address 0x0000_0000_0000_0400.

6.4.6 Instruction Segment Exception (x0480)

An instruction segment exception occurs when no higher priority exception exists and the next instruction to be executed cannot be fetched because instruction address translation is enabled (MSR[IR] = ’1’) and the translation of the effective address of the next instruction to be executed is not found in the SLB.

Register settings for instruction segment exceptions are shown in *Table 6-11*.

Table 6-12. Instruction Segment Exception—Register Settings

Register	Setting Description							
SRR0	Set to the effective address of the instruction that the processor would have attempted to execute next if no exception conditions were present (if the exception occurs on attempting to fetch a branch target, SRR0 is set to the branch target address).							
SRR1	32-36	Cleared						
	42–47	Cleared						
	Others	Loaded with equivalent bits from the MSR						
MSR	SF	1	PR	0	SE	0	DR	0
	POW	0	FP	0	BE	0	PMM	0
	ILE	—	ME	—	FE1	0	RI	0
	EE	0	FE0	0			LE	Set to value of ILE

When an instruction segment exception is taken, instruction execution resumes at effective address 0x0000_0000_0000_0480.

6.4.7 External Interrupt (0x00500)

An external interrupt exception is signaled to the processor by the assertion of the external interrupt signal. The exception may be delayed by other higher priority exceptions or if the MSR[EE] bit is ‘0’ when the exception is detected.

Note: The occurrence of this exception does not cancel the external request.

The register settings for the external interrupt exception are shown in *Table 6-13*.

Table 6-13. External Interrupt—Register Settings

Register	Setting Description							
SRR0	Set to the effective address of the instruction that the processor would have attempted to execute next if no interrupt conditions were present.							
SRR1	0	Loaded with equivalent bit from the MSR						
	33–36	Cleared						
	42–47	Cleared						
	48–55	Loaded with equivalent bits from the MSR						
	57–59	Loaded with equivalent bits from the MSR						
	62–63	Loaded with equivalent bits from the MSR						
	Note: Depending on the implementation, additional bits in the MSR may be copied to SRR1.							
MSR	SF	1	PR	0	SE	0	DR	0
	POW	0	FP	0	BE	0	PMM	0
	ILE	—	ME	—	FE1	0	RI	0
	EE	0	FE0	0			LE	Set to value of ILE

When an external interrupt exception is taken, instruction execution resumes at effective address 0x0000_0000_0000_0500.

6.4.8 Alignment Exception (0x00600)

This section describes conditions that can cause alignment exceptions in the processor. Similar to DSI exceptions, alignment exceptions use the SRR0 and SRR1 to save the machine state and the DSISR to determine the source of the exception. An alignment exception occurs when no higher priority exception exists and the implementation cannot perform a memory access for one of the following reasons:

- The operand of a floating-point load or store instruction is not word-aligned or crosses a virtual page boundary.
- The operand of **lmw**, **stmw**, **lwarx**, **ldarx**, **stwcx.**, **stdcx.**, **eciwx**, or **ecowx** is not aligned.
- The operand of a single-register load or store is not aligned and the processor is in little-endian mode.
- The instruction is **lmw**, **stmw**, **lswi**, **lswx**, **stswi**, or **stswx** and the operand is in memory that is Write Through Required or Caching Inhibited, or the processor is in little-endian mode.
- The operand of **lmw** or **stmw** crosses a segment boundary, or crosses a boundary between virtual pages that have different memory control attributes.
- The operand of a load or store is not aligned and is in memory that is write-through required or caching inhibited.
- The operand of **dcbz**, **lwarx**, **ldarx**, **stwcx.**, or **stdcx.**, is in memory that is write-through-required or caching inhibited.

If a **stwcx.** or **stdcx.** would not perform its store in the absence of an alignment exception and the specified effective address refers to memory that is Write Through Required or Caching Inhibited, it is implementation-dependent whether an alignment exception occurs.

Setting the DSISR and DAR as described below is optional for implementations on which alignment exceptions occur rarely, if ever, for cases that the alignment exception handler emulates. For such implementations, if the DSISR and DAR are not set as described below they are set to undefined values.

The term, ‘protection boundary’, refers to the boundary between protection domains. A protection domain is a segment, a virtual 4-Kbyte page or implementation specific larger size, or a range of unmapped effective addresses. Protection domains are defined only when the corresponding address translation (instruction or data) is enabled (MSR[IR] or MSR[DR] = ’1’).

The register settings for alignment exceptions are shown in *Table 6-14*.

Table 6-14. Alignment Exception—Register Settings

Register	Setting Description	
SRR0	Set to the effective address of the instruction that caused the exception.	
SRR1	0	Loaded with equivalent bit from the MSR
	33–36	Cleared
	42–47	Cleared
	48–55	Loaded with equivalent bits from the MSR
	57–59	Loaded with equivalent bits from the MSR
	62–63	Loaded with equivalent bits from the MSR
	Note: Depending on the implementation, additional bits in the MSR may be copied to SRR1.	
MSR	SF	1
	POW	0
	ILE	—
	EE	0
	PR	0
	FP	0
	ME	—
	FE0	0
	SE	0
	BE	0
	FE1	0
	DR	0
	PMM	0
	RI	0
	LE	Set to value of ILE
DSISR	0–11	Cleared
	12–13	For 64-bit instructions that use immediate addressing—set to bits [30–31] if DS-form. Otherwise cleared.
	14	Cleared
	15–16	For instructions that use register indirect with index addressing (X-form)—set to bits [29–30] of the instruction encoding. For instructions that use register indirect with immediate index addressing (D or DS-form)—cleared
	17	For instructions that use register indirect with index addressing (X-form)—set to bit [25] of the instruction encoding. For instructions that use register indirect with immediate index addressing (D or DS-form)— set to bit [5] of the instruction encoding.
	18–21	For instructions that use register indirect with index addressing (X-form)—set to bits [21–24] of the instruction encoding. For instructions that use register indirect with immediate index addressing (D or DS-form)—set to bits [1–4] of the instruction encoding.
	22–26	Set to bits [6–10] (identifying either the source or destination) of the instruction encoding. Undefined for **dcbz**.
	27–31	Set to bits [11–15] of the instruction encoding (**r**A) for update-form instructions Set to either bits [11–15] of the instruction encoding or to any register number not in the range of registers loaded by a valid form instruction for **lmw**, **lswi**, and **lswx** instructions. Otherwise undefined.
DAR	Set to the EA of the data access as computed by the instruction causing the alignment exception. **Note:** If a 64-bit processor is running in 32-bit mode, the 32 high-order bits are cleared.	

For load or store instructions that use register indirect with index addressing, the DSISR can be set to the same value that would have resulted if the corresponding instruction uses register indirect with immediate index addressing had caused the exception. Similarly, for load or store instructions that use register indirect with immediate index addressing, DSISR can hold a value that would have resulted from an instruction that uses register indirect with index addressing. For example, a misaligned **lwarx** instruction that crosses a protection boundary would normally cause the DSISR to be set to the following binary value:

    000000000000 00 0 01 0 0101 ttttt ?????

The value ttttt refers to the destination and ????? indicates undefined bits.

However, this register may be set as if the instruction were **lwa**, as follows:

    000000000000 10 0 00 0 1101 ttttt ?????

If there is no corresponding instruction (such as for the **lwaux** instruction), no alternative value can be specified.

The instruction pairs that can use the same DSISR values are as follows:

lhz / **lhzx**	**lhzu** / **lhzux**	**lha** / **lhax**	**lhau** / **lhaux**
lwz / **lwzx**	**lwzu** / **lwzux**	**lwa** / **lwax**	
ld / **ldx**	**ldu** / **ldux**		
sth / **sthx**	**sthu** / **sthux**	**stw** / **stwx**	**stwu** / **stwux**
std / **stdx**	**stdu** / **stdux**		
lfs / **lfsx**	**lfsu** / **lfsux**	**lfd** / **lfdx**	**lfdu** / **lfdux**
stfs / **stfsx**	**stfsu** / **stfsux**	**stfd** / **stfdx**	**stfdu** / **stfdux**

The architecture does not support the use of a misaligned effective address by load/store with reservation instructions or by the **eciwx** and **ecowx** instructions. If one of these instructions specifies a misaligned effective address, the exception handler should not emulate the instruction, but should treat the occurrence as a programming error.

When an alignment exception is taken, instruction execution resumes at effective address 0x0000_0000_0000_0600.

6.4.8.1 Integer Alignment Exceptions

Operations that are not naturally aligned may suffer performance degradation, depending on the processor design, the type of operation, the boundaries crossed, and the mode that the processor is in during execution. More specifically, these operations may either cause an alignment exception or they may cause the processor to break the memory access into multiple, smaller accesses with respect to the cache and the memory subsystem.

Page Address Translation Access Considerations

A page address translation access occurs when MSR[DR] is set.

Note: A **dcbz** instruction causes an alignment exception if the access is to a page with the write-through (W) or cache-inhibit (I) bit set.

Misaligned memory accesses that do not cause an alignment exception may not perform as well as an aligned access of the same type. The resulting performance degradation due to misaligned accesses depends on how well each individual access behaves with respect to the memory hierarchy.

Particular details regarding page address translation is implementation-dependent; the reader should consult the user's manual for the appropriate processor for more information.

6.4.8.2 Little-Endian Mode Alignment Exceptions

The OEA allows implementations to take alignment exceptions on misaligned accesses (as described in *Section 3.1.4 PowerPC Byte Ordering*) in little-endian mode but does not require them to do so. Some implementations may perform some misaligned accesses without taking an alignment exception.

6.4.8.3 Interpretation of the DSISR as Set by an Alignment Exception

For most alignment exceptions, an exception handler may be designed to emulate the instruction that causes the exception. To do this, the handler requires the following characteristics of the instruction:

- Load or store
- Length (halfword, word, or doubleword)
- String, multiple, or normal load/store
- Integer or floating-point
- Whether the instruction performs update
- Whether the instruction performs byte reversal
- Whether it is a **dcbz** instruction

The PowerPC Architecture provides this information implicitly, by setting opcode bits in the DSISR that identify the excepting instruction type. The exception handler does not need to load the excepting instruction from memory. The mapping for all exception possibilities is unique except for the few exceptions discussed below.

Table 6-15 shows the inverse mapping—how the DSISR bits identify the instruction that caused the exception.

The alignment exception handler cannot distinguish a floating-point load or store that causes an exception because it is misaligned, However, this does not matter; in either case it is emulated with integer instructions. Floating-point instructions are distinguished from integer instructions because different register files must be accessed while emulating each class. Bits [15-21] of the DSISR are used to identify whether the instruction is integer or floating-point.

Table 6-15. DSISR(15–21) Settings to Determine Misaligned Instruction

DSISR[15–21]	Instruction	DSISR[15–21]	Instruction
00 0 0000	**lwarx**, **lwz**, special cases[1]	01 1 0010	**stdux**
00 0 0010	**ldarx**	01 1 0101	**lwaux**
00 0 0010	**stw**	10 0 0010	**stwcx.**
00 0 0100	**lhz**	10 0 0011	**stdcx.**
00 0 0101	**lha**	10 0 1000	**lwbrx**
00 0 0110	**sth**	10 0 1010	**stwbrx**
00 0 0111	**lmw**	10 0 1100	**lhbrx**
00 0 1000	**lfs**	10 0 1110	**sthbrx**
00 0 1001	**lfd**	10 1 0100	**eciwx**
00 0 1010	**stfs**	10 1 0110	**ecowx**
00 0 1011	**stfd**	10 1 1111	**dcbz**
00 0 1101	**ld**, **ldu**, **lwa**[2]	11 0 0000	**lwzx**
00 0 1111	**std**, **stdu**[2]	11 0 0010	**stwx**
00 1 0000	**lwzu**	11 0 0100	**lhzx**
00 1 0010	**stwu**	11 0 0101	**lhax**
00 1 0100	**lhzu**	11 0 0110	**sthx**
00 1 0101	**lhau**	11 0 1000	**lfsx**
00 1 0110	**sthu**	11 0 1001	**lfdx**
00 1 0111	**stmw**	11 0 1010	**stfsx**
00 1 1000	**lfsu**	11 0 1011	**stfdx**
00 1 1001	**lfdu**	11 0 1111	**stfiwx**
00 1 1010	**stfsu**	11 1 0000	**lwzux**
00 1 1011	**stfdu**	11 1 0010	**stwux**
01 0 0000	**ldx**	11 1 0100	**lhzux**
01 0 0010	**stdx**	11 1 0101	**lhaux**
01 0 0101	**lwax**	11 1 0110	**sthux**
01 0 1000	**lswx**	11 1 1000	**lfsux**
01 0 1001	**lswi**	11 1 1001	**lfdux**
01 0 1010	**stswx**	11 1 1010	**stfsux**
01 0 1011	**stswi**	11 1 1011	**stfdux**
01 1 0000	**ldux**		

1. The instructions **lwz** and **lwarx** give the same DSISR bits (all zero). But if **lwarx** causes an alignment exception, it is an invalid form, so it need not be emulated in any precise way. It is adequate for the alignment exception handler to simply emulate the instruction as if it were an **lwz**. It is important that the emulator use the address in the DAR, rather than computing it from **r**A/**r**B/D, because **lwz** and **lwarx** use different addressing modes.

   If opcode 0 (“illegal or reserved”) can cause an alignment exception, it will be indistinguishable to the exception handler from **lwarx** and **lwz**.
2. These instructions are distinguished by DSISR[12–13], which are not shown in this table.

6.4.9 Program Exception (0x00700)

A program exception occurs when no higher priority exception exists and one or more of the following exception conditions, which correspond to bit settings in SRR1, occur during execution of an instruction:

- System IEEE floating-point enabled exception—A system IEEE floating-point enabled exception can be generated when FPSCR[FEX] is set and either (or both) of the MSR[FE0] or MSR[FE1] bits is set.

  FPSCR[FEX] is set by the execution of a floating-point instruction that causes an enabled exception or by the execution of a “move to FPSCR” type instruction that sets an exception bit when its corresponding enable bit is set. Floating-point exceptions are described in *Section 3.3.6 Floating-Point Program Exceptions*.
- Illegal instruction—An illegal instruction program exception is generated when execution of an instruction is attempted with an illegal opcode or illegal combination of opcode and extended opcode fields (these include PowerPC instructions not implemented in the processor), or when execution of an optional or a reserved instruction not provided in the processor is attempted.

  Implementations are permitted to generate an illegal instruction program exception when encountering the following instructions. If an illegal instruction exception is not generated, then the alternative is shown in parenthesis.
  - An instruction corresponds to an invalid class (the results may be boundedly undefined)
  - An **lswx** instruction for which **r**A or **r**B is in the range of registers to be loaded (may cause results that are boundedly undefined)
  - An **mtspr** or **mfspr** instruction with an SPR field that does not contain one of the defined values, or an **mftb** instruction with a TBR field that does not contain one of the defined values
- Privileged instruction—A privileged instruction type program exception is generated when the execution of a privileged instruction is attempted and the processor is operating in user mode (MSR[PR] is set). It is also generated for **mtspr** or **mfspr** instructions that have an invalid SPR field that contain one of the defined values having spr[0] = ’1’ and if MSR[PR] = ’1’. Some implementations may also generate a privileged instruction program exception if a specified SPR field (for a move to/from SPR instruction) is not defined for a particular implementation, but spr[0] = ’1’; in this case, the implementation may cause either a privileged instruction program exception, or an illegal instruction program exception may occur instead.
- Trap—A trap program exception is generated when any of the conditions specified in a trap instruction is met. Trap instructions are described in *Section 4.2.4.6 Trap Instructions*.

The register settings when a program exception is taken are shown in *Table 6-16*.

Table 6-16. Program Exception—Register Settings

Register	Setting Description			
SRR0	The contents of SRR0 differ according to the following situations: • For all program exceptions except floating-point enabled exceptions when operating in imprecise mode (MSR[FE0-FE1] = ’10’ or ‘01’ respectively), SRR0 contains the effective address of the instruction that caused the exception. • When the processor is in floating-point imprecise mode, SRR0 may contain the effective address of the excepting instruction or that of a subsequent unexecuted instruction. If the subsequent instruction is **sync, ptesync**, or **isync**, SRR0 points not more than four bytes beyond the **sync, ptesync**, or **isync** instruction. • If FPSCR[FEX] = ’1’, but IEEE floating-point enabled exceptions are disabled (MSR[FE0] = MSR[FE1] = ’0’), the program exception occurs before the next synchronizing event if an instruction alters those bits (thus enabling the program exception). When this occurs, SRR0 points to the instruction that would have executed next and not to the instruction that modified MSR.			
SRR1	0	Loaded with equivalent bit from the MSR		
	33–36	Cleared		
	42	Cleared		
	43	Set for an IEEE floating-point enabled program exception; otherwise cleared.		
	44	Set for an illegal instruction program exception; otherwise cleared.		
	45	Set for a privileged instruction program exception; otherwise cleared.		
	46	Set for a trap program exception; otherwise cleared.		
	47	Cleared if SRR0 contains the address of the instruction causing the exception, and set if SRR0 contains the address of a subsequent instruction.		
	48–55	Loaded with equivalent bits from the MSR		
	57–59	Loaded with equivalent bits from the MSR		
	62–63	Loaded with equivalent bits from the MSR		
	Note: Only one of bits [43:46] can be set to 1. **Note:** Depending on the implementation, additional bits in the MSR may be copied to SRR1.			
MSR	SF 1	PR 0		DR 0
	POW 0	FP 0	BE 0	PMM 0
	ILE —	ME —	FE1 0	RI 0
	EE 0	FE0 0		LE Set to value of ILE

When a program exception is taken, instruction execution resumes at effective address 0x000_0000_0000_0700.

6.4.10 Floating-Point Unavailable Exception (0x00800)

A floating-point unavailable exception occurs when no higher priority exception exists, an attempt is made to execute a floating-point instruction (including floating-point load, store, or move instructions), and the floating-point available bit in the MSR is cleared, (MSR[FP] = ’0’).

The register settings for floating-point unavailable exceptions are shown in *Table 6-17*.

Table 6-17. Floating-Point Unavailable Exception—Register Settings

Register	Setting Description							
SRR0	Set to the effective address of the instruction that caused the exception.							
SRR1	0 33–36 42–47 48–55 57–59 62–63		Loaded with equivalent bit from the MSR Cleared Cleared Loaded with equivalent bits from the MSR Loaded with equivalent bits from the MSR Loaded with equivalent bits from the MSR					
	Note that depending on the implementation, additional bits in the MSR may be copied to SRR1.							
MSR	SF POW ILE EE	1 0 — 0	PR FP ME FE0	0 0 — 0	SE BE FE1	0 0 0	DR PMM RI LE	0 0 0 Set to value of ILE

When a floating-point unavailable exception is taken, instruction execution resumes at effective address 0x0000_0000_0000_0800.

6.4.11 Decrementer Exception (0x00900)

A decrementer exception occurs when no higher priority exception exists, a decrementer exception condition occurs (for example, the decrementer register has completed decrementing), and MSR[EE] = ’1’. The decrementer register counts down, causing an exception request when it passes through zero. A decrementer exception request remains pending until the decrementer exception is taken and then it is cancelled. The decrementer implementation meets the following requirements:

- The counters for the decrementer and the time-base counter are driven by the same fundamental time base.
- Loading a GPR from the decrementer does not affect the decrementer.
- Storing a GPR value to the decrementer replaces the value in the decrementer with the value in the GPR.
- Whenever bit [0] of the decrementer changes from ‘0’ to ‘1’, a decrementer exception request is signaled. If multiple decrementer exception requests are received before the first can be reported, only one exception is reported. The occurrence of a decrementer exception cancels the request.
- If the decrementer is altered by software and if bit [0] is changed from ‘0’ to ‘1’, an exception request is signaled.

The register settings for the decrementer exception are shown in *Table 6-18*.

Table 6-18. Decrementer Exception—Register Settings

Register	Setting Description			
SRR0	Set to the effective address of the instruction that the processor would have attempted to execute next if no exception conditions were present.			
SRR1	0 33–36 42–47 48–55 57–59 62–63	Loaded with equivalent bit from the MSR Cleared Cleared Loaded with equivalent bits from the MSR Loaded with equivalent bits from the MSR Loaded with equivalent bits from the MSR		
	Note that depending on the implementation, additional bits in the MSR may be copied to SRR1.			
MSR	SF 1 POW 0 ILE — EE 0	PR 0 FP 0 ME — FE0 0	SE 0 BE 0 FE1 0	DR 0 PMM 0 RI 0 LE Set to value of ILE

When a decrementer exception is taken, instruction execution resumes at effective address 0x0000_0000_0000_0900.

6.4.12 System Call Exception (0x00C00)

A system call exception occurs when a System Call (**sc**) instruction is executed. The effective address of the instruction following the **sc** instruction is placed into SRR0. MSR bits are saved in SRR1, as shown in *Table 6-19*, and then a system call exception is generated.

The system call exception causes the next instruction to be fetched from effective address 0x0000_0000_0000_0C00. As with most other exceptions, this exception is context-synchronizing. Refer to *Context Synchronization* on page 208 for more information on the actions performed by a context-synchronizing operation. Register settings are shown in *Table 6-19*.

Table 6-19. System Call Exception—Register Settings

Register	Setting Description			
SRR0	Set to the effective address of the instruction following the System Call instruction			
SRR1	0 33–36 42–47 48–55 57–59 62–63	Loaded with equivalent bit from the MSR Cleared Cleared Loaded with equivalent bits from the MSR Loaded with equivalent bits from the MSR Loaded with equivalent bits from the MSR		
	Note: Depending on the implementation, additional bits in the MSR may be copied to SRR1.			
MSR	SF 1 POW 0 ILE — EE 0	PR 0 FP 0 ME — FE0 0	SE 0 BE 0 FE1 0	DR 0 PMM 0 RI 0 LE Set to value of ILE

When a system call exception is taken, instruction execution resumes at effective address 0x0000_0000_0000_0C00.

6.4.13 Trace Exception (0x00D00)

The trace exception is optional to the PowerPC Architecture, and specific information about how it is implemented can be found in user's manuals for individual processors.

The trace exception provides a means of tracing the flow of control of a program for debugging and performance analysis purposes. It is controlled by MSR bits [SE] and [BE] as follows:

- MSR[SE] = '1' and any instruction except **rfid** is successfully completed.
- MSR[BE] = '1': the processor generates a branch-type trace exception after completing the execution of a branch instruction, whether or not the branch is taken.

If this facility is implemented, a trace exception occurs when no higher priority exception exists and either of the conditions described above exist. The following are not traced:

- **rfid** instruction
- **sc**, and trap instructions that trap
- Other instructions that cause exceptions (other than trace exceptions)
- The first instruction of any exception handler
- Instructions that are emulated by software

MSR[SE, BE] are both cleared when the trace exception is taken. In the normal use of this function, MSR[SE, BE] are restored when the exception handler returns to the interrupted program using an **rfid** instruction.

Register settings for the trace mode are described in *Table 6-20*.

Table 6-20. Trace Exception—Register Settings

Register	Setting Description							
SRR0	Set to the effective address of the next instruction to be executed in the program for which the trace exception was generated.							
SRR1	0 33–36 42–47 48–55 57–59 62–63	Loaded with equivalent bit from the MSR Cleared Cleared Loaded with equivalent bits from the MSR Loaded with equivalent bits from the MSR Loaded with equivalent bits from the MSR						
	Note that depending on the implementation, additional bits in the MSR may be copied to SRR1.							
MSR	SF POW ILE EE	1 0 — 0	PR FP ME FE0	0 0 — 0	SE BE FE1	0 0 0	DR PMM RI LE	0 0 0 Set to value of ILE

When a trace exception is taken, instruction execution resumes at effective address 0x0000_0000_0000_0D00.

6.4.14 Performance Monitor Exception (0x00F00)

The performance monitor exception is part of the optional performance monitor facility. If the performance monitor facility is not implemented or does not use this interrupt, the corresponding interrupt vector is treated as reserved.

A performance monitor facility provides a means of collecting information about program and system performance. The resources (for example, SPR numbers) that a performance monitor facility may use are identified elsewhere in this manual. All other aspects of any performance monitor facility are implementation-dependent.

When a performance monitor exception is taken, instruction execution resumes at effective address 0x0000_0000_0000_0F00.

7. Memory Management

This chapter describes the memory management unit (MMU) specifications provided by the PowerPC operating environment architecture (OEA) for PowerPC processors. The primary function of the MMU in a PowerPC processor is to translate logical (effective) addresses to physical addresses (referred to as real addresses in the architecture specification) for memory accesses and I/O accesses (most I/O accesses are assumed to be memory-mapped). In addition, the MMU provides various levels of access protection on a segment, block, or page basis.

Note: There are many aspects of memory management that are implementation-specific. This chapter describes the conceptual model of a PowerPC MMU; however, PowerPC processors may differ in the specific hardware used to implement the MMU model of the OEA, depending on the many design trade-offs inherent in each implementation.

Two general types of accesses generated by PowerPC processors require address translation—instruction accesses, and data accesses to memory generated by load and store instructions. In addition, the addresses specified by cache instructions and the optional external control instructions also require translation. Generally, the address translation mechanism is defined in terms of segment descriptors and page tables used by PowerPC processors to locate the effective to physical address mapping for instruction and data accesses. The segment information translates the effective address to an interim virtual address, and the page table information translates the virtual address to a physical address.

The definition of the segment and page table data structures provides significant flexibility for the implementation of performance enhancement features in a wide range of processors. Therefore, the performance enhancements used to store the segment or page table information on-chip vary from implementation to implementation.

Translation lookaside buffers (TLBs) are commonly implemented in PowerPC processors to keep recently-used page address translations on-chip. Although their exact characteristics are not specified in the OEA, the general concepts that are pertinent to the system software are described.

Note: In contrast to earlier versions of the architecture, an implementation is required to have an SLB, but explicit representation of a segment table in memory is not required. The SLB is software managed, and so memory management software can maintain an explicit segment table in memory, or can implement an implicit segment table by generating new SLB entries as needed. References to the segment table in this chapter do not presume the explicit table in memory that was specified in previous versions of the architecture.

The segment information, used to generate the interim virtual addresses, is stored as segment descriptors. These descriptors may reside in segment table entries (STEs) in memory. In much the same way that TLBs cache recently-used page address translations, 64-bit processors may contain segment lookaside buffers (SLBs) on-chip that cache recently-used segment table entries. Although the exact characteristics of SLBs are not specified, there is general information pertinent to those implementations that provide SLBs.

Temporary 64-Bit Bridge

The OEA defines an additional, optional bridge to the 64-bit architecture that may make it easier for 32-bit operating systems to migrate to 64-bit processors. The 64-bit bridge retains certain aspects of the 32-bit architecture that otherwise are not supported, and in some cases not permitted, by the 64-bit version of the architecture. In processors that implement this bridge, segment descriptors are implemented by using 16 SLB entries to emulate segment registers, which, like those defined for the 32-bit architecture, divide the 32-bit memory space (4 Gbytes) into sixteen 256-Mbyte segments. These segment descriptors however use the format of the segment table entries as defined in the 64-bit architecture and are maintained in SLBs rather than in architecture-defined segment registers.

The MMU, together with the exception processing mechanism, provides the necessary support for the operating system to implement a paged virtual memory environment and for enforcing protection of designated memory areas. Exception processing is described in *Chapter 6, Exceptions*. *Section 2.3.1 Machine State Register (MSR)* describes the MSR, which controls some of the critical functionality of the MMU.

7.1 MMU Features

The memory management specification of the PowerPC OEA includes models for both 32 and 64-bit implementations. The MMU of a 64-bit PowerPC processor provides 2^{64} bytes of effective address space accessible to supervisor and user programs with support for two page sizes; a 4-Kbyte page size (2^{12}) and a large page whose size is implementation dependent (2^p where $13 \leq p \leq 28$). The MMU of 64-bit PowerPC processors uses an interim virtual address (between 65 and 80 bits) and hashed page tables in the generation of physical addresses that are ≤ 62 bits in length.

Table 7-1 summarizes the features of PowerPC MMUs for 64-bit implementations.

Table 7-1. MMU Features Summary

Feature Category	64-Bit Implementations	
	Conventional	Temporary 64-Bit Bridge
Address ranges	2^{64} bytes of effective address	232 bytes of effective address
	2^n where $65 \le n \le 80$ bytes of virtual address	252 bytes of virtual address
	$\le 2^m$ ($m \le 62$) bytes of physical address	< 232 bytes of physical address
Page size	4 Kbytes Some large page sizes (2^p where $13 \le p \le 28$)	4 Kbytes
Segment size	256 Mbytes	Same
Block address translation	Not applicable	Not applicable
	Not applicable	Not applicable
Memory protection	Segments selectable as no-execute	Same
	Pages selectable as user/supervisor and read-only	Same
	Blocks selectable as user/supervisor and read-only	Same
Page history	Referenced and changed bits defined and maintained	Same
Page address translation	Translations stored as PTEs in hashed page tables in memory	Same
	Page table size determined by size programmed into SDR1 register	Same
TLBs	Instructions for maintaining optional TLBs	Same
Segment descriptors	Stored as STEs (explicit or implicit segment tables)	Stored in 16 SLB entries in the same format as the STEs defined for 64-bit implementations.
	Instructions for maintaining SLBs	16 SLB entries are required to emulate the segment registers defined for 32-bit addressing. The **slbie** and **slbia** instructions should not be executed when using the 64-bit bridge.

Note: This chapter describes address translation mechanisms from the perspective of the programming model. As such, it describes the structure of the page and segment tables, the MMU conditions that cause exceptions, the instructions provided for programming the MMU, and the MMU registers. The hardware implementation details of a particular MMU (including whether the hardware automatically performs a page table search in memory) are not contained in the architectural definition of PowerPC processors and are invisible to the PowerPC programming model; therefore, they are not described in this manual. In the case that some of the OEA model is implemented with some software assist mechanism, this software should be contained in the area of memory reserved for implementation-specific use and should not be visible to the operating system.

Temporary 64-Bit Bridge

In addition to the features described above, the OEA provides optional features that facilitate the migration of operating systems from 32-bit processor designs to 64-bit processors. These features, which can be implemented in part or in whole, include the following:

- Support for several 32-bit instructions that are otherwise defined as illegal in 64-bit processors. These include the following—**mtsr**, **mtsrin**, **mfsr**, **mfsrin**.
- The **mtmsr** instruction, which is otherwise illegal in the 64-bit architecture may optionally be implemented in 64-bit bridge implementations.
- The bridge defines the optional bit ASR[V] (bit [63]) may be implemented to indicate whether ASR[STABORG] holds a valid physical base address for the segment table.

To determine whether a processor implements any or all of the bridge features, consult the user's manual for that processor.

7.2 MMU Overview

The PowerPC MMU and exception models support demand-paged virtual memory. Virtual memory management permits execution of programs larger than the size of physical memory; the term demand paged implies that individual pages are loaded into physical memory from backing storage only as they are accessed by an executing program.

The memory management model includes the concept of a virtual address that is not only larger than that of the maximum physical memory allowed, but a virtual address space that is also larger than the effective address space. Effective addresses generated by 64-bit implementations are 64 bits wide. In the address translation process, the processor converts an effective address to a virtual address between 65 and 80 bits, as per the information in the selected descriptor. Then the address is translated back to a physical address the size (or less) of the effective address.

Implementations for 64-bit designs have the option of supporting virtual address in the range of 65 to 80 bits. The remainder of this chapter describes the virtual address for 64-bit processors as consisting of $65 \le n \le 80$ bits. For implementations that support a virtual address less than 80 bits, the high-order bits of the 80-bit virtual address are assumed to be zero.

Note: For 64-bit implementations the physical address space size is 2^m bytes, where $m \le 62$. The value of "*m*" is implementation dependent. When used to address memory, the high-order "62-m" bits of the 62-bit physical address must be zeros.

The operating system manages the system's physical memory resources. Consequently, the operating system initializes the MMU registers (address space register (ASR) and SDR1 register) and sets up page tables and segment tables in memory appropriately. The MMU then assists the operating system by managing page status and optionally caching the recently-used address translation information on-chip for quick access.

Effective address spaces are divided into 256-Mbyte regions called segments for virtual addressing. Segments that correspond to virtual memory can be further subdivided into pages (4KB or large page size whose size is implementation dependent). For programs using virtual addressing, only the most recently used 4-Kbyte (or large) pages need be resident in memory.

For each page, the operating system creates an address descriptor (page table entry (PTE)). The MMU then uses these descriptors to generate the physical address, the protection information, and other access control information each time an address within the page is accessed. Address descriptors for the pages reside in tables (as PTEs) in memory and can be cached in the TLBs.

This section provides an overview of the high-level organization and operational concepts of the MMU in PowerPC processors, and a summary of all MMU control registers. For more information about the MSR, see *Section 2.3.1 Machine State Register (MSR)*. *Section 7.5.1.1 SDR1 Register Definition* describes the SDR1.

7.2.1 Memory Addressing

A program references memory using the effective (logical) address computed by the processor when it executes a load, store, branch, or cache instruction, and when it fetches the next instruction. The effective address is translated to a physical (real) address according to the procedures described throughout this chapter. The memory subsystem uses the physical address for the access.

7.2.1.1 Effective Addresses in 32-Bit Mode

In addition to the 64 and 32-bit memory management models defined by the OEA, the PowerPC Architecture also defines a 32-bit mode of operation for 64-bit implementations. In this 32-bit mode (MSR[SF] = 0), the 64-bit effective address is first calculated as usual, and then the high-order 32 bits of the effective address are treated as zero for the purposes of addressing memory. This occurs for both instruction and data accesses, and occurs independently from the setting of the MSR[IR] and MSR[DR] bits that enable instruction and data address translation, respectively. The truncation of the effective address is the only way in which memory accesses are affected by the 32-bit mode of operation.

Temporary 64-Bit Bridge

Some 64-bit processors implement optional features that simplify the conversion of an operating system from the 32-bit to the 64-bit portion of the architecture. This architecturally-defined bridge allows an operating system to use 16 on-chip SLB entries in the same manner that 32-bit implementations use the segment registers, which are otherwise not supported in the 64-bit architecture. These bridge features are available if the ASR[V] bit is implemented, and they are enabled when both ASR[V] and MSR[SF] are cleared.

For a complete discussion of effective address calculation, see *Section 4.1.4.2 Effective Address Calculation*.

7.2.1.2 Predefined Physical Memory Locations

There are four areas of the physical memory map that have predefined uses. Except for the first 256 bytes, which are reserved for software use, the physical (real) page beginning at physical address 0x0000_0000_0000_0000 is used for exception vectors. The two contiguous real pages beginning at real address 0x0000_0000_0000_1000 are reserved for implementation-specific purposes. A contiguous sequence of real pages beginning at the physical address specified by the SDR1 contains the Page Table. These predefined memory areas are summarized in *Table 7-2*. Refer to *Chapter 6, Exceptions* for more detailed information on the assignment of the exception vector offsets.

Table 7-2. Predefined Physical Memory Locations

Memory Area	Physical Address Range	Predefined Use
1	0x0_0000 – 0x0_00FF	Operating system
2	0x0_0100 – 0x0_0FFF	Exception vectors
3	0x0_1000 – 0x0_2FFF	Implementation-specific
4	Software-specified—contiguous sequence of physical pages Software-specified—single physical page	Page table

7.2.2 MMU Organization

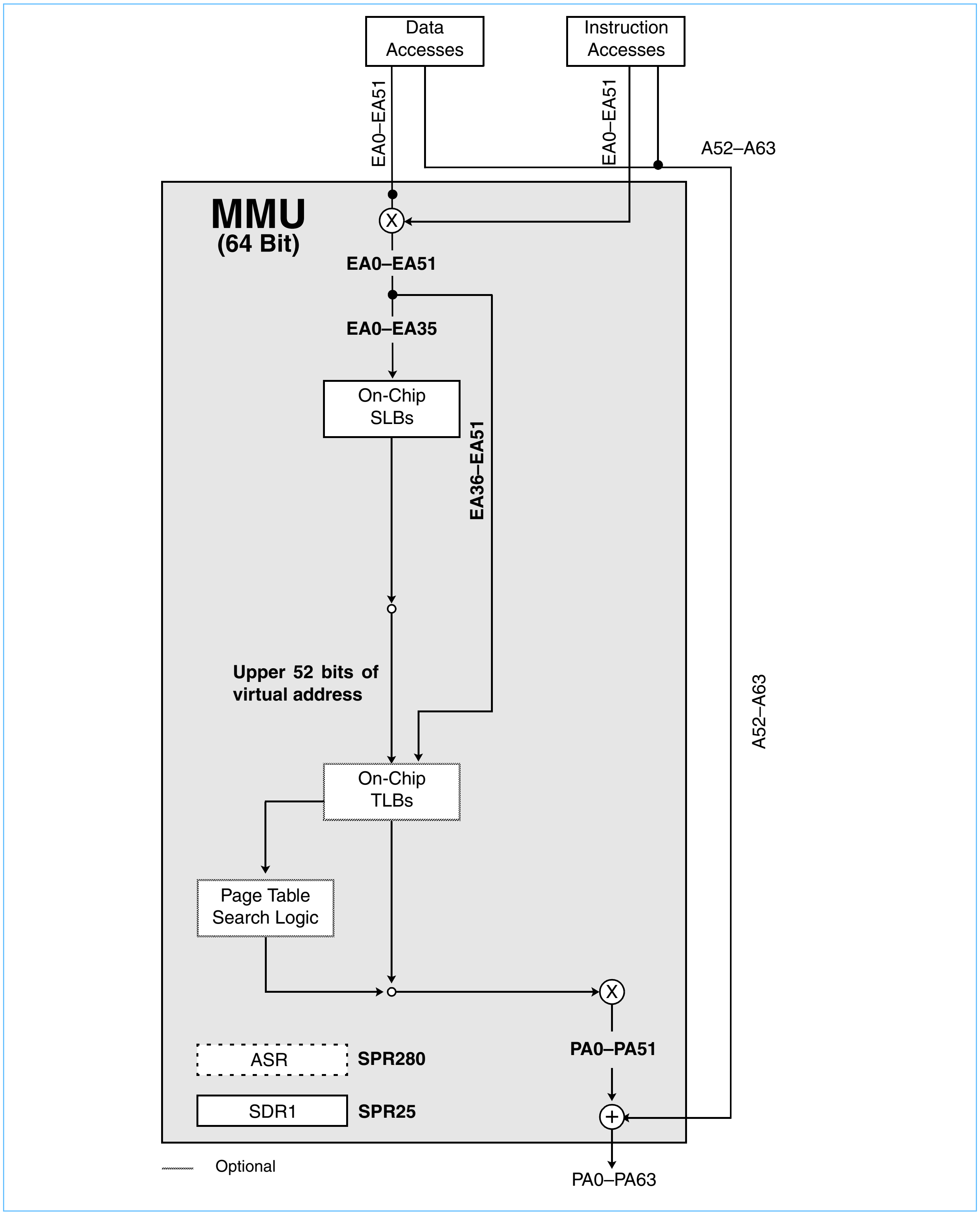
Figure 7-1 shows the conceptual organization of the MMU; note that it does not describe the specific hardware used to implement the memory management function for a particular processor, and other hardware features (invisible to the system software) not depicted in the figure may be implemented. For example, the memory management function can be implemented with parallel MMUs that translate addresses for instruction and data accesses independently.

The instruction addresses shown in *Figure 7-1* are generated by the processor for sequential instruction fetches and addresses that correspond to a change of program flow. Memory addresses are generated by load and store instructions, by cache instructions, and by the optional external control instructions.

As shown in *Figure 7-1*, for 4KB pages, bits EA0-EA51 are translated; for a large page translation, bits EA0-EAx are translated, where x=63-p if the size of large pages is 2^p. The lower-order address bits are untranslated and therefore identical for both effective and physical addresses. After translating the address, the MMU passes the resulting 64-bit physical address to the memory subsystem.

In addition to the higher-order address bits, the MMU automatically keeps an indicator of whether each access was generated as an instruction or data access and a supervisor/user indicator that reflects the state of the MSR[PR] bit when the effective address was generated. In addition, for data accesses, there is an indicator of whether the access is for a load or a store operation. This information is then used by the MMU to appropriately direct the address translation and to enforce the protection hierarchy programmed by the operating system. See *Section 2.3.1 Machine State Register (MSR)* for more information about the MSR.

Figure 7-1. MMU Conceptual Block Diagram



As shown in *Figure 7-1*, processors optionally implement on-chip translation lookaside buffers (TLBs) and optionally support the automatic search of the page tables for page table entries (PTEs).

The address space register (ASR) can be used to define the physical address of the base of the segment table in memory, if it exits. The architecture does not require that such a table be built. Instead, segment descriptors can be generated as needed by memory management software, and placed in the on-chip SLB.

Temporary 64-Bit Bridge

Processors that implement the 64-bit bridge implement segment descriptors as a table of 16 segment table entries.

7.2.3 Address Translation Mechanisms

PowerPC processors support the following two types of address translation:

- Page address translation—translates the page frame address for a 4-Kbyte or large page size
- Real addressing mode—when address translation is disabled, the physical address is identical to the effective address.

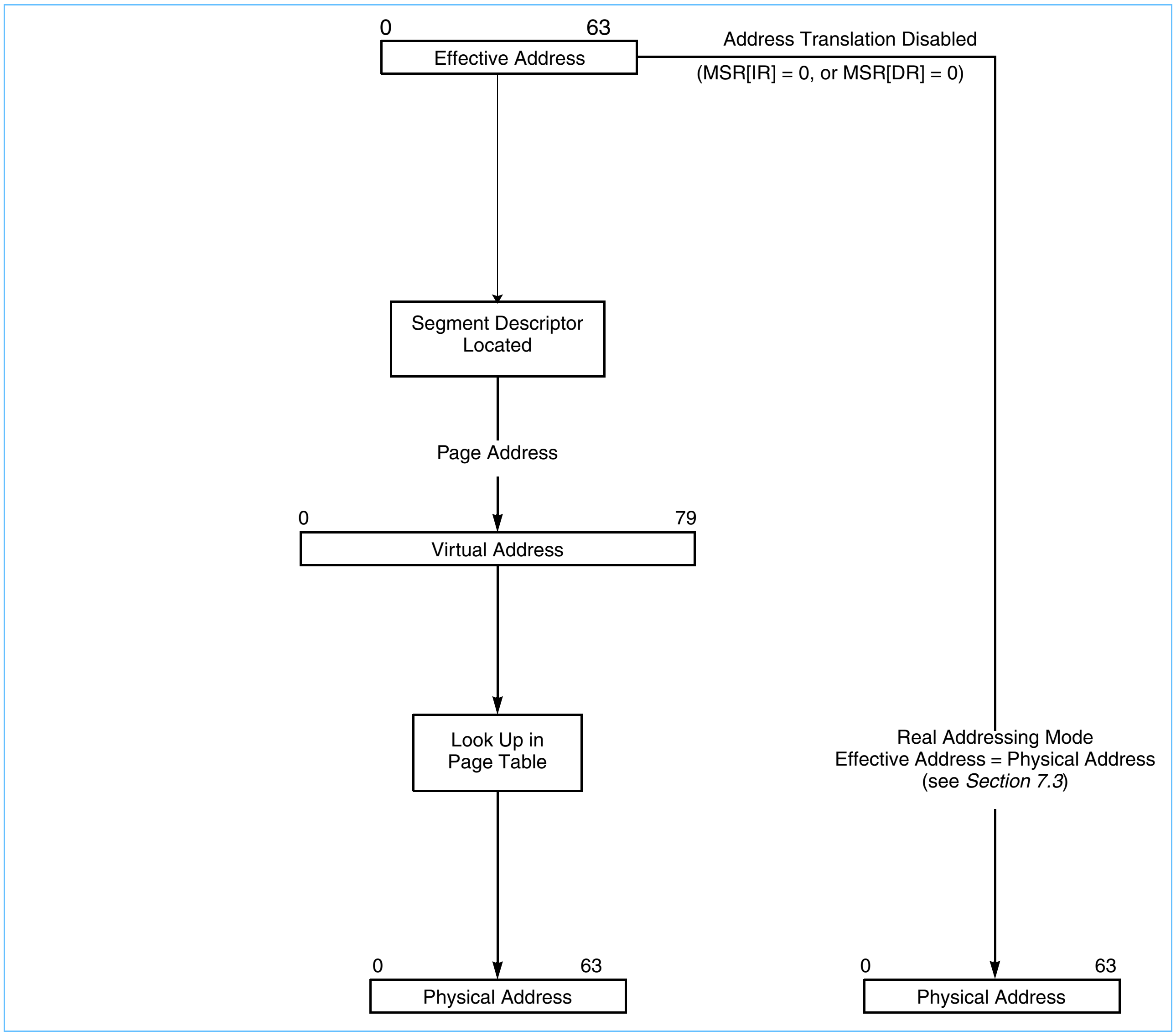
Figure 7-2 shows the address translation mechanisms provided by the MMU. The segment descriptors shown in the figure controls the page address translation mechanism. When an access uses the page address translation, the appropriate segment descriptor is required. In 64-bit implementations, the segment descriptor is located via a search of the segment table in memory for the appropriate segment table entry (STE), if an explicit table is used, and is otherwise generated by the operating system.

Temporary 64-Bit Bridge

Processors that implement the 64-bit bridge divide the 32-bit address space into sixteen 256-Mbyte segments defined by a table of 16 STEs maintained in 16 SLB entries.

For memory accesses translated by a segment descriptor, the interim virtual address is generated using the information in the segment descriptor. Page address translation corresponds to the conversion of this virtual address into the 64-bit physical address used by the memory subsystem. In some cases, the physical address for the page resides in an on-chip TLB and is available for quick access. However, if the page address translation misses in a TLB, the MMU searches the page table in memory (using the virtual address information and a hashing function) to locate the required physical address. Some implementations may have dedicated hardware to perform the page table search automatically, while others may define an exception handler routine that searches the page table with software.

Figure 7-2. Address Translation Types



Temporary 64-Bit Bridge

Figure 7-2 shows address sizes for a 64-bit processor operating in 64-bit mode. If the 64-bit bridge is enabled (ASR[V] is cleared), only the 32-bit address space is available and only 52 bits of the virtual address are used. However, the bridge supports cross-memory operations that permit an operating system to establish addressability to an address space, to copy data to it from another address space, and then to destroy the new addressability, without altering the segment table.

Real addressing mode address translation occurs when address translation is disabled; in this case, the physical address generated is identical to the effective address. Instruction and data address translation is enabled with the MSR[IR] and MSR[DR] bits, respectively. Thus, when the processor generates an access, and the corresponding address translation enable bit in the MSR (MSR[IR] for instruction accesses and MSR[DR] for data accesses) is cleared, the resulting physical address is identical to the effective address and all other translation mechanisms are ignored. See *Section 7.2.6.1 Real Addressing Mode Selection* for more information.

7.2.4 Memory Protection Facilities

In addition to the translation of effective addresses to physical addresses, the MMU provides access protection of supervisor areas from user access and can designate areas of memory as read-only, as well as no-execute. *Table 7-3* shows the eight protection options supported by the MMU for pages.

Table 7-3. Access Protection Options for Pages

Option	User Read		User Write	Supervisor Read		Supervisor Write
	I-Fetch	Data		I-Fetch	Data	
Supervisor-only	—	—	—	Y	Y	Y
Supervisor-only-no-execute	—	—	—	—	Y	Y
Supervisor-write-only	Y	Y	—	Y	Y	Y
Supervisor-write-only-no-execute	—	Y	—	—	Y	Y
Both user/supervisor	Y	Y	Y	Y	Y	Y
Both user/supervisor-no-execute	—	Y	Y	—	Y	Y
Both read-only	Y	Y	—	Y	Y	—
Both read-only-no-execute	—	Y	—	—	Y	—

Y	Access permitted
—	Protection violation

The operating system programs whether or not instruction fetches are allowed from an area of memory with the no-execute option provided in the segment descriptor. The remaining options are enforced based on a combination of information in the segment descriptor and the page table entry. Thus, the supervisor-only option allows only read and write operations generated while the processor is operating in supervisor mode (corresponding to MSR[PR] = ‘0’) to access the page. User accesses that map into a supervisor-only page cause an exception to be taken.

Note: Independent of the protection mechanisms, care must be taken when writing to instruction areas as coherency must be maintained with on-chip copies of instructions that may have been prefetched into a queue or an instruction cache. Refer to *Section 5.1.5.2 Instruction Cache Instructions* for more information on coherency within instruction areas.

As shown in the table, the supervisor-write-only option allows both user and supervisor accesses to read from the page, but only supervisor programs can write to that area. There is also an option that allows both supervisor and user programs read and write access (both user/supervisor option), and finally, there is an option to designate a page as read-only, both for user and supervisor programs (both read-only option).

A facility defined in the VEA and OEA allows pages to be designated as guarded, preventing out-of-order accesses that may cause undesired side effects. For example, areas of the memory map that are used to control I/O devices can be marked as guarded so that accesses (such as, instruction prefetches) do not occur unless they are explicitly required by the program. Refer to *Out-of-Order Accesses to Guarded Memory* on page 203, for a complete description of how accesses to guarded memory are restricted.

7.2.5 Page History Information

The MMU of PowerPC processors also defines referenced (R) and changed (C) bits in the page address translation mechanism that can be used as history information relevant to the usage of a page. The C-bit is used by the operating system to determine which pages have changed and must be written back to disk when new pages are replacing them in main memory. The R-bit is used to determine that a reference (for example a load instruction) has been made to a page and the operating system can use this information when trying to decide which page not to remove from memory. While these bits are initially allocated by the operating system into the page table, the architecture specifies that the R and C-bits are updated by the processor when a program executes a load (R) or store (C) to a page.

7.2.6 General Flow of MMU Address Translation

The following sections describe the general flow used by PowerPC processors to translate effective addresses to virtual and then physical addresses.

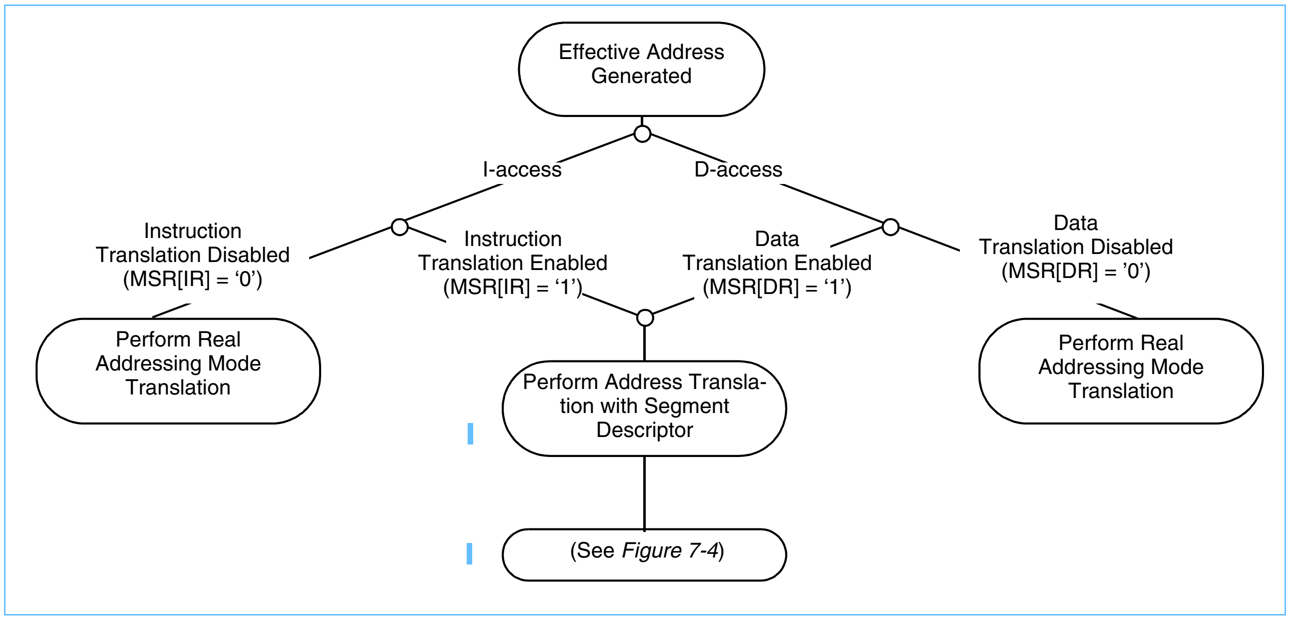
Note: Although there are references to the concept of an on-chip TLB, they may not be present in a particular hardware implementation for performance enhancement (and a particular implementation may have one or more TLBs). Thus, TLBs are shown here as optional and only the software ramifications of the existence of a TLB is discussed.

7.2.6.1 Real Addressing Mode Selection

When an instruction or data access is generated and the corresponding instruction or data translation is disabled (MSR[IR] = ‘0’ or MSR[DR] = ‘0’), real addressing mode translation is used (physical address equals effective address) and the access continues to the memory subsystem as described in *Section 7.3 Real Addressing Mode*.

Figure 7-3 shows the flow used by the MMU in determining whether to select real addressing mode or to use the segment descriptor to select page address translation.

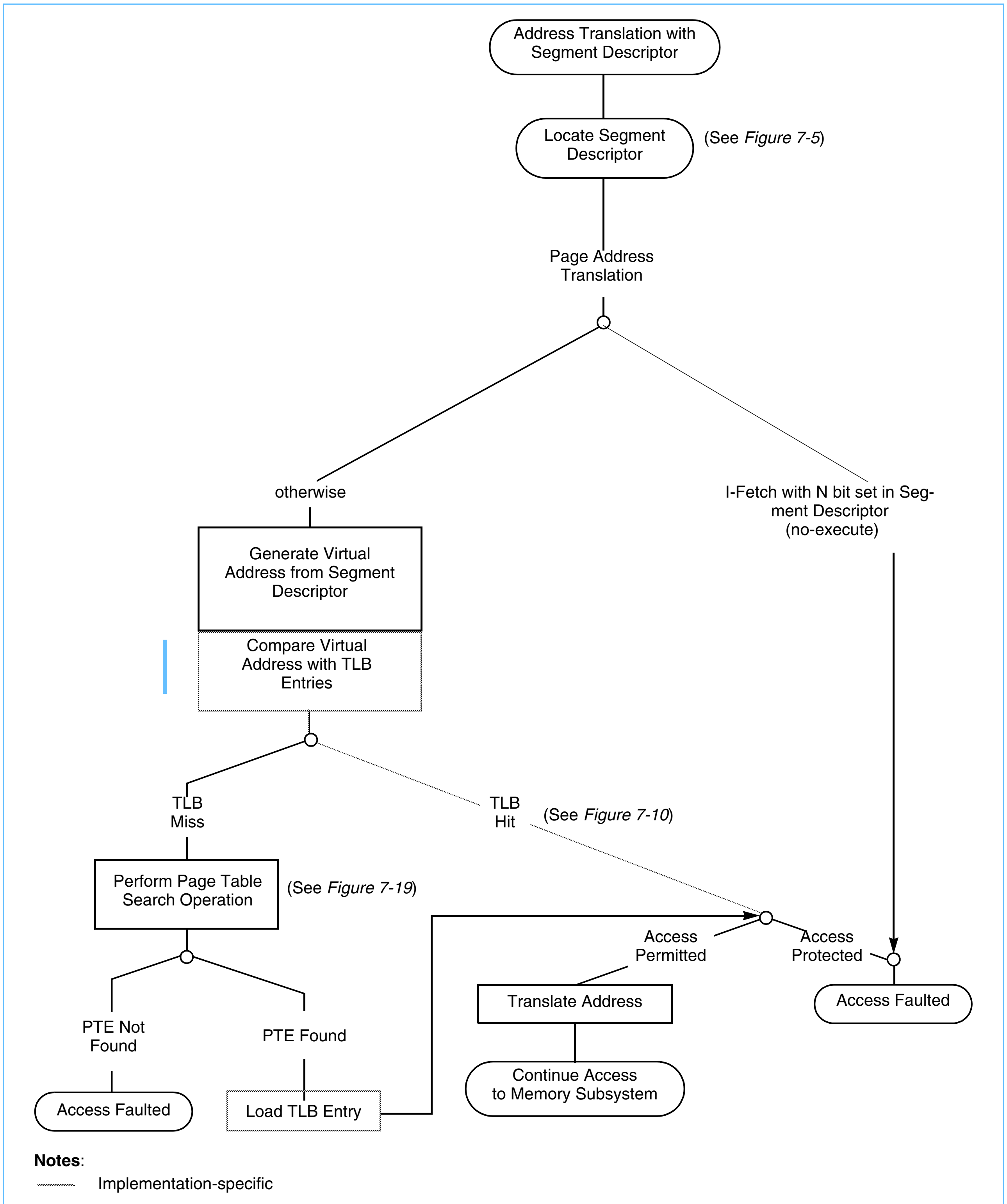
Figure 7-3. General Flow of Address Translation (Real Addressing Mode)


Effective Address Generated
I-access
D-access
Instruction Translation Disabled (MSR[IR] = ‘0’)
Instruction Translation Enabled (MSR[IR] = ‘1’)
Data Translation Enabled (MSR[DR] = ‘1’)
Data Translation Disabled (MSR[DR] = ’0’)
Perform Real Addressing Mode Translation
Perform Address Translation with Segment Descriptor
Perform Real Addressing Mode Translation
(See *Figure 7-4*)

7.2.6.2 Page Address Translation Selection

If address translation is enabled (real addressing mode translation not selected), then the segment descriptor must be located. *Figure 7-4* also shows the way in which the no-execute protection is enforced; if the N-bit in the segment descriptor is set and the access is an instruction fetch, the access is faulted.

Figure 7-4. General Flow of Page Address Translation

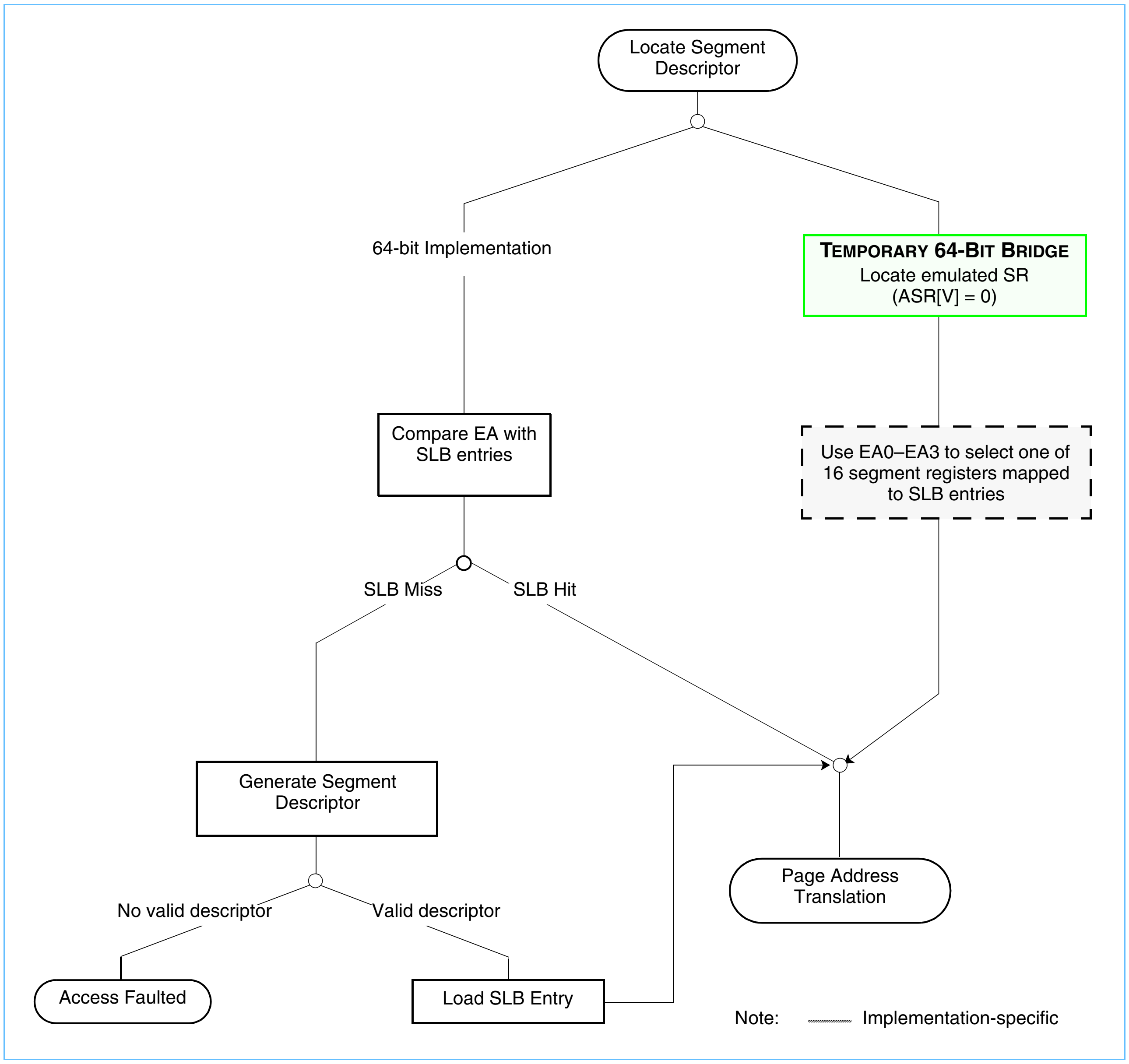

Address Translation with Segment Descriptor
Locate Segment Descriptor
(See *Figure 7-5*)
Page Address Translation
otherwise
I-Fetch with N bit set in Segment Descriptor (no-execute)
Generate Virtual Address from Segment Descriptor
Compare Virtual Address with TLB Entries
TLB Miss
TLB Hit
(See *Figure 7-10*)
Perform Page Table Search Operation
(See *Figure 7-19*)
Access Permitted
Access Protected
Translate Address
Access Faulted
PTE Not Found
PTE Found
Continue Access to Memory Subsystem
Access Faulted
Load TLB Entry
Notes:
Implementation-specific

The segment descriptor for each access is generated by the operating system and placed in the SLB. Alternately, an explicit segment table can be built by the operating system, from which segment descriptors are copied, as needed, into the SLB.

Temporary 64-Bit Bridge

Processors that implement the 64-bit bridge maintain segment descriptors on-chip by emulating segment tables in 16 SLB entries. As shown in *Figure 7-5*, this feature is enabled by clearing the optional ASR[V] bit. This indicates that any value in the STABORG is invalid and that segment table hashing is not implemented.

Figure 7-5. Location of Segment Descriptors


Locate Segment Descriptor
64-bit Implementation
TEMPORARY 64-BIT BRIDGE
Locate emulated SR (ASR[V] = 0)
Compare EA with SLB entries
Use EA0–EA3 to select one of 16 segment registers mapped to SLB entries
SLB Miss
SLB Hit
Generate Segment Descriptor
Page Address Translation
No valid descriptor
Valid descriptor
Access Faulted
Load SLB Entry
Note:
Implementation-specific

Selection of Page Address Translation

The information in the segment descriptor is used to generate the n-bit ($65 \leq n \leq 80$) virtual address. The virtual address is then used to identify the page address translation information (stored as page table entries (PTEs) in a page table in memory). Although the architecture does not require the existence of a TLB, one or more TLBs may be implemented in the hardware to store copies of recently-used PTEs on-chip for increased performance. A TLB is used like a small cache of the much larger PTE tables in memory.

If an access hits in the TLB, the page translation occurs and the physical address bits are forwarded to the memory subsystem. If the translation is not found in the TLB, the MMU requires a search of the page table. The hardware of some implementations may perform the table search automatically, while others may trap to an exception handler for the system software to perform the page table search. If the translation is found, a new TLB entry is created and the page translation is once again attempted. This time, the TLB is guaranteed to hit. When the PTE is located, the access is qualified with the appropriate protection bits. If the access is determined to be protected (not allowed), an exception (ISI or DSI exception) is generated.

If the PTE is not found by the table search operation, an ISI or DSI exception is generated. This is also known as a page fault.

7.2.7 MMU Exceptions Summary

In order to complete any memory access, the effective address must be translated to a physical address. A translation exception condition occurs if this translation fails for one of the following reasons:

- There is no valid entry in the page table for the page specified by the effective address (and segment descriptor).
- There is no valid segment descriptor.
- An address translation is found but the access is not allowed by the memory protection mechanism.

The translation exception conditions cause either the ISI or the DSI exception to be taken as shown in *Table 7-4*. The state saved by the processor for each of these exceptions contains information that identifies the address of the failing instruction. Refer to *Chapter 6, Exceptions* for a more detailed description of exception processing, and the bit settings of SRR1 and DSISR when an exception occurs.

Table 7-4. Translation Exception Conditions

Condition	Description	Exception
Page fault (no PTE found)	No matching PTE found in page tables	I access: ISI exception SRR1[33] = ‘1’
		D access: DSI exception DSISR[1] = ‘1’
Segment fault (no STE found)	No matching STE found in the segment tables	I access: ISI exception SRR1[42] = ‘1’
		D access: DSI exception DSISR[10] = ’1’
Page protection violation	Conditions described in *Table 7-12* for page protection	I access: ISI exception SRR1[36] = ‘1’
		D access: DSI exception DSISR[4] = ‘1’
No-execute protection violation	Attempt to fetch instruction when SR[N] = ‘1’ or STE[N] = ‘1’	ISI exception SRR1[35] = ‘1’
Instruction fetch from guarded memory	Attempt to fetch instruction when MSR[IR] = ‘1’ and PTE[G] = ‘1’	ISI exception SRR1[35] = ‘1’

In addition to the translation exceptions, there are other MMU-related conditions (some of them implementation-specific) that can cause an exception to occur. These conditions map to the exceptions as shown in *Table 7-5*. The only MMU exception conditions that occur when MSR[DR] = ‘0’ are the conditions that cause the alignment exception for data accesses. For more detailed information about the conditions that cause the alignment exception (in particular for string/multiple instructions), see *Section 6.4.8 Alignment Exception (0x00600)*. Refer to *Chapter 6, Exceptions* for a complete description of the SRR1 and DSISR bit settings for these exceptions.

Table 7-5. Other MMU Exception Conditions

Condition	Description	Exception
dcbz with W = ‘1’ or I = ‘1’ (might cause exception or operation might be performed to memory)	**dcbz** instruction to write-through or cache-inhibited segment	Alignment exception (implementation-dependent)
ldarx, **stdcx.**, **lwarx**, or **stwcx.** with W = ‘1’ (might cause exception or execute correctly)	Reservation instruction to write-through segment	DSI exception (implementation-dependent) DSISR[5] = ‘1’
eciwx or **ecowx** attempted when external control facility disabled	**eciwx** or **ecowx** attempted with EAR[E] = ‘0’	DSI exception DSISR[11] = ‘1’
lmw, **stmw**, **lswi**, **lswx**, **stswi**, or **stswx** instruction attempted in little-endian mode	**lmw**, **stmw**, **lswi**, **lswx**, **stswi**, or **stswx** instruction attempted while MSR[LE] = ‘1’	Alignment exception
Operand misalignment	Translation enabled and operand is misaligned as described in *Chapter 6, Exceptions*.	Alignment exception (some of these cases are implementation-dependent)

7.2.8 MMU Instructions and Register Summary

The MMU instructions and registers provide the operating system with the ability to set up the segment descriptors in the SLB, and the page table in memory from which entries can be cached in a TLB, if implemented.

Note: Because the implementation of TLBs is optional, the instructions that refer to these structures are also optional. However, as these structures serve as caches of the page table, there must be a software protocol for maintaining coherency between these caches and the tables in memory whenever changes are made to the tables in memory. Therefore, the PowerPC OEA specifies that a processor implementing a TLB is guaranteed to have a means for doing the following:

- Invalidating an individual TLB entry (the architecture defines the optional **tlbie** instruction for this purpose)
- Invalidating the entire TLB (the architecture defines the optional **tlbia** instruction for this purpose)

Similarly, a processor is guaranteed to have a means for doing the following:

- Invalidating an individual SLB entry (the architecture defines the **slbie** instruction for this purpose)
- Invalidating the entire SLB (the architecture defines the **slbia** instruction for this purpose)

TEMPORARY 64-BIT BRIDGE

When the processor is using the 64-bit bridge, neither the **slbie** or **slbia** instruction should be executed.

When the tables in memory are changed, the operating system purges these caches of the corresponding entries, allowing the translation caching mechanism to refetch from the tables when the corresponding entries are required.

A processor may implement one or more of the instructions described in this section to support table invalidation. Alternatively, an algorithm may be specified that performs one of the functions listed above (for example, a loop invalidating individual TLB entries may be used to invalidate the entire TLB), or different instructions may be provided.

A processor may also perform additional functions (not described here), as well as those described in the implementation of some of these instructions. For example, the **tlbie** instruction may be implemented to purge all TLB entries in a congruence class (that is, all TLB entries indexed by the specified effective address which can include corresponding entries in data and instruction TLBs) or the entire TLB.

Note: If a processor does not implement an optional instruction it treats the instruction as a no-op or as an illegal instruction, depending on the implementation. Also, note that the TLB concepts described here are conceptual; that is, a processor may implement parallel sets of TLBs for instructions and data.

Because the MMU specification for PowerPC processors is so flexible, it is recommended that the software that uses these instructions and registers be encapsulated into subroutines to minimize the impact of migrating across the family of implementations.

Table 7-6 summarizes the PowerPC instructions that specifically control the MMU. For more detailed information about the instructions, refer to *Chapter 8, Instruction Set*.

Table 7-6. Instruction Summary—Control MMU

Instruction	Description
mtsr SR,**r**S	Move to Segment Register SR[SR]⟵ **r**S 64-bit bridge only
mtsrin **r**S,**r**B	Move to Segment Register Indirect SR[**r**B[0–3]]⟵**r**S 64-bit bridge only
mfsr **r**D,SR	Move from Segment Register **r**D⟵SR[SR] 64-bit bridge only
mfsrin **r**D,**r**B	Move from Segment Register Indirect **r**D⟵SR[**r**B[0–3]] 64-bit bridge only
tlbia (optional)	Translation Lookaside Buffer Invalidate All For all TLB entries, TLB[V]⟵0 Causes invalidation of TLB entries only for the processor that executed the **tlbia**
tlbie **r**B (optional)	Translation Lookaside Buffer Invalidate Entry If TLB hit (for effective address specified as **r**B), TLB[V]⟵0 Causes TLB invalidation of entry in all processors in the system
tlbsync (optional)	Translation Lookaside Buffer Synchronize Ensures that all **tlbie** instructions previously executed by the processor executing the **tlbsync** instruction have completed on all processors
slbia	Segment Table Lookaside Buffer Invalidate All For all SLB entries, SLB[V]⟵0 64-bit implementations only
slbie **r**B (optional)	Segment Table Lookaside Buffer Invalidate Entry If SLB hit (for effective address specified as **r**B), SLB[V]⟵0 64-bit implementations only
slbmte **r**S, **r**B	SLB Move to Entry SLB[**r**B(52..63)]⟵ **r**S,**r**B
slbmfev **r**D, **r**B	SLB Move from Entry VSID **r**D⟵ SLB[**r**B(52..63)]$_{VSID}$
slbmfee **r**D, **r**B	SLB Move from Entry ESID **r**D⟵ SLB[**r**B(52..63)]$_{ESID}$

The operating system uses the SDR1 register to program the MMU. The SDR1 register specifies the base and size of the page tables in memory. SDR1 is defined as a 64-bit register and is a special-purpose register that is accessed by the **mtspr** and **mfspr** instructions.

7.2.9 TLB Entry Invalidation

Optionally, PowerPC processors implement TLB structures that store on-chip copies of the PTEs that are resident in physical memory. These processors have the ability to invalidate resident TLB entries through the use of the **tlbie** and **tlbia** instructions. Additionally, these instructions may also enable a TLB invalidate signalling mechanism in hardware so that other processors also invalidate their resident copies of the matching PTE. See *Chapter 8, Instruction Set* for detailed information about the **tlbie** and **tlbia** instructions.

7.3 Real Addressing Mode

If address translation is disabled (MSR[IR] = ‘0’ or MSR[DR] = ‘0’) for a particular access, the effective address is treated as the physical address and is passed directly to the memory subsystem as a real addressing mode address translation. If an implementation has a smaller physical address range than effective address range, the extra high-order bits of the effective address may be ignored in the generation of the physical address.

Section 2.3.16 Synchronization Requirements for Special Registers and for Lookaside Buffers describes the synchronization requirements for changes to MSR[IR] and MSR[DR].

The addresses for accesses that occur in real addressing mode bypass all memory protection checks as described in *Section 7.4.4 Page Memory Protection* and do not cause the recording of referenced and changed information (described in *Section 7.4.3 Page History Recording*).

For data accesses that use real addressing mode, the memory access mode bits (WIMG) are assumed to be ‘0011’. That is, the cache is write-back and memory does not need to be updated immediately (W = ‘0’), caching is enabled (I = ‘0’), data coherency is enforced with memory, I/O, and other processors (caches) (M = ‘1’, so data is global), and the memory is guarded (G = ‘1’). For instruction accesses in real addressing mode, the memory access mode bits (WIMG) are assumed to be either ‘0001’ or ‘0011’. That is, caching is enabled (I = ‘0’) and the memory is guarded (G = ‘1’). Additionally, coherency may or may not be enforced with memory, I/O, and other processors (caches) (M = ‘0’ or ‘1’, so data may or may not be considered global). For a complete description of the WIMG bits, refer to *Section 5.2.1 Memory/Cache Access Attributes*.

Note: The attempted execution of the **eciwx** or **ecowx** instructions while MSR[DR] = ‘0’ causes boundedly-undefined results.

Whenever an exception occurs, the processor clears both the MSR[IR] and MSR[DR] bits. Therefore, at least at the beginning of all exception handlers (including reset), the processor operates in real addressing mode for instruction and data accesses. If address translation is required for the exception handler code, the software must explicitly enable address translation by accessing the MSR as described in *Chapter 2, PowerPC Register Set*.

Note: An attempt to access a physical address that is not physically present in the system may cause a machine check exception (or even a checkstop condition), depending on the response by the system for this case. Thus, care must be taken when generating addresses in real addressing mode. This can also occur when translation is enabled and the ASR or SDR1 registers set up the translation such that nonexistent memory is accessed. See *Section 6.4.2 Machine Check Exception (0x00200)* for more information on machine check exceptions.

TEMPORARY 64-BIT BRIDGE

Note: If ASR[V] = ‘0’, a reference to a nonexistent address in the STABORG field does not cause a machine check exception.

7.4 Memory Segment Model

Memory in the PowerPC OEA is divided into 256-Mbyte segments. This segmented memory model provides a way to map 4-Kbyte (or implementation specific larger size) pages of effective addresses to pages in physical memory (page address translation), while providing the programming flexibility afforded by a large virtual address space (up to 80 bits).

The page translation proceeds in the following two steps:

1. From effective address to the virtual address, and
2. From virtual address to physical address.

The page address translation mechanism is described in the following sections, followed by a summary of page address translation with a detailed flow diagram.

7.4.1 Recognition of Addresses in Segments

The page address translation uses segment descriptors, which provide virtual address and protection information, and page table entries (PTEs), which provide the physical address and page protection information. The segment descriptors are programmed by the operating system to provide the virtual ID for a segment. In addition, the operating system also creates the page table in memory that provides the virtual-to-physical address mappings (in the form of PTEs) for the pages in memory.

Segments in the OEA can be classified as memory segments. An effective address in these segments represents a virtual address that is used to define the physical address of the page.

All accesses generated by the processor can be mapped to a segment descriptor; however, if translation is disabled (MSR[IR] = ‘0’ or MSR[DR] = ‘0’ for an instruction or data access, respectively), real addressing mode translation is performed as described in *Section 7.3 Real Addressing Mode*. Otherwise the access maps to memory space and page address translation is performed.

After a memory segment is selected, the processor creates the virtual address for the segment and searches for the PTE that dictates the physical page number to be used for the access.

Note: I/O devices can be easily mapped into memory space and used as memory-mapped I/O.

7.4.2 Page Address Translation Overview

The first step in page address translation for 64-bit implementations is the conversion of the 64-bit effective address of an access into the virtual address (between 65 and 80 bits depending on the implementation). The virtual address is then used to locate the PTE in the page table in memory. The physical page number is then extracted from the PTE and used in the formation of the physical address of the access.

Note: For increased performance, some processors may implement on-chip TLBs to store copies of recently-used PTEs.

Figure 7-6 shows an overview of the translation of an effective address to a physical address for 64-bit implementations (assuming an 80-bit virtual address) as follows:

- Bits [0–35] of the effective address comprise the effective segment ID (ESID) used to select a segment descriptor, from which the virtual segment ID (VSID) is extracted.
- Bits [36–(63-p)] of the effective address correspond to the page number (index) within the segment; these bits are concatenated with the VSID from the segment descriptor to form the virtual page number (VPN).

The VPN is used to search for the PTE in either an on-chip TLB or the page table. The PTE then provides the physical page number (also known as the real page number or RPN).

Note: Bits [36–40] form the abbreviated page index (API) which is used to compare with page table entries during hashing. This is described in detail in *Section 7.5.1.8 PTEG Address Mapping Example* on page 284.

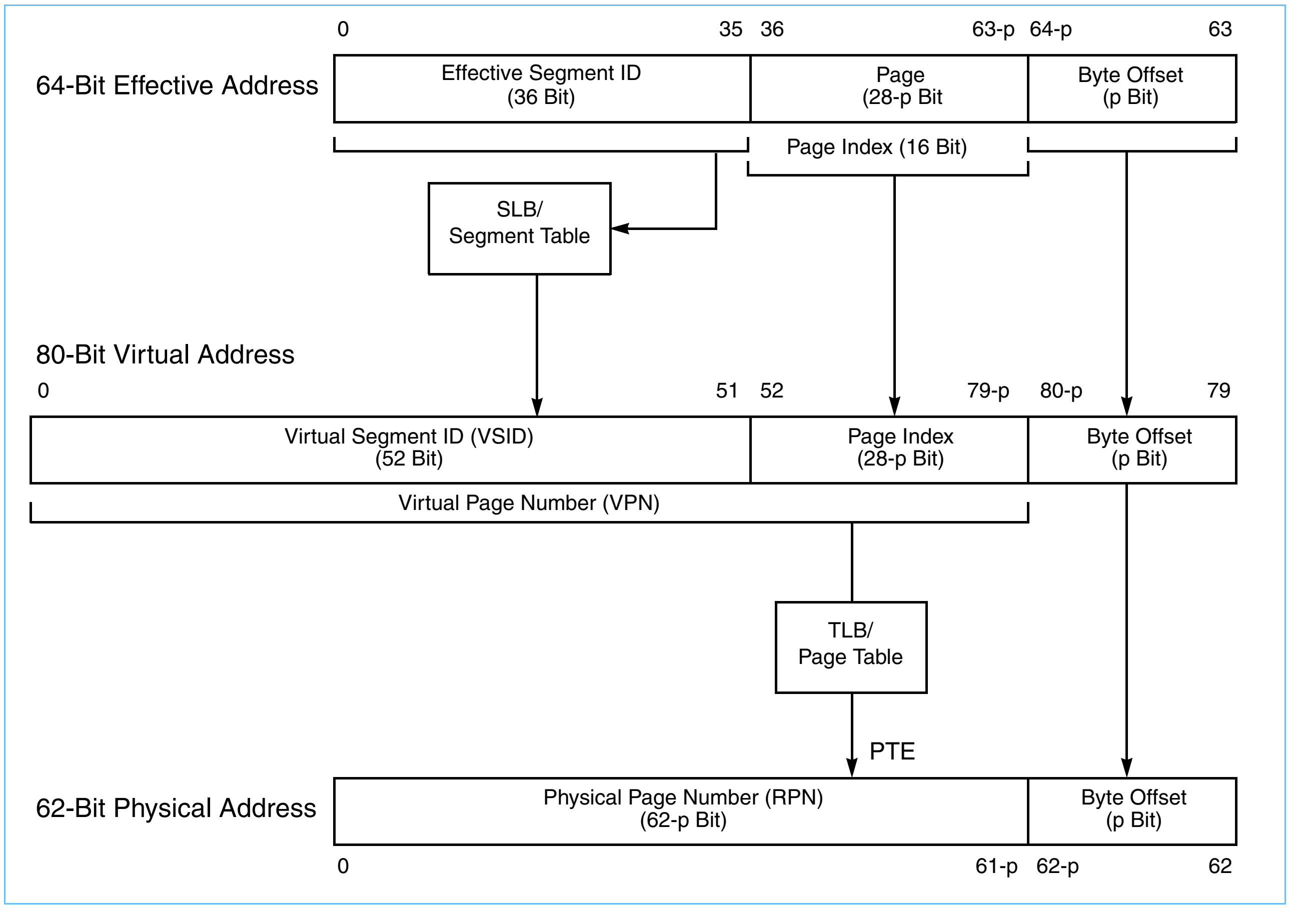
- Bits [(64-p)–63] of the effective address are the byte offset within the page; these are concatenated with the real page number (RPN) field of a PTE to form the physical (real) address used to access memory.

TEMPORARY 64-BIT BRIDGE

Because processors that implement the 64-bit bridge access only a 32-bit address space, only 16 STEs are required to define the entire 4-Gbyte address space. Page address translation for 64-bit processors using the 64-bit bridge uses a subset of the functionality described here for 64-bit implementations. For example, only bits [32–35] are used to select a segment descriptor, and as in the 32-bit portion of the architecture, only 16 on-chip segment registers are required. These segment descriptors are maintained in 16 SLB entries.

Refer to *Section 7.6 Migration of Operating Systems from 32-Bit Implementations to 64-Bit Implementations* on page 292 for details concerning the 64-bit bridge.

Figure 7-6. Page Address Translation Overview



7.4.2.1 Segment Lookaside Buffer (SLB)

The Segment Lookaside Buffer (SLB) specifies the mapping between Effective Segment IDs (ESIDs) and Virtual Segment IDs (VSIDs). The number of SLB entries is implementation-dependent, except that all implementations provide at least 32 entries.

The contents of the SLB are managed by software, using the instructions described in *Table 7-6*. See *Section 4.1.5.1 Context Synchronizing Instructions* for the rules that software must follow when updating the SLB.

Each SLB entry (SLBE) maps one ESID to one VSID. *Figure 7-7* illustrates an SLB entry.

Figure 7-7. SLB Entry

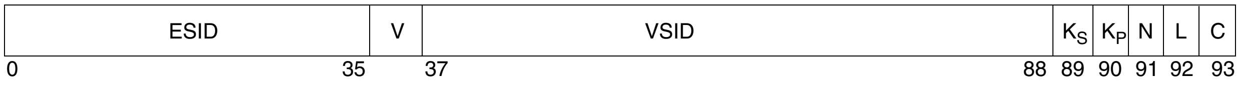


Table 7-7. SLB Entry Bit Description – 64-bit Implementations

Bit	Name	Description
0-35	ESID	Effective segment ID
36	V	Entry valid (V=’1’) or invalid (V=’0’)
37-88	VSID	Virtual segment ID
89	K_S	Supervisor state storage key
90	K_P	User (problem) state storage key
91	N	No-execute segment if N=1
92	L	Virtual pages are large (L=1) or 4KB (L=0)
93	C	Class

On implementations that support a virtual address size of only n bits, n< 80, bits [0 to 79-n] of the VSID field are treated as reserved bits, and software must set them to zeros.

A No-execute segment (N=’1’) contains data that should not be executed.

The L bit selects between the two virtual page sizes, 4 KB (p=12) and “large.” The large page size is an implementation-dependent value that is a power of 2 and is in the range 8 KB to 256 MB ($13 \leq p \leq 28$). Some implementations may provide a means by which software can select the large page size from a set of several implementation-dependent sizes during system initialization.

If “large page” is used in reference to physical (real) memory, it means the sequence of contiguous real (4 KB) pages to which a large virtual page is mapped. The Class field is used in conjunction with the **slbie** instruction.

Software must ensure that the SLB contains at most one entry that translates a given effective address (for example, that a given ESID is contained in no more than one SLB entry).

Because the virtual page size is used both in searching the Page Table and in forming the real address using the matching Page Table Entry (PTE), and PTEs contain no indication of the virtual page size, the virtual page size must be the same for all address translations that use a given VSID value. This has the following consequences, which apply collectively to all processors that use the same Page Table.

- The value of the L bit must be the same in all SLB entries that contain a given VSID value.
- Before changing the value of the L bit in an SLB entry, software must invalidate all SLB entries, TLB entries, and PTEs that contain the corresponding VSID value.

SLB Search

When the hardware searches the SLB, all entries are tested for a match with the EA. For a match to exist, the following must be true:

- SLBE[V] = ‘1’
- SLBE[ESID] = EA[0-35]

If the SLB search succeeds, the virtual address (VA) is formed by concatenating the VSID from the matching SLB entry with bits [36-63] of the effective address.

The Virtual Page Number (VPN) is bits [0 to 79-p] of the virtual address.

If the SLB search fails, a segment fault occurs. This is an Instruction Segment exception or a Data Segment exception, depending on whether the effective address is for an instruction fetch or for a data access.

7.4.2.2 Page Table Entry (PTE) Definition and Format

Page table entries (PTEs) are generated and placed in a page table in memory by the operating system using the hashing algorithm described in *Section 7.5.1.3 Page Table Hashing Functions*. Some of the fields are defined as follows:

- The virtual segment ID field corresponds to the high-order bits of the virtual page number (VPN), and, along with the H, V, and API fields, it is used to locate the PTE (used as match criteria in comparing the PTE with the segment information).
- The R and C bits maintain history information for the page as described in *Section 7.4.3 Page History Recording*.
- The WIMG bits define the memory/cache control mode for accesses to the page.
- The PP bits define the remaining access protection constraints for the page. The page protection provided by PowerPC processors is described in *Section 7.4.4 Page Memory Protection*.

Conceptually, the page table in memory must be searched to translate the address of every reference. For performance reasons, however, some processors use on-chip TLBs to cache copies of recently-used PTEs so that the table search time is eliminated for most accesses. In this case, the TLB is searched for the address translation first. If a copy of the PTE is found, then no page table search is performed. As TLBs are noncoherent caches of PTEs, software that changes the page table in any way must perform the appropriate TLB invalidate operations to keep the on-chip TLBs coherent with respect to the page table in memory.

Each PTE is a 128-bit entity (two doublewords) that maps a virtual page number (VPN) to a physical page number (RPN). Information in the PTE is used in the page table search process (to determine a page table hit) and provides input to the memory protection mechanism. *Figure 7-8* shows the format of the two doublewords that comprise a PTE for 64-bit implementations.

Figure 7-8. Page Table Entry Format

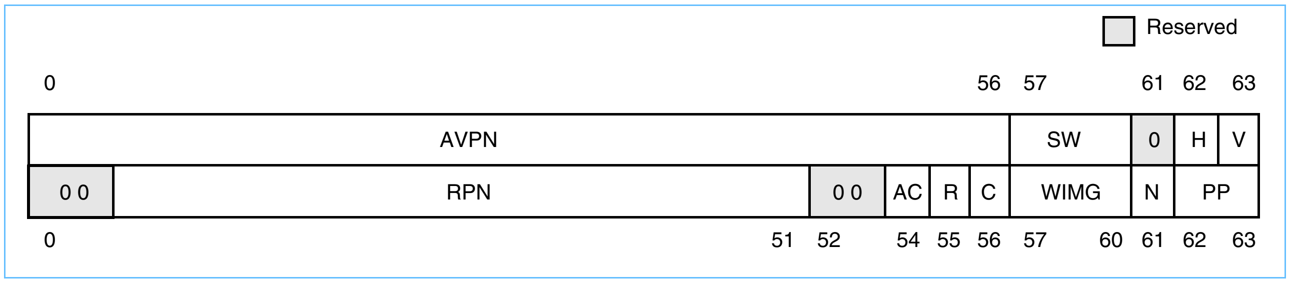


Table 7-8 lists the corresponding bit definitions for each doubleword in a PTE as defined.

Table 7-8. PTE Bit Definitions

Doubleword	Bit	Name	Description
0	0-56	AVPN	Abbreviated virtual page number
	57-60	SW	Available for software use
	61	—	Reserved
	62	H	Hash function identifier
	63	V	Entry valid (V = ‘1’) or invalid (V = ‘0’)
1	0-1	—	Reserved
	2-51	RPN	Physical page number
	52–53	—	Reserved
	54	AC	Address compare bit
	55	R	Referenced bit
	56	C	Changed bit
	57–60	WIMG	Memory/cache access control bits
	61	N	No execute page if N = ‘1’
	62–63	PP	Page protection bits

If $p \leq 23$, the Abbreviated Virtual Page Number (AVPN) field contains bits [0-56] of the VPN. Otherwise bits [0 - (79-p)] of the AVPN field contain bits [0 to (79-p)] of the VPN, and bits [(80-p) to 56] of the AVPN field must be zeros.

Note: If $p \leq 23$, the AVPN field omits the low-order (23-p) bits of the VPN. These bits are not needed in the PTE, because the low-order 11 bits of the VPN are always used in selecting the PTEGs to be searched.

On implementations that support a virtual address size of only n bits, n< 80, bits [0 to (79-n)] of the AVPN field must be zeros.

The RPN field contains the page number of the real page that contains the first byte of the block of real storage to which the virtual page is mapped. If p> 12, the low-order p-12 bits of the RPN field (bits [(64-p) to 51] of doubleword 1 of the PTE) must be ‘0’. On implementations that support a real address size of only m bits, m< 62, bits [0 to (61-m)] of the RPN field must be zeros.

Note: For a large virtual page, the high-order [62-p] bits of the RPN field (bits [0 to (61-p)]) comprise the large real page number.

7.4.3 Page History Recording

Referenced (R) and changed (C) bits reside in each PTE to keep history information about the page. The operating system then uses this information to determine which areas of memory to write back to disk when new pages must be allocated in main memory. Furthermore, R and C bits are maintained only for accesses made while address translation is enabled (MSR[IR] = ‘1’ or MSR[DR] = ‘1’).

In general, the referenced and changed bits are updated to reflect the status of the page based on the access, as shown in *Table 7-9*.

Table 7-9. Table Search Operations to Update History Bits

R and C bits	Processor Action
00	Read: Table search operation to update R Write: Table search operation to update R and C
01	Combination doesn’t occur
10	Read: No special action Write: Table search operation to update C
11	No special action for read or write

In processors that implement a TLB, the processor may perform the R and C-bit updates based on the copies of these bits resident in the TLB. For example, the processor may update the C-bit based only on the status of the C-bit in the TLB entry in the case of a TLB hit (the R-bit may be assumed to be set in the page tables if there is a TLB hit). Therefore, when software clears the R and C-bits in the page tables in memory, it must invalidate the TLB entries associated with the pages whose referenced and changed bits were cleared. See *Section 7.5.3 Page Table Updates* for all of the constraints imposed on the software when updating the referenced and changed bits in the page tables.

The R-bit for a page may be set by the execution of the **dcbt** or **dcbtst** instruction to that page. However, neither of these instructions cause the C-bit to be set.

The update of the referenced and changed bits is performed by PowerPC processors as if address translation were disabled (real addressing mode address).

7.4.3.1 Referenced Bit

The referenced bit for each virtual (real) page is located in the PTE. Every time a page is referenced (by an instruction fetch, or any other read or write access) the referenced bit is set in the page table. The referenced bit may be set immediately, or the setting may be delayed until the memory access is determined to be successful. Because the reference to a page is what causes a PTE to be loaded into the TLB, some processors may assume the R-bit in the TLB is always set. The processor never automatically clears the referenced bit.

The referenced bit is only a hint to the operating system about the activity of a page. At times, the referenced bit may be set although the access was not logically required by the program or even if the access was prevented by memory protection. Examples of this include the following:

- Fetching of instructions not subsequently executed
- Accesses generated by an **lswx** or **stswx** instruction with a zero length
- Accesses generated by a **stwcx.** or **stdcx.** instruction when no store is performed
- Accesses that cause exceptions and are not completed

7.4.3.2 Changed Bit

The changed bit for each virtual page is located both in the PTE in the page table and in the copy of the PTE loaded into the TLB (if a TLB is implemented). Whenever a data store instruction is executed successfully, if the TLB search (for page address translation) results in a hit, the changed bit in the matching TLB entry is checked. If it is already set, no additional action is required. If the TLB changed bit is ‘0’, it is set and a table search operation is performed to set the C-bit in the corresponding PTE in the page table.

Processors cause the changed bit (in both the PTE in the page tables and in the TLB if implemented) to be set only when a store operation is allowed by the page memory protection mechanism and the store is guaranteed to be in the execution path, unless an exception, other than those caused by one of the following occurs:

- System-caused interrupts (system reset, machine check, external, and decrementer interrupts)
- Floating-point enabled exception type program exceptions when the processor is in an imprecise mode
- Floating-point assist exceptions for instructions that cause no other kind of precise exception

Furthermore, the following conditions may cause the C-bit to be set:

- The execution of an **stwcx.** or **stdcx.** instruction is allowed by the memory protection mechanism but a store operation is not performed.
- The execution of an **stswx** instruction is allowed by the memory protection mechanism but a store operation is not performed because the specified length is zero.

No other cases cause the C-bit to be set.

7.4.3.3 Scenarios for Referenced and Changed Bit Recording

This section provides a summary of the model (defined by the OEA) used by PowerPC processors that maintain the referenced and changed bits automatically in hardware, in the setting of the R and C-bits. In some scenarios, the bits are guaranteed to be set by the processor; in some scenarios, the architecture allows that the bits may be set (not absolutely required); and in some scenarios, the bits are guaranteed to not be set. Note that when the hardware updates the R and C-bits in memory, the accesses are performed as a physical memory access, as if the WIMG bit settings were ‘0010’ (that is, as unguarded cacheable operations in which coherency is required).

In implementations that do not maintain the R and C-bits in hardware, software assistance is required. For these processors, the information in this section still applies, except that the software performing the updates is constrained to the rules described (that is, must set bits shown as guaranteed to be set and must not set bits shown as guaranteed to not be set).

Note: This software should be contained in the area of memory reserved for implementation-specific use and should be invisible to the operating system.

Table 7-10 defines a prioritized list of the R and C-bit settings for all scenarios. The entries in the table are prioritized from top to bottom, such that a matching scenario occurring closer to the top of the table takes precedence over a matching scenario closer to the bottom of the table. For example, if an **stwcx.** instruction causes a protection violation and there is no reservation, the C-bit is not altered, as shown for the protection violation case.

Note: In *Table 7-10*, load operations include those generated by load instructions, by the **eciwx** instruction, and by the cache management instructions that are treated as loads with respect to address translation. Similarly, store operations include those operations generated by store instructions, by the **ecowx** instruction, and by the cache management instructions that are treated as stores with respect to address translation.

Table 7-10. Model for Guaranteed R and C Bit Settings

Priority	Scenario	Causes Setting of R-Bit	Causes Setting of C-Bit
1	Page protection violation	Maybe	No
2	Out-of-order instruction fetch or load operation	Maybe	No
3	Out-of-order store operation for instructions that will cause no other kind of precise exception (in the absence of system-caused, imprecise, or floating-point assist exceptions)	Maybe[1]	Maybe[1]
4	All other out-of-order store operations	Maybe[1]	No
5	In-order Load-type instruction	Maybe	No
6	In-order Store-type instruction	Maybe	Maybe[1]
7	In-order instruction fetch	Yes[2]	No
8	Load instruction or **eciwx**	Yes	No
9	Store instruction, **ecowx** or **dcbz** instruction	Yes	Yes
10	**icbi**, **dcbt**, **dcbtst**, **dcbst**, or **dcbf** instruction	Maybe	No

Note:
[1] If C is set, R is guaranteed to also be set.
[2] This includes the case in which the instruction was fetched out of order and R was not set.

7.4.3.4 Synchronization of Memory Accesses and Referenced and Changed Bit Updates

Although the processor updates the referenced and changed bits in the page tables automatically, these updates are not guaranteed to be immediately visible to the program after the load, store, or instruction fetch operation that caused the update. If processor A executes a load or store or fetches an instruction, the following conditions are met with respect to performing the access and performing any R and C-bit updates:

- If processor A subsequently executes a **sync** instruction, both the updates to the bits in the page table and the load or store operation are guaranteed to be performed with respect to all processors and mechanisms before the **sync** instruction completes on processor A.
- Additionally, if processor B executes a **tlbie** instruction that
  - signals the invalidation to the hardware,
  - invalidates the TLB entry for the access in processor A, and
  - is detected by processor A after processor A has begun the access,

  and processor B executes a **tlbsync** instruction after it executes the **tlbie**, both the updates to the bits and the original access are guaranteed to be performed with respect to all processors and mechanisms before the **tlbsync** instruction completes on processor A.

7.4.4 Page Memory Protection

In addition to the no-execute option that can be programmed at the segment descriptor level to prevent instructions from being fetched from a given segment (shown in *Figure 7-4*). There are a number of other memory protection options that can be programmed at the page level. The page memory protection mechanism allows selectively granting read access, granting read/write access, and prohibiting access to areas of memory based on a number of control criteria.

The memory protection used by the page address translation mechanism is different in that the page address translation protection defines a key bit that, in conjunction with the PP bits, determines whether supervisor and user programs can access a page.

For page address translation, the memory protection mechanism is controlled by the following:

- MSR[PR] which defines the mode of the access as follows:
  - MSR[PR] = ‘0’ corresponds to supervisor mode
  - MSR[PR] = ‘1’ corresponds to user mode
- K_S and K_P, the supervisor and user key bits, which define the key for the page
- The PP bits, which define the access options for the page
- For instruction fetches only:
  - No-execute (N) value used for the access
  - PTE[G], the guarded (G) bit in the page table entry used to translate the effective address.

The key bits (K_S and K_P) and the PP bits are located as follows for page address translation:

- K_S and K_P are located in the segment descriptor.
- The PP bits are located in the PTE.

The key bits, the PP bits, and the MSR[PR] bit are used as follows:

- When an access is generated, one of the key bits is selected to be the key as follows:
  - For supervisor accesses (MSR[PR] = ‘0’), the K_S bit is used and K_P is ignored
  - For user accesses (MSR[PR] = ‘1’), the K_P bit is used and K_S is ignored

  That is, key = (K_P & MSR[PR]) | (K_S & ¬MSR[PR])
- For an instruction fetch, the access is not permitted if the N = ‘1’ or PTE[G] = ‘1’
- The selected key is used with the PP bits to determine if instruction fetching, load access, or store access is allowed

Table 7-11 shows the types of accesses that are allowed for the general case (all possible Ks, Kp, and PP bit combinations), assuming that the N-bit in the segment descriptor is cleared (the no-execute option is not selected).

Table 7-11. Access Protection Control with Key

Key[1]	PP[2]	Page Type
0	00	Read/write
0	01	Read/write
0	10	Read/write
0	11	Read only
1	00	No access
1	01	Read only
1	10	Read/write
1	11	Read only

Note:
[1] K_S or K_P selected by state of MSR[PR]
[2] PP protection option bits in PTE

Thus, the conditions that cause a protection violation (not including the no-execute protection option for instruction fetches) are depicted in *Table 7-12* and as a flow diagram in *Figure 7-11*. Any access attempted (read or write) when the key = ‘1’ and PP = ‘00’, causes a protection violation exception condition. When key = ‘1’ and PP = ‘01’, an attempt to perform a write access causes a protection violation exception condition. When PP = ‘10’, all accesses are allowed, and when PP = ‘11’, write accesses always cause an exception. The processor takes either the ISI or the DSI exception (for an instruction or data access, respectively) when there is an attempt to violate the memory protection.

Table 7-12. Exception Conditions for Key and PP Combinations

Key	PP	Prohibited Accesses
0	0x	None
1	00	Read/write
1	01	Write
x	10	None
x	11	Write

Any combination of the K_S, K_P, and PP bits is allowed. One example is if the K_S and K_P bits are programmed so that the value of the key bit for *Table 7-11* directly matches the MSR[PR] bit for the access. In this case, the encoding of K_S = '0' and K_P = '1' is used for the PTE, and the PP bits then enforce the protection options shown in *Table 7-13*.

Table 7-13. Access Protection Encoding of PP Bits for K_S = '0' and K_P = '1'

PP Field	Option	User Read (Key = '1')	User Write (Key = '1')	Supervisor Read (Key = '0')	Supervisor Write (Key = '0')
00	Supervisor-only	Violation	Violation	Y	Y
01	Supervisor-write-only	Y	Violation	Y	Y
10	Both user/supervisor	Y	Y	Y	Y
11	Both read-only	Y	Violation	Y	Violation

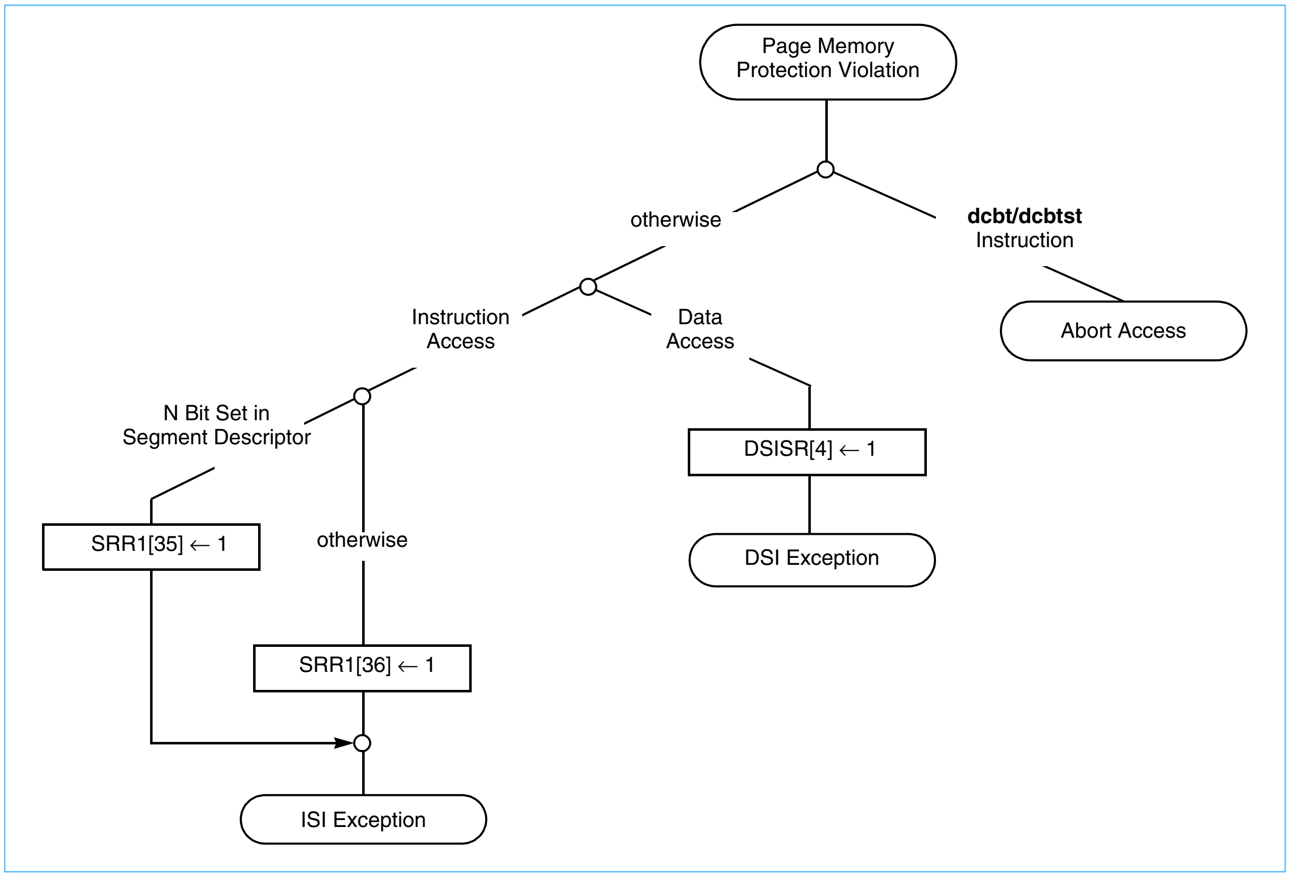
However, if the setting K_S = '1' is used, supervisor accesses are treated as user reads and writes with respect to *Table 7-13*. Likewise, if the setting K_P = '0' is used, user accesses to the page are treated as supervisor accesses in relation to *Table 7-13*. Therefore, by modifying one of the key bits (in the segment descriptor), the way the processor interprets accesses (supervisor or user) in a particular segment can easily be changed.

Note: Only supervisor programs are allowed to modify the key bits for the segment descriptor. Although access to the ASR is privileged, the operating system must protect write accesses to the segment table as well.

As shown in *Figure 7-9*, when the memory protection mechanism prohibits a reference, one of the following occurs depending on the type of access that was attempted:

- For data accesses, a DSI exception is generated and DSISR[4] is set. If the access is a store, DSISR[6] is also set.
- For instruction accesses,
  – an ISI exception is generated and SRR1[36] is set, or
  – an ISI exception is generated and SRR1[35] is set if the segment is designated as no-execute.

Figure 7-9. Memory Protection Violation Flow for Pages



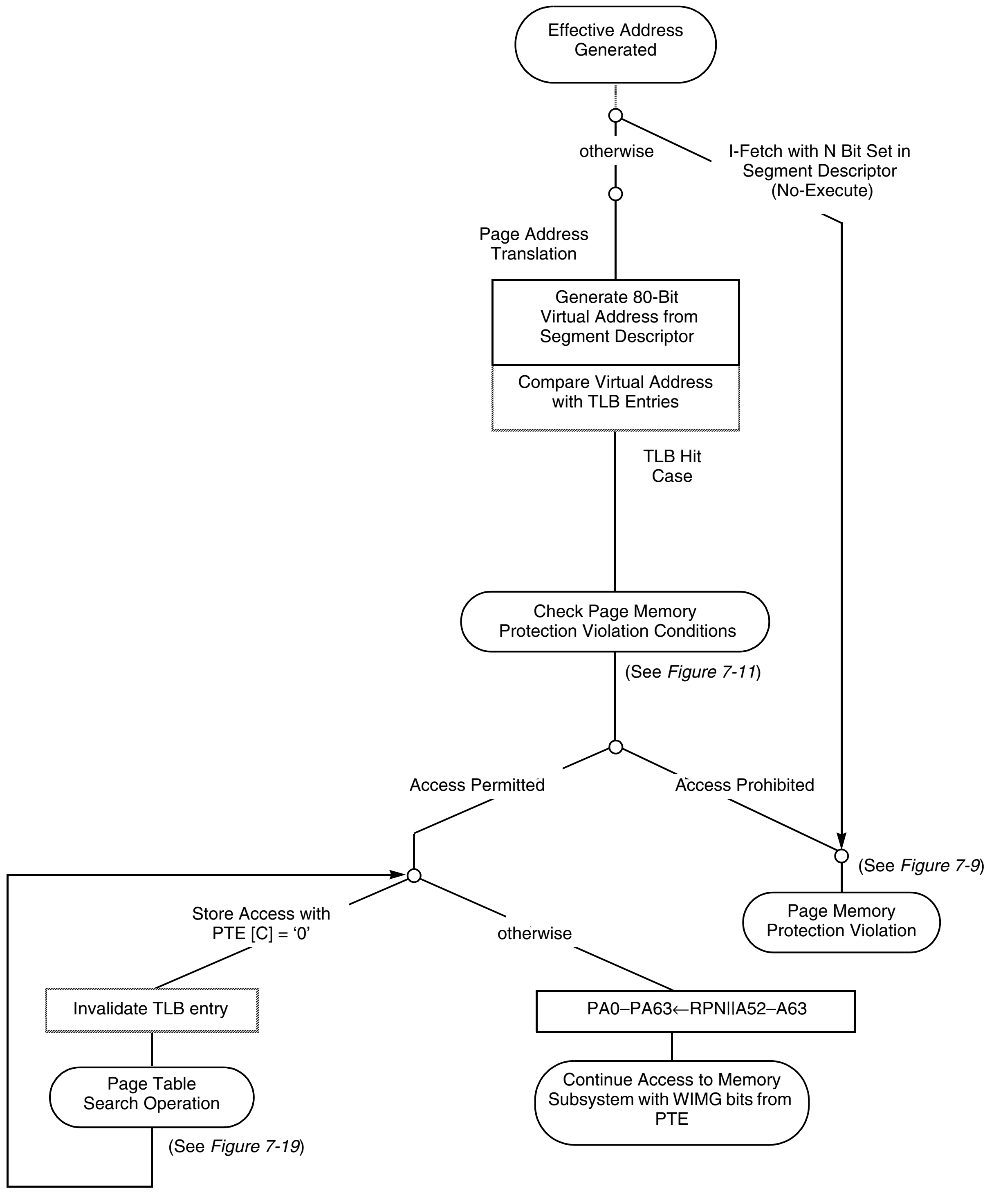
If the page protection mechanism prohibits a store operation, the changed bit is not set (in either the TLB or in the page tables in memory); however, a prohibited store access may cause a PTE to be loaded into the TLB and consequently cause the referenced bit to be set in a PTE (both in the TLB and in the page table in memory).

7.4.5 Page Address Translation Summary

Figure 7-10 provides the detailed flow for the page address translation mechanism in 64-bit implementations. The figure includes the checking of the N-bit in the segment descriptor and then expands on the 'TLB Hit' branch of *Figure 7-4*. The detailed flow for the 'TLB Miss' branch of *Figure 7-4* is described in *Section 7.5.2 Page Table Search Process*. The checking of memory protection violation conditions for page address translation is shown in *Figure 7-11*. The 'Invalidate TLB Entry' box shown in *Figure 7-10* is marked as implementation-specific as this level of detail for TLBs (and the existence of TLBs) is not dictated by the architecture.

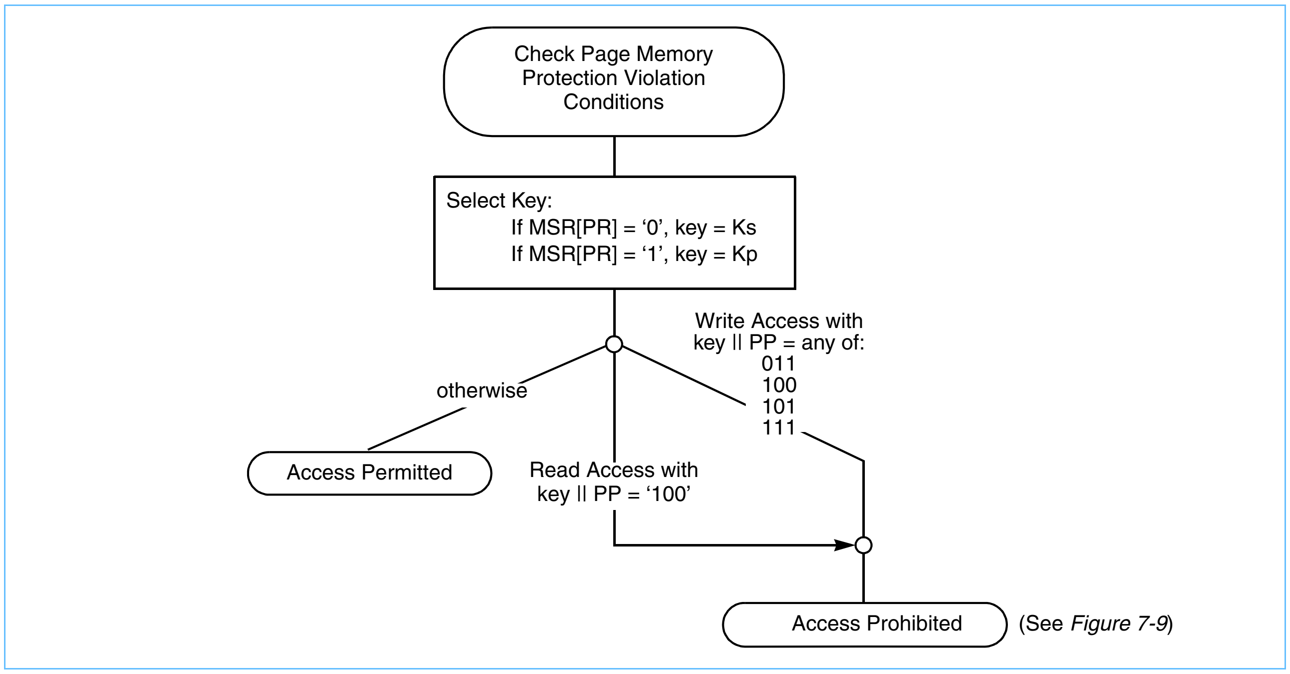
Note: The figure does not show the detection of all exception conditions shown in *Table 7-4* and *Table 7-5*; the flow for many of these exceptions is implementation-specific.

Figure 7-10. Page Address Translation Flow—TLB Hit


Effective Address Generated
otherwise
I-Fetch with N Bit Set in Segment Descriptor (No-Execute)
Page Address Translation
Generate 80-Bit Virtual Address from Segment Descriptor
Compare Virtual Address with TLB Entries
TLB Hit Case
Check Page Memory Protection Violation Conditions
(See Figure 7-11)
Access Permitted
Access Prohibited
(See Figure 7-9)
Page Memory Protection Violation
Store Access with PTE [C] = '0'
otherwise
Invalidate TLB entry
PA0–PA63←RPN||A52–A63
Page Table Search Operation
Continue Access to Memory Subsystem with WIMG bits from PTE
(See Figure 7-19)

Note: ▨ Implementation-specific

Figure 7-11. Page Memory Protection Violation Conditions for Page Address Translation


Check Page Memory Protection Violation Conditions
Select Key:
If MSR[PR] = ‘0’, key = Ks
If MSR[PR] = ‘1’, key = Kp
Write Access with key || PP = any of: 011 100 101 111
otherwise
Access Permitted
Read Access with key || PP = ‘100’
Access Prohibited
(See *Figure 7-9*)

7.5 Hashed Page Tables

If a copy of the PTE corresponding to the VPN for an access is not resident in a TLB (corresponding to a miss in the TLB, provided a TLB is implemented), the processor must search for the PTE in the page tables set up by the operating.

The algorithm specified by the architecture for accessing the page tables includes a hashing function on some of the virtual address bits. Thus, the addresses for PTEs are allocated more evenly within the page tables and the hit rate of the page tables is maximized. This algorithm must be synthesized by the operating system for it to correctly place the page table entries in main memory.

If page table search operations are performed automatically by the hardware, they are performed using physical addresses and as if the memory access attribute bit M = ‘1’ (memory coherency enforced in hardware). If the software performs the page table search operations, the accesses must be performed in real addressing mode (MSR[DR] = ‘0’); this additionally guarantees that M = ‘1’.

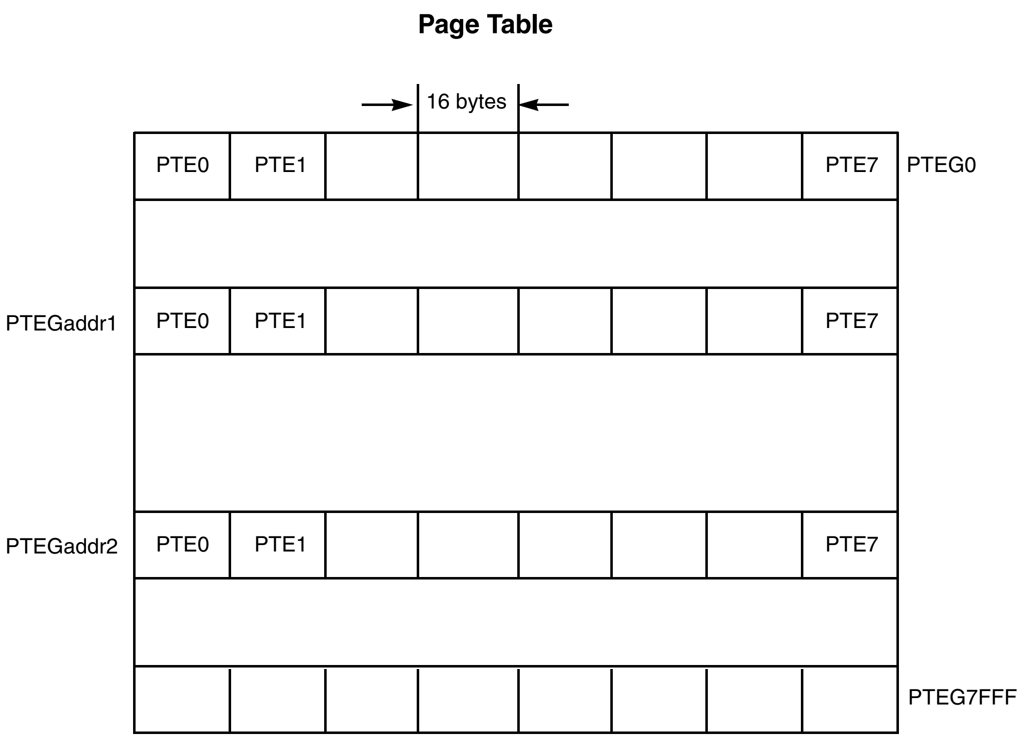
This section describes the format of the page tables and the algorithm used to access them. In addition, the constraints imposed on the software in updating the page tables (and other MMU resources) are described.

7.5.1 Page Table Definition

The hashed page table is a variable-sized data structure that defines the mapping between virtual page numbers and physical page numbers. The page table size is a power of ‘2’, its starting address is a multiple of its size, and the table must reside in memory with the WIMG attributes of ‘0010’.

The page table contains a number of page table entry groups (PTEGs). For 64-bit implementations, a PTEG contains eight page table entries (PTEs) of 16 bytes each; therefore, each PTEG is 128 bytes long. PTEG addresses are entry points for table search operations. *Figure 7-12* shows two PTEG addresses (PTEGaddr1 and PTEGaddr2) where a given PTE may reside.

Figure 7-12. Page Table Definitions



A given PTE can reside in one of two possible PTEGS—one is the primary PTEG and the other is the secondary PTEG. Additionally, a given PTE can reside in any of the PTE locations within an addressed PTEG. Thus, a given PTE may reside in one of 16 possible locations within the page table. If a given PTE is not in either the primary or secondary PTEG, a page table miss occurs, corresponding to a page fault condition.

A table search operation is defined as the search for a PTE within a primary and secondary PTEG. When a table search operation commences, a primary hashing function is performed on the virtual address. The output of the hashing function is then concatenated with bits programmed into the SDR1 register by the operating system to create the physical address of the primary PTEG. The PTEs in the PTEG are then checked, one by one, to see if there is a hit within the PTEG. If the PTE is not located, a secondary hashing function is performed, a new physical address is generated for the PTEG, and the PTE is searched for again, using the secondary PTEG address.

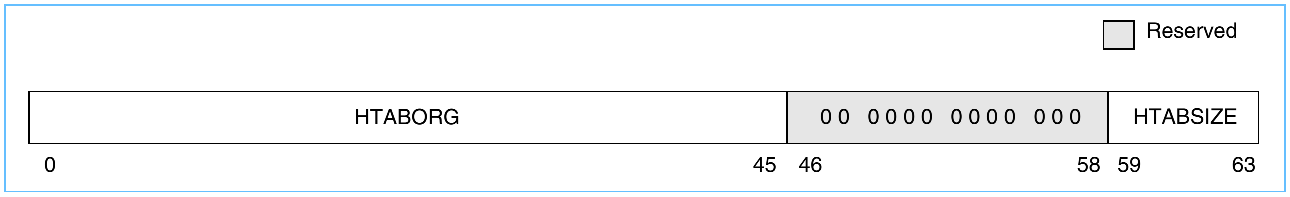
Note: Although a given PTE may reside in one of 16 possible locations, an address that is a primary PTEG address for some accesses also functions as a secondary PTEG address for a second set of accesses (as defined by the secondary hashing function). Therefore, these 16 possible locations are really shared by two different sets of effective addresses. *Section 7.5.1.7 Page Table Structure Example*, illustrates how PTEs map into the 16 possible locations as primary and secondary PTEs.

7.5.1.1 SDR1 Register Definition

The Storage Description Register 1 (SDR1) contains the control information for the page table structure in that it defines the high-order bits for the physical base address of the page table and it defines the size of the table. Note that there are certain synchronization requirements for writing to SDR1 that are described in *Section 2.3.16 Synchronization Requirements for Special Registers and for Lookaside Buffers*.

The format of the SDR1 register is shown in *Figure 7-13*.

Figure 7-13. SDR1 Register Format



The bit settings for SDR1 are described in *Table 7-14*.

Table 7-14. SDR1 Register Bit Settings

Bits	Name	Description
0–1	—	Reserved
2–45	HTABORG	Physical base address of page table
46–58	—	Reserved
59-63	HTABSIZE	Encoded size of page table (used to generate mask)

The HTABORG field in SDR1 contains the high-order 46 bits of the 64-bit physical address of the page table. Therefore, the beginning of the page table lies on a 2^{18} byte (256 Kbyte) boundary at a minimum. If the processor does not support 64 bits of physical address, software should write zeros to those unsupported bits in the HTABORG field (as the implementation treats them as reserved). Otherwise, a machine check exception can occur.

A page table can be any size 2^{n} bytes where $18 \le n \le 46$. The HTABSIZE field in SDR1 contains an integer value that specifies how many bits from the output of the hashing function are used as the page table index. This number must not exceed 28. HTABSIZE is used to generate a mask of the form ‘00...011...1’ (a string of *n* ‘0’ bits (where *n* is 28 – HTABSIZE) followed by a string of ‘1’ bits, the number of which is equal to the value of HTABSIZE). As the table size increases, more bits are used from the output of the hashing function to index into the table. The ‘1’ bits in the mask determine how many additional bits (beyond the minimum of 11) from the hash are used in the index; the HTABORG field must have this same number of low-order bits equal to 0. See *Figure 7-17* for an example of the primary PTEG address generation in a 64-bit implementation.

On implementations that support a real address size of only m bits, m< 62, bits [0 to (61- m)] of the HTABORG field are treated as reserved bits, and software must set them to zeros.

Let n equal the virtual address size (in bits) supported by the implementation. If n< 67, software should set the HTABSIZE field to a value that does not exceed n- 39. Because the high-order [80- n] bits of the VSID are assumed to be zeros, the hash value used in the Page Table search will have the high-order [67- n] bits either all ‘0’s (primary hash) or all ‘1’s (secondary hash). If HTABSIZE > [n- 39], some of these hash value bits will be used to index into the Page Table, with the result that certain PTEGs will not be searched.

7.5.1.2 Page Table Size

The number of entries in the page table directly affects performance because it influences the hit ratio in the page table and thus the rate of page fault exception conditions. If the table is too small, not all virtual pages that have physical page frames assigned may be mapped via the page table. This can happen if more than 16 entries map to the same primary/secondary pair of PTEGs; in this case, many hash collisions may occur.

The minimum allowable size for a page table is 256 Kbytes (2^{11} PTEGs of 128 bytes each). However, it is recommended that the total number of PTEGs in the page table be at least half the number of physical page frames to be mapped. While avoidance of hash collisions cannot be guaranteed for any size page table, making the page table larger than the recommended minimum size reduces the frequency of such collisions, by making the primary PTEGs more sparsely populated, and further reducing the need to use the secondary PTEGs.

Table 7-15 shows example sizes for total main memory. The recommended minimum page table sizes for these example memory sizes are then outlined, along with their corresponding HTABORG and HTABSIZE settings.

Note: Systems with less than 16 Mbytes of main memory may be designed with 64-bit implementations, but the minimum amount of memory that can be used for the page tables is 256 Kbytes in these cases.

Table 7-15. Minimum Recommended Page Table Sizes

Total Main Memory	Recommended Minimum			Settings for Recommended Minimum	
	Memory for Page Tables	Number of Mapped Pages (PTEs)	Number of PTEGs	HTABORG (Maskable Bits [18-45])	HTABSIZE (28-Bit Mask)
16 Mbytes (2^{24})	256 Kbytes (2^{18})	2^{14}	2^{11}	x xxxx	0 0000 (0 . . . . 0000)
32 Mbytes (2^{25})	512 Kbytes (2^{19})	2^{15}	2^{12}	x xxx0	0 0001 (0 . . . . 0001)
64 Mbytes (2^{26})	1 Mbyte (2^{20})	2^{16}	2^{13}	x xx00	0 0010 (0 . . . . 0011)
128 Mbytes (2^{27})	2 Mbytes (2^{21})	2^{17}	2^{14}	x x000	0 0011 (0 . . . . 0111)
256 Mbytes (2^{28})	4 Mbytes (2^{22})	2^{18}	2^{15}	x . . .x 0000	0 0100 (0 . . .0 1111)
. . .	. . .	. . .	. . .	. . .	. . .
2^{51} Bytes	2^{45} Bytes	2^{41}	2^{38}	x 0 . . . 0000	1 1011 (0 1 . . . 1111)
2^{52} Bytes	2^{46} Bytes	2^{42}	2^{39}	0 0000	1 1100 (1 . . . .1111)

As an example, if the physical memory size is 2^{31} bytes (2 Gbyte), there are $2^{31} – 2^{12}$ (4 Kbyte page size) = 2^{19} (512 Kbyte) total page frames. If this number of page frames is divided by 2, the resultant minimum recommended page table size is 2^{18} PTEGs, or 2^{25} bytes (32 Mbytes) of memory for the page tables.

7.5.1.3 Page Table Hashing Functions

The MMU uses two different hashing functions, a primary and a secondary, in the creation of the physical addresses used in a page table search operation. These hashing functions distribute the PTEs within the page table, in that there are two possible PTEGs where a given PTE can reside. Additionally, there are eight possible PTE locations within a PTEG where a given PTE can reside. If a PTE is not found using the primary hashing function, the secondary hashing function is performed, and the secondary PTEG is searched.

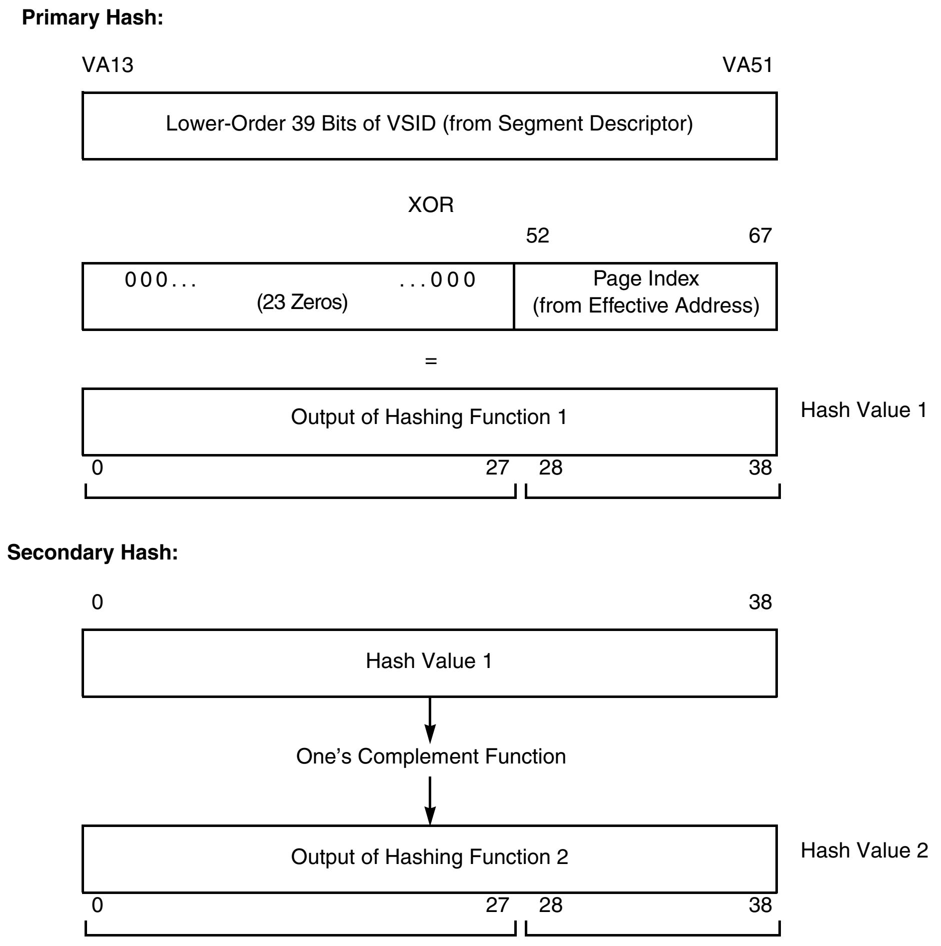
Note: These two functions must also be used by the operating system to set up the page tables in memory appropriately.

Typically, the hashing functions provide a high probability that a required PTE is resident in the page table, without requiring the definition of all possible PTEs in main memory. However, if a PTE is not found in the secondary PTEG, a page fault occurs and an exception is taken. Thus, the required PTE can then be placed into either the primary or secondary PTEG by the system software, and on the next TLB miss to this page (in those processors that implement a TLB), the PTE will be found in the page tables (and loaded into an on-chip TLB).

The address of a PTEG is derived from the HTABORG field of the SDR1 register, and the output of the corresponding hashing function (primary hashing function for primary PTEG and secondary hashing function for a secondary PTEG). The value in the HTABSIZE field of SDR1 determines how many of the higher-order hash value bits are masked and how many are used in the generation of the physical address of the PTEG.

Figure 7-14 depicts the hashing functions defined by the PowerPC OEA for page tables (4KB page size). The inputs to the primary hashing function are the lower-order 39 bits of the VSID field of the STE (bits [13–51] of the 80-bit virtual address), and the page index field of the effective address (bits [52–67] of the virtual address) concatenated with 23 higher-order bits of zero. The XOR of these two values generates the output of the primary hashing function (hash value 1).

Figure 7-14. Hashing Functions for Page Tables (4KB page size)



When the secondary hashing function is required, the output of the primary hashing function is complemented with one's complement arithmetic, to provide Hash Value 2.

7.5.1.4 Translation Lookaside Buffer (TLB)

Conceptually, the page table is searched by the address relocation hardware to translate every reference. For performance reasons, the hardware usually keeps a Translation Lookaside Buffer (TLB) that holds PTEs that have recently been used. The TLB is searched prior to searching the page table. As a consequence, when software makes changes to the page table it must perform the appropriate TLB invalidate operations to maintain the consistency of the TLB with the page table.

Page table entries may or may not be cached in a TLB. It is possible that the hardware implements more than one TLB, such as one for data and one for instructions. In this case the size and shape of the TLBs may differ, as may the values contained therein. A **tlbie** or **tlbia** instruction should be used to ensure that the TLB no longer contains a mapping corresponding to an entry that has been deleted from the page table.

7.5.1.5 Page Table Address Generation

The following section illustrates the generation of the addresses used for accessing the hashed page tables. As stated earlier, the operating system must synthesize the table search algorithm for setting up the tables.

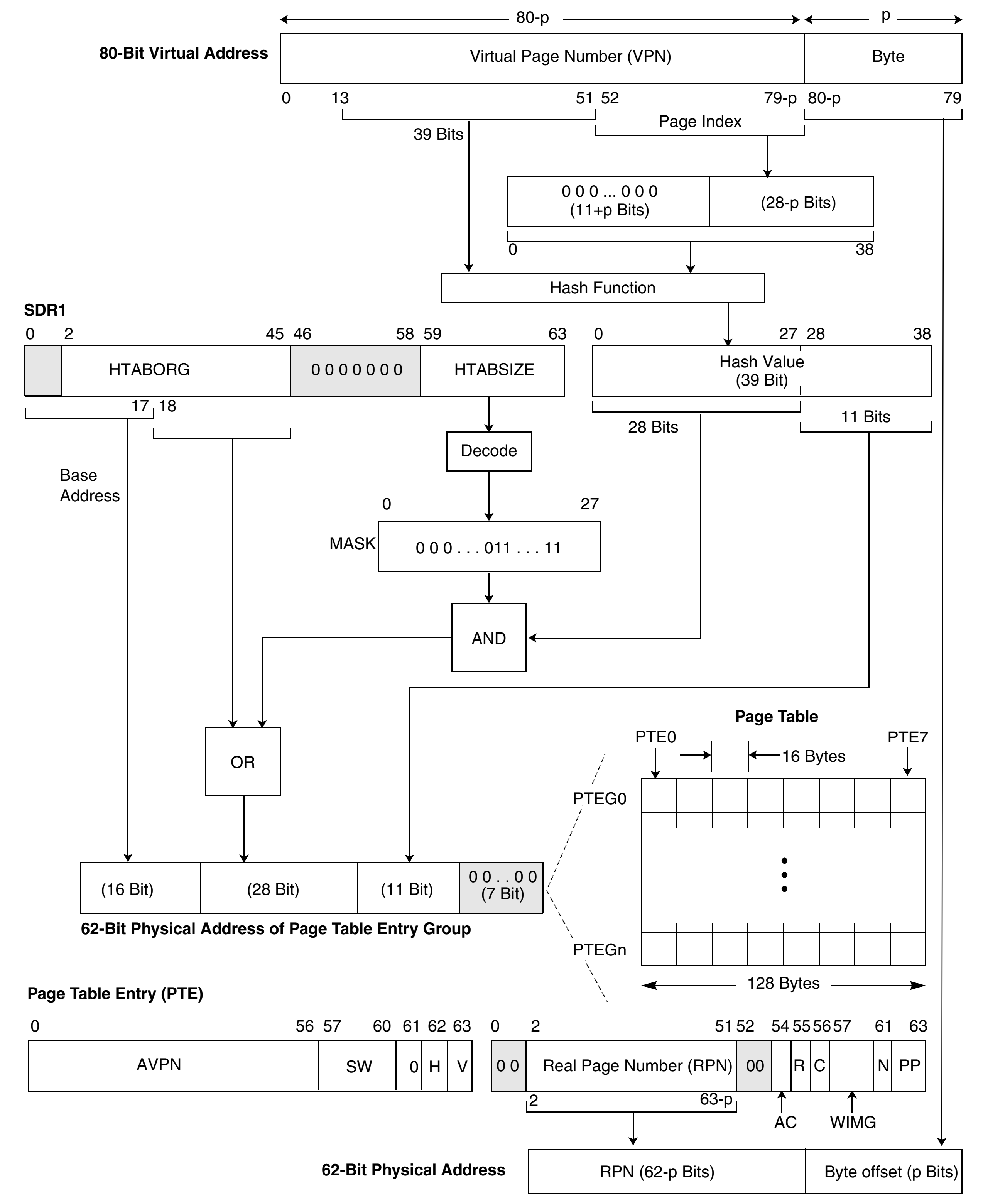
Two of the elements that define the virtual address (the VSID field of the segment descriptor and the page index field of the effective address) are used as inputs into a hashing function. Depending on whether the primary or secondary PTEG is to be accessed, the processor uses either the primary or secondary hashing function as described in *Section 7.5.1.3 Page Table Hashing Functions*.

Note: When address translation is enabled (MSR[DR] or MSR[IR] = ‘1’), the SDR1 must point to a valid page table. Otherwise, a machine check exception can occur.

Additionally, care should be given that page table addresses not conflict with those that correspond to areas of the physical address map reserved for the exception vector table or other implementation-specific purposes (refer to *Section 7.2.1.2 Predefined Physical Memory Locations*).

The base address of the page table is defined by the high-order bits of SDR1[HTABORG]. Effectively, bits [18–45] of the PTEG address are derived from the masking of the higher-order bits of the hash value (as defined by SDR1[HTABSIZE]) concatenated with (implemented as an OR function) the high-order bits of SDR1[HTABORG] as defined by HTABSIZE. Bits [46–56] of the PTEG address are the 11 lower-order bits of the hash value, and bits [57–63] of the PTEG address are zero. In the process of searching for a PTE, the processor checks up to eight PTEs located in the primary PTEG and up to eight PTEs located in the secondary PTEG, if required, searching for a match. *Figure 7-15* provides a graphical description of the generation of the PTEG addresses.

Figure 7-15. Generation of Addresses for Page Tables



7.5.1.6 Page Table Structure Summary

In the process of searching for a PTE, the processor interprets the values read from memory as described in *Section 7.4.2.2 Page Table Entry (PTE) Definition and Format*. The VSID and the abbreviated page index (API) fields of the virtual address of the access are compared to those same fields of the PTEs in memory. In addition, the valid (V) bit and the hashing function (H) bit are also checked. For a hit to occur, the V-bit of the PTE in memory must be set. If the fields match and the entry is valid, the PTE is considered a hit if the H-bit is set as follows:

- If this is the primary PTEG, H = ‘0’
- If this is the secondary PTEG, H = ‘1’

The physical address of the PTE(s) to be checked is derived as shown in *Figure 7-15*, and the generated address is the address of a group of eight PTEs (a PTEG). During a table search operation, the processor compares up to 16 PTEs: PTE0–PTE7 of the primary PTEG (defined by the primary hashing function) and PTE0–PTE7 of the secondary PTEG (defined by the secondary hashing function).

If the VSID and API fields do not match (or if V or H are not set appropriately) for any of these PTEs, a page fault occurs and an exception is taken. Thus, if a valid PTE is located in the page tables, the page is considered resident; if no matching (and valid) PTE is found for an access, the page in question is interpreted as nonresident (page fault) and the operating system must load the page into main memory and update the PTE accordingly.

The architecture does not specify the order in which the PTEs are checked. Note that for maximum performance however, PTEs should be allocated by the operating system first beginning with the PTE0 location within the primary PTEG, then PTE1, and so on. If more than eight PTEs are required within the address space that defines a PTEG address, the secondary PTEG can be used (again, allocation of PTE0 of the secondary PTEG first, and so on is recommended). Additionally, it may be desirable to place the PTEs that will require most frequent access at the beginning of a PTEG and reserve the PTEs in the secondary PTEG for the least frequently accessed PTEs.

The architecture also allows for multiple matching entries to be found within a table search operation. Multiple matching PTEs are allowed if they meet the match criteria described above, as well as have identical RPN, WIMG, and PP values, allowing for differences in the R and C-bits. In this case, one of the matching PTEs is used and the R and C-bits are updated according to this PTE. In the case that multiple PTEs are found that meet the match criteria but differ in the RPN, WIMG, or PP fields, the translation is undefined and the resultant R and C-bits in the matching entries are also undefined.

Note: Multiple matching entries can also differ in the setting of the H-bit, but the H-bit must be set according to whether the PTE was located in the primary or secondary PTEG, as described above.

7.5.1.7 Page Table Structure Example

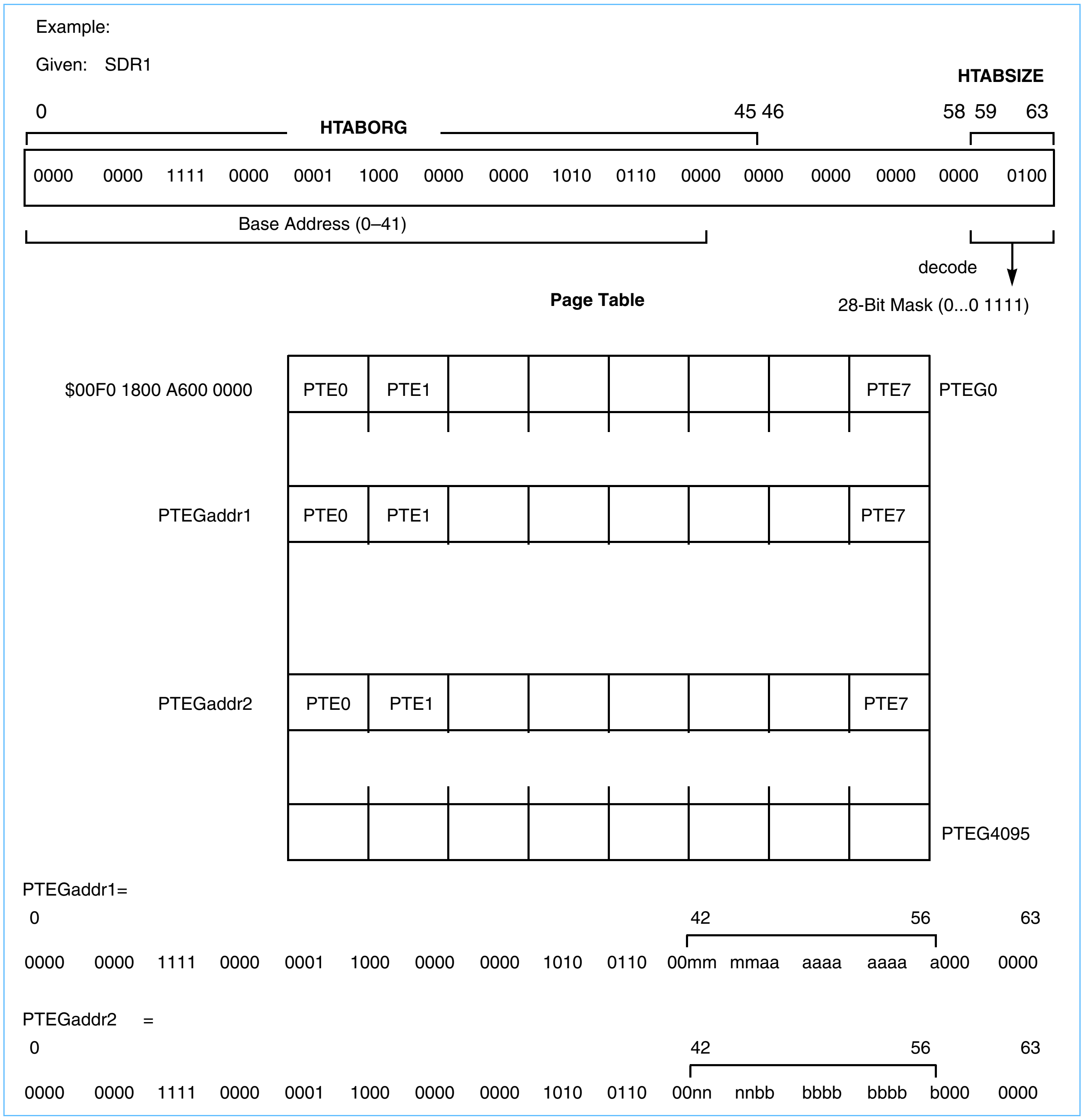
Figure 7-16 shows the structure of an example page table. The base address of the page table is defined by SDR1[HTABORG] concatenated with 18 zero bits. In this example, the address is identified by bits [0–41] in SDR1[HTABORG]; note that bits [42–45] of HTABORG must be zero because the HTABSIZE field specifies an integer mask size of four, which decodes to four mask bits of ones. The addresses for individual PTEGs within this page table are then defined by bits [42–56] as an offset from bits [0–41] of this base address. Thus, the size of the page table is defined as 0x7FFF (32K) PTEGs.

Two example PTEG addresses are shown in *Figure 7-16* as PTEGaddr1 and PTEGaddr2. Bits [42–56] of each PTEG address in this example page table are derived from the output of the hashing function (bits [57-63] are zero to start with PTE0 of the PTEG). In this example, the ‘b’ bits in PTEGaddr2 are the one’s complement of the ‘a’ bits in PTEGaddr1. The ‘n’ bits are also the one’s complement of the ‘m’ bits, but these four bits are generated from bits [24–27] of the output of the hashing function, logically ORed with bits [42–45] of the HTABORG field (which must be zero). If bits [42–56] of PTEGaddr1 were derived by using the primary hashing function, PTEGaddr2 corresponds to the secondary PTEG.

Note: Bits [42–56] in PTEGaddr2 can also be derived from a combination of effective address bits, segment descriptor bits, and the primary hashing function. In this case, then PTEGaddr1 corresponds to the secondary PTEG. Thus, while a PTEG may be considered a primary PTEG for some effective addresses (and segment descriptor bits), it may also correspond to the secondary PTEG for a different effective address (and segment descriptor value).

It is the value of the H-bit in each of the individual PTEs that identifies a particular PTE as either primary or secondary (there may be PTEs that correspond to a primary PTEG and PTEs that correspond to a secondary PTEG, all within the same physical PTEG address space). Thus, only the PTEs that have H = ‘0’ are checked for a hit during a primary PTEG search. Likewise, only PTEs with H = ‘1’ are checked in the case of a secondary PTEG search.

Figure 7-16. Example Page Table Structure



7.5.1.8 PTEG Address Mapping Example

This section contains a sample effective address and how its address translation (the PTE) maps into the primary PTEG in physical memory. This example illustrates how the processor generates PTEG addresses for a table search operation; this is also the algorithm that must be used by the operating system in creating page tables.

In the example shown in *Figure 7-17*, the value in SDR1 defines a page table at address 0x0F05_8400_0F00_0000 that contains 2^{17} PTEGs. The highest order 36 bits of the effective address uniquely map to a segment descriptor. The segment descriptor is then located and the contents of the segment descriptor are used along with bits [36–63] of the effective address to create the 80-bit virtual address.

To generate the address of the primary PTEG, bits [13–51], and bits [52–67] of the virtual address are then used as inputs into the primary hashing function (XOR) to generate hash value 1. The low-order 17 bits of hash value 1 are then concatenated with the high-order 40 bits of HTABORG and with seven low-order ‘0’ bits, defining the address of the primary PTEG (0x0F05_8400_0F3F_F300). The ANDing of the 28 high-order bits of hash value 1 with the mask (defined by the HTABSIZE field) and the ORing with bits [18–45] of HTABORG are implicitly shown in the figure. The ANDing with the mask selects six additional bits of hash value 1 to be used (in addition to the 11 prescribed bits) producing a total of 17 bits of hash value 1 bits to be used. The ORing causes those selected six bits of hash value 1 to comprise bits [40–45] of the PTEG address (as bits [40–45] of HTABORG should be zero).

Figure 7-17. Example Primary PTEG Address Generation

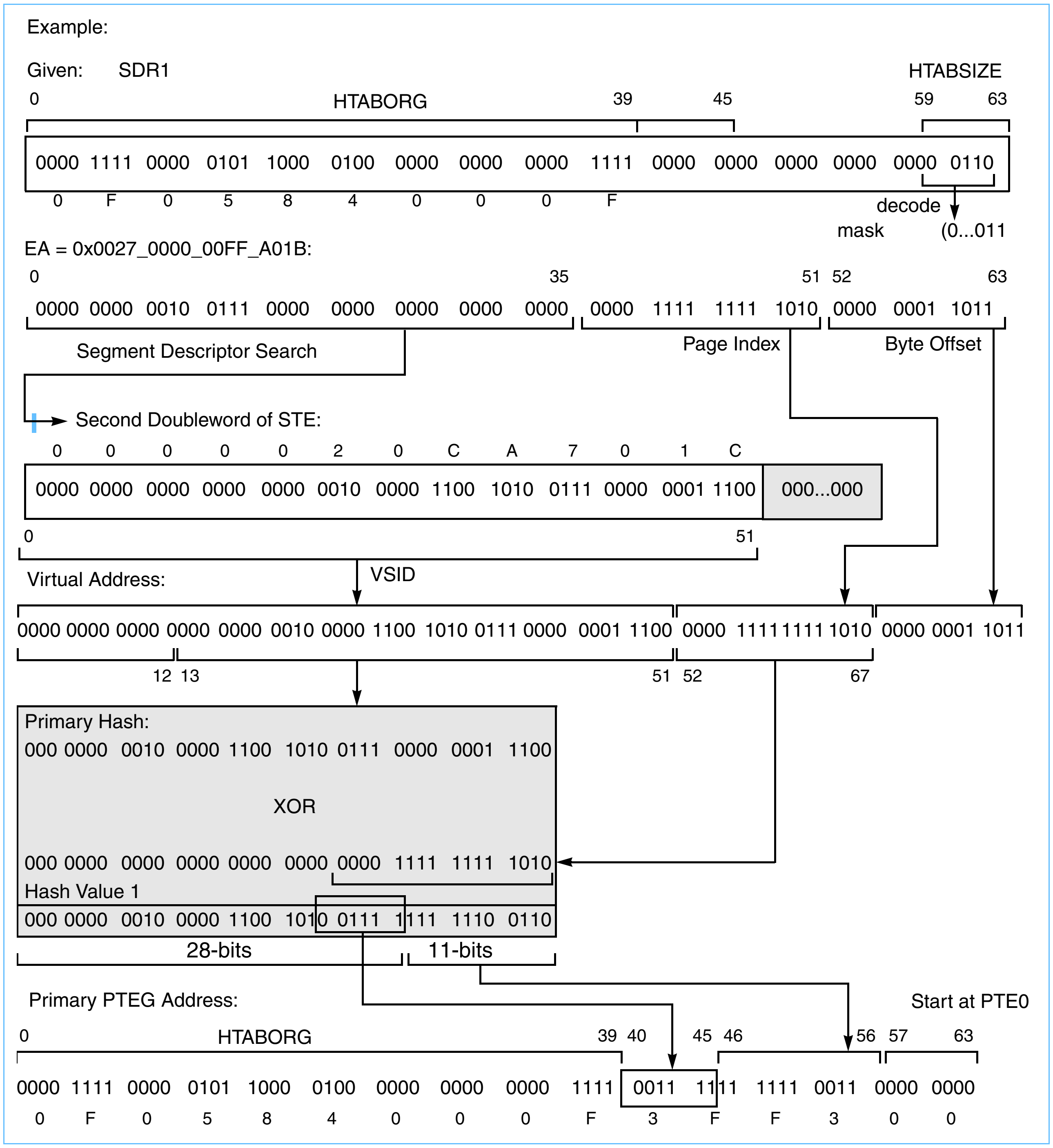
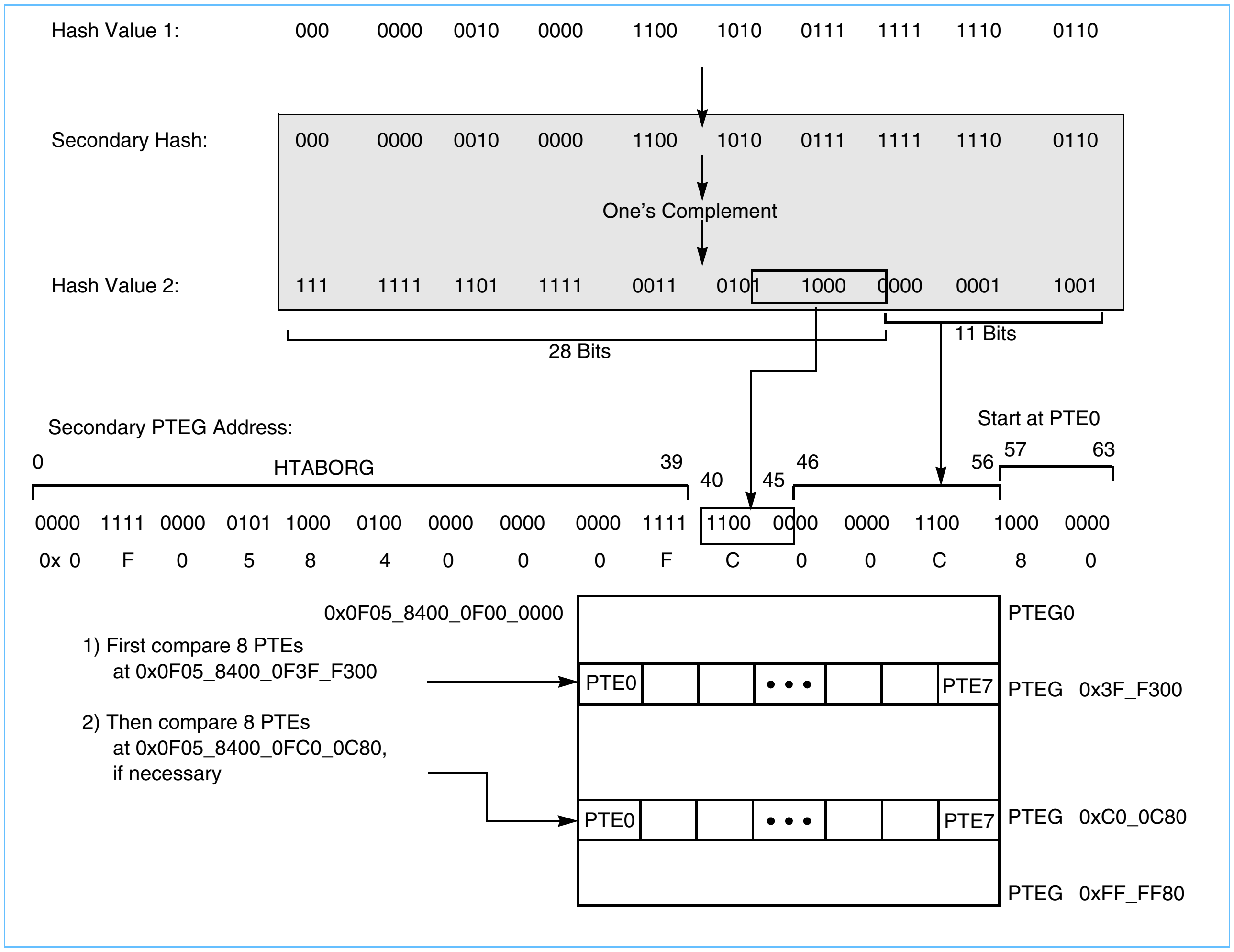


Figure 7-18 shows the generation of the secondary PTEG address for this example. If the secondary PTEG is required, the secondary hash function is performed and the low-order 17 bits of hash value 2 are then ORed with the high-order 46 bits of HTABORG (bits [40–45] should be zero), and concatenated with seven low-order 0 bits, defining the address of the secondary PTEG (0x0F05_8400_0FC0_0C80).

As described in *Figure 7-15*, the 11 low-order bits of the page index field are always used in the generation of a PTEG address (through the hashing function). This is why only the 5-bit abbreviated page index (API) is defined for a PTE (the entire page index field does not need to be checked). For a given effective address, the low-order 11 bits of the page index (at least) contribute to the PTEG address (both primary and secondary) where the corresponding PTE may reside in memory. Therefore, if the high-order 5 bits (the API field) of the page index match with the API field of a PTE within the specified PTEG, the PTE mapping is guaranteed to be the unique PTE required.

Figure 7-18. Example Secondary PTEG Address Generation



Note: A given PTEG address does not map back to a unique effective address. Not only can a given PTEG be considered both a primary and a secondary PTEG (as described in *Section 7.5.1.7 Page Table Structure Example*), but if the mask defined has four '1' bits or less (not the case shown in the example in the figure), some bits of the page index field of the virtual address are not used to generate the PTEG address. Therefore, any combination of these unused bits will map to the same pair of PTEG addresses. (However, these bits are part of the API and are therefore compared for each PTE within the PTEG to determine if there is a hit.) Furthermore, an effective address can select a different segment descriptor with a different value such that the output of the primary (or secondary) hashing function happens to equal the hash values shown in the example. Thus, these effective addresses would also map to the same PTEG addresses shown.

7.5.2 Page Table Search Process

An outline of the page table search process is as follows:

1. The 64-bit physical addresses of the primary and secondary PTEGs are generated as described in *Section 7.5.1.5 Page Table Address Generation* on page 279.
2. As many as 16 PTEs (from the primary and secondary PTEGs) are read from memory (the architecture does not specify the order of these reads, allowing multiple reads to occur in parallel). PTE reads occur with an implied WIM memory/cache mode control bit setting of ‘001’. Therefore, they are considered cacheable.
3. The PTEs in the selected PTEGs are tested for a match with the virtual page number (VPN) of the access. The VPN is the VSID concatenated with the page index field of the virtual address. For a match to occur, the following must be true:
   - PTE[H] = ‘0’ for primary PTEG; PTE[H] = ‘1’ for secondary PTEG
   - PTE[V] = ‘1’
   - PTE[VSID] = VA[0-51]
   - PTE[API] = VA[52-56]
4. If a match is not found within the eight PTEs of the primary PTEG and the eight PTEs of the secondary PTEG, an exception is generated as described in step 8. If a match (or multiple matches) is found, the table search process continues.
5. If multiple matches are found, all of the following must be true:
   - PTE[RPN] is equal for all matching entries
   - PTE[WIMG] is equal for all matching entries
   - PTE[PP] is equal for all matching entries
6. If one of the fields in step 5 does not match, the translation is undefined, and R and C-bit of matching entries are undefined. Otherwise, the R and C-bits are updated based on one of the matching entries.
7. A copy of the PTE is written into the on-chip TLB (if implemented) and the R-bit is updated in the PTE in memory (if necessary). If there is no memory protection violation, the C-bit is also updated in memory (if necessary) and the table search is complete.
8. If a match is not found within the primary or secondary PTEG, the search fails, and a page fault exception condition occurs (either an ISI or DSI exception).

Reads from memory for page table search operations are performed as if the WIMG bit settings were ‘0010’ (that is, as unguarded cacheable operations in which coherency is required).

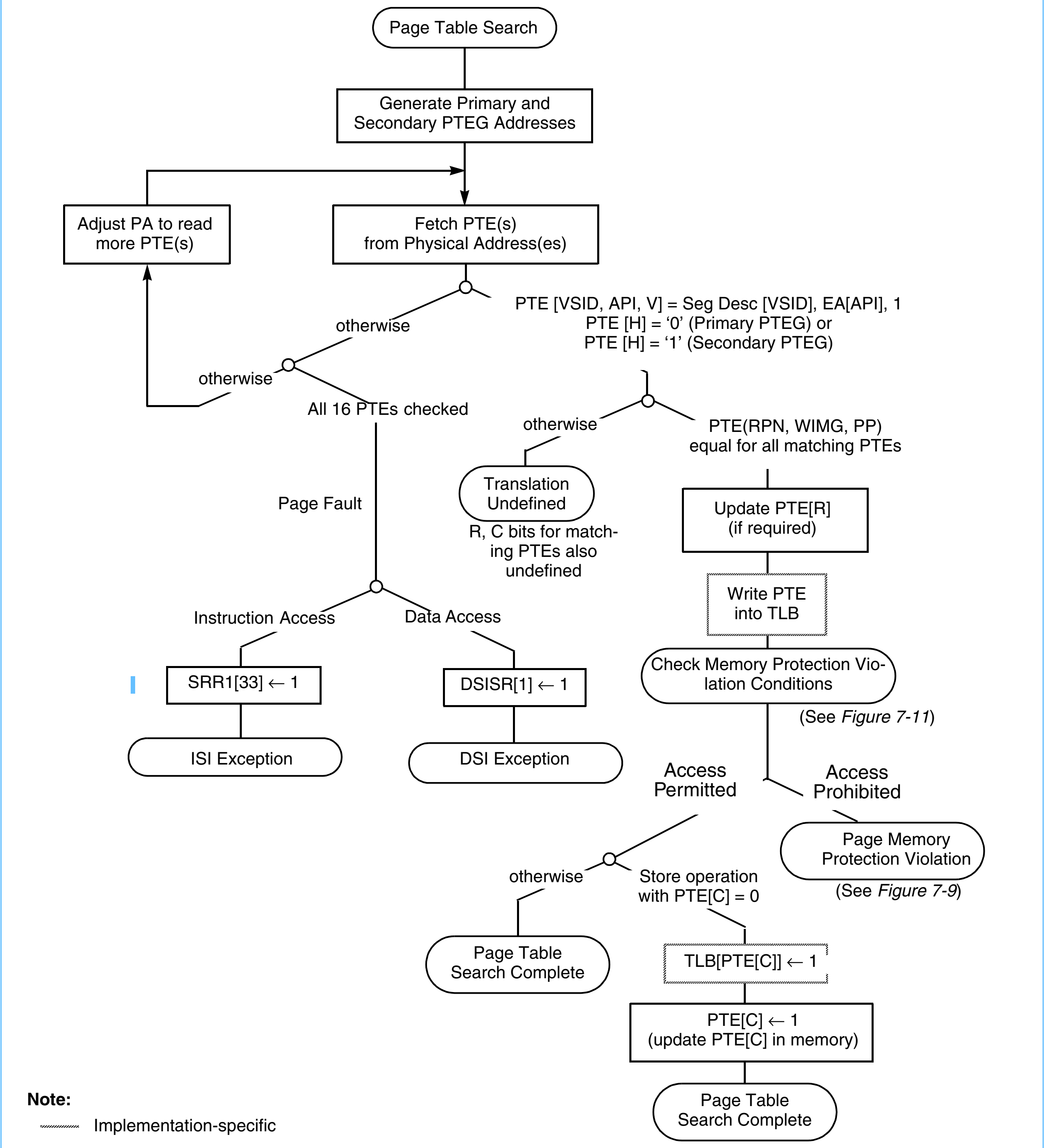
7.5.2.1 Flow for Page Table Search Operation

Figure 7-19 Page Table Search Flow provides a detailed flow diagram of a page table search operation.

Note: The references to TLBs are shown as optional because TLBs are not required; if they do exist, the specifics of how they are maintained are implementation-specific.

Figure 7-19 shows only a few cases of R-bit and C-bit updates. For a complete list of the R and C-bit updates dictated by the architecture, refer to *Table 7-10 Model for Guaranteed R and C Bit Settings*.

Figure 7-19. Page Table Search Flow


Page Table Search
Generate Primary and Secondary PTEG Addresses
Adjust PA to read more PTE(s)
Fetch PTE(s) from Physical Address(es)
otherwise
PTE [VSID, API, V] = Seg Desc [VSID], EA[API], 1
PTE [H] = '0' (Primary PTEG) or
PTE [H] = '1' (Secondary PTEG)
otherwise
All 16 PTEs checked
otherwise
PTE(RPN, WIMG, PP) equal for all matching PTEs
Page Fault
Translation Undefined
R, C bits for matching PTEs also undefined
Update PTE[R] (if required)
Write PTE into TLB
Instruction Access
Data Access
Check Memory Protection Violation Conditions
(See Figure 7-11)
SRR1[33] ← 1
DSISR[1] ← 1
ISI Exception
DSI Exception
Access Permitted
Access Prohibited
Page Memory Protection Violation
(See Figure 7-9)
otherwise
Store operation with PTE[C] = 0
Page Table Search Complete
TLB[PTE[C]] ← 1
PTE[C] ← 1 (update PTE[C] in memory)
Page Table Search Complete
Note:
Implementation-specific

7.5.3 Page Table Updates

This section describes the requirements on the software when updating page tables in memory via some pseudocode examples. Multiprocessor systems must follow the rules described in this section so that all processors operate with a consistent set of page tables. Even single processor systems must follow certain rules, because software changes must be synchronized with the other instructions in execution and with automatic updates that may be made by the hardware (referenced and changed bit updates). Updates to the tables include the following operations:

- Adding a PTE
- Modifying a PTE, including modifying the R and C-bits of a PTE
- Deleting a PTE

PTEs must be locked on multiprocessor systems. Access to PTEs must be appropriately synchronized by software locking of (that is, guaranteeing exclusive access to) PTEs or PTEGs if more than one processor can modify the table at that time. In the examples below, software locks should be performed to provide exclusive access to the PTE being updated. However, the architecture does not dictate the specific protocol to be used for locking (for example, a single lock, a lock per PTEG, or a lock per PTE can be used). See *Appendix D Synchronization Programming Examples* for more information about the use of the reservation instructions (such as the **lwarx** and **stwcx.** instructions) to perform software locking.

When TLBs are implemented they are defined as noncoherent caches of the page tables. TLB entries must be invalidated explicitly with the TLB invalidate entry instruction (**tlbie**) whenever the corresponding PTE is modified. In a multiprocessor system, the **tlbie** instruction must be controlled by software locking, so that the **tlbie** is issued on only one processor at a time.

The PowerPC OEA defines the **tlbsync** instruction that ensures that TLB invalidate operations executed by this processor have caused all appropriate actions in other processors. In a system that contains multiple processors, the **tlbsync** functionality must be used in order to ensure proper synchronization with the other PowerPC processors.

Note: A **sync** (or **ptesync**) instruction must also follow the **tlbsync** to ensure that the **tlbsync** has completed execution on this processor.

On single processor systems, PTEs need not be locked and the **eieio** instructions (in between the **tlbie** and **tlbsync** instructions) and the **tlbsync** instructions themselves are not required. The **sync** instructions shown are required even for single processor systems (to ensure that all previous changes to the page tables and all preceding **tlbie** instructions have completed).

Any processor, including the processor modifying the page table, may access the page table at any time in an attempt to reload a TLB entry. An inconsistent PTE must never accidentally become visible (if V = ‘1’); thus, there must be synchronization between modifications to the valid bit and any other modifications (to avoid corrupted data).

In the pseudocode examples that follow, changes made to a PTE or STE shown as a single line in the example is assumed to be performed with an atomic store instruction. Appropriate modifications must be made to these examples if this assumption is not satisfied (for example, if a store doubleword operation is performed with two store word instructions).

Updates of R and C-bits by the processor are not synchronized with the accesses that cause the updates. When modifying the low-order half of a PTE, software must take care to avoid overwriting a processor update of these bits and to avoid having the value written by a store instruction overwritten by a processor update. The processor does not alter any other fields of the PTE.

Explicitly altering certain MSR bits (using the **mtmsrd** instruction), or explicitly altering STEs, PTEs, or certain system registers, may have the side effect of changing the effective or physical addresses from which the current instruction stream is being fetched. This kind of side effect is defined as an implicit branch. For example, an **mtmsrd** instruction may change the value of MSR[SF], changing the effective addresses from which the current instruction stream is being fetched, causing an implicit branch. Implicit branches are not supported and an attempt to perform one causes boundedly-undefined results. Therefore, PTEs and STEs must not be changed in a manner that causes an implicit branch. *Section 2.3.16 Synchronization Requirements for Special Registers and for Lookaside Buffers* lists the possible implicit branch conditions that can occur when system registers and MSR bits are changed.

For a complete list of the synchronization requirements for executing the MMU instructions, see *Section 2.3.16 Synchronization Requirements for Special Registers and for Lookaside Buffers*.

The following examples show the required sequence of operations. However, other instructions may be interleaved within the sequences shown.

7.5.3.1 Adding a Page Table Entry

This is the simplest page table case. The valid bit of the old entry is assumed to be ‘0’. The following sequence can be used to create a PTE, maintain a consistent state, and ensure that a subsequent reference to the virtual address translated by the new entry will use the correct real address and associated attributes.

```
PTE[RPN,AC,R,C,WIMG,N,PP] ← new values
eieio                    /* order 1st update before 2nd */
PTE[AVPN,SW,H,V] ← new values (V = 1)
ptesync                  /* order updates before next page table search and before next data access */
```

7.5.3.2 Modifying a Page Table Entry

This section describes several scenarios for modifying a PTE.

General Case

If a valid entry is to be modified and the translation instantiated by the entry being modified is to be invalidated, the following sequence can be used to modify the PTE, maintain a consistent state, ensure that the translation instantiated by the old entry is no longer available, and ensure that a subsequent reference to the virtual address translated by the new entry will use the correct real address and associated attributes. (The sequence is equivalent to deleting the PTE and then adding a new one.)

```
PTE[V] ← 0                      /* (other fields don’t matter) */
ptesync                         /* order updated before tlbie and before next page table search */
tblie (old_VPN[32-79-p, OLD_L] /* invalidate old translation */
eieio                           /* order tlbie before tlbsync */
tlbsync                         /* order tblie before ptesync */
ptesync                         /* order tlbie, tlbsync and first update before second update */
PTE[RPN,AC,R,C,WIMG,N,PP] ← new values
eieio                           /* order second update before third */
PTE[AVPN,SW,H,V] ← new values (V = ‘1’)
ptesync                         /* order second and third updates before next page table search and before
                                next data access */
```

Clearing the Referenced (R) Bit

When the PTE is modified only to clear the R bit to ‘0’, a much simpler algorithm suffices because the R-bit need not be maintained exactly.

```
lock(PTE)
oldR ←PTE[R]           /*get old R */
if oldR = 1, then
PTE[R] ← 0             /* store byte (R = ‘0’, other bits unchanged) */
tlbie(PTE)             /* invalidate entry */
eieio                  /* order tlbie before tlbsync */
tlbsync                /* ensure tlbie completed on all processors */
ptesync                /* order tlbie, tlbsync, and update before next page table
                       search and before next data access */
unlock(PTE)
```

Since only the R and C-bits are modified by the processor, and since they reside in different bytes, the R-bit can be cleared by reading the current contents of the byte in the PTE containing R (bits [48–55] of the second doubleword), ANDing the value with ‘FE’, and storing the byte back into the PTE.

Modifying the Virtual Address

If the virtual address translated by a valid PTE is to be modified and the new virtual address hashes to the same two PTEGs as does the old virtual address, the following sequence can be used to modify the PTE, maintain a consistent state, ensure that the translation instantiated by the old entry is no longer available, and ensure that a subsequent reference to the virtual address translated by the new entry uses the correct real address and associated attributes.

```
PTE[AVPN,SW,H,V] ← new values (V = 1)
ptesync                /* order update before tblie and before next page table search */
tlbie(old_EA)          /* invalidate old translation */
eieio                  /* order tlbie before tlbsync */
tlbsync                /* order tlbie before ptesync */
ptesync                /* order tlbie, tlbsync and update before next data access */
unlock(PTE)
```

To modify the AC, N, or PP bits without overwriting a reference or change bit update being performed by the processor or by some other processor, a sequence similar to that shown above can be used except that the first line would be replaced by a **ptesync** instruction followed by a loop containing a **ldarx**/**stdcx.** pair that emulates an atomic “Compare and Swap” of the low-order doubleword of the PTE.

7.5.3.3 Deleting a Page Table Entry

The following sequence can be used to ensure that the translation instantiated by an existing entry is no longer available.

```
lock(PTE)
PTE[V] ← 0          /* (other fields don't matter) */
ptesync             /* order update before tlbie and before next page table search */
tlbie(old_EA)       /* invalidate old translation */
eieio               /* order tlbie before tlbsync */
tlbsync             /* order tlbie before ptesync */
ptesync             /* order tlbie, tlbsync and update before next data access */
unlock(PTE)
```

7.5.4 ASR Updates

There are certain synchronization requirements for writing to the ASR or using the move to segment register instructions. These are described in *Section 2.3.16 Synchronization Requirements for Special Registers and for Lookaside Buffers*.

7.6 Migration of Operating Systems from 32-Bit Implementations to 64-Bit Implementations

The facilities and instructions described in this section may optionally be provided by a 64-bit implementation to reduce the amount of software change required to migrate an operating system from a 32-bit implementation to a 64-bit implementation. Using the bridge facility allows the operating system to treat the MSR as a 32-bit register and to continue to use the segment register manipulation instructions (**mtsr**, **mtsrin**, **mfsr**, and **mfsrin**) which are defined for 32-bit implementations. These instructions are otherwise illegal in the 64-bit architecture. Although the 64-bit bridge does not literally implement the 16 registers as they are defined by the 32-bit portion of the architecture, the segment register manipulation instructions are used to access the 16 predefined segment descriptors stored in the on-chip SLBs.

The bridge features do not conceal the differences in format of the page table and SDR1 between 32-bit and 64-bit implementations—the operating system must be converted explicitly to use the 64-bit formats.

Note: An operating system that uses the bridge features does not take full advantage of the 64-bit implementation (for example, it can generate only 32-bit effective addresses).

An operating system that uses the 64-bit bridge architecture should observe the following:

- The boot process should do the following:
  - Clear MSR[SF].
  - Initialize the ASR, clearing ASR[V].
  - Invalidate all SLB entries.
- The operating system should do the following:
  - Support only 32-bit applications.
  - If any 64-bit instructions are used, for example, to modify a PTE or a 64-bit SPR, ensure either that exceptions cannot occur or that the exception handler saves and restores all 64 bits of the GPRs.

– Manipulate only the low-order 32 bits of the MSR, leaving the high-order 32 bits unchanged.

– Always have ASR[V] = 0.

– Manage virtual segments using the 32-bit segment register manipulation instructions (**mtsr**, **mtsrin**, **mfsr**, and **mfsrin**).

– Always map segments 0–15 in the SLB when translation is enabled. They may be mapped with a VSID for which there are no valid PTEs.

– Never execute an **slbie** or **slbia** instruction.

– Never generate an effective address greater than 232 – 1 when MSR[SF] = ‘1’.

7.6.1 Segment Register Manipulation Instructions in the 64-Bit Bridge

The four segment register manipulation instructions, **mtsr**, **mtsrin**, **mfsr**, and **mfsrin**, defined as part of the 32-bit portion of the architecture may optionally be provided by a 64-bit implementation that uses the 64-bit bridge. As part of the 64-bit bridge, these instructions operate as described below, and are implemented as a group and not individually. Attempting to execute one of these instructions on a 64-bit processor on which it is not supported causes an illegal instruction type program exception.

These instructions allow software to associate effective segments 0 through 15 with any of virtual segments 0 through 224 – 1 without altering the segment table in memory. Sixteen indexed SLB entries serve as virtual segment registers. The **mtsr** and **mtsrin** instructions move 32 bits from a selected GPR to a selected SLB entry. The **mfsr** and **mfsrin** instructions move 64 bits from a selected SLB entry to a selected GPR and can be used to read an SLB entry that was created with **mtsr** or **mtsrin**.

The software synchronization requirements for any of the move to segment register instructions in a 64-bit implementation are the same as for those defined by the 32-bit architecture.

To ensure that SLB entries contain unique ESIDs when the bridge is used, an ESID mapped by any of the move to segment register instructions must not have been mapped to that SLB entry by the segment table when ASR[V] was set.

If an SLB entry that software established using one of the move to segment register instructions is overwritten while ASR[V] = ‘1’, software must be able to handle any exception caused when a segment descriptor cannot be located.

Executing an **mfsr** or **mfsrin** instruction may set **r**D to an undefined value if ASR[V] has been set at any time since execution of the **mtsr** or **mtsrin** instruction that established the selected SLB entry, because that SLB entry may have been overwritten by the processor in the meantime.

Typically, 16 fixed SLB entries are used by the segment register manipulation instructions, while SLB reload from the segment table selects SLB entries based on some other replacement policy such as LRU.

With respect to updating any SLB replacement history used by the SLB replacement policy, implementations will treat the execution of an **mtsr** or **mtsrin** instruction the same as an SLB reload from the segment table.

The following sections describe the move to and move from segment register instructions as they are defined for the 64-bit bridge.

7.6.2 64-Bit Bridge Implementation of Segment Register Instruction

The following sections describe the **mfsr**, **mfsrin**, **mtsr**, and **mtsrin** instructions that are defined for the 32-bit architecture and are allowed in the 64-bit bridge architecture only if ASR[V] is implemented. Otherwise, attempting to execute one of these instructions is illegal on a 64-bit implementation.

7.6.2.1 Move from Segment Register—mfsr

The **mfsr** instruction syntax is as follows:

```
mfsr rD,SR
```

The operation of the instruction is described as follows:

```
rD ← SLB(SR)
```

When executed as part of the 64-bit bridge, the contents of the SLB entry selected by SR are placed into **r**D; the contents of **r**D correspond to a segment table entry containing values as shown in *Table 7-16*.

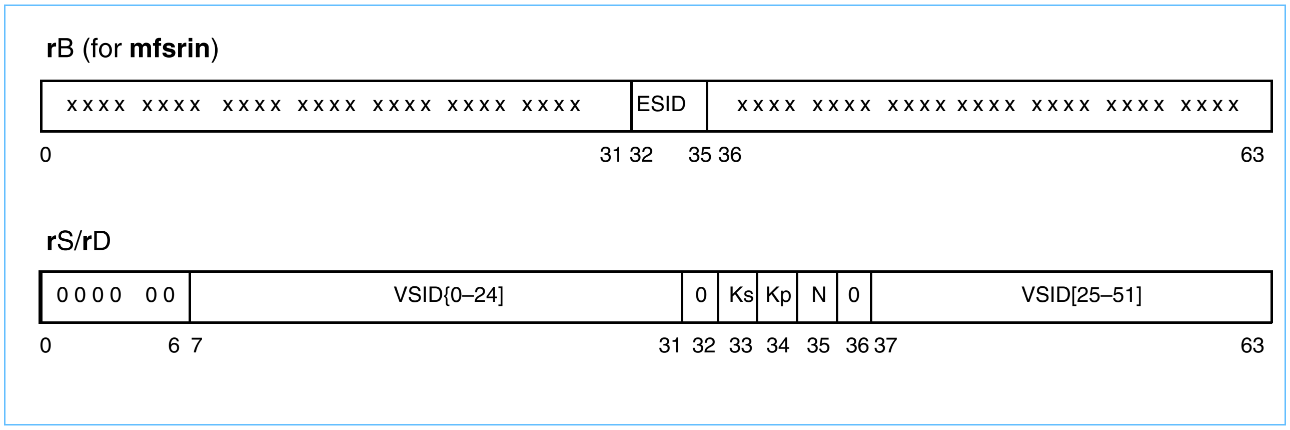
Table 7-16. Contents of rD after Executing ***mfsr***

Doubleword	Bit(s)	Contents	Description
0	0–31	0x0000_0000	ESID[0–31]
	32–35	SR	ESID[32–35]
	36–57	—	—
	58–59	**r**D[33–34]	Ks, Kp
	60–61	**r**D[35–36]	N, reserved bit, or ‘0’
	62–63	—	—
1	0–24	**r**D[7–31]	VSID[0–24]
	25–51	**r**D[37–63]	VSID[25–51]
	52–63	—	—
Note: The contents of **r**D[0–6] are cleared automatically.			

If the SLB entry selected by SR was not created by an **mtsr** instruction, the contents of **r**D are undefined. Formatting for GPR contents is shown in *Figure 7-20*. Fields shown as x’s are ignored. Fields shown as slashes correspond to reserved bits in the segment table entry.

This is a supervisor-level instruction.

*Figure 7-20. GPR Contents for **mfsr** and **mfsrin***



7.6.2.2 Move from Segment Register Indirect—mfsrin

The **mfsrin** instruction syntax is as follows:

```
mfsrin rD,rB
```

The operation of the instruction is described as follows:

```
rD ← SLB(rB[32–35])
```

The contents of the SLB entry selected by **r**B[32–35] are placed into **r**D; the contents of **r**D correspond to a segment table entry containing values as shown in *Table 7-17*.

*Table 7-17. Contents of rD after Executing **mtsr***

Doubleword	Bit(s)	Contents	Description
0	0–31	0x0000_0000	ESID[0–31]
	32–35	**r**B[32–35]	ESID[32–35]
	36–57	—	—
	58–59	**r**D[33–34]	Ks, Kp
	60–61	**r**D[35–36]	N, reserved bit, or ‘0’
1	0–24	**r**D[7–31]	VSID[0–24]
	25–51	**r**D[37–63]	VSID[25–51]
	52–63	—	—
Note: The contents of **r**D[0–6] are cleared automatically.			

If the SLB entry selected by **r**B[32–35] was not created by an **mtsr** instruction, the contents of **r**D are undefined. Formatting for GPR contents is shown in *Figure 7-20*. Fields shown as x’s are ignored. Fields shown as slashes correspond to reserved bits in the segment table entry.

This is a supervisor-level instruction.

7.6.2.3 Move to Segment Register—mtsr

The **mtsr** instruction syntax is as follows:

```
mtsr SR,rS
```

The operation of the instruction is described as follows:

```
SLB(SR) ← (rS[32–63])
```

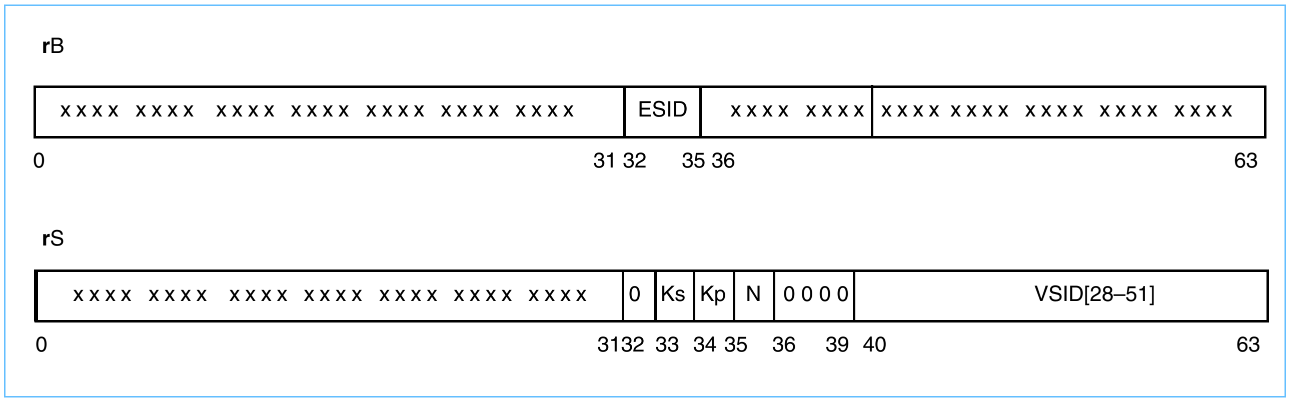
The SLB entry selected by SR is set as though it were loaded from a segment table entry, as shown in *Table 7-18*.

Table 7-18. SLB Entry selected by SR

Doubleword	Bit(s)	Contents	Description
0	0–31	0x0000_0000	ESID[0–31]
	32–35	SR	ESID[32–35]
	36–55	—	—
	56	0b1	V
	57	—	—
	58–59	**r**S[33–34]	Ks, Kp
	60–61	**r**S[35–36]	N, reserved bit, or ‘0’
	62–63	—	—
1	0–24	0x0000_00‖0b0	VSID[0–24]
	25–51	**r**S[37–63]	VSID[25–51]
	51–63	—	—

This is a supervisor-level instruction. Formatting for GPR contents is shown in *Figure 7-21*. Fields shown as x’s are ignored. Fields shown as slashes correspond to reserved bits in the segment table entry.

*Figure 7-21. GPR Contents for **mtsr** and **mtsrin***



Note: When creating a memory segment using the **mtsr** instruction, **r**S[36–39] should be cleared.

7.6.2.4 Move to Segment Register Indirect—mtsrin

The **mtsrin** instruction syntax is as follows:

```
mtsrin rS,rB
```

The operation of the instruction is described as follows:

```
SLB(rB[32–35]) ← (rS[32–63])
```

The SLB entry selected by bits [32–35] of **r**B is set as though it were loaded from a segment table entry, as shown in *Table 7-19*.

Table 7-19. SLB Entry Selected by Bits [32-35] or ***r****B*

Doubleword	Bit(s)	Contents	Description
0	0–31	0x0000_0000	ESID[0–31]
	32–35	**r**B[32–35]	ESID[32–35]
	36–55	—	—
	56	0b1	V
	57	—	—
	58–59	**r**S[33–34]	Ks, Kp
	60–61	**r**S[35–36]	N, reserved bit, or '0'
	62–63	—	—
1	0–24	0x0000_00ǁ0b0	VSID[0–24]
	25–51	**r**S[37–63]	VSID[25–51]
	52–63	—	—

This is a supervisor-level instruction. Formatting for GPR contents is shown in *Figure 7-21*. Fields shown as x's are ignored. Fields shown as slashes correspond to reserved bits in the segment table entry.

Note: When creating a memory segment using the **mtsrin** instruction, **r**S[36–39] should be cleared.

THIS PAGE INTENTIONALLY LEFT BLANK

8. Instruction Set

This chapter lists the PowerPC instruction set in alphabetical order by mnemonic and the instruction format. The format diagrams show, horizontally, all valid combinations of instruction fields; for a graphical representation of these instruction formats, see *Appendix A PowerPC Instruction Set Listings*. A description of the instruction fields and pseudocode conventions are also provided.

For more information on the PowerPC instruction set, refer to *Chapter 4, Addressing Modes and Instruction Set Summary*.

Note: The architecture specification refers to user-level and supervisor-level as problem state and privileged state, respectively.

8.1 Instruction Formats

Instructions are four bytes long and word-aligned, so when instruction addresses are presented to the processor (as in branch instructions) the two low-order bits are ignored. Similarly, whenever the processor develops an instruction address, its two low-order bits are zero.

Bits [0–5] always specify the primary opcode. Many instructions also have an extended opcode. The remaining bits of the instruction contain one or more fields for the different instruction formats.

Some instruction fields are reserved or must contain a predefined value as shown in the individual instruction layouts. If a reserved field does not have all bits cleared, 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 *Chapter 4, Addressing Modes and Instruction Set Summary*.

Within the instruction format diagram the instruction operation code and extended operation code (if extended form) are specified in decimal. These fields have been converted to hexadecimal and are shown on line two for each instruction definition.

8.1.1 Split-Field Notation

Some instruction fields occupy more than one contiguous sequence of bits or occupy a contiguous sequence of bits used in permuted order. Such a field is called a split field. Split fields that represent the concatenation of the sequences from left to right are shown in lowercase letters. These split fields—mb, me, sh, spr, and tbr—are described in *Table 8-1*.

Table 8-1. Split-Field Notation and Conventions

Field	Description
mb (21–26)	This field is used in rotate instructions to specify the first 1 bit of a 64-bit mask, as described in *Section 4.2.1.4 Integer Rotate and Shift Instructions*. This field is defined in 64-bit implementations only.
me (21–26)	This field is used in rotate instructions to specify the last 1 bit of a 64-bit mask, as described in *Section 4.2.1.4 Integer Rotate and Shift Instructions*. This field is defined in 64-bit implementations only.
sh (16–20) and sh (30)	These fields are used to specify a shift amount (64-bit implementations only).
spr (11–20)	This field is used to specify a special-purpose register for the **mtspr** and **mfspr** instructions. The encoding is described in *Section 4.4.2.2 Move to/from Special-Purpose Register Instructions (OEA)*.
tbr (11–20)	This field is used to specify either the time base lower (TBL) or time base upper (TBU).

Split fields that represent the concatenation of the sequences in some order, which need not be left to right (as described for each affected instruction), are shown in uppercase letters. These split fields—MB, ME, and SH—are described in *Table 8-2*.

8.1.2 Instruction Fields

Table 8-2 describes the instruction fields used in the various instruction formats.

Table 8-2. Instruction Syntax Conventions

Field	Description
AA (30)	Absolute address bit. 0 The immediate field represents an address relative to the current instruction address (CIA). (For more information on the CIA, see *Table 8-3*.) The effective (logical) address of the branch is either the sum of the LI field sign-extended to 64 bits and the address of the branch instruction or the sum of the BD field sign-extended to 64 bits and the address of the branch instruction. 1 The immediate field represents an absolute address. The effective address (EA) of the branch is the LI field sign-extended to 64 bits or the BD field sign-extended to 64 bits.
BD (16–29)	Immediate field specifying a 14-bit signed two's complement branch displacement that is concatenated on the right with ‘00’ and sign-extended to 64 bits.
BI (11–15)	This field is used to specify a bit in the CR to be used as the condition of a branch conditional instruction.
BO (6–10)	This field is used to specify options for the branch conditional instructions. The encoding is described in *Section 4.2.4.2 Conditional Branch Control*.
crbA (11–15)	This field is used to specify a bit in the CR to be used as a source.
crbB (16–20)	This field is used to specify a bit in the CR to be used as a source.
crbD (6–10)	This field is used to specify a bit in the CR, or in the FPSCR, as the destination of the result of an instruction.
crfD (6–8)	This field is used to specify one of the CR fields, or one of the FPSCR fields, as a destination.
crfS (11–13)	This field is used to specify one of the CR fields, or one of the FPSCR fields, as a source.
CRM (12–19)	This field mask is used to identify the CR fields that are to be updated by the **mtcrf** instruction.
d (16–31)	Immediate field specifying a 16-bit signed two's complement integer that is sign-extended to 64 bits.

Table 8-2. Instruction Syntax Conventions (Continued)

Field	Description
ds (16–29)	Immediate field specifying a 14-bit signed two’s complement integer which is concatenated on the right with ‘00’ and sign-extended to 64 bits. This field is defined in 64-bit implementations only.
FM (7–14)	This field mask is used to identify the FPSCR fields that are to be updated by the **mtfsf** instruction.
frA (11–15)	This field is used to specify an FPR as a source.
frB (16–20)	This field is used to specify an FPR as a source.
frC (21–25)	This field is used to specify an FPR as a source.
frD (6–10)	This field is used to specify an FPR as the destination.
frS (6–10)	This field is used to specify an FPR as a source.
IMM (16–19)	Immediate field used as the data to be placed into a field in the FPSCR.
L (9-10)	Field used by the synchronize instruction. This field is defined in 64-bit implementations only.
L (10)	Field used to specify whether an integer compare instruction is to compare 64-bit numbers or 32-bit numbers. This field is defined in 64-bit implementations only. Field used by the TLB Invalidate Entry instruction.
L (15)	Field used to specify whether an integer compare instruction is to compare 64-bit numbers or 32-bit numbers. This field is defined in 64-bit implementations only. Field used by the Move To Machine State Register instruction.
LI (6–29)	Immediate field specifying a 24-bit signed two's complement integer that is concatenated on the right with ‘00’ and sign-extended to 64 bits.
LK (31)	Link bit. 0 Does not update the link register (LR). 1 Updates the LR. If the instruction is a branch instruction, the address of the instruction following the branch instruction is placed into the LR.
MB (21–25) and ME (26–30)	These fields are used in rotate instructions to specify a -bit mask consisting of ‘1’ bits from bit MB + 32 through bit ME + 32 inclusive, and ‘0’ bits elsewhere, as described in *Section 4.2.1.4 Integer Rotate and Shift Instructions*.
MB (21–26)	Field used in the MD-form and MDS-form instructions to specify the first ‘1’ bit of a 64-bit mask as described in *Section 4.2.1.4 Integer Rotate and Shift Instructions*. This field is defined in 64-bit implementations only.
ME (21–26)	Field used in the MD-form and MDS-form instructions to specify the last ‘1’ bit of a 64-bit mask as described in *Section 4.2.1.4 Integer Rotate and Shift Instructions*. This field is defined in 64-bit implementations only.
NB (16–20)	This field is used to specify the number of bytes to move in an immediate string load or store.
OE (21)	This field is used for extended arithmetic to enable setting OV and SO in the XER.
OPCD (0–5)	Primary opcode field.
rA (11–15)	This field is used to specify a GPR to be used as a source or destination.
rB (16–20)	This field is used to specify a GPR to be used as a source.
Rc (31)	Record bit. 0 Does not update the condition register (CR). 1 Updates the CR to reflect the result of the operation. For integer instructions, CR bits [0–2] are set to reflect the result as a signed quantity and CR bit [3] receives a copy of the summary overflow bit, XER[SO]. The result as an unsigned quantity or a bit string can be deduced from the EQ bit. For floating-point instructions, CR bits [4–7] are set to reflect floating-point exception, floating-point enabled exception, floating-point invalid operation exception, and floating-point overflow exception. **Note:** Eexceptions are referred to as interrupts in the architecture specification.
rD (6–10)	This field is used to specify a GPR to be used as a destination.
rS (6–10)	This field is used to specify a GPR to be used as a source.

Table 8-2. Instruction Syntax Conventions (Continued)

Field	Description
S (10)	Field used by the **tlbie** instruction that is part of the optional large page facility.
SH (16–20, or 16–20 and 30)	This field is used to specify a shift amount.
SIMM (16–31)	This immediate field is used to specify a 16-bit signed integer.
SPR (11–20)	Field used to specify a Special Purpose Register for the **mtspr** and **mfspr** instructions.
64-Bit Bridge SR (12–15)	This field is used to specify one of the 16 segment registers in 64-bit implementations that provide the optional **mtsr** and **mfsr** instructions.
TBR (11–20)	Field used by the move from time base instruction.
TH (9–10)	Field used by the optional data stream variant of the **dcbt** instruction.
TO (6–10)	This field is used to specify the conditions on which to trap. The encoding is described in *Section 4.2.4.6 Trap Instructions*.
UIMM (16–31)	This immediate field is used to specify a 16-bit unsigned integer.
XO (21–29, 21–30, 22–30, 26–30, 27–29, 27–30, or 30–31)	Extended opcode field. Bits [21–29, 27–29, 27–30, 30–31] pertain to 64-bit implementations only.

8.1.3 Notation and Conventions

The operation of some instructions is described by a semiformal language (pseudocode). See *Table 8-3* for a list of pseudocode notation and conventions used throughout this chapter.

Table 8-3. Notation and Conventions

Notation/Convention	Meaning
$\leftarrow$	Assignment
$\leftarrow$iea	Assignment of an instruction effective address. In 32-bit mode of a 64-bit implementation the high-order 32 bits of the 64-bit target are cleared.
¬	NOT logical operator
×, ∗	Multiplication
÷	Division (yielding quotient)
+	Two's-complement addition
–	Two's-complement subtraction, unary minus
=, ≠	Equals and Not Equals relations
<, ≤, >, ≥	Signed comparison relations
. (period)	Update. When used as a character of an instruction mnemonic, a period (.) means that the instruction updates the condition register field.
c	Carry. When used as a character of an instruction mnemonic, a 'c' indicates a carry out in XER[CA].
e	Extended Precision. When used as the last character of an instruction mnemonic, an 'e' indicates the use of XER[CA] as an operand in the instruction and records a carry out in XER[CA].
o	Overflow. When used as a character of an instruction mnemonic, an 'o' indicates the record of an overflow in XER[OV] and CR0[SO] for integer instructions or CR1[SO] for floating-point instructions.
<U, >U	Unsigned comparison relations

Table 8-3. Notation and Conventions (Continued)

Notation/Convention	Meaning
?	Unordered comparison relation
&, |	AND, OR logical operators
||	Used to describe the concatenation of two values (that is, 010 || 111 is the same as ‘010111’)
⊕, ≡	Exclusive-OR, Equivalence logical operators (for example, (a ≡ b) = (a ⊕ ¬ b))
0b*nnnn*	A number expressed in binary format.
0xnnnn or *x’nnnn nnnn*’	A number expressed in hexadecimal format.
(*n*)x	The replication of x, *n* times (that is, x concatenated to itself *n* – 1 times). (*n*)0 and (*n*)1 are special cases. A description of the special cases follows: • (*n*)0 means a field of *n* bits with each bit equal to ‘0’. Thus (5)0 is equivalent to ‘00000’. • (*n*)1 means a field of *n* bits with each bit equal to ‘1’. Thus (5)1 is equivalent to ‘11111’.
(**r**A|0)	The contents of **r**A if the **r**A field has the value 1–31, or the value ‘0’ if the **r**A field is ‘0’.
(**r**X)	The contents of **r**X
x[*n*]	*n* is a bit or field within x, where x is a register
x^n	x is raised to the *n*th power
ABS(x)	Absolute value of x
CEIL(x)	Least integer ≥ x
Characterization	Reference to the setting of status bits in a standard way that is explained in the text.
CIA	Current instruction address. The 64 or 32-bit address of the instruction being described by a sequence of pseudocode. Used by relative branches to set the next instruction address (NIA) and by branch instructions with LK = ’1’ to set the link register. **Note:** In 32-bit mode of 64-bit implementations, the high-order 32 bits of CIA are always cleared. Does not correspond to any architected register.
Clear	Clear the leftmost or rightmost *n* bits of a register to 0. This operation is used for rotate and shift instructions.
Clear left and shift left	Clear the leftmost *b* bits of a register, then shift the register left by *n* bits. This operation can be used to scale a known non-negative array index by the width of an element. These operations are used for rotate and shift instructions.
Cleared	Bits are set to ‘0’.
Do	Do loop. • Indenting shows range. • “To” and/or “by” clauses specify incrementing an iteration variable. • “While” clauses give termination conditions.
DOUBLE(x)	Result of converting x from floating-point single-precision format to floating-point double-precision format.
Extract	Select a field of *n* bits starting at bit position *b* in the source register, right or left justify this field in the target register, and clear all other bits of the target register to zero. This operation is used for rotate and shift instructions.
EXTS(x)	Result of extending x on the left with sign bits
GPR(x)	General purpose register x
if...then...else...	Conditional execution, indenting shows range, else is optional.
Insert	Select a field of *n* bits in the source register, insert this field starting at bit position *b* of the target register, and leave other bits of the target register unchanged. (No simplified mnemonic is provided for insertion of a field when operating on doublewords; such an insertion requires more than one instruction.) This operation is used for rotate and shift instructions. **Note:** Simplified mnemonics are referred to as extended mnemonics in the architecture specification.

Table 8-3. Notation and Conventions (Continued)

Notation/Convention	Meaning
Leave	Leave innermost do loop, or the do loop described in leave statement.
MASK(x, y)	Mask having ones in positions x through y (wrapping if x > y) and zeros elsewhere.
MEM(x, y)	Contents of y bytes of memory starting at address x. **Note:** In 32-bit mode of a 64-bit implementation, the high-order 32 bits of the 64-bit value x are ignored.
NIA	Next instruction address, which is the 64 or 32-bit address of the next instruction to be executed (the branch destination) after a successful branch. In pseudocode, a successful branch is indicated by assigning a value to NIA. For instructions which do not branch, the next instruction address is CIA + 4. **Note:** In 32-bit mode of 64-bit implementations, the high-order 32 bits of NIA are always cleared. Does not correspond to any architected register.
OEA	PowerPC operating environment architecture
Rotate	Rotate the contents of a register right or left *n* bits without masking. This operation is used for rotate and shift instructions.
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
Set	Bits are set to ‘1’.
Shift	Shift the contents of a register right or left *n* bits, clearing vacated bits (logical shift). This operation is used for rotate and shift instructions.
SINGLE(x)	Result of converting x from floating-point double-precision format to floating-point single-precision format.
SPR(x)	Special-purpose register x
TRAP	Invoke the system trap handler.
Undefined	An undefined value. The value may vary from one implementation to another, and from one execution to another on the same implementation.
UISA	PowerPC user instruction set architecture
VEA	PowerPC virtual environment architecture

Table 8-4 describes instruction field notation conventions used throughout this chapter.

Table 8-4. Instruction Field Conventions

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

Precedence rules for pseudocode operators are summarized in *Table 8-5*.

Table 8-5. Precedence Rules

Operators	Associativity
x[*n*], function evaluation	Left to right
(*n*)x or replication, x(*n*) or 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
←, ←iea	None

Operators higher in *Table 8-5* 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, "–" (unary minus) associates from left to right, so a – b – c = (a – b) – c. Parentheses are used to override the evaluation order implied by *Table 8-5*, or to increase clarity; parenthesized expressions are evaluated before serving as operands.

8.1.4 Computation Modes

The PowerPC Architecture allows for the following types of implementations:

- 64-bit implementations, in which all registers except some special-purpose registers (SPRs) are 64 bits long and effective addresses are 64 bits long. All 64-bit implementations have two modes of operation: 64-bit mode (which is the default) and 32-bit mode. The mode controls how the effective address is interpreted, how condition bits are set, and how the count register (CTR) is tested by branch conditional instructions. All instructions provided for 64-bit implementations are available in both 64 and 32-bit modes.
- 32-bit implementations, in which all registers except the FPRs are 32 bits long and effective addresses are 32 bits long.

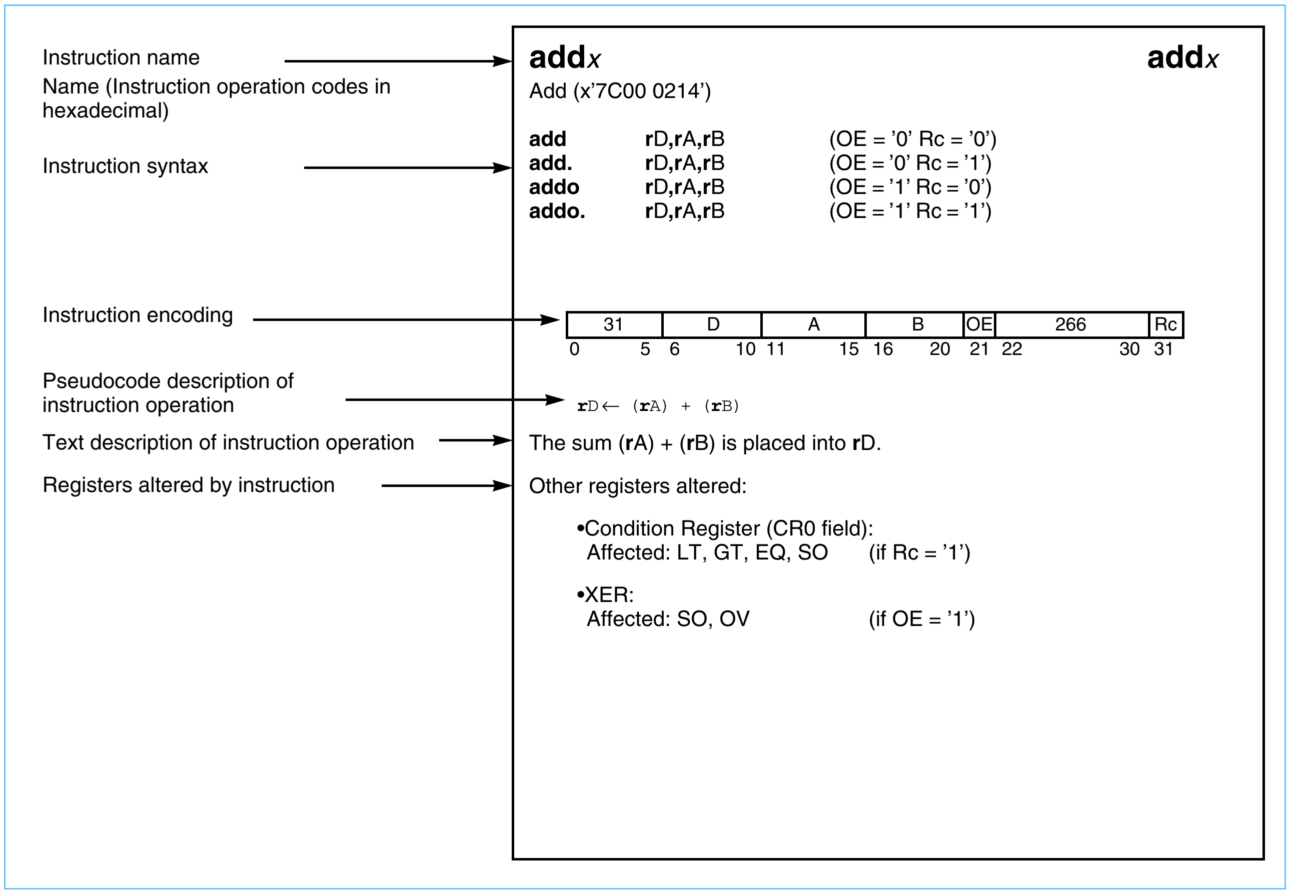
Instructions defined in this chapter are provided in both 64-bit implementations and 32-bit implementations unless otherwise stated. Instructions that are provided only for 64-bit implementations are illegal in 32-bit implementations, and vice versa.

Note: All pseudocode examples are given in the default 64-bit mode (unless otherwise stated). To determine 32-bit mode bit field equivalents, simply subtract 32.

8.2 PowerPC Instruction Set

The remainder of this chapter lists and describes the instruction set for the PowerPC Architecture. The instructions are listed in alphabetical order by mnemonic. *Figure 8-1* shows the format for each instruction description page.

Figure 8-1. Instruction Description


Instruction name
Name (Instruction operation codes in hexadecimal)
Instruction syntax
Instruction encoding
Pseudocode description of instruction operation
Text description of instruction operation
Registers altered by instruction

add*x* **add***x*

Add (x’7C00 0214’)

add	r**D**,**r**A,**r**B	(OE = ’0’ Rc = ’0’)
add.	r**D**,**r**A,**r**B	(OE = ’0’ Rc = ’1’)
addo	r**D**,**r**A,**r**B	(OE = ’1’ Rc = ’0’)
addo.	r**D**,**r**A,**r**B	(OE = ’1’ Rc = ’1’)

0–5	6–10	11–15	16–20	21	22–30	31
31	D	A	B	OE	266	Rc

```
rD ← (rA) + (rB)
```

The sum (**r**A) + (**r**B) is placed into **r**D.

Other registers altered:

- Condition Register (CR0 field):
  Affected: LT, GT, EQ, SO (if Rc = ’1’)
- XER:
  Affected: SO, OV (if OE = ’1’)

Note: The execution unit that executes the instruction may not be the same for all PowerPC processors.

add*x* **add*x***

Add (x’7C00 0214’)

add	r**D**,**r**A,**r**B	(OE = ’0’ Rc = ’0’)
add.	**r**D,**r**A,**r**B	(OE = ’0’ Rc = ’1’)
addo	**r**D,**r**A,**r**B	(OE = ’1’ Rc = ’0’)
addo.	**r**D,**r**A,**r**B	(OE = ’1’ Rc = ’1’)

31	D	A	B	OE	266	Rc
0 5	6 10	11 15	16 20	21	22 30	31

```
rD← (rA) + (rB)
```

The sum (**r**A) + (**r**B) is placed into **r**D.

The **add** instruction is preferred for addition because it sets few status bits.

Other registers altered:

- Condition Register (CR0 field)
  Affected: LT, GT, EQ, SO (if Rc = ’1’)

  Note: CR0 field may not reflect the infinitely precise result if overflow occurs (see next bullet item).

- XER
  Affected: SO, OV (if OE = ’1’)

  Note: For more information on condition codes see *Section 2.1.3 Condition Register (CR)* and *Section 2.1.5 XER Register (XER)*.

  Note: The setting of the affected bits in the XER is mode-dependent, and reflects overflow of the 64-bit result in 64-bit mode and overflow of the low-order 32-bit result in 32-bit mode. For further information about 64-bit mode and 32-bit mode in 64-bit implementations, see *Chapter 4.1.2, Computation Modes*.

addc*x*

addc*x*

Add Carrying (x’7C00 0014’)

addc	**r**D,**r**A,**r**B	(OE = ’0’ Rc = ’0’)
addc.	**r**D,**r**A,**r**B	(OE = ’0’ Rc = ’1’)
addco	**r**D,**r**A,**r**B	(OE = ’1’ Rc = ’0’)
addco.	**r**D,**r**A,**r**B	(OE = ’1’ Rc = ’1’)

31	D	A	B	OE	10	Rc
0 5	6 10	11 15	16 20	21	22 30	31

```
rD← (rA) + (rB)
```

The sum (**r**A) + (**r**B) is placed into **r**D.

Other registers altered:

- Condition Register (CR0 field)
  Affected: LT, GT, EQ, SO (if Rc = ’1’)

  Note: CR0 field may not reflect the infinitely precise result if overflow occurs (see XER below).
- XER
  Affected: CA
  Affected: SO, OV (if OE = ’1’)

  Note: The setting of the affected bits in the XER is mode-dependent, and reflects overflow of the 64-bit result in 64-bit mode and overflow of the low-order 32-bit result in 32-bit mode. For further information about 64-bit mode and 32-bit mode in 64-bit implementations, see *Chapter 4.1.2, Computation Modes*.

adde*x* adde*x*

Add Extended (x’7C00 0114’)

adde	**r**D,**r**A,**r**B	(OE = ’0’ Rc = ’0’)
adde.	**r**D,**r**A,**r**B	(OE = ’0’ Rc = ’1’)
addeo	**r**D,**r**A,**r**B	(OE = ’1’ Rc = ’0’)
addeo.	**r**D,**r**A,**r**B	(OE = ’1’ Rc = ’1’)

31	D	A	B	OE	138	Rc
0 5	6 10	11 15	16 20	21	22 30	31

```
rD← (rA) + (rB) + XER[CA]
```

The sum (**r**A) + (**r**B) + XER[CA] is placed into **r**D.

Other registers altered:

- Condition Register (CR0 field)
  Affected: LT, GT, EQ, SO (if Rc = ’1’)

  Note: CR0 field may not reflect the infinitely precise result if overflow occurs (see XER below).

- XER
  Affected: CA
  Affected: SO, OV (if OE = ’1’)

  Note: The setting of the affected bits in the XER is mode-dependent, and reflects overflow of the 64-bit result in 64-bit mode and overflow of the low-order 32-bit result in 32-bit mode. For further information about 64-bit mode and 32-bit mode in 64-bit implementations, see *Chapter 4.1.2, Computation Modes*.

addi

Add Immediate (x'3800 0000')

addi **r**D,**r**A,SIMM

14	D	A	SIMM
0 5	6 10	11 15	16 31

```
if rA = 0 then rD ← EXTS(SIMM)
else    rD ← rA + EXTS(SIMM)
```

The sum (**r**A|0) + sign extended SIMM is placed into **r**D.

The **addi** instruction is preferred for addition because it sets few status bits.

Note: **addi** uses the value '0', not the contents of GPR0, if **r**A = '0'.

Other registers altered:

- None

Simplified mnemonics:

li	**r**D,value	equivalent to	**addi**	**r**D,**0**,value
la	**r**D,disp(**r**A)	equivalent to	**addi**	**r**D,**r**A,disp
subi	**r**D,**r**A,value	equivalent to	**addi**	**r**D,**r**A,–value

addic

Add Immediate Carrying (x’3000 0000’)

addic **r**D,**r**A,SIMM

12	D	A	SIMM
0–5	6–10	11–15	16–31

```
rD← (rA) + EXTS(SIMM)
```

The sum (**r**A) + SIMM is placed into **r**D.

Other registers altered:

- XER
  Affected: CA

  Note: The setting of the affected bits in the XER is mode-dependent, and reflects overflow of the 64-bit result in 64-bit mode and overflow of the low-order 32-bit result in 32-bit mode. For further information about 64-bit mode and 32-bit mode in 64-bit implementations, see *Chapter 4.1.2, Computation Modes*.

Simplified mnemonics:

subic	**r**D,**r**A,value	equivalent to	**addic**	**r**D,**r**A,–value

addic. addic.

Add Immediate Carrying and Record (x’3400 0000’)

addic. **r**D,**r**A,SIMM

13	D	A	SIMM
0 5	6 10	11 15	16 31

```
rD← (rA) + EXTS(SIMM)
```

The sum (**r**A) + SIMM is placed into **r**D.

Other registers altered:

- Condition Register (CR0 field)
  Affected: LT, GT, EQ, SO

  Note: CR0 field may not reflect the infinitely precise result if overflow occurs (see XER below).

- XER
  Affected: CA

  Note: The setting of the affected bits in the XER is mode-dependent, and reflects overflow of the 64-bit result in 64-bit mode and overflow of the low-order 32-bit result in 32-bit mode. For further information about 64-bit mode and 32-bit mode in 64-bit implementations, see *Chapter 4.1.2, Computation Modes*.

Simplified mnemonics:

subic.	**r**D,**r**A,value	equivalent to	**addic.**	**r**D,**r**A,–value

addis

Add Immediate Shifted (x’3C00 0000’)

addis **r**D,**r**A,SIMM

15	D	A	SIMM
0 5	6 10	11 15	16 31

```
if rA = 0 then rD← EXTS(SIMM || (16)0)
else      rD← (rA) + EXTS(SIMM || (16)0)
```

The sum (**r**A|0) + (SIMM || 0x0000) is placed into **r**D.

The **addis** instruction is preferred for addition because it sets few status bits.

Note: **addis** uses the value ‘0’, not the contents of GPR0, if **r**A = ‘0’.

Other registers altered:

- None

Simplified mnemonics:

lis	**r**D,value	equivalent to	**addis**	**r**D,**0**,value
subis	**r**D,**r**A,value	equivalent to	**addis**	**r**D,**r**A,–value

addme*x* **addme*x***

Add to Minus One Extended (x’7C00 01D4’)

addme	r**D**,**r**A	(OE = ’0’ Rc = ’0’)
addme.	**r**D,**r**A	(OE = ’0’ Rc = ‘1’)
addmeo	**r**D,**r**A	(OE = ‘1’ Rc = ’0’)
addmeo.	**r**D,**r**A	(OE = ‘1’ Rc = ‘1’)

▨ Reserved

31	D	A	▨ 0 0 0 0 0	OE	234	Rc
0 5	6 10	11 15	16 20	21	22 30	31

```
rD← (rA) + XER[CA] - 1
```

The sum (**r**A) + XER[CA] + 0xFFFF_FFFF_FFFF_FFFF is placed into **r**D.

Other registers altered:

- Condition Register (CR0 field)
  Affected: LT, GT, EQ, SO (if Rc = ’1’)

  Note: CR0 field may not reflect the infinitely precise result if overflow occurs (see XER below).

- XER
  Affected: CA
  Affected: SO, OV (if OE = ’1’)

  Note: The setting of the affected bits in the XER is mode-dependent, and reflects overflow of the 64-bit result in 64-bit mode and overflow of the low-order 32-bit result in 32-bit mode. For further information about 64-bit mode and 32-bit mode in 64-bit implementations, see *Chapter 4.1.2, Computation Modes*.

addze*x* addze*x*

Add to Zero Extended (x'7C00 0194')

addze	**r**D,**r**A	(OE = '0' Rc = '0')
addze.	**r**D,**r**A	(OE = '0' Rc = '1')
addzeo	**r**D,**r**A	(OE = '1' Rc = '0')
addzeo.	**r**D,**r**A	(OE = '1' Rc = '1')

▨ Reserved

31	D	A	0 0 0 0 0	OE	202	Rc
0 5	6 10	11 15	16 20	21	22 30	31

```
rD← (rA) + XER[CA]
```

The sum (**r**A) + XER[CA] is placed into **r**D.

Other registers altered:

- Condition Register (CR0 field)
  Affected: LT, GT, EQ, SO (if Rc = '1')

  Note: CR0 field may not reflect the infinitely precise result if overflow occurs (see XER below).

- XER
  Affected: CA
  Affected: SO, OV (if OE = '1')

  Note: The setting of the affected bits in the XER is mode-dependent, and reflects overflow of the 64-bit result in 64-bit mode and overflow of the low-order 32-bit result in 32-bit mode. For further information about 64-bit mode and 32-bit mode in 64-bit implementations, see *Chapter 4.1.2, Computation Modes*.

and*x* and*x*

AND (x’7C00 0038’)

and	**r**A,**r**S,**r**B	(Rc = ’0’)
and.	**r**A,**r**S,**r**B	(Rc = ’1’)

31	S	A	B	28	Rc
0 5	6 10	11 15	16 20	21 30	31

```
rA ← (rS) & (rB)
```

The contents of **r**S are ANDed with the contents of **r**B and the result is placed into **r**A.

Other registers altered:

- Condition Register (CR0 field)
  Affected: LT, GT, EQ, SO (if Rc = ’1’)

andc*x* **andc*x***

AND with Complement (x’7C00 0078’)

andc	**r**A,**r**S,**r**B	(Rc = ’0’)
andc.	**r**A,**r**S,**r**B	(Rc = ’1’)

31	S	A	B	60	Rc
0 5	6 10	11 15	16 20	21 30	31

```
rA ← (rS) + ¬ (rB)
```

The contents of **r**S are ANDed with the one’s complement of the contents of **r**B and the result is placed into **r**A.

Other registers altered:

- Condition Register (CR0 field)
  Affected: LT, GT, EQ, SO (if Rc = ’1’)

andi. andi.

AND Immediate (x’7000 0000’)

andi. **r**A,**r**S,UIMM

28	S	A	UIMM
0 5	6 10	11 15	16 31

```
rA← (rS) & ((48)0 || UIMM)
```

The contents of **r**S are ANDed with 0x0000_0000_0000 || UIMM and the result is placed into **r**A.

Other registers altered:

- Condition Register (CR0 field)
  Affected: LT, GT, EQ, SO

andis. andis.

AND Immediate Shifted (x’7400 0000’)

andis. **r**A,**r**S,UIMM

29	S	A	UIMM
0 5	6 10	11 15	16 31

```
rA ← (rS) + ((32)0 || UIMM || (16)0)
```

The contents of **r**S are ANDed with 0x0000_0000 || UIMM || 0x0000 and the result is placed into **r**A.

Other registers altered:

- Condition Register (CR0 field)
  Affected: LT, GT, EQ, SO

b*x* **b***x*

Branch (x’4800 0000’)

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

18	LI	AA	LK
0 5	6 29	30	31

```
if AA then NIA ←iea EXTS(LI || ’00’)
else NIA ←iea CIA + EXTS(LI || ’00’)
if LK then LR ←iea CIA + 4
```

target_addr specifies the branch target address.

If AA = ’0’, then the branch target address is the sum of LI || ‘00’ sign-extended and the address of this instruction, with the high-order 32 bits of the branch target address cleared in 32-bit mode of 64-bit implementations.

If AA = ’1’, then the branch target address is the value LI || ‘00’ sign-extended, with the high-order 32 bits of the branch target address cleared in 32-bit mode of 64-bit implementations.

If LK = ’1’, then the effective address of the instruction following the branch instruction is placed into the link register.

Other registers altered:

- Affected: Link Register (LR) (if LK = ’1’)

bc*x* bc*x*

Branch Conditional (x’4000 0000’)

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

16	BO	BI	BD	AA	LK
0 5	6 10	11 15	16 29	30	31

```
if (64-bit implementation) & (64-bit mode)
then m← 0
else m← 32
if ¬ BO[2] then CTR ← CTR - 1
ctr_ok ← BO[2] | ((CTR[m–63] ≠0) ⊕ BO[3])
cond_ok ←BO[0] | (CR[BI] ≡ BO[1])
if ctr_ok & cond_ok then
 if AA then NIA ←iea EXTS(BD || ’00’)
 else NIA ←iea CIA + EXTS(BD || ’00’)
 if LK then LR ←iea CIA + 4
```

The BI field specifies the bit in the condition register (CR) to be used as the condition of the branch. The BO field is encoded as described in *Table 4-20 BO Operand Encodings*. Additional information about BO field encoding is provided in *Section 4.2.4.2 Conditional Branch Control*. target_addr specifies the branch target address.

If AA = ’0’, then the branch target address is the sum of BD || ’00’ sign-extended and the address of this instruction, with the high-order 32 bits of the branch target address cleared in 32-bit mode of 64-bit implementations.

If AA = ’1’, the branch target address is the value BD || ’00’ sign-extended, with the high-order 32 bits of the branch target address cleared in 32-bit mode of 64-bit implementations.

If LK = ’1’, the effective address of the instruction following the branch instruction is placed into the link register.

Other registers altered:

• Count Register (CTR)	(if BO[2] = ’0’)
• Link Register (LR)	(if LK = ’1’)

Simplified mnemonics:

blt	target	equivalent to	**bc**	12,0,target
bne	cr2,target	equivalent to	**bc**	4,10,target
bdnz	target	equivalent to	**bc**	16,0,target

bcctr*x* **bcctr*x***

Branch Conditional to Count Register (x'4C00 0420')

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

▨ Reserved

19	BO	BI	0 0 0	BH	528	LK
0 5	6 10	11 15	16 18	19 20	21 30	31

```
cond_ok ← BO[0] | (CR[BI] ≡ BO[1])
if cond_ok then
 NIA ←iea CTR[0–61] || '00'
 if LK then LR ←iea CIA + 4
```

The BI field specifies the bit in the condition register to be used as the condition of the branch. The BO field is encoded as described in *Table 4-20 BO Operand Encodings*. The BH field is used as described in *Table 4-22 BH Field Encodings*. The branch target address is CTR[0–61] || '00', with the high-order 32 bits of the branch target address cleared in 32-bit mode of 64-bit implementations. Additional information about BO field encoding is provided in *Section 4.2.4.2 Conditional Branch Control*.

If LK = '1' the effective address of the instruction following the branch instruction is placed into the link register.

If the "decrement and test CTR" option is specified (BO[2] = '0'), the instruction form is invalid.

Other registers altered:

- Link Register (LR) (if LK = '1')

Simplified mnemonics:

bcctr	4,6	equivalent to	**bcctr**	4,6,0
bltctr		equivalent to	**bcctr**	12,0,0
bnectr	cr2	equivalent to	**bcctr**	4,10,0

bclr*x* **bclr***x*

Branch Conditional to Link Register (x’4C00 0020’)

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

▨ Reserved

19	BO	BI	▨ 0 0 0	BH	16	LK
0 5	6 10	11 15	16 18	19 20	21 30	31

```
if (64-bit implementation) & (64-bit mode)
then m← 0
else m← 32
if ¬ BO[2] then CTR ← CTR – 1
ctr_ok ← BO[2] | ((CTR[m–63] ≠ 0) ⊕ BO[3])
cond_ok ← BO[0] | (CR[BI] ≡ BO[1])
if ctr_ok & cond_ok then
 NIA ←iea LR[0–61] || ’00’
 if LK then LR ←iea CIA + 4
```

The BI field specifies the bit in the condition register to be used as the condition of the branch. The BO field is encoded as described in *Table 4-20 BO Operand Encodings*. The BH field is used as described in *Table 4-22 BH Field Encodings*. The branch target address is LR[0–61] || ’00’, with the high-order 32 bits of the branch target address cleared in 32-bit mode of a 64-bit implementations. Additional information about BO field encoding is provided in *Section 4.2.4.2 Conditional Branch Control*.

If LK = ’1’, then the effective address of the instruction following the branch instruction is placed into the link register.

Other registers altered:

- Count Register (CTR) (if BO[2] = ’0’)
- Link Register (LR) (if LK = ’1’)

Simplified mnemonics:

bclr	4,6	equivelent to	**bclr**	4,6,0
bltlr		equivalent to	**bclr**	12,0,0
bnelr	cr2	equivalent to	**bclr**	4,10,0
bdnzlr		equivalent to	**bclr**	16,0,0

cmp cmp

Compare (x’7C00 0000’)

cmp **crf**D,L,**r**A,**r**B

▨ Reserved

31	crfD	0 ▨	L	A	B	0	0 ▨
0 5	6 8	9	10	11 15	16 20	21 30	31

```
if L = ’0’ then a ← EXTS(rA[32-63])
        b ← EXTS(rB[32-63])
else    a ← (rA)
        b ← (rB)
if   a < b then c ← ’100’
else if a > b then c ← ‘010’
else     c ← ’001’
CR[(4 × crfD) - (4 × crfD + 3)] ← c || XER[SO]
```

The contents of **r**A (or the low-order 32 bits of **r**A if L = ’0’) are compared with the contents of **r**B (or the low-order 32 bits of **r**B if L = ’0’), treating the operands as signed integers. The result of the comparison is placed into CR field **crf**D.

Other registers altered:

- Condition Register (CR field specified by operand **crf**D)
  Affected: LT, GT, EQ, SO

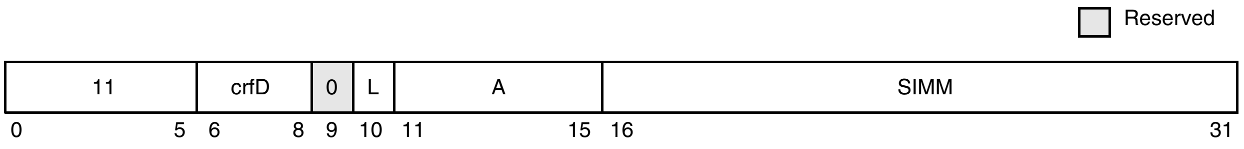
Simplified mnemonics:

cmpd	**r**A,**r**B	equivalent to	**cmp**	**0,1,r**A,**r**B
cmpw	**cr3,r**A,**r**B	equivalent to	**cmp**	**3,0,r**A,**r**B

cmpi

Compare Immediate (x’2C00 0000’)

cmpi **crf**D,L,**r**A,SIMM



```
if L = ’0’ then a ← EXTS(rA[32-63])
       elsea ← (rA)
if   a < EXTS(SIMM) then c ← ’100’
else if a > EXTS(SIMM) then c ← ‘010’
else   c ← ’001’
CR[(4 × crfD) - (4 × crfD + 3)] ← c || XER[SO]
```

The contents of **r**A (or the low-order 32 bits of **r**A sign-extended to 64 bits if L = ’0’) are compared with the sign-extended value of the SIMM field, treating the operands as signed integers. The result of the comparison is placed into CR field **crf**D.

Other registers altered:

- Condition Register (CR field specified by operand **crf**D)
  Affected: LT, GT, EQ, SO

Simplified mnemonics:

cmpdi	**r**A,value	equivalent to	**cmpi**	**0,1,r**A,value
cmpwi	**cr3,r**A,value	equivalent to	**cmpi**	**3,0,r**A,value

cmpl cmpl

Compare Logical (x’7C00 0040’)

cmpl **crf**D,L,**r**A,**r**B

▨ Reserved

31	crfD	▨ 0	L	A	B	32	▨ 0
0 5	6 8	9	10	11 15	16 20	21	31

```
if L = 0 then a ← (32)0 || rA[32–63]
         b← (32)0 || rB[32–63]
       else a ← (rA)
         b ← (rB)
if   a <U b then c ← ’100’
else if a >U b then c ← ’010’
else      c ← ’001’
CR[(4 × crfD) – (4 × crfD + 3)] ← c || XER[SO]
```

The contents of **r**A (or the low-order 32 bits of **r**A if L = ’0’) are compared with the contents of **r**B (or the low-order 32 bits of **r**B if L = ’0’), treating the operands as unsigned integers. The result of the comparison is placed into CR field **crf**D.

Other registers altered:

- Condition Register (CR field specified by operand **crf**D)
  Affected: LT, GT, EQ, SO

Simplified mnemonics:

cmpld	**r**A,**r**B	equivalent to	**cmpl**	**0,1,r**A,**r**B
cmplw	**cr3,r**A,**r**B	equivalent to	**cmpl**	**3,0,r**A,**r**B

cmpli

cmpli

Compare Logical Immediate (x’2800 0000’)

cmpli **crf**D,L,**r**A,UIMM

▨ Reserved

10	crfD	▨ 0	L	A	UIMM
0 5	6 8	9	10	11 15	16 31

```
if L = 0 then a ← (32)0 || rA[32–63]
       else a ← (rA)
if   a <U ((48)0 || UIMM) then c ← ’100’
else if a >U ((48)0 || UIMM) then c ← ’010’
else  c ← ’00’1
CR[(4 × crfD) - (4 × crfD + 3)] ← c || XER[SO]
```

The contents of **r**A (or the low-order 32 bits of **r**A zero-extended to 64-bits if L = ’0’) are compared with 0x0000_0000_0000 || UIMM, treating the operands as unsigned integers. The result of the comparison is placed into CR field **crf**D.

Other registers altered:

- Condition Register (CR field specified by operand **crf**D)
  Affected: LT, GT, EQ, SO

Simplified mnemonics:

cmpldi	**r** A,value	equivalent to	**cmpli**	**0,1,r**A,value
cmplwi	**cr3,r**A,value	equivalent to	**cmpli**	**3,0,r**A,value

cntlzd*x* **cntlzd***x*

Count Leading Zeros Doubleword (x’7C00 0074’)

cntlzd	**r**A,**r**S	(Rc = ’0’)
cntlzd.	**r**A,**r**S	(Rc = ’1’)

▨ Reserved

31	S	A	0 0 0 0 0 ▨	58	Rc
0 – 5	6 – 10	11 – 15	16 – 20	21 – 30	31

```
n ← 0
do while n < 64
   if rS[n] = 1 then leave
   n ← n + 1
rA ← n
```

A count of the number of consecutive zero bits starting at bit [0] of register **r**S is placed into **r**A. This number ranges from 0 to 64, inclusive.

Other registers altered:

- Condition Register (CR0 field)
  Affected: LT, GT, EQ, SO (Rc = ’1’)

  Note: If Rc = ’1’, then LT is cleared in the CR0 field.

cntlzw*x* **cntlzw***x*

Count Leading Zeros Word (x’7C00 0034’)

cntlzw	**r**A,**r**S	(Rc = ’0’)
cntlzw.	**r**A,**r**S	(Rc = ’1’)

▨ Reserved

31	S	A	0 0 0 0 0	26	Rc
0 5	6 10	11 15	16 20	21 30	31

```
n ← 32
do while n < 64
if rS[n] = 1 then leave
n ← n + 1
rA ← n– 32
```

A count of the number of consecutive zero bits starting at bit [32] of **r**S is placed into **r**A. This number ranges from 0 to 32, inclusive.

Other registers altered:

- Condition Register (CR0 field)
  Affected: LT, GT, EQ, SO (if Rc = ’1’)

  Note: If Rc = ’1’, then LT is cleared in the CR0 field.

crand crand

Condition Register AND (x’4C00 0202’)

crand **crb**D,**crb**A,**crb**B

▨ Reserved

19	**crb**D	**crb**A	**crb**B	257	▨ 0
0 5	6 10	11 15	16 20	21 30	31

```
CR[crbD] ← CR[crbA] & CR[crbB]
```

The bit in the condition register specified by **crb**A is ANDed with the bit in the condition register specified by **crb**B. The result is placed into the condition register bit specified by **crb**D.

Other registers altered:

- Condition Register
  Affected: Bit specified by operand **crb**D

crandc

crandc

Condition Register AND with Complement (x’4C00 0102’)

crandc **crb**D,**crb**A,**crb**B

▨ Reserved

19	crbD	crbA	crbB	129	▨ 0
0 5	6 10	11 15	16 20	21 30	31

```
CR[crbD] ← CR[crbA] & ¬ CR[crbB]
```

The bit in the condition register specified by **crb**A is ANDed with the complement of the bit in the condition register specified by **crb**B and the result is placed into the condition register bit specified by **crb**D.

Other registers altered:

- Condition Register
  Affected: Bit specified by operand **crb**D

creqv

creqv

Condition Register Equivalent (x’4C00 0242’)

creqv **crb**D,**crb**A,**crb**B

▨ Reserved

19	crbD	crbA	crbB	289	0
0 5	6 10	11 15	16 20	21 30	31

```
CR[crbD] ← CR[crbA] ≡ CR[crbB]
```

The bit in the condition register specified by **crb**A is XORed with the bit in the condition register specified by **crb**B and the complemented result is placed into the condition register bit specified by **crb**D.

Other registers altered:

- Condition Register
  Affected: Bit specified by operand **crb**D

Simplified mnemonics:

crset	**crb**D	equivalent to	**creqv**	**crb**D,**crb**D,**crb**D

crnand crnand

Condition Register NAND (x’4C00 01C2’)

crnand **crb**D,**crb**A,**crb**B

▨ Reserved

19	crbD	crbA	crbB	225	0
0 5	6 10	11 15	16 20	21 30	31

```
CR[crbD] ← ¬ (CR[crbA] & CR[crbB])
```

The bit in the condition register specified by **crb**A is ANDed with the bit in the condition register specified by **crb**B and the complemented result is placed into the condition register bit specified by **crb**D.

Other registers altered:

- Condition Register
  Affected: Bit specified by operand **crb**D

crnor crnor

Condition Register NOR (x’4C00 0042’)

crnor **crb**D,**crb**A,**crb**B

▨ Reserved

19	crbD	crbA	crbB	33	▨ 0
0 5	6 10	11 15	16 20	21 30	31

```
CR[crbD] ← ¬ (CR[crbA] | CR[crbB])
```

The bit in the condition register specified by **crb**A is ORed with the bit in the condition register specified by **crb**B and the complemented result is placed into the condition register bit specified by **crb**D.

Other registers altered:

- Condition Register
  Affected: Bit specified by operand **crb**D

Simplified mnemonics:

crnot	**crb**D,**crb**A	equivalent to	**crnor**	**crb**D,**crb**A,**crb**A

cror cror

Condition Register OR (x’4C00 0382’)

cror **crb**D,**crb**A,**crb**B

▨ Reserved

19	crbD	crbA	crbB	449	0
0 5	6 10	11 15	16 20	21 30	31

```
CR[crbD] ← CR[crbA] | CR[crbB]
```

The bit in the condition register specified by **crb**A is ORed with the bit in the condition register specified by **crb**B. The result is placed into the condition register bit specified by **crb**D.

Other registers altered:

- Condition Register
  Affected: Bit specified by operand **crb**D

Simplified mnemonics:

crmove	**crb**D,**crb**A	equivalent to	**cror**	**crb**D,**crb**A,**crb**A

crorc crorc

Condition Register OR with Complement (x’4C00 0342’)

crorc **crb**D,**crb**A,**crb**B

▨ Reserved

19	crbD	crbA	crbB	417	0
0 5	6 10	11 15	16 20	21 30	31

```
CR[crbD] ← CR[crbA] | ¬ CR[crbB]
```

The bit in the condition register specified by **crb**A is ORed with the complement of the condition register bit specified by **crb**B and the result is placed into the condition register bit specified by **crb**D.

Other registers altered:

- Condition Register
  Affected: Bit specified by operand **crb**D

crxor crxor

Condition Register XOR (x’4C00 0182’)

crxor **crb**D,**crb**A,**crb**B

▨ Reserved

19	crbD	crbA	crbB	193	0
0 5	6 10	11 15	16 20	21 30	31

```
CR[crbD] ← CR[crbA] ⊕ CR[crbB]
```

The bit in the condition register specified by **crb**A is XORed with the bit in the condition register specified by **crb**B and the result is placed into the condition register specified by **crb**D.

Other registers altered:

- Condition Register
  Affected: Bit specified by **crb**D

Simplified mnemonics:

crclr	**crb**D	equivalent to	**crxor**	**crb**D,**crb**D,**crb**D

dcbf

Data Cache Block Flush (x’7C00 00AC’)

dcbf **r**A,**r**B

▨ Reserved

0–5	6–10	11–15	16–20	21–30	31
31	0 0 0 0 0 ▨	A	B	86	0 ▨

EA is the sum (**r**A|0) + (**r**B).

The action taken depends on the memory mode associated with the block containing the byte addressed and the state of that block. If the system is a multiprocessor implementation, then the block is marked coherency-required, the processor will, if necessary, send an address-only broadcast to other processors. The broadcast of the **dcbf** instruction causes another processor to copy the block to memory, if it has dirty data, and then invalidate the block from the cache. The list below describes the action taken for the two states of the memory coherency attribute (M-bit).

- Coherency required (requires the use of address broadcast)
  - Unmodified block—Invalidates copies of the block in the data caches of all processors.
  - Modified block— Copies the block to memory and invalidates it. (In whatever processor it resides, there should be only one modified block).
  - Absent block—If modified copies of the block are in the data caches of other processors, it causes them to be copied to memory and invalidated in those data caches. If unmodified copies are in the data caches of other processors, it causes those copies to be invalidated in those data caches.
- Coherency not required (no address broadcast required)
  - Unmodified block—Invalidates the block in the processor’s data cache.
  - Modified block—Copies the block to memory. Invalidates the block in the processor’s data cache.
  - Absent block—No action is taken.

The function of this instruction is independent of the write-through, write-back and caching-inhibited/allowed modes of the block containing the byte addressed by the effective address.

This instruction is treated as a load from the addressed byte with respect to address translation and memory protection. It is also treated as a load for referenced and changed bit recording except that referenced and changed bit recording may not occur.

Other registers altered:

- None

dcbst

Data Cache Block Store (x’7C00 006C’)

dcbst **r**A,**r**B

▨ Reserved

31	▨ 0 0 0 0 0	A	B	54	▨ 0
0 5	6 10	11 15	16 20	21 30	31

EA is the sum (**r**A|0) + (**r**B).

The **dcbst** instruction executes as follows:

- Coherency required (requires the use of address broadcast)
  - Unmodified block—No action in this processor. Signals other processors to copy to memory any modified cache block.
  - Modified block—The cache block is written to memory. (Only one processor should have a copy of a modified block)
  - Absent block —No action in this processor. If a modified copy of the block is in the data cache of another processor, the cache line is written to memory.
- Coherency not required (no address broadcast required)
  - Unmodified block—No action is taken.
  - Modified block— The cache block is written to memory.
  - Absent block—No action is taken.

  Note: For modified cache blocks written to memory the architecture does not stipulate whether or not to clear the modified state of the cache block. It is left up to the processor designer to determine the final state of the cache block. Either modified or valid is logically correct.

The function of this instruction is independent of the write-through and caching-inhibited/allowed modes of the block containing the byte addressed by EA.

The processor treats this instruction as a load from the addressed byte with respect to address translation and memory protection, except that the system data storage error handler is not invoked, and the reference and change recording does not need to be done.

Other registers altered:

- None

dcbt dcbt

Data Cache Block Touch (x’7C00 022C’)

dcbt **r**A,**r**B,TH

▨ Reserved

0–5	6–8	9–10	11–15	16–20	21–30	31
31	0 0 0	TH	A	B	278	0

EA is the sum (**r**A|0) + (**r**B).

This instruction is a hint that performance will possibly be improved if the block containing the byte addressed by EA and the TH field is fetched into the data cache, because the program will probably soon load from the addressed byte. If the block is caching-inhibited, the hint is ignored and the instruction is treated as a no-op. Executing **dcbt** does not cause the system alignment error handler to be invoked.

The encodings of the TH field are as follows:

Table 8-6. Encodings of the TH Field

TH	Description
00	The memory location is the block containing the byte addressed by the effective address.
01	The memory locations are the block containing the byte addressed by the effective address and sequentially following blocks (i.e., the blocks containing the bytes addressed by EA + n × block_size, where n = 0, 1, 2, ...).
10	Reserved **Note:** The TH field should not be set to ’10’, because the value may be assigned a meaning in some future version of the architecture.
11	The memory locations are the block containing the byte addressed by the effective address and sequentially preceding blocks (i.e., the blocks containing the bytes addressed by EA - n × block_size, where n = 0, 1, 2, ...).

The actions (if any) taken by the processor in response to the hint are not considered to be “caused by” or “associated with” the **dcbt** instruction (for example, **dcbt** is considered not to cause any data accesses). No means are provided by which software can synchronize these actions with the execution of the instruction stream. For example, these actions are not ordered by memory barriers.

This instruction is treated as a load from the addressed byte with respect to address translation, memory protection, and reference and change recording except that referenced and changed bit recording may not occur. Additionally, no exception occurs in the case of a translation fault or protection violation.

The program uses the **dcbt** instruction to request a cache block fetch before it is actually needed by the program. The program can later execute load instructions to put data into registers. However, the processor is not obliged to load the addressed block into the data cache.

Note: This instruction is defined architecturally to perform the same functions as the **dcbtst** instruction. Both are defined in order to allow implementations to differentiate the bus actions when fetching into the cache for the case of a load and for a store.

In response to the hint provided by **dcbt**, the processor may prefetch the specified block into the data cache, or take other actions that reduce the latency of subsequent load or store instructions that refer to the block.

Note: Earlier implementations that do not support the optional version of **dcbt** ignore the TH field (i.e., treat it as if it were set to '00'), and do not necessarily ignore the hint provided by **dcbt** if the specified block is in storage that is Guarded and not Caching Inhibited. Therefore a **dcbt** instruction with TH[1] = '1' should not specify an EA in such memory if the program is to be run on such implementations.

Earlier implementations do not necessarily ignore the hint provided by **dcbt** if the specified block is in memory that is Guarded and not Caching Inhibited. Therefore a **dcbt** instruction should not specify an EA in such memory if the program is to be run on such implementations.

Other registers altered:

- None

dcbtst dcbtst

Data Cache Block Touch for Store (x’7C00 01EC’)

dcbtst **r**A,**r**B

▨ Reserved

31	▨ 00 000	A	B	246	▨ 0
0 5	6 10	11 15	16 20	21 30	31

EA is the sum (**r**A|0) + (**r**B).

This instruction is a hint that performance will possibly be improved if the block containing the byte addressed by EA is fetched into the data cache, because the program will probably soon store from the addressed byte. If the block is caching-inhibited or guarded, the hint is ignored and the instruction is treated as a no-op. Executing **dcbtst** does not cause the system alignment error handler to be invoked.

This instruction is treated as a load from the addressed byte with respect to address translation, memory protection, and reference and change recording except that referenced and changed bit recording may not occur. Additionally, no exception occurs in the case of a translation fault or protection violation.

The program uses **dcbtst** to request a cache block fetch to potentially improve performance for a subsequent store to that EA, as that store would then be to a cached location. However, the processor is not obliged to load the addressed block into the data cache.

Note: This instruction is defined architecturally to perform the same functions as the **dcbt** instruction. Both are defined in order to allow implementations to differentiate the bus actions when fetching into the cache for the case of a load and for a store.

Note: In response to the hint provided by **dcbtst**, the processor may prefetch the specified block into the data cache, or take other actions that reduce the latency of subsequent load or store instructions that refer to the block.

Earlier implementations do not necessarily ignore the hint provided by **dcbtst** if the specified block is in memory that is Guarded and not Caching Inhibited. Therefore a **dcbtst** instruction should not specify an EA in such memory if the program is to be run on such implementations.

Other registers altered:

- None

dcbz dcbz

Data Cache Block Clear to Zero (x’7C00 07EC’)

dcbz **r**A,**r**B

▨ Reserved

0–5	6–10	11–15	16–20	21–30	31
31	▨ 0 0 0 0 0	A	B	1014	0

```
if A = 0 then b ← 0
else b ← (RA)
EA ← b + (B)
n ← block size (bytes)
m ← $\log_2$(n)
ea ← EA[(0-63)-m || (m)0)
MEM(ea, n) ← (n)0x00
```

EA is the sum (**r**A|0) + (**r**B).

All bytes in the block containing the byte addressed by the effective address are set to zero.

This instruction is treated as a store to the addressed byte with respect to address translation, memory protection, referenced and changed recording. It is also treated as a store with respect to the ordering enforced by **eieio** and the ordering enforced by the combination of caching-inhibited and guarded attributes for a page (or block).

The **dcbz** instruction executes as follows:

- **dcbz** does not cause the block to exist in the data cache if the block is in memory that is caching inhibited.
- For memory that is neither write-through required nor caching inhibited, **dcbz** provides an efficient means of setting blocks of memory to zero. It can be used to initialize large areas of such memory, in a manner that is likely to consume less memory bandwidth than an equivalent sequence of store instructions.
- If the page containing the byte addressed by EA is in caching-inhibited or write-through mode, either all bytes of main memory that correspond to the addressed cache block are cleared or the alignment exception handler is invoked. The exception handler can then clear all bytes in main memory that correspond to the addressed cache block.
- For memory that is either write-through required or caching inhibited, **dcbz** is likely to take significantly longer to execute than an equivalent sequence of store instructions.

Other registers altered:

- None

The PowerPC OEA describes how the **dcbz** instruction may establish a block in the data cache without verifying that the associated physical address is valid. This scenario can cause a delayed machine check exception; see *Chapter 6, Exceptions* for a discussion about this type of machine check exception.

divd*x* **divd*x***

Divide Doubleword (x’7C00 03D2’)

divd	r**D**,**r**A,**r**B	(OE = ’0’ Rc = ’0’)
divd.	r**D**,**r**A,**r**B	(OE = ’0’ Rc = ’1’)
divdo	r**D**,**r**A,**r**B	(OE = ’1’ Rc = ’0’)
divdo.	r**D**,**r**A,**r**B	(OE = ’1’ Rc = ’1’)

31	D	A	B	OE	489	Rc
0 5	6 10	11 15	16 20	21	22 30	31

```
dividend[0–63] ← (rA)
divisor[0–63] ← (rB)
rD ← dividend + divisor
```

The 64-bit dividend is the contents of **r**A. The 64-bit divisor is the contents of **r**B. The 64-bit quotient is placed into **r**D. The remainder is not supplied as a result.

Both the operands and the quotient are interpreted as signed integers. The quotient is the unique signed integer that satisfies the equation—dividend = (quotient × divisor) + r—where $0 \leq r < |\text{divisor}|$ if the dividend is non-negative, and $-|\text{divisor}| < r \leq 0$ if the dividend is negative.

If an attempt is made to perform the divisions—0x8000_0000_0000_0000 ÷ –1 or <anything> ÷ 0—the contents of **r**D are undefined, as are the contents of the LT, GT, and EQ bits of the CR0 field (if Rc = ’1’). In this case, if OE = ’1’ then OV is set.

The 64-bit signed remainder of dividing (**r**A) by (**r**B) can be computed as follows, except in the case that (**r**A) = -2^{63} and (**r**B) = –1:

divd	**r**D,**r**A,**r**B	# **r**D = quotient
mulld	**r**D,**r**D,**r**B	# **r**D = quotient × divisor
subf	**r**D,**r**D,**r**A	# **r**D = remainder

Other registers altered:

- Condition Register (CR0 field):
  Affected: LT, GT, EQ, SO (if Rc = ’1’)
- XER:
  Affected: SO, OV (if OE = ’1’)
  Note: The setting of the affected bits in the XER is mode-independent, and reflects overflow of the 64-bit result.

divdu*x* **divdu*x***

Divide Doubleword Unsigned (x’7C00 0392’)

divdu	r**D**,**r**A,**r**B	(OE = ’0’ Rc = ’0’)
divdu.	**r**D,**r**A,**r**B	(OE = ’0’ Rc = ’1’)
divduo	**r**D,**r**A,**r**B	(OE = ’1’ Rc = ’0’)
divduo.	**r**D,**r**A,**r**B	(OE = ’1’ Rc = ’1’)

31	D	A	B	OE	457	Rc
0–5	6–10	11–15	16–20	21	22–30	31

```
dividend[0–63] ← (rA)
divisor[0–63] ← (rB)
rD ← dividend ÷ divisor
```

The 64-bit dividend is the contents of **r**A. The 64-bit divisor is the contents of **r**B. The 64-bit quotient of the dividend and divisor is placed into **r**D. The remainder is not supplied as a result.

Both the operands and the quotient are interpreted as unsigned integers, except that if Rc is set to 1, then the first three bits of CR0 field are set by signed comparison of the result to zero. The quotient is the unique unsigned integer that satisfies the equation—dividend = (quotient $\times$ divisor) + r—where $0 \leq r <$ divisor.

If an attempt is made to perform the division—<anything> ÷ 0—the contents of **r**D are undefined as are the contents of the LT, GT, and EQ bits of the CR0 field (if Rc = ’1’). In this case, if OE = ’1’ then OV is set.

The 64-bit unsigned remainder of dividing (**r**A) by (**r**B) can be computed as follows:

divdu	**r**D,**r**A,**r**B	# **r**D = quotient
mulld	**r**D,**r**D,**r**B	# **r**D = quotient * divisor
subf	**r**D,**r**D,**r**A	# **r**D = remainder

Other registers altered:

- Condition Register (CR0 field):
  Affected: LT, GT, EQ, SO (if Rc = ’1’)
- XER:
  Affected: SO, OV (if OE = ’1’)

  Note: The setting of the affected bits in the XER is mode-independent, and reflects overflow of the 64-bit result.

divw*x* **divw*x***

Divide Word (x’7C00 03D6’)

divw	**r**D,**r**A,**r**B	(OE = ’0’ Rc = ’0’)
divw.	**r**D,**r**A,**r**B	(OE = ’0’ Rc = ’1’)
divwo	**r**D,**r**A,**r**B	(OE = ’1’ Rc = ’0’)
divwo.	**r**D,**r**A,**r**B	(OE = ’1’ Rc = ’1’)

31	D	A	B	OE	491	Rc
0 5	6 10	11 15	16 20	21	22 30	31

```
dividend[0-63]← EXTS(rA[32-63])
divisor[0-63] ← EXTS(rB[32-63])
rD[32-63] ← dividend ÷ divisor
rD[0-31] ← undefined
```

The 64-bit dividend is the sign-extended value of the contents of the low-order 32 bits of **r**A. The 64-bit divisor is the sign-extended value of the contents of the low-order 32 bits of **r**B. The 64-bit quotient is formed. The low-order 32 bits of the 64-bit quotient are placed into the low-order 32 bits of **r**D. The contents of the high-order 32 bits of **r**D are undefined. The remainder is not supplied as a result.

Both the operands and the quotient are interpreted as signed integers. The quotient is the unique signed integer that satisfies the equation—dividend = (quotient × divisor) + r where $0 \leq r < |\text{divisor}|$ (if the dividend is non-negative), and $-|\text{divisor}| < r \leq 0$ (if the dividend is negative).

If an attempt is made to perform either of the divisions— 0x8000_0000 ÷ –1 or <anything> ÷ 0, then the contents of **r**D are undefined, as are the contents of the LT, GT, and EQ bits of the CR0 field (if Rc = 1). In this case, if OE = ’1’ then OV is set.

The 32-bit signed remainder of dividing the contents of the low-order 32 bits of **r**A by the contents of the low-order 32 bits of **r**B can be computed as follows, except in the case that the contents of the low-order 32 bits of **r**A = -2^{31} and the contents of the low-order 32 bits of **r**B = ‘–1’.

divw	**r**D,**r**A,**r**B	# **r**D = quotient
mullw	**r**D,**r**D,**r**B	# **r**D = quotient × divisor
subf	**r**D,**r**D,**r**A	# **r**D = remainder

Other registers altered:

- Condition Register (CR0 field):
  Affected: LT, GT, EQ, SO (if Rc = ’1’)
  LT, GT, EQ undefined (if Rc = ‘1’ and 64-bit mode)
- XER:
  Affected: SO, OV (if OE = ’1’)
  Note: The setting of the affected bits in the XER is mode-independent, and reflects overflow of the low-order 32-bit result.

divwu*x* divwu*x*

Divide Word Unsigned (x’7C00 0396’)

divwu	r**D**,**r**A,**r**B	(OE = ’0’ Rc = ’0’)
divwu.	r**D**,**r**A,**r**B	(OE = ’0’ Rc = ’1’)
divwuo	r**D**,**r**A,**r**B	(OE = ’1’ Rc = ’0’)
divwuo.	r**D**,**r**A,**r**B	(OE = ’1’ Rc = ’1’)

31	D	A	B	OE	459	Rc
0 5	6 10	11 15	16 20	21	22 30	31

```
dividend[0–63] ← (32)0 || rA[32–63]
divisor[0-63] ← (32)0‖rB[32-63]
rD[32-63] ← dividend ÷ divisor
rD[0-31] ← undefined
```

The 64-bit dividend is the zero-extended value of the contents of the low-order 32 bits of **r**A. The 64-bit divisor is the zero-extended value the contents of the low-order 32 bits of **r**B. A 64-bit quotient is formed. The low-order 32 bits of the 64-bit quotient are placed into the low-order 32 bits of **r**D. The contents of the high-order 32 bits of **r**D 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 CR0 field are set by signed comparison of the result to zero. The quotient is the unique unsigned integer that satisfies the equation—dividend = (quotient × divisor) + r (where 0 ≤ r < divisor). If an attempt is made to perform the division—<anything> ÷ 0—then the contents of **r**D are undefined as are the contents of the LT, GT, and EQ bits of the CR0 field (if Rc = ’1’). In this case, if OE = ’1’ then OV is set.

The 32-bit unsigned remainder of dividing the contents of the low-order 32 bits of **r**A by the contents of the low-order 32 bits of **r**B can be computed as follows:

divwu	r**D**,**r**A,**r**B	# **r**D = quotient
mullw	**r**D,**r**D,**r**B	# **r**D = quotient × divisor
subf	**r**D,**r**D,**r**A	# **r**D = remainder

Other registers altered:

- Condition Register (CR0 field):

Affected: LT, GT, EQ, SO	(if Rc = ’1’)
LT, GT, EQ undefined	(if Rc = ‘1’ and 64-bit mode)

- XER:

Affected: SO, OV	(if OE = ’1’)

Note: The setting of the affected bits in the XER is mode-independent, and reflects overflow of the low-order 32-bit result.

eciwx eciwx

External Control In Word Indexed (x’7C00 026C’)

eciwx **r**D,**r**A,**r**B

▨ Reserved

31	D	A	B	310	▨ 0
0 5	6 10	11 15	16 20	21 30	31

The **eciwx** instruction and the EAR register can be very efficient when mapping special devices such as graphics devices that use addresses as pointers.

```
if rA = 0 then b ← 0
else b← (rA)
EA ← b + (rB)
paddr ← address translation of EA
send load word request for paddr to device identified by EAR[RID]
rD ← (32)0 || word from device
```

EA is the sum (**r**A|0) + (**r**B).

A load word request for the physical address (referred to as real address in the architecture specification) corresponding to EA is sent to the device identified by EAR[RID], bypassing the cache. The word returned by the device is placed in the low-order 32 bits of **r**D. The contents of the high-order 32 bits of **r**D are cleared.

EAR[E] must be ‘1’. If it is not, a DSI exception is generated.

EA must be a multiple of four. If it is not, one of the following occurs:

- A system alignment exception is generated.
- A DSI exception is generated (possible only if EAR[E] = ’0’).
- The results are boundedly undefined.

If this instruction is executed when MSR[DR] = ’0’ (real addressing mode), the results are boundedly undefined. This instruction is treated as a load from the addressed byte with respect to address translation, memory protection, referenced and changed bit recording, and the ordering performed by **eieio**. This instruction is optional in the PowerPC Architecture.

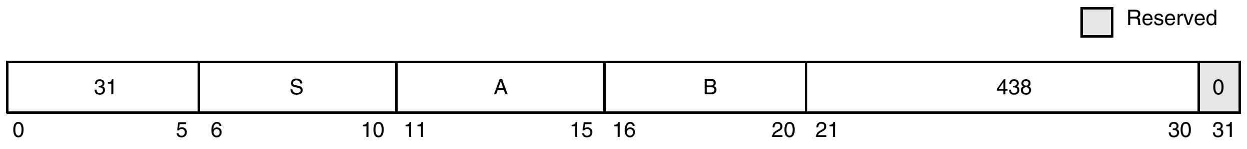
Other registers altered:

- None

ecowx ecowx

External Control Out Word Indexed (x’7C00 036C’)

ecowx **r**S,**r**A,**r**B



The **ecowx** instruction and the EAR register can be very efficient when mapping special devices such as graphics devices that use addresses as pointers.

```
if rA = 0 then b ← 0
else b ← (rA)
EA ← b + (rB)
paddr ← address translation of EA
send store word request for paddr to device identified by EAR[RID]
send rS[32–63] to device
```

EA is the sum (**r**A|0) + (**r**B).

A store word request for the physical address corresponding to EA and the contents of the low-order 32 bits of **r**S are sent to the device identified by EAR[RID], bypassing the cache.

EAR[E] must be ‘1’, if it is not, a DSI exception is generated. EA must be a multiple of four. If it is not, one of the following occurs:

- A system alignment exception is generated.
- A DSI exception is generated (possible only if EAR[E] = ’0’).
- The results are boundedly undefined.

If this instruction is executed when MSR[DR] = ’0’ (real addressing mode), the results are boundedly undefined. This instruction is treated as a store from the addressed byte with respect to address translation, memory protection, and referenced and changed bit recording, and the ordering performed by **eieio**.

Note: Software synchronization is required in order to ensure that the data access is performed in program order with respect to data accesses caused by other store or **ecowx** instructions, even though the addressed byte is assumed to be caching-inhibited and guarded. This instruction is optional in the PowerPC Architecture.

Other registers altered:

- None

eieio

Enforce In-Order Execution of I/O (x’7C00 06AC’)

▨ Reserved

31	00 000	0 0000	0000 0	854	0
0 5	6 10	11 15	16 20	21 30	31

The **eieio** instruction provides an ordering function for the effects of load and store instructions executed by a processor. These loads and stores are divided into two sets, which are ordered separately. The memory accesses caused by a **dcbz** or an **ecowx** instruction are ordered like a store, and the memory access caused by an **eciwx** instruction is ordered as a load. The two sets follow:

1. Loads and stores to memory that is both caching-inhibited and guarded, and stores to memory that is write-through required.

   The **eieio** instruction controls the order in which the accesses are performed in main memory. It ensures that all applicable memory accesses caused by instructions preceding the **eieio** instruction have completed with respect to main memory before any applicable memory accesses caused by instructions following the **eieio** instruction access main memory. It acts like a barrier that flows through the memory queues and to main memory, preventing the reordering of memory accesses across the barrier. No ordering is performed for **dcbz** if the instruction causes the system alignment error handler to be invoked.

   All accesses in this set are ordered as a single set—that is, there is not one order for loads and stores to caching-inhibited and guarded memory and another order for stores to write-through required memory.

   The ordering done by the memory barrier for accesses in this set is not cumulative.
2. Stores to memory that have all of the following attributes—caching-allowed, write-through not required, and memory-coherency required.

   The **eieio** instruction controls the order in which the accesses are performed with respect to coherent memory. It ensures that all applicable stores caused by instructions preceding the **eieio** instruction have completed with respect to coherent memory before any applicable stores caused by instructions following the **eieio** instruction complete with respect to coherent memory.

The **eieio** instruction may complete before memory accesses caused by instructions preceding the **eieio** instruction have been performed with respect to main memory or coherent memory as appropriate.

The **eieio** instruction is intended for use in managing shared data structures, in accessing memory-mapped I/O, and in preventing load/store combining operations in main memory. For the first use, the shared data structure and the lock that protects it must be altered only by stores that are in the same set (1 or 2; see previous discussion). For the second use, **eieio** can be thought of as placing a barrier into the stream of memory accesses issued by a processor, such that any given memory access appears to be on the same side of the barrier to both the processor and the I/O device.

Because the processor performs store operations in order to memory that is designated as both caching-inhibited and guarded (refer to *Section 5.1.1 Memory Access Ordering*), the **eieio** instruction is needed for such memory only when loads must be ordered with respect to stores or with respect to other loads.

Note: The **eieio** instruction does not connect hardware considerations to it such as multiprocessor implementations that send an **eieio** address-only broadcast (useful in some designs). For example, if a design has an external buffer that re-orders loads and stores for better bus efficiency, the **eieio** broadcast signals to that buffer that previous loads/stores (marked caching-inhibited, guarded, or write-through required) must complete before any following loads/stores (marked caching-inhibited, guarded, or write-through required).

Other registers altered:

- None

eqv*x* **eqv***x*

Equivalent (x’7C00 0238’)

eqv	**r**A,**r**S,**r**B	(Rc = ’0’)
eqv.	**r**A,**r**S,**r**B	(Rc = ’1’)

31	S	A	B	284	Rc
0 5	6 10	11 15	16 21	22 30	31

$\mathbf{r}\mathrm{A} \leftarrow (\mathbf{r}\mathrm{S}) \equiv (\mathbf{r}\mathrm{B})$

The contents of **r**S are XORed with the contents of **r**B and the complemented result is placed into **r**A.

Other registers altered:

- Condition Register (CR0 field):
  Affected: LT, GT, EQ, SO (if Rc = ’1’)

extsb*x* **extsb***x*

Extend Sign Byte (x’7C00 0774’)

extsb	**r**A,**r**S	(Rc = ’0’)
extsb.	**r**A,**r**S	(Rc = ’1’)

▨ Reserved

31	S	A	0 0 0 0 0	954	Rc
0–5	6–10	11–15	16–20	21–30	31

```
S ← rS[56]
rA[56–63] ← rS[56–63]
rA[0–55] ← (56)S
```

The contents of the low-order eight bits of **r**S [56-63] are placed into the low-order eight bits of **r**A . Bit [56] of **r**S is placed into bits **r**A.

Other registers altered:

- Condition Register (CR0 field):
  Affected: LT, GT, EQ, SO (if Rc = ’1’)

extsh*x* extsh*x*

Extend Sign Halfword (x’7C00 0734’)

extsh	**r**A,**r**S	(Rc = ’0’)
extsh.	**r**A,**r**S	(Rc = ’1’)

▨ Reserved

31	S	A	▨ 0 0 0 0 0	922	Rc
0 5	6 10	11 15	16 20	21 30	31

```
S ← rS[48]
rA[48–63] ← rS[48–63]
rA[0–47] ← (48)S
```

The contents of the low-order 16 bits of **r**S are placed into the low-order 16 bits of **r**A. Bit [48] of **r**S is placed into the remaining bits of **r**A.

Other registers altered:

- Condition Register (CR0 field):
  Affected: LT, GT, EQ, SO (if Rc = ’1’)

extsw*x* extsw*x*

Extend Sign Word (x'7C00 07B4')

extsw	**r**A,**r**S	(Rc = '0')
extsw.	**r**A,**r**S	(Rc = '1')

▨ Reserved

0–5	6–10	11–15	16–20	21–30	31
31	S	A	0 0 0 0 0 ▨	986	Rc

```
S ← rS[32]
rA[32–63] ← rS[32–63]
rA[0–31] ← (32)S
```

The contents of the low-order 32 bits of **r**S are placed into the low-order 32 bits of **r**A. Bit [32] of **r**S is placed into the high-order 32 bits of **r**A.

Other registers altered:

- Condition Register (CR0 field):
  Affected: LT, GT, EQ, SO (if Rc = '1')

fabs*x* fabs*x*

Floating Absolute Value (x’FC00 0210’)

fabs	**fr**D,**fr**B	(Rc = ’0’)
fabs.	**fr**D,**fr**B	(Rc = ’1’)

▨ Reserved

63	D	0 0000	B	264	Rc
0 5	6 10	11 15	16 20	21 30	31

The contents of **fr**B with bit [0] cleared are placed into **fr**D.

Note: The **fabs** instruction treats NaNs just like any other kind of value. That is, the sign bit of a NaN may be altered by **fabs**. This instruction does not alter the FPSCR.

Other registers altered:

- Condition Register (CR1 field):
  Affected: FX, FEX, VX, OX (if Rc = ’1’)

fadd*x* **fadd***x*

Floating Add (Double-Precision) (x’FC00 002A’)

fadd	**fr**D,**fr**A,**fr**B	(Rc = ’0’)
fadd.	**fr**D,**fr**A,**fr**B	(Rc = ’1’)

▨ Reserved

63	D	A	B	▨ 0 0 0 0 0	21	Rc
0 5	6 10	11 15	16 20	21 25	26 30	31

The floating-point operand in **fr**A is added to the floating-point operand in **fr**B.

If the most- significant bit of the resultant significand is not a one, the result is normalized. The result is rounded to double-precision under control of the floating-point rounding control field RN of the FPSCR and placed into **fr**D.

Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands. All 53 bits in 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’.

Other registers altered:

- Condition Register (CR1 field):
  Affected: FX, FEX, VX, OX (if Rc = ’1’)
- Floating-Point Status and Control Register:
  Affected: FPRF, FR, FI, FX, OX, UX, XX,VXSNAN, VXISI

fadds*x* fadds*x*

Floating Add Single (x’EC00 002A’)

fadds	**fr**D,**fr**A,**fr**B	(Rc = ’0’)
fadds.	**fr**D,**fr**A,**fr**B	(Rc = ’1’)

▨ Reserved

59	D	A	B	000 00 ▨	21	Rc
0 5	6 10	11 15	16 20	21 25	26 30	31

The floating-point operand in **fr**A is added to the floating-point operand in **fr**B. If the most-significant bit of the resultant significand is not a one, the result is normalized. The result is rounded to the single-precision under control of the floating-point rounding control field RN of the FPSCR and placed into **fr**D.

Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands. All 53 bits in 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’.

Other registers altered:

- Condition Register (CR1 field):
  Affected: FX, FEX, VX, OX (if Rc = ’1’)
- Floating-Point Status and Control Register:
  Affected: FPRF, FR, FI, FX, OX, UX, XX,VXSNAN, VXISI

fcfid*x* fcfid*x*

Floating Convert from Integer Doubleword (x’FC00 069C’)

fcfid	**fr**D,**fr**B	(Rc = ’0’)
fcfid.	**fr**D,**fr**B	(Rc = ’1’)

▨ Reserved

0–5	6–10	11–15	16–20	21–30	31
63	D	0 0000	B	846	Rc

The 64-bit signed fixed-point operand in register **fr**B is converted to an infinitely precise floating-point integer. The result of the conversion is rounded to double-precision using the rounding mode specified by FPSCR[RN] and placed into register **fr**D.

FPSCR[FPRF] is set to the class and sign of the result. FPSCR[FR] is set if the result is incremented when rounded. FPSCR[FI] is set if the result is inexact.

The conversion is described fully in *Appendix C.4.3 Floating-Point Convert from Integer Model*.

Other registers altered:

- Condition Register (CR1 field):
  Affected: FX, VX, FEX, OX (if Rc = ’1’)
- Floating-point Status and Control Register:
  Affected: FPRF, FR, FI, FX, XX

fcmpo

fcmpo

Floating Compare Ordered (x’FC00 0040’)

fcmpo **crf**D,**fr**A,**fr**B

▨ Reserved

63	crfD	0 0	A	B	32	0
0 5	6 8	9 10	11 15	16 20	21 30	31

```
if (frA) is a NaN or
(frB) is a NaN then c ← ‘0001’
else if (frA)< (frB) thenc ← ‘1000’
else if (frA)> (frB) thenc ← ‘0100’
else        c ← ‘0010’

FPCC ← c
CR[(4 × crfD) – (4 × crfD + 3)] ← c

if (frA) is an SNaN or
(frB) is an SNaN then
  VXSNAN ← 1
  if VE = 0 then VXVC ← 1
else if (frA) is a QNaN or
  (frB) is a QNaN then VXVC ← 1
```

The floating-point operand in **fr**A is compared to the floating-point operand in **fr**B. The result of the compare is placed into CR field **crf**D and the FPCC.

If one of the operands is a NaN, either quiet or signaling, then CR field **crf**D and the FPCC are set to reflect unordered. If one of the operands is a signaling NaN, then VXSNAN is set, and if invalid operation is disabled (VE = ’0’) then VXVC is set. Otherwise, if one of the operands is a QNaN, then VXVC is set.

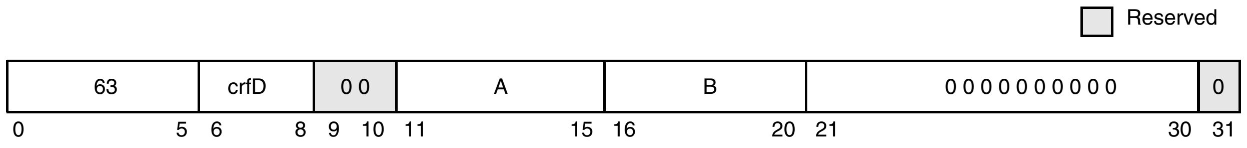
Other registers altered:

- Condition Register (CR field specified by operand **crf**D):
  Affected: LT, GT, EQ, UN
- Floating-Point Status and Control Register:
  Affected: FPCC, FX, VXSNAN, VXVC

fcmpu

Floating Compare Unordered (x'FC00 0000')

fcmpu **crf**D,**fr**A,**fr**B



```
if (frA) is a NaN or
(frB) is a NaN then c ← ‘0001’
else if (frA) < (frB) thenc ← ‘1000’
else if (frA) > (frB) thenc ← ‘0100’
else        c ← ‘0010’

FPCC ← c
CR[$(4 \times$ crfD$) - (4 \times$ crfD $+ 3)$] ← c

if (frA) is an SNaN or
(frB) is an SNaN then
   VXSNAN ← 1
```

The floating-point operand in register **fr**A is compared to the floating-point operand in register **fr**B. The result of the compare is placed into CR field **crf**D and the FPCC.

If one of the operands is a NaN, either quiet or signaling, then CR field **crf**D and the FPCC are set to reflect unordered. If one of the operands is a signaling NaN, then VXSNAN is set.

Other registers altered:

- Condition Register (CR field specified by operand **crf**D):
  Affected: LT, GT, EQ, UN
- Floating-Point Status and Control Register:
  Affected: FPCC, FX, VXSNAN

fctid*x* fctid*x*

Floating Convert to Integer Doubleword (x’FC00 065C’)

fctid	**fr**D,**fr**B	(Rc = ’0’)
fctid.	**fr**D,**fr**B	(Rc = ’1’)

▨ Reserved

63	D	0 0000	B	814	Rc
0 5	6 10	11 15	16 20	21 30	31

The floating-point operand in **fr**B is converted to a 64-bit signed fixed-point integer, using the rounding mode specified by FPSCR[RN], and placed into **fr**D.

If the operand in **fr**B is greater than 2^{63}– 1, then **fr**D is set to 0x7FFF_FFFF_FFFF_FFFF. If the operand in **fr**B is less than $–2^{63}$, then **fr**D is set to 0x8000_0000_0000_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.

The conversion is described fully in *Appendix C.4.2 Floating-Point Convert to Integer Model*.

Other registers altered:

- Condition Register (CR1 field):
  Affected: FX, FEX, VX, OX (if Rc = ’1’)
- Floating-Point Status and Control Register:
  Affected: FPRF (undefined), FR, FI, FX, XX, VXSNAN, VXCVI

fctidz*x* fctidz*x*

Floating Convert to Integer Doubleword with Round toward Zero (x’FC00 065E’)

fctidz	**fr**D,**fr**B	(Rc = ’0’)
fctidz.	**fr**D,**fr**B	(Rc = ’1’)

▨ Reserved

63	D	0 0000	B	815	Rc
0 5	6 10	11 15	16 20	21 30	31

The floating-point operand in **fr**B is converted to a 64-bit signed fixed-point integer, using the rounding mode round toward zero, and placed into **fr**D.

If the operand in **fr**B is greater than $2^{63} – 1$, then **fr**D is set to 0x7FFF_FFFF_FFFF_FFFF. If the operand in **fr**B is less than $–2^{63}$, then **fr**D is set to 0x8000_0000_0000_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.

The conversion is described fully in *Section C.4.2 Floating-Point Convert to Integer Model*.

Other registers altered:

- Condition Register (CR1 field):
  Affected: FX, FEX, VX, OX (if Rc = ’1’)
- Floating-Point Status and Control Register:
  Affected: FPRF (undefined), FR, FI, FX, XX, VXSNAN, VXCVI

fctiw*x* **fctiw***x*

Floating Convert to Integer Word (x’FC00 001C’)

fctiw	**fr**D,**fr**B	(Rc = ’0’)
fctiw.	**fr**D,**fr**B	(Rc = ’1’)

▨ Reserved

63	D	▨ 0 0000	B	14	Rc
0 5	6 10	11 15	16 20	21 30	31

The floating-point operand in register **fr**B is converted to a 32-bit signed integer, using the rounding mode specified by FPSCR[RN], and placed in bits [32–63] of **fr**D. Bits [0–31] of **fr**D are undefined.

If the operand in **fr**B are greater than $2^{31} – 1$, bits [32–63] of **fr**D are set to 0x7FFF_FFFF.

If the operand in **fr**B are less than $–2^{31}$, bits [32–63] of **fr**D are set to 0x8000_0000.

The conversion is described fully in *Appendix C.4.2 Floating-Point Convert to Integer Model*.

Except for trap-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.

(**Programmers note**: A **stfiwz** instruction should be used to store the 32-bit resultant integer because bits [0-31] of **fr**D are undefined.)

Other registers altered:

- Condition Register (CR1 field):
  Affected: FX, FEX, VX, OX (if Rc = ’1’)
- Floating-Point Status and Control Register:
  Affected: FPRF (undefined), FR, FI, FX, XX, VXSNAN, VXCVI

fctiwz*x* fctiwz*x*

Floating Convert to Integer Word with Round toward Zero (x’FC00 001E’)

fctiwz	**fr**D,**fr**B	(Rc = ’0’)
fctiwz.	**fr**D,**fr**B	(Rc = ’1’)

▨ Reserved

63	D	0 0000	B	15	Rc
0 5	6 10	11 15	16 20	21 30	31

The floating-point operand in register **fr**B is converted to a 32-bit signed integer, using the rounding mode round toward zero, and placed in bits [32–63] of **fr**D. Bits [0–31] of **fr**D are undefined.

If the operand in **fr**B is greater than $2^{31} – 1$, bits [32–63] of **fr**D are set to 0x7FFF_FFFF.
If the operand in **fr**B is less than $–2^{31}$, bits [32–63] of **fr**D are set to 0x 8000_0000.

The conversion is described fully in *Appendix C.4.2 Floating-Point Convert to Integer Model*.

Except for trap-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.

(**Programmers note**: A **stfiwz** instruction should be used to store the 32-bit resultant integer because bits [0-31] of **fr**D are undefined.)

Other registers altered:

- Condition Register (CR1 field):
  Affected: FX, FEX, VX, OX (if Rc = ’1’)
- Floating-Point Status and Control Register:
  Affected: FPRF (undefined), FR, FI, FX, XX, VXSNAN, VXCVI

fdiv*x* **fdiv*x***

Floating Divide (Double-Precision) (x’FC00 0024’)

fdiv	**fr**D,**fr**A,**fr**B	(Rc = ’0’)
fdiv.	**fr**D,**fr**A,**fr**B	(Rc = ’1’)

▨ Reserved

63	D	A	B	0 0 0 0 0	18	Rc
0 5	6 10	11 15	16 20	21 25	26 30	31

The floating-point operand in register **fr**A is divided by the floating-point operand in register **fr**B. The remainder is not supplied as a result.

If the most-significant bit of the resultant significand is not a one, the result is normalized. The result is rounded to double-precision under control of the floating-point rounding control field RN of the FPSCR and placed into **fr**D.

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

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = ’1’ and zero divide exceptions when FPSCR[ZE] = ’1’.

Other registers altered:

- Condition Register (CR1 field):
  Affected: FX, FEX, VX, OX (if Rc = ’1’)
- Floating-Point Status and Control Register:
  Affected: FPRF, FR, FI, FX, OX, UX, ZX, XX, VXSNAN, VXIDI, VXZDZ

fdivs*x* **fdivs***x*

Floating Divide Single (x’EC00 0024’)

fdivs	**fr**D,**fr**A,**fr**B	(Rc = ’0’)
fdivs.	**fr**D,**fr**A,**fr**B	(Rc = ’1’)

▨ Reserved

59	D	A	B	0 0 0 0 0	18	Rc
0 5	6 10	11 15	16 20	21 25	26 30	31

The floating-point operand in register **fr**A is divided by the floating-point operand in register **fr**B. The remainder is not supplied as a result.

If the most-significant bit of the resultant significand is not a one, the result is normalized. The result is rounded to single-precision under control of the floating-point rounding control field RN of the FPSCR and placed into **fr**D.

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

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = ’1’ and zero divide exceptions when FPSCR[ZE] = ’1’.

Other registers altered:

- Condition Register (CR1 field):
  Affected: FX, FEX, VX, OX (if Rc = ’1’)
- Floating-Point Status and Control Register:
  Affected: FPRF, FR, FI, FX, OX, UX, ZX, XX, VXSNAN, VXIDI, VXZDZ

fmadd*x* fmadd*x*

Floating Multiply-Add (Double-Precision) (x’FC00 003A’)

fmadd	**fr**D,**fr**A,**fr**C,**fr**B	(Rc = ’0’)
fmadd.	**fr**D,**fr**A,**fr**C,**fr**B	(Rc = ’1’)

63	D	A	B	C	29	Rc
0 5	6 10	11 15	16 20	21 25	26 30	31

The following operation is performed:

```
frD ← (frA * frC) + frB
```

The floating-point operand in register **fr**A is multiplied by the floating-point operand in register **fr**C. The floating-point operand in register **fr**B is added to this intermediate result.

If the most-significant bit of the resultant significand is not a one, the result is normalized. The result is rounded to double-precision under control of the floating-point rounding control field RN of the FPSCR and placed into **fr**D.

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = ’1’.

Other registers altered:

- Condition Register (CR1 field):
  Affected: FX, FEX, VX, OX (if Rc = ’1’)
- Floating-Point Status and Control Register:
  Affected: FPRF, FR, FI, FX, OX, UX, XX, VXSNAN, VXISI, VXIMZ

fmadds*x* fmadds*x*

Floating Multiply-Add Single (x’EC00 003A’)

fmadds	**fr**D,**fr**A,**fr**C,**fr**B	(Rc = ’0’)
fmadds.	**fr**D,**fr**A,**fr**C,**fr**B	(Rc = ’1’)

59	D	A	B	C	29	Rc
0 5	6 10	11 15	16 20	21 25	26 30	31

The following operation is performed:

```
frD ← (frA × frC) + frB
```

The floating-point operand in register **fr**A is multiplied by the floating-point operand in register **fr**C. The floating-point operand in register **fr**B is added to this intermediate result.

If the most-significant bit of the resultant significand is not a one, the result is normalized. The result is rounded to single-precision under control of the floating-point rounding control field RN of the FPSCR and placed into **fr**D.

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = ’1’.

Other registers altered:

- Condition Register (CR1 field):
  Affected: FX, FEX, VX, OX (if Rc = ’1’)
- Floating-Point Status and Control Register:
  Affected: FPRF, FR, FI, FX, OX, UX, XX, VXSNAN, VXISI, VXIMZ

fmr*x* **fmr***x*

Floating Move Register (Double-Precision) (x’FC00 0090’)

fmr	**fr**D,**fr**B	(Rc = ’0’)
fmr.	**fr**D,**fr**B	(Rc = ’1’)

▨ Reserved

63	D	▨ 0 0000	B	72	Rc
0 5	6 10	11 15	16 20	21 30	31

The following operation is performed:

```
frD ← (frB)
```

The contents of register **fr**B are placed into **fr**D.

Other registers altered:

- Condition Register (CR1 field):
  Affected: FX, FEX, VX, OX (if Rc = ’1’)

fmsub*x* fmsub*x*

Floating Multiply-Subtract (Double-Precision) (x’FC00 0038’)

fmsub	**fr**D,**fr**A,**fr**C,**fr**B	(Rc = ’0’)
fmsub.	**fr**D,**fr**A,**fr**C,**fr**B	(Rc = ’1’)

63	D	A	B	C	28	Rc
0 5	6 10	11 15	16 20	21 25	26 30	31

The following operation is performed:

$$\mathbf{fr}\mathrm{D} \leftarrow [\mathbf{fr}\mathrm{A} \times \mathbf{fr}\mathrm{C}] - \mathbf{fr}\mathrm{B}$$

The floating-point operand in register **fr**A is multiplied by the floating-point operand in register **fr**C. The floating-point operand in register **fr**B is subtracted from this intermediate result.

If the most-significant bit of the resultant significand is not a one, the result is normalized. The result is rounded to double-precision under control of the floating-point rounding control field RN of the FPSCR and placed into **fr**D.

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = ’1’.

Other registers altered:

- Condition Register (CR1 field):
  Affected: FX, FEX, VX, OX (if Rc = ’1’)
- Floating-Point Status and Control Register:
  Affected: FPRF, FR, FI, FX, OX, UX, XX, VXSNAN, VXISI, VXIMZ

fmsubs*x* **fmsubs*x***

Floating Multiply-Subtract Single (x’EC00 0038’)

fmsubs	**fr**D,**fr**A,**fr**C,**fr**B	(Rc = ’0’)
fmsubs.	**fr**D,**fr**A,**fr**C,**fr**B	(Rc = ’1’)

59	D	A	B	C	28	Rc
0 5	6 10	11 15	16 20	21 25	26 30	31

The following operation is performed:

```
frD ← [frA × frC] - frB
```

The floating-point operand in register **fr**A is multiplied by the floating-point operand in register **fr**C. The floating-point operand in register **fr**B is subtracted from this intermediate result.

If the most-significant bit of the resultant significand is not a one, the result is normalized. The result is rounded to single-precision under control of the floating-point rounding control field RN of the FPSCR and placed into **fr**D.

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = ’1’.

Other registers altered:

- Condition Register (CR1 field):
  Affected: FX, FEX, VX, OX (if Rc = ’1’)
- Floating-Point Status and Control Register:
  Affected: FPRF, FR, FI, FX, OX, UX, XX, VXSNAN, VXISI, VXIMZ

fmul*x* **fmul***x*

Floating Multiply (Double-Precision) (x’FC00 0032’)

fmul	**fr**D,**fr**A,**fr**C	(Rc = ’0’)
fmul.	**fr**D,**fr**A,**fr**C	(Rc = ’1’)

▨ Reserved

63	D	A	0 0 0 0 0	C	25	Rc
0 5	6 10	11 15	16 20	21 25	26 30	31

The following operation is performed:

```
frD ← (frA) × (frC)
```

The floating-point operand in register **fr**A is multiplied by the floating-point operand in register **fr**C.

If the most-significant bit of the resultant significand is not a one, the result is normalized. The result is rounded to double-precision under control of the floating-point rounding control field RN of the FPSCR and placed into **fr**D.

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

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = ’1’.

Other registers altered:

- Condition Register (CR1 field):
  Affected: FX, FEX, VX, OX (if Rc = ’1’)
- Floating-Point Status and Control Register:
  Affected: FPRF, FR, FI, FX, OX, UX, XX, VXSNAN, VXIMZ

fmuls*x* **fmuls***x*

Floating Multiply Single (x’EC00 0032’)

fmuls	**fr**D,**fr**A,**fr**C	(Rc = ’0’)
fmuls.	**fr**D,**fr**A,**fr**C	(Rc = ’1’)

▨ Reserved

59	D	A	0 0 0 0 0	C	25	Rc
0 5	6 10	11 15	16 20	21 25	26 30	31

The following operation is performed:

```
frD ← (frA) × (frC)
```

The floating-point operand in register **fr**A is multiplied by the floating-point operand in register **fr**C.

If the most-significant bit of the resultant significand is not a one, the result is normalized. The result is rounded to single-precision under control of the floating-point rounding control field RN of the FPSCR and placed into **fr**D.

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

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = ’1’.

Other registers altered:

- Condition Register (CR1 field):
  Affected: FX, FEX, VX, OX (if Rc = ’1’)
- Floating-Point Status and Control Register:
  Affected: FPRF, FR, FI, FX, OX, UX, XX, VXSNAN, VXIMZ

fnabs*x* fnabs*x*

Floating Negative Absolute Value (x’FC00 0110’)

fnabs	**fr**D,**fr**B	(Rc = ’0’)
fnabs.	**fr**D,**fr**B	(Rc = ’1’)

▨ Reserved

63	D	0 0000	B	136	Rc
0 5	6 10	11 15	16 20	21 25 26 30	31

The following operation is performed:

```
frD ← 1 || frB[1-63]
```

The contents of register **fr**B with bit [0] set are placed into **fr**D.

Note: The **fnabs** instruction treats NaNs just like any other kind of value. That is, the sign bit of a NaN may be altered by **fnabs**. This instruction does not alter the FPSCR.

Other registers altered:

- Condition Register (CR1 field):
  Affected: FX, FEX, VX, OX (if Rc = ’1’)

fneg*x*

Floating Negate (x’FC00 0050’)

fneg	**fr**D,**fr**B	(Rc = ’0’)
fneg.	**fr**D,**fr**B	(Rc = ’1’)

▨ Reserved

63	D	0 0000	B	40	Rc
0 5	6 10	11 15	16 20	21 30	31

The following operation is performed:

```
frD ←¬ frB[0] || frB[1-63]
```

The contents of register **fr**B with bit [0] inverted are placed into **fr**D.

Note: The **fneg** instruction treats NaNs just like any other kind of value. That is, the sign bit of a NaN may be altered by **fneg**. This instruction does not alter the FPSCR.

Other registers altered:

- Condition Register (CR1 field):
  Affected: FX, FEX, VX, OX (if Rc = ’1’)

fnmadd*x* fnmadd*x*

Floating Negative Multiply-Add (Double-Precision) (x'FC00 003E')

fnmadd	**fr**D,**fr**A,**fr**C,**fr**B	(Rc = '0')
fnmadd.	**fr**D,**fr**A,**fr**C,**fr**B	(Rc = '1')

63	D	A	B	C	31	Rc
0 5	6 10	11 15	16 20	21 25	26 30	31

The following operation is performed:

```
frD ← - ([frA × frC] + frB)
```

The floating-point operand in register **fr**A is multiplied by the floating-point operand in register **fr**C. The floating-point operand in register **fr**B is added to this intermediate result. If the most-significant bit of the resultant significand is not a one, the result is normalized. The result is rounded to double-precision under control of the floating-point rounding control field RN of the FPSCR, then negated and placed into **fr**D.

This instruction produces the same result as would be obtained by using the Floating Multiply-Add (**fmadd***x*) 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 zero.
- 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'.

Other registers altered:

- Condition Register (CR1 field):
  Affected: FX, FEX, VX, OX (if Rc = '1')
- Floating-Point Status and Control Register:
  Affected: FPRF, FR, FI, FX, OX, UX, XX, VXSNAN, VXISI, VXIMZ

fnmadds*x* fnmadds*x*

Floating Negative Multiply-Add Single (x'EC00 003E')

fnmadds	**fr**D,**fr**A,**fr**C,**fr**B	(Rc = '0')
fnmadds.	**fr**D,**fr**A,**fr**C,**fr**B	(Rc = '1')

59	D	A	B	C	31	Rc
0 5	6 10	11 15	16 20	21 25	26 30	31

The following operation is performed:

```
frD ← - ([frA × frC] + frB)
```

The floating-point operand in register **fr**A is multiplied by the floating-point operand in register **fr**C. The floating-point operand in register **fr**B is added to this intermediate result. If the most-significant bit of the resultant significand is not a one, the result is normalized. The result is rounded to single-precision under control of the floating-point rounding control field RN of the FPSCR, then negated and placed into **fr**D.

This instruction produces the same result as would be obtained by using the Floating Multiply-Add Single (**fmadds***x*) 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 zero.
- 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'.

Other registers altered:

- Condition Register (CR1 field):
  Affected: FX, FEX, VX, OX (if Rc = '1')
- Floating-Point Status and Control Register:
  Affected: FPRF, FR, FI, FX, OX, UX, XX, VXSNAN, VXISI, VXIMZ

fnmsub*x* fnmsub*x*

Floating Negative Multiply-Subtract (Double-Precision) (x'FC00 003C')

fnmsub	**fr**D,**fr**A,**fr**C,**fr**B	(Rc = '0')
fnmsub.	**fr**D,**fr**A,**fr**C,**fr**B	(Rc = '1')

63	D	A	B	C	30	Rc
0 5	6 10	11 15	16 20	21 25	26 30	31

The following operation is performed:

```
frD ← - ([frA × frC] - frB)
```

The floating-point operand in register **fr**A is multiplied by the floating-point operand in register **fr**C. The floating-point operand in register **fr**B is subtracted from this intermediate result.

If the most-significant bit of the resultant significand is not one, the result is normalized. The result is rounded to double-precision under control of the floating-point rounding control field RN of the FPSCR, then negated and placed into **fr**D.

This instruction produces the same result obtained by negating the result of a Floating Multiply-Subtract (**fmsub***x*) instruction 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 zero.
- 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'.

Other registers altered:

- Condition Register (CR1 field)
  Affected: FX, FEX, VX, OX (if Rc = '1')
- Floating-Point Status and Control Register:
  Affected: FPRF, FR, FI, FX, OX, UX, XX, VXSNAN, VXISI, VXIMZ

fnmsubs*x* **fnmsubs***x*

Floating Negative Multiply-Subtract Single (x’EC00 003C’)

fnmsubs	**fr**D,**fr**A,**fr**C,**fr**B	(Rc = ’0’)
fnmsubs.	**fr**D,**fr**A,**fr**C,**fr**B	(Rc = ’1’)

59	D	A	B	C	30	Rc
0 5	6 10	11 15	16 20	21 25	26 30	31

The following operation is performed:

```
frD ← - ([frA × frC] - frB)
```

The floating-point operand in register **fr**A is multiplied by the floating-point operand in register **fr**C. The floating-point operand in register **fr**B is subtracted from this intermediate result.

If the most-significant bit of the resultant significand is not one, the result is normalized. The result is rounded to single-precision under control of the floating-point rounding control field RN of the FPSCR, then negated and placed into **fr**D.

This instruction produces the same result obtained by negating the result of a Floating Multiply-Subtract Single (**fmsubs***x*) instruction 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 zero.
- 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’.

Other registers altered:

- Condition Register (CR1 field)
  Affected: FX, FEX, VX, OX (if Rc = ’1’)
- Floating-Point Status and Control Register:
  Affected: FPRF, FR, FI, FX, OX, UX, XX, VXSNAN, VXISI, VXIMZ

fres*x* **fres***x*

Floating Reciprocal Estimate Single (x’EC00 0030’)

fres	**fr**D,**fr**B	(Rc = ’0’)
fres.	**fr**D,**fr**B	(Rc = ’1’)

▨ Reserved

0–5	6–10	11–15	16–20	21–25	26–30	31
59	D	0 0000 ▨	B	000 00 ▨	24	Rc

The following operation is performed:

```
frD ← estimate[1/(frB)]
```

A single-precision estimate of the reciprocal of the floating-point operand in register **fr**B is placed into register **fr**D. The estimate placed into register **fr**D is correct to a precision of one part in 256 of the reciprocal of **fr**B. That is,

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

where x is the initial value in **fr**B. Note that the value placed into register **fr**D may vary between implementations, and between different executions on the same implementation.

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

*Table 8-7. **fres** Operand Values*

Operand	Result	Exception
$-\infty$	-0	None
-0	$-\infty$(1)	ZX
$+0$	$+\infty$(1)	ZX
$+\infty$	$+0$	None
SNaN	QNaN(2)	VXSNAN
QNaN	QNaN	None

Notes:
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’.

Note: The PowerPC Architecture makes no provision for a double-precision version of the **fres***x* instruction. This is because graphics applications are expected to need only the single-precision version, and no other important performance-critical applications are expected to require a double-precision version of the **fres***x* instruction.

Note: This instruction is optional in the PowerPC Architecture.

Other registers altered:

- Condition Register (CR1 field):
  Affected: FX, FEX, VX, OX (if Rc = ’1’)
- Floating-Point Status and Control Register:
  Affected: FPRF, FR (undefined), FI (undefined), FX, OX, UX, ZX, VXSNAN

frsp*x* **frsp***x*

Floating Round to Single (x’FC00 0018’)

frsp	**fr**D,**fr**B	(Rc = ’0’)
frsp.	**fr**D,**fr**B	(Rc = ’1’)

▨ Reserved

63	D	0 0000	B	12	Rc
0 5	6 10	11 15	16 20	21 30	31

The following operation is performed:

```
frD ← Round_single( frB )
```

If it is already in single-precision range, the floating-point operand in register **fr**B is placed into **fr**D. Otherwise, the floating-point operand in register **fr**B is rounded to single-precision using the rounding mode specified by FPSCR[RN] and placed into **fr**D.

The rounding is described fully in *Appendix C.4.1 Floating-Point Round to Single-Precision Model*.

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = ’1’.

Other registers altered:

- Condition Register (CR1 field):
  Affected: FX, FEX, VX, OX (if Rc = ’1’)
- Floating-Point Status and Control Register:
  Affected: FPRF, FR, FI, FX, OX, UX, XX, VXSNAN

frsqrte*x* **frsqrte***x*

Floating Reciprocal Square Root Estimate (x’FC00 0034’)

frsqrte	**fr**D,**fr**B	(Rc = ’0’)
frsqrte.	**fr**D,**fr**B	(Rc = ’1’)

▨ Reserved

63	D	0 0000 ▨	B	000 00 ▨	26	Rc
0–5	6–10	11–15	16–20	21–25	26–30	31

A double-precision estimate of the reciprocal of the square root of the floating-point operand in register **fr**B is placed into register **fr**D. The estimate placed into register **fr**D is correct to a precision of one part in 32 of the reciprocal of the square root of **fr**B. That is,

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

where x is the initial value in **fr**B. Note that the value placed into register **fr**D may vary between implementations, and between different executions on the same implementation.

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

*Table 8-8. **frsqrte** Operand Values*

Operand	Result	Exception
$-\infty$	QNaN(2)	VXSQRT
<0	QNaN(2)	VXSQRT
–0	$-\infty$(1)	ZX
+0	$+\infty$(1)	ZX
$+\infty$	+0	None
SNaN	QNaN(2)	VXSNAN
QNaN	QNaN	None

Notes:
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’.

Note: No single-precision version of the **frsqrte** instruction is provided; however, both **fr**B and **fr**D are representable in single-precision format.

Note: This instruction is optional in the PowerPC Architecture.

Other registers altered:

- Condition Register (CR1 field):
  Affected: FX, FEX, VX, OX (if Rc = ’1’)
- Floating-Point Status and Control Register:
  Affected: FPRF, FR (undefined), FI (undefined), FX, ZX, VXSNAN, VXSQRT

fsel*x* fsel*x*

Floating Select (x’FC00 002E’)

fsel	**fr**D,**fr**A,**fr**C,**fr**B	(Rc = ’0’)
fsel.	**fr**D,**fr**A,**fr**C,**fr**B	(Rc = ’1’)

63	D	A	B	C	23	Rc
0 5	6 10	11 15	16 20	21 25	26 30	31

```
if (frA) ≥ 0.0
then frD ← (frC)
else frD ← (frB)
```

The floating-point operand in register **fr**A is compared to the value zero. If the operand is greater than or equal to zero, register **fr**D is set to the contents of register **fr**C. If the operand is less than zero or is a NaN, register **fr**D is set to the contents of register **fr**B. The comparison ignores the sign of zero (that is, regards +0 as equal to –0).

Care must be taken in using **fsel** if IEEE compatibility is required, or if the values being tested can be NaNs or infinities.

For examples of uses of this instruction, see *Appendix C.3 Floating-Point Conversions* and *Appendix C.5 Floating-Point Selection*.

Note: This instruction is optional in the PowerPC Architecture.

Other registers altered:

- Condition Register (CR1 field):
  Affected: FX, FEX, VX, OX (if Rc = ’1’)

fsqrt*x* **fsqrt***x*

Floating Square Root (Double-Precision) (x’FC00 002C’)

fsqrt	**fr**D,**fr**B	(Rc = ’0’)
fsqrt.	**fr**D,**fr**B	(Rc = ’1’)

▨ Reserved

0–5	6–10	11–15	16–20	21–25	26–30	31
63	D	0 0000 ▨	B	000 00 ▨	22	Rc

The following operation is performed:

```
frD ← (Square_rootfrB)
```

The square root of the floating-point operand in register **fr**B is placed into register **fr**D.

If the most-significant bit of the resultant significand is not a one the result is normalized. The result is rounded to the target precision under control of the floating-point rounding control field RN of the FPSCR and placed into register **fr**D.

Operation with various special values of the operand is summarized in *Table 8-9*.

Table 8-9. ***frsqrt*** *with Special Operand Values*

Operand	Result	Exception
$-\infty$	QNaN(1)	VXSQRT
<0	QNaN(1)	VXSQRT
–0	–0	None
$+\infty$	$+\infty$	None
SNaN	QNaN(1)	VXSNAN
QNaN	QNaN	None

Note:
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’.

Note: This instruction is optional in the PowerPC Architecture.

Other registers altered:

- Condition Register (CR1 field):
  Affected: FX, FEX, VX, OX (if Rc = ’1’)
- Floating-Point Status and Control Register:
  Affected: FPRF, FR, FI, FX, XX, VXSNAN, VXSQRT

fsqrts*x* fsqrts*x*

Floating Square Root Single (x'EC00 002C')

fsqrts	**fr**D,**fr**B	(Rc = '0')
fsqrts.	**fr**D,**fr**B	(Rc = '1')

▨ Reserved

59	D	0 0000	B	000 00	22	Rc
0 5	6 10	11 15	16 20	21 25	26 30	31

The following operation is performed:

```
frD ← (Square_rootfrB)
```

The square root of the floating-point operand in register **fr**B is placed into register **fr**D.

If the most-significant bit of the resultant significand is not a one the result is normalized. The result is rounded to the target precision under control of the floating-point rounding control field RN of the FPSCR and placed into register **fr**D.

Operation with various special values of the operand is summarized in *Table 8-9*.

*Table 8-10. **frsqrts** with Special Operand Values*

Operand	Result	Exception
–∞	QNaN(1)	VXSQRT
<0	QNaN(1)	VXSQRT
–0	–0	None
+∞	+∞	None
SNaN	QNaN(1)	VXSNAN
QNaN	QNaN	None

Note:
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'.

Note: This instruction is optional in the PowerPC Architecture.

Other registers altered:

- Condition Register (CR1 field):
  Affected: FX, FEX, VX, OX (if Rc = '1')
- Floating-Point Status and Control Register:
  Affected: FPRF, FR, FI, FX, XX, VXSNAN, VXSQRT

fsub*x* fsub*x*

Floating Subtract (Double-Precision) (x’FC00 0028’)

fsub	**fr**D,**fr**A,**fr**B	(Rc = ’0’)
fsub.	**fr**D,**fr**A,**fr**B	(Rc = ’1’)

▨ Reserved

63	D	A	B	0 0 0 0 0	20	Rc
0 5	6 10	11 15	16 20	21 25	26 30	31

The following operation is performed:

```
frD ← (frA) - (frB)
```

The floating-point operand in register **fr**B is subtracted from the floating-point operand in register **fr**A. If the most-significant bit of the resultant significand is not a one, the result is normalized. The result is rounded to double-precision under control of the floating-point rounding control field RN of the FPSCR and placed into **fr**D.

The execution of the **fsub** instruction is identical to that of **fadd**, except that the contents of **fr**B participate in the operation with its 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’.

Other registers altered:

- Condition Register (CR1 field):
  Affected: FX, FEX, VX, OX (if Rc = ’1’)
- Floating-Point Status and Control Register:
  Affected: FPRF, FR, FI, FX, OX, UX, XX, VXSNAN, VXISI

fsubs*x* **fsubs***x*

Floating Subtract Single (x’EC00 0028’)

fsubs	**fr**D,**fr**A,**fr**B	(Rc = ’0’)
fsubs.	**fr**D,**fr**A,**fr**B	(Rc = ’1’)

▨ Reserved

0–5	6–10	11–15	16–20	21–25	26–30	31
59	D	A	B	0 0 0 0 0 ▨	20	Rc

The floating-point operand in register **fr**B is subtracted from the floating-point operand in register **fr**A. If the most-significant bit of the resultant significand is not a one, the result is normalized. The result is rounded to single-precision under control of the floating-point rounding control field RN of the FPSCR and placed into **fr**D.

The execution of the **fsubs** instruction is identical to that of **fadds**, except that the contents of **fr**B participate in the operation with its 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’.

Other registers altered:

- Condition Register (CR1 field):
  Affected: FX, FEX, VX, OX (if Rc = ’1’)
- Floating-Point Status and Control Register:
  Affected: FPRF, FR, FI, FX, OX, UX, XX, VXSNAN, VXISI

icbi

Instruction Cache Block Invalidate (x'7C00 07AC')

icbi **r**A,**r**B

▨ Reserved

31	00 000	A	B	982	0
0 – 5	6 – 10	11 – 15	16 – 20	21 – 30	31

The effective address is the sum (**r**A|0) + (**r**B).

If the block containing the byte addressed by EA is in coherency-required mode, and a block containing the byte addressed by EA is in the instruction cache of any processor, the block is made invalid in all such instruction caches, so that subsequent references cause the block to be refetched.

If the block containing the byte addressed by EA is in coherency-not-required mode, and a block containing the byte addressed by EA is in the instruction cache of this processor, the block is made invalid in that instruction cache, so that subsequent references cause the block to be refetched.

The function of this instruction is independent of the write-through, write-back, and caching-inhibited/allowed modes of the block containing the byte addressed by EA.

This instruction is treated as a load from the addressed byte with respect to address translation and memory protection. It may also be treated as a load for referenced and changed bit recording except that referenced and changed bit recording may not occur. Implementations with a combined data and instruction cache treat the **icbi** instruction as a no-op, except that they may invalidate the target block in the instruction caches of other processors if the block is in coherency-required mode.

The **icbi** instruction invalidates the block at EA (**r**A|0 + **r**B). If the processor is a multiprocessor implementation and the block is marked coherency-required, the processor will send an address-only broadcast to other processors causing those processors to invalidate the block from their instruction caches.

For faster processing, many implementations will not compare the entire EA (**r**A|0 + **r**B) with the tag in the instruction cache. Instead, they will use the bits in the EA to locate the set that the block is in, and invalidate all blocks in that set.

Other registers altered:

- None

isync isync

Instruction Synchronize (x’4C00 012C’)

isync

▨ Reserved

19	00 000	0 0000	0000 0	150	0
0 – 5	6 – 10	11 – 15	16 – 20	21 – 30	31

The **isync** instruction provides an ordering function for the effects of all instructions executed by a processor. Executing an **isync** instruction ensures that all instructions preceding the **isync** instruction have completed before the **isync** instruction completes, except that memory accesses caused by those instructions need not have been performed with respect to other processors and mechanisms. It also ensures that no subsequent instructions are initiated by the processor until after the **isync** instruction completes. Finally, it causes the processor to discard any prefetched instructions, 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 has no effect on the other processors or on their caches.

This instruction is context synchronizing.

Context synchronization is necessary after certain code sequences that perform complex operations within the processor. These code sequences are usually operating system tasks that involve memory management. For example, if an instruction A changes the memory translation rules in the memory management unit (MMU), the **isync** instruction should be executed so that the instructions following instruction A will be discarded from the pipeline and refetched according to the new translation rules.

Note: All exceptions and the **rfid** instruction are also context synchronizing.

Other registers altered:

- None

lbz lbz

Load Byte and Zero (x’8800 0000’)

lbz r**D**,d(**r**A)

34	D	A	d
0 5	6 10	11 15	16 31

```
if rA = 0 then b ← 0
else           b ← (rA)
EA ← b + EXTS(d)
rD ← (56)0 || MEM(EA, 1)
```

EA is the sum (**r**A|0) + d. The byte in memory addressed by EA is loaded into the low-order eight bits of **r**D. The remaining bits in **r**D are cleared.

Other registers altered:

- None

lbzu lbzu

Load Byte and Zero with Update (x’8C00 0000’)

lbzu **r**D,d**(r**A**)**

35	D	A	d
0 5	6 10	11 15	16 31

```
EA ← (rA) + EXTS(d)
rD← (56)0 || MEM(EA, 1)
rA← EA
```

EA is the sum (**r**A) + d. The byte in memory addressed by EA is loaded into the low-order eight bits of **r**D. The remaining bits in **r**D are cleared.

EA is placed into **r**A.

If **r**A = ’0’, or **r**A = **r**D, the instruction form is invalid.

Other registers altered:

- None

lbzux

lbzux

Load Byte and Zero with Update Indexed (x’7C00 00EE’)

lbzux **r**D,**r**A,**r**B

▨ Reserved

31	D	A	B	119	0
0–5	6–10	11–15	16–20	21–30	31

```
EA ← (rA) + (rB)
rD ← (56)0 || MEM(EA, 1)
rA ← EA
```

EA is the sum (**r**A) + (**r**B). The byte in memory addressed by EA is loaded into the low-order eight bits of **r**D. The remaining bits in **r**D are cleared.

EA is placed into **r**A.

If **r**A = ’0’ or **r**A = **r**D, the instruction form is invalid.

Other registers altered:

- None

lbzx   lbzx

Load Byte and Zero Indexed (x’7C00 00AE’)

lbzx **r**D,**r**A,**r**B

▨ Reserved

31	D	A	B	87	▨ 0
0 5	6 10	11 15	16 20	21 30	31

```
if rA = 0 then b ← 0
else      b ← (rA)
EA ← b + (rB)
rD ← (56)0 || MEM(EA, 1)
```

EA is the sum (**r**A|0) + (**r**B). The byte in memory addressed by EA is loaded into the low-order eight bits of **r**D. The remaining bits in **r**D are cleared.

Other registers altered:

- None

ld ld

Load Doubleword (x’E800 0000’)

ld **r**D,ds**(r**A**)**

58	D	A	ds	0 0
0 5	6 10	11 15	16 29	30 31

```
if rA = 0 then b ← 0
else    b ← (rA)
EA ← b + EXTS(ds || ’00’)
rD ← MEM(EA, 8)
```

EA is the sum (**r**A|0) + (ds || ’00’). The doubleword in memory addressed by EA is loaded into **r**D.

Other registers altered:

- None

ldarx ldarx

Load Doubleword and Reserve Indexed (x’7C00 00A8’)

ldarx **r**D,**r**A,**r**B

▨ Reserved

31	D	A	B	84	▨ 0
0 5	6 10	11 15	16 20	21 30	31

```
if rA = 0 then b ← 0
else      b ← (rA)
EA ← b + (rB)
RESERVE ← 1
RESERVE_ADDR ← physical_addr(EA)
rD ← MEM(EA, 8)
```

EA is the sum (**r**A|0) + (**r**B). The doubleword in memory addressed by EA is loaded into **r**D.

This instruction creates a reservation for use by a Store Doubleword Conditional Indexed (**stdcx.**) instruction. An address computed from the EA is associated with the reservation, and replaces any address previously associated with the reservation.

EA must be a multiple of eight. If it is not, either the system alignment exception handler is invoked or the results are boundedly undefined. For additional information about alignment and DSI exceptions, see *Section 6.4.3 DSI Exception (0x00300)*.

Other registers altered:

- None

ldu ldu

Load Doubleword with Update (x’E800 0001’)

ldu **r**D,ds**(r**A**)**

58	D	A	ds	0 1
0 5	6 10	11 15	16 29	30 31

```
EA ← (rA) + EXTS(ds || ’00’)
rD ← MEM(EA, 8)
rA ← EA
```

EA is the sum (**r**A) + (ds || ’00’). The doubleword in memory addressed by EA is loaded into **r**D.

EA is placed into **r**A.

If **r**A = ’0’ or **r**A = **r**D, the instruction form is invalid.

Other registers altered:

- None

ldux

Load Doubleword with Update Indexed (x’7C00 006A’)

ldux **r**D,**r**A,**r**B

▨ Reserved

31	D	A	B	53	0
0 5	6 10	11 15	16 20	21 30	31

```
EA ← (rA) + (rB)
rD ← MEM(EA, 8)
rA ← EA
```

EA is the sum (**r**A) + (**r**B). The doubleword in memory addressed by EA is loaded into **r**D.

EA is placed into **r**A.

If **r**A = ’0’ or **r**A = **r**D, the instruction form is invalid.

Other registers altered:

- None

ldx ldx

Load Doubleword Indexed (x’7C00 002A’)

ldx **r**D,**r**A,**r**B

▨ Reserved

31	D	A	B	21	▨ 0
0 5	6 10	11 15	16 20	21 30	31

```
if rA = 0 then b ← 0
else       b ← (rA)
EA ← b + (rB)
rD ← MEM(EA, 8)
```

EA is the sum (**r**A|0) + (**r**B). The doubleword in memory addressed by EA is loaded into **r**D.

Other registers altered:

- None

lfd lfd

Load Floating-Point Double (x’C800 0000’)

lfd **fr**D,d(**r**A)

50	D	A	d
0 5	6 10	11 15	16 31

```
if rA = 0 then b ← 0
else     b ← (rA)
EA ← b + EXTS(d)
frD ← MEM(EA, 8)
```

EA is the sum (**r**A|0) + d.

The doubleword in memory addressed by EA is placed into **fr**D.

Other registers altered:

- None

lfdu lfdu

Load Floating-Point Double with Update (x’CC00 0000’)

lfdu **fr**D,d(**r**A)

51	D	A	d
0 5	6 10	11 15	16 31

```
EA ← (rA) + EXTS(d)
frD ← MEM(EA, 8)
rA ← EA
```

EA is the sum (**r**A) + d.

The doubleword in memory addressed by EA is placed into **fr**D.

EA is placed into **r**A.

If **r**A = ’0’, the instruction form is invalid.

Other registers altered:

- None

lfdux

Load Floating-Point Double with Update Indexed (x’7C00 04EE’)

lfdux **fr**D,**r**A,**r**B

▨ Reserved

31	D	A	B	631	0
0 5	6 10	11 15	16 20	21 30	31

```
EA ← (rA) + (rB)
frD ←MEM(EA, 8)
rA ← EA
```

EA is the sum (**r**A) + (**r**B).

The doubleword in memory addressed by EA is placed into **fr**D.

EA is placed into **r**A.

If **r**A = ’0’, the instruction form is invalid.

Other registers altered:

- None

lfdx lfdx

Load Floating-Point Double Indexed (x’7C00 04AE’)

lfdx **fr**D,**r**A,**r**B

▨ Reserved

31	D	A	B	599	▨ 0
0 5	6 10	11 15	16 20	21 30	31

```
if rA = 0 then b ← 0
else      b ← (rA)
EA ← b + (rB)
frD ← MEM(EA, 8)
```

EA is the sum (**r**A|0) + (**r**B).

The doubleword in memory addressed by EA is placed into **fr**D.

Other registers altered:

- None

lfs **lfs**

Load Floating-Point Single (x’C000 0000’)

lfs **fr**D,d(**r**A)

48	D	A	d
0 5	6 10	11 15	16 31

```
if rA = 0 then b ← 0
else      b ← (rA)
EA ← b + EXTS(d)
frD ← DOUBLE(MEM(EA, 4))
```

EA is the sum (**r**A|0) + d.

The word in memory addressed by EA is interpreted as a floating-point single-precision operand. This word is converted to floating-point double-precision (see *Appendix C.6 Floating-Point Load Instructions*) and placed into **fr**D.

Other registers altered:

- None

lfsu

lfsu

Load Floating-Point Single with Update (x’C400 0000’)

lfsu **fr**D,d(**r**A)

49	D	A	d
0 – 5	6 – 10	11 – 15	16 – 31

```
EA ← (rA) + EXTS(d)
frD ← DOUBLE(MEM(EA, 4))
rA ← EA
```

EA is the sum (**r**A) + d.

The word in memory addressed by EA is interpreted as a floating-point single-precision operand. This word is converted to floating-point double-precision (see *Appendix C.6 Floating-Point Load Instructions*) and placed into **fr**D.

EA is placed into **r**A.

If **r**A = ’0’, the instruction form is invalid.

Other registers altered:

- None

lfsux lfsux

Load Floating-Point Single with Update Indexed (x'7C00 046E')

lfsux **fr**D,**r**A,**r**B

▨ Reserved

31	D	A	B	567	▨ 0
0 5	6 10	11 15	16 20	21 30	31

```
EA ← (rA) + (rB)
frD ← DOUBLE(MEM(EA, 4))
rA ← EA
```

EA is the sum (**r**A) + (**r**B).

The word in memory addressed by EA is interpreted as a floating-point single-precision operand. This word is converted to floating-point double-precision (see *Appendix C.6 Floating-Point Load Instructions*) and placed into **fr**D.

EA is placed into **r**A.

If **r**A = '0', the instruction form is invalid.

Other registers altered:

- None

lfsx　　　　lfsx

Load Floating-Point Single Indexed (x’7C00 042E’)

lfsx　　　　**fr**D,**r**A,**r**B

▨ Reserved

31	D	A	B	535	▨ 0
0　　5	6　　10	11　　15	16　　20	21　　30	31

```
if rA = 0 then b ← 0
else      b ← (rA)
EA ← b + (rB)
frD ← DOUBLE(MEM(EA, 4))
```

EA is the sum (**r**A|0) + (**r**B).

The word in memory addressed by EA is interpreted as a floating-point single-precision operand. This word is converted to floating-point double-precision (see *Appendix C.6 Floating-Point Load Instructions*) and placed into **fr**D.

Other registers altered:

- None

lha lha

Load Halfword Algebraic (x’A800 0000’)

lha r**D**,d(**r**A)

42	D	A	d
0 5	6 10	11 15	16 31

```
if rA = 0 then b ← 0
else        b ← (rA)
EA ← b + EXTS(d)
rD ← EXTS(MEM(EA, 2))
```

EA is the sum (**r**A|0) + d. The halfword in memory addressed by EA is loaded into the low-order 16 bits of **r**D. The remaining bits in **r**D are filled with a copy of the most-significant bit of the loaded halfword.

Other registers altered:

- None

lhau lhau

Load Halfword Algebraic with Update (x'AC00 0000')

lhau r D,d(r A)

43	D	A	d
0 5	6 10	11 15	16 31

```
EA ← (rA) + EXTS(d)
rD ← EXTS(MEM(EA, 2))
rA ← EA
```

EA is the sum (**r**A) + d. The halfword in memory addressed by EA is loaded into the low-order 16 bits of **r**D. The remaining bits in **r**D are filled with a copy of the most-significant bit of the loaded halfword.

EA is placed into **r**A.

If **r**A = '0' or **r**A = **r**D, the instruction form is invalid.

Other registers altered:

- None

lhaux lhaux

Load Halfword Algebraic with Update Indexed (x’7C00 02EE’)

lhaux **r**D,**r**A,**r**B

▨ Reserved

31	D	A	B	375	▨ 0
0 5	6 10	11 15	16 20	21 30	31

```
EA ← (rA) + (rB)
rD ← EXTS(MEM(EA, 2))
rA ← EA
```

EA is the sum (**r**A) + (**r**B). The halfword in memory addressed by EA is loaded into the low-order 16 bits of **r**D. The remaining bits in **r**D are filled with a copy of the most-significant bit of the loaded halfword.

EA is placed into **r**A.

If **r**A = ’0’ or **r**A = **r**D, the instruction form is invalid.

Other registers altered:

- None

lhax lhax

Load Halfword Algebraic Indexed (x’7C00 02AE’)

lhax **r**D,**r**A,**r**B

▨ Reserved

31	D	A	B	343	▨ 0
0 5	6 10	11 15	16 20	21 30	31

```
if rA = 0 then b ← 0
else      b ← (rA)
EA ← b + (rB)
rD ← EXTS(MEM(EA, 2))
```

EA is the sum (**r**A|0) + (**r**B). The halfword in memory addressed by EA is loaded into the low-order 16 bits of **r**D. The remaining bits in **r**D are filled with a copy of the most-significant bit of the loaded halfword.

Other registers altered:

- None

lhbrx lhbrx

Load Halfword Byte-Reverse Indexed (x’7C00 062C’)

lhbrx **r**D,**r**A,**r**B

▨ Reserved

31	D	A	B	790	0 ▨
0 5	6 10	11 15	16 20	21 30	31

```
if rA = 0 then b ← 0
else      b ← (rA)
EA ← b + (rB)
rD ← (48)0 || MEM(EA + 1, 1) || MEM(EA, 1)
```

EA is the sum (**r**A|0) + (**r**B). Bits [0–7] of the halfword in memory addressed by EA are loaded into the low-order eight bits of **r**D. Bits [8–15] of the halfword in memory addressed by EA are loaded into the subsequent low-order eight bits of **r**D. The remaining bits in **r**D are cleared.

The PowerPC Architecture cautions programmers that some implementations of the architecture may run the **lhbrx** instructions with greater latency than other types of load instructions.

Other registers altered:

- None

lhz lhz

Load Halfword and Zero (x’A000 0000’)

lhz rD,d(**r**A)

40	D	A	d
0 5	6 10	11 15	16 31

```
if rA = 0 then b ← 0
else       b ← (rA)
EA ← b + EXTS(d)
rD ← (48)0 || MEM(EA, 2)
```

EA is the sum (**r**A|0) + d. The halfword in memory addressed by EA is loaded into the low-order 16 bits of **r**D. The remaining bits in **r**D are cleared.

Other registers altered:

- None

lhzu lhzu

Load Halfword and Zero with Update (x’A400 0000’)

lhzu **r**D,d(**r**A)

41	D	A	d
0 5	6 10	11 15	16 31

```
EA ← rA + EXTS(d)
rD ← (48)0 || MEM(EA, 2)
rA ← EA
```

EA is the sum (**r**A) + d. The halfword in memory addressed by EA is loaded into the low-order 16 bits of **r**D. The remaining bits in **r**D are cleared.

EA is placed into **r**A.

If **r**A = ’0’ or **r**A = **r**D, the instruction form is invalid.

Other registers altered:

- None

lhzux lhzux

Load Halfword and Zero with Update Indexed (x’7C00 026E’)

lhzux **r**D,**r**A,**r**B

▨ Reserved

31	D	A	B	311	0 ▨
0 5	6 10	11 15	16 20	21 30	31

```
EA ← (rA) + (rB)
rD← (48)0 || MEM(EA, 2)
rA← EA
```

EA is the sum (**r**A) + (**r**B). The halfword in memory addressed by EA is loaded into the low-order 16 bits of **r**D. The remaining bits in **r**D are cleared.

EA is placed into **r**A.

If **r**A = ’0’ or **r**A = **r**D, the instruction form is invalid.

Other registers altered:

- None

lhzx lhzx

Load Halfword and Zero Indexed (x’7C00 022E’)

lhzx **r**D,**r**A,**r**B

▨ Reserved

31	D	A	B	279	0
0 5	6 10	11 15	16 20	21 30	31

```
if rA = 0 then b ← 0
else     b ← (rA)
EA ← b + (rB)
rD ← (48)0 || MEM(EA, 2)
```

EA is the sum (**r**A|0) + (**r**B). The halfword in memory addressed by EA is loaded into the low-order 16 bits of **r**D. The remaining bits in **r**D are cleared.

Other registers altered:

- None

lmw

Load Multiple Word (x’B800 0000’)

lmw r**D**,d(**r**A)

46	D	A	d
0 5	6 10	11 15	16 31

```
if rA = 0 then b ← 0
else     b ← (rA)
EA ← b + EXTS(d)
r ← rD
do while r ≤ 31
   GPR(r) ← (32)0 || MEM(EA, 4)
   r ← r + 1
   EA ← EA + 4
```

EA is the sum (**r**A|0) + d.

n = (32 – **r**D).

n consecutive words starting at EA are loaded into the low-order 32 bits of GPRs **r**D through **r**31. The high-order 32 bits of these GPRs are cleared.

EA must be a multiple of four. If it is not, either the system alignment exception handler is invoked or the results are boundedly undefined. For additional information about alignment and DSI exceptions, see *Section 6.4.3 DSI Exception (0x00300)*.

If **r**A is in the range of registers specified to be loaded, including the case in which **r**A = ’0’, the instruction form is invalid.

Note: In some implementations, this instruction is likely to have a greater latency and take longer to execute, perhaps much longer, than a sequence of individual load or store instructions that produce the same results.

Other registers altered:

- None

lswi lswi

Load String Word Immediate (x’7C00 04AA’)

lswi **r**D,**r**A,NB

▨ Reserved

31	D	A	NB	597	0 ▨
0 5	6 10	11 15	16 20	21 30	31

```
if rA = 0 then EA ← 0
else EA ← (rA)
if NB = 0 then n ← 32
elsen ← NB
r ← rD - 1
i ← 32
do while n > 0
   if i = 32 then
     r ← r + 1 (mod 32)
     GPR(r) ← 0
   GPR(r)[i-(i + 7)] ← MEM(EA, 1)
   i ← i + 8
   if i = 64 then i ← 32
   EA ← EA + 1
   n ← n - 1
```

The effective address is (**r**A|0).

Let *n* = NB if NB ≠ 0, *n* = 32 if NB = ’0’; *n* is the number of bytes to load.
Let *nr* = CEIL(*n* ÷ 4); *nr* is the number of registers to be loaded with data.

n consecutive bytes starting at EA are loaded into GPRs **r**D through **r**D + *nr* – 1. Data is loaded into the low-order four bytes of each GPR; the high-order four bytes are cleared.

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

If **r**A is in the range of registers specified to be loaded, including the case in which **r**A = ’0’, the instruction form is invalid.

Under certain conditions (for example, segment boundary crossing) the data alignment exception handler may be invoked. For additional information about data alignment exceptions, see *Section 6.4.3 DSI Exception (0x00300)*.

Note: In some implementations, this instruction is likely to have greater latency and take longer to execute, perhaps much longer, than a sequence of individual load or store instructions that produce the same results.

Other registers altered:

- None

lswx lswx

Load String Word Indexed (x'7C00 042A')

lswx **r**D,**r**A,**r**B

▨ Reserved

31	D	A	B	533	0 ▨
0 5	6 10	11 15	16 20	21 30	31

```
if rA = 0 then b ← 0
else      b ← (rA)
EA ← b + (rB)
n ← XER[57-63]
r ← rD - 1
i ← 32
rD ← undefined
 do while n > 0
  if i = 32 then
    r ← r + 1 (mod 32)
    GPR(r) ← 0
  GPR(r)[i-(i + 7)] ← MEM(EA, 1)
  i ← i + 8
  if i = 64 then i ← 32
  EA ← EA + 1
  n ← n - 1
```

EA is the sum (**r**A|0) + (**r**B). Let *n* = XER[57-63]; *n* is the number of bytes to load.
Let *nr* = CEIL(*n* ÷ 4); *nr* is the number of registers to receive data.
If *n* > 0, *n* consecutive bytes starting at EA are loaded into GPRs **r**D through **r**D + *nr* – 1. Data is loaded into the low-order four bytes of each GPR; the high-order four bytes are cleared.

Bytes are loaded left to right in each register. The sequence of registers wraps around through **r**0 if required. If the low-order four bytes of **r**D + *nr* – 1 are only partially filled, the unfilled low-order byte(s) of that register are cleared. If *n* = '0', the contents of **r**D are undefined.

If **r**A or **r**B is in the range of registers specified to be loaded, including the case in which **r**A = '0', either the system illegal instruction error handler is invoked or the results are boundedly undefined.

If **r**D = **r**A or **r**D = **r**B, the instruction form is invalid.

If **r**D and **r**A both specify GPR0, the form is invalid.

Under certain conditions (for example, segment boundary crossing) the data alignment exception handler may be invoked. For additional information about data alignment exceptions, see *Section 6.4.3 DSI Exception (0x00300)*.

Note: In some implementations, this instruction is likely to have a greater latency and take longer to execute, perhaps much longer, than a sequence of individual load or store instructions that produce the same results.

Other registers altered:
- None

lwa lwa

Load Word Algebraic (x’E800 0002’)

lwa **r**D,ds(**r**A)

58	D	A	ds	1 0
0 5	6 10	11 15	16 29	30 31

```
if rA = 0 then b ← 0
else     b ← (rA)
EA ← b + EXTS(ds || ’00’)
rD ← EXTS(MEM(EA, 4))
```

EA is the sum (**r**A|0) + (ds || ’00’). The word in memory addressed by EA is loaded into the low-order 32 bits of **r**D. The contents of the high-order 32 bits of **r**D are filled with a copy of bit [0] of the loaded word.

Other registers altered:

- None

lwarx lwarx

Load Word and Reserve Indexed (x’7C00 0028’)

lwarx **r**D,**r**A,**r**B

▨ Reserved

31	D	A	B	20	0
0 5	6 10	11 15	16 20	21 30	31

```
if rA = 0 then b ← 0
else     b ← (rA)
EA ← b + (rB)
RESERVE ← 1
RESERVE_ADDR ← physical_addr(EA)
rD ← (32)0 || MEM(EA,4)
```

EA is the sum (**r**A|0) + (**r**B).

The word in memory addressed by EA is loaded into the low-order 32 bits of **r**D. The contents of the high-order 32 bits of **r**D are cleared.

This instruction creates a reservation for use by a store word conditional indexed (**stwcx.**) instruction. The physical address computed from EA is associated with the reservation, and replaces any address previously associated with the reservation.

EA must be a multiple of four. If it is not, either the system alignment exception handler is invoked or the results are boundedly undefined. For additional information about alignment and DSI exceptions, see *Section 6.4.3 DSI Exception (0x00300)*.

When the RESERVE bit is set, the processor enables hardware snooping for the block of memory addressed by the RESERVE address. If the processor detects that another processor writes to the block of memory it has reserved, it clears the RESERVE bit. The **stwcx.** instruction will only do a store if the RESERVE bit is set. The **stwcx.** instruction sets the CR0[EQ] bit if the store was successful and clears it if it failed. The **lwarx** and **stwcx.** combination can be used for atomic read-modify-write sequences.

Note: The atomic sequence is not guaranteed, but its failure can be detected if CR0[EQ] = ’0’ after the **stwcx.** instruction.

Other registers altered:

- None

lwaux lwaux

Load Word Algebraic with Update Indexed (x’7C00 02EA’)

lwaux **r**D,**r**A,**r**B

▨ Reserved

31	D	A	B	373	▨ 0
0 5	6 10	11 15	16 20	21 30	31

```
EA ← (rA) + (rB)
rD ← EXTS(MEM(EA, 4))
rA ← EA
```

EA is the sum (**r**A) + (**r**B). The word in memory addressed by EA is loaded into the low-order 32 bits of **r**D. The high-order 32 bits of **r**D are filled with a copy of bit 0 of the loaded word.

EA is placed into **r**A.

If **r**A = ’0’ or **r**A = **r**D, the instruction form is invalid.

Other registers altered:

- None

lwax

Load Word Algebraic Indexed (x'7C00 02AA')

lwax **r**D,**r**A,**r**B

▨ Reserved

31	D	A	B	341	0
0 5	6 10	11 15	16 20	21 30	31

```
if rA = 0 then b ← 0
else     b ← (rA)
EA ← b + (rB)
rD ← EXTS(MEM(EA, 4))
```

EA is the sum (**r**A|0) + (**r**B). The word in memory addressed by EA is loaded into the low-order 32 bits of **r**D. The high-order 32 bits of **r**D are filled with a copy of bit 0 of the loaded word.

Other registers altered:

- None

lwbrx

Load Word Byte-Reverse Indexed (x’7C00 042C’)

lwbrx **r**D,**r**A,**r**B

▨ Reserved

0–5	6–10	11–15	16–20	21–30	31
31	D	A	B	534	▨ 0

```
if rA = 0 then b ← 0
else      b ← (rA)
EA ← b + (rB)
rD ← (32)0 || MEM(EA + 3, 1) || MEM(EA + 2, 1) || MEM(EA + 1, 1) || MEM(EA, 1)
```

EA is the sum (**r**A|0) + **r**B. Bits 0–7 of the word in memory addressed by EA are loaded into the low-order 8 bits of **r**D. Bits [8–15] of the word in memory addressed by EA are loaded into the subsequent low-order 8 bits of **r**D. Bits [16–23] of the word in memory addressed by EA are loaded into the subsequent low-order 8 bits of **r**D. Bits [24–31] of the word in memory addressed by EA are loaded into the subsequent low-order 8 bits of **r**D. The high-order 32 bits of **r**D are cleared.

The PowerPC Architecture cautions programmers that some implementations of the architecture may run the **lwbrx** instructions with greater latency than other types of load instructions.

Other registers altered:

- None

lwz

Load Word and Zero (x'8000 0000')

lwz r**D**,d(**r**A)

32	D	A	d
0–5	6–10	11–15	16–31

```
if rA = 0 then b ← 0
else     b ← (rA)
EA ← b + EXTS(d)
rD ← (32)0 || MEM(EA, 4)
```

EA is the sum (**r**A|0) + d. The word in memory addressed by EA is loaded into the low-order 32 bits of **r**D. The high-order 32 bits of **r**D are cleared.

Other registers altered:

- None

lwzu lwzu

Load Word and Zero with Update (x’8400 0000’)

lwzu **r**D,d(**r**A)

33	D	A	d
0 5	6 10	11 15	16 31

```
EA ← rA + EXTS(d)
rD ← (32)0 || MEM(EA, 4)
rA ← EA
```

EA is the sum (**r**A) + d. The word in memory addressed by EA is loaded into the low-order 32 bits of **r**D. The high-order 32 bits of **r**D are cleared.

EA is placed into **r**A.

If **r**A = ’0’, or **r**A = **r**D, the instruction form is invalid.

Other registers altered:

- None

lwzux lwzux

Load Word and Zero with Update Indexed (x'7C00 006E')

lwzux **r**D,**r**A,**r**B

▨ Reserved

31	D	A	B	55	0
0 5	6 10	11 15	16 20	21 30	31

```
EA ← (rA) + (rB)
rD ← (32)0 || MEM(EA, 4)
rA ← EA
```

EA is the sum (**r**A) + (**r**B). The word in memory addressed by EA is loaded into the low-order 32 bits of **r**D. The high-order 32 bits of **r**D are cleared.

EA is placed into **r**A.

If **r**A = '0', or **r**A = **r**D, the instruction form is invalid.

Other registers altered:

- None

lwzx lwzx

Load Word and Zero Indexed (x’7C00 002E’)

lwzx **r**D,**r**A,**r**B

▨ Reserved

31	D	A	B	23	0 ▨
0 5	6 10	11 15	16 20	21 30	31

```
if rA = 0 then b ← 0
else      b ← (rA)
EA ← b + rB
rD ← (32)0 || MEM(EA, 4)
```

EA is the sum (**r**A|0) + (**r**B). The word in memory addressed by EA is loaded into the low-order 32 bits of **r**D. The high-order 32 bits of **r**D are cleared.

Other registers altered:

- None

mcrf

Move Condition Register Field (x’4C00 0000’)

mcrf **crf**D,**crf**S

▨ Reserved

19	crfD	0 0	crfS	0 0	0 0 0 0 0	0 0 0 0 0 0 0 0 0 0	0
0 5	6 8	9 10	11 13	14 15	16 20	21 30	31

$$CR[(4 \times \mathbf{crf}D) \text{ to } (4 \times \mathbf{crf}D + 3)] \leftarrow CR[(4 \times \mathbf{crf}S) \text{ to } (4 \times \mathbf{crf}S + 3)]$$

The contents of condition register field **crf**S are copied into condition register field **crf**D. All other condition register fields remain unchanged.

Other registers altered:

- Condition Register (CR field specified by operand **crf**D):
  Affected: LT, GT, EQ, SO

mcrfs

mcrfs

Move to Condition Register from FPSCR (x’FC00 0080’)

mcrfs **crf**D,**crf**S

▨ Reserved

63	**crf**D	0 0	**crf**S	0 0	0 0 0 0 0	64	0
0 5	6 8	9 10	11 13	14 15	16 20	21 30	31

The contents of FPSCR field **crf**S are copied to CR field **crf**D. All exception bits copied (except FEX and VX) are cleared in the FPSCR.

Other registers altered:

- Condition Register (CR field specified by operand **crf**D):
  Affected: FX, FEX, VX, OX
- Floating-Point Status and Control Register:

Affected: FX, OX	(if **crf**S = ’0’)
Affected: UX, ZX, XX, VXSNAN	(if **crf**S = ‘1’)
Affected: VXISI, VXIDI, VXZDZ, VXIMZ	(if **crf**S = ‘2’)
Affected: VXVC	(if **crf**S = ‘3’)
Affected: VXSOFT, VXSQRT, VXCVI	(if **crf**S = ‘5’)

mfcr mfcr

Move from Condition Register (x’7C00 0026’)

mfcr **r**D

▨ Reserved

31	D	0 0 0 0 0 ▨	0 0 0 0 0 ▨	19	0
0–5	6–10	11–15	16–20	21–30	31

```
rD ← (32)0 || CR
```

The contents of the condition register (CR) are placed into the low-order 32 bits of **r**D. The high-order 32 bits of **r**D are cleared.

Other registers altered:

- None

mfocrf

Move from One Condition Register Field (x’7C20 0026’)

mfocrf **r**D,CRM

▨ Reserved

0–5	6–10	11	12–19	20	21–30	31
31	D	1	CRM	0	19	0

```
rD ← undefined
count ← 0
do i = 0 to 7
  if CRM$_i$ = 1 then
  n ← i
  count ← count + 1
if count = 1 then rD[(32+4×n) - (32+4×n+3)] ← CR[(4×n) - (4×n+3)]
```

If exactly one bit of the CRM field is set to 1, let n be the position of that bit in the field ($0 \le n \le 7$). The contents of CR field n (CR bits [(4×n) to (4×n+3)]) are placed into bits [(32+4×n) to (32+4×n + 3)] of register **r**D and the contents of the remaining bits of register **r**D are undefined. Otherwise, the contents of register **r**D are undefined.

Note: This form of the **mfocrf** instruction is intended to replace the old form of the instruction which will eventually be phased out of the architecture. The new form is backward compatible with most processors that comply with versions of the architecture that precede Version 2.01. On those processors, the new form is treated as the old form. However, on some processors that comply with versions of the architecture that precede Version 2.01 the new form of **mfocrf** may copy the contents of an SPR, possibly a privileged SPR, into register **r**D.

Other registers altered:

- None

mffs*x* mffs*x*

Move from FPSCR (x’FC00 048E’)

mffs	**fr**D	(Rc = ’0’)
mffs.	**fr**D	(Rc = ’1’)

▨ Reserved

63	D	0 0 0 0 0	0 0 0 0 0	583	Rc
0 5	6 10	11 15	16 20	21 30	31

```
frD[32-63]← FPSCR
```

The contents of the floating-point status and control register (FPSCR) are placed into the low-order bits of register **fr**D. The high-order bits of register **fr**D are undefined.

Other registers altered:

- Condition Register (CR1 field):
  Affected: FX, FEX, VX, OX (if Rc = ’1’)

mfmsr mfmsr

Move from Machine State Register (x’7C00 00A6’)

mfmsr **r**D

▨ Reserved

31	D	0 0000	0000 0	83	0
0 5	6 10	11 15	16 20	21 30	31

```
rD ← MSR
```

The contents of the MSR are placed into **r**D.

This is a supervisor-level instruction.

Other registers altered:

- None

mfspr mfspr

Move from Special-Purpose Register (x’7C00 02A6’)

mfspr **r**D,SPR

▨ Reserved

31	D	spr*	339	▨ 0
0–5	6–10	11–20	21–30	31

*Note: This is a split field.

```
n ← spr[5–9] || spr[0–4]
if length (SPR(n)) = 64 then
  rD ← SPR(n)
else
  rD ← (32)0 || SPR(n)
```

In the PowerPC UISA, the SPR field denotes a special-purpose register, encoded as shown in *Table 8-11*. The contents of the designated special-purpose register are placed into **r**D.

For special-purpose registers that are 32 bits long, the low-order 32 bits of **r**D receive the contents of the special-purpose register and the high-order 32 bits of **r**D are cleared.

Table 8-11. PowerPC UISA SPR Encodings for mfspr

SPR[1]			Register Name
Decimal	spr[5–9]	spr[0–4]	
1	00000	00001	XER
8	00000	01000	LR
9	00000	01001	CTR

Note:
1. The order of the two 5-bit halves of the SPR number is reversed compared with the actual instruction coding.

If the SPR field contains any value other than one of the values shown in *Table 8-11* (and the processor is in user mode), one of the following occurs:

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

Other registers altered:

- None

Simplified mnemonics:

mfxer	**r**D	equivalent to	**mfspr**	**r**D,**1**
mflr	**r**D	equivalent to	**mfspr**	**r**D,**8**
mfctr	**r**D	equivalent to	**mfspr**	**r**D,**9**

In the PowerPC OEA, the SPR field denotes a special-purpose register, encoded as shown in *Table 8-12*. The contents of the designated SPR are placed into **r**D. For SPRs that are 32 bits long, the low-order 32 bits of **r**D receive the contents of the SPR and the high-order 32 bits of **r**D are cleared.

SPR[0] = '1' if and only if reading the register is supervisor-level. Execution of this instruction specifying a defined and supervisor-level register when MSR[PR] = '1' will result in a privileged instruction type program exception.

If MSR[PR] = '1', the only effect of executing an instruction with an SPR number that is not shown in *Table 8-12* and has SPR[0] = '1' is to cause a supervisor-level instruction type program exception or an illegal instruction type program exception. For all other cases, MSR[PR] = '0' or SPR[0] = '0'. If the SPR field contains any value that is not shown in *Table 8-12*, either an illegal instruction type program exception occurs or the results are boundedly undefined.

Other registers altered:

- None

Table 8-12. PowerPC OEA SPR Encodings for ***mfspr***

SPR[1]			Register Name	Access
Decimal	spr[5–9]	spr[0–4]		
1	00000	00001	XER	User
8	00000	01000	LR	User
9	00000	01001	CTR	User
18	00000	10010	DSISR	Supervisor
19	00000	10011	DAR	Supervisor
22	00000	10110	DEC	Supervisor
25	00000	11001	SDR1	Supervisor
26	00000	11010	SRR0	Supervisor
27	00000	11011	SRR1	Supervisor
272	01000	10000	SPRG0	Supervisor
273	01000	10001	SPRG1	Supervisor
274	01000	10010	SPRG2	Supervisor
275	01000	10011	SPRG3	Supervisor
280	01000	11000	ASR[2]	Supervisor
282	01000	11010	EAR	Supervisor
287	01000	11111	PVR	Supervisor
1013	11111	10101	DABR	Supervisor

Note:

1. For **mtspr** and **mfspr** instructions, the SPR number coded in assembly language does not appear directly as a 10-bit binary number in the instruction. The number coded is split into two 5-bit halves that are reversed in the instruction, with the high-order five bits appearing in bits [16–20] of the instruction and the low-order five bits in bits [11–15; compared with actual instruction coding].
2. 64-bit implementations only.

mfsr

mfsr

Move from Segment Register (x’7C00 04A6’)

mfsr **r**D,SR

▨ Reserved

31	D	0	SR	0 0 0 0 0	595	0
0 5	6 10	11	12 15	16 20	21 30	31

The contents of the low-order 27 bits of the VSID field, and the contents of the K_S, K_P, N, and L fields, of the SLB entry specified by SR are placed into register **r**D, as follows:

SLBE Bit(s)	Copied to	SLB Field
62 - 88	rD[37-63]	VSID[25-51]
89 - 91	rD[33-35]	K_S K_P N
92	rD[36]	L (SBE[L] must be ‘0’)

rD[32] is set to ‘0’. The contents of **r**D[0-31] are undefined.

This is a supervisor-level instruction.

This instruction must be used only to read an SLB entry that was, or could have been, created by **mtsr** or **mtsrin** and has not subsequently been invalidated (i.e., an SLB entry in which ESID< 16, V= ’1’, VSID< 2^{27}, L= ’0’, and C= ’0’). Otherwise the contents of register **r**D are undefined.

Note: MSR[SF] must be '0' when this instruction is executed. Otherwise, the results are boundedly undefined.

Other registers altered:

- None

mfsrin mfsrin

Move from Segment Register Indirect (x’7C00 0526’)

mfsrin **r**D,**r**B

▨ Reserved

0–5	6–10	11–15	16–20	21–30	31
31	D	0 0000	B	659	0

The contents of the low-order 27 bits of the VSID field, and the contents of the K_S, K_P, N, and L fields, of the SLB entry specified by **r**B[32:35] are placed into register **r**D, as follows:

SLBE Bit(s)	Copied to	SLB Field
62 - 88	rD[37-63]	VSID[25-51]
89 - 91	rD[33-35]	K_S K_P N
92	rD[36]	L (SBE[L] must be ‘0’)

rD[32] is set to ‘0’. The contents of **r**D[0-31] are undefined.

This is a supervisor-level instruction.

Note: MSR[SF] must be '0' when this instruction is executed. Otherwise, the results are boundedly undefined.

This instruction must be used only to read an SLB entry that was, or could have been, created by **mtsr** or **mtsrin** and has not subsequently been invalidated (i.e., an SLB entry in which ESID< 16, V= ’1’, VSID< 2^{27}, L= ’0’, and C= ’0’). Otherwise the contents of register **r**D are undefined.

Other registers altered:

- None

mftb

Move from Time Base (x’7C00 02E6’)

mftb **r**D,TBR

▨ Reserved

31	D	tbr*	371	0 ▨
0–5	6–10	11–20	21–30	31

***Note:** This is a split field.

```
n ← tbr[5-9] || tbr[0-4]
if n = 268 then
  rD ← TB
else if n = 269 then
  rD ← (32)0 || TB[0-31]
```

The TBR field denotes either the Time Base or Time Base Upper, encoded as shown in *Table 8-13*. The contents of the designated register are placed into register **r**D. When reading Time Base Upper, the high-order 32 bits of register RT are set to zero.

Table 8-13. TBR Encodings for ***mftb***

TBR[1]			Register Name	Access
Decimal	tbr[5–9]	tbr[0–4]		
268	01000	01100	TBL	User
269	01000	01101	TBU	User

1. The order of the two 5-bit halves of the TBR number is reversed.

If the TBR field contains any value other than one of the values shown in *Table 8-13*, then one of the following occurs:

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

It is important to note that some implementations may implement **mftb** and **mfspr** identically, therefore, a TBR number must not match an SPR number.

For more information on the time base refer to *Section 2.2 PowerPC VEA Register Set—Time Base*.

Other registers altered:

- None

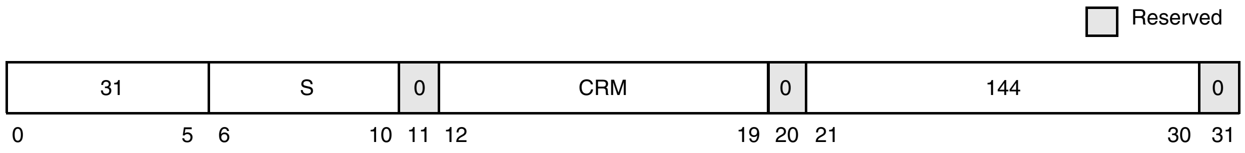
Simplified mnemonics:

mftb	**r**D	equivalent to	**mftb**	**r**D,268
mftbu	**r**D	equivalent to	**mftb**	**r**D,269

mtcrf mtcrf

Move to Condition Register Fields (x’7C00 0120’)

mtcrf CRM,**r**S



```
mask ← (4)(CRM[0]) || (4)(CRM[1]) ||... (4)(CRM[7])
CR ← (rS[32-63] & mask) | (CR & ¬ mask)
```

The contents of the low-order 32 bits of **r**S are placed into the condition register under control of the field mask specified by CRM. The field mask identifies the 4-bit fields affected. Let i be an integer in the range 0-7. If CRM(i) = ’1’, CR field i (CR bits [(4 × i) through (4 × i + 3)]) is set to the contents of the corresponding field of the low-order 32 bits of **r**S.

Note: Updating a subset of the eight fields of the condition register may have a substantially poorer performance on some implementations than updating all of the fields.

Other registers altered:

- CR fields selected by mask

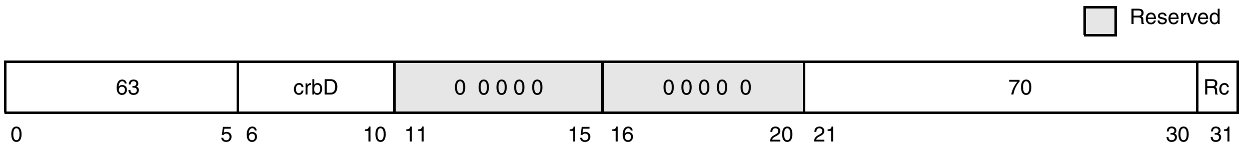
Simplified mnemonics:

mtcr	**r**S	equivalent to	**mtcrf**	0xFF,**r**S

mtfsb0*x* mtfsb0*x*

Move to FPSCR Bit 0 (x'FC00 008C')

mtfsb0	**crb**D	(Rc = '0')
mtfsb0.	**crb**D	(Rc = '1')



Bit **crb**D of the FPSCR is cleared.

Other registers altered:

- Condition Register (CR1 field):
  Affected: FX, FEX, VX, OX (if Rc = '1')
- Floating-Point Status and Control Register:
  Affected: FPSCR bit **crb**D

  Note: Bits [1] and [2] (FEX and VX) cannot be explicitly cleared.

mtfsb1*x* mtfsb1*x*

Move to FPSCR Bit 1 (x'FC00 004C')

mtfsb1	**crb**D	(Rc = '0')
mtfsb1.	**crb**D	(Rc = '1')

▨ Reserved

0–5	6–10	11–15	16–20	21–30	31
63	crbD	0 0000	0000 0	38	Rc

Bit **crb**D of the FPSCR is set.

Other registers altered:

- Condition Register (CR1 field):
  Affected: FX, FEX, VX, OX (if Rc = '1')
- Floating-Point Status and Control Register:
  Affected: FPSCR bit **crb**D and FX

  Note: Bits [1] and [2] (FEX and VX) cannot be explicitly set.

mtfsf*x* mtfsf*x*

Move to FPSCR Fields (x’FC00 058E’)

mtfsf	FM,**fr**B	(Rc = ’0’)
mtfsf.	FM,**fr**B	(Rc = ’1’)

▨ Reserved

63	▨ 0	FM	▨ 0	B	711	Rc
0 5	6	7 14	15	16 20	21 30	31

The low-order 32 bits of **fr**B are placed into the FPSCR under control of the field mask specified by FM. The field mask identifies the 4-bit fields affected. Let i be an integer in the range 0–7. If FM[i] = ’1’, FPSCR field i (FPSCR bits [(4 × i) through (4 × i + 3)]) is set to the contents of the corresponding field of the low-order 32 bits of register **fr**B.

FPSCR[FX] is altered only if FM[0] = ’1’.

Note: Updating fewer than all eight fields of the FPSCR may have a substantially poorer performance on some implementations than updating all the fields.

When FPSCR[0–3] is specified, bits [0] (FX) and [3] (OX) are set to the values of **fr**B[32] and **fr**B[35] (that is, even if this instruction causes OX to change from ‘0’ to ‘1’, FX is set from **fr**B[32] and not by the usual rule that FX is set when an exception bit changes from ‘0’ to ‘1’). Bits [1] and [2] (FEX and VX) are set according to the usual rule and not from **fr**B[33–34].

Other registers altered:

- Condition Register (CR1 field):
  Affected: FX, FEX, VX, OX (if Rc = ’1’)
- Floating-Point Status and Control Register:
  Affected: FPSCR fields selected by mask

mtfsfi*x* mtfsfi*x*

Move to FPSCR Field Immediate (x’FC00 010C’)

mtfsfi	**crf**D,IMM	(Rc = ’0’)
mtfsfi.	**crf**D,IMM	(Rc = ’1’)

▨ Reserved

63	crfD	0 0	0 0000	IMM	0	134	Rc
0 5	6 8	9 10	11 12 15	16 19	20	21 30	31

```
FPSCR[crfD] ← IMM
```

The value of the IMM field is placed into FPSCR field **crf**D.

FPSCR[FX] is altered only if **crf**D = ’0’.

When FPSCR[0–3] is specified, bits [0] (FX) and [3] (OX) are set to the values of IMM[0] and IMM[3] (that is, even if this instruction causes OX to change from ‘0’ to ‘1’, FX is set from IMM[0] and not by the usual rule that FX is set when an exception bit changes from ‘0’ to ‘1’). Bits [1] and [2] (FEX and VX) are set according to the usual rule and not from IMM[1–2].

Other registers altered:

- Condition Register (CR1 field):
  Affected: FX, FEX, VX, OX (if Rc = ’1’)
- Floating-Point Status and Control Register:
  Affected: FPSCR field **crf**D

mtmsr

Move to Machine State Register (x’7C00 0124’)

mtmsr rS,L

▨ Reserved

31	S	▨ 0000	L	▨ 0000 0	146	▨ 0
0–5	6–10	11–14	15	16–20	21–30	31

```
MSR ← (rS)
if L = 0 then
  MSR[58] ← (rS[58] | rS[49])
  MSR[59] ← (rS[59] | rS[49])
  MSR[32-47,49,50,52-57,60-63] ← rS[32-47,49,50,52-57,60-63]
else
  MSR[48,62] ← rS[48,62]
```

The MSR is set based on the contents of register **r**S and the L field.

L= ‘0’	The result of ORing bits [58] and [49] of register **r**S is placed into MSR[58]. The result of ORing bits [59] and [49] of register **r**S is placed into MSR[59]. Bits [32-47, 49, 50, 52-57, 60-63] of register **r**S are placed into the corresponding bits of the MSR. The remaining bits of the MSR are unchanged.
L= ’1’	Bits [48, 62] of **r**S are placed into the corresponding bits of the MSR. The remaining bits of the MSR are unchanged.

This instruction is a supervisor-level instruction. If L= ’0’ this instruction is context synchronizing except with respect to alterations to the [LE] bit. If L= ’1’ this instruction is execution synchronizing; in addition, the alterations of the [EE] and [RI] bits take effect as soon as the instruction completes.

Note: A reference to an **mtmsr** instruction that modifies an MSR bit other than the EE or RI bit implies L= ’0’.

Other registers altered:

- MSR

Note: **mtmsr** serves as both a basic and an extended mnemonic. The assembler will recognize an **mtmsr** mnemonic with two operands as the basic form, and an **mtmsr** mnemonic with one operand as the extended form. In the extended form the L operand is omitted and assumed to be ‘0’.

mtmsrd mtmsrd

Move to Machine State Register Doubleword (x’7C00 0164’)

mtmsrd **r**S,L

▨ Reserved

31	S	▨ 0 0 0 0	L	▨ 0 0 0 0 0	178	▨ 0
0 5	6 10	11 14	15	16 20	21 30	31

The MSR is set based on the contents of register **r**S and the L field.

L= ‘0’	The result of ORing bits [0] and [1] of register **r**S is placed into MSR[0]. The result of ORing bits [59] and [49] of register **r**S is placed into MSR[59]. Bits [1-2, 4-47, 49, 50, 52-57, 60-63] of register **r**S are placed into the corresponding bits of the MSR. The remaining bits of the MSR are unchanged.
L= ’1’	Bits [48, 62] of **r**S are placed into the corresponding bits of the MSR. The remaining bits of the MSR are unchanged.

This instruction is a supervisor-level instruction. If L= ‘0’ this instruction is context synchronizing except with respect to alterations to the [LE] bit. If L= ’1’ this instruction is execution synchronizing; in addition, the alterations of the [EE] and [RI] bits take effect as soon as the instruction completes.

Note: Processors designed prior to Version 2.01 of the architecture ignore the L field. These processors set the MSR as if L were ‘0’, and perform synchronization as if L were ‘1’. Therefore software that uses **mtmsrd** and runs on such processors must obey the following rules.

1. If L= ’1’, the contents of bits of register **r**S other than bits [48] and [62] must be such that if L were ‘0’ the instruction would not alter the contents of the corresponding MSR bits.
2. If L = ‘0’ and the instruction alters the contents of any of the MSR bits listed below, the instruction must be followed by a context synchronizing instruction or event in order to ensure that the context alteration caused by the **mtmsrd** instruction has taken effect on such processors.

To obtain the best performance on processors, if the context synchronizing instruction is **isync** the **isync** should immediately follow the **mtmsrd**. (Some such processors treat an **isync** instruction that immediately follows an **mtmsrd** instruction having L = ’0’ as a no-op, thereby avoiding the performance penalty of a second context synchronization.)

Note: **mtmsrd** serves as both a basic and an extended mnemonic. The Assembler will recognize an **mtmsrd** mnemonic with two operands as the basic form, and an **mtmsrd** mnemonic with one operand as the extended form. In the extended form the L operand is omitted and assumed to be ‘0’.

Other registers altered:

- MSR

mtocrf

Move to One Condition Register Field (x'7C20 0120')

mtocrf CRM,**r**S

▨ Reserved

31	S	1	CRM	0	144	0
0 – 5	6 – 10	11	12 – 19	20	21 – 30	31

```
count ← 0
do i = 0 to 7
  if $CRM^i$ = 1 then
  n ← i
  count ← count + 1
if count = 1 then CR[4×n to 4×n+3] ← rS[32+4×n to 32+4×n+3]
else CR ← undefined
```

If exactly one bit of the CRM field is set to 1, let n be the position of that bit in the field (0 ≤ n ≤ 7). The contents of bits [32+4×n to 32+4×n + 3] of register **r**S are placed into CR field n (CR bits [4×n to 4×n+3]). Otherwise, the contents of the Condition Register are undefined.

Note: This form of the **mtocrf** instruction is intended to replace the old form of the instruction (**mtcrf)** which will eventually be phased out of the architecture. The new form is backward compatible with most processors that comply with versions of the architecture prior to Version 2.01. On those processors, the new form is treated as the old form. However, on some processors that comply with versions of the architecture that precede Version 2.01 the new form of **mtocrf** may cause the system illegal instruction error handler to be invoked.

Other registers altered:

- CR fields selected by CRM

mtspr mtspr

Move to Special-Purpose Register (x’7C00 03A6’)

mtspr SPR,**r**S

▨ Reserved

31	S	spr*	467	0
0 5	6 10	11 20	21 30	31

***Note:** This is a split field.

```
n ← spr[5-9] || spr[0-4]
if length (SPR(n)) = 64 then
  SPR(n) ← (rS)
else
  SPR(n) ← rS[32-63]
```

In the PowerPC UISA, the SPR field denotes a special-purpose register, encoded as shown in *Table 8-14*. The contents of **r**S are placed into the designated special-purpose register. For special-purpose registers that are 32 bits long, the low-order 32 bits of **r**S are placed into the SPR.

Table 8-14. PowerPC UISA SPR Encodings for ***mtspr***

SPR[1]			Register Name
Decimal	spr[5–9]	spr[0–4]	
1	00000	00001	XER
8	00000	01000	LR
9	00000	01001	CTR

Note:
1. The order of the two 5-bit halves of the SPR number is reversed compared with actual instruction coding.

If the SPR field contains any value other than one of the values shown in *Table 8-14*, and the processor is operating in user mode, one of the following occurs:

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

Other registers altered: See *Table 8-14*.

Simplified mnemonics:

mtxer	**r**D	equivalent to	**mtspr**	1,**r**D
mtlr	**r**D	equivalent to	**mtspr**	8,**r**D
mtctr	**r**D	equivalent to	**mtspr**	9,**r**D

In the PowerPC OEA, the SPR field denotes a special-purpose register, encoded as shown in *Table 8-15*. The contents of **r**S are placed into the designated special-purpose register. For special-purpose registers that are 32 bits long, the low-order 32 bits of **r**S are placed into the SPR.

For this instruction, SPRs TBL and TBU are treated as separate 32-bit registers; setting one leaves the other unaltered.

The value of SPR[0] = ’1’ if and only if writing the register is a supervisor-level operation. Execution of this instruction specifying a defined and supervisor-level register when MSR[PR] = ’1’ results in a privileged instruction type program exception.

If MSR[PR] = ’1’ then the only effect of executing an instruction with an SPR number that is not shown in *Table 8-15* and has SPR[0] = ’1’ is to cause a privileged instruction type program exception or an illegal instruction type program exception. For all other cases, MSR[PR] = ‘0’ or SPR[0] = ‘0’, if the SPR field contains any value that is not shown in *Table 8-15*, either an illegal instruction type program exception occurs or the results are boundedly undefined.

Other registers altered: See *Table 8-15*.

*Table 8-15. PowerPC OEA SPR Encodings for **mtspr***

SPR[1]			Register Name	Access
Decimal	spr[5–9]	spr[0–4]		
1	00000	00001	XER	User
8	00000	01000	LR	User
9	00000	01001	CTR	User
18	00000	10010	DSISR	Supervisor
19	00000	10011	DAR	Supervisor
22	00000	10110	DEC	Supervisor
25	00000	11001	SDR1	Supervisor
26	00000	11010	SRR0	Supervisor
27	00000	11011	SRR1	Supervisor
272	01000	10000	SPRG0	Supervisor
273	01000	10001	SPRG1	Supervisor
274	01000	10010	SPRG2	Supervisor
275	01000	10011	SPRG3	Supervisor
280	01000	11000	ASR[2]	Supervisor
282	01000	11010	EAR	Supervisor
284	01000	11100	TBL	Supervisor
285	01000	11101	TBU	Supervisor
1013	11111	10101	DABR	Supervisor

Notes:

1. The order of the two 5-bit halves of the SPR number is reversed. For **mtspr** and **mfspr** instructions, the SPR number coded in assembly language does not appear directly as a 10-bit binary number in the instruction. The number coded is split into two 5-bit halves that are reversed in the instruction, with the high-order five bits appearing in bits [16–20] of the instruction and the low-order five bits in bits [11–15].
2. 64-bit implementations only.

mtsr mtsr

Move to Segment Register (x’7C00 01A4’)

mtsr SR,**r**S

▨ Reserved

0–5	6–10	11	12–15	16–20	21–30	31
31	S	0 ▨	SR	0 0 0 0 0 ▨	210	0 ▨

This is a supervisor-level instruction.

Note: MSR[SF] must be '0' when this instruction is executed. Otherwise, the results are boundedly undefined.

The SLB entry specified by SR is loaded from register **r**S, as follows.

SLBE Bit(s)	Set to	SLB Field(s)
0-31	0x0000 0000	ESID[0-31]
32-35	SR	ESID[32-35]
3	‘1’	V
37-61	0x00_0000 ‖ 0b0	VSID[0-24]
62-88	**r**S[37-63]	VSID[25-51]
89-91	**r**S[33-35]	K_S K_P N
92	**r**S[36]	L (**r**S[36] must be ‘0’)
93	‘0’	C

Other registers altered:

- None

mtsrin mtsrin

Move to Segment Register Indirect (x’7C00 01E4’)

mtsrin **r**S,**r**B

▨ Reserved

31	S	0 0000	B	242	0
0 5	6 10	11 15	16 20	21 30	31

This is a supervisor-level instruction.

Note: MSR[SF] must be '0' when this instruction is executed. Otherwise, the results are boundedly undefined.

The SLB entry specified by **r**B[32-35] is loaded from register **r**S, as follows.

Other registers altered:

SLBE Bit(s)	Set to	SLB Field(s)
0-31	0x0000 0000	ESID[0-31]
32-35	SR	ESID[32-35]
3	‘1’	V
37-61	0x00_0000 ‖ 0b0	VSID[0-24]
62-88	**r**S[37-63]	VSID[25-51]
89-91	**r**S[33-35]	K_S K_P N
92	**r**S[36]	L (**r**S[36] must be ‘0’)
93	‘0’	C

- None

mulhd*x* mulhd*x*

Multiply High Doubleword (x’7C00 0092’)

mulhd	**r**D,**r**A,**r**B	(Rc = ’0’)
mulhd.	**r**D,**r**A,**r**B	(Rc = ’1’)

31	D	A	B	0	73	Rc
0 5	6 10	11 15	16 20	21	22 30	31

```
prod[0-127] ← (rA) × (rB)
rD ← prod[0-63]
```

The 64-bit operands are (**r**A) and (**r**B). The high-order 64 bits of the 128-bit product of the operands are placed into **r**D.

Both the operands and the product are interpreted as signed integers.

This instruction may execute faster on some implementations if **r**B contains the operand having the smaller absolute value.

Other registers altered:

- Condition Register (CR0 field):
  Affected: LT, GT, EQ, SO (if Rc = ’1’)

  Note: The setting of CR0 bits LT, GT, and EQ is mode-dependent, and reflects overflow of the 64-bit result.

mulhdu*x*

mulhdu*x*

Multiply High Doubleword Unsigned (x’7C00 0012’)

mulhdu	**r**D,**r**A,**r**B	(Rc = ’0’)
mulhdu.	**r**D,**r**A,**r**B	(Rc = ’1’)

31	D	A	B	0	9	Rc
0 5	6 10	11 15	16 20	21	22 30	31

```
prod[0-127] ← (rA) × (rB)
rD ← prod[0-63]
```

The 64-bit operands are (**r**A) and (**r**B). The high-order 64 bits of the 128-bit product of the operands are placed into **r**D.

Both the operands and the product are interpreted as unsigned integers, except that if Rc = ’1’ the first three bits of CR0 field are set by signed comparison of the result to zero.

This instruction may execute faster on some implementations if **r**B contains the operand having the smaller absolute value.

Other registers altered:

- Condition Register (CR0 field):
  Affected: LT, GT, EQ, SO (if Rc = ’1’)

  Note: The setting of CR0 bits LT, GT, and EQ is mode-dependent, and reflects overflow of the 64-bit result.

mulhw*x* mulhw*x*

Multiply High Word (x’7C00 0096’)

mulhw	**r**D,**r**A,**r**B	(Rc = ’0’)
mulhw.	**r**D,**r**A,**r**B	(Rc = ’1’)

▨ Reserved

31	D	A	B	▨ 0	75	Rc
0–5	6–10	11–15	16–20	21	22–30	31

```
prod[0-63] ← rA[32-63] × rB[32-63]
rD[32-63] ← prod[0-31]
rD[0-31] ← undefined
```

The 64-bit product is formed from the contents of the low-order 32 bits of **r**A and **r**B. The high-order 32 bits of the 64-bit product of the operands are placed into the low-order 32 bits of **r**D. The high-order 32 bits of **r**D are undefined.

Both the operands and the product are interpreted as signed integers.

This instruction may execute faster on some implementations if **r**B contains the operand having the smaller absolute value.

Other registers altered:

- Condition Register (CR0 field):
  Affected: LT, GT, EQ, SO (if Rc = ’1’)
  LT, GT, EQ undefined (if Rc = ’1’ and 64-bit mode)

  Note: The setting of CR0 bits LT, GT, and EQ is mode-dependent, and reflects overflow of the 32-bit result.

mulhwu*x* mulhwu*x*

Multiply High Word Unsigned (x’7C00 0016’)

mulhwu	r**D**,**r**A,**r**B	(Rc = ’0’)
mulhwu.	r**D**,**r**A,**r**B	(Rc = ’1’)

▨ Reserved

31	D	A	B	0	11	Rc
0 5	6 10	11 15	16 20	21	22 30	31

```
prod[0-63] ← rA[32-63] × rB[32-63]
rD[32-63] ← prod[0-31]
rD[0-31] ← undefined
```

The 32-bit operands are the contents of the low-order 32 bits of **r**A and **r**B. The high-order 32 bits of the 64-bit product of the operands are placed into the low-order 32 bits of **r**D. The high-order 32 bits of **r**D are undefined.

Both the operands and the product are interpreted as unsigned integers, except that if Rc = ’1’ the first three bits of CR0 field are set by signed comparison of the result to zero.

This instruction may execute faster on some implementations if **r**B contains the operand having the smaller absolute value.

Other registers altered:

- Condition Register (CR0 field):

Affected: LT, GT, EQ, SO	(if Rc = ’1’)
LT, GT, EQ undefined	(if Rc = ‘1’ and 64-bit mode)

Note: The setting of CR0 bits LT, GT, and EQ is mode-dependent, and reflects overflow of the 32-bit result.

mulld*x* **mulld*x***

Multiply Low Doubleword (x’7C00 01D2’)

mulld	**r**D,**r**A,**r**B	(OE = ’0’ Rc = ’0’)
mulld.	**r**D,**r**A,**r**B	(OE = ’0’ Rc = ’1’)
mulldo	**r**D,**r**A,**r**B	(OE = ’1’ Rc = ’0’)
mulldo.	**r**D,**r**A,**r**B	(OE = ’1’ Rc = ’1’)

31	D	A	B	OE	233	Rc
0–5	6–10	11–15	16–20	21	22–30	31

```
prod[0-127] ← (rA) × (rB)
rD ← prod[64-127]
```

The 64-bit operands are the contents of **r**A and **r**B. The low-order 64 bits of the 128-bit product of the operands are placed into **r**D.

Both the operands and the product are interpreted as signed integers. The low-order 64 bits of the product are independent of whether the operands are regarded as signed or unsigned 64-bit integers. If OE = ’1’, then OV is set if the product cannot be represented in 64 bits.

This instruction may execute faster on some implementations if **r**B contains the operand having the smaller absolute value.

Other registers altered:

- Condition Register (CR0 field):
  Affected: LT, GT, EQ, SO (if Rc = ’1’)

  Note: CR0 field may not reflect the infinitely precise result if overflow occurs (see XER below).

- XER:
  Affected: SO, OV (if OE = ’1’)

  Note: The setting of the affected bits in the XER is mode-independent, and reflects overflow of the 64-bit result.

mulli

Multiply Low Immediate (x’1C00 0000’)

mulli **r**D,**r**A,SIMM

07	D	A	SIMM
0 5	6 10	11 15	16 31

```
prod[0-127] ← (rA) × EXTS(SIMM)
rD ← prod[64-127]
```

The 64-bit first operand is (**r**A). The 64-bit second operand is the sign-extended value of the SIMM field. The low-order 64-bits of the 128-bit product of the operands are placed into **r**D.

Both the operands and the product are interpreted as signed integers. The low-order 64 bits of the product are calculated independently of whether the operands are treated as signed or unsigned 64-bit integers.

This instruction can be used with **mulhd***x* or **mulhw***x* to calculate a full 128-bit product.

Other registers altered:

- None

mullw*x* mullw*x*

Multiply Low Word (x’7C00 01D6’)

mullw	**r**D,**r**A,**r**B	(OE = ’0’ Rc = ’0’)
mullw.	**r**D,**r**A,**r**B	(OE = ’0’ Rc = ’1’)
mullwo	**r**D,**r**A,**r**B	(OE = ’1’ Rc = ’0’)
mullwo.	**r**D,**r**A,**r**B	(OE = ’1’ Rc = ’1’)

31	D	A	B	OE	235	Rc
0–5	6–10	11–15	16–20	21	22–30	31

```
rD ← rA[32-63] × rB[32-63]
```

The 32-bit operands are the contents of the low-order 32 bits of **r**A and **r**B. The low-order 32 bits of the 64-bit product (**r**A) × (**r**B) are placed into **r**D.

If [OE] = ’1’, then [OV] is set if the product cannot be represented in 32 bits. Both the operands and the product are interpreted as signed integers.

This instruction can be used with **mulhw***x* to calculate a full 64-bit product.

Note: This instruction may execute faster on some implementations if **r**B contains the operand having the smaller absolute value.

Other registers altered:

- Condition Register (CR0 field):
  Affected: LT, GT, EQ, SO (if Rc = ’1’)

  Note: CR0 field may not reflect the infinitely precise result if overflow occurs (see XER below).
- XER:
  Affected: SO, OV (if OE = ’1’)

  Note: The setting of the affected bits in the XER is mode-independent, and reflects overflow of the low-order 32-bit result.

nand*x* nand*x*

NAND (x’7C00 03B8’)

nand	**r**A,**r**S,**r**B	(Rc = ’0’)
nand.	**r**A,**r**S,**r**B	(Rc = ’1’)

31	S	A	B	476	Rc
0 5	6 10	11 15	16 20	21 30	31

```
rA ← ¬ ((rS) & (rB))
```

The contents of **r**S are ANDed with the contents of **r**B and the complemented result is placed into **r**A.

A **nand** with **r**S = **r**B can be used to obtain the one's complement.

Other registers altered:

- Condition Register (CR0 field):
  Affected: LT, GT, EQ, SO (if Rc = ’1’)

neg*x* **neg*x***

Negate (x’7C00 00D0’)

neg	**r**D,**r**A	(OE = ‘0’ Rc = ‘0’)
neg.	**r**D,**r**A	(OE = ‘0’ Rc = ’1’)
nego	**r**D,**r**A	(OE = ’1’ Rc = ‘0’)
nego.	**r**D,**r**A	(OE = ’1’ Rc = ’1’)

▨ Reserved

31	D	A	▨ 0 0 0 0 0	OE	104	Rc
0 5	6 10	11 15	16 20	21	22 30	31

```
rD ← ¬ (rA) + 1
```

The value ’1’ is added to the complement of the value in **r**A, and the resulting two’s complement is placed into **r**D.

If executing in the default 64-bit mode and **r**A contains the most negative 64-bit number (0x8000_0000_0000_0000), the result is the most negative number and, if OE = ’1’, OV is set. Similarly, if executing in 32-bit mode of a 64-bit implementation and the low-order 32 bits of **r**A contains the most negative 32-bit number (0x8000_0000), then the low-order 32 bits of the result contain the most negative 32-bit number and, if OE = ’1’, OV is set.

Other registers altered:

- Condition Register (CR0 field):
  Affected: LT, GT, EQ, SO (if Rc = ’1’)
- XER:
  Affected: SO OV (if OE = ’1’)

nor*x* **nor*x***

NOR (x’7C00 00F8’)

nor	**r**A,**r**S,**r**B	(Rc = ’0’)
nor.	**r**A,**r**S,**r**B	(Rc = ’1’)

31	S	A	B	124	Rc
0 5	6 10	11 15	16 20	21 30	31

$$\mathbf{r}\mathrm{A} \leftarrow \neg\ ((\mathbf{r}\mathrm{S})\ |\ (\mathbf{r}\mathrm{B}))$$

The contents of **r**S are ORed with the contents of **r**B and the complemented result is placed into **r**A.

A **nor** with **r**S = **r**B can be used to obtain the one’s complement.

Other registers altered:

- Condition Register (CR0 field):
  Affected: LT, GT, EQ, SO (if Rc = ’1’)

Simplified mnemonics:

not	**r**D,**r**S	equivalent to	**nor**	**r**A,**r**S,**r**S

or*x* **or***x*

OR (x’7C00 0378’)

or	**r**A,**r**S,**r**B	(Rc = ’0’)
or.	**r**A,**r**S,**r**B	(Rc = ’1’)

31	S	A	B	444	Rc
0 5	6 10	11 15	16 20	21 30	31

```
rA ← (rS) | (rB)
```

The contents of **r**S are ORed with the contents of **r**B and the result is placed into **r**A.

The simplified mnemonic **mr** (shown below) demonstrates the use of the **or** instruction to move register contents.

Other registers altered:

- Condition Register (CR0 field):
  Affected: LT, GT, EQ, SO (if Rc = ’1’)

Simplified mnemonics:

mr	**r**A,**r**S	equivalent to	**or**	**r**A,**r**S,**r**S

orc*x* **orc***x*

OR with Complement (x’7C00 0338’)

orc	**r**A,**r**S,**r**B	(Rc = ‘0’)
orc.	**r**A,**r**S,**r**B	(Rc = ’1’)

31	S	A	B	412	Rc
0 5	6 10	11 15	16 20	21 30	31

```
rA ← (rS) | ¬ (rB)
```

The contents of **r**S are ORed with the complement of the contents of **r**B and the result is placed into **r**A.

Other registers altered:

- Condition Register (CR0 field):
  Affected: LT, GT, EQ, SO (if Rc = ’1’)

ori ori

OR Immediate (x’6000 0000’)

ori **r**A,**r**S,UIMM

24	S	A	UIMM
0 5	6 10	11 15	16 31

```
rA ← (rS) | ((48)0 || UIMM)
```

The contents of **r**S are ORed with 0x0000_0000_0000 || UIMM and the result is placed into **r**A.

The preferred no-op (an instruction that does nothing) is **ori 0,0,0**.

Other registers altered:

- None

Simplified mnemonics:

nop	equivalent to	**ori**	**0,0,0**

oris oris

OR Immediate Shifted (x’6400 0000’)

oris **r**A,**r**S,UIMM

25	S	A	UIMM
0 5	6 10	11 15	16 31

```
rA ← (rS) | ((32)0 || UIMM || (16)0)
```

The contents of **r**S are ORed with 0x0000_0000 || UIMM || 0x0000 and the result is placed into **r**A.

Other registers altered:

- None

rfid rfid

Return from Interrupt Doubleword (x'4C00 0024')

▨ Reserved

19	00 000 ▨	0 0000 ▨	0000 0 ▨	18	0 ▨
0 5	6 10	11 15	16 20	21 30	31

```
MSR[0] ← SRR1[0] | SRR1[1]
MSR[58] ← SRR1[58] | SRR1[49]
MSR59] ← SRR1[59] | SRR1[49]
MSR[1-2,4-32,37-41,49-50,52-57,60-63] ← SRR1[1-2,4-32,37-41,49-50,52-57,60-63]
NIA ← iea SRR0[0-61] || '00'
```

Bit [0] of SRR1 is placed into MSR[0]. If MSR[3] = '1' then bits [3,51] of SRR1 are placed into the corresponding bits of the MSR. The result of ORing bits [58] and [49] of SRR1 is placed into MSR[58]. The result of ORing bits [59] and [49] of SRR1 is placed into MSR[59]. Bits [1-2, 4-32, 37-41, 48-50, 52-57, and 60-63] of SRR1 are placed into the corresponding bits of the MSR.

If the new MSR value does not enable any pending exceptions, then the next instruction is fetched, under control of the new MSR value, from the address SRR0[0–61] || '00' (when MSR[SF] = '1') or 0x0000_0000 || SRR0[32–61] || '00' (when MSR[SF] = '0'). If the new MSR value enables one or more pending exceptions, the exception associated with the highest priority pending exception is generated; in this case the value placed into SRR0 by the exception processing mechanism is the address of the instruction that would have been executed next had the exception not occurred.

Note: An implementation may define additional MSR bits, and in this case, may also cause them to be saved to SRR1 from MSR on an exception and restored to MSR from SRR1 on an **rfid**.

This is a supervisor-level, context synchronizing instruction.

Other registers altered:

- MSR

rldcl*x* rldcl*x*

Rotate Left Doubleword then Clear Left (x'7800 0010')

rldcl	**r**A,**r**S,**r**B,MB	(Rc = '0')
rldcl.	**r**A,**r**S,**r**B,MB	(Rc = '1')

30	S	A	B	mb*	8	Rc
0 5	6 10	11 15	16 20	21 26	27 30	31

***Note:** This is a split field.

```
n ← rB[58-63]
r ← ROTL[64](rS, n)
b ← mb[5] || mb[0-4]
m ← MASK(b, 63)
rA ← r & m
```

The contents of **r**S are rotated left the number of bits specified by operand in the low-order six bits of **r**B. A mask is generated having '1' bits from bit [MB] through bit [63] and '0' bits elsewhere. The rotated data is ANDed with the generated mask and the result is placed into **r**A.

Note that the **rldcl** instruction can be used to extract and rotate bit fields using the methods shown below:

- To extract an *n*-bit field, that starts at variable bit position *b* in register **r**S, right-justified into **r**A (clearing the remaining 64 – *n* bits of **r**A), set the low-order six bits of **r**B to *b* + *n* and MB = 64 – *n*.
- To rotate the contents of a register left by variable *n* bits, set the low-order six bits of **r**B to *n* and MB = '0', and to shift the contents of a register right, set the low-order six bits of **r**B to(64 – *n*), and MB = '0'.

Other registers altered:

- Condition Register (CR0 field):
  Affected: LT, GT, EQ, SO (if Rc = '1')

Simplified mnemonics:

rotld	**r**A,**r**S,**r**B	equivalent to	**rldcl**	**r**A,**r**S,**r**B,0

rldcr*x* rldcr*x*

Rotate Left Doubleword then Clear Right (x'7800 0012')

rldcr	**r**A,**r**S,**r**B,ME	(Rc = '0')
rldcr.	**r**A,**r**S,**r**B,ME	(Rc = '1')

30	S	A	B	me*	9	Rc
0 5	6 10	11 15	16 20	21 26	27 30	31

***Note:** This is a split field.

```
n ← rB[58-63]
r ← ROTL[64](rS, n)
e ← me[5] || me[0-4]
m ← MASK(0, e)
rA ← r & m
```

The contents of **r**S are rotated left the number of bits specified by the low-order six bits of **r**B. A mask is generated having '1' bits from bit [0] through bit [ME] and 0 bits elsewhere. The rotated data is ANDed with the generated mask and the result is placed into **r**A.

Note that **rldcr** can be used to extract and rotate bit fields using the methods shown below:

- To extract an *n*-bit field, that starts at variable bit position *b* in register **r**S, left-justified into **r**A (clearing the remaining 64 – *n* bits of **r**A), set the low-order six bits of **r**B to *b* and ME = *n* – 1.
- To rotate the contents of a register left by variable *n* bits, set the low-order six bits of **r**B to *n* and ME = 63, and to shift the contents of a register right, set the low-order six bits of **r**B to(64 – *n*), and ME = 63.

Other registers altered:

- Condition Register (CR0 field):
  Affected: LT, GT, EQ, SO (if Rc = '1')

For a detailed list of simplified mnemonics for the **rldcr** instruction, refer to *Appendix E Simplified Mnemonics*.

rldic*x* rldic*x*

Rotate Left Doubleword Immediate then Clear (x'7800 0008')

rldic	**r**A,**r**S,SH,MB	(Rc = '0')
rldic.	**r**A,**r**S,SH,MB	(Rc = '1')

30	S	A	sh*	mb*	2	sh*	Rc
0 5	6 10	11 15	16 20	21 26	27 29	30	31

***Note:** This is a split field.

```
n ← sh[5] || sh[0-4]
r ← ROTL[64](rS, n)
b ← mb[5] || mb[0-4]
m ← MASK(b, ¬n)
rA ← r & m
```

The contents of **r**S are rotated left the number of bits specified by operand SH. A mask is generated having '1' bits from bit [MB] through bit [63 – SH] and 0 bits elsewhere. The rotated data is ANDed with the generated mask and the result is placed into **r**A.

The **rldic** can be used to clear and shift bit fields using the methods shown below:

- To clear the high-order *b* bits of the contents of a register and then shift the result left by *n* bits, set SH = *n* and MB = *b* – *n*.
- To clear the high-order *n* bits of a register, set SH = '0' and MB = *n*.

Other registers altered:

- Condition Register (CR0 field):
  Affected: LT, GT, EQ, SO (if Rc = '1')

Simplified mnemonics:

clrlsldi **r**A,**r**S,*b*,*n*	equivalent to	**rldic**	**r**A,**r**S,*n*,*b* – *n*

For a more detailed list of simplified mnemonics for the **rldic** instruction, refer to *Appendix E Simplified Mnemonics*.

rldicl*x* rldicl*x*

Rotate Left Doubleword Immediate then Clear Left (x’7800 0000’)

rldicl	**r**A,**r**S,SH,MB	(Rc = ’0’)
rldicl.	**r**A,**r**S,SH,MB	(Rc = ’1’)

30	S	A	sh*	mb*	0	sh*	Rc
0 5	6 10	11 15	16 20	21 26	27 29	30	31

***Note:** This is a split field.

```
n ←  sh[5] || sh[0-4]
r ←  ROTL[64](rS, n)
b ←  mb[5] || mb[0-4]
m ←  MASK(b, 63)
rA ←  r & m
```

The contents of **r**S are rotated left the number of bits specified by operand SH. A mask is generated having ’1’ bits from bit MB through bit 63 and 0 bits elsewhere. The rotated data is ANDed with the generated mask and the result is placed into **r**A.

The **rldicl** can be used to extract, rotate, shift, and clear bit fields using the methods shown below:

- To extract an *n*-bit field, that starts at bit position *b* in **r**S, right-justified into **r**A (clearing the remaining 64-*n* bits of **r**A), set SH = *b* + *n* and MB = 64 – *n*.
- To rotate the contents of a register left by *n* bits, set SH = *n* and MB = ’0’; to rotate the contents of a register right by *n* bits, set SH = (64 - *n*), and MB = ’0’.
- To shift the contents of a register right by *n* bits, set SH = 64 - *n* and MB = *n*.
- To clear the high-order *n* bits of a register, set SH = ’0’ and MB = *n*.

Other registers altered:

- Condition Register (CR0 field):
  Affected: LT, GT, EQ, SO (if Rc = ’1’)

Simplified mnemonics:

extrdi **r**A,**r**S,*n*,*b* (*n* > 0)	equivalent to	**rldicl** **r**A,**r**S,*b* **+** *n*,64 – *n*
rotldi **r**A,**r**S,*n*	equivalent to	**rldicl** **r**A,**r**S,*n*,**0**
rotrdi **r**A,**r**S,*n*	equivalent to	**rldicl** **r**A,**r**S,64 – *n*,**0**
srdi **r**A,**r**S,*n* (*n* < 64)	equivalent to	**rldicl** **r**A,**r**S,64 – *n*,*n*
clrldi **r**A,**r**S,*n* (*n* < 64)	equivalent to	**rldicl** **r**A,**r**S,**0**,*n*

rldicrx **rldicr*x***

Rotate Left Doubleword Immediate then Clear Right (x’7800 0004’)

rldicr	**r**A,**r**S,SH,ME	(Rc = ’0’)
rldicr.	**r**A,**r**S,SH,ME	(Rc = ’1’)

30	S	A	sh*	me*	1	sh*	Rc
0 5	6 10	11 15	16 20	21 26	27 29	30	31

***Note:** This is a split field.

```
n ← sh[5] || sh[0-4]
r ← ROTL[64](rS, n)
e ← me[5] || me[0-4]
m ← MASK(0, e)
rA ← r & m
```

The contents of **r**S are rotated left the number of bits specified by operand SH. A mask is generated having ’1’ bits from bit [0] through bit [ME] and ‘0’ bits elsewhere. The rotated data is ANDed with the generated mask and the result is placed into **r**A.

The **rldicr** can be used to extract, rotate, shift, and clear bit fields using the methods shown below:

- To extract an *n*-bit field, that starts at bit position *b* in **r**S, left-justified into **r**A (clearing the remaining 64-*n* bits of **r**A), set SH = *b* and ME = *n* – 1.
- To rotate the contents of a register left (right) by *n* bits, set SH = *n* (64 – *n*) and ME = 63.
- To shift the contents of a register left by *n* bits, by setting SH = *n* and ME = 63 – *n*.
- To clear the low-order *n* bits of a register, by setting SH = ’0’ and ME = 63 – *n*.

Other registers altered:

- Condition Register (CR0 field):
  Affected: LT, GT, EQ, SO (if Rc = ’1’)

Simplified mnemonics:

extldi	**r**A,**r**S,*n*,*b*	equivalent to	**rldicr**	**r**A,**r**S,*b*,*n* – 1
sldi	**r**A,**r**S,*n*	equivalent to	**rldicr**	**r**A,**r**S,*n*,63 – *n*
clrrdi	**r**A,**r**S,*n*	equivalent to	**rldicr**	**r**A,**r**S,**0**,63 – *n*

rldimi*x* rldimi*x*

Rotate Left Doubleword Immediate then Mask Insert (x’7800 000C’)

rldimi	**r**A,**r**S,SH,MB	(Rc = ’0’)
rldimi.	**r**A,**r**S,SH,MB	(Rc = ’1’)

30	S	A	sh*	mb*	3	sh*	Rc
0 5	6 10	11 15	16 20	21 26	27 29	30	31

*Note: This is a split field.

```
n ← sh[5] || sh[0-4]
r ← ROTL[64](rS, n)
b ← mb[5] || mb[0-4]
m ← MASK(b, ¬ n)
rA ← (r & m) | (rA & ¬ m)
```

The contents of **r**S are rotated left the number of bits specified by operand SH. A mask is generated having ’1’ bits from bit MB through bit 63 – SH and 0 bits elsewhere. The rotated data is inserted into **r**A under control of the generated mask.

Note: **rldimi** can be used to insert an *n*-bit field, that is right-justified in **r**S, into **r**A starting at bit position *b*, by setting SH = 64 – (*b* + *n*) and MB = *b*.

Other registers altered:

- Condition Register (CR0 field):
  Affected: LT, GT, EQ, SO (if Rc = ’1’)

Simplified mnemonics:

insrdi	**r**A,**r**S,*n*,*b*	equivalent to	**rldimi**	**r**A,**r**S,64 – (*b* + *n*),*b*

For a more detailed list of simplified mnemonics for the **rldimi** instruction, refer to *Appendix E Simplified Mnemonics*.

rlwimi*x* rlwimi*x*

Rotate Left Word Immediate then Mask Insert (x'5000 0000')

rlwimi	**r**A,**r**S,SH,MB,ME	(Rc = '0')
rlwimi.	**r**A,**r**S,SH,MB,ME	(Rc = '1')

20	S	A	SH	MB	ME	Rc
0 5	6 10	11 15	16 20	21 25	26 30	31

```
n ← SH
r ← ROTL[32](rS[32-63], n)
m ← MASK(MB + 32, ME + 32)
rA ← (r & m) | (rA & ¬ m)
```

The contents of **r**S are rotated left the number of bits specified by operand SH. A mask is generated having '1' bits from bit [MB + 32] through bit [ME + 32] and '0' bits elsewhere. The rotated data is inserted into **r**A under control of the generated mask.

rlwimi can be used to insert a bit field into the contents of **r**A using the methods shown below:

- To insert an *n*-bit field, that is left-justified in the low-order 32 bits of **r**S, into the high-order 32 bits of **r**A starting at bit position *b*, set SH = 32 – *b*, MB = *b*, and ME = (*b* + *n*) – 1.
- To insert an *n*-bit field, that is right-justified in the low-order 32 bits of **r**S, into the high-order 32 bits of **r**A starting at bit position *b*, set SH = 32 – (*b* + *n*), MB = *b*, and ME = (*b* + *n*) – 1.

Other registers altered:

- Condition Register (CR0 field):
  Affected: LT, GT, EQ, SO (if Rc = '1')

Simplified mnemonics:

inslwi **r**A,**r**S,*n*,*b*	equivalent to **rlwimi**	**r**A,**r**S,32 – *b*,*b*,*b* + *n* – 1
insrwi **r**A,**r**S,*n*,*b* (n > 0)	equivalent to **rlwimi**	**r**A,**r**S,32 – (*b* + *n*),*b*,(*b* + *n*) – 1

rlwinm*x* rlwinm*x*

Rotate Left Word Immediate then AND with Mask (x'5400 0000')

rlwinm	r A,**r**S,SH,MB,ME	(Rc = '0')
rlwinm.	r A,**r**S,SH,MB,ME	(Rc = '1')

21	S	A	SH	MB	ME	Rc
0 5	6 10	11 15	16 20	21 25	26 30	31

```
n ←  SH
r ←  ROTL[32](rS[32-63], n)
m ←  MASK(MB + 32, ME + 32)
rA ←  r & m
```

The contents of **r**S[32-63] are rotated left the number of bits specified by operand SH. A mask is generated having '1' bits from bit [MB + 32] through bit [ME + 32] and '0' bits elsewhere. The rotated data is ANDed with the generated mask and the result is placed into **r**A. The upper 32 bits of **r**A are cleared.

rlwinm can be used to extract, rotate, shift, and clear bit fields using the methods shown below:

- To extract an *n*-bit field, that starts at bit position *b* in the high-order 32 bits of **r**S, right-justified into **r**A (clearing the remaining 32 – *n* bits of **r**A), set SH = *b* + *n*, MB = 32 – *n*, and ME = 31.
- To extract an *n*-bit field, that starts at bit position *b* in the high-order 32 bits of **r**S, left-justified into **r**A (clearing the remaining 32 – *n* bits of **r**A), set SH = *b*, MB = '0', and ME = *n* – 1.
- To rotate the contents of a register left (or right) by *n* bits, set SH = *n* (32 – *n*), MB = '0', and ME = 31.
- To shift the contents of a register right by *n* bits, by setting SH = 32 – *n*, MB = *n*, and ME = 31. It can be used to clear the high-order *b* bits of a register and then shift the result left by *n* bits by setting SH = *n*, MB = *b* – *n* and ME = 31 – *n*.
- To clear the low-order *n* bits of a register, by setting SH = '0', MB = '0', and ME = 31 – *n*.

For all uses mentioned, the high-order 32 bits of **r**A are cleared.

Other registers altered:

- Condition Register (CR0 field):
  Affected: LT, GT, EQ, SO (if Rc = '1')

Simplified mnemonics:

extlwi **r**A,**r**S,*n*,*b* (*n* > 0)	equivalent to	**rlwinm** **r**A,**r**S,b,**0**,*n* – 1
extrwi **r**A,**r**S,*n*,*b* (*n* > 0)	equivalent to	**rlwinm** **r**A,**r**S,b + *n*,32 – *n*,**31**
rotlwi **r**A,**r**S,*n*	equivalent to	**rlwinm** **r**A,**r**S,*n*,**0**,**31**
rotrwi **r**A,**r**S,*n*	equivalent to	**rlwinm** **r**A,**r**S,**32** – *n*,**0**,**31**
slwi **r**A,**r**S,*n* (*n* < 32)	equivalent to	**rlwinm** **r**A,**r**S,*n*,**0**,31–*n*
srwi **r**A,**r**S,*n* (*n* < 32)	equivalent to	**rlwinm** **r**A,**r**S,32 **–** *n*,*n*,**31**
clrlwi **r**A,**r**S,*n* (*n* < 32)	equivalent to	**rlwinm** **r**A,**r**S,**0**,*n*,**31**
clrrwi **r**A,**r**S,*n* (*n* < 32)	equivalent to	**rlwinm** **r**A,**r**S,**0**,**0**,31 – *n*
clrlslwi **r**A,**r**S,*b*,*n* (*n* ≤ *b* < 32)	equivalent to	**rlwinm** **r**A,**r**S,*n*,*b* – *n*,31 – *n*

rlwnm*x* **rlwnm*x***

Rotate Left Word then AND with Mask (x’5C00 0000’)

rlwnm	**r**A,**r**S,**r**B,MB,ME	(Rc = ’0’)
rlwnm.	**r**A,**r**S,**r**B,MB,ME	(Rc = ’1’)

23	S	A	B	MB	ME	Rc
0 5	6 10	11 15	16 20	21 25	26 30	31

```
n ←  rB[59-63]
r ←  ROTL[32](rS[32-63], n)
m ←  MASK(MB + 32, ME + 32)
rA ←  r & m
```

The contents of **r**S are rotated left the number of bits specified by the low-order five bits of **r**B. A mask is generated having ‘1’ bits from bit [MB + 32] through bit [ME + 32] and ‘0’ bits elsewhere. The rotated data is ANDed with the generated mask and the result is placed into **r**A.

rlwnm can be used to extract and rotate bit fields using the methods shown as follows:

- To extract an *n*-bit field, that starts at variable bit position *b* in the high-order 32 bits of **r**S, right-justified into **r**A (clearing the remaining 32 – *n* bits of **r**A), by setting the low-order five bits of **r**B to *b* + *n*, MB = 32 – *n*, and ME = 31.
- To extract an *n*-bit field, that starts at variable bit position *b* in the high-order 32 bits of **r**S, left-justified into **r**A (clearing the remaining 32 – *n* bits of **r**A), by setting the low-order five bits of **r**B to *b*, MB = ’0’, and ME = *n* – 1.
- To rotate the contents of a register left (or right) by *n* bits, by setting the low-order five bits of **r**B to *n* (32-*n*), MB = ’0’, and ME = 31.

For all uses mentioned, the high-order 32 bits of **r**A are cleared.

Other registers altered:

- Condition Register (CR0 field):
  Affected: LT, GT, EQ, SO (if Rc = ’1’)

Simplified mnemonics:

rotlw	**r**A,**r**S,**r**B	equivalent to	**rlwnm**	**r**A,**r**S,**r**B,**0,31**

sc sc

System Call (x’4400 0002’)

▨ Reserved

0–5	6–10	11–15	16–29	30	31
17	00 000	0 0000	0000 0000 0000 00	1	0

In the PowerPC UISA, the **sc** instruction calls the operating system to perform a service. When control is returned to the program that executed the system call, the content of the registers depends on the register conventions used by the program providing the system service.

This instruction is context synchronizing, as described in *Section 4.1.5.1 Context Synchronizing Instructions*.

Other registers altered:

- Dependent on the system service

In PowerPC OEA, the **sc** instruction does the following:

```
SRR0 ←iea CIA + 4
SRR1[33-36,42-47] ← 0
SRR1[0] ← MSR[0]
MSR ←  new_value (see below)
NIA ← 0x0000 _0000_0000_0C00
```

The EA of the instruction following the **sc** instruction is placed into SRR0. Bits [0-32, 37-41, 48-63] of the MSR are placed into the corresponding bits of SRR1, and bits [33–36 and 42–47]of SRR1 are set to zero.

Note: An implementation may define additional MSR bits, and in this case, may also cause them to be saved to SRR1 from MSR on an exception and restored to MSR from SRR1 on an **rfid**.

Then a system call exception is generated. The exception causes the MSR to be altered as described in *Section 6.4 Exception Definitions*.

The exception causes the next instruction to be fetched from interrupt vector 0x00C00.

Note: **sc** serves as both a basic and an extended mnemonic. The Assembler recognizes an **sc** mnemonic with one operand as the basic form, and an **sc** mnemonic with no operand as the extended form.

Other registers altered:

- SRR0
- SRR1
- MSR

slbia slbia

SLB Invalidate All (x’7C00 03E4’)

▨ Reserved

31	00 000	0 0000	0000 0	498	0
0 5	6 10	11 15	16 20	21 30	31

```
for each SLB entry except SLB entry 0 SLBE[V] ← 0
all other fields of SLBE ← undefined
```

For all SLB entries except SLB entry 0, the [V] bit in the entry is set to 0, making the entry invalid, and the remaining fields of the entry are set to undefined values. SLB entry 0 is not altered.

Note: If **slbia** is executed when instruction address translation is enabled (MSR[IR]= ’1’), software can ensure that attempting to fetch the instruction following the **slbia** does not cause an Instruction Segment interrupt by placing the **slbia** and the subsequent instruction in the effective segment mapped by SLB entry 0. (This assumes that no other interrupts occur between executing the **slbia** and executing the subsequent instruction.)

This instruction is supervisor-level.

It is not necessary that the ASR point to a valid segment table when issuing **slbia**.

Other registers altered:

- None

slbie slbie

SLB Invalidate Entry (x’7C00 0364’)

slbie **r**B

▨ Reserved

31	00 000	0 0000	B	434	0
0 5	6 10	11 15	16 20	21 30	31

```
esid ← (rB)0:35
class ← (rB)36
if class = SLBE[C] for SLB entry that translates
   or most recently translated esid
then for SLB entry (if any) that translates esid
   SLBE[V] ← 0
   all other fields of SLBE ← undefined
else translation of esid ← undefined
```

Let the Effective Segment ID (ESID) be **r**B[0-35]. Let the class be **r**B[36]. The class value must be the same as the class value in the SLB entry that translates the ESID, or the class value that was in the SLB entry that most recently translated the ESID if the translation is no longer in the SLB. If the class value is not the same, the results of translating effective addresses for which EA[0-35] = ESID are undefined, and the next paragraph need not apply.

If the SLB contains an entry that translates the specified ESID, the [V] bit in that entry is set to ‘0’, making the entry invalid, and the remaining fields of the entry are set to undefined values.

rB[37-63] must be zeroes.

If this instruction is executed in 32-bit mode, **r**B[0-31] must be zeros (i.e., the ESID must be in the range [0-15]).

This instruction is supervisor-level.

Note: If the optional “Bridge” facility is implemented, the Move To Segment Register instructions create SLB entries in which the class value is ‘0’.

Other registers altered:

- None

slbmfee slbmfee

SLB Move From Entry ESID (x’7C00 0726’)

slbmfee **r**D, **r**B

▨ Reserved

31	D	▨ 00 000	B	915	0
0 5	6 10	11 15	16 20	21 30	31

If the SLB entry specified by bits [52-63] of register **r**B is valid (V= ’1’), the contents of the ESID and V fields of the entry are placed into register **r**D.

rD

▨ Reserved

ESID	V	▨ 0s
0 35	36	37 63

rB

▨ 0s	Index
0 51	52 63

```
rD[0-35] ESID
rD[36] V
rD[37-63] must be 0b000|| 0x00_0000
rB[0-51] must be 0x0_0000_0000_0000
rB[52-6]3 index, which selects the SLB entry
```

If the SLB entry specified by bits [52-63] of register **r**B is invalid (V= ’0’), **r**D[36] is set to 0 and the contents of **r**D[0-35] and **r**D[37-63] are undefined. The high-order bits of **r**B[52-63] that correspond to SLB entries beyond the size of the SLB provided by the implementation must be zeros.

This instruction is supervisor-level.

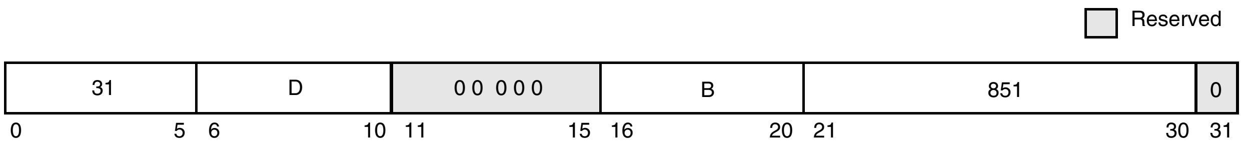
Other registers altered:

- None

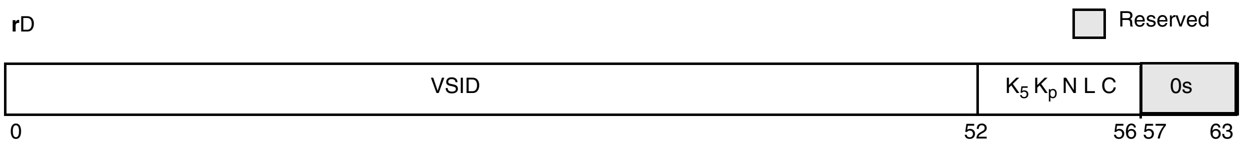
slbmfev slbmfev

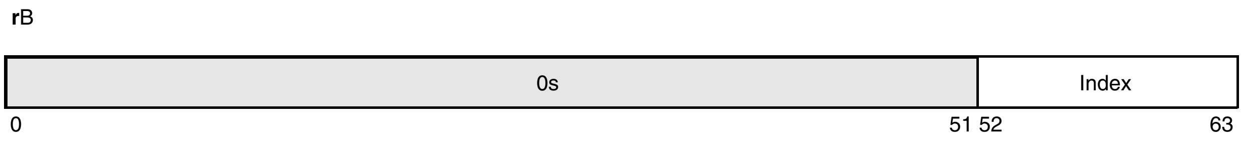
SLB Move From Entry VSID (x’7C00 06A6’)

slbmfev **r**D, **r**B



If the SLB entry specified by bits [52-63] of register **r**B is valid (V= ’1’), the contents of the VSID, K_S, K_P, N, L, and C fields of the entry are placed into register **r**D.





```
rD[0-51] VSID
rD[52] Ks
rD[53] KP
rD[54] N
rD[55] L
rD[56] C
rD[57-63] must be 0b000_0000
rB[0-51] must be 0x0_0000_0000_0000
rB[52-63] index, which selects the SLB entry
```

On implementations that support a virtual address size of only n bits, n< 80, **r**D[0 to 79- n] are set to zeros. If the SLB entry specified by bits [52-63] of register **r**B is invalid (V= ’0’), the contents of register **r**D are undefined. The high-order bits of **r**B[52-63] that correspond to SLB entries beyond the size of the SLB provided by the implementation must be zeros.

This is a supervisor-level instruction.

Other registers altered:

- None

slbmte slbmte

SLB Move To Entry (x’7C00 0324’)

slbmte **r**S, **r**B

▨ Reserved

31	S	00 000	B	402	0
0 – 5	6 – 10	11 – 15	16 – 20	21 – 30	31

The SLB entry specified by bits [52-63] of register **r**B is loaded from register **r**S and from the remainder of register **r**B.

rS

▨ Reserved

VSID	K_5K_pNLC	0s
0 – 51	52 – 56	63

rB

▨ Reserved

ESID	V	0s	Index
0 – 35	36	37 – 51	52 – 63

```
rS[0-51] VSID
rS[52]Ks
rS[53] Kp
rS[54] N
rS[55]L
rS[56] C
rS[57-63] must be 0b000_0000
rB[0-35] ESID
rB[36] V
rB[37-5]1 must be 0b000 ‖0x000
rB[52-63] index, which selects the SLB entry
```

On implementations that support a virtual address size of only n bits, n< 80, **r**S[0 to 79- n] must be zeros. The high-order bits of **r**B[52-63] that correspond to SLB entries beyond the size of the SLB provided by the implementation must be zeros. If this instruction is executed in 32-bit mode, **r**B[0-31] must be zeros (i.e., the ESID must be in the range 0-15). This instruction cannot be used to invalidate an SLB. This is a supervisor-level instruction.

Other registers altered:

- None

sld*x* **sld***x*

Shift Left Doubleword (x’7C00 0036’)

sld	**r**A,**r**S,**r**B	(Rc = ’0’)
sld.	**r**A,**r**S,**r**B	(Rc = ’1’)

31	S	A	B	27	Rc
0 5	6 10	11 15	16 20	21 30	31

```
n ← rB[58-63]
r ← ROTL[64](rS, n)
if rB[57] = 0 then
  m ← MASK(0, 63 - n)
else m ← (64)0
rA ← r & m
```

The contents of **r**S are shifted left the number of bits specified by the low-order seven bits of **r**B. Bits shifted out of position 0 are lost. Zeros are supplied to the vacated positions on the right. The result is placed into **r**A. Shift amounts from 64 to 127 give a zero result.

Other registers altered:

- Condition Register (CR0 field):
  Affected: LT, GT, EQ, SO (if Rc = ’1’)

slw*x* slw*x*

Shift Left Word (x’7C00 0030’)

slw	**r**A,**r**S,**r**B	(Rc = ’0’)
slw.	**r**A,**r**S,**r**B	(Rc = ’1’)

31	S	A	B	24	Rc
0 5	6 10	11 15	16 20	21 30	31

```
n← rB[59-63]
r ← ROTL[32](rS[32-63], n)
if rB[58] = 0 then
m ← MASK(32, 63 - n)
else m ← (64)0
rA← r & m
```

The contents of the low-order 32 bits of **r**S are shifted left the number of bits specified by the low-order six bits of **r**B. 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 the low-order 32 bits of **r**A. The high-order 32 bits of **r**A are cleared. Shift amounts from 32 to 63 give a zero result.

Other registers altered:

- Condition Register (CR0 field):
  Affected: LT, GT, EQ, SO (if Rc = ’1’)

srad*x* srad*x*

Shift Right Algebraic Doubleword (x'7C00 0634')

srad	**r**A,**r**S,**r**B	(Rc = '0')
srad.	**r**A,**r**S,**r**B	(Rc = '1')

31	S	A	B	794	Rc
0–5	6–10	11–15	16–20	21–30	31

```
n ← rB[58-63]
r ← ROTL[64](rS, 64 - n)
if rB[57] = 0 then
  m ← MASK(n, 63)
else m ← (64)0
S ← rS[0]
rA ← (r & m) | (((64)S) & ¬ m)
XER[CA] ← S & ((r & ¬ m) ¦ 0)
```

The contents of **r**S are shifted right the number of bits specified by the low-order seven bits of **r**B. Bits shifted out of position 63 are lost. Bit [0] of **r**S is replicated to fill the vacated positions on the left. The result is placed into **r**A. XER[CA] is set if **r**S is negative and any '1' bits are shifted out of position 63; otherwise XER[CA] is cleared. A shift amount of zero causes **r**A to be set equal to **r**S, and XER[CA] to be cleared. Shift amounts from 64 to 127 give a result of 64 sign bits in **r**A, and cause XER[CA] to receive the sign bit of **r**S.

Note: The **srad** instruction, followed by **addze**, can by used to divide quickly by 2^n. The setting of the CA bit, by **srad**, is independent of mode.

Other registers altered:

- Condition Register (CR0 field):
  Affected: LT, GT, EQ, SO (if Rc = '1')
- XER:
  Affected: CA

sradi*x* **sradi***x*

Shift Right Algebraic Doubleword Immediate (x’7C00 0674’)

sradi	**r**A,**r**S,SH	(Rc = ’0’)
sradi.	**r**A,**r**S,SH	(Rc = ’1’)

31	S	A	sh*	413	sh*	Rc
0 5	6 10	11 15	16 20	21	30	31

***Note:** This is a split field.

```
n ← sh[5] || sh[0-4]
r ← ROTL[64](rS, 64 - n)
m ← MASK(n, 63)
S ← rS[0]
rA ← (r & m) | (((64)S) & ¬ m)
XER[CA] ← S & ((r & ¬ m) ≠ 0)
```

The contents of **r**S are shifted right SH bits. Bits shifted out of position 63 are lost. Bit 0 of **r**S is replicated to fill the vacated positions on the left. The result is placed into **r**A. XER[CA] is set if **r**S is negative and any ‘1’ bits are shifted out of position 63; otherwise XER[CA] is cleared. A shift amount of zero causes **r**A to be set equal to **r**S, and XER[CA] to be cleared.

Note: The **sradi** instruction, followed by **addze**, can by used to divide quickly by 2^n. The setting of the XER[CA] bit, by **sradi**, is independent of mode.

Other registers altered:

- Condition Register (CR0 field):
  Affected: LT, GT, EQ, SO (if Rc = ’1’)
- XER:
  Affected: CA

sraw*x* **sraw***x*

Shift Right Algebraic Word (x'7C00 0630')

sraw	**r**A,**r**S,**r**B	(Rc = '0')
sraw.	**r**A,**r**S,**r**B	(Rc = '1')

31	S	A	B	792	Rc
0 5	6 10	11 15	16 20	21 30	31

```
n← rB[59-63]
r ← ROTL[32](rS[32-63], 64 - n)
if rB[58] = 0 then
m← MASK(n + 32, 63)
else m← (64)0
S ← rS[32]
rA ← r & m | (64)S & ¬ m
XER[CA] ← S & (r & ¬ m[32-63] ≠ 0
```

The contents of the low-order 32 bits of **r**S are shifted right the number of bits specified by the low-order six bits of **r**B. Bits shifted out of position 63 are lost. Bit [32] of **r**S is replicated to fill the vacated positions on the left. The 32-bit result is placed into the low-order 32 bits of **r**A. Bit [32] of **r**S is replicated to fill the high-order 32 bits of **r**A. XER[CA] is set if the low-order 32 bits of **r**S contain a negative number and any '1' bits are shifted out of position 63; otherwise XER[CA] is cleared. A shift amount of zero causes **r**A to receive the sign-extended value of the low-order 32 bits of **r**S, and XER[CA] to be cleared. Shift amounts from 32 to 63 give a result of 64 sign bits, and cause XER[CA] to receive the sign bit of the low-order 32 bits of **r**S.

Note: The **sraw** instruction, followed by **addze**, can by used to divide quickly by 2^n. The setting of the XER[CA] bit, by **sraw**, is independent of mode.

Other registers altered:

- Condition Register (CR0 field):
  Affected: LT, GT, EQ, SO (if Rc = '1')
- XER:
  Affected: CA

srawi*x* **srawi***x*

Shift Right Algebraic Word Immediate (x’7C00 0670’)

srawi	**r**A,**r**S,SH	(Rc = ’0’)
srawi.	**r**A,**r**S,SH	(Rc = ’1’)

31	S	A	SH	824	Rc
0 5	6 10	11 15	16 20	21 30	31

```
n ← SH
r ← ROTL[32](rS[32-63], 64 - n)
m← MASK(n + 32, 63)
S ← rS[32]
rA ← r & m | (64)S & ¬ m
XER[CA] ← S & ((r & ¬ m)[32-63] ≠ 0)
```

The contents of the low-order 32 bits of **r**S are shifted right SH bits. Bits shifted out of position 63 are lost. Bit [32] of **r**S is replicated to fill the vacated positions on the left. The 32-bit result is placed into the low-order 32 bits of **r**A. Bit [32] of **r**S is replicated to fill the high-order 32 bits of **r**A. XER[CA] is set if the low-order 32 bits of **r**S contain a negative number and any ‘1’ bits are shifted out of position 63; otherwise XER[CA] is cleared. A shift amount of zero causes **r**A to receive the sign-extended value of the low-order 32 bits of **r**S, and XER[CA] to be cleared.

Note: The **srawi** instruction, followed by **addze**, can be used to divide quickly by 2^n. The setting of the CA bit, by **srawi**, is independent of mode.

Other registers altered:

- Condition Register (CR0 field):
  Affected: LT, GT, EQ, SO (if Rc = ’1’)
- XER:
  Affected: CA

srd*x* srd*x*

Shift Right Doubleword (x’7C00 0436’)

srd	**r**A,**r**S,**r**B	(Rc = ’0’)
srd.	**r**A,**r**S,**r**B	(Rc = ’1’)

31	S	A	B	539	Rc
0 5	6 10	11 15	16 20	21 30	31

```
n ← rB[58-63]
r ← ROTL[64](rS, 64 - n)
if rB[57] = 0 then
  m ← MASK(n, 63)
else m ← (64)0
rA ← r & m
```

The contents of **r**S are shifted right the number of bits specified by the low-order seven bits of **r**B. Bits shifted out of position 63 are lost. Zeros are supplied to the vacated positions on the left. The result is placed into **r**A. Shift amounts from 64 to 127 give a zero result.

Other registers altered:

- Condition Register (CR0 field):
  Affected: LT, GT, EQ, SO (if Rc = ’1’)

srw*x* **srw***x*

Shift Right Word (x’7C00 0430’)

srw	**r**A,**r**S,**r**B	(Rc = ’0’)
srw.	**r**A,**r**S,**r**B	(Rc = ’1’)

31	S	A	B	536	Rc
0 5	6 10	11 15	16 20	21 30	31

```
n ← rB[58-63]
r ← ROTL[32](rS[32-63], 64 - n)
if rB[58] = 0 then
  m ← MASK(n + 32, 63)
else m ← (64)0
rA ← r & m
```

The contents of the low-order 32 bits of **r**S are shifted right the number of bits specified by the low-order six bits of **r**B. 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 the low-order 32 bits of **r**A. The high-order 32 bits of **r**A are cleared. Shift amounts from 32 to 63 give a zero result.

Other registers altered:

- Condition Register (CR0 field):
  Affected: LT, GT, EQ, SO (if Rc = ’1’)

stb stb

Store Byte (x’9800 0000’)

stb rS,d(rA)

38	S	A	d
0 5	6 10	11 15	16 31

```
if rA = 0 then b ← 0
elseb ← (rA)
EA ← b + EXTS(d)
MEM(EA, 1) ← rS[56-63]
```

The effective address is the sum (**r**A|0) + d. The contents of the low-order eight bits of **r**S are stored into the byte in memory addressed by EA.

Other registers altered:

- None

stbu stbu

Store Byte with Update (x’9C00 0000’)

stbu **r**S,d(**r**A)

39	S	A	d
0 5	6 10	11 15	16 31

```
EA ← (rA) + EXTS(d)
MEM(EA, 1) ← rS[56–63]
rA ← EA
```

The effective address is the sum (**r**A) + d. The contents of the low-order eight bits of **r**S are stored into the byte in memory addressed by EA.

EA is placed into **r**A.

If **r**A = ’0’, the instruction form is invalid.

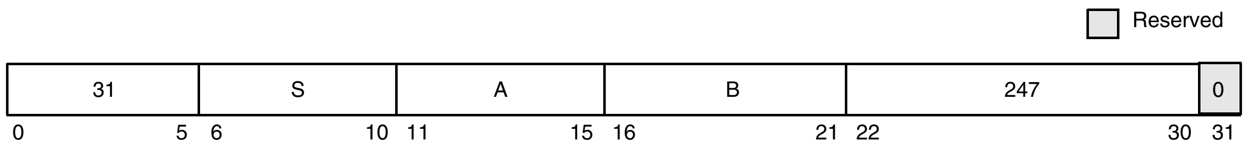
Other registers altered:

- None

stbux

Store Byte with Update Indexed (x’7C00 01EE’)

stbux **r**S,**r**A,**r**B



```
EA ← (rA) + (rB)
MEM(EA, 1) ← rS[56-63]
rA ← EA
```

EA is the sum (**r**A) + (**r**B). The contents of the low-order eight bits of **r**S are stored into the byte in memory addressed by EA.

EA is placed into **r**A.

If **r**A = ’0’, the instruction form is invalid.

Other registers altered:

- None

stbx　　　　　　　　　　　　　　　　　　　　　　　　　　　　stbx

Store Byte Indexed (x’7C00 01AE’)

stbx　　　　　　　　　　　　**r**S,**r**A,**r**B

▨ Reserved

31	S	A	B	215	▨ 0
0　　5	6　　10	11　　15	16　　21	22　　30	31

```
if rA = 0 then b ← 0
else b ← (rA)
EA ← b + (rB)
MEM(EA, 1) ← rS[56-63]
```

EA is the sum (**r**A|0) + (**r**B). The contents of the low-order eight bits of **r**S are stored into the byte in memory addressed by EA.

Other registers altered:

- None

std std

Store Doubleword (x’F800 0000’)

std **r**S,ds(**r**A)

62	S	A	ds	0 0
0 5	6 10	11 15	16 29	30 31

```
if rA = 0 then b ← 0
elseb ← (rA)
EA ← b + EXTS(ds || ’00’)
(MEM(EA, 8)) ← (rS)
```

EA is the sum (**r**A|0) + (ds || ’00’). The contents of **r**S are stored into the doubleword in memory addressed by EA.

Other registers altered:

- None

stdcx. stdcx.

Store Doubleword Conditional Indexed (x’7C00 01AD’)

stdcx. rS,rA,rB

31	S	A	B	214	1
0 5	6 10	11 15	16 20	21 30	31

```
if rA = 0 then b ← 0
else b ← (rA)
EA ← b + (rB)
if RESERVE then
  if RESERVE_ADDR = physical_addr(EA)
   MEM(EA, 8) ← (rS)
   CR0 ← ‘00’ || ‘1’ || XER[SO]
  else
   u ← undefined 1-bit value
   if u then MEM(EA, 8) ← (rS)
   CR0 ← ’00’ || u || XER[SO]
  RESERVE ← 0
else
  CR0 ← ’00’ || ’0’ || XER[SO]
```

EA is the sum (**r**A|0) + (**r**B).

If a reservation exists, and the memory address specified by the **stdcx.** instruction is the same as that specified by the load and reserve instruction that established the reservation, the contents of **r**S are stored into the doubleword in memory addressed by EA and the reservation is cleared.

If a reservation exists, but the memory address specified by the **stdcx.** instruction 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 contents of **r**S are stored into the doubleword in memory addressed by EA.

If no reservation exists, the instruction completes without altering memory.

CR0 field is set to reflect whether the store operation was performed as follows.

CR0[LT GT EQ S0] = 0b00 || store_performed || XER[SO]

EA must be a multiple of eight. If it is not, either the system alignment exception handler is invoked or the results are boundedly undefined. For additional information about alignment and DSI exceptions, see *Section 6.4.3 DSI Exception (0x00300)*.

Note: When used correctly, 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 memory.

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 memory 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) EA can be used to clear any reservation held by the processor. Examples of correct uses of these instructions, to emulate primitives such as fetch and add, test and set, and compare and swap can be found in *Appendix D Synchronization Programming Examples*.

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.

Other registers altered:

- Condition Register (CR0 field):
  Affected: LT, GT, EQ, SO

stdu

stdu

Store Doubleword with Update (x’F800 0001’)

stdu **r**S,ds**(r**A**)**

62	S	A	ds	0 1
0 5	6 10	11 15	16 29	30 31

```
EA ← (rA) + EXTS(ds || ’00’)
(MEM(EA, 8)) ← (rS)
rA ← EA
```

EA is the sum (**r**A) + (ds || ’00’). The contents of **r**S are stored into the doubleword in memory addressed by EA.

EA is placed into **r**A.

If **r**A = ’0’, the instruction form is invalid.

Other registers altered:

- None

stdux stdux

Store Doubleword with Update Indexed (x’7C00 016A’)

stdux **r**S,**r**A,**r**B

▨ Reserved

31	S	A	B	181	0 ▨
0 5	6 10	11 15	16 20	21 30	31

```
EA ← (rA) + (rB)
MEM(EA, 8) ← (rS)
rA ← EA
```

EA is the sum (**r**A) + (**r**B). The contents of **r**S are stored into the doubleword in memory addressed by EA.

EA is placed into **r**A.

If **r**A = ’0’, the instruction form is invalid.

Other registers altered:

- None

stdx stdx

Store Doubleword Indexed (x'7C00 012A')

stdx **r**S,**r**A,**r**B

▨ Reserved

31	S	A	B	149	0
0 5	6 10	11 15	16 20	21 30	31

```
if rA = 0 then b ← 0
elseb ← (rA)
EA ← b + (rB)
(MEM(EA, 8)) ← (rS)
```

EA is the sum (**r**A|0) + (**r**B). The contents of **r**S are stored into the doubleword in memory addressed by EA.

Other registers altered:

- None

stfd stfd

Store Floating-Point Double (x’D800 0000’)

stfd **fr**S,d(**r**A)

54	S	A	d
0 5	6 10	11 15	16 31

```
if rA = 0 then b ← 0
elseb ← (rA)
EA ← b + EXTS(d)
MEM(EA, 8) ← (frS)
```

EA is the sum (**r**A|0) + d.

The contents of register **fr**S are stored into the doubleword in memory addressed by EA.

Other registers altered:

- None

stfdu stfdu

Store Floating-Point Double with Update (x’DC00 0000’)

stfdu **fr**S,d(**r**A)

55	S	A	d
0 5	6 10	11 15	16 31

```
EA ← (rA) + EXTS(d)
MEM(EA, 8) ← (frS)
rA ← EA
```

EA is the sum (**r**A) + d.

The contents of register **fr**S are stored into the doubleword in memory addressed by EA.

EA is placed into **r**A.

If **r**A = ’0’, the instruction form is invalid.

Other registers altered:

- None

stfdux stfdux

Store Floating-Point Double with Update Indexed (x’7C00 05EE’)

stfdux **fr**S,**r**A,**r**B

▨ Reserved

31	S	A	B	759	0
0 5	6 10	11 15	16 20	21 30	31

```
EA ← (rA) + (rB)
MEM(EA, 8) ← (frS)
rA ← EA
```

EA is the sum (**r**A) + (**r**B).

The contents of register **fr**S are stored into the doubleword in memory addressed by EA.

EA is placed into **r**A.

If **r**A = ’0’, the instruction form is invalid.

Other registers altered:

- None

stfdx stfdx

Store Floating-Point Double Indexed (x’7C00 05AE’)

stfdx **fr**S,**r**A,**r**B

▨ Reserved

31	S	A	B	727	0
0 5	6 10	11 15	16 20	21 30	31

```
if rA = 0 then b ← 0
elseb ← (rA)
EA ← b + (rB)
MEM(EA, 8) ← (frS)
```

EA is the sum (**r**A|0) + (**r**B).

The contents of register **fr**S are stored into the doubleword in memory addressed by EA.

Other registers altered:

- None

stfiwx stfiwx

Store Floating-Point as Integer Word Indexed (x’7C00 07AE’)

stfiwx **fr**S,**r**A,**r**B

▨ Reserved

31	S	A	B	983	0
0 5	6 10	11 15	16 20	21 30	31

```
if rA = 0 then b ← 0
else b ← (rA)
EA ← b + (rB)
MEM(EA, 4) ← frS[32-63]
```

EA is the sum (**r**A|0) + (**r**B).

The contents of the low-order 32 bits of register **fr**S are stored, without conversion, into the word in memory addressed by EA.

If the contents of register **fr**S were produced, either directly or indirectly, by an **lfs** instruction, a single-precision arithmetic instruction, or **frsp**, then the value stored is undefined. The contents of **fr**S are produced directly by such an instruction if **fr**S is the target register for the instruction. The contents of **fr**S are produced indirectly by such an instruction if **fr**S is the final target register of a sequence of one or more floating-point move instructions, with the input to the sequence having been produced directly by such an instruction.

Other registers altered:

- None

stfs stfs

Store Floating-Point Single (x’D000 0000’)

stfs **fr**S,d(**r**A)

52	S	A	d
0 5	6 10	11 15	16 31

```
if rA = 0 then b ← 0
elseb ← (rA)
EA ← b + EXTS(d)
MEM(EA, 4) ← SINGLE(frS)
```

EA is the sum (**r**A|0) + d.

The contents of register **fr**S are converted to single-precision and stored into the word in memory addressed by EA. Note that the value to be stored should be in single-precision format prior to the execution of the **stfs** instruction. For a discussion on floating-point store conversions, see *Appendix C.7 Floating-Point Store Instructions*.

Other registers altered:

- None

stfsu stfsu

Store Floating-Point Single with Update (x’D400 0000’)

stfsu **fr**S,d(**r**A)

53	S	A	d
0 5	6 10	11 15	16 31

```
EA ← (rA) + EXTS(d)
MEM(EA, 4) ← SINGLE(frS)
rA ← EA
```

EA is the sum (**r**A) + d.

The contents of **fr**S are converted to single-precision and stored into the word in memory addressed by EA. Note that the value to be stored should be in single-precision format prior to the execution of the **stfsu** instruction. For a discussion on floating-point store conversions, see *Appendix C.7 Floating-Point Store Instructions*.

EA is placed into **r**A.

If **r**A = ’0’, the instruction form is invalid.

Other registers altered:

- None

stfsux stfsux

Store Floating-Point Single with Update Indexed (x’7C00 056E’)

stfsux **fr**S,**r**A,**r**B

▨ Reserved

31	S	A	B	695	▨ 0
0 5	6 10	11 15	16 20	21 30	31

```
EA ← (rA) + (rB)
MEM(EA, 4) ← SINGLE(frS)
rA ← EA
```

EA is the sum (**r**A) + (**r**B).

The contents of **fr**S are converted to single-precision and stored into the word in memory addressed by EA. For a discussion on floating-point store conversions, see *Appendix C.7 Floating-Point Store Instructions*.

EA is placed into **r**A.

If **r**A = ’0’, the instruction form is invalid.

Other registers altered:

- None

stfsx stfsx

Store Floating-Point Single Indexed (x’7C00 052E’)

stfsx **fr**S,**r**A,**r**B

▨ Reserved

31	S	A	B	663	0 ▨
0 5	6 10	11 15	16 20	21 30	31

```
if rA = 0 then b ← 0
else b ← (rA)
EA ← b + (rB)
MEM(EA, 4) ← SINGLE(frS)
```

EA is the sum (**r**A|0) + (**r**B).

The contents of register **fr**S are converted to single-precision and stored into the word in memory addressed by EA. For a discussion on floating-point store conversions, see *Appendix C.7 Floating-Point Store Instructions*.

Other registers altered:

- None

sth sth

Store Halfword (x’B000 0000’)

sth r**S**,d(**r**A)

44	S	A	d
0 5	6 10	11 15	16 31

```
if rA = 0 then b ← 0
elseb ← (rA)
EA ← b + EXTS(d)
MEM(EA, 2) ← rS[48-63]
```

EA is the sum (**r**A|0) + d. The contents of the low-order 16 bits of **r**S are stored into the halfword in memory addressed by EA.

Other registers altered:

- None

sthbrx sthbrx

Store Halfword Byte-Reverse Indexed (x’7C00 072C’)

sthbrx **r**S,**r**A,**r**B

▨ Reserved

31	S	A	B	918	0
0 5	6 10	11 15	16 20	21 30	31

```
if rA = 0 then b ← 0
else b ← (rA)
EA ← b + (rB)
MEM(EA, 2) ← rS[56-63] || rS[48-55]
```

EA is the sum (**r**A|0) + (**r**B). The contents of the low-order eight bits of **r**S are stored into bits [0–7] of the half-word in memory addressed by EA. The contents of the subsequent low-order eight bits of **r**S are stored into bits [8–15] of the halfword in memory addressed by EA.

Other registers altered:

- None

sthu sthu

Store Halfword with Update (x’B400 0000’)

sthu r**S**,d(**r**A)

45	S	A	d
0 5	6 10	11 15	16 31

```
EA ← (rA) + EXTS(d)
MEM(EA, 2) ← rS[48-63]
rA ← EA
```

EA is the sum (**r**A) + d. The contents of the low-order 16 bits of **r**S are stored into the halfword in memory addressed by EA.

EA is placed into **r**A.

If **r**A = ’0’, the instruction form is invalid.

Other registers altered:

- None

sthux sthux

Store Halfword with Update Indexed (x’7C00 036E’)

sthux **r**S,**r**A,**r**B

▨ Reserved

31	S	A	B	439	0
0 5	6 10	11 15	16 20	21 30	31

```
EA ← (rA) + (rB)
MEM(EA, 2) ← rS[48-63]
rA ← EA
```

EA is the sum (**r**A) + (**r**B). The contents of the low-order 16 bits of **r**S are stored into the halfword in memory addressed by EA.

EA is placed into **r**A.

If **r**A = ’0’, the instruction form is invalid.

Other registers altered:

- None

sthx sthx

Store Halfword Indexed (x’7C00 032E’)

sthx **r**S,**r**A,**r**B

▨ Reserved

31	S	A	B	407	0 ▨
0 5	6 10	11 15	16 20	21 30	31

```
if rA = 0 then b ← 0
elseb ← (rA)
EA ← b + (rB)
MEM(EA, 2) ← rS[48-63]
```

EA is the sum (**r**A|0) + (**r**B). The contents of the low-order 16 bits of **r**S are stored into the halfword in memory addressed by EA.

Other registers altered:

- None

stmw stmw

Store Multiple Word (x'BC00 0000')

stmw **r**S,d(**r**A)

47	S	A	d
0 5	6 10	11 15	16 31

```
if rA = 0 then b ← 0
else b ← (rA)
EA ← b + EXTS(d)
r ← rS
do while r ≤ 31
  MEM(EA, 4) ← GPR(r)[32-63]
  r ← r + 1
  EA ← EA + 4
```

EA is the sum (**r**A|0) + d.

n = (32 – **r**S).

n consecutive words starting at EA are stored from the low-order 32 bits of GPRs **r**S through **r**31. For example, if **r**S = 30, 2 words are stored.

EA must be a multiple of four. If it is not, either the system alignment exception handler is invoked or the results are boundedly undefined. For additional information about alignment and DSI exceptions, see *Section 6.4.3 DSI Exception (0x00300)*.

Note: In some implementations, this instruction is likely to have a greater latency and take longer to execute, perhaps much longer, than a sequence of individual load or store instructions that produce the same results.

Other registers altered:

- None

stswi stswi

Store String Word Immediate (x’7C00 05AA’)

stswi **r**S,**r**A,NB

▨ Reserved

31	S	A	NB	725	0 ▨
0 5	6 10	11 15	16 20	21 30	31

```
if rA = 0 then EA ← 0
elseEA ← (rA)
if NB = 0 then n ← 32
elsen ← NB
r ← rS - 1
i ← 32
do while n > 0
  if i = 32 then r ← r + 1 (mod 32)
  MEM(EA, 1) ← GPR(r)[i-i + 7]
  i ← i + 8
  if i = 64 then i ← 32
  EA ← EA + 1
  n ← n - 1
```

EA is (**r**A|0). Let *n* = NB if NB ≠ 0, *n* = 32 if NB = ’0’; *n* is the number of bytes to store. Let *nr* = CEIL(*n* ÷ 4); *nr* is the number of registers to supply data.

n consecutive bytes starting at EA are stored from GPRs **r**S through **r**S + *nr* – 1. Data is 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 through **r**0 if required.

Under certain conditions (for example, segment boundary crossing) the data alignment exception handler may be invoked. For additional information about data alignment exceptions, see *Section 6.4.3 DSI Exception (0x00300)*.

Note: In some implementations, this instruction is likely to have a greater latency and take longer to execute, perhaps much longer, than a sequence of individual load or store instructions that produce the same results.

Other registers altered:

- None

stswx stswx

Store String Word Indexed (x’7C00 052A’)

stswx **r**S,**r**A,**r**B

▨ Reserved

31	S	A	B	661	▨ 0
0 5	6 10	11 15	16 20	21 30	31

```
if rA = 0 then b ← 0
elseb ← (rA)
EA ← b + (rB)
n ← XER[25-31]
r ← rS - 1
i ← 32
do while n > 0
  if i = 32 then r ← r + 1 (mod 32)
  MEM(EA, 1) ← GPR(r)[i-i + 7]
  i ← i + 8
  if i = 64 then i ← 32
  EA ← EA + 1
  n ← n - 1
```

EA is the sum (**r**A|0) + (**r**B). Let *n* = XER[25–31]; *n* is the number of bytes to store. Let *nr* = CEIL(*n* ÷ 4); *nr* is the number of registers to supply data.

n consecutive bytes starting at EA are stored from GPRs **r**S through **r**S + *nr* – 1. Data is 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 through **r**0 if required. If *n* = ’0’, no bytes are stored.

Under certain conditions (for example, segment boundary crossing) the data alignment exception handler may be invoked. For additional information about data alignment exceptions, see *Section 6.4.3 DSI Exception (0x00300).*

Note: In some implementations, this instruction is likely to have a greater latency and take longer to execute, perhaps much longer, than a sequence of individual load or store instructions that produce the same results.

Other registers altered:

- None

stw stw

Store Word (x’9000 0000’)

stw rS,d(**r**A)

36	S	A	d
0 5	6 10	11 15	16 31

```
if rA = 0 then b ← 0
elseb ← (rA)
EA ← b + EXTS(d)
MEM(EA, 4) ← rS[32-63]
```

EA is the sum (**r**A|0) + d. The contents of the low-order 32 bits of **r**S are stored into the word in memory addressed by EA.

Other registers altered:

- None

stwbrx stwbrx

Store Word Byte-Reverse Indexed (x’7C00 052C’)

stwbrx **r**S,**r**A,**r**B

▨ Reserved

31	S	A	B	662	0
0 5	6 10	11 15	16 20	21 30	31

```
if rA = 0 then b ← 0
elseb ← (rA)
EA ← b + (rB)
MEM(EA, 4) ← rS[56-63] || rS[48-55] || rS[40-47] || rS[32-39]
```

EA is the sum (**r**A|0) + (**r**B). The contents of the low-order eight bits of **r**S are stored into bits [0–7] of the word in memory addressed by EA. The contents of the subsequent eight low-order bits of **r**S are stored into bits [8-15] of the word in memory addressed by EA. The contents of the subsequent eight low-order bits of **r**S are stored into bits [16–23] of the word in memory addressed by EA. The contents of the subsequent eight low-order bits of **r**S are stored into bits [24–31] of the word in memory addressed by EA.

Other registers altered:

- None

stwcx. stwcx.

Store Word Conditional Indexed (x'7C00 012D')

stwcx. **r**S,**r**A,**r**B

31	S	A	B	150	1
0 5	6 10	11 15	16 20	21 30	31

```
if rA = 0 then b ← 0
else b ← (rA)
EA ← b + (rB)
if RESERVE then
  if RESERVE_ADDR = physical_addr(EA)
   MEM(EA, 4) ← rS[32-63]
   CR0 ← '00' || '1' || XER[SO]
  else
   u ← undefined 1-bit value
   if u then MEM(EA, 4) ← rS[32-63]
   CR0 ← '00' || u || XER[SO]
  RESERVE ← 0
else
  CR0 ← '00' || '0' || XER[SO]
```

EA is the sum (**r**A|0) + (**r**B). If the reserved bit is set, the **stwcx.** instruction stores **r**S to effective address (**r**A + **r**B), clears the reserved bit, and sets CR0[EQ]. If the reserved bit is not set, the **stwcx.** instruction does not do a store; it leaves the reserved bit cleared and clears CR0[EQ]. Software must look at CR0[EQ] to see if the **stwcx.** was successful.

The reserved bit is set by the **lwarx** instruction. The reserved bit is cleared by any **stwcx.** instruction to any address, and also by snooping logic if it detects that another processor does any kind of store to the block indicated in the reservation buffer when reserved is set.

If a reservation exists, and the memory address specified by the **stwcx.** instruction is the same as that specified by the load and reserve instruction that established the reservation, the contents of the low-order 32 bits of **r**S are stored into the word in memory addressed by EA and the reservation is cleared.

If a reservation exists, but the memory address specified by the **stwcx.** instruction 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 contents of the low-order 32 bits of **r**S are stored into the word in memory addressed by EA.

If no reservation exists, the instruction completes without altering memory.

CR0 field is set to reflect whether the store operation was performed as follows:

CR0[LT GT EQ S0] = 0b00 || store_performed || XER[SO]

EA must be a multiple of four. If it is not, either the system alignment exception handler is invoked or the results are boundedly undefined. For additional information about alignment and DSI exceptions, see *Section 6.4.3 DSI Exception (0x00300)*.

The granularity with which reservations are managed is implementation-dependent. Therefore, the memory to be accessed by the load and reserve and store conditional instructions should be allocated by a system library program.

Other registers altered:

- Condition Register (CR0 field):
  Affected: LT, GT, EQ, SO

stwu stwu

Store Word with Update (x’9400 0000’)

stwu r S,d(**r**A)

37	S	A	**d**
0 5	6 10	11 15	16 31

```
EA ← (rA) + EXTS(d)
MEM(EA, 4) ← rS[32-63]
rA ← EA
```

EA is the sum (**r**A) + d. The contents of the low-order 32 bits of **r**S are stored into the word in memory addressed by EA.

EA is placed into **r**A.

If **r**A = ’0’, the instruction form is invalid.

Other registers altered:

- None

stwux stwux

Store Word with Update Indexed (x’7C00 016E’)

stwux **r**S,**r**A,**r**B

▨ Reserved

31	S	A	B	183	0
0 5	6 10	11 15	16 20	21 30	31

```
EA ← (rA) + (rB)
MEM(EA, 4) ← rS[32-63]
rA ← EA
```

EA is the sum (**r**A) + (**r**B). The contents of the low-order 32 bits of **r**S are stored into the word in memory addressed by EA.

EA is placed into **r**A.

If **r**A = ’0’, the instruction form is invalid.

Other registers altered:

- None

stwx stwx

Store Word Indexed (x’7C00 012E’)

stwx rS,**r**A,**r**B

▨ Reserved

31	S	A	B	151	0
0 5	6 10	11 15	16 20	21 30	31

```
if rA = 0 then b ← 0
elseb ← (rA)
EA ← b + (rB)
MEM(EA, 4) ← rS[32-63]
```

EA is the sum (**r**A|0) + (**r**B). The contents of the low-order 32 bits of **r**S are is stored into the word in memory addressed by EA.

Other registers altered:

- None

subf*x* subf*x*

Subtract From (x’7C00 0050’)

subf	**r**D,**r**A,**r**B	(OE = ’0’ Rc = ’0’)
subf.	**r**D,**r**A,**r**B	(OE = ’0’ Rc = ’1’)
subfo	**r**D,**r**A,**r**B	(OE = ’1’ Rc = ’0’)
subfo.	**r**D,**r**A,**r**B	(OE = ’1’ Rc = ’1’)

31	D	A	B	OE	40	Rc
0 5	6 10	11 15	16 20	21	22 30	31

```
rD ← ¬ (rA) + (rB) + 1
```

The sum ¬ (**r**A) + (**r**B) + 1 is placed into **r**D.

The **subf** instruction is preferred for subtraction because it sets few status bits.

Other registers altered:

- Condition Register (CR0 field):
  Affected: LT, GT, EQ, SO (if Rc = ’1’)
- XER:
  Affected: SO, OV (if OE = ’1’)

Simplified mnemonics:

sub	**r**D,**r**A,**r**B	equivalent to	**subf**	**r**D,**r**B,**r**A

subfc*x* subfc*x*

Subtract from Carrying (x’7C00 0010’)

subfc	**r**D,**r**A,**r**B	(OE = ‘0’ Rc = ‘0’)
subfc.	**r**D,**r**A,**r**B	(OE = ‘0’ Rc = ’1’)
subfco	**r**D,**r**A,**r**B	(OE = ’1’ Rc = ‘0’)
subfco.	**r**D,**r**A,**r**B	(OE = ’1’ Rc = ’1’)

31	D	A	B	OE	8	Rc
0 5	6 10	11 15	16 20	21	22 30	31

```
rD ← ¬ (rA) + (rB) + 1
```

The sum ¬ (**r**A) + (**r**B) + 1 is placed into **r**D.

Other registers altered:

- Condition Register (CR0 field):
  Affected: LT, GT, EQ, SO (if Rc = ’1’)
  Note: CR0 field may not reflect the infinitely precise result if overflow occurs.
- XER:
  Affected: CA
  Affected: SO, OV (if OE = ’1’)
  Note: The setting of the affected bits in the XER is mode-dependent, and reflects overflow of the 64-bit result in 64-bit mode and overflow of the low-order 32-bit result in 32-bit mode. For further information about 64-bit mode and 32-bit mode in 64-bit implementations, see *Chapter 3, Operand Conventions*.

Simplified mnemonics:

subc	**r**D,**r**A,**r**B	equivalent to	**subfc**	**r**D,**r**B,**r**A

subfe*x* **subfe***x*

Subtract from Extended (x'7C00 0110')

subfe	**r**D,**r**A,**r**B	(OE = '0' Rc = '0')
subfe.	**r**D,**r**A,**r**B	(OE = '0' Rc = '1')
subfeo	**r**D,**r**A,**r**B	(OE = '1' Rc = '0')
subfeo.	**r**D,**r**A,**r**B	(OE = '1' Rc = '1')

31	D	A	B	OE	136	Rc
0 5	6 10	11 15	16 20	21	22 30	31

```
rD ← ¬ (rA) + (rB) + XER[CA]
```

The sum ¬ (**r**A) + (**r**B) + XER[CA] is placed into **r**D.

Other registers altered:

- Condition Register (CR0 field):
  Affected: LT, GT, EQ, SO (if Rc = '1')
  Note: CR0 field may not reflect the infinitely precise result if overflow occurs.
- XER:
  Affected: CA
  Affected: SO, OV (if OE = '1')
  Note: The setting of the affected bits in the XER is mode-dependent, and reflects overflow of the 64-bit result in 64-bit mode and overflow of the low-order 32-bit result in 32-bit mode. For further information about 64-bit mode and 32-bit mode in 64-bit implementations, see *Chapter 3, Operand Conventions*.

subfic subfic

Subtract from Immediate Carrying (x’2000 0000’)

subfic **r**D,**r**A,SIMM

08	D	A	SIMM
0 5	6 10	11 15	16 31

```
rD ← ¬ (rA) + EXTS(SIMM) + 1
```

The sum ¬ (**r**A) + EXTS(SIMM) + 1 is placed into **r**D.

Other registers altered:

- XER:
  Affected: CA
  Note: The setting of the affected bits in the XER is mode-dependent, and reflects overflow of the 64-bit result in 64-bit mode and overflow of the low-order 32-bit result in 32-bit mode. For further information about 64-bit mode and 32-bit mode in 64-bit implementations, see *Chapter 3, Operand Conventions*.

subfme*x* subfme*x*

Subtract from Minus One Extended (x’7C00 01D0’)

subfme	**r**D,**r**A	(OE = ’0’ Rc = ’0’)
subfme.	**r**D,**r**A	(OE = ’0’ Rc = ’1’)
subfmeo	**r**D,**r**A	(OE = ’1’ Rc = ’0’)
subfmeo.	**r**D,**r**A	(OE = ’1’ Rc = ’1’)

▨ Reserved

31	D	A	0 0 0 0 0	OE	232	Rc
0 5	6 10	11 15	16 20	21	22 30	31

```
rD ← ¬ (rA) + XER[CA] - 1
```

The sum ¬ (**r**A) + XER[CA] + (64)1 is placed into **r**D.

Other registers altered:

- Condition Register (CR0 field):
  Affected: LT, GT, EQ, SO (if Rc = ’1’)
  Note: CR0 field may not reflect the infinitely precise result if overflow occurs.
- XER:
  Affected: CA
  Affected: SO, OV (if OE = ’1’)
  Note: The setting of the affected bits in the XER is mode-dependent, and reflects overflow of the 64-bit result in 64-bit mode and overflow of the low-order 32-bit result in 32-bit mode. For further information about 64-bit mode and 32-bit mode in 64-bit implementations, see *Chapter 3, Operand Conventions*.

subfze*x*

Subtract from Zero Extended (x'7C00 0190')

subfze	**r**D,**r**A	(OE = '0' Rc = '0')
subfze.	**r**D,**r**A	(OE = '0' Rc = '1')
subfzeo	**r**D,**r**A	(OE = '1' Rc = '0')
subfzeo.	**r**D,**r**A	(OE = '1' Rc = '1')

▨ Reserved

0–5	6–10	11–15	16–20	21	22–30	31
31	D	A	0 0 0 0 0 ▨	OE	200	Rc

```
rD ← ¬ (rA) + XER[CA]
```

The sum ¬ (**r**A) + XER[CA] is placed into **r**D.

Other registers altered:

- Condition Register (CR0 field):
  Affected: LT, GT, EQ, SO (if Rc = '1')
  Note: CR0 field may not reflect the infinitely precise result if overflow occurs.
- XER:
  Affected: CA
  Affected: SO, OV (if OE = '1')
  Note: The setting of the affected bits in the XER is mode-dependent, and reflects overflow of the 64-bit result in 64-bit mode and overflow of the low-order 32-bit result in 32-bit mode. For further information about 64-bit mode and 32-bit mode in 64-bit implementations, see *Chapter 3, Operand Conventions*.

sync

Synchronize (x’7C00 04AC’)

sync L

▨ Reserved

31	0 0 0	L	0 0 0 0 0	0 0 0 0 0	598	0
0 5	6 8	9 10	11 15	16 20	21 30	31

The sync instruction creates a memory barrier. The set of memory accesses that is ordered by the memory barrier depends on the value of the L field.

L = ‘0’ (“heavyweight **sync**”)	The memory barrier provides an ordering function for the memory accesses associated with all instructions that are executed by the processor executing the **sync** instruction. The applicable pairs are all pairs a_i,b_j in which b_j is a data access, except that if a_i is the memory access caused by an **icbi** instruction then b_j may be performed with respect to the processor executing the **sync** instruction before a_i is performed with respect to that processor.
L= ‘1’ (“lightweight **sync**”)	The memory barrier provides an ordering function for the memory accesses caused by Load, Store, and **dcbz** instructions that are executed by the processor executing the **sync** instruction and for which the specified memory location is in memory that is Memory Coherence Required and is neither Write Through Required nor Caching Inhibited. The applicable pairs are all pairs a_i,b_j of such accesses except those in which a_i is an access caused by a store or **dcbz** instruction and b_j is an access caused by a load instruction.
L= ‘2’ (**ptesync**)	This variant of the synchronize instruction is designated the page table entry sync and is specified by the extended mnemonic **ptesync**. This variant has all of the properties of **sync** with L = ‘0’ and with some additional properties.
L= ‘3’	Reserved. The results of executing a **sync** instruction with L= ‘3’ are boundedly undefined.

The ordering done by the memory barrier is cumulative. The sync instruction may complete before memory accesses associated with instructions preceding the sync instruction have been performed.

If L= ’0’, the sync instruction has the following additional properties:

- Executing the **sync** instruction ensures that all instructions preceding the **sync** instruction have completed before the **sync** instruction completes, and that no subsequent instructions are initiated until after the **sync** instruction completes.
- The **sync** instruction is execution synchronizing. However, address translation and reference and change recording associated with subsequent instructions may be performed before the **sync** instruction completes.
- The memory barrier provides the additional ordering function such that if a given instruction that is the result of a Store in set B is executed, all applicable memory accesses in set A have been performed with respect to the processor executing the instruction to the extent required by the associated memory coher-

ence properties. The single exception is that any memory access in set A that is caused by an **icbi** instruction executed by the processor executing the **sync** instruction (P1) may not have been performed with respect to P1.

The cumulative properties of the barrier apply to the execution of the given instruction as they would to a Load that returned a value that was the result of a Store in set B.

If L=’2’, the **sync** instruction (**ptesync**) has the following additional properties:

- The memory barrier created by the **ptesync** instruction provides an ordering function for the memory accesses associated with all instructions that are executed by the processor executing the **ptesync** instruction and, as elements of set A, for all reference and change bit updates associated with additional address translations that were performed, by the processor executing the **ptesync** instruction, before the **ptesync** instruction is executed. The applicable pairs are all pairs a_i,b_j in which b_j is a data access and a_i is not an instruction fetch.
- The **ptesync** instruction causes all reference and change bit updates associated with address translations that were performed, by the processor executing the **ptesync** instruction, before the **ptesync** instruction is executed, to be performed with respect to that processor before the **ptesync** instruction's memory barrier is created.
- The **ptesync** instruction provides an ordering function for all stores to the page table caused by store instructions preceding the **ptesync** instruction with respect to searches of the page table that are performed, by the processor executing the **ptesync** instruction, after the **ptesync** instruction completes. Executing a **ptesync** instruction ensures that all such stores will be performed, with respect to the processor executing the **ptesync** instruction, before any implicit accesses to the affected page table entries, by such page table searches, are performed with respect to that processor.
- In conjunction with the **tlbie** and **tlbsync** instructions, the **ptesync** instruction provides an ordering function for TLB invalidations and related memory accesses on other processors as described in the **tlbsync** instruction description.

Note: The functions performed by the **ptesync** instruction may take a significant amount of time to complete, so this form of the instruction should be used only if the functions listed above are needed. Otherwise **sync** with L = ‘0’ should be used (or **sync** with L = ‘1’ or **eieio**, if appropriate).

This instruction is execution synchronizing. For more information on execution synchronization, see *Section 4.1.5 Synchronizing Instructions*.

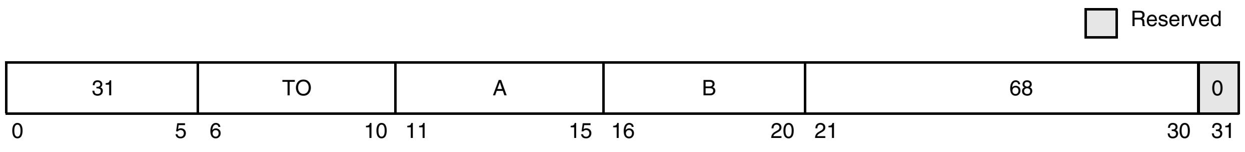
Other registers altered:

- None

td td

Trap Doubleword (x’7C00 0088’)

td TO,**r**A,**r**B



```
a ← (rA)
b ← (rB)
if (a < b) & TO[0] then TRAP
if (a > b) & TO[1] then TRAP
if (a = b) & TO[2] then TRAP
if (a <U b) & TO[3] then TRAP
if (a >U b) & TO[4] then TRAP
```

The contents of **r**A are compared with the contents of **r**B. If any bit in the TO field is set and its corresponding condition is met by the result of the comparison, then the system trap handler is invoked.

Other registers altered:

- None

Simplified mnemonics:

tdge	**r**A,**r**B	equivalent to	**td**	**12,r**A,**r**B
tdlnl	**r**A,**r**B	equivalent to	**td**	**5,r**A,**r**B

tdi tdi

Trap Doubleword Immediate (x’0800 0000’)

tdi TO,**r**A,SIMM

02	TO	A	SIMM
0 5	6 10	11 15	16 31

```
a ← (rA)
if (a < EXTS(SIMM)) & TO[0] then TRAP
if (a > EXTS(SIMM)) & TO[1] then TRAP
if (a = EXTS(SIMM)) & TO[2] then TRAP
if (a <U EXTS(SIMM)) & TO[3] then TRAP
if (a >U EXTS(SIMM)) & TO[4] then TRAP
```

The contents of **r**A are compared with the sign-extended value of the SIMM field. If any bit in the TO field is set and its corresponding condition is met by the result of the comparison, then the system trap handler is invoked.

Other registers altered:

- None

Simplified mnemonics:

tdlti	**r**A,value	equivalent to	**tdi**	16,**r**A,value
tdnei	**r**A,value	equivalent to	**tdi**	24,**r**A,value

tlbia tlbia

Translation Lookaside Buffer Invalidate All (x’7C00 02E4’)

▨ Reserved

31	00 000	0 0000	0000 0	370	0
0 5	6 10	11 15	16 20	21 30	31

```
All TLB entries ← invalid
```

The entire translation lookaside buffer (TLB) is invalidated (that is, all entries are removed).

The TLB is invalidated regardless of the settings of MSR[IR] and MSR[DR]. The invalidation is done without reference to the SLB, segment table, or segment registers.

This instruction does not cause the entries to be invalidated in other processors.

This is a supervisor-level instruction and optional in the PowerPC Architecture.

Other registers altered:

- None

tlbie

tlbie

Translation Lookaside Buffer Invalidate Entry (x’7C00 0264’)

tlbie **r**B, L

▨ Reserved

31	0 0 0 0	L	0 0 0 0 0	B	306	0
0 5	6	10	11 15	16 20	21 30	31

```
if L = 0
  then pg_size ← 4 KB
  else pg_size ← large page size
p ← log_base_2(pg_size)
for each processor in the partition
  for each TLB entry
  if (entry_VPN[32 to 79-p] = (RB[16 to63-p]) & (entry_pg_size = pg_size)
  then TLB entry ← invalid
```

The contents of **r**B[0-15] must be 0x0000. If the L field of the instruction is ‘1’ let the page size be large; otherwise let the page size be 4 KB.

All TLB entries that have all of the following properties are made invalid on all processors that are in the same partition as the processor executing the tlbie instruction.

- The entry translates a virtual address for which VPN[32 to 79- p] is equal to **r**B[16 to 63- p].
- The page size of the entry matches the page size specified by the L field of the instruction.

Additional TLB entries may also be made invalid on any processor that is in the same partition as the processor executing the tlbie instruction.

MSR[SF] must be ‘1’ when this instruction is executed; otherwise the results are undefined.

The operation performed by this instruction is ordered by the **eieio** (or **sync** or **ptesync**) instruction with respect to a subsequent **tlbsync** instruction executed by the processor executing the tlbie instruction. The operations caused by **tlbie** and **tlbsync** are ordered by **eieio** as a third set of operations, which is independent of the other two sets that **eieio** orders.

This is a supervisor-level instruction and optional in the PowerPC Architecture.

Other registers altered:

- None

tlbiel tlbiel

Translation Lookaside Buffer Invalidate Entry Local (x’7C00 0224’)

tlbiel **r**B,L

▨ Reserved

31	00 000	L	0 0000	rB	274	0
0 5	6 9	10	11 15	16 20	21 30	31

```
if L = 0
  then pg_size ← 4 KB
  else pg_size ← large page size
p ← log_base_2(pg_size)
  for each TLB entry
         if (entry_VPN[32 to (79-p)] = rB[16 to (63-p)] &
            (entry_pg_size = pg_size)
         then TLB entry ← invalid
```

The contents of **r**B[0-15] must be 0x0000. If the L field of the instruction is ‘1’ let the page size be large; otherwise let the page size be 4KB.

All TLB entries that have all of the following properties are made invalid on the processor which executes this instruction.

- The entry translates a virtual address for which VPN[32 to (79- p)] is equal to **r**B[16 to (63- p)].
- The page size of the entry matches the page size specified by the L field of the instruction.

Only TLB entries on the processor executing this instruction are affected. **r**B[52 - 63] must be zero. MSR[SF] must be ‘1’ when this instruction is executed; otherwise the results are undefined.

The operation performed by this instruction is ordered by the **eieio** (or **sync** or **ptesync**) instruction with respect to a subsequent **tlbsync** instruction executed by the processor executing the **tlbiel** instruction. The operations caused by **tlbiel** and **tlbsync** are ordered by **eieio** as a third set of operations, which is independent of the other two sets that **eieio** orders.

This is a supervisor-level instruction and optional in the PowerPC Architecture.

Support of large pages for **tlbiel** is optional. On implementations that do not support large pages for **tlbiel**, the following properties apply:

- The syntax of the instruction is “**tlbiel r**B”.
- Bit [10] of the instruction is a reserved bit.
- In the RTL, the first three lines and the third from last line are ignored.
- The last list item in the paragraph that begins “All TLB entries ...”, namely “The page size of the entry matches the page size specified by the L field of the instruction”, is ignored.

Note: To synchronize the completion of this processor local form of **tlbie**, only a **ptesync** is required (**tlbsync** should not be used).

Other registers altered:
- None

tlbsync tlbsync

TLB Synchronize (x’7C00 046C’)

▨ Reserved

31	00 000	0 0000	0000 0	566	0
0 5	6 10	11 15	16 20	21 30	31

If an implementation sends a broadcast for **tlbie** then it will also send a broadcast for **tlbsync**. Executing a **tlbsync** instruction ensures that all **tlbie** instructions previously executed by the processor executing the **tlbsync** instruction have completed on all other processors.

The operation performed by this instruction is treated as a caching-inhibited and guarded data access with respect to the ordering done by **eieio**.

Refer to *Section 7.4.3.4 Synchronization of Memory Accesses and Referenced and Changed Bit Updates* and *Section 7.5.3 Page Table Updates* for other requirements associated with the use of this instruction.

This instruction is supervisor-level and optional in the PowerPC Architecture.

Note: tlbsync should not be used to synchronize the completion of **tlbiel**.

Other registers altered:

- None

tw   tw

Trap Word (x’7C00 0008’)

tw TO,**r**A,**r**B

▨ Reserved

31	TO	A	B	4	▨ 0
0 5	6 10	11 15	16 20	21 30	31

```
a ← EXTS(rA[32-63])
b ← EXTS(rB[32-63])
if (a < b) & TO[0] then TRAP
if (a > b) & TO[1] then TRAP
if (a = b) & TO[2] then TRAP
if (a <U b) & TO[3] then TRAP
if (a >U b) & TO[4] then TRAP
```

The contents of the low-order 32 bits of **r**A are compared with the contents of the low-order 32 bits of **r**B. If any bit in the TO field is set and its corresponding condition is met by the result of the comparison, then the system trap handler is invoked.

Other registers altered:

- None

Simplified mnemonics:

tweq	**r**A,**r**B	equivalent to	**tw**	4,**r**A,**r**B
twlge	**r**A,**r**B	equivalent to	**tw**	5,**r**A,**r**B
trap		equivalent to	**tw**	31,0,0

twi twi

Trap Word Immediate (x’0C00 0000’)

twi TO,**r**A,SIMM

03	TO	A	SIMM
0 5	6 10	11 15	16 31

```
a ← EXTS(rA[32-63])
if (a < EXTS(SIMM)) & TO[0] then TRAP
if (a > EXTS(SIMM)) & TO[1] then TRAP
if (a = EXTS(SIMM)) & TO[2] then TRAP
if (a <U EXTS(SIMM)) & TO[3] then TRAP
if (a >U EXTS(SIMM)) & TO[4] then TRAP
```

The contents of the low-order 32 bits of **r**A are compared with the sign-extended value of the SIMM field. If any bit in the TO field is set and its corresponding condition is met by the result of the comparison, then the system trap handler is invoked.

Other registers altered:

- None

Simplified mnemonics:

twgti	**r**A,value	equivalent to	**twi**	8,**r**A,value
twllei	**r**A,value	equivalent to	**twi**	6,**r**A,value

xor*x* **xor***x*

XOR (x’7C00 0278’)

xor	**r**A,**r**S,**r**B	(Rc = ’0’)
xor.	**r**A,**r**S,**r**B	(Rc = ’1’)

31	S	A	B	316	Rc
0 5	6 10	11 15	16 20	21 30	31

$$\mathbf{r}\text{A} \leftarrow (\mathbf{r}\text{S}) \oplus (\mathbf{r}\text{B})$$

The contents of **r**S is XORed with the contents of **r**B and the result is placed into **r**A.

Other registers altered:

- Condition Register (CR0 field):
  Affected: LT, GT, EQ, SO (if Rc = ’1’)

xori

XOR Immediate (x’6800 0000’)

xori **r**A,**r**S,UIMM

26	S	A	UIMM
0 – 5	6 – 10	11 – 15	16 – 31

$$\mathbf{r}\mathrm{A} \leftarrow (\mathbf{r}\mathrm{S}) \oplus ((48)0 \,\|\, \mathrm{UIMM})$$

The contents of **r**S are XORed with 0x0000_0000_0000 || UIMM and the result is placed into **r**A.

Other registers altered:

- None

xoris

XOR Immediate Shifted (x’6C00 0000’)

xoris **r**A,**r**S,UIMM

27	S	A	UIMM
0 5	6 10	11 15	16 31

```
rA ← (rS) ⊕ ((32)0 || UIMM || (16)0)
```

The contents of **r**S are XORed with 0x0000_0000 || UIMM || 0x0000 and the result is placed into **r**A.

Other registers altered:

- None

Appendix A. PowerPC Instruction Set Listings

This appendix lists the PowerPC Architecture's instruction set. Instructions are sorted by mnemonic, opcode, function, and form. Also included in this appendix is a quick reference table that contains general information, such as the architecture level, privilege level, and form, and indicates if the instruction is 64-bit and/or optional.

Note: Split fields, which represent the concatenation of sequences from left to right, are shown in lowercase. For more information refer to *Chapter 8, Instruction Set*.

A.1 Instructions Sorted by Mnemonic

Table A-1 lists the instructions implemented in the PowerPC Architecture in alphabetical order by mnemonic.

Key:

▨ Reserved bits

Table A-1. Complete Instruction List Sorted by Mnemonic

Name	0 5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
							▨					▨				▨											▨
addc*x*	31	D					A					B					OE	10									Rc
adde*x*	31	D					A					B					OE	138									Rc
addi	14	D					A					SIMM															
addic	12	D					A					SIMM															
addic.	13	D					A					SIMM															
addis	15	D					A					SIMM															
addme*x*	31	D					A					0 0 0 0 0					OE	234									Rc
add*x*	31	D					A					B					OE	266									Rc
addze*x*	31	D					A					0 0 0 0 0					OE	202									Rc
andc*x*	31	S					A					B					60										Rc
andi.	28	S					A					UIMM															
andis.	29	S					A					UIMM															
and*x*	31	S					A					B					28										Rc
bcctr*x*	19	BO					BI					0 0 0			BH		528										LK
bclr*x*	19	BO					BI					0 0 0			BH		16										LK
bc*x*	16	BO					BI					BD														AA	LK
b*x*	18	LI																								AA	LK

Note:
1. 64-bit instruction
2. Optional instruction
3. Supervisor level instruction
4. Load/store string/multiple instruction
5. Supervisor and user-level instruction
6. Optional 64-bit bridge instruction

Table A-1. Complete Instruction List Sorted by Mnemonic

Name	0 5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
cmp	31	crfD			0	L	A					B					0										0
cmpi	11	crfD			0	L	A					SIMM															
cmpl	31	crfD			0	L	A					B					32										0
cmpli	10	crfD			0	L	A					UIMM															
cntlzd*x* [1]	31	S					A					0 0 0 0 0					58										Rc
cntlzw*x*	31	S					A					0 0 0 0 0					26										Rc
crand	19	crbD					crbA					crbB					257										0
crandc	19	crbD					crbA					crbB					129										0
creqv	19	crbD					crbA					crbB					289										0
crnand	19	crbD					crbA					crbB					225										0
crnor	19	crbD					crbA					crbB					33										0
cror	19	crbD					crbA					crbB					449										0
crorc	19	crbD					crbA					crbB					417										0
crxor	19	crbD					crbA					crbB					193										0
dcbf	31	0 0 0 0 0					A					B					86										0
dcbst	31	0 0 0 0 0					A					B					54										0
dcbt	31	0 0 0 0 0					A					B					278										0
dcbtst	31	0 0 0 0 0					A					B					246										0
dcbz	31	0 0 0 0 0					A					B					1014										0
divdu*x* [1]	31	D					A					B					OE	457									Rc
divd*x* [1]	31	D					A					B					OE	489									Rc
divwu*x*	31	D					A					B					OE	459									Rc
divw*x*	31	D					A					B					OE	491									Rc
eciwx	31	D					A					B					310										0
ecowx	31	S					A					B					438										0
eieio	31	0 0 0 0 0					0 0 0 0 0					0 0 0 0 0					854										0
eqv*x*	31	S					A					B					284										Rc
extsb*x*	31	S					A					0 0 0 0 0					954										Rc
extsh*x*	31	S					A					0 0 0 0 0					922										Rc
extsw*x* [1]	31	S					A					0 0 0 0 0					986										Rc
fabs*x*	63	D					0 0 0 0 0					B					264										Rc
fadds*x*	59	D					A					B					0 0 0 0 0					21					Rc

Note:

1. 64-bit instruction
2. Optional instruction
3. Supervisor level instruction
4. Load/store string/multiple instruction
5. Supervisor and user-level instruction
6. Optional 64-bit bridge instruction

Table A-1. Complete Instruction List Sorted by Mnemonic

Name	0 5	6 7 8	9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25	26 27 28 29 30	31
fadd*x*	63	D		A	B	0 0 0 0 0	21	Rc
fcfid*x* [1]	63	D		0 0 0 0 0	B	846		Rc
fcmpo	63	crfD	0 0	A	B	32		0
fcmpu	63	crfD	0 0	A	B	0		0
fctid*x* [1]	63	D		0 0 0 0 0	B	814		Rc
fctidz*x* [1]	63	D		0 0 0 0 0	B	815		Rc
fctiw*x*	63	D		0 0 0 0 0	B	14		Rc
fctiwz*x*	63	D		0 0 0 0 0	B	15		Rc
fdivs*x*	59	D		A	B	0 0 0 0 0	18	Rc
fdiv*x*	63	D		A	B	0 0 0 0 0	18	Rc
fmadds*x*	59	D		A	B	C	29	Rc
fmadd*x*	63	D		A	B	C	29	Rc
fmr*x*	63	D		0 0 0 0 0	B	72		Rc
fmsubs*x*	59	D		A	B	C	28	Rc
fmsub*x*	63	D		A	B	C	28	Rc
fmuls*x*	59	D		A	0 0 0 0 0	C	25	Rc
fmul*x*	63	D		A	0 0 0 0 0	C	25	Rc
fnabs*x*	63	D		0 0 0 0 0	B	136		Rc
fneg*x*	63	D		0 0 0 0 0	B	40		Rc
fnmadds*x*	59	D		A	B	C	31	Rc
fnmadd*x*	63	D		A	B	C	31	Rc
fnmsubs*x*	59	D		A	B	C	30	Rc
fnmsub*x*	63	D		A	B	C	30	Rc
fres*x* [2]	59	D		0 0 0 0 0	B	0 0 0 0 0	24	Rc
frsp*x*	63	D		0 0 0 0 0	B	12		Rc
frsqrte*x* [2]	63	D		0 0 0 0 0	B	0 0 0 0 0	26	Rc
fsel*x* [2]	63	D		A	B	C	23	Rc
fsqrts*x* [2]	59	D		0 0 0 0 0	B	0 0 0 0 0	22	Rc
fsqrt*x* [2]	63	D		0 0 0 0 0	B	0 0 0 0 0	22	Rc
fsubs*x*	59	D		A	B	0 0 0 0 0	20	Rc
fsub*x*	63	D		A	B	0 0 0 0 0	20	Rc
icbi	31	0 0 0 0 0		A	B	982		0

Note:

1. 64-bit instruction
2. Optional instruction
3. Supervisor level instruction
4. Load/store string/multiple instruction
5. Supervisor and user-level instruction
6. Optional 64-bit bridge instruction

Table A-1. Complete Instruction List Sorted by Mnemonic

Name	0 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29	30	31
isync	19	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	150		0
lbz	34	D	A	d			
lbzu	35	D	A	d			
lbzux	31	D	A	B	119		0
lbzx	31	D	A	B	87		0
ld [1]	58	D	A	ds		0	
ldarx [1]	31	D	A	B	84		0
ldu [1]	58	D	A	ds		1	
ldux [1]	31	D	A	B	53		0
ldx [1]	31	D	A	B	21		0
lfd	50	D	A	d			
lfdu	51	D	A	d			
lfdux	31	D	A	B	631		0
lfdx	31	D	A	B	599		0
lfs	48	D	A	d			
lfsu	49	D	A	d			
lfsux	31	D	A	B	567		0
lfsx	31	D	A	B	535		0
lha	42	D	A	d			
lhau	43	D	A	d			
lhaux	31	D	A	B	375		0
lhax	31	D	A	B	343		0
lhbrx	31	D	A	B	790		0
lhz	40	D	A	d			
lhzu	41	D	A	d			
lhzux	31	D	A	B	311		0
lhzx	31	D	A	B	279		0
lmw [4]	46	D	A	d			
lswi [4]	31	D	A	NB	597		0
lswx [4]	31	D	A	B	533		0
lwa [1]	58	D	A	ds		2	
lwarx	31	D	A	B	20		0

Note:

1. 64-bit instruction
2. Optional instruction
3. Supervisor level instruction
4. Load/store string/multiple instruction
5. Supervisor and user-level instruction
6. Optional 64-bit bridge instruction

Table A-1. Complete Instruction List Sorted by Mnemonic

Name	0 5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
lwaux [1]	31	D					A					B					373										0
lwax [1]	31	D					A					B					341										0
lwbrx	31	D					A					B					534										0
lwz	32	D					A					d															
lwzu	33	D					A					d															
lwzux	31	D					A					B					55										0
lwzx	31	D					A					B					23										0
mcrf	19	crfD			0 0		crfS			0 0		0 0 0 0 0					0										0
mcrfs	63	crfD			0 0		crfS			0 0		0 0 0 0 0					64										0
mcrxr	31	crfD			0 0		0 0 0 0 0					0 0 0 0 0					512										0
mfcr	31	D					0 0 0 0 0					0 0 0 0 0					19										0
mffs*x*	63	D					0 0 0 0 0					0 0 0 0 0					583										Rc
mfmsr [3]	31	D					0 0 0 0 0					0 0 0 0 0					83										0
mfocrf	31	D					1	CRM								0	19										0
mfspr [5]	31	D					spr										339										0
mfsr [3, 6]	31	D					0	SR				0 0 0 0 0					595										0
mfsrin [3, 6]	31	D					0 0 0 0 0					B					659										0
mftb	31	D					tbr										371										0
mtcrf	31	S					0	CRM								0	144										0
mtfsb0*x*	63	crbD					0 0 0 0 0					0 0 0 0 0					70										Rc
mtfsb1*x*	63	crbD					0 0 0 0 0					0 0 0 0 0					38										Rc
mtfsfi*x*	63	crfD			0 0		0 0 0 0 0					IMM				0	134										Rc
mtfsf*x*	63	0	FM								0	B					711										Rc
mtmsr [3, 6]	31	S					0 0 0 0				L	0 0 0 0 0					146										0
mtmsrd [1, 3]	31	S					0 0 0 0				L	0 0 0 0 0					178										0
mtocrf	31	S					1	CRM								0	144										0
mtspr [5]	31	S					spr										467										0
mtsr [3, 6]	31	S					0	SR				0 0 0 0 0					210										0
mtsrin [3, 6]	31	S					0 0 0 0 0					B					242										0
mulhdu*x* [1]	31	D					A					B					0	9									Rc
mulhd*x* [1]	31	D					A					B					0	73									Rc
mulhwu*x*	31	D					A					B					0	11									Rc

Note:

1. 64-bit instruction
2. Optional instruction
3. Supervisor level instruction
4. Load/store string/multiple instruction
5. Supervisor and user-level instruction
6. Optional 64-bit bridge instruction

Table A-1. Complete Instruction List Sorted by Mnemonic

Name	0 5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
mulhw*x*	31	D					A					B					0	75									Rc
mulld*x* [1]	31	D					A					B					OE	233									Rc
mulli	7	D					A					SIMM															
mullw*x*	31	D					A					B					OE	235									Rc
nand*x*	31	S					A					B					476										Rc
neg*x*	31	D					A					0 0 0 0 0					OE	104									Rc
nor*x*	31	S					A					B					124										Rc
orc*x*	31	S					A					B					412										Rc
ori	24	S					A					UIMM															
oris	25	S					A					UIMM															
or*x*	31	S					A					B					444										Rc
rfid [1, 3]	19	0 0 0 0 0					0 0 0 0 0					0 0 0 0 0					18										0
rldcl*x* [1]	30	S					A					B					mb						8				Rc
rldcr*x* [1]	30	S					A					B					me						9				Rc
rldicl*x* [1]	30	S					A					sh					mb						0			sh	Rc
rldicr*x* [1]	30	S					A					sh					me						1			sh	Rc
rldic*x* [1]	30	S					A					sh					mb						2			sh	Rc
rldimi*x* [1]	30	S					A					sh					mb						3			sh	Rc
rlwimi*x*	20	S					A					SH					MB					ME					Rc
rlwinmx	21	S					A					SH					MB					ME					Rc
rlwnm*x*	23	S					A					B					MB					ME					Rc
sc	17	0 0 0 0 0					0 0 0 0 0					0 0 0 0 0 0 0 0 0 0 0 0 0 0														1	0
slbia [1,2,3]	31	0 0 0 0 0					0 0 0 0 0					0 0 0 0 0					498										0
slbie [1,2,3]	31	0 0 0 0 0					0 0 0 0 0					B					434										0
slbmfee [1]	31	D					0 0 0 0 0					B					915										0
slbmfev [1]	31	D					0 0 0 0 0					B					851										0
slbmte [1]	31	D					0 0 0 0 0					B					402										0
sld*x* [1]	31	S					A					B					27										Rc
slw*x*	31	S					A					B					24										Rc
sradi*x* [1]	31	S					A					sh					413									sh	Rc
srad*x* [1]	31	S					A					B					794										Rc
srawi*x*	31	S					A					SH					824										Rc

Note:

1. 64-bit instruction
2. Optional instruction
3. Supervisor level instruction
4. Load/store string/multiple instruction
5. Supervisor and user-level instruction
6. Optional 64-bit bridge instruction

Table A-1. Complete Instruction List Sorted by Mnemonic

Name	0 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29	30	31
sraw*x*	31	S	A	B	792		Rc
srdx [1]	31	S	A	B	539		Rc
srw*x*	31	S	A	B	536		Rc
stb	38	S	A	d			
stbu	39	S	A	d			
stbux	31	S	A	B	247		0
stbx	31	S	A	B	215		0
std [1]	62	S	A	ds		0	
stdcx. [1]	31	S	A	B	214		1
stdu [1]	62	S	A	ds		1	
stdux [1]	31	S	A	B	181		0
stdx [1]	31	S	A	B	149		0
stfd	54	S	A	d			
stfdu	55	S	A	d			
stfdux	31	S	A	B	759		0
stfdx	31	S	A	B	727		0
stfiwx [2]	31	S	A	B	983		0
stfs	52	S	A	d			
stfsu	53	S	A	d			
stfsux	31	S	A	B	695		0
stfsx	31	S	A	B	663		0
sth	44	S	A	d			
sthbrx	31	S	A	B	918		0
sthu	45	S	A	d			
sthux	31	S	A	B	439		0
sthx	31	S	A	B	407		0
stmw [4]	47	S	A	d			
stswi [4]	31	S	A	NB	725		0
stswx [4]	31	S	A	B	661		0
stw	36	S	A	d			
stwbrx	31	S	A	B	662		0
stwcx.	31	S	A	B	150		1

Note:

1. 64-bit instruction
2. Optional instruction
3. Supervisor level instruction
4. Load/store string/multiple instruction
5. Supervisor and user-level instruction
6. Optional 64-bit bridge instruction

Table A-1. Complete Instruction List Sorted by Mnemonic

Name	0 5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
stwu	37	S					A					d															
stwux	31	S					A					B					183										0
stw*x*	31	S					A					B					151										0
subfc*x*	31	D					A					B					OE	8									Rc
subfe*x*	31	D					A					B					OE	136									Rc
subfic	08	D					A					SIMM															
subfme*x*	31	D					A					0 0 0 0 0					OE	232									Rc
subf*x*	31	D					A					B					OE	40									Rc
subfze*x*	31	D					A					0 0 0 0 0					OE	200									Rc
sync	31	0 0 0			L		0 0 0 0 0					0 0 0 0 0					598										0
td [1]	31	TO					A					B					68										0
tdi [1]	02	TO					A					SIMM															
tlbia [2,3]	31	0 0 0 0 0					0 0 0 0 0					0 0 0 0 0					370										0
tlbie [2,3]	31	0 0 0 0 0				L	0 0 0 0 0					B					306										0
tlbiel [2,3]	31	0 0 0 0 0				L	0 0 0 0 0					B					274										0
tlbsync [2,3]	31	0 0 0 0 0					0 0 0 0 0					0 0 0 0 0					566										0
tw	31	TO					A					B					4										0
twi	03	TO					A					SIMM															
xori	26	S					A					UIMM															
xoris	27	S					A					UIMM															
xor*x*	31	S					A					B					316										Rc

Note:

1. 64-bit instruction
2. Optional instruction
3. Supervisor level instruction
4. Load/store string/multiple instruction
5. Supervisor and user-level instruction
6. Optional 64-bit bridge instruction

A.2 Instructions Sorted by Opcode

Table A-2 lists the instructions defined in the PowerPC Architecture in numeric order by opcode.

Key:

▨	Reserved bits

Table A-2. Complete Instruction List Sorted by Opcode

Name	0 5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
tdi [1]	0 0 0 0 1 0	TO					A					SIMM															
twi	0 0 0 0 1 1	TO					A					SIMM															
mulli	0 0 0 1 1 1	D					A					SIMM															
subfic	0 0 1 0 0 0	D					A					SIMM															
cmpli	0 0 1 0 1 0	crfD			0	L	A					UIMM															
cmpi	0 0 1 0 1 1	crfD			0	L	A					SIMM															
addic	0 0 1 1 0 0	D					A					SIMM															
addic.	0 0 1 1 0 1	D					A					SIMM															
addi	0 0 1 1 1 0	D					A					SIMM															
addis	0 0 1 1 1 1	D					A					SIMM															
bc*x*	0 1 0 0 0 0	BO					BI					BD														AA	LK
sc	0 1 0 0 0 1	0 0 0 0 0					0 0 0 0 0					0 0 0 0 0 0 0 0 0 0 0 0 0 0 0														1	0
b*x*	0 1 0 0 1 0	LI																								AA	LK
mcrf	0 1 0 0 1 1	crfD			0 0		crfS			0 0		0 0 0 0 0					0 0 0 0 0 0 0 0 0 0										0
bclr*x*	0 1 0 0 1 1	BO					BI					0 0 0			BH		0 0 0 0 0 1 0 0 0 0										
rfid [1, 3]	0 1 0 0 1 1	0 0 0 0 0					0 0 0 0 0					0 0 0 0 0					0 0 0 0 0 1 0 0 1 0										0
crnor	0 1 0 0 1 1	crbD					crbA					crbB					0 0 0 0 1 0 0 0 0 1										0
crandc	0 1 0 0 1 1	crbD					crbA					crbB					0 0 1 0 0 0 0 0 0 1										0
isync	0 1 0 0 1 1	0 0 0 0 0					0 0 0 0 0					0 0 0 0 0					0 0 1 0 0 1 0 1 1 0										0
crxor	0 1 0 0 1 1	crbD					crbA					crbB					0 0 1 1 0 0 0 0 0 1										0
crnand	0 1 0 0 1 1	crbD					crbA					crbB					0 0 1 1 1 0 0 0 0 1										0
crand	0 1 0 0 1 1	crbD					crbA					crbB					0 1 0 0 0 0 0 0 0 1										0
creqv	0 1 0 0 1 1	crbD					crbA					crbB					0 1 0 0 1 0 0 0 0 1										0
crorc	0 1 0 0 1 1	crbD					crbA					crbB					0 1 1 0 1 0 0 0 0 1										0
cror	0 1 0 0 1 1	crbD					crbA					crbB					0 1 1 1 0 0 0 0 0 1										0
bcctr*x*	0 1 0 0 1 1	BO					BI					0 0 0			BH		1 0 0 0 0 1 0 0 0 0										LK

Note:

1. 64-bit instruction
2. Optional instruction
3. Supervisor level instruction
4. Load/store string/multiple instruction
5. Supervisor and user-level instruction
6. Optional 64-bit bridge instruction

Table A-2. Complete Instruction List Sorted by Opcode

Name	0 5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
rlwimi*x*	0 1 0 1 0 0	S					A					SH					MB					ME					Rc
rlwinm*x*	0 1 0 1 0 1	S					A					SH					MB					ME					Rc
rlwnm*x*	0 1 0 1 1 1	S					A					B					MB					ME					Rc
ori	0 1 1 0 0 0	S					A					UIMM															
oris	0 1 1 0 0 1	S					A					UIMM															
xori	0 1 1 0 1 0	S					A					UIMM															
xoris	0 1 1 0 1 1	S					A					UIMM															
andi.	0 1 1 1 0 0	S					A					UIMM															
andis.	0 1 1 1 0 1	S					A					UIMM															
rldicl*x*[1]	0 1 1 1 1 0	S					A					sh					mb						0 0 0			sh	Rc
rldicr*x*[1]	0 1 1 1 1 0	S					A					sh					me						0 0 1			sh	Rc
rldic*x*[1]	0 1 1 1 1 0	S					A					sh					mb						0 1 0			sh	Rc
rldimi*x*[1]	0 1 1 1 1 0	S					A					sh					mb						0 1 1			sh	Rc
rldcl*x*[1]	0 1 1 1 1 0	S					A					B					mb					0 1 0 0 0					Rc
rldcr*x*[1]	0 1 1 1 1 0	S					A					B					me						0 1 0 0 1				Rc
cmp	0 1 1 1 1 1	crfD			0	L	A					B					0 0 0 0 0 0 0 0 0 0										0
tw	0 1 1 1 1 1	TO					A					B					0 0 0 0 0 0 0 1 0 0										0
subfc*x*	0 1 1 1 1 1	D					A					B					OE	0 0 0 0 0 1 0 0 0									Rc
mulhdu*x*[1]	0 1 1 1 1 1	D					A					B					0	0 0 0 0 0 1 0 0 1									Rc
addc*x*	0 1 1 1 1 1	D					A					B					OE	0 0 0 0 0 1 0 1 0									Rc
mulhwu*x*	0 1 1 1 1 1	D					A					B					0	0 0 0 0 0 1 0 1 1									Rc
mfcr	0 1 1 1 1 1	D					0 0 0 0 0					0 0 0 0 0					0 0 0 0 0 1 0 0 1 1										0
lwarx	0 1 1 1 1 1	D					A					B					0 0 0 0 0 1 0 1 0 0										0
ldx[1]	0 1 1 1 1 1	D					A					B					0 0 0 0 0 1 0 1 0 1										0
lwzx	0 1 1 1 1 1	D					A					B					0 0 0 0 0 1 0 1 1 1										0
slwx	0 1 1 1 1 1	S					A					B					0 0 0 0 0 1 1 0 0 0										Rc
cntlzw*x*	0 1 1 1 1 1	S					A					0 0 0 0 0					0 0 0 0 0 1 1 0 1 0										Rc
sld*x*[1]	0 1 1 1 1 1	S					A					B					0 0 0 0 0 1 1 0 1 1										Rc
and*x*	0 1 1 1 1 1	S					A					B					0 0 0 0 0 1 1 1 0 0										Rc
cmpl	0 1 1 1 1 1	crfD			0	L	A					B					0 0 0 0 1 0 0 0 0 0										0
subf*x*	0 1 1 1 1 1	D					A					B					OE	0 0 0 1 0 1 0 0 0									Rc
ldux[1]	0 1 1 1 1 1	D					A					B					0 0 0 0 1 1 0 1 0 1										0

Note:

1. 64-bit instruction
2. Optional instruction
3. Supervisor level instruction
4. Load/store string/multiple instruction
5. Supervisor and user-level instruction
6. Optional 64-bit bridge instruction

Table A-2. Complete Instruction List Sorted by Opcode

Name	0 5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
dcbst	0 1 1 1 1 1	0 0 0 0 0					A					B					0 0 0 0 1 1 0 1 1 0										0
lwzu_x_	0 1 1 1 1 1	D					A					B					0 0 0 0 1 1 0 1 1 1										0
cntlzdx [1]	0 1 1 1 1 1	S					A					0 0 0 0 0					0 0 0 0 1 1 1 0 1 0										Rc
andc_x_	0 1 1 1 1 1	S					A					B					0 0 0 0 1 1 1 1 0 0										Rc
td [1]	0 1 1 1 1 1	TO					A					B					0 0 0 1 0 0 0 1 0 0										0
mulhd_x_ [1]	0 1 1 1 1 1	D					A					B					0	0 0 1 0 0 1 0 0 1									Rc
mulhw_x_	0 1 1 1 1 1	D					A					B					0	0 0 1 0 0 1 0 1 1									Rc
mfmsr [3]	0 1 1 1 1 1	D					0 0 0 0 0					0 0 0 0 0					0 0 0 1 0 1 0 0 1 1										0
ldarx [1]	0 1 1 1 1 1	D					A					B					0 0 0 1 0 1 0 1 0 0										0
dcbf	0 1 1 1 1 1	0 0 0 0 0					A					B					0 0 0 1 0 1 0 1 1 0										0
lbz_x_	0 1 1 1 1 1	D					A					B					0 0 0 1 0 1 0 1 1 1										0
neg_x_	0 1 1 1 1 1	D					A					0 0 0 0 0					OE	0 0 1 1 0 1 0 0 0									Rc
lbzu_x_	0 1 1 1 1 1	D					A					B					0 0 0 1 1 1 0 1 1 1										0
nor_x_	0 1 1 1 1 1	S					A					B					0 0 0 1 1 1 1 1 0 0										Rc
subfe_x_	0 1 1 1 1 1	D					A					B					OE	0 1 0 0 0 1 0 0 0									Rc
adde_x_	0 1 1 1 1 1	D					A					B					OE	0 1 0 0 0 1 0 1 0									Rc
mtcrf	0 1 1 1 1 1	S					0	CRM								0	0 0 1 0 0 1 0 0 0 0										0
mtmsr [3, 6]	31	S					0 0 0 0				L	0 0 0 0 0					0 0 1 0 0 1 0 0 1 0										0
stdx [1]	0 1 1 1 1 1	S					A					B					0 0 1 0 0 1 0 1 0 1										0
stwcx.	0 1 1 1 1 1	S					A					B					0 0 1 0 0 1 0 1 1 0										1
stwx	0 1 1 1 1 1	S					A					B					0 0 1 0 0 1 0 1 1 1										0
mtmsrd [1, 3]	0 1 1 1 1 1	S					0 0 0 0				L	0 0 0 0 0					0 0 1 0 1 1 0 0 1 0										0
stdux [1]	0 1 1 1 1 1	S					A					B					0 0 1 0 1 1 0 1 0 1										0
stwux	0 1 1 1 1 1	S					A					B					0 0 1 0 1 1 0 1 1 1										0
subfze_x_	0 1 1 1 1 1	D					A					0 0 0 0 0					OE	0 1 1 0 0 1 0 0 0									Rc
addze_x_	0 1 1 1 1 1	D					A					0 0 0 0 0					OE	0 1 1 0 0 1 0 1 0									Rc
mtsr [6]	0 1 1 1 1 1	S					0	SR				0 0 0 0 0					0 0 1 1 0 1 0 0 1 0										0
stdcx. [1]	0 1 1 1 1 1	S					A					B					0 0 1 1 0 1 0 1 1 0										1
stbx	0 1 1 1 1 1	S					A					B					0 0 1 1 0 1 0 1 1 1										0
subfme_x_	0 1 1 1 1 1	D					A					0 0 0 0 0					OE	0 1 1 1 0 1 0 0 0									Rc
mulld_x_ [1]	0 1 1 1 1 1	D					A					B					OE	0 1 1 1 0 1 0 0 1									Rc
addme_x_	0 1 1 1 1 1	D					A					0 0 0 0 0					OE	0 1 1 1 0 1 0 1 0									Rc

Note:

1. 64-bit instruction
2. Optional instruction
3. Supervisor level instruction
4. Load/store string/multiple instruction
5. Supervisor and user-level instruction
6. Optional 64-bit bridge instruction

Table A-2. Complete Instruction List Sorted by Opcode

Name	0 5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
mullw*x*	0 1 1 1 1 1	D					A					B					OE	0 1 1 1 0 1 0 1 1									Rc
mtsrin [6]	0 1 1 1 1 1	S					0 0 0 0 0					B					0 0 1 1 1 1 0 0 1 0										0
mtocrf	0 1 1 1 1 1	S					1	CRM								0	0 0 1 0 0 1 0 0 0 0										0
dcbtst	0 1 1 1 1 1	0 0 0 0 0					A					B					0 0 1 1 1 1 0 1 1 0										0
stbux	0 1 1 1 1 1	S					A					B					0 0 1 1 1 1 0 1 1 1										0
add*x*	0 1 1 1 1 1	D					A					B					OE	1 0 0 0 0 1 0 1 0									Rc
dcbt	0 1 1 1 1 1	0 0 0 0 0					A					B					0 1 0 0 0 1 0 1 1 0										0
lhzx	0 1 1 1 1 1	D					A					B					0 1 0 0 0 1 0 1 1 1										0
eqv*x*	0 1 1 1 1 1	S					A					B					0 1 0 0 0 1 1 1 0 0										Rc
tlbiel [3]	0 1 1 1 1 1	0 0 0 0				L	0 0 0 0 0					B					0 1 0 0 0 1 0 0 1 0										0
tlbie [3]	0 1 1 1 1 1	0 0 0 0				L	0 0 0 0 0					B					0 1 0 0 1 1 0 0 1 0										0
eciwx	0 1 1 1 1 1	D					A					B					0 1 0 0 1 1 0 1 1 0										0
lhzux	0 1 1 1 1 1	D					A					B					0 1 0 0 1 1 0 1 1 1										0
xor*x*	0 1 1 1 1 1	S					A					B					0 1 0 0 1 1 1 1 0 0										Rc
mfspr [5]	0 1 1 1 1 1	D					spr										0 1 0 1 0 1 0 0 1 1										0
lwax [1]	0 1 1 1 1 1	D					A					B					0 1 0 1 0 1 0 1 0 1										0
lhax	0 1 1 1 1 1	D					A					B					0 1 0 1 0 1 0 1 1 1										0
tlbia [3]	0 1 1 1 1 1	0 0 0 0 0					0 0 0 0 0					0 0 0 0 0					0 1 0 1 1 1 0 0 1 0										0
mftb	0 1 1 1 1 1	D					tbr										0 1 0 1 1 1 0 0 1 1										0
lwaux [1]	0 1 1 1 1 1	D					A					B					0 1 0 1 1 1 0 1 0 1										0
lhaux	0 1 1 1 1 1	D					A					B					0 1 0 1 1 1 0 1 1 1										0
sthx	0 1 1 1 1 1	S					A					B					0 1 1 0 0 1 0 1 1 1										0
orc*x*	0 1 1 1 1 1	S					A					B					0 1 1 0 0 1 1 1 0 0										Rc
sradi*x* [1]	0 1 1 1 1 1	S					A					sh					1 1 0 0 1 1 1 0 1									sh	Rc
slbie [1, 2, 3]	0 1 1 1 1 1	0 0 0 0 0					0 0 0 0 0					B					0 1 1 0 1 1 0 0 1 0										0
ecowx	0 1 1 1 1 1	S					A					B					0 1 1 0 1 1 0 1 1 0										0
sthux	0 1 1 1 1 1	S					A					B					0 1 1 0 1 1 0 1 1 1										0
or*x*	0 1 1 1 1 1	S					A					B					0 1 1 0 1 1 1 1 0 0										Rc
divdu*x* [1]	0 1 1 1 1 1	D					A					B					OE	1 1 1 0 0 1 0 0 1									Rc
divwu*x*	0 1 1 1 1 1	D					A					B					OE	1 1 1 0 0 1 0 1 1									Rc
mtspr [5]	0 1 1 1 1 1	S					spr										0 1 1 1 0 1 0 0 1 1										0
nand*x*	0 1 1 1 1 1	S					A					B					0 1 1 1 0 1 1 1 0 0										Rc

Note:

1. 64-bit instruction
2. Optional instruction
3. Supervisor level instruction
4. Load/store string/multiple instruction
5. Supervisor and user-level instruction
6. Optional 64-bit bridge instruction

Table A-2. Complete Instruction List Sorted by Opcode

Name	0 5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
divd*x* [1]	0 1 1 1 1 1	D					A					B					OE	1 1 1 1 0 1 0 0 1									Rc
divw*x*	0 1 1 1 1 1	D					A					B					OE	1 1 1 1 0 1 0 1 1									Rc
slbia [1, 2, 3]	0 1 1 1 1 1	0 0 0 0 0					0 0 0 0 0					0 0 0 0 0					0 1 1 1 1 1 0 0 1 0										0
lswx [4]	0 1 1 1 1 1	D					A					B					1 0 0 0 0 1 0 1 0 1										0
lwbrx	0 1 1 1 1 1	D					A					B					1 0 0 0 0 1 0 1 1 0										0
lfsx	0 1 1 1 1 1	D					A					B					1 0 0 0 0 1 0 1 1 1										0
srw*x*	0 1 1 1 1 1	S					A					B					1 0 0 0 0 1 1 0 0 0										Rc
srd*x* [1]	0 1 1 1 1 1	S					A					B					1 0 0 0 0 1 1 0 1 1										Rc
tlbsync [, 3]	0 1 1 1 1 1	0 0 0 0 0					0 0 0 0 0					0 0 0 0 0					1 0 0 0 1 1 0 1 1 0										0
lfsux	0 1 1 1 1 1	D					A					B					1 0 0 0 1 1 0 1 1 1										0
mfsr [3, 6]	0 1 1 1 1 1	D					0	SR				0 0 0 0 0					1 0 0 1 0 1 0 0 1 1										0
lswi [4]	0 1 1 1 1 1	D					A					NB					1 0 0 1 0 1 0 1 0 1										0
sync	0 1 1 1 1 1	0 0 0			L		0 0 0 0 0					0 0 0 0 0					1 0 0 1 0 1 0 1 1 0										0
lfdx	0 1 1 1 1 1	D					A					B					1 0 0 1 0 1 0 1 1 1										0
lfdux	0 1 1 1 1 1	D					A					B					1 0 0 1 1 1 0 1 1 1										0
mfsrin [3, 6]	0 1 1 1 1 1	D					0 0 0 0 0					B					1 0 1 0 0 1 0 0 1 1										0
slbmfee [1]	0 1 1 1 1 1	D					0 0 0 0 0					B					1 1 1 0 0 1 0 0 1 1										0
slbmfev [1]	0 1 1 1 1 1	D					0 0 0 0 0					B					1 1 0 1 0 1 0 0 1 1										0
slbmte [1]	0 1 1 1 1 1	D					0 0 0 0 0					B					0 1 1 0 0 1 0 0 1 0										0
mfocrf	0 1 1 1 1 1	D					1	CRM								0	0 0 0 0 0 1 0 0 1 1										0
stswx [4]	0 1 1 1 1 1	S					A					B					1 0 1 0 0 1 0 1 0 1										0
stwbrx	0 1 1 1 1 1	S					A					B					1 0 1 0 0 1 0 1 1 0										0
stfsx	0 1 1 1 1 1	S					A					B					1 0 1 0 0 1 0 1 1 1										0
stfsux	0 1 1 1 1 1	S					A					B					1 0 1 0 1 1 0 1 1 1										0
stswi [4]	0 1 1 1 1 1	S					A					NB					1 0 1 1 0 1 0 1 0 1										0
stfdx	0 1 1 1 1 1	S					A					B					1 0 1 1 0 1 0 1 1 1										0
stfdux	0 1 1 1 1 1	S					A					B					1 0 1 1 1 1 0 1 1 1										0
lhbrx	0 1 1 1 1 1	D					A					B					1 1 0 0 0 1 0 1 1 0										0
sraw*x*	0 1 1 1 1 1	S					A					B					1 1 0 0 0 1 1 0 0 0										Rc
srad*x* [1]	0 1 1 1 1 1	S					A					B					1 1 0 0 0 1 1 0 1 0										Rc
srawi*x*	0 1 1 1 1 1	S					A					SH					1 1 0 0 1 1 1 0 0 0										Rc
eieio	0 1 1 1 1 1	0 0 0 0 0					0 0 0 0 0					0 0 0 0 0					1 1 0 1 0 1 0 1 1 0										0

Note:

1. 64-bit instruction
2. Optional instruction
3. Supervisor level instruction
4. Load/store string/multiple instruction
5. Supervisor and user-level instruction
6. Optional 64-bit bridge instruction

Table A-2. Complete Instruction List Sorted by Opcode

Name	0 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30	31
sthbrx	0 1 1 1 1 1	S	A	B	1 1 1 0 0 1 0 1 1 0	0
extsh*x*	0 1 1 1 1 1	S	A	0 0 0 0 0	1 1 1 0 0 1 1 0 1 0	Rc
extsb*x*	0 1 1 1 1 1	S	A	0 0 0 0 0	1 1 1 0 1 1 1 0 1 0	Rc
icbi	0 1 1 1 1 1	0 0 0 0 0	A	B	1 1 1 1 0 1 0 1 1 0	0
stfiwx [2]	0 1 1 1 1 1	S	A	B	1 1 1 1 0 1 0 1 1 1	0
extsw*x* [1]	0 1 1 1 1 1	S	A	0 0 0 0 0	1 1 1 1 0 1 1 0 1 0	Rc
dcbz	0 1 1 1 1 1	0 0 0 0 0	A	B	1 1 1 1 1 1 0 1 1 0	0
lwz	1 0 0 0 0 0	D	A	d		
lwzu	1 0 0 0 0 1	D	A	d		
lbz	1 0 0 0 1 0	D	A	d		
lbzu	1 0 0 0 1 1	D	A	d		
stw	1 0 0 1 0 0	S	A	d		
stwu	1 0 0 1 0 1	S	A	d		
stb	1 0 0 1 1 0	S	A	d		
stbu	1 0 0 1 1 1	S	A	d		
lhz	1 0 1 0 0 0	D	A	d		
lhzu	1 0 1 0 0 1	D	A	d		
lha	1 0 1 0 1 0	D	A	d		
lhau	1 0 1 0 1 1	D	A	d		
sth	1 0 1 1 0 0	S	A	d		
sthu	1 0 1 1 0 1	S	A	d		
lmw [4]	1 0 1 1 1 0	D	A	d		
stmw [4]	1 0 1 1 1 1	S	A	d		
lfs	1 1 0 0 0 0	D	A	d		
lfsu	1 1 0 0 0 1	D	A	d		
lfd	1 1 0 0 1 0	D	A	d		
lfdu	1 1 0 0 1 1	D	A	d		
stfs	1 1 0 1 0 0	S	A	d		
stfsu	1 1 0 1 0 1	S	A	d		
stfd	1 1 0 1 1 0	S	A	d		
stfdu	1 1 0 1 1 1	S	A	d		
ld [1]	1 1 1 0 1 0	D	A	ds		0 0

Note:

1. 64-bit instruction
2. Optional instruction
3. Supervisor level instruction
4. Load/store string/multiple instruction
5. Supervisor and user-level instruction
6. Optional 64-bit bridge instruction

Table A-2. Complete Instruction List Sorted by Opcode

Name	0 5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
ldu[1]	1 1 1 0 1 0	D					A					ds														0 1	
lwa[1]	1 1 1 0 1 0	D					A					ds														1 0	
fdivs*x*	1 1 1 0 1 1	D					A					B					0 0 0 0 0					1 0 0 1 0					Rc
fsubs*x*	1 1 1 0 1 1	D					A					B					0 0 0 0 0					1 0 1 0 0					Rc
fadds*x*	1 1 1 0 1 1	D					A					B					0 0 0 0 0					1 0 1 0 1					Rc
fsqrts*x*[2]	1 1 1 0 1 1	D					0 0 0 0 0					B					0 0 0 0 0					1 0 1 1 0					Rc
fres*x*[2]	1 1 1 0 1 1	D					0 0 0 0 0					B					0 0 0 0 0					1 1 0 0 0					Rc
fmuls*x*	1 1 1 0 1 1	D					A					0 0 0 0 0					C					1 1 0 0 1					Rc
fmsubs*x*	1 1 1 0 1 1	D					A					B					C					1 1 1 0 0					Rc
fmadds*x*	1 1 1 0 1 1	D					A					B					C					1 1 1 0 1					Rc
fnmsubs*x*	1 1 1 0 1 1	D					A					B					C					1 1 1 1 0					Rc
fnmadds*x*	1 1 1 0 1 1	D					A					B					C					1 1 1 1 1					Rc
std[1]	1 1 1 1 1 0	S					A					ds														0 0	
stdu[1]	1 1 1 1 1 0	S					A					ds														0 1	
fcmpu	1 1 1 1 1 1	crfD			0 0		A					B					0 0 0 0 0 0 0 0 0 0										0
frsp*x*	1 1 1 1 1 1	D					0 0 0 0 0					B					0 0 0 0 0 0 1 1 0 0										Rc
fctiw*x*	1 1 1 1 1 1	D					0 0 0 0 0					B					0 0 0 0 0 0 1 1 1 0										
fctiwz*x*	1 1 1 1 1 1	D					0 0 0 0 0					B					0 0 0 0 0 0 1 1 1 1										Rc
fdiv*x*	1 1 1 1 1 1	D					A					B					0 0 0 0 0					1 0 0 1 0					Rc
fsub*x*	1 1 1 1 1 1	D					A					B					0 0 0 0 0					1 0 1 0 0					Rc
fadd*x*	1 1 1 1 1 1	D					A					B					0 0 0 0 0					1 0 1 0 1					Rc
fsqrt*x*[2]	1 1 1 1 1 1	D					0 0 0 0 0					B					0 0 0 0 0					1 0 1 1 0					Rc
fsel*x*[2]	1 1 1 1 1 1	D					A					B					C					1 0 1 1 1					Rc
fmul*x*	1 1 1 1 1 1	D					A					0 0 0 0 0					C					1 1 0 0 1					Rc
frsqrte*x*[2]	1 1 1 1 1 1	D					0 0 0 0 0					B					0 0 0 0 0					1 1 0 1 0					Rc
fmsub*x*	1 1 1 1 1 1	D					A					B					C					1 1 1 0 0					Rc
fmadd*x*	1 1 1 1 1 1	D					A					B					C					1 1 1 0 1					Rc
fnmsub*x*	1 1 1 1 1 1	D					A					B					C					1 1 1 1 0					Rc
fnmadd*x*	1 1 1 1 1 1	D					A					B					C					1 1 1 1 1					Rc
fcmpo	1 1 1 1 1 1	crfD			0 0		A					B					0 0 0 0 1 0 0 0 0 0										0
mtfsb1*x*	1 1 1 1 1 1	crbD					0 0 0 0 0					0 0 0 0 0					0 0 0 0 1 0 0 1 1 0										Rc
fneg*x*	1 1 1 1 1 1	D					0 0 0 0 0					B					0 0 0 0 1 0 1 0 0 0										Rc

Note:

1. 64-bit instruction
2. Optional instruction
3. Supervisor level instruction
4. Load/store string/multiple instruction
5. Supervisor and user-level instruction
6. Optional 64-bit bridge instruction

Table A-2. Complete Instruction List Sorted by Opcode

Name	0 5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
mcrfs	1 1 1 1 1 1	crfD			0 0		crfS			0 0		0 0 0 0 0					0 0 0 1 0 0 0 0 0 0										0
mtfsb0*x*	1 1 1 1 1 1	crbD					0 0 0 0 0					0 0 0 0 0					0 0 0 1 0 0 0 1 1 0										Rc
fmr*x*	1 1 1 1 1 1	D					0 0 0 0 0					B					0 0 0 1 0 0 1 0 0 0										Rc
mtfsfi*x*	1 1 1 1 1 1	crfD			0 0		0 0 0 0 0					IMM				0	0 0 1 0 0 0 0 1 1 0										Rc
fnabs*x*	1 1 1 1 1 1	D					0 0 0 0 0					B					0 0 1 0 0 0 1 0 0 0										Rc
fabs*x*	1 1 1 1 1 1	D					0 0 0 0 0					B					0 1 0 0 0 0 1 0 0 0										Rc
mffs*x*	1 1 1 1 1 1	D					0 0 0 0 0					0 0 0 0 0					1 0 0 1 0 0 0 1 1 1										Rc
mtfsf*x*	1 1 1 1 1 1	0	FM								0	B					1 0 1 1 0 0 0 1 1 1										Rc
fctid*x*[1]	1 1 1 1 1 1	D					0 0 0 0 0					B					1 1 0 0 1 0 1 1 1 0										Rc
fctidz*x*[1]	1 1 1 1 1 1	D					0 0 0 0 0					B					1 1 0 0 1 0 1 1 1 1										Rc
fcfidx[1]	1 1 1 1 1 1	D					0 0 0 0 0					B					1 1 0 1 0 0 1 1 1 0										Rc

Note:

1. 64-bit instruction
2. Optional instruction
3. Supervisor level instruction
4. Load/store string/multiple instruction
5. Supervisor and user-level instruction
6. Optional 64-bit bridge instruction

A.3 Instructions Grouped by Functional Categories

Table A-3 through *Table A-30* list the PowerPC instructions grouped by function.

Key: ▨ Reserved bits

Table A-3. Integer Arithmetic Instructions

Name	05	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
add*x*	31	D					A					B					OE	266									Rc
addc*x*	31	D					A					B					OE	10									Rc
adde*x*	31	D					A					B					OE	138									Rc
addi	14	D					A					SIMM															
addic	12	D					A					SIMM															
addic.	13	D					A					SIMM															
addis	15	D					A					SIMM															
addme*x*	31	D					A					0 0 0 0 0					OE	234									Rc
addze*x*	31	D					A					0 0 0 0 0					OE	202									Rc
divd*x* [1]	31	D					A					B					OE	489									Rc
divdu*x* [1]	31	D					A					B					OE	457									Rc
divw***x***	31	D					A					B					OE	491									Rc
divwu*x*	31	D					A					B					OE	459									Rc
mulhd*x* [1]	31	D					A					B					0	73									Rc
mulhdu***x*** [1]	31	D					A					B					0	9									Rc
mulhw*x*	31	D					A					B					0	75									Rc
mulhwu*x*	31	D					A					B					0	11									Rc
mulld [1]	31	D					A					B					OE	233									Rc
mulli	07	D					A					SIMM															
mullw*x*	31	D					A					B					OE	235									Rc
neg*x*	31	D					A					0 0 0 0 0					OE	104									Rc
subf*x*	31	D					A					B					OE	40									Rc
subfc*x*	31	D					A					B					OE	8									Rc
subfic*x*	08	D					A					SIMM															
subfe*x*	31	D					A					B					OE	136									Rc
subfme*x*	31	D					A					0 0 0 0 0					OE	232									Rc
subfze*x*	31	D					A					0 0 0 0 0					OE	200									Rc

Note:

1. 64-bit instruction

Table A-4. Integer Compare Instructions

Name	0 5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
cmp	31	crfD			0	L	A					B					0 0 0 0 0 0 0 0 0 0										0
cmpi	11	crfD			0	L	A					SIMM															
cmpl	31	crfD			0	L	A					B					32										0
cmpli	10	crfD			0	L	A					UIMM															

Table A-5. Integer Logical Instructions

Name	0 5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
and*x*	31	S					A					B					28										Rc
andc*x*	31	S					A					B					60										Rc
andi.	28	S					A					UIMM															
andis.	29	S					A					UIMM															
cntlzd*x* [1]	31	S					A					0 0 0 0 0					58										Rc
cntlzw*x*	31	S					A					0 0 0 0 0					26										Rc
eqv*x*	31	S					A					B					284										Rc
extsb*x*	31	S					A					0 0 0 0 0					954										Rc
extsh*x*	31	S					A					0 0 0 0 0					922										Rc
extsw*x* [1]	31	S					A					0 0 0 0 0					986										Rc
nand*x*	31	S					A					B					476										Rc
nor*x*	31	S					A					B					124										Rc
or*x*	31	S					A					B					444										Rc
orcx	31	S					A					B					412										Rc
ori	24	S					A					UIMM															
oris	25	S					A					UIMM															
xor*x*	31	S					A					B					316										Rc
xori	26	S					A					UIMM															
xoris	27	S					A					UIMM															

Note:

1. 64-bit instruction

Table A-6. Integer Rotate Instructions

Name	0 5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
rldclx [1]	30	S					A					B					mb						8				Rc
rldcr*x* [1]	30	S					A					B					me						9				Rc
rldic*x* [1]	30	S					A					sh					mb						2			sh	Rc
rldicl*x* [1]	30	S					A					sh					mb						0			sh	Rc
rldicr*x* [1]	30	S					A					sh					me						1			sh	Rc
rldimi*x* [1]	30	S					A					sh					mb						3			sh	Rc
rlwimi*x*	22	S					A					SH					MB					ME					Rc
rlwinm*x*	20	S					A					SH					MB					ME					Rc
rlwnm*x*	21	S					A					SH					MB					ME					Rc

Note:

1. 64-bit instruction

Table A-7. Integer Shift Instructions

Name	0 5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
sld*x* [1]	31	S					A					B					27										Rc
slw*x*	31	S					A					B					24										Rc
srad*x* [1]	31	S					A					B					794										Rc
sradi*x* [1]	31	S					A					sh					413									sh	Rc
sraw*x*	31	S					A					B					792										Rc
srawi*x*	31	S					A					SH					824										Rc
srd*x* [1]	31	S					A					B					539										Rc
srw*x*	31	S					A					B					536										Rc

Note:

1. 64-bit instruction

Table A-8. Floating-Point Arithmetic Instructions

Name	0 5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
fadd_x_	63	D					A					B					0 0 0 0 0					21					Rc
fadds_x_	59	D					A					B					0 0 0 0 0					21					Rc
fdiv_x_	63	D					A					B					0 0 0 0 0					18					Rc
fdivs_x_	59	D					A					B					0 0 0 0 0					18					Rc
fmul_x_	63	D					A					0 0 0 0 0					C					25					Rc
fmuls_x_	59	D					A					0 0 0 0 0					C					25					Rc
fres_x_ [1]	59	D					0 0 0 0 0					B					0 0 0 0 0					24					Rc
frsqrte_x_ [1]	63	D					0 0 0 0 0					B					0 0 0 0 0					26					Rc
fsub_x_	63	D					A					B					0 0 0 0 0					20					Rc
fsubs_x_	59	D					A					B					0 0 0 0 0					20					Rc
fsel_x_ [1]	63	D					A					B					C					23					Rc
fsqrtx [1]	63	D					0 0 0 0 0					B					0 0 0 0 0					22					Rc
fsqrts_x_ [1]	59	D					0 0 0 0 0					B					0 0 0 0 0					22					Rc

Note:

1. Optional instruction

Table A-9. Floating-Point Multiply-Add Instructions

Name	0 5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
fmadd_x_	63	D					A					B					C					29					Rc
fmadds_x_	59	D					A					B					C					29					Rc
fmsub_x_	63	D					A					B					C					28					Rc
fmsubs_x_	59	D					A					B					C					28					Rc
fnmadd_x_	63	D					A					B					C					31					Rc
fnmadds_x_	59	D					A					B					C					31					Rc
fnmsub_x_	63	D					A					B					C					30					Rc
fnmsubs_x_	59	D					A					B					C					30					Rc

Table A-10. Floating-Point Rounding and Conversion Instructions

Name	0 5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
fcfid*x* [1]	63	D					0 0 0 0 0					B					846										Rc
fctid*x* [1]	63	D					0 0 0 0 0					B					814										Rc
fctidz*x* [1]	63	D					0 0 0 0 0					B					815										Rc
fctiw*x*	63	D					0 0 0 0 0					B					14										Rc
fctiwz*x*	63	D					0 0 0 0 0					B					15										Rc
frsp*x*	63	D					0 0 0 0 0					B					12										Rc

Note:

1. 64-bit instruction

Table A-11. Floating-Point Compare Instructions

Name	0 5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
fcmpo	63	crfD			0 0		A					B					32										0
fcmpu	63	crfD			0 0		A					B					0										0

Table A-12. Floating-Point Status and Control Register Instructions

Name	0 5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
mcrfs	63	crfD			0 0		crfS			0 0		0 0 0 0 0					64										0
mffs*x*	63	D					0 0 0 0 0					0 0 0 0 0					583										Rc
mtfsb0*x*	63	crbD					0 0 0 0 0					0 0 0 0 0					70										Rc
mtfsb1*x*	63	crbD					0 0 0 0 0					0 0 0 0 0					38										Rc
mtfsf*x*	31	0	FM								0	B					711										Rc
mtfsfi*x*	63	crfD			0 0		0 0 0 0 0					IMM				0	134										Rc

Table A-13. Integer Load Instructions

Name	0 5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
lbz	34	D					A					d															
lbzu	35	D					A					d															
lbzux	31	D					A					B					119										0
lbz*x*	31	D					A					B					87										0
ld [1]	58	D					A					ds														0	
ldu [1]	58	D					A					ds														1	
ldu*x* [1]	31	D					A					B					53										0
ldx [1]	31	D					A					B					21										0
lha	42	D					A					d															
lhau	43	D					A					d															
lhaux	31	D					A					B					375										0
lhax	31	D					A					B					343										0
lhz	40	D					A					d															
lhzu	41	D					A					d															
lhzux	31	D					A					B					311										0
lhzx	31	D					A					B					279										0
lwa [1]	58	D					A					ds														2	
lwaux [1]	31	D					A					B					373										0
lwax [1]	31	D					A					B					341										0
lwz	32	D					A					d															
lwzu	33	D					A					d															
lwzux	31	D					A					B					55										0
lwzx	31	D					A					B					23										0

Note:

1. 64-bit instruction

Table A-14. Integer Store Instructions

Name	0 5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
stb	38	S					A					d															
stbu	39	S					A					d															
stbux	31	S					A					B					247										0
stbx	31	S					A					B					215										0
std [1]	62	S					A					ds															0
stdu [1]	62	S					A					ds															1
stdux [1]	31	S					A					B					181										0
stdx [1]	31	S					A					B					149										0
sth	44	S					A					d															
sthu	45	S					A					d															
sthux	31	S					A					B					439										0
sthx	31	S					A					B					407										0
stw	36	S					A					d															
stwu	37	S					A					d															
stwux	31	S					A					B					183										0
stwx	31	S					A					B					151										0

Note:

1. 64-bit instruction

Table A-15. Integer Load and Store with Byte Reverse Instructions

Name	0 5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
lhbrx	31	D					A					B					790										0
lwbrx	31	D					A					B					534										0
sthbrx	31	S					A					B					918										0
stwbrx	31	S					A					B					662										0

Table A-16. Integer Load and Store Multiple Instructions

Name	0 5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
lmw [1]	46	D					A					d															
stmw [1]	47	S					A					d															

Note:

1. Load/store string/multiple instruction

Table A-17. Integer Load and Store String Instructions

Name	0 5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
lswi [1]	31	D					A					NB					597										0
lswx [1]	31	D					A					B					533										0
stswi [1]	31	S					A					NB					725										0
stswx [1]	31	S					A					B					661										0

Note:

1. Load/store string/multiple instruction

Table A-18. Memory Synchronization Instructions

Name	0 5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
eieio	31	0 0 0 0 0					0 0 0 0 0					0 0 0 0 0					854										0
isync	19	0 0 0 0 0					0 0 0 0 0					0 0 0 0 0					150										0
ldarx [1]	31	D					A					B					84										0
lwarx	31	D					A					B					20										0
stdcx. [1]	31	S					A					B					214										1
stwcx.	31	S					A					B					150										1
sync	31	0 0 0			L		0 0 0 0 0					0 0 0 0 0					598										0

Note:

1. 64-bit instruction

Table A-19. Floating-Point Load Instructions

Name	0 5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
lfd	50	D					A					d															
lfdu	51	D					A					d															
lfdux	31	D					A					B					631										0
lfdx	31	D					A					B					599										0
lfs	48	D					A					d															
lfsu	49	D					A					d															
lfsux	31	D					A					B					567										0
lfsx	31	D					A					B					535										0

Table A-20. Floating-Point Store Instructions

Name	0 5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
stfd	54	S					A					d															
stfdu	55	S					A					d															
stfdux	31	S					A					B					759										0
stfdx	31	S					A					B					727										0
stfiwx [1]	31	S					A					B					983										0
stfs	52	S					A					d															
stfsu	53	S					A					d															
stfsux	31	S					A					B					695										0
stfsx	31	S					A					B					663										0

Note:

1. Optional instruction

Table A-21. Floating-Point Move Instructions

Name	0 5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
fabs*x*	63	D					0 0 0 0 0					B					264										Rc
fmr*x*	63	D					0 0 0 0 0					B					72										Rc
fnabs*x*	63	D					0 0 0 0 0					B					136										Rc
fneg*x*	63	D					0 0 0 0 0					B					40										Rc

Table A-22. Branch Instructions

Name	0 5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
b*x*	18	LI																								AA	LK
bc*x*	16	BO					BI					BD														AA	LK
bcctr*x*	19	BO					BI					0 0 0			BH		528										LK
bclr*x*	19	BO					BI					0 0 0			BH		16										LK

Table A-23. Condition Register Logical Instructions

Name	0 5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
crand	19	crbD					crbA					crbB					257										0
crandc	19	crbD					crbA					crbB					129										0
creqv	19	crbD					crbA					crbB					289										0
crnand	19	crbD					crbA					crbB					225										0
crnor	19	crbD					crbA					crbB					33										0
cror	19	crbD					crbA					crbB					449										0
crorc	19	crbD					crbA					crbB					417										0
crxor	19	crbD					crbA					crbB					193										0
mcrf	19	crfD			0 0		crfS			0 0		0 0 0 0 0					0 0 0 0 0 0 0 0 0 0										0

Table A-24. System Linkage Instructions

Name	0 5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
rfid [1, 2]	19	0 0 0 0 0					0 0 0 0 0					0 0 0 0 0					18										0
sc	17	0 0 0 0 0					0 0 0 0 0					0 0 0 0 0 0 0 0 0 0 0 0 0 0														1	0

Note:
1. Supervisor-level instruction
2. 64-bit instruction

Table A-25. Trap Instructions

Name	0 5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
td [1]	31	TO					A					B					68										0
tdi [1]	03	TO					A					SIMM															
tw	31	TO					A					B					4										0
twi	03	TO					A					SIMM															

Note:
1. 64-bit instruction

Table A-26. Processor Control Instructions

Name	05	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
mfcr	31	D					0 0 0 0 0					0 0 0 0 0					19										0
mfocrf	31	D					1	CRM								0	19										0
mfmsr [1]	31	D					0 0 0 0 0					0 0 0 0 0					83										0
mfspr [2]	31	D					spr										339										0
mftb	31	D					tpr										371										0
mtcrf	31	S					0	CRM								0	144										0
mtocrf	31	S					1	CRM								0	144										0
mtmsr [1, 3]	31	S					0 0 0 0				L	0 0 0 0 0					146										0
mtmsrd [1, 4]	31	S					0 0 0 0				L	0 0 0 0 0					178										0
mtspr [2]	31	D					spr										467										0

Note:

1. Supervisor-level instruction
2. Supervisor and user-level instruction
3. Optional 64-bit bridge instruction
4. 64-bit instruction

Table A-27. Cache Management Instructions

Name	0 5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
dcbf	31	0 0 0 0 0					A					B					86										0
dcbst	31	0 0 0 0 0					A					B					54										0
dcbt	31	0 0 0 0 0					A					B					278										0
dcbtst	31	0 0 0 0 0					A					B					246										0
dcbz	31	0 0 0 0 0					A					B					1014										0
icbi	31	0 0 0 0 0					A					B					982										0

Table A-28. Segment Register Manipulation Instructions

Name	05	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
mfsr [1, 2]	31	D					0	SR				0 0 0 0 0					595										0
mfsrin [1, 2]	31	D					0 0 0 0 0					B					659										0
mtsr [1, 2]	31	S					0	SR				0 0 0 0 0					210										0
mtsrin [1, 2]	31	S					0 0 0 0 0					B					242										0

Note:

1. Supervisor-level instruction
2. Optional 64-bit bridge instruction

Table A-29. Lookaside Buffer Management Instructions

Name	0 5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
slbia [1,2,3]	31	0 0 0 0 0					0 0 0 0 0					0 0 0 0 0					498										0
slbie [1,2,3]	31	0 0 0 0 0					0 0 0 0 0					B					434										0
slbmfee [3]	31	D					0 0 0 0 0					B					915										0
slbmfev [3]	31	D					0 0 0 0 0					B					851										0
slbmte [3]	31	D					0 0 0 0 0					B					402										0
tlbia [1,2]	31	0 0 0 0 0					0 0 0 0 0					0 0 0 0 0					370										0
tlbie [1,2]	31	0 0 0 0				L	0 0 0 0 0					B					306										0
tlbiel [1,2]	31	0 0 0 0				L	0 0 0 0 0					B					274										0
tlbsync [1,2]	31	0 0 0 0 0					0 0 0 0 0					0 0 0 0 0					566										0

Note:

1. Supervisor-level instruction
2. Optional instruction
3. 64-bit instruction

Table A-30. External Control Instructions

Name	0 5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
eciwx	31	D					A					B					310										0
ecowx	31	S					A					B					438										0

A.4 Instructions Sorted by Form

Table A-31 through *Table A-45* list the PowerPC instructions grouped by form.

Key:

▨	Reserved bits

Table A-31. I-Form

	OPCD	LI																								AA	LK
Specific Instruction																											
Name	0 5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
b*x*	18	LI																								AA	LK

Table A-32. B-Form

	OPCD	BO					BI					BD														AA	LK
Specific Instruction																											
Name	0 5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
bc*x*	16	BO					BI					BD														AA	LK

Table A-33. SC-Form

	OPCD	0 0 0 0 0					0 0 0 0 0					0 0 0 0 0 0 0 0 0 0 0 0 0 0 0														1	0
Specific Instruction																											
Name	0 5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
sc	17	0 0 0 0 0					0 0 0 0 0					0 0 0 0 0 0 0 0 0 0 0 0 0 0 0														1	0

Table A-34. D-Form

	OPCD	D					A					d															
	OPCD	D					A					SIMM															
	OPCD	S					A					d															
	OPCD	S					A					UIMM															
	OPCD	crfD			0	L	A					SIMM															
	OPCD	crfD			0	L	A					UIMM															
	OPCD	TO					A					SIMM															
Specific Instructions																											
Name	0 5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
addi	14	D					A					SIMM															
addic	12	D					A					SIMM															
addic.	13	D					A					SIMM															
addis	15	D					A					SIMM															
andi.	28	S					A					UIMM															
andis.	29	S					A					UIMM															
cmpi	11	crfD			0	L	A					SIMM															
cmpli	10	crfD			0	L	A					UIMM															
lbz	34	D					A					d															
lbzu	35	D					A					d															
lfd	50	D					A					d															
lfdu	51	D					A					d															
lfs	48	D					A					d															
lfsu	49	D					A					d															
lha	42	D					A					d															
lhau	43	D					A					d															
lhz	40	D					A					d															
lhzu	41	D					A					d															
lmw [1]	46	D					A					d															
lwz	32	D					A					d															
lwzu	33	D					A					d															
mulli	7	D					A					SIMM															
ori	24	S					A					UIMM															
oris	25	S					A					UIMM															
stb	38	S					A					d															

Note:

1. Load/store string/multiple instruction
2. 64-bit instruction

Table A-34. D-Form

stbu	39	S	A	d
stfd	54	S	A	d
stfdu	55	S	A	d
stfs	52	S	A	d
stfsu	53	S	A	d
sth	44	S	A	d
sthu	45	S	A	d
stmw [1]	47	S	A	d
stw	36	S	A	d
stwu	37	S	A	d
subfic	08	D	A	SIMM
tdi [2]	02	TO	A	SIMM
twi	03	TO	A	SIMM
xori	26	S	A	UIMM
xoris	27	S	A	UIMM

Note:

1. Load/store string/multiple instruction
2. 64-bit instruction

Table A-35. DS-Form

	OPCD	D	A	ds	XO
	OPCD	S	A	ds	XO
Specific Instructions					
Name	0 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20 21 22 23 24 25 26 27 28 29	30 31
ld [1]	58	D	A	ds	0
ldu [1]	58	D	A	ds	1
lwa [1]	58	D	A	ds	2
std [1]	62	S	A	ds	0
stdu [1]	62	S	A	ds	1

Note:

1. 64-bit instruction

Table A-36. X-Form

	OPCD	D					A					B					XO										0
	OPCD	D					A					NB					XO										0
	OPCD	D					0 0 0 0 0					B					XO										0
	OPCD	D					0 0 0 0 0					0 0 0 0 0					XO										0
	OPCD	D					0	SR				0 0 0 0 0					XO										0
	OPCD	S					A					B					XO										Rc
	OPCD	S					A					B					XO										1
	OPCD	S					A					B					XO										0
	OPCD	S					A					NB					XO										0
	OPCD	S					A					0 0 0 0 0					XO										Rc
	OPCD	S					0 0 0 0 0					B					XO										0
	OPCD	S					0 0 0 0 0					0 0 0 0 0					XO										0
	OPCD	S					0	SR				0 0 0 0 0					XO										0
	OPCD	S					A					SH					XO										Rc
	OPCD	crfD			0	L	A					B					XO										0
	OPCD	crfD			0 0		A					B					XO										0
	OPCD	crfD			0 0		crfS			0 0		0 0 0 0 0					XO										0
	OPCD	crfD			0 0		0 0 0 0 0					0 0 0 0 0					XO										0
	OPCD	crfD			0 0		0 0 0 0 0					IMM				0	XO										Rc
	OPCD	TO					A					B					XO										0
	OPCD	D					0 0 0 0 0					B					XO										Rc
	OPCD	D					0 0 0 0 0					0 0 0 0 0					XO										Rc
	OPCD	crbD					0 0 0 0 0					0 0 0 0 0					XO										Rc
	OPCD	0 0 0 0 0					A					B					XO										0
	OPCD	0 0 0 0 0					0 0 0 0 0					B					XO										0
	OPCD	0 0 0 0 0					0 0 0 0 0					0 0 0 0 0					XO										0
	Specific Instructions																										
Name	0 5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
and*x*	31	S					A					B					28										Rc
andc*x*	31	S					A					B					60										Rc
cmp	31	crfD			0	L	A					B					0										0
cmpl	31	crfD			0	L	A					B					32										0
cntlzd*x* [1]	31	S					A					0 0 0 0 0					58										Rc
cntlzw*x*	31	S					A					0 0 0 0 0					26										Rc
dcbf	31	0 0 0 0 0					A					B					86										0

Note:

1. 64-bit instruction
2. Optional instruction
3. Load/store string/multiple instruction
4. Optional 64-bit bridge instruction

Table A-36. X-Form

dcbst	31	0 0 0 0 0		A	B	54	0
dcbt	31	0 0 0 0 0		A	B	278	0
dcbtst	31	0 0 0 0 0		A	B	246	0
dcbz	31	0 0 0 0 0		A	B	1014	0
eciwx	31	D		A	B	310	0
ecowx	31	S		A	B	438	0
eieio	31	0 0 0 0 0		0 0 0 0 0	0 0 0 0 0	854	0
eqv*x*	31	S		A	B	284	Rc
extsb*x*	31	S		A	0 0 0 0 0	954	Rc
extsh*x*	31	S		A	0 0 0 0 0	922	Rc
extsw*x*[1]	31	S		A	0 0 0 0 0	986	Rc
fabs*x*	63	D		0 0 0 0 0	B	264	Rc
fcfid*x*[1]	63	D		0 0 0 0 0	B	846	Rc
fcmpo	63	crfD	0 0	A	B	32	0
fcmpu	63	crfD	0 0	A	B	0	0
fctid*x*[1]	63	D		0 0 0 0 0	B	814	Rc
fctidz*x*[1]	63	D		0 0 0 0 0	B	815	Rc
fctiw*x*	63	D		0 0 0 0 0	B	14	Rc
fctiwz*x*	63	D		0 0 0 0 0	B	15	Rc
fmr*x*	63	D		0 0 0 0 0	B	72	Rc
fnabs*x*	63	D		0 0 0 0 0	B	136	Rc
fneg*x*	63	D		0 0 0 0 0	B	40	Rc
frsp*x*	63	D		0 0 0 0 0	B	12	Rc
icbi	31	0 0 0 0 0		A	B	982	0
lbzux	31	D		A	B	119	0
lbzx	31	D		A	B	87	0
ldarx[1]	31	D		A	B	84	0
ldux[1]	31	D		A	B	53	0
ldx[1]	31	D		A	B	21	0
lfdux	31	D		A	B	631	0
lfdx	31	D		A	B	599	0
lfsux	31	D		A	B	567	0
lfsx	31	D		A	B	535	0
lhaux	31	D		A	B	375	0
lhax	31	D		A	B	343	0
lhbrx	31	D		A	B	790	0

Note:

1. 64-bit instruction
2. Optional instruction
3. Load/store string/multiple instruction
4. Optional 64-bit bridge instruction

Table A-36. X-Form

lhzux	31	D		A		B		311	0
lhzx	31	D		A		B		279	0
lswi [4]	31	D		A		NB		597	0
lswx [4]	31	D		A		B		533	0
lwarx	31	D		A		B		20	0
lwaux [1]	31	D		A		B		373	0
lwax [1]	31	D		A		B		341	0
lwbrx	31	D		A		B		534	0
lwzux	31	D		A		B		55	0
lwzx	31	D		A		B		23	0
mcrfs	63	crfD	0 0	crfS	0 0	0 0 0 0 0		64	0
mfcr	31	D		0 0 0 0 0		0 0 0 0 0		19	0
mfocrf	31	D		1	CRM		0	19	0
mffs*x*	63	D		0 0 0 0 0		0 0 0 0 0		583	Rc
mfmsr [3]	31	D		0 0 0 0 0		0 0 0 0 0		83	0
mfsr [3]	31	D		0	SR	0 0 0 0 0		595	0
mfsrin [3]	31	D		0 0 0 0 0		B		659	0
mtfsb0*x*	63	crbD		0 0 0 0 0		0 0 0 0 0		70	Rc
mtfsb1*x*	63	crfD		0 0 0 0 0		0 0 0 0 0		38	Rc
mtfsfi*x*	63	crbD	0 0	0 0 0 0 0		IMM	0	134	Rc
mtmsr [3]	31	S		0 0 0 0	L	0 0 0 0 0		146	0
mtmsrd [1, 3]	31	S		0 0 0 0	L	0 0 0 0 0		178	0
mtsr [3]	31	S		0	SR	0 0 0 0 0		210	0
mtsrin [3]	31	S		0 0 0 0 0		B		242	0
nand*x*	31	S		A		B		476	Rc
nor*x*	31	S		A		B		124	Rc
or*x*	31	S		A		B		444	Rc
orc*x*	31	S		A		B		412	Rc
slbia [1, 2, 3]	31	0 0 0 0 0		0 0 0 0 0		0 0 0 0 0		498	0
slbie [1, 2, 3]	31	0 0 0 0 0		0 0 0 0 0		B		434	0
slbmfee [1]	31	D		0 0 0 0 0		B		915	0
slbmfev [1]	31	D		0 0 0 0 0		B		851	0
slbmte [1]	31	D		0 0 0 0 0		B		402	0
sld*x* [1]	31	S		A		B		27	Rc
slw*x*	31	S		A		B		24	Rc
srad*x* [1]	31	S		A		B		794	Rc

Note:

1. 64-bit instruction
2. Optional instruction
3. Load/store string/multiple instruction
4. Optional 64-bit bridge instruction

Table A-36. X-Form

sraw*x*	31	S		A	B	792	Rc
srawi*x*	31	S		A	SH	824	Rc
srd*x* [1]	31	S		A	B	539	Rc
srw*x*	31	S		A	B	536	Rc
stbux	31	S		A	B	247	0
stbx	31	S		A	B	215	0
stdcx. [1]	31	S		A	B	214	1
stdux [1]	31	S		A	B	181	0
stdx [1]	31	S		A	B	149	0
stfdux	31	S		A	B	759	0
stfdx	31	S		A	B	727	0
stfiwx [2]	31	S		A	B	983	0
stfsux	31	S		A	B	695	0
stfsx	31	S		A	B	663	0
sthbrx	31	S		A	B	918	0
sthux	31	S		A	B	439	0
sthx	31	S		A	B	407	0
stswi [4]	31	S		A	NB	725	0
stswx [4]	31	S		A	B	661	0
stwbrx	31	S		A	B	662	0
stwcx.	31	S		A	B	150	1
stwux	31	S		A	B	183	0
stwx	31	S		A	B	151	0
sync	31	0 0 0	L	0 0 0 0 0	0 0 0 0 0	598	0
td [1]	31	TO		A	B	68	0
tlbia [2, 3]	31	0 0 0 0 0		0 0 0 0 0	0 0 0 0 0	370	0
tlbie [2, 3]	31	0 0 0 0	L	0 0 0 0 0	B	306	0
tlbiel [2, 3]	31	0 0 0 0	L	0 0 0 0 0	B	274	0
tlbsync [2, 3]	31	0 0 0 0 0		0 0 0 0 0	0 0 0 0 0	566	0
tw	31	TO		A	B	4	0
xor*x*	31	S		A	B	316	Rc

Note:
1. 64-bit instruction
2. Optional instruction
3. Load/store string/multiple instruction
4. Optional 64-bit bridge instruction

Table A-37. XL-Form

Name	0 … 5	6 7 8	9 10	11 12 13	14 15	16 17 18	19 20	21 … 30	31
	OPCD	BO		BI		0 0 0 0 0		XO	LK
	OPCD	crbD		crbA		crbB		XO	0
	OPCD	crfD	0 0	crfS	0 0	0 0 0 0 0		XO	0
	OPCD	0 0 0 0 0		0 0 0 0 0		0 0 0 0 0		XO	0
Specific Instructions									
bcctr*x*	19	BO		BI		0 0 0	BH	528	LK
bclr*x*	19	BO		BI		0 0 0	BH	16	LK
crand	19	crbD		crbA		crbB		257	0
crandc	19	crbD		crbA		crbB		129	0
creqv	19	crbD		crbA		crbB		289	0
crnand	19	crbD		crbA		crbB		225	0
crnor	19	crbD		crbA		crbB		33	0
cror	19	crbD		crbA		crbB		449	0
crorc	19	crbD		crbA		crbB		417	0
crxor	19	crbD		crbA		crbB		193	0
isync	19	0 0 0 0 0		0 0 0 0 0		0 0 0 0 0		150	0
mcrf	19	crfD	0 0	crfS	0 0	0 0 0 0 0		0	0
rfid [1, 2]	19	0 0 0 0 0		0 0 0 0 0		0 0 0 0 0		18	0

Note:
1. Supervisor-level instruction
2. 64-bit instruction

Table A-38. XFX-Form

	OPCD	D					spr										XO										0
	OPCD	D					0	CRM								0	XO										0
	OPCD	S					spr										XO										0
	OPCD	D					tbr										XO										0
	Specific Instructions																										
Name	0 5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
mfspr [1]	31	D					spr										339										0
mftb	31	D					tbr										371										0
mtcrf	31	S					0	CRM								0	144										0
mtocrf	31	S					1	CRM								0	144										0
mtspr [1]	31	D					spr										467										0

Note:

1. Supervisor and user-level instruction

Table A-39. XFL-Form

	OPCD	0	FM								0	B					XO										Rc
	Specific Instructions																										
Name	0 5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
mtfsf*x*	63	0	FM								0	B					711										Rc

Table A-40. XS-Form

	OPCD	S					A					sh					XO									sh	Rc
	Specific Instructions																										
Name	0 5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
sradi*x* [1]	31	S					A					sh					413									sh	Rc

Note:

1. 64-bit instruction

Table A-41. XO-Form

	OPCD	D	A	B	OE	XO	Rc
	OPCD	D	A	B	0	XO	Rc
	OPCD	D	A	0 0 0 0 0	OE	XO	Rc
Specific Instructions							
Name	0 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21	22 23 24 25 26 27 28 29 30	31
add*x*	31	D	A	B	OE	266	Rc
addc*x*	31	D	A	B	OE	10	Rc
adde*x*	31	D	A	B	OE	138	Rc
addme*x*	31	D	A	0 0 0 0 0	OE	234	Rc
addze*x*	31	D	A	0 0 0 0 0	OE	202	Rc
divd*x* [1]	31	D	A	B	OE	489	Rc
divdu*x* [1]	31	D	A	B	OE	457	Rc
divw*x*	31	D	A	B	OE	491	Rc
divwu*x*	31	D	A	B	OE	459	Rc
mulhd*x* [1]	31	D	A	B	0	73	Rc
mulhdu*x* [1]	31	D	A	B	0	9	Rc
mulhw*x*	31	D	A	B	0	75	Rc
mulhwu*x*	31	D	A	B	0	11	Rc
mulld*x* [1]	31	D	A	B	OE	233	Rc
mullw*x*	31	D	A	B	OE	235	Rc
neg*x*	31	D	A	0 0 0 0 0	OE	104	Rc
subf*x*	31	D	A	B	OE	40	Rc
subfc*x*	31	D	A	B	OE	8	Rc
subfe*x*	31	D	A	B	OE	136	Rc
subfme*x*	31	D	A	0 0 0 0 0	OE	232	Rc
subfze*x*	31	D	A	0 0 0 0 0	OE	200	Rc

Note:

1. 64-bit instruction

Table A-42. A-Form

	OPCD	D	A	B	0 0 0 0 0	XO	Rc
	OPCD	D	A	B	C	XO	Rc
	OPCD	D	A	0 0 0 0 0	C	XO	Rc
	OPCD	D	0 0 0 0 0	B	0 0 0 0 0	XO	Rc
Specific Instructions							
Name	0 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25	26 27 28 29 30	31
fadd*x*	63	D	A	B	0 0 0 0 0	21	Rc
fadds*x*	59	D	A	B	0 0 0 0 0	21	Rc
fdiv*x*	63	D	A	B	0 0 0 0 0	18	Rc
fdivs*x*	59	D	A	B	0 0 0 0 0	18	Rc
fmadd*x*	63	D	A	B	C	29	Rc
fmadds*x*	59	D	A	B	C	29	Rc
fmsub*x*	63	D	A	B	C	28	Rc
fmsubs*x*	59	D	A	B	C	28	Rc
fmul*x*	63	D	A	0 0 0 0 0	C	25	Rc
fmuls*x*	59	D	A	0 0 0 0 0	C	25	Rc
fnmadd*x*	63	D	A	B	C	31	Rc
fnmadds*x*	59	D	A	B	C	31	Rc
fnmsub*x*	63	D	A	B	C	30	Rc
fnmsubs*x*	59	D	A	B	C	30	Rc
fresx [1]	59	D	0 0 0 0 0	B	0 0 0 0 0	24	Rc
frsqrte*x* [1]	63	D	0 0 0 0 0	B	0 0 0 0 0	26	Rc
fsel*x* [1]	63	D	A	B	C	23	Rc
fsqrt*x* [1]	63	D	0 0 0 0 0	B	0 0 0 0 0	22	Rc
fsqrts*x* [1]	59	D	0 0 0 0 0	B	0 0 0 0 0	22	Rc
fsub*x*	63	D	A	B	0 0 0 0 0	20	Rc
fsubs*x*	59	D	A	B	0 0 0 0 0	20	Rc

Note:

1. Optional instruction

Table A-43. M-Form

	OPCD	S					A					SH					MB					ME					Rc
	OPCD	S					A					B					MB					ME					Rc
Specific Instructions																											
Name	0 5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
rlwimi*x*	20	S					A					SH					MB					ME					Rc
rlwinm*x*	21	S					A					SH					MB					ME					Rc
rlwnm*x*	23	S					A					B					MB					ME					Rc

Table A-44. MD-Form

	OPCD	S					A					sh					mb						XO			sh	Rc
	OPCD	S					A					sh					me						XO			sh	Rc
Specific Instructions																											
Name	0 5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
rldic*x* [1]	30	S					A					sh					mb						2			sh	Rc
rldicl*x* [1]	30	S					A					sh					mb						0			sh	Rc
rldicr*x* [1]	30	S					A					sh					me						1			sh	Rc
rldimi*x* [1]	30	S					A					sh					mb						3			sh	Rc

Note:
1. 64-bit instruction

Table A-45. MDS-Form

	OPCD	S					A					B					mb						XO				Rc
	OPCD	S					A					B					me						XO				Rc
Specific Instructions																											
Name	0 5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
rldclx [1]	30	S					A					B					mb						8				Rc
rldcrx [1]	30	S					A					B					me						9				Rc

Note:
1. 64-bit instruction

A.5 Instruction Set Legend

Table A-46 provides general information on the PowerPC instruction set (such as the architectural level, privilege level, and form).

Table A-46. PowerPC Instruction Set Legend

Instruction	UISA	VEA	OEA	Supervisor Level	64-Bit Only	64-Bit Bridge	Optional	Form
add*x*	Yes							XO
addc*x*	Yes							XO
adde*x*	Yes							XO
addi	Yes							D
addic	Yes							D
addic.	Yes							D
addis	Yes							D
addme*x*	Yes							XO
addze*x*	Yes							XO
and*x*	Yes							X
andc*x*	Yes							X
andi.	Yes							D
andis.	Yes							D
b*x*	Yes							I
bc*x*	Yes							B
bcctr*x*	Yes							XL
bclr*x*	Yes							XL
cmp	Yes							X
cmpi	Yes							D
cmpl	Yes							X
cmpli	Yes							D
cntlzd*x*	Yes				Yes			X
cntlzw*x*	Yes							X
crand	Yes							XL
crandc	Yes							XL
creqv	Yes							XL
crnand	Yes							XL
crnor	Yes							XL
cror	Yes							XL
crorc	Yes							XL
crxor	Yes							XL
dcbf		Yes						X

Note:

1. Supervisor and user-level instruction
2. Load/store string or multiple instruction
3. New or newly optional in the PowerPC Architecture Specification 2.01.

Table A-46. PowerPC Instruction Set Legend (Continued)

Instruction	UISA	VEA	OEA	Supervisor Level	64-Bit Only	64-Bit Bridge	Optional	Form
dcbst		Yes						X
dcbt		Yes						X
dcbtst		Yes						X
dcbz		Yes						X
divd*x*	Yes				Yes			XO
divdu*x*	Yes				Yes			XO
divw*x*	Yes							XO
divwu*x*	Yes							XO
eciwx		Yes					Yes	X
ecowx		Yes					Yes	X
eieio		Yes						X
eqv*x*	Yes							X
extsb*x*	Yes							X
extsh*x*	Yes							X
extsw*x*	Yes				Yes			X
fabs*x*	Yes							X
fadd*x*	Yes							A
fadds*x*	Yes							A
fcfid*x*	Yes				Yes			X
fcmpo	Yes							X
fcmpu	Yes							X
fctid*x*	Yes				Yes			X
fctidz*x*	Yes				Yes			X
fctiw*x*	Yes							X
fctiwz*x*	Yes						Yes	X
fdiv*x*	Yes							A
fdivs*x*	Yes							A
fmadd*x*	Yes							A
fmadds*x*	Yes							A
fmr*x*	Yes							X
fmsub*x*	Yes							A
fmsubs*x*	Yes							A
fmul*x*	Yes							A
fmuls*x*	Yes							A
fnabs*x*	Yes							X
fneg*x*	Yes							X

Note:

1. Supervisor and user-level instruction
2. Load/store string or multiple instruction
3. New or newly optional in the PowerPC Architecture Specification 2.01.

Table A-46. PowerPC Instruction Set Legend (Continued)

Instruction	UISA	VEA	OEA	Supervisor Level	64-Bit Only	64-Bit Bridge	Optional	Form
fnmadd*x*	Yes							A
fnmadds*x*	Yes							A
fnmsub*x*	Yes							A
fnmsubs*x*	Yes							A
fres*x*	Yes						Yes	A
frsp*x*	Yes							X
frsqrte*x*	Yes						Yes	A
fsel*x*	Yes						Yes	A
fsqrt*x*	Yes						Yes	A
fsqrts*x*	Yes						Yes	A
fsub*x*	Yes							A
fsubs*x*	Yes							A
icbi		Yes						X
isync		Yes						XL
lbz	Yes							D
lbzu	Yes							D
lbzux	Yes							X
lbzx	Yes							X
ld	Yes				Yes			DS
ldarx	Yes				Yes			X
ldu	Yes				Yes			DS
ldux	Yes				Yes			X
ldx	Yes				Yes			X
lfd	Yes							D
lfdu	Yes							D
lfdux	Yes							X
lfdx	Yes							X
lfs	Yes							D
lfsu	Yes							D
lfsux	Yes							X
lfsx	Yes							X
lha	Yes							D
lhau	Yes							D
lhaux	Yes							X
lhax	Yes							X
lhbrx	Yes							X

Note:

1. Supervisor and user-level instruction
2. Load/store string or multiple instruction
3. New or newly optional in the PowerPC Architecture Specification 2.01.

Table A-46. PowerPC Instruction Set Legend (Continued)

Instruction	UISA	VEA	OEA	Supervisor Level	64-Bit Only	64-Bit Bridge	Optional	Form
lhz	Yes							D
lhzu	Yes							D
lhzux	Yes							X
lhzx	Yes							X
lmw [1]	Yes							D
lswi [1]	Yes							X
lswx [1]	Yes							X
lwa	Yes				Yes			DS
lwarx	Yes							X
lwaux	Yes				Yes			X
lwax	Yes				Yes			X
lwbrx	Yes							X
lwz	Yes							D
lwzu	Yes							D
lwzux	Yes							X
lwzx	Yes							X
mcrf	Yes							XL
mcrfs	Yes							X
mfcr	Yes							X
mfocrf [3]								
mffs	Yes							X
mfmsr			Yes	Yes				X
mfspr [1]	Yes		Yes	Yes				XFX
mfsr			Yes	Yes		Yes	Yes	X
mfsrin			Yes	Yes		Yes	Yes	X
mftb		Yes						XFX
mtcrf	Yes							XFX
mtocrf [3]								
mtfsb0*x*	Yes							X
mtfsb1*x*	Yes							X
mtfsf*x*	Yes							XFL
mtfsfi*x*	Yes							X
mtmsr			Yes	Yes		Yes	Yes	X
mtmsrd			Yes	Yes	Yes			X
mtspr [1]	Yes		Yes	Yes				XFX
mtsr			Yes	Yes		Yes	Yes	X

Note:

1. Supervisor and user-level instruction
2. Load/store string or multiple instruction
3. New or newly optional in the PowerPC Architecture Specification 2.01.

Table A-46. PowerPC Instruction Set Legend (Continued)

Instruction	UISA	VEA	OEA	Supervisor Level	64-Bit Only	64-Bit Bridge	Optional	Form
mtsrin			Yes	Yes		Yes	Yes	X
mulhd*x*	Yes				Yes			XO
mulhdu*x*	Yes				Yes			XO
mulhw*x*	Yes							XO
mulhwu*x*	Yes							XO
mulld*x*	Yes				Yes			XO
mulli	Yes							D
mullw*x*	Yes							XO
nand*x*	Yes							X
neg*x*	Yes							XO
nor*x*	Yes							X
or*x*	Yes							X
orc*x*	Yes							X
ori	Yes							D
oris	Yes							D
rfid			Yes	Yes	Yes	Yes		XL
rldcl*x*	Yes				Yes			MDS
rldcr*x*	Yes				Yes			MDS
rldic*x*	Yes				Yes			MD
rldicl*x*	Yes				Yes			MD
rldicr*x*	Yes				Yes			MD
rldimi*x*	Yes				Yes			MD
rlwimi*x*	Yes							M
rlwinm*x*	Yes							M
rlwnm*x*	Yes							M
sc	Yes		Yes					SC
slbia			Yes	Yes	Yes		Yes	X
slbie			Yes	Yes	Yes		Yes	X
slbmfee [3]			Yes	Yes	Yes			
slbmfev [3]			Yes	Yes	Yes			
slbmte [3]			Yes	Yes	Yes			
sld*x*	Yes				Yes			X
slw*x*	Yes							X
srad*x*	Yes				Yes			X
sradi*x*	Yes				Yes			XS
sraw*x*	Yes							X

Note:
1. Supervisor and user-level instruction
2. Load/store string or multiple instruction
3. New or newly optional in the PowerPC Architecture Specification 2.01.

Table A-46. PowerPC Instruction Set Legend (Continued)

Instruction	UISA	VEA	OEA	Supervisor Level	64-Bit Only	64-Bit Bridge	Optional	Form
srawi*x*	Yes							X
srd*x*	Yes				Yes			X
srw*x*	Yes							X
stb	Yes							D
stbu	Yes							D
stbux	Yes							X
stbx	Yes							X
std	Yes				Yes			DS
stdcx.	Yes				Yes			X
stdu	Yes				Yes			DS
stdux	Yes				Yes			X
stdx	Yes				Yes			X
stfd	Yes							D
stfdu	Yes							D
stfdux	Yes							X
stfdx	Yes							X
stfiwx	Yes							X
stfs	Yes							D
stfsu	Yes							D
stfsux	Yes							X
stfsx	Yes							X
sth	Yes							D
sthbrx	Yes							X
sthu	Yes							D
sthux	Yes							X
sthx	Yes							X
stmw [2]	Yes							D
stswi [2]	Yes							X
stswx [2]	Yes							X
stw	Yes							D
stwbrx	Yes							X
stwcx.	Yes							X
stwu	Yes							D
stwux	Yes							X
stwx	Yes							X
subf*x*	Yes							XO

Note:

1. Supervisor and user-level instruction
2. Load/store string or multiple instruction
3. New or newly optional in the PowerPC Architecture Specification 2.01.

Table A-46. PowerPC Instruction Set Legend (Continued)

Instruction	UISA	VEA	OEA	Supervisor Level	64-Bit Only	64-Bit Bridge	Optional	Form
subfc*x*	Yes							XO
subfe*x*	Yes							XO
subfic	Yes							D
subfme*x*	Yes							XO
subfze*x*	Yes							XO
sync	Yes							X
td	Yes				Yes			X
tdi	Yes				Yes			D
tlbia*x*			Yes	Yes			Yes	X
tlbie*x*			Yes	Yes			Yes	X
tlbiel [3]			Yes	Yes			Yes	X
tlbsync			Yes	Yes			Yes	X
tw	Yes							X
twi	Yes							D
xor*x*	Yes							X
xori	Yes							D
xoris	Yes							D

Note:

1. Supervisor and user-level instruction
2. Load/store string or multiple instruction
3. New or newly optional in the PowerPC Architecture Specification 2.01.

THIS PAGE INTENTIONALLY LEFT BLANK

Appendix B. Multiple-Precision Shifts

This appendix gives examples of how multiple precision shifts can be programmed. A multiple-precision shift is initially defined to be a shift of an *n*-double word quantity (64-bit mode) or an *n*-word quantity (32-bit mode), where $n > 1$. The quantity to be shifted is contained in *n* registers (in the low-order 32 bits in 32-bit mode). The shift amount is specified either by an immediate value in the instruction or by bits [57–63] (64-bit mode) or [58-63] (32-bit mode) of 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 0–127 (64-bit mode), or 0–63 (32-bit mode), which are the maximum ranges supported by the shift instructions used. However if $n > 2$, the shift amount must be in the range 0–63 (64-bit mode), or 0–31 (32-bit mode), for the examples to yield the desired result. The specific instance shown for $n > 2$ is $n = 3$: extending those instruction sequences to larger *n* is straightforward, as is reducing them to the case $n = 2$ when the more stringent restriction on shift amount is met. For shifts with immediate shift amounts, only the case $n = 3$ is shown because the more stringent restriction on shift amount is always met.

In the examples it is assumed that GPRs 2 and 3 (and 4) contain the quantity to be shifted, and that the result is to be placed into the same registers, except for the immediate left shifts in 64-bit mode for which the result is placed into GPRs 3, 4, and 5. In all cases, for both input and result, the lowest-numbered register contains the highest-order part of the data and highest-numbered register contains the lowest-order part. In 32-bit mode, the high-order 32 bits of these registers are assumed not to be part of the quantity to be shifted nor of the result. For non-immediate shifts, the shift amount is assumed to be in bits [57–63] (64-bit mode), or [58-63] (32-bit mode), of GPR6. For immediate shifts, the shift amount is assumed to be greater than zero. GPRs 0-31 are used as scratch registers. For $n > 2$, the number of instructions required is $2n - 1$ (immediate shifts) or $3n - 1$ (non-immediate shifts).

The following section provide an example of multiple-precision shifts in 64-bit mode.

B.1 Multiple-Precision Shifts

Table B-1. Multiple-Precision Shifts (64-bit Mode vs. 32-bit Mode)

64-bit Mode		32-bit Mode	
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
Shift Right Immediate, *n* = 3 (Shift Amount < 64)		Shift Right Immediate, *n* = 3 (Shift Amount < 32)	
rldimi	r4,r3,0,64 – sh	**rlwinm**	r4,r4,32 – sh,sh,31
rldicl	r4,r4,64 – sh,0	**rlwimi**	r4,r3,32 – sh,0,sh – 1
rldimi	r3,r2,0,64 – sh	**rlwinm**	r3,r3,32 – sh,sh,31
rldicl	r3,r3,64 – sh,0	**rlwimi**	r3,r2,32 – sh,0,sh – 1
rldicl	r2,r2,64 – sh,sh	**rlwinm**	r2,r2,32 – sh,sh,31
Shift Right, *n* = 2 (Shift Amount < 128)		Shift Right, *n* = 2 (Shift Amount < 64)	
subfic	r31,r6,64	**subfic**	r31,r6,32
srd	r3,r3,r6	**srw**	r3,r3,r6
sld	r0,r2,r31	**slw**	r0,r2,r31
or	r3,r3,r0	**or**	r3,r3,r0
addi	r31,r6,–64	**addi**	r31,r6, –32
srd	r0,r2,r31	**srw**	r0,r2,r31
or	r3,r3,r0	**or**	r3,r3,r0
srd	r2,r2,r6	**srw**	r2,r2,r6

Table B-1. Multiple-Precision Shifts (64-bit Mode vs. 32-bit Mode)

64-bit Mode		32-bit Mode	
Shift Right, *n* = 3 (Shift Amount < 64)		Shift Right, *n* = 3 (Shift Amount < 32)	
subfic	r31,r6,64	**subfic**	r31,r6,–32
srd	r4,r4,r6	**srw**	r4,r4,r6
sld	r0,r3,r31	**slw**	r0,r3,r31
or	r4,r4,r0	**or**	r4,r4,r0
srd	r3,r3,r6	**srw**	r3,r3,r6
sld	r0,r2,r31	**slw**	r0,r2,r31
or	r3,r3,r0	**or**	r3,r3,r0
srd	r2,r2,r6	**srw**	r2,r2,r6
Shift Right Algebraic Immediate, *n* = 3 (Shift Amount < 64)		Shift Right Algebraic Immediate, *n* = 3 (Shift Amount < 32)	
rldimi	r4,r4,0,64 – sh	**rlwinm**	r4,r4,32 – sh,sh,31
rldicl	r4,r4,64 – sh,0	**rlwimi**	r4,r3,32 – sh,0,sh – 1
rldimi	r3,r2,0,64 – sh	**rlwinm**	r3,r3,32 – sh,sh,31
rldicl	r3,r3,64 – sh,0	**rlwimi**	r3,r2,32 – sh,0,sh – 1
sradi	r2,r2,sh	**srawi**	r2,r2,sh
Shift Right Algebraic, *n* = 2 (Shift Amount < 128)		Shift Right Algebraic, *n* = 2 (Shift Amount < 64)	
subfic	r31,r6,64	**subfic**	r31,r6,32
srd	r3,r3,r6	**srw**	r3,r3,r6
sld	r0,r2,r31	**slw**	r0,r2,r31
or	r3,r3,r0	**or**	r3,r3,r0
addic.	r31,r6,–64	**addic.**	r31,r6,–32
srad	r0,r2,r31	**sraw**	r0,r2,r31
ble	$+8	**ble**	$+8
ori	r3,r0,0	**ori**	r3,r0,0
srad	r2,r2,r6	**sraw**	r2,r2,r6
Shift Right Algebraic, *n* = 3 (Shift Amount < 64)		Shift Right Algebraic, *n* = 3 (Shift Amount < 32)	
subfic	r31,r6,64	**subfic**	r31,r6,32
srd	r4,r4,r6	**srw**	r4,r4,r6
sld	r0,r3,r31	**slw**	r0,r3,r31
or	r4,r4,r0	**or**	r4,r4,r0
srd	r3,r3,r6	**srw**	r3,r3,r6
sld	r0,r2,r31	**slw**	r0,r2,r31
or	r3,r3,r0	**or**	r3,r3,r0
srad	r2,r2,r6	**sraw**	r2,r2,r6

THIS PAGE INTENTIONALLY LEFT BLANK

Appendix C. Floating-Point Models

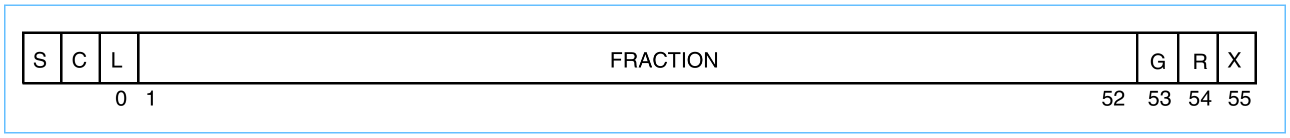
This appendix describes the execution model for IEEE operations and gives examples of how the floating-point conversion instructions can be used to perform various conversions as well as providing models for floating-point instructions.

C.1 Execution Model for IEEE Operations

The following description uses double-precision arithmetic as an example; single-precision arithmetic is similar except that the fraction field 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 performed with a floating-point accumulator where bits [0–55], shown in *Figure C-1*, comprise the significand of the intermediate result.

Figure C-1. IEEE 64-Bit Execution Model



The bits and fields for the IEEE double-precision execution model are defined as follows:

- The S bit is the sign bit.
- The C bit is the carry bit that captures the carry out of the significand.
- The L bit is the leading unit bit of the significand that receives the implicit bit from the operands.
- The FRACTION is a 52-bit field that accepts the fraction of the operands.
- The guard (G), round (R), and sticky (X) bits are extensions to the low-order bits of the accumulator. The G and R bits are required for postnormalization of the result. The G, R, and X bits are required during rounding to determine if the intermediate result is equally near the two nearest representable values. The X bit serves as an extension to the G and R bits by representing the logical OR of all bits that may appear to the low-order side of the R bit, due to either shifting the accumulator right or to other generation of low-order result bits. The G and R bits participate in the left shifts with zeros being shifted into the R bit.

Table C-1 shows the significance of the G, R, and X bits with respect to the intermediate result (IR), the next lower in magnitude representable number (NL), and the next higher in magnitude representable number (NH).

Table C-1. 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 & NH

Table C-1. Interpretation of G, R, and X Bits (Continued)

G	R	X	Interpretation
1	0	1	IR closer to NH
1	1	0	
1	1	1	

The significand of the intermediate result is made up of the L bit, the FRACTION, and the G, R, and X bits.

The infinitely precise intermediate result of an operation is the result normalized in bits L, FRACTION, G, R, and X of the floating-point accumulator.

After normalization, the intermediate result is rounded, using the rounding mode specified by FPSCR[RN]. If rounding causes a carry into C, the significand is shifted right one position and the exponent is incremented by one. This causes an inexact result and possibly exponent overflow. Fraction bits to the left of the bit position used for rounding are stored into the FPR, 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 3.3.5 Rounding*. For rounding, the conceptual guard, round, and sticky bits are defined in terms of accumulator bits.

Table C-2 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 C-2. Location of the Guard, Round, and Sticky Bits—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 are nonzero, the result is inexact.

Z1 and Z2, defined in *Section 3.3.5 Rounding*, 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 one (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 zero (result midway between closest representable values) then if the low-order bit of the result is one, the result is incremented. Otherwise (the low-order bit of the result is zero) the result is truncated (this is the case of a tie rounded to even).

  If during the round-to-nearest process, truncation of the unrounded number produces the maximum magnitude for the specified precision, the following action is taken:

  - Guard bit = ‘1’: Store infinity with the sign of the unrounded result.
  - Guard bit = ‘0’: Store the truncated (maximum magnitude) value.

- 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.

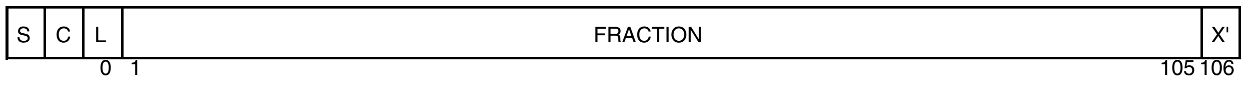
Where the result is to have fewer than 53 bits of precision because the instruction is a floating round to single-precision or single-precision arithmetic instruction, the intermediate result either is normalized or is placed in correct denormalized form before being rounded.

C.2 Execution Model for Multiply-Add Type Instructions

The PowerPC Architecture makes use of a special instruction form that performs up to three operations in one instruction (a multiply, an add, and a negate). With this added capability comes the special ability to produce a more exact intermediate result as an input to the rounder. Single-precision arithmetic is similar except that the fraction field is smaller. Note that the rounding occurs only after add; therefore, the computation of the sum and product together are infinitely precise before the final result is rounded to a representable format.

The multiply-add significand arithmetic is considered to be performed with a floating-point accumulator, where bits [1–106] comprise the significand of the intermediate result. The format is shown in *Figure C-2*.

Figure C-2. Multiply-Add 64-Bit Execution Model



The first part of the operation is a multiply. The multiply has two 53-bit significands as inputs, which are assumed to be prenormalized, and produces a result conforming to the above model. If there is a carry out of the significand (into the C bit), the significand is shifted right one position, placing the L bit into the most-significant bit of the FRACTION and placing the C bit into the L bit. All 106 bits (L bit plus 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 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 add is then normalized, with all bits of the add 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 C-3* 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 C-3. Location of the Guard, Round, and Sticky Bits—Multiply-Add Execution Model

Format	Guard	Round	Sticky
Double	53	54	OR of [55–105], X'
Single	24	25	OR of [26–10]5, X'

The rules for rounding the intermediate result are the same as those given in *Appendix C.1 Execution Model for IEEE Operations*.

If the instruction is floating negative multiply-add or floating negative multiply-subtract, the final result is negated.

Floating-point multiply-add instructions combine a multiply and an add operation without an intermediate rounding operation. The fraction part of the intermediate product is 106 bits wide, 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 (for example, an **fmul** instruction followed by an **fadd** instruction). That is, multiplication of infinity by 0 or of anything by an SNaN, causes the corresponding exception bits to be set.

C.3 Floating-Point Conversions

This section provides examples of floating-point conversion instructions. Note that some of the examples use the optional Floating Select (**fsel)** instruction. Care must be taken in using **fsel** if IEEE compatibility is required, or if the values being tested can be NaNs or infinities.

C.3.1 Conversion from Floating-Point Number to Floating-Point Integer

The full convert to floating-point integer function can be implemented with the following sequence assuming the floating-point value to be converted is in FPR1, and the result is returned in FPR3.

```
mtfsb0    23           #clear VXCVI
fctid[z]  f3,f1        #convert to fx int
fcfid     f3,f3        #convert back again
mcrfs     7,5          #VXCVI to CR
bf        31,$+8       #skip if VXCVI was 0
fmr       f3,f1        #input was fp int
```

C.3.2 Conversion from Floating-Point Number to Signed Fixed-Point Integer Double Word

The full convert to signed fixed-point integer double word function can be implemented with the following sequence, assuming the floating-point value to be converted is in FPR1, the result is returned in GPR3, and a double word at displacement (disp) from the address in GPR1 can be used as scratch space.

```
fctid[z]  f2,f1             #convert to dword int
stfd      f2,disp(r1)       #store float
ld        r3,disp(r1)       #load dword
```

C.3.3 Conversion from Floating-Point Number to Unsigned Fixed-Point Integer Double Word

The full convert to unsigned fixed-point integer double word function can be implemented with the following sequence, assuming the floating-point value to be converted is in FPR1, the value zero is in FPR0, the value $2^{64} - 2048$ is in FPR3, the value 2^{63} is in FPR4 and GPR4, the result is returned in GPR3, and a double word at displacement (disp) from the address in GPR1 can be used as scratch space.

```
fsel       f2,f1,f1,f0         #use 0 if < 0
fsub       f5,f3,f1            #use max if > max
fsel       f2,f5,f2,f3
fsub       f5,f2,f4            #subtract 2**63
fcmpu      cr2,f2,f4           #use diff if 2**63
fsel       f2,f5,f5,f2
fctid[z]   f2,f2               #convert to fx int
stfd       f2,disp(r1)         #store float
ld         r3,disp(r1)         #load dword
blt        cr2,$+8             #add 2**63 if input
add        r3,r3,r4            #was 2**63
```

C.3.4 Conversion from Floating-Point Number to Signed Fixed-Point Integer Word

The full convert to signed fixed-point integer word function can be implemented with the following sequence, assuming that the floating-point value to be converted is in FPR1, the result is returned in GPR3, and a double word at displacement (disp) from the address in GPR1 can be used as scratch space.

```
fctiw[z]   f2,f1               #convert to fx int
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 Fixed-Point Integer Word

The full convert to unsigned fixed-point integer word function can be implemented with the following sequence, assuming the floating-point value to be converted is in FPR1, the value zero is in FPR0, the value $2^{32} - 1$ is in FPR3, the result is returned in GPR3, and a double word at displacement (disp) from the address in GPR1 can be used as scratch space.

```
fsel      f2,f1,f1,f0         #use 0 if < 0
fsub      f4,f3,f1            #use max if > max
fsel      f2,f4,f2,f3
fctid[z]  f2,f2               #convert to fx int
stfd      f2,disp(r1)         #store float
lwz       r3,disp + 4(r1)     #load word and zero
```

C.3.6 Conversion from Signed Fixed-Point Integer Double Word to Floating-Point Number

The full convert from signed fixed-point integer double word function, using the rounding mode specified by FPSCR[RN], can be implemented with the following sequence, assuming the fixed-point value to be converted is in GPR3, the result is returned in FPR1, and a double word at displacement (disp) from the address in GPR1 can be used as scratch space.

```
std       r3,disp(r1)         #store dword
lfd       f1,disp(r1)         #load float
fcfid     f1,f1               #convert to fpu int
```

C.3.7 Conversion from Unsigned Fixed-Point Integer Double Word to Floating-Point Number

The full convert from unsigned fixed point integer double word function, using the rounding mode specified by FPSCR[RN], can be implemented with the following sequence, assuming the fixed-point value to be converted is in GPR3, the value 2^{32} is in FPR4, the result is returned in FPR1, and two double words at displacement (disp) from the address in GPR1 is used as scratch space.

```
rldicl    r2,r3,32,32         #isolate high half
rldicl    r0,r3,0,32          #isolate low half
std       r2,disp(r1)         #store dword both
std       r0,disp + 8(r1)
lfd       f2,disp(r1)         #load float both
lfd       f1,disp + 8(r1)     #load float both
fcfid     f2,f2               #convert each half to
fcfid     f1,f1               #fpu int (no rnd)
fmadd     f1,f4,f2,f1         #(2**32)*high+low
                              (only add can rnd)
```

An alternative, shorter, sequence can be used if rounding according to FPSCR[RN] is desired and FPSCR[RN] specifies round toward +infinity or round toward –infinity, or if it is acceptable for the rounded answer to be either of the two representable floating-point integers nearest to the given fixed-point integer. In this case the full convert from unsigned fixed-point integer double word function can be implemented with the following sequence, assuming the value 2^{64} is in FPR2.

```
std       r3,disp(r1)         #store dword
lfd       f1,disp(r1)         #load float
fcfid     f1,f1               #convert to fpu int
fadd      f4,f1,f2            #add 2**64
fsel      f1,f1,f1,f4         #if r3 < 0
```

C.3.8 Conversion from Signed Fixed-Point Integer Word to Floating-Point Number

The full convert from signed fixed-point integer word function can be implemented with the following sequence, assuming the fixed-point value to be converted is in GPR3, the result is returned in FPR1, and a double word at displacement (disp) from the address in GPR1 can be used as scratch space. (The result is exact.)

```
extsw     r3,r3              #extend sign
std       r3,disp(r1)        #store dword
lfd       f1,disp(r1)        #load float
fcfid     f1,f1              #convert to fpu int
```

C.3.9 Conversion from Unsigned Fixed-Point Integer Word to Floating-Point Number

The full convert from unsigned fixed-point integer word function can be implemented with the following sequence, assuming the fixed-point value to be converted is in GPR3, the result is returned in FPR1, and a double word at displacement (disp) from the address in GPR1 can be used as scratch space. (The result is exact.)

```
rldicl    r0,r3,0,32         #zero-extend
std       r0,disp(r1)        #store dword
lfd       f1,disp(r1)        #load float
fcfid     f1,f1              #convert to fpu int
```

C.4 Floating-Point Models

This section describes models for floating-point instructions.

C.4.1 Floating-Point Round to Single-Precision Model

The following algorithm describes the operation of the Floating Round to Single-Precision (**frsp**) instruction.

```
If frB[1–11] < 897 and frB[1-63] > 0 then
          Do
                  If FPSCR[UE] = 0 then goto Disabled Exponent Underflow
                  If FPSCR[UE] = 1 then goto Enabled Exponent Underflow
          End

If frB[1-11] > 1150 and frB[1-11] < 2047 then
          Do
          If FPSCR[OE] = 0 then goto Disabled Exponent Overflow
          If FPSCR[OE] = 1 then goto Enabled Exponent Overflow
          End

If frB[1-11] > 896 and frB[1-11] < 1151 then goto Normal Operand

If frB[1-63] = 0 then goto Zero Operand

If frB[1-11] = 2047 then
          Do
          If frB[12-63] = 0 then goto Infinity Operand
          If frB[12] = 1 then goto QNaN Operand
          If frB[12] = 0 and frB[13-63] > 0 then goto SNaN Operand
          End
```

Disabled Exponent Underflow

```
sign ← frB[0]
If frB[1-11] = 0 then
            Do
            exp ← -1022
            frac[0-52] ← 0b0 || frB[12-63]
            End
If frB[1-11] > 0 then
            Do
            exp ← frB[1-11] - 1023
            frac[0-52] ← 0b1 || frB[12-63]
            End
Denormalize operand:
            G || R || X ← 0b000
            Do while exp < -126
            exp ← exp + 1
            frac[0-52] || G || R || X ← 0b0 || frac || G || (R | X)
            End
FPSCR[UX] ← frac[24-52] || G || R || X > 0
Round single(sign,exp,frac[0-52],G,R,X)
FPSCR[XX] ← FPSCR[XX] | FPSCR[FI]
If frac[0-52] = 0 then
            Do
            frD[0] ← sign
            frD[1-63] ← 0
            If sign = 0 then FPSCR[FPRF] ← +zero
            If sign = 1 then FPSCR[FPRF] ← -zero
            End
If frac[0-52] > 0 then
            Do
            If frac[0] = 1 then
                     Do
                           If sign = 0 then FPSCR[FPRF] ← +normal number
                           If sign = 1 then FPSCR[FPRF] ← -normal number
                     End
            If frac[0] = 0 then
                     Do
                           If sign = 0 then FPSCR[FPRF] ← +denormalized number
                           If sign = 1 then FPSCR[FPRF] ← -denormalized number
                     End
            Normalize operand:
                     Do while frac[0] = 0
                           exp ← exp - 1
                           frac[0-52] ← frac[1-52] || 0b0
                     End
            frD[0] ← sign
            frD[1-11] ← exp + 1023
            frD[12-63] ← frac[1-52]
            End
Done
```

Enabled Exponent Underflow

```
FPSCR[UX] ← 1
sign ← frB[0]
If frB[1-11] = 0 then
          Do
                  exp ← -1022
                  frac[0-52] ← 0b0 || frB[12-63]
          End
If frB[1-11] > 0 then
          Do
                  exp ← frB[1-11] - 1023
                  frac[0-52] ← 0b1 || frB[12-63]
          End
Normalize operand:
          Do while frac[0] = 0
                  exp ← exp - 1
                  frac[0-52] ← frac[1-52] || 0b0
          End
Round single(sign,exp,frac[0-52],0,0,0)
FPSCR[XX] ← FPSCR[XX] | FPSCR[FI]
exp ← exp + 192
frD[0] ← sign
frD[1-11] ← exp + 1023
frD[12-63] ← frac[1-52]
If sign = 0 then FPSCR[FPRF] ← +normal number
If sign = 1 then FPSCR[FPRF] ← -normal number
Done
```

Disabled Exponent Overflow

```
FPSCR[OX] ←1
If FPSCR[RN] = 0b00 then        /* Round to Nearest */
          Do
                  If frB[0] = 0 then frD ← 0x7FF0_0000_0000_0000
                  If frB[0] = 1 then frD ←0xFFF0_0000_0000_0000
                  If frB[0] = 0 then FPSCR[FPRF] ←+infinity
                  If frB[0] = 1 then FPSCR[FPRF] ←-infinity
          End
If FPSCR[RN] = 0b01 then        /* Round Truncate */
          Do
                  If frB[0] = 0 then frD ←0x47EF_FFFF_E000_0000
                  If frB[0] = 1 then frD ←0xC7EF_FFFF_E000_0000
                  If frB[0] = 0 then FPSCR[FPRF] ←+normal number
                  If frB[0] = 1 then FPSCR[FPRF] ←-normal number
          End
If FPSCR[RN] = 0b10 then        /* Round to +Infinity */
          Do
                  If frB[0] = 0 then frD ←0x7FF0_0000_0000_0000
                  If frB[0] = 1 then frD ←0xC7EF_FFFF_E000_0000
                  If frB[0] = 0 then FPSCR[FPRF] ←+infinity
                  If frB[0] = 1 then FPSCR[FPRF] ←-normal number
          End
If FPSCR[RN] = 0b11 then        /* Round to -Infinity */
          Do
                  If frB[0] = 0 then frD ←0x47EF_FFFF_E000_0000
                  If frB[0] = 1 then frD ← 0xFFF0_0000_0000_0000
                  If frB[0] = 0 then FPSCR[FPRF] ←+normal number
                  If frB[0] = 1 then FPSCR[FPRF] ←-infinity
          End
FPSCR[FR] ← undefined
FPSCR[FI] ← 1
FPSCR[XX] ← 1
Done
```

Enabled Exponent Overflow

```
sign ←frB[0]
exp ←frB[1-11] - 1023
          frac[0-52] ← 0b1 || frB[12-63]
          Round single(sign,exp,frac[0-52],0,0,0)
          FPSCR[XX] ← FPSCR[XX] | FPSCR[FI]
Enabled Overflow
          FPSCR[OX] ←1
          exp ←exp - 192
          frD[0] ←sign
          frD[1-11] ←exp + 1023
          frD[12-63] ←frac[1-52]
          If sign = 0 then FPSCR[FPRF] ←+normal number
          If sign = 1 then FPSCR[FPRF] ←-normal number
Done
```

Zero Operand

```
frD ←frB
If frB[0] = 0 then FPSCR[FPRF] ←+zero
If frB[0] = 1 then FPSCR[FPRF] ←-zero
FPSCR[FR FI] ← 0b00
Done
```

Infinity Operand

```
frD ← frB
If frB[0] = 0 then FPSCR[FPRF] ← +infinity
If frB[0] = 1 then FPSCR[FPRF] ← -infinity
Done
```

QNaN Operand

```
frD ← frB[0-34] || 0b0_0000_0000_0000_0000_0000_0000_0000
FPSCR[FPRF] ← QNaN
FPSCR[FR FI] ← 0b00
Done
```

SNaN Operand

```
FPSCR[VXSNAN] ← 1
If FPSCR[VE] = 0 then
          Do
                  frD[0-11] ← frB[0-11]
                  frD[12] ← 1
                  frD[13-63] ← frB[13-34] ||
0b0_0000_0000_0000_0000_0000_0000_0000
                  FPSCR[FPRF] ← QNaN
          End
FPSCR[FR FI] ← 0b00
Done
```

Normal Operand

```
sign ← frB[0]
exp ← frB[1-11] - 1023
frac[0-52] ← 0b1 || frB[12-63]
Round single(sign,exp,frac[0-52],0,0,0)
FPSCR[XX] ← FPSCR[XX] | FPSCR[FI]
If exp > +127 and FPSCR[OE] = 0 then go to Disabled Exponent Overflow
If exp > +127 and FPSCR[OE] = 1 then go to Enabled Overflow
frD[0] ← sign
frD[1-11] ← exp + 1023
frD[12-63] ← frac[1-52]
If sign = 0 then FPSCR[FPRF] ← +normal number
If sign = 1 then FPSCR[FPRF] ← -normal number
Done
```

Round Single (sign,exp,frac[0–52],G,R,X)

```
inc ← 0
lsb ← frac[23]
gbit ← frac[24]
rbit ← frac[25]
xbit ← (frac[26-52] || G || R || X) ¦ 0
If FPSCR[RN] = 0b00 then
          Do
                  If sign || lsb || gbit || rbit || xbit = 0bu11uu then inc ← 1
                  If sign || lsb || gbit || rbit || xbit = 0bu011u then inc ← 1
                  If sign || lsb || gbit || rbit || xbit = 0bu01u1 then inc ← 1
          End
If FPSCR[RN] = 0b10 then
          Do
                  If sign || lsb || gbit || rbit || xbit = 0b0u1uu then inc ← 1
                  If sign || lsb || gbit || rbit || xbit = 0b0uu1u then inc ← 1
                  If sign || lsb || gbit || rbit || xbit = 0b0uuu1 then inc ← 1
          End
If FPSCR[RN] = 0b11 then
          Do
                  If sign || lsb || gbit || rbit || xbit = 0b1u1uu then inc ← 1
                  If sign || lsb || gbit || rbit || xbit = 0b1uu1u then inc ← 1
                  If sign || lsb || gbit || rbit || xbit = 0b1uuu1 then inc ← 1
          End
frac[0-23] ← frac[0-23] + inc
If carry_out =1 then
          Do
                  frac[0-23] ← 0b1 || frac[0-22]
                  exp ← exp + 1
          End
frac[24-52] ← (29)0
FPSCR[FR] ← inc
FPSCR[FI] ← gbit | rbit | xbit
Return
```

C.4.2 Floating-Point Convert to Integer Model

The following algorithm describes the operation of the floating-point convert to integer instructions. In this example, ‘u’ represents an undefined hexadecimal digit.

```
If Floating Convert to Integer Word
          Then Do
                  Then round_mode ← FPSCR[RN]
                  tgt_precision ← 32-bit integer
          End
If Floating Convert to Integer Word with round toward Zero
          Then Do
                  round_mode ← 0b01
                  tgt_precision ← 32-bit integer
          End
If Floating Convert to Integer Double Word
          Then Do
                  round_mode ← FPSCR[RN]
                  tgt_precision ← 64-bit integer
          End
If Floating Convert to Integer Double Word with Round toward Zero
          Then Do
                  round_mode ← 0b01
                  tgt_precision ← 64-bit integer
          End
sign ← frB[0]
If frB[1-11] = 2047 and frB[12-63] = 0 then goto Infinity Operand
If frB[1-11] = 2047 and frB[12] = 0 then goto SNaN Operand
If frB[1-11] = 2047 and frB[12] = 1 then goto QNaN Operand
If frB[1-11] > 1054 then goto Large Operand


If frB[1-11] > 0 then exp ← frB[1-11] - 1023 /* exp - bias */
If frB[1-11] = 0 then exp ← -1022
If frB[1-11] > 0 then frac[0-64]← 0b01 || frB[12-63] || (11)0 /*normal*/
If frB[1-11] = 0 then frac[0-64]← 0b00 || frB[12-63] || (11)0 /*denormal*/


gbit || rbit || xbit ← 0b000
Do i = 1,63 - exp               /*do the loop 0 times if exp = 63*/
          frac[0-64] || gbit || rbit || xbit ← 0b0 || frac[0-64] || gbit || (rbit
| xbit)
End
```

Round Integer (sign,frac[0–64],gbit,rbit,xbit,round_mode)

In this example, ‘u’ represents an undefined hexadecimal digit. Comparisons ignore the u bits.

```
If sign = 1 then frac[0-64] ← ¬frac[0-64] + 1 /* needed leading 0 for -2^64 < frB
< -2^63*/

If tgt_precision = 32-bit integer and frac[0-64] > +2^31 - 1
          then goto Large Operand
If tgt_precision = 64-bit integer and frac[0-64] > +2^63 - 1
          then goto Large Operand
If tgt_precision = 32-bit integer and frac[0-64] < -2^31 then goto Large Operand

FPSCR[XX] ← FPSCR[XX] | FPSCR[FI]

If tgt_precision = 64-bit integer and frac[0-64] < -2^63 then goto Large Operand
If tgt_precision = 32-bit integer
          then frD ← 0xxuuu_uuuu || frac[33-64]
If tgt_precision = 64-bit integer then frD ← frac[1-64]
FPSCR[FPRF] ← undefined
Done
```

Round Integer(sign,frac[0–64],gbit,rbit,xbit,round_mode)

In this example, ‘u’ represents an undefined hexadecimal digit. Comparisons ignore the u bits.

```
inc ← 0
If round_mode = 0b00 then
          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
          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
          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 tgt_precision = 32-bit integer then
                  Do
                  If sign = 0 then frD ← 0xuuuu_uuuu_7FFF_FFFF
                  If sign = 1 then frD ← 0xuuuu_uuuu_8000_0000
                  End
          Else
                  Do
                  If sign = 0 then frD ← 0x7FFF_FFFF_FFFF_FFFF
                  If sign = 1 then frD ← 0x8000_0000_0000_0000
                  End
          FPSCR[FPRF] ←undefined
          End
Done
```

SNaN Operand

```
FPSCR[FR FI VXCVI VXSNAN] ← 0b0011
If FPSCR[VE] = 0 then
          Do
          If tgt_precision = 32-bit integer
                  then frD ← 0xuuuu_uuuu_8000_0000
          If tgt_precision = 64-bit integer
                  then frD ← 0x8000_0000_0000_0000
                  FPSCR[FPRF] ← undefined
          End
Done
```

QNaN Operand

```
FPSCR[FR FI VXCVI] ← 0b001
If FPSCR[VE] = 0 then
          Do
          If tgt_precision = 32-bit integer then frD ← 0xuuuu_uuuu_8000_0000
          If tgt_precision = 64-bit integer then frD ← 0x8000_0000_0000_0000
                  FPSCR[FPRF] ←undefined
          End
Done
```

Large Operand

```
FPSCR[FR FI VXCVI] ← 0b001
If FPSCR[VE] = 0 then Do
          If tgt_precision = 32-bit integer then
                  Do
                  If sign = 0 then frD ← 0xuuuu_uuuu_7FFF_FFFF
                  If sign = 1 then frD ← 0xuuuu_uuuu_8000_0000
                  End
          Else
                  Do
                  If sign = 0 then frD ← 0x7FFF_FFFF_FFFF_FFFF
                  If sign = 1 then frD ← 0x8000_0000_0000_0000
                  End
          FPSCR[FPRF] ← undefined
          End
Done
```

C.4.3 Floating-Point Convert from Integer Model

The following describes, algorithmically, the operation of the floating-point convert from integer instructions.

```
sign ← frB[0]
exp ← 63
frac[0-63] ← frB

If frac[0-63] = 0 then go to Zero Operand

If sign = 1 then frac[0-63] ← ¬frac[0-63] + 1

Do while frac[0] = 0
          frac[0-63] ← frac[1-63] || '0'
          exp ← exp - 1
End
```

Round Float(sign,exp,frac[0–63],FPSCR[RN])

```
If sign = 1 then FPSCR[FPRF] ← -normal number
If sign = 0 then FPSCR[FPRF] ← +normal number
frD[0] ← sign
frD[1-11] ← exp + 1023
frD[12-63] ← frac[1-52]
Done
```

Zero Operand

```
FPSCR[FR FI] ← 0b00
FPSCR[FPRF] ← "+zero"
frD ← 0x0000_0000_0000_0000
Done
```

Round Float(sign,exp,frac[0–63],round_mode)

In this example ‘u’ represents an undefined hexadecimal digit. Comparisons ignore the u bits.

```
inc ← 0
lsb ← frac[52]
gbit ← frac[53]
rbit ← frac[54]
xbit ← frac[55-63] > 0
If round_mode = 0b00 then
          Do
          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
          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
          If sign || lsb || gbit || rbit || xbit = 0b1u1uu then inc ← 1
          If sign || lsb || gbit || rbit || xbit = 0b1uu1u then inc ← 1
          If sign || lsb || gbit || rbit || xbit = 0b1uuu1 then inc ← 1
          End
frac[0-52] ← frac[0-52] + inc
If carry_out = 1 then exp ← exp + 1
FPSCR[FR] ← inc
FPSCR[FI] ← gbit | rbit | xbit
FPSCR[XX] ← FPSCR[XX] | FPSCR[FI]
Return
```

C.5 Floating-Point Selection

The following are examples of how the optional **fsel** 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 PowerPC 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 *Appendix C.3 Floating-Point Conversions*.

Note: Care must be taken in using **fsel** if IEEE compatibility is required, or if the values being tested can be NaNs or infinities.

C.5.1 Comparison to Zero

This section provides examples in a program fragment code sequence for the comparison to zero case.

Table C-4. Comparison to Zero

High Level Language	PowerPC	Note(s) from *Appendix C.5.4 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.5.2 Minimum and Maximum

This section provides examples in a program fragment code sequence for the minimum and maximum cases.

Table C-5. Minimum and Maximum

High Level Language	PowerPC	Note(s) from *Appendix C.5.4 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.5.3 Simple If-Then-Else Constructions

This section provides examples in a program fragment code sequence for simple if-then-else statements.

Table C-6. Simple If-Then-Else

High Level Language	PowerPC	Note(s) from *Appendix C.5.4 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.5.4 Notes

The following notes apply to the examples found in *Appendix C.5.1 Comparison to Zero*, *Appendix C.5.2 Minimum and Maximum*, and *Appendix C.5.3 Simple If-Then-Else Constructions*, and to the corresponding cases using the other three arithmetic relations (<, ≥, and ≠). These notes should also be considered when any other use of **fsel** is contemplated.

In these notes the “optimized program” is the PowerPC program shown, and the “unoptimized program” (not shown) is the corresponding PowerPC 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. (Note that the architecture specification also refers to exceptions as interrupts.)
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.

C.6 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-precision format prior to loading the operands into the target FPR. The conversion and loading steps follow:

Let WORD[0–31] be the floating point single-precision operand accessed from memory.

Normalized Operand

```
If WORD[1-8] > 0 and WORD[1-8] < 255
          frD[0-1] ← WORD[0-1]
          frD[2] ← ¬ WORD[1]
          frD[3] ← ¬ WORD[1]
          frD[4] ← ¬ WORD[1]
          frD[5-63] ← WORD[2-31] || (29)0
```

Denormalized Operand

```
If WORD[1-8] = 0 and WORD[9-31] ¦ 0
          sign ← WORD[0]
          exp ← -126
          frac[0-52] ← 0b0 || WORD[9-31] || (29)0
          normalize the operand
          Do while frac[0] = 0
                  frac ← frac[1-52] || 0b0
          exp ← exp - 1
          End
          frD[0] ← sign
          frD[1-11] ← exp + 1023
          frD[12-63] ← frac[1-52]
```

Infinity / QNaN / SNaN / Zero

```
If WORD[1-8] = 255 or WORD[1-31] = 0
          frD[0-1] ← WORD[0-1]
          frD[2] ← WORD[1]
          frD[3] ← WORD[1]
          frD[4] ← WORD[1]
          frD[5-63] ← WORD[2-31] || (29)0
```

For double-precision floating-point load instructions, no conversion is required as the data from memory is copied directly into the FPRs.

Many floating-point load instructions have an update form in which register **r**A is updated with the EA. For these forms, if operand **r**A ≠ 0, the effective address (EA) is placed into register **r**A and the memory element (word or double word) addressed by the EA is loaded into the floating-point register specified by operand **fr**D; if operand **r**A = 0, the instruction form is invalid.

Recall that **r**A, **r**B, and **r**D denote GPRs, while **fr**A, **fr**B, **fr**C, **fr**S, and **fr**D denote FPRs.

C.7 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 **stfiwx** instruction. Because the FPRs support only floating-point double format for floating-point data, single-precision store floating-point instructions convert double-precision data to single-precision format prior to storing the operands into memory. The conversion steps follow:

Let WORD[0–31] be the word written to in memory.

No Denormalization Required (includes Zero/Infinity/NaN)

```
if frS[1-11] > 896 or frS[1-63] = 0 then
          WORD[0-1] ← frS[0-1]
          WORD[2-31] ← frS[5-34]
```

Denormalization Required

```
if 874 ≤ frS[1-11] ≤ 896 then
          sign ← frS[0]
          exp ← frS[1-11] - 1023
          frac ← 0b1 || frS[12-63]
          Denormalize operand
                  Do while exp < -126
                        frac ← 0b0 || frac[0-62]
                        exp ← exp + 1
                  End
          WORD[0] ← sign
          WORD[1-8] ← 0x00
          WORD[9-31] ← frac[1-23]
else WORD ← undefined
```

Notice that if the value to be stored by a single-precision store floating-point instruction is larger in magnitude than the maximum number representable in single format, the first case mentioned, “No Denormalization Required,” applies. The result stored in WORD is then a well-defined value, but is not numerically equal to the value in the source register (that is, the result of a single-precision load floating-point from WORD will not compare equal to the contents of the original source register).

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

It is important to note that for double-precision store floating-point instructions and for the store floating-point as integer word instruction no conversion is required as the data from the FPR is copied directly into memory.

THIS PAGE INTENTIONALLY LEFT BLANK

Appendix D. Synchronization Programming Examples

The examples in this appendix show how synchronization instructions can be used to emulate various synchronization primitives and how to provide more complex forms of synchronization.

For each of these examples, it is assumed that a similar sequence of instructions is used by all processes requiring synchronization of the accessed data.

D.1 General Information

The following points provide general information about the **lwarx** and **stwcx.** instructions:

- In general, **lwarx** and **stwcx.** instructions should be paired, with the same effective address (EA) used for both. The only exception is that an unpaired **stwcx.** instruction to any (scratch) effective address can be used to clear any reservation held by the processor.
- It is acceptable to execute an **lwarx** instruction for which no **stwcx.** instruction is executed. Such a dangling **lwarx** instruction occurs in the example shown in *Appendix D.2.5 Test and Set* if the value loaded is not zero.
- 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 in *Appendix D.2.5 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.** instructions.
- The manner in which **lwarx** and **stwcx.** are communicated to other processors and mechanisms, and between levels of the memory subsystem within a given processor, is implementation-dependent. In some implementations, performance may be improved by minimizing looping on an **lwarx** instruction that fails to return a desired value. For example, in the example provided in *Appendix D.2.5 Test and Set* if the program stays in the loop until the word loaded is zero, the programmer can change the "**bne**- $+12" to "**bne**- loop."

  In some implementations, better performance may be obtained by using an ordinary load instruction to do the initial checking of the value, as follows:

```
loop:  lwz     r5,0(r3)     #load the word
       cmpwi   r5,0         #loop back if word
       bne-    loop         #not equal to 0
       lwarx   r5,0,r3      #try again, reserving
       cmpwi   r5,0         #(likely to succeed)
       bne     loop         #try to store nonzero
       stwcx.  r4,0,r3      #
       bne-    loop         #loop if lost reservation
```

- In a multiprocessor, livelock (a state in which processors interact in a way such that no processor makes progress) is possible if a loop containing an **lwarx**/**stwcx.** pair also contains an ordinary store instruction for which any byte of the affected memory area is in the reservation granule of the reservation. For example, the first code sequence shown in *Appendix D.5 List Insertion* can cause livelock if two list elements have next element pointers in the same reservation granule.

Note: The examples in this appendix use the **lwarx**/**stwcx.** instructions, which address words in memory. For 64-bit implementations, these examples can be modified to address double words by changing all **lwarx** instructions to **ldarx** instructions, all **stwcx.** instructions to **stdcx.** instructions, all **stw** instructions to **std** instructions, and all **cmpw** and **cmpwi** extended mnemonics to **cmpd** and **cmpdi**, respectively.

D.2 Synchronization Primitives

The following examples show how the **lwarx** and **stwcx.** instructions can be used to emulate various synchronization primitives. The sequences used to emulate the various primitives consist primarily of a loop using the **lwarx** and **stwcx.** instructions. Additional synchronization is unnecessary, because the **stwcx.** will fail, clearing the EQ bit, if the word loaded by **lwarx** has changed before the **stwcx.** is executed.

D.2.1 Fetch and No-Op

The fetch and no-op primitive atomically loads the current value in a word in memory. In this example, it is assumed that the address of the word to be loaded is in GPR3 and the data loaded are returned in GPR4.

```
loop:      lwarx   r4,0,r3     #load and reserve
           stwcx.  r4,0,r3     #store old value if still reserved
           bne-    loop        #loop if lost reservation
```

The **stwcx.**, if it succeeds, stores to the destination location the same value that was loaded by the preceding **lwarx**. While the store is redundant with respect to the value in the location, its success ensures that the value loaded by the **lwarx** was the current value (that is, 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).

D.2.2 Fetch and Store

The fetch and store primitive atomically loads and replaces a word in memory.

In this example, it is assumed that the address of the word to be loaded and replaced is in GPR3, the new value is in GPR4, and the old value is returned in GPR5.

```
loop:      lwarx   r5,0,r3     #load and reserve
           stwcx.  r4,0,r3     #store new value if still reserved
           bne-    loop        #loop if lost reservation
```

D.2.3 Fetch and Add

The fetch and add primitive atomically increments a word in memory.

In this example, it is assumed that the address of the word to be incremented is in GPR3, the increment is in GPR4, and the old value is returned in GPR5.

```
loop:      lwarx   r5,0,r3     #load and reserve
           add     r0,r4,r5    #increment word
           stwcx.  r0,0,r3     #store new value if still reserved
           bne-    loop        #loop if lost reservation
```

D.2.4 Fetch and AND

The fetch and AND primitive atomically ANDs a value into a word in memory.

In this example, it is assumed that the address of the word to be ANDed is in GPR3, the value to AND into it is in GPR4, and the old value is returned in GPR5.

```
loop:     lwarx   r5,0,r3     #load and reserve
          and     r0,r4,r5    #AND word
          stwcx.  r0,0,r3     #store new value if still reserved
          bne-    loop        #loop if lost reservation
```

This sequence can be changed to perform another Boolean operation atomically on a word in memory, simply by changing the AND instruction to the desired Boolean instruction (OR, XOR, etc.).

D.2.5 Test and Set

This version of the test and set primitive atomically loads a word from memory, ensures that the word in memory is a nonzero value, and sets CR0[EQ] according to whether the value loaded is zero.

In this example, it is assumed that the address of the word to be tested is in GPR3, the new value (nonzero) is in GPR4, and the old value is returned in GPR5.

```
loop:     lwarx   r5,0,r3     #load and reserve
          cmpwi   r5, 0       #done if word
          bne     $+12        #not equal to 0
          stwcx.  r4,0,r3     #try to store non-zero
          bne-    loop        #loop if lost reservation
```

D.3 Compare and Swap

The compare and swap primitive atomically compares a value in a register with a word in memory. If they are equal, it stores the value from a second register into the word in memory. If they are unequal, it loads the word from memory into the first register, and sets the EQ bit of the CR0 field to indicate the result of the comparison.

In this example, it is assumed that the address of the word to be tested is in GPR3, the word that is compared is in GPR4, the new value is in GPR5, and the old value is returned in GPR4.

```
loop:     lwarx   r6,0,r3     #load and reserve
          cmpw    r4,r6       #first 2 operands equal ?
          bne-    exit        #skip if not
          stwcx.  r5,0,r3     #store new value if still reserved
          bne-    loop        #loop if lost reservation
exit:     mr      r4,r6       #return value from memory
```

Notes:

1. The semantics in this example are based on the IBM System/370™ compare and swap instruction. Other architectures may define this 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 the **lwarx** and **stwcx.** instructions. Although the instruction is atomic, it checks only for whether the current value matches the old value. An error can occur if the value had been changed and restored before being tested.

3. In some applications, the second **bne-** instruction and/or the **mr** instruction can be omitted. The first **bne-** is needed only if the application requires that if the EQ bit of CR0 field on exit indicates not equal, then the original compared value in **r**4 and **r**6 are in fact not equal. The **mr** is needed only if the application requires that if the compared values are not equal, then the word from memory is loaded into the register with which it was compared (rather than into a third register). If either, or both, of these instructions is omitted, the resulting compare and swap does not obey the IBM System/370 semantics.

D.4 Lock Acquisition and Release

This section gives examples of how dependencies and the synchronization instructions can be used to implement locks, import and export barriers, and similar constructs.

D.4.1 Lock Acquisition and Import Barriers

An “import barrier” is an instruction or sequence of instructions that prevents memory accesses caused by instructions following the barrier from being performed before memory accesses that acquire a lock have been performed. An import barrier can be used to ensure that a shared data structure protected by a lock is not accessed until the lock has been acquired. A **sync** instruction can be used as an import barrier, but the approaches shown below will generally yield better performance because they order only the relevant memory accesses.

D.4.1.1 Acquire Lock and Import Shared Memory

If **lwarx** and **stwcx.** instructions are used to obtain the lock, an import barrier can be constructed by placing an **isync** instruction immediately following the loop containing the **lwarx** and **stwcx.**. The following example uses the “Compare and Swap” primitive to acquire the lock.

In this example it is assumed that the address of the lock is in GPR 3, the value indicating that the lock is free is in GPR 4, the value to which the lock should be set is in GPR 5, the old value of the lock is returned in GPR 6, and the address of the shared data structure is in GPR 9.

```
loop:     lwarx r6,0,r3       #load lock and reserve
          cmpw r4,r6          #skip ahead if
          bne- wait           # lock not free
          stwcx. r5,0,r3      #try to set lock
          bne- loop           #loop if lost reservation
          isync               #import barrier
          lwz r7,data1(r9)    #load shared data
          ..
          wait: ...           #wait for lock to free
```

The second **bne-** does not complete until CR0 has been set by the **stwcx.**. The **stwcx.** does not set CR0 until it has completed (successfully or unsuccessfully). The lock is acquired when the **stwcx.** completes successfully. Together, the second **bne-** and the subsequent **isync** create an import barrier that prevents the load from “data1” from being performed until the branch has been resolved not to be taken.

If the shared data structure is in memory that is neither Write Through Required nor Caching Inhibited, an **lwsync** instruction can be used instead of the **isync** instruction. If **lwsync** is used, the load from “data1” may be performed before the **stwcx.**. But if the **stwcx.** fails, the second branch is taken and the **lwarx** is reexecuted. If the **stwcx.** succeeds, the value returned by the load from “data1” is valid even if the load is performed before the **stwcx.**, because the **lwsync** ensures that the load is performed after the instance of the **lwarx** that created the reservation used by the successful **stwcx.**.

D.4.1.2 Obtain Pointer and Import Shared Memory

If **lwarx** and **stwcx.** instructions are used to obtain a pointer into a shared data structure, an import barrier is not needed if all the accesses to the shared data structure depend on the value obtained for the pointer. The following example uses the “Fetch and Add” primitive to obtain and increment the pointer.

In this example it is assumed that the address of the pointer is in GPR 3, the value to be added to the pointer is in GPR 4, and the old value of the pointer is returned in GPR 5.

```
loop:     lwarx r5,0,r3        #load pointer and reserve
          add r0,r4,r5         #increment the pointer
          stwcx. r0,0,r3       #try to store new value
          bne- loop            #loop if lost reservation
          lwz r7,data1(r5)     #load shared data
```

The load from “data1” cannot be performed until the pointer value has been loaded into GPR 5 by the **lwarx**. The load from “data1” may be performed before the **stwcx.**, but if the **stwcx.** fails, the branch is taken and the value returned by the load from “data1” is discarded. If the **stwcx.** succeeds, the value returned by the load from “data1” is valid even if the load is performed before the **stwcx.**, because the load uses the pointer value returned by the instance of the **lwarx** that created the reservation used by the successful **stwcx.**.

An **isync** instruction could be placed between the **bne-** and the subsequent **lwz**, but no **isync** is needed if all accesses to the shared data structure depend on the value returned by the **lwarx**.

D.4.2 Lock Release and Export Barriers

An “export barrier” is an instruction or sequence of instructions that prevents the store that releases a lock from being performed before stores caused by instructions preceding the barrier have been performed. An export barrier can be used to ensure that all stores to a shared data structure protected by a lock will be performed with respect to any other processor before the store that releases the lock is performed with respect to that processor.

D.4.2.1 Export Shared Memory and Release Lock

A **sync** instruction can be used as an export barrier independent of the memory control attributes (for example, presence or absence of the Caching Inhibited attribute) of the memory containing the shared data structure. Because the lock must be in memory that is neither Write Through Required nor Caching Inhibited, if the shared data structure is in memory that is Write Through Required or Caching Inhibited a **sync** instruction must be used as the export barrier.

In this example it is assumed that the shared data structure is in memory that is Caching Inhibited, the address of the lock is in GPR 3, the value indicating that the lock is free is in GPR 4, and the address of the shared data structure is in GPR 9.

```
          stw r7,data1(r9)     #store shared data (last)
          sync                 #export barrier
          stw r4,lock(r3)      #release lock
```

The **sync** ensures that the store that releases the lock will not be performed with respect to any other processor until all stores caused by instructions preceding the **sync** have been performed with respect to that processor.

D.4.2.2 Export Shared Memory and Release Lock using EIEIO or LYSYNC

If the shared data structure is in memory that is neither Write Through Required nor Caching Inhibited, an **eieio** instruction can be used as the export barrier. Using **eieio** rather than **sync** will yield better performance in most systems.

In this example it is assumed that the shared data structure is in memory that is neither Write Through Required nor Caching Inhibited, the address of the lock is in GPR 3, the value indicating that the lock is free is in GPR 4, and the address of the shared data structure is in GPR 9.

```
stw r7,data1(r9)     #store shared data (last)
eieio                #export barrier
stw r4,lock(r3)      #release lock
```

The **eieio** 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 **eieio** have been performed with respect to that processor.

However, for memory that is neither Write Through Required nor Caching Inhibited, **eieio** orders only stores and has no effect on loads. If the portion of the program preceding the **eieio** 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, **lwsync** should be used instead of **eieio**. Alternatively, the technique described in*Appendix D.4.3 Safe Fetch* can be used.

D.4.3 Safe Fetch

If a load must be performed before a subsequent store (for example, 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 memory operand to be loaded is in GPR 3, the contents of the memory operand are returned in GPR 4, and the address of the memory operand to be stored is in GPR 5.

```
lwz r4,0(r3)         #load shared data
cmpw r4,r4           #set CR0 to "equal"
bne- $-8             #branch never taken
stw r7,0(r5)         #store other shared data
```

An alternative is to use a technique similar to that described in *Appendix D.4.1.2 Obtain Pointer and Import Shared Memory*, by causing the **stw** to depend on the value returned by the **lwz** and omitting the **cmpw** and **bne-**. The dependency could be created by ANDing the value returned by the **lwz** with zero and then adding the result to the value to be stored by the **stw**. If both memory operands are in memory that is neither Write Through Required nor Caching Inhibited, another alternative is to replace the **cmpw** and ***bne-*** with an **lwsync** instruction.

D.5 List Insertion

The following example shows how the **lwarx** and **stwcx.** instructions can be used to implement simple LIFO (last-in-first-out) insertion into a singly-linked list. (Complicated list insertion, in which multiple values must be changed atomically, or in which the correct order of insertion depends on the contents of the elements, cannot be implemented in the manner shown below, and requires a more complicated strategy such as using locks.)

The next element pointer from the list element after which the new element is to be inserted, here called the parent element, is stored into the new element, so that the new element points to the next element in the list—this store is performed unconditionally. Then the address of the new element is conditionally stored into the parent element, thereby adding the new element to the list.

In this example, it is assumed that the address of the parent element is in GPR3, the address of the new element is in GPR4, and the next element pointer is at offset zero from the start of the element. It is also assumed that the next element pointer of each list element is in a reservation granule separate from that of the next element pointer of all other list elements.

```
loop:     lwarx   r2,0,r3     #get next pointer
          stw     r2,0(r4)    #store in new element
          eieio               #order stw before stwcx.
          stwcx.  r4,0,r3     #add new element to list
          bne-    loop        #loop if stwcx. failed
```

In the preceding example, **lwsync** can be used instead of **eieio**.

In the preceding example, if two list elements have next element pointers in the same reservation granule in a multiprocessor system, livelock can occur. Livelock is a state in which processors interact in a way such that no processor makes forward progress.

If it is not possible to allocate list elements such that each element's next element pointer is in a different reservation granule, then livelock can be avoided by using the following sequence:

```
          lwz     r2,0(r3)    #get next pointer
loop1:    mr      r5,r2       #keep a copy
          stw     r2,0(r4)    #store in new element
          sync                #order stw before stwcx.
                              # and before lwarx

loop2:    lwarx   r2,0,r3     #get it again
          cmpw    r2,r5       #loop if changed (someone
          bne-    loop1       #else progressed)
          stwcx.  r4,0,r3     #add new element to list
          bne-    loop2       #loop if failed
```

In the preceding example, livelock is avoided by the fact that each processor reexecutes the **stw** only if some other processor has made forward progress.

D.6 Notes

1. To increase the likelihood that forward progress is made, it is important that looping on **lwarx**/**stwcx.** pairs be minimized. For example, in the “Test and Set” sequence shown in *Appendix D.2.5* , this is achieved by testing the old value before attempting the store; were the order reversed, more **stwcx.** instructions might be executed, and reservations might more often be lost between the **lwarx** and the **stwcx.**.

2. The manner in which **lwarx** and **stwcx.** are communicated to other processors and mechanisms, and between levels of the memory hierarchy within a given processor, is implementation-dependent. In some implementations performance may be improved by minimizing looping on a **lwarx** instruction that fails to return a desired value. For example, in the “Test and Set” sequence shown in *Appendix D.2.5* , 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
       cmpwi   r5,0         #loop back if word
       bne-    loop         # not equal to 0
       lwarx   r5,0,r3      #try again, reserving
       cmpwi   r5,0         # (likely to succeed)
       bne-    loop
       stwcx.  r4,0,r3      #try to store non-0
       bne-    loop         #loop if lost reservation
```

3. In a multiprocessor, livelock is possible if there is a Store instruction (or any other instruction that can clear another processor's reservation) between the **lwarx** and the **stwcx.** of a **lwarx**/**stwcx.** loop and any byte of the memory location specified by the Store is in the reservation granule. For example, the first code sequence shown in *Appendix D.5 List Insertion* can cause livelock if two list elements have next element pointers in the same reservation granule.

Appendix E. Simplified Mnemonics

This appendix is provided in order to simplify the writing and comprehension of assembler language programs. Included are a set of simplified mnemonics and symbols that define the simple shorthand used for the most frequently-used forms of branch conditional, compare, trap, rotate and shift, and certain other instructions.

Note: The architecture specification refers to simplified mnemonics as extended mnemonics.

E.1 Symbols

The symbols in *Table E-1* are defined for use in instructions (basic or simplified mnemonics) that specify a condition register (CR) field or a bit in the CR.

Table E-1. Condition Register Bit and Identification Symbol Descriptions

Symbol	Value	Bit Field Range	Description
lt	0	—	Less than. Identifies a bit number within a CR field.
gt	1	—	Greater than. Identifies a bit number within a CR field.
eq	2	—	Equal. Identifies a bit number within a CR field.
so	3	—	Summary overflow. Identifies a bit number within a CR field.
un	3	—	Unordered (after floating-point comparison). Identifies a bit number in a CR field.
cr0	0	0–3	CR0 field
cr1	1	4–7	CR1 field
cr2	2	8–11	CR2 field
cr3	3	12–15	CR3 field
cr4	4	16–19	CR4 field
cr5	5	20–23	CR5 field
cr6	6	24–27	CR6 field
cr7	7	28–31	CR7 field
Note: To identify a CR bit, an expression in which a CR field symbol is multiplied by 4 and then added to a bit-number-within-CR-field symbol can be used.			

Note: The simplified mnemonics in *Appendix E.5.2 Basic Branch Mnemonics* and *Appendix E.6 Simplified Mnemonics for Condition Register Logical Instructions* require identification of a CR bit—if one of the CR field symbols is used, it must be multiplied by 4 and added to a bit-number-within-CR-field (value in the range of [0–3], explicit or symbolic). The simplified mnemonics in *Appendix E.5.3 Branch Mnemonics Incorporating Conditions* and *Appendix E.3 Simplified Mnemonics for Compare Instructions* require identification of a CR field—if one of the CR field symbols is used, it must not be multiplied by 4. (For the simplified mnemonics in *Appendix E.5.3 Branch Mnemonics Incorporating Conditions* the bit number within the CR field is part of the simplified mnemonic. The CR field is identified, and the assembler does the multiplication and addition required to produce a CR bit number for the BI field of the underlying basic mnemonic.)

E.2 Simplified Mnemonics for Subtract Instructions

This section discusses simplified mnemonics for the subtract instructions.

E.2.1 Subtract Immediate

Although there is no subtract immediate instruction, its effect can be achieved by using an add immediate instruction with the immediate operand negated. Simplified mnemonics are provided that include this negation, making the intent of the computation more clear.

subi rD,**r**A,value	(equivalent to	**addi r**D,**r**A,–value)
subis rD,**r**A,value	(equivalent to	**addis r**D,**r**A,–value)
subic rD,**r**A,value	(equivalent to	**addic r**D,**r**A,–value)
subic. rD,**r**A,value	(equivalent to	**addic. r**D,**r**A,–value)

E.2.2 Subtract

The subtract from instructions subtract the second operand (**r**A) from the third (**r**B). Simplified mnemonics are provided that use the more normal order in which the third operand is subtracted from the second. Both these mnemonics can be coded with an **o** suffix and/or dot (**.**) suffix to cause the OE and/or Rc bit to be set in the underlying instruction.

sub rD,**r**A,**r**B	(equivalent to	**subf**	**r**D,**r**B,**r**A)
subc rD,**r**A,**r**B	(equivalent to	**subfc**	**r**D,**r**B,**r**A)

E.3 Simplified Mnemonics for Compare Instructions

The L field in the integer compare instructions controls whether the operands are treated as 64-bit quantities (when L = ‘1’) or as 32-bit quantities (when L = ‘0’). Simplified mnemonics are provided that represent the L value in the mnemonic rather than requiring it to be coded as a numeric operand.

The **crf**D field can be omitted if the result of the comparison is to be placed into the CR0 field. Otherwise, the target CR field must be specified as the first operand. One of the CR field symbols defined in *Appendix E.1 Symbols* can be used for this operand.

Note: The **crf**D field can normally be omitted when the CR0 field is the target, if L is specified the assembler requires that **crf**D be specified explicitly.

E.3.1 Double-Word Comparisons

The instructions listed in *Table E-2* are simplified mnemonics that should be supported by assemblers provided for 64-bit implementations.

Table E-2. Simplified Mnemonics for Double-Word Compare Instructions

Operation	Simplified Mnemonic	Equivalent to:
Compare Double Word Immediate	**cmpdi crf**D,**r**A,SIMM	**cmpi crf**D,**1**,**r**A,SIMM
Compare Double Word	**cmpd crf**D,**r**A,**r**B	**cmp crf**D,**1**,**r**A,**r**B
Compare Logical Double Word Immediate	**cmpldi crf**D,**r**A,UIMM	**cmpli crf**D,**1**,**r**A,UIMM
Compare Logical Double Word	**cmpld crf**D,**r**A,**r**B	**cmpl crf**D,**1**,**r**A,**r**B

Following are examples using the double-word compare mnemonics.

1. Compare **r**A and immediate value 100 as unsigned 64-bit integers and place result in CR0.

   cmpldi rA,**100** (equivalent to **cmpli 0,1,r**A,**100**)

2. Same as (1), but place result in CR4.

   cmpldi cr4,rA,**100** (equivalent to **cmpli 4,1,r**A,**100**)

3. Compare **r**A and **r**B as signed 64-bit integers and place result in CR0.

   cmpd rA,**r**B (equivalent to **cmp 0,1,r**A,**rB**)

E.3.2 Word Comparisons

The instructions listed in *Table E-3* are simplified mnemonics that should be supported by assemblers for all PowerPC implementations.

Table E-3. Simplified Mnemonics for Word Compare Instructions

Operation	Simplified Mnemonic	Equivalent to:
Compare Word Immediate	**cmpwi crf**D,**r**A,SIMM	**cmpi crf**D,**0**,**r**A,SIMM
Compare Word	**cmpw crf**D,**r**A,**r**B	**cmp crf**D,**0**,**r**A,**r**B
Compare Logical Word Immediate	**cmplwi crf**D,**r**A,UIMM	**cmpli crf**D,**0**,**r**A,UIMM
Compare Logical Word	**cmplw crf**D,**r**A,**r**B	**cmpl crf**D,**0**,**r**A,**r**B

Following are examples using the word compare mnemonics.

1. Compare **r**A[32–63] with immediate value 100 as signed 32-bit integers and place result in CR0.

   cmpwi rA,**100** (equivalent to **cmpi 0,0,r**A,**100**)

2. Same as (1), but place results in CR4.

   cmpwi cr4,rA,**100** (equivalent to **cmpi 4,0,r**A,**100**)

3. Compare **r**A[32–63] and **r**B[32–63] as unsigned 32-bit integers and place result in CR0.

   cmplw rA,**r**B (equivalent to **cmpl 0,0,r**A,**r**B)

E.4 Simplified Mnemonics for Rotate and Shift Instructions

The rotate and shift instructions provide powerful and general ways to manipulate register contents, but can be difficult to understand. Simplified mnemonics that allow some of the simpler operations to be coded easily are provided for the following types of operations:

Extract	Select a field of *n* bits starting at bit position *b* in the source register; left or right justify this field in the target register; clear all other bits of the target register.
Insert	Select a left-justified or right-justified field of *n* bits in the source register; insert this field starting at bit position *b* of the target register; leave other bits of the target register unchanged. (No simplified mnemonic is provided for insertion of a left-justified field, when operating on double words, because such an insertion requires more than one instruction.)
Rotate	Rotate the contents of a register right or left *n* bits without masking.
Shift	Shift the contents of a register right or left *n* bits, clearing vacated bits (logical shift).
Clear	Clear the leftmost or rightmost *n* bits of a register.
Clear left and shift left	Clear the leftmost *b* bits of a register, then shift the register left by *n* bits. This operation can be used to scale a (known non-negative) array index by the width of an element.

E.4.1 Operations on Double Words

The operations shown in *Table E-4* are available only in 64-bit implementations. All these mnemonics can be coded with a dot (**.**) suffix to cause the Rc bit to be set in the underlying instruction.

Table E-4. Double-Word Rotate and Shift Instructions

Operation	Simplified Mnemonic	Equivalent to:
Extract and left justify immediate	**extldi** rA,**r**S,*n*,*b* (*n* > 0)	**rldicr** rA,**r**S,*b*,*n* – 1
Extract and right justify immediate	**extrdi** rA,**r**S,*n*,*b* (*n* > 0)	**rldicl** rA,**r**S,*b* + *n*, 64 – *n*
Insert from right immediate	**insrdi** rA,**r**S,*n*,*b* (*n* > 0)	**rldimi** rA,**r**S,64 – (*b* + *n*),*b*
Rotate left immediate	**rotldi** rA,**r**S,*n*	**rldicl** rA,**r**S,*n*,**0**
Rotate right immediate	**rotrdi** rA,**r**S,*n*	**rldicl** rA,**r**S,64 – *n*,**0**
Rotate left	**rotld** rA,**r**S,**r**B	**rldcl** rA,**r**S,**r**B,**0**
Shift left immediate	**sldi** rA,**r**S,*n* (*n* < 64)	**rldicr** rA,**r**S,*n*,63 – *n*
Shift right immediate	**srdi** rA,**r**S,*n* (*n* < 64)	**rldicl** rA,**r**S,64 – *n*,*n*
Clear left immediate	**clrldi** rA,**r**S,*n* (*n* < 64)	**rldicl** rA,**r**S,**0**,*n*
Clear right immediate	**clrrdi** rA,**r**S,*n* (*n* < 64)	**rldicr** rA,**r**S,**0**,63 – *n*
Clear left and shift left immediate	**clrlsldi** rA,**r**S,*b*,*n* (*n* ≤ *b* ≤ 63)	**rldic** rA,**r**S,*n*,*b* – *n*

Examples using double-word mnemonics follow:

1. Extract the sign bit (bit 0) of **r**S and place the result right-justified into **r**A.

   extrdi rA,**r**S,**1,0** (equivalent to **rldicl r**A,**r**S,**1,63**)

2. Insert the bit extracted in (1) into the sign bit (bit [0]) of **r**B.

   insrdi rB,**r**A,**1,0** (equivalent to **rldimi r**B,**r**A,**63,0**)

3. Shift the contents of **r**A left 8 bits.

   sldi rA,**r**A,**8** (equivalent to **rldicr r**A,**r**A,**8,55**)

4. Clear the high-order 32 bits of **r**S and place the result into **r**A.

   clrldi rA,**r**S,**32** (equivalent to **rldicl r**A,**r**S,**0,32**)

E.4.2 Operations on Words

The operations shown in *Table E-5* are available in all implementations. All these mnemonics can be coded with a dot (**.**) suffix to cause the Rc bit to be set in the underlying instruction. The operations, as described in *Appendix E.4.1 Operations on Double Words* apply only to the low-order 32 bits of the registers. The insert operations either preserve the high-order 32 bits of the target register or place rotated data there; the other operations clear these bits.

Table E-5. Word Rotate and Shift Instructions

Operation	Simplified Mnemonic	Equivalent to:
Extract and left justify immediate	**extlwi r**A,**r**S,*n*,*b* (*n* > 0)	**rlwinm r**A,**r**S,*b*,**0**,*n* – 1
Extract and right justify immediate	**extrwi r**A,**r**S,*n*,*b* (*n* > 0)	**rlwinm r**A,**r**S,*b* + *n*, 32 – *n*,**31**
Insert from left immediate	**inslwi r**A,**r**S,*n*,*b* (*n* > 0)	**rlwimi r**A,**r**S,32 – *b*,*b*,(*b* + *n*) – 1
Insert from right immediate	**insrwi r**A,**r**S,*n*,*b* (*n* > 0)	**rlwimi r**A,**r**S,32 – (*b* + *n*),*b*,(*b* + *n*) – 1
Rotate left immediate	**rotlwi r**A,**r**S,*n*	**rlwinm r**A,**r**S,*n*,**0,31**
Rotate right immediate	**rotrwi r**A,**r**S,*n*	**rlwinm r**A,**r**S,32 – *n*,**0,31**
Rotate left	**rotlw r**A,**r**S,**r**B	**rlwnm r**A,**r**S,**r**B,**0,31**
Shift left immediate	**slwi r**A,**r**S,*n* (*n* < 32)	**rlwinm r**A,**r**S,*n*,**0**,31 – *n*
Shift right immediate	**srwi r**A,**r**S,*n* (*n* < 32)	**rlwinm r**A,**r**S,32 – *n*,*n*,**31**
Clear left immediate	**clrlwi r**A,**r**S,*n* (*n* < 32)	**rlwinm r**A,**r**S,**0**,*n*,**31**
Clear right immediate	**clrrwi r**A,**r**S,*n* (*n* < 32)	**rlwinm r**A,**r**S,**0**,**0**,31 – *n*
Clear left and shift left immediate	**clrlslwi r**A,**r**S,*b*,*n* (*n* ≤ *b* ≤ 31)	**rlwinm r**A,**r**S,*n*,*b* – *n*,31 – *n*

Examples using word mnemonics follow:

1. Extract the sign bit (bit [32]) of **r**S and place the result right-justified into **r**A.

   extrwi rA,**r**S,**1,0** (equivalent to **rlwinm r**A,**r**S,**1,31,31**)

2. Insert the bit extracted in (1) into the sign bit (bit [32]) of **r**B.

   insrwi rB,**r**A,**1,0** (equivalent to **rlwimi r**B,**r**A,**31,0,0**)

3. Shift the contents of **r**A left 8 bits, clearing the high-order 32 bits.

   slwi rA,**r**A,**8** (equivalent to **rlwinm r**A,**r**A,**8,0,23**)

4. Clear the high-order 16 bits of the low-order 32 bits of **r**S and place the result into **r**A, clearing the high-order 32 bits of **r**A.

   clrlwi rA,**r**S,**16** (equivalent to **rlwinm r**A,**r**S,**0,16,31**)

E.5 Simplified Mnemonics for Branch Instructions

Mnemonics are provided so that branch conditional instructions can be coded with the condition as part of the instruction mnemonic rather than as a numeric operand. Some of these are shown as examples with the branch instructions.

The mnemonics discussed in this section are variations of the branch conditional instructions.

E.5.1 BO and BI Fields

The 5-bit BO field in branch conditional instructions encodes the following operations.

- Decrement count register (CTR)
- Test CTR equal to zero
- Test CTR not equal to zero
- Test condition true
- Test condition false
- Branch prediction (taken, fall through)

The 5-bit BI field in branch conditional instructions specifies which of the 32 bits in the CR represents the condition to test.

To provide a simplified mnemonic for every possible combination of BO and BI fields would require 2^{10} = 1024 mnemonics and most of these would be only marginally useful. The abbreviated set found in *Appendix E.5.2 Basic Branch Mnemonics*, is intended to cover the most useful cases. Unusual cases can be coded using a basic branch conditional mnemonic (**bc**, **bclr**, **bcctr**) with the condition to be tested specified as a numeric operand.

E.5.2 Basic Branch Mnemonics

The mnemonics in *Table E-6* allow all the common BO operand encodings to be specified as part of the mnemonic, along with the absolute address (AA), and set link register (LR) bits.

Notice that there are no simplified mnemonics for relative and absolute unconditional branches. For these, the basic mnemonics **b**, **ba**, **bl**, and **bla** are used.

Table E-6 provides the abbreviated set of simplified mnemonics for the most commonly performed conditional branches.

Table E-6. Simplified Branch Mnemonics

Branch Semantics	LR Update Not Enabled				LR Update Enabled			
	bc Relative	**bca** Absolute	**bclr** to LR	**bcctr** to CTR	**bcl** Relative	**bcla** Absolute	**bclrl** to LR	**bcctrl** to CTR
Branch unconditionally	—	—	**blr**	**bctr**	—	—	**blrl**	**bctrl**
Branch if condition true	**bt**	**bta**	**btlr**	**btctr**	**btl**	**btla**	**btlrl**	**btctrl**
Branch if condition false	**bf**	**bfa**	**bflr**	**bfctr**	**bfl**	**bfla**	**bflrl**	**bfctrl**
Decrement CTR, branch if CTR non-zero	**bdnz**	**bdnza**	**bdnzlr**	—	**bdnzl**	**bdnzla**	**bdnzlrl**	—
Decrement CTR, branch if CTR non-zero AND condition true	**bdnzt**	**bdnzta**	**bdnztlr**	—	**bdnztl**	**bdnztla**	**bdnztlrl**	—
Decrement CTR, branch if CTR non-zero AND condition false	**bdnzf**	**bdnzfa**	**bdnzflr**	—	**bdnzfl**	**bdnzfla**	**bdnzflrl**	—
Decrement CTR, branch if CTR zero	**bdz**	**bdza**	**bdzlr**	—	**bdzl**	**bdzla**	**bdzlrl**	—
Decrement CTR, branch if CTR zero AND condition true	**bdzt**	**bdzta**	**bdztlr**	—	**bdztl**	**bdztla**	**bdztlrl**	—
Decrement CTR, branch if CTR zero AND condition false	**bdzf**	**bdzfa**	**bdzflr**	—	**bdzfl**	**bdzfla**	**bdzflrl**	—

The simplified mnemonics shown in *Table E-6* that test a condition require a corresponding CR bit as the first operand of the instruction. The symbols defined in *Appendix E.1 Symbols* can be used in the operand in place of a numeric value.

The simplified mnemonics found in *Table E-6* are used in the following examples:

Decrement CTR and branch if it is still nonzero (closure of a loop controlled by a count loaded into CTR).

bdnz target (equivalent to **bc 16,0,**target)

Same as (1) but branch only if CTR is non-zero and condition in CR0 is “equal.”

bdnzt eq,target (equivalent to **bc 8,2,**target)

Same as (2), but “equal” condition is in CR5.

bdnzt 4 * cr5 + eq,target (equivalent to **bc 8,22,**target)

Branch if bit 27 of CR is false.

bf 27,target (equivalent to **bc 4,27,**target)

Same as (4), but set the link register. This is a form of conditional call.

bfl 27,target (equivalent to **bcl 4,27,**target)

Table E-7 provides the simplified mnemonics for the **bc** and **bca** instructions without link register updating, and the syntax associated with these instructions.

Note: The default condition register specified by the simplified mnemonics in the table is CR0.

Table E-7. Simplified Branch Mnemonics for bc and bca Instructions without Link Register Update

Branch Semantics	LR Update Not Enabled			
	bc Relative	Simplified Mnemonic	**bca** Absolute	Simplified Mnemonic
Branch unconditionally	—	—	—	—
Branch if condition true	**bc** 12,0,target	**bt** 0,target	**bca** 12,0,target	**bta** 0,target
Branch if condition false	**bc** 4,0,target	**bf** 0,target	**bca** 4,0,target	**bfa** 0,target
Decrement CTR, branch if CTR nonzero	**bc** 16,0,target	**bdnz** target	**bca** 16,0,target	**bdnza** target
Decrement CTR, branch if CTR nonzero AND condition true	**bc** 8,0,target	**bdnzt** 0,target	**bca** 8,0,target	**bdnzta** 0,target
Decrement CTR, branch if CTR nonzero AND condition false	**bc** 0,0,target	**bdnzf** 0,target	**bca** 0,0,target	**bdnzfa** 0,target
Decrement CTR, branch if CTR zero	**bc** 18,0,target	**bdz** target	**bca** 18,0,target	**bdza** target
Decrement CTR, branch if CTR zero AND condition true	**bc** 10,0,target	**bdzt** 0,target	**bca** 10,0,target	**bdzta** 0,target
Decrement CTR, branch if CTR zero AND condition false	**bc** 2,0,target	**bdzf** 0,target	**bca** 2,0,target	**bdzfa** 0,target

Table E-8 provides the simplified mnemonics for the **bclr** and **bcclr** instructions without link register updating, and the syntax associated with these instructions. Note that the default condition register specified by the simplified mnemonics in the table is CR0.

*Table E-8. Simplified Branch Mnemonics for **bclr** and **bcclr** Instructions without Link Register Update*

Branch Semantics	LR Update Not Enabled			
	bclr to LR	Simplified Mnemonic	**bcctr** to CTR	Simplified Mnemonic
Branch unconditionally	**bclr** 20,0	**blr**	**bcctr** 20,0	**bctr**
Branch if condition true	**bclr** 12,0	**btlr** 0	**bcctr** 12,0	**btctr** 0
Branch if condition false	**bclr** 4,0	**bflr** 0	**bcctr** 4,0	**bfctr** 0
Decrement CTR, branch if CTR nonzero	**bclr** 16,0	**bdnzlr**	—	—
Decrement CTR, branch if CTR nonzero AND condition true	**bclr** 10,0	**bdztlr** 0	—	—
Decrement CTR, branch if CTR nonzero AND condition false	**bclr** 0,0	**bdnzflr** 0	—	—
Decrement CTR, branch if CTR zero	**bclr** 18,0	**bdzlr**	—	—
Decrement CTR, branch if CTR zero AND condition true	**bclr** 10,0	**bdztlr** 0	—	—
Decrement CTR, branch if CTR zero AND condition false	**bcctr** 0,0	**bdzflr** 0	—	—

Table E-9 provides the simplified mnemonics for the **bcl** and **bcla** instructions with link register updating, and the syntax associated with these instructions. Note that the default condition register specified by the simplified mnemonics in the table is CR0.

*Table E-9. Simplified Branch Mnemonics for **bcl** and **bcla** Instructions with Link Register Update*

Branch Semantics	LR Update Enabled			
	bcl Relative	Simplified Mnemonic	**bcla** Absolute	Simplified Mnemonic
Branch unconditionally	—	—	—	—
Branch if condition true	**bcl** 1 2,0,target	**btl** 0,target	**bcla** 12,0,target	**btla** 0,target
Branch if condition false	**bcl** 4,0,target	**bfl** 0,target	**bcla** 4,0,target	**bfla** 0,target
Decrement CTR, branch if CTR nonzero	**bcl** 16,0,target	**bdnzl** target	**bcla** 16,0,target	**bdnzla** target
Decrement CTR, branch if CTR nonzero AND condition true	**bcl** 8,0,target	**bdnztl** 0,target	**bcla** 8,0,target	**bdnztla** 0,target
Decrement CTR, branch if CTR nonzero AND condition false	**bcl** 0,0,target	**bdnzfl** 0,target	**bcla** 0,0,target	**bdnzfla** 0,target
Decrement CTR, branch if CTR zero	**bcl** 18,0,target	**bdzl** target	**bcla** 18,0,target	**bdzla** target
Decrement CTR, branch if CTR zero AND condition true	**bcl** 10,0,target	**bdztl** 0,target	**bcla** 10,0,target	**bdztla** 0,target
Decrement CTR, branch if CTR zero AND condition false	**bcl** 2,0,target	**bdzfl** 0,target	**bcla** 2,0,target	**bdzfla** 0,target

Table E-10 provides the simplified mnemonics for the **bclrl** and **bcctrl** instructions with link register updating, and the syntax associated with these instructions. Note that the default condition register specified by the simplified mnemonics in the table is CR0.

Table E-10. Simplified Branch Mnemonics for bclrl and bcctrl Instructions with Link Register Update

Branch Semantics	LR Update Enabled			
	bclrl to LR	Simplified Mnemonic	**bcctrl** to CTR	Simplified Mnemonic
Branch unconditionally	**bclrl** 20,0	**blrl**	**bcctrl** 20,0	**bctrl**
Branch if condition true	**bclrl** 12,0	**btlrl** 0	**bcctrl** 12,0	**btctrl** 0
Branch if condition false	**bclrl** 4,0	**bflrl** 0	**bcctrl** 4,0	**bfctrl** 0
Decrement CTR, branch if CTR nonzero	**bclrl** 16,0	**bdnzlrl**	—	—
Decrement CTR, branch if CTR nonzero AND condition true	**bclrl** 8,0	**bdnztlrl** 0	—	—
Decrement CTR, branch if CTR nonzero AND condition false	**bclrl** 0,0	**bdnzflrl** 0	—	—
Decrement CTR, branch if CTR zero	**bclrl** 18,0	**bdzlrl**	—	—
Decrement CTR, branch if CTR zero AND condition true	**bdztlrl** 0	**bdztlrl** 0	—	—
Decrement CTR, branch if CTR zero AND condition false	**bclrl** 4,0	**bflrl** 0	—	—

E.5.3 Branch Mnemonics Incorporating Conditions

The mnemonics defined in *Table E-6* are variations of the branch if condition true and branch if condition false BO encodings, with the most useful values of BI represented in the mnemonic rather than specified as a numeric operand.

A standard set of codes (shown in *Table E-11*) has been adopted for the most common combinations of branch conditions.

Table E-11. Standard Coding for Branch Conditions

Code	Description
lt	Less than
le	Less than or equal
eq	Equal
ge	Greater than or equal
gt	Greater than
nl	Not less than
ne	Not equal
ng	Not greater than
so	Summary overflow
ns	Not summary overflow
un	Unordered (after floating-point comparison)
nu	Not unordered (after floating-point comparison)

Table E-12 shows the simplified branch mnemonics incorporating conditions.

Table E-12. Simplified Branch Mnemonics with Comparison Conditions

Branch Semantics	LR Update Not Enabled				LR Update Enabled			
	bc Relative	**bca** Absolute	**bclr** to LR	**bcctr** to CTR	**bcl** Relative	**bcla** Absolute	**bclrl** to LR	**bcctrl** to CTR
Branch if less than	**blt**	**blta**	**bltlr**	**bltctr**	**bltl**	**bltla**	**bltlrl**	**bltctrl**
Branch if less than or equal	**ble**	**blea**	**blelr**	**blectr**	**blel**	**blela**	**blelrl**	**blectrl**
Branch if equal	**beq**	**beqa**	**beqlr**	**beqctr**	**beql**	**beqla**	**beqlrl**	**beqctrl**
Branch if greater than or equal	**bge**	**bgea**	**bgelr**	**bgectr**	**bgel**	**bgela**	**bgelrl**	**bgectrl**
Branch if greater than	**bgt**	**bgta**	**bgtlr**	**bgtctr**	**bgtl**	**bgtla**	**bgtlrl**	**bgtctrl**
Branch if not less than	**bnl**	**bnla**	**bnllr**	**bnlctr**	**bnll**	**bnlla**	**bnllrl**	**bnlctrl**
Branch if not equal	**bne**	**bnea**	**bnelr**	**bnectr**	**bnel**	**bnela**	**bnelrl**	**bnectrl**
Branch if not greater than	**bng**	**bnga**	**bnglr**	**bngctr**	**bngl**	**bngla**	**bnglrl**	**bngctrl**
Branch if summary overflow	**bso**	**bsoa**	**bsolr**	**bsoctr**	**bsol**	**bsola**	**bsolrl**	**bsoctrl**

Table E-12. Simplified Branch Mnemonics with Comparison Conditions (Continued)

Branch Semantics	LR Update Not Enabled				LR Update Enabled			
	bc Relative	**bca** Absolute	**bclr** to LR	**bcctr** to CTR	**bcl** Relative	**bcla** Absolute	**bclrl** to LR	**bcctrl** to CTR
Branch if not summary overflow	**bns**	**bnsa**	**bnslr**	**bnsctr**	**bnsl**	**bnsla**	**bnslrl**	**bnsctrl**
Branch if unordered	**bun**	**buna**	**bunlr**	**bunctr**	**bunl**	**bunla**	**bunlrl**	**bunctrl**
Branch if not unordered	**bnu**	**bnua**	**bnulr**	**bnuctr**	**bnul**	**bnula**	**bnulrl**	**bnuctrl**

Instructions using the mnemonics in *Table E-12* specify the condition register field in an optional first operand. If the CR field being tested is CR0, this operand need not be specified. One of the CR field symbols defined in *Appendix E.1 Symbols* can be used for this operand.

The simplified mnemonics found in *Table E-12* are used in the following examples:

1. Branch if CR0 reflects condition “not equal.”
   bne target (equivalent to **bc 4,2,**target)
2. Same as (1) but condition is in CR3.
   bne cr3,target (equivalent to **bc 4,14,**target)
3. Branch to an absolute target if CR4 specifies “greater than,” setting the link register. This is a form of conditional “call.”
   bgtla cr4,target (equivalent to **bcla 12,17,**target)
4. Same as (3), but target address is in the CTR.
   bgtctrl cr4 (equivalent to **bcctrl 12,17**)

Table E-13 shows the simplified branch mnemonics for the **bc** and **bca** instructions without link register updating, and the syntax associated with these instructions. Note that the default condition register specified by the simplified mnemonics in the table is CR0.

*Table E-13. Simplified Branch Mnemonics for **bc** and **bca** Instructions without Comparison Conditions and Link Register Updating*

Branch Semantics	LR Update Not Enabled			
	bc Relative	Simplified Mnemonic	**bca** Absolute	Simplified Mnemonic
Branch if less than	**bc** 12,0,target	**blt** target	**bca** 12,0,target	**blta** target
Branch if less than or equal	**bc** 4,1,target	**ble** target	**bca** 4,1,target	**blea** target
Branch if equal	**bc** 12,2,target	**beq** target	**bca** 12,2,target	**beqa** target
Branch if greater than or equal	**bc** 4,0,target	**bge** target	**bca** 4,0,target	**bgea** target
Branch if greater than	**bc** 12,1,target	**bgt** target	**bca** 12,1,target	**bgta** target
Branch if not less than	**bc** 4,0,target	**bnl** target	**bca** 4,0,target	**bnla** target
Branch if not equal	**bc** 4,2,target	**bne** target	**bca** 4,2,target	**bnea** target
Branch if not greater than	**bc** 4,1,target	**bng** target	**bca** 4,1,target	**bnga** target
Branch if summary overflow	**bc** 12,3,target	**bso** target	**bca** 12,3,target	**bsoa** target

*Table E-13. Simplified Branch Mnemonics for **bc** and **bca** Instructions without Comparison Conditions and Link Register Updating*

Branch Semantics	LR Update Not Enabled			
	bc Relative	Simplified Mnemonic	**bca** Absolute	Simplified Mnemonic
Branch if not summary overflow	**bc** 4,3,target	**bns** target	**bca** 4,3,target	**bnsa** target
Branch if unordered	**bc** 12,3,target	**bun** target	**bca** 12,3,target	**buna** target
Branch if not unordered	**bc** 4,3,target	**bnu** target	**bca** 4,3,target	**bnua** target

Table E-14 shows the simplified branch mnemonics for the **bclr** and **bcctr** instructions without link register updating, and the syntax associated with these instructions.

Note: The default condition register specified by the simplified mnemonics in the table is CR0.

*Table E-14. Simplified Branch Mnemonics for **bclr** and **bcctr** Instructions without Comparison Conditions and Link Register Updating*

Branch Semantics	LR Update Not Enabled			
	bclr to LR	Simplified Mnemonic	**bcctr** to CTR	Simplified Mnemonic
Branch if less than	**bclr** 12,0	**bltlr**	**bcctr** 12,0	**bltctr**
Branch if less than or equal	**bclr** 4,1	**blelr**	**bcctr** 4,1	**blectr**
Branch if equal	**bclr** 12,2	**beqlr**	**bcctr** 12,2	**beqctr**
Branch if greater than or equal	**bclr** 4,0	**bgelr**	**bcctr** 4,0	**bgectr**
Branch if greater than	**bclr** 12,1	**bgtlr**	**bcctr** 12,1	**bgtctr**
Branch if not less than	**bclr** 4,0	**bnllr**	**bcctr** 4,0	**bnlctr**
Branch if not equal	**bclr** 4,2	**bnelr**	**bcctr** 4,2	**bnectr**
Branch if not greater than	**bclr** 4,1	**bnglr**	**bcctr** 4,1	**bngctr**
Branch if summary overflow	**bclr** 12,3	**bsolr**	**bcctr** 12,3	**bsoctr**
Branch if not summary overflow	**bclr** 4,3	**bnslr**	**bcctr** 4,3	**bnsctr**
Branch if unordered	**bclr** 12,3	**bunlr**	**bcctr** 12,3	**bunctr**
Branch if not unordered	**bclr** 4,3	**bnulr**	**bcctr** 4,3	**bnuctr**

Table E-15 shows the simplified branch mnemonics for the **bcl** and **bcla** instructions with link register updating, and the syntax associated with these instructions.

Note: The default condition register specified by the simplified mnemonics in the table is CR0.

Table E-15. Simplified Branch Mnemonics for ***bcl*** *and* ***bcla*** *Instructions with Comparison Conditions and Link Register Update*

Branch Semantics	LR Update Enabled			
	bcl Relative	Simplified Mnemonic	**bcla** Absolute	Simplified Mnemonic
Branch if less than	**bcl** 12,0,target	**bltl** target	**bcla** 12,0,target	**bltla** target
Branch if less than or equal	**bcl** 4,1,target	**blel** target	**bcla** 4,1,target	**blela** target
Branch if equal	**beql** target	**beql** target	**bcla** 12,2,target	**beqla** target
Branch if greater than or equal	**bcl** 4,0,target	**bgel** target	**bcla** 4,0,target	**bgela** target
Branch if greater than	**bcl** 12,1,target	**bgtl** target	**bcla** 12,1,target	**bgtla** target
Branch if not less than	**bcl** 4,0,target	**bnll** target	**bcla** 4,0,target	**bnlla** target
Branch if not equal	**bcl** 4,2,target	**bnel** target	**bcla** 4,2,target	**bnela** target
Branch if not greater than	**bcl** 4,1,target	**bngl** target	**bcla** 4,1,target	**bngla** target
Branch if summary overflow	**bcl** 12,3,target	**bsol** target	**bcla** 12,3,target	**bsola** target
Branch if not summary overflow	**bcl** 4,3,target	**bnsl** target	**bcla** 4,3,target	**bnsla** target
Branch if unordered	**bcl** 12,3,target	**bunl** target	**bcla** 12,3,target	**bunla** target
Branch if not unordered	**bcl** 4,3,target	**bnul** target	**bcla** 4,3,target	**bnula** target

Table E-16 shows the simplified branch mnemonics for the **bclrl** and **bcctl** instructions with link register updating, and the syntax associated with these instructions.

Note: The default condition register specified by the simplified mnemonics in the table is CR0.

Table E-16. Simplified Branch Mnemonics for ***bclrl*** *and* ***bcctl*** *Instructions with Comparison Conditions and Link Register Update*

Branch Semantics	LR Update Enabled			
	bclrl to LR	Simplified Mnemonic	**bcctrl** to CTR	Simplified Mnemonic
Branch if less than	**bclrl** 12,0	**bltlrl** 0	**bcctrl** 12,0	**bltctrl** 0
Branch if less than or equal	**bclrl** 4,1	**blelrl** 0	**bcctrl** 4,1	**blectrl** 0
Branch if equal	**bclrl** 12,2	**beqlrl** 0	**bcctrl** 12,2	**beqctrl** 0
Branch if greater than or equal	**bclrl** 4,0	**bgelrl** 0	**bcctrl** 4,0	**bgectrl** 0
Branch if greater than	**bclrl** 12,1	**bgtlrl** 0	**bcctrl** 12,1	**bgtctrl** 0
Branch if not less than	**bclrl** 4,0	**bnllrl** 0	**bcctrl** 4,0	**bnlctrl** 0
Branch if not equal	**bclrl** 4,2	**bnelrl** 0	**bcctrl** 4,2	**bnectrl** 0
Branch if not greater than	**bclrl** 4,1	**bnglrl** 0	**bcctrl** 4,1	**bngctrl** 0
Branch if summary overflow	**bclrl** 12,3	**bsolrl** 0	**bcctrl** 12,3	**bsoctrl** 0
Branch if not summary overflow	**bclrl** 4,3	**bnslrl** 0	**bcctrl** 4,3	**bnsctrl** 0
Branch if unordered	**bclrl** 12,3	**bunlrl** 0	**bcctrl** 12,3	**bunctrl** 0
Branch if not unordered	**bclrl** 4,3	**bnulrl** 0	**bcctrl** 4,3	**bnuctrl** 0

E.5.4 Branch Prediction

Software can use the “at” bits of Branch Conditional instructions to provide a hint to the processor about the behavior of the branch. If, for a given such instruction, the branch is almost always taken or almost always not taken, a suffix can be added to the mnemonic indicating the value to be used for the “at” bits.

+	Predict branch to be taken (at=’11’)
-	Predict branch not to be taken (at=’10’)

Such a suffix can be added to any Branch Conditional mnemonic, either basic or extended, that tests either the Count Register or a CR bit (but not both). Assemblers should use 0b00 as the default value for the “at” bits, indicating that software has offered no prediction.

E.5.4.1 Examples of Branch Prediction

Examples of branch prediction are as follows:

1. Branch if CR0 reflects condition “less than”, specifying that the branch should be predicted to be taken.
   blt+ target
2. Same as (1), but target address is in the Link Register and the branch should be predicted not to be taken.
   bltlr–

E.6 Simplified Mnemonics for Condition Register Logical Instructions

The condition register logical instructions, shown in *Table E-17*, can be used to set, clear, copy, or invert a given condition register bit. Simplified mnemonics are provided that allow these operations to be coded easily. Note that the symbols defined in *Appendix E.1 Symbols* can be used to identify the condition register bit.

Table E-17. Condition Register Logical Mnemonics

Operation	Simplified Mnemonic	Equivalent to
Condition register set	**crset** bx	**creqv** bx,bx,bx
Condition register clear	**crclr** bx	**crxor** bx,bx,bx
Condition register move	**crmove** bx,by	**cror** bx,by,by
Condition register not	**crnot** bx,by	**crnor** bx,by,by

Examples using the condition register logical mnemonics follow:

1. Set CR bit 25.
   crset 25 (equivalent to **creqv 25,25,25**)
2. Clear the SO bit of CR0.
   crclr so (equivalent to **crxor 3,3,3**)
3. Same as (2), but SO bit to be cleared is in CR3.
   crclr 4 * cr3 + so (equivalent to **crxor 15,15,15**)
4. Invert the EQ bit.
   crnot eq,eq (equivalent to **crnor 2,2,2**)
5. Same as (4), but EQ bit to be inverted is in CR4, and the result is to be placed into the EQ bit of CR5.
   crnot 4 * cr5 + eq, 4 * cr4 + eq (equivalent to **crnor 22,18,18**)

E.7 Simplified Mnemonics for Trap Instructions

A standard set of codes, shown in *Table E-18*, has been adopted for the most common combinations of trap conditions.

Table E-18. Standard Codes for Trap Instructions

Code	Description	TO Encoding	<	>	=	<U	>U
lt	Less than	16	1	0	0	0	0
le	Less than or equal	20	1	0	1	0	0
eq	Equal	4	0	0	1	0	0
ge	Greater than or equal	12	0	1	1	0	0
gt	Greater than	8	0	1	0	0	0
nl	Not less than	12	0	1	1	0	0
ne	Not equal	24	1	1	0	0	0
ng	Not greater than	20	1	0	1	0	0
llt	Logically less than	2	0	0	0	1	0
lle	Logically less than or equal	6	0	0	1	1	0
lge	Logically greater than or equal	5	0	0	1	0	1
lgt	Logically greater than	1	0	0	0	0	1
lnl	Logically not less than	5	0	0	1	0	1
lng	Logically not greater than	6	0	0	1	1	0
—	Unconditional	31	1	1	1	1	1

Note: The symbol “<U” indicates an unsigned less than evaluation will be performed. The symbol “>U” indicates an unsigned greater than evaluation will be performed.

The mnemonics defined in *Table E-19* are variations of trap instructions, with the most useful values of TO represented in the mnemonic rather than specified as a numeric operand.

Examples of the uses of trap mnemonics, shown in *Table E-19*, follow:

Table E-19. Trap Mnemonics

Trap Semantics	64-Bit Comparison		32-Bit Comparison	
	tdi Immediate	**td** Register	**twi** Immediate	**tw** Register
Trap unconditionally	—	—	—	**trap**
Trap if less than	**tdlti**	**tdlt**	**twlti**	**twlt**
Trap if less than or equal	**tdlei**	**tdle**	**twlei**	**twle**
Trap if equal	**tdeqi**	**tdeq**	**tweqi**	**tweq**
Trap if greater than or equal	**tdgei**	**tdge**	**twgei**	**twge**
Trap if greater than	**tdgti**	**tdgt**	**twgti**	**twgt**
Trap if not less than	**tdnli**	**tdnl**	**twnli**	**twnl**
Trap if not equal	**tdnei**	**tdne**	**twnei**	**twne**
Trap if not greater than	**tdngi**	**tdng**	**twngi**	**twng**
Trap if logically less than	**tdllti**	**tdllt**	**twllti**	**twllt**
Trap if logically less than or equal	**tdllei**	**tdlle**	**twllei**	**twlle**
Trap if logically greater than or equal	**tdlgei**	**tdlge**	**twlgei**	**twlge**
Trap if logically greater than	**tdlgti**	**tdlgt**	**twlgti**	**twlgt**
Trap if logically not less than	**tdlnli**	**tdlnl**	**twlnli**	**twlnl**
Trap if logically not greater than	**tdlngi**	**tdlng**	**twlngi**	**twlng**

1. Trap if 64-bit register **r**A is not zero.
   tdnei **r**A,**0** (equivalent to **tdi 24**,**r**A,**0**)
2. Trap if 64-bit register **r**A is not equal to **r**B.
   tdne **r**A, **r**B (equivalent to **td 24**,**r**A,**r**B)
3. Trap if **r**A, considered as a 32-bit quantity, is logically greater than 0x7FF.
   twlgti rA, 0x7FF (equivalent to **twi 1**,**r**A, 0x7FF)
4. Trap unconditionally.
   trap (equivalent to **tw 31,0,0**)

Trap instructions evaluate a trap condition as follows:

- The contents of register **r**A are compared with either the sign-extended SIMM field or the contents of register **r**B, depending on the trap instruction.
- For **tdi** and **td**, the entire contents of **r**A (and **r**B) participate in the comparison; for **twi** and **tw**, only the contents of the low- order 32 bits of **r**A (and **r**B) participate in the comparison.

The comparison results in five conditions which are ANDed with operand TO. If the result is not 0, the trap exception handler is invoked. (Note that exceptions are referred to as interrupts in the architecture specification.) See *Table E-20* for these conditions.

Table E-20. TO Operand Bit Encoding

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

E.8 Simplified Mnemonics for Special-Purpose Registers

The **mtspr** and **mfspr** instructions specify a special-purpose register (SPR) as a numeric operand. Simplified mnemonics are provided that represent the SPR in the mnemonic rather than requiring it to be coded as a numeric operand. *Table E-21* provides a list of the simplified mnemonics that should be provided by assemblers for SPR operations.

Table E-21. Simplified Mnemonics for SPRs

Special-Purpose Register	Move to SPR		Move from SPR	
	Simplified Mnemonic	Equivalent to	Simplified Mnemonic	Equivalent to
XER	**mtxer r**S	**mtspr** 1,**r**S	**mfxer r**D	**mfspr r**D**,1**
Link register	**mtlr r**S	**mtspr 8,r**S	**mflr r**D	**mfspr r**D**,8**
Count register	**mtctr r**S	**mtspr 9,r**S	**mfctr r**D	**mfspr r**D**,9**
DSISR	**mtdsisr r**S	**mtspr 18,r**S	**mfdsisr r**D	**mfspr r**D**,18**
Data address register	**mtdar r**S	**mtspr 19,r**S	**mfdar r**D	**mfspr r**D**,19**
Decrementer	**mtdec r**S	**mtspr 22,r**S	**mfdec r**D	**mfspr r**D**,22**
SDR1	**mtsdr1 r**S	**mtspr 25,r**S	**mfsdr1 r**D	**mfspr r**D**,25**
Save and restore register 0	**mtsrr0 r**S	**mtspr 26,r**S	**mfsrr0 r**D	**mfspr r**D**,26**
Save and restore register 1	**mtsrr1 r**S	**mtspr 27,r**S	**mfsrr1 r**D	**mfspr r**D**,27**
SPRG0–SPRG3	**mtspr** *n,* **r**S	**mtspr** 272 + *n*,**r**S	**mfsprg r**D*, n*	**mfspr r**D,272 + *n*
Address space register	**mtasr r**S	**mtspr 280,r**S	**mfasr r**D	**mfspr r**D**,280**
External access register	**mtear r**S	**mtspr 282,r**S	**mfear r**D	**mfspr r**D**,282**
Time base lower	**mttbl r**S	**mtspr 284,r**S	**mftb r**D	**mftb r**D**,268**
Time base upper	**mttbu r**S	**mtspr 285,r**S	**mftbu r**D	**mftb r**D**,269**
Processor version register	—	—	**mfpvr r**D	**mfspr r**D**,287**

Following are examples using the SPR simplified mnemonics found in *Table E-21*:

1. Copy the contents of the low-order 32 bits of **r**S to the XER.

mtxer rS	(equivalent to	**mtspr 1,r**S)

2. Copy the contents of the LR to **r**S.

mflr rS	(equivalent to	**mfspr r**S,**8**)

3. Copy the contents of **r**S to the CTR.

mtctr rS	(equivalent to	**mtspr 9,r**S)

E.9 Recommended Simplified Mnemonics

This section describes some of the most commonly-used operations (such as no-op, load immediate, load address, move register, and complement register).

E.9.1 No-Op (nop)

Many PowerPC instructions can be coded in a way that, effectively, no operation is performed. An additional mnemonic is provided for the preferred form of no-op. If an implementation performs any type of run-time optimization related to no-ops, the preferred form is the no-op that triggers the following:

nop	(equivalent to	**ori 0,0,0**)

E.9.2 Load Immediate (li)

The **addi** and **addis** instructions can be used to load an immediate value into a register. Additional mnemonics are provided to convey the idea that no addition is being performed but that data is being moved from the immediate operand of the instruction to a register.

1. Load a 16-bit signed immediate value into **r**D.

li rD**,**value	(equivalent to	**addi r**D**,0,**value)

2. Load a 16-bit signed immediate value, shifted left by 16 bits, into **r**D.

lis rD**,**value	(equivalent to	**addis r**D**,0,**value)

E.9.3 Load Address (la)

This mnemonic permits computing the value of a base-displacement operand, using the **addi** instruction which normally requires a separate register and immediate operands.

la rD,d(**r**A)	(equivalent to	**addi r**D**,r**A,d)

The **la** mnemonic is useful for obtaining the address of a variable specified by name, allowing the assembler to supply the base register number and compute the displacement. If the variable *v* is located at offset d*v* bytes from the address in register **r***v,* and the assembler has been told to use register **r***v* as a base for references to the data structure containing *v*, the following line causes the address of *v* to be loaded into register **r**D:

la rD,*v*	(equivalent to	**addi r**D**,r***v*,d*v)*

E.9.4 Move Register (mr)

Several PowerPC instructions can be coded to copy the contents of one register to another. A simplified mnemonic is provided that signifies that no computation is being performed, but merely that data is being moved from one register to another.

The following instruction copies the contents of **r**S into **r**A. This mnemonic can be coded with a dot (**.**) suffix to cause the Rc bit to be set in the underlying instruction.

mr rA**,r**S	(equivalent to **or r**A**,r**S**,r**S)

E.9.5 Complement Register (not)

Several PowerPC instructions can be coded in a way that they complement the contents of one register and place the result into another register. A simplified mnemonic is provided that allows this operation to be coded easily.

The following instruction complements the contents of **r**S and places the result into **r**A. This mnemonic can be coded with a dot (**.**) suffix to cause the Rc bit to be set in the underlying instruction.

not rA**,r**S	(equivalent to **nor r**A**,r**S**,r**S)

E.9.6 Move to/from Condition Register (mtcr/mfcr)

This mnemonic permits copying the contents of the low-order 32 bits of a GPR to the condition register, using the same syntax as the **mfcr** instruction.

mtcr rS	(equivalent to **mtcrf** 0xFF**,r**S)

The following instructions may generate either the (old) **mtcrf** or **mfcr** instructions or the (new) **mtocrf** or **mfocrf** instruction, respectively, depending on the target machine type assembler parameter.

mtcrf	CRM,**r**S
mfcr	**r**S

All three extended mnemonics in this subsection are being phased out. In future assemblers the form "**mtcr r**S" may not exist, and the **mtcrf** and **mfcr** mnemonics may generate the old form instructions (with bit 11 = 0) regardless of the target machine type assembler parameter, or may cease to exist.

THIS PAGE INTENTIONALLY LEFT BLANK

Appendix F. Glossary of Terms and Abbreviations

The glossary contains an alphabetical list of terms, phrases, and abbreviations used in this book. Some of the terms and definitions included in the glossary are reprinted from *IEEE Std. 754-1985, IEEE Standard for Binary Floating-Point Arithmetic*, copyright ©1985 by the Institute of Electrical and Electronics Engineers, Inc. with the permission of the IEEE.

Note that some terms are defined in the context of how they are used in this book.

A

Architecture. A detailed specification of requirements for a processor or computer system. It does not specify details of how the processor or computer system must be implemented; instead it provides a template for a family of compatible *implementations*.

Asynchronous exception. *Exceptions* that are caused by events external to the processor's execution. In this manual, the term 'asynchronous exception' is used interchangeably with the word *interrupt*.

Atomic access. A bus access that attempts to be part of a read-write operation to the same address uninterrupted by any other access to that address (the term refers to the fact that the transactions are indivisible). The PowerPC Architecture implements atomic accesses through the **lwarx**/**stwcx.** (**ldarx**/**stdcx.** in 64-bit implementations) instruction pair.

B

Biased exponent. An *exponent* whose range of values is shifted by a constant (bias). Typically a bias is provided to allow a range of positive values to express a range that includes both positive and negative values.

Big-endian. A byte-ordering method in memory where the address n of a word corresponds to the *most-significant byte*. In an addressed memory word, the bytes are ordered (left to right) 0, 1, 2, 3, with 0 being the most-significant byte. *See* Little-endian.

Boundedly undefined. A characteristic of results of certain operations that are not rigidly prescribed by the PowerPC Architecture. Boundedly undefined results for a given operation may vary among implementations, and between execution attempts in the same implementation.

Although the architecture does not prescribe the exact behavior for when results are allowed to be boundedly undefined, the results of executing instructions in contexts where results are allowed to be boundedly undefined are constrained to ones that could have been achieved by executing an arbitrary sequence of defined instructions, in valid form, starting in the state the machine was in before attempting to execute the given instruction.

C

Cache. High-speed memory component containing recently-accessed data and/or instructions (subset of main memory).

Cache block. A small region of contiguous memory that is copied from memory into a *cache*. The size of a cache block may vary among processors; the maximum block size is one *page*. In PowerPC processors, *cache coherency* is maintained on a cache-block basis. Note that the term 'cache block' is often used interchangeably with 'cache line'.

Cache coherency. An attribute wherein an accurate and common view of memory is provided to all devices that share the same memory system. Caches are coherent if a processor performing a read from its cache is supplied with data corresponding to the most recent value written to memory or to another processor's cache.

Cache flush. An operation that removes from a cache any data from a specified address range. This operation ensures that any modified data within the specified address range is written back to main memory. This operation is generated typically by a Data Cache Block Flush (**dcbf**) instruction.

Caching-inhibited. A memory update policy in which the *cache* is bypassed and the load or store is performed to or from main memory.

Cast-outs. *Cache blocks* that must be written to memory when a cache miss causes a cache block to be replaced.

Changed bit. One of two *page history bits* found in each *page table entry* (PTE). The processor sets the changed bit if any store is performed into the *page*. *See also* Page access history bits and Referenced bit.

Clear. To cause a bit or bit field to register a value of zero. *See also* Set.

Context synchronization. An operation that ensures that all instructions in execution complete past the point where they can produce an *exception*, that all instructions in execution complete in the context in which they began execution, and that all subsequent instructions are *fetched* and executed in the new context. Context synchronization may result from executing specific instructions (such as **isync** or **rfid**) or when certain events occur (such as an exception).

Copy-back. An operation in which modified data in a *cache block* is copied back to memory.

D

Denormalized number. A nonzero floating-point number whose *exponent* has a reserved value, usually the format's minimum, and whose explicit or implicit leading significand bit is zero.

Direct-mapped cache. A cache in which each main memory address can appear in only one location within the cache, operates more quickly when the memory request is a cache hit.

E

Effective address (EA). The 32 or 64-bit address specified for a load, store, or an instruction fetch. This address is then submitted to the MMU for translation to either a *physical memory* address or an I/O address.

Exception. A condition encountered by the processor that requires special, supervisor-level processing.

Exception handler. A software routine that executes when an exception is taken. Normally, the exception handler corrects the condition that caused the exception, or performs some other meaningful task (that may include aborting the program that caused the exception). The address for each exception handler is identified by an exception vector offset defined by the architecture and a prefix selected via the MSR.

Extended opcode. A secondary opcode field generally located in instruction bits [21–30], that further defines the instruction type. All PowerPC instructions are one word in length. The most significant 6 bits of the instruction are the *primary opcode*, identifying the type of instruction. *See also* Primary opcode.

Execution synchronization. A mechanism by which all instructions in execution are architecturally complete before beginning execution (appearing to begin execution) of the next instruction. Similar to context synchronization, but doesn't force the contents of the instruction buffers to be deleted and refetched.

Exponent. In the binary representation of a floating-point number, the exponent is the component that normally signifies the integer power to which the value two is raised in determining the value of the represented number. *See also* Biased exponent.

F

Fetch. Retrieving instructions from either the cache or main memory and placing them into the instruction queue.

Floating-point register (FPR). Any of the 32 registers in the floating-point register file. These registers provide the source operands and destination results for floating-point instructions. Load instructions move data from memory to FPRs and store instructions move data from FPRs to memory. The FPRs are 64 bits wide and store floating-point values in double-precision format.

Fraction. In the binary representation of a floating-point number, the field of the *significand* that lies to the right of its implied binary point.

Fully-associative. Addressing scheme where every cache location (every byte) can have any possible address.

G

General-purpose register (GPR). Any of the 32 registers in the general-purpose register file. These registers provide the source operands and destination results for all integer data manipulation instructions. Integer load instructions move data from memory to GPRs and store instructions move data from GPRs to memory.

Guarded. The guarded attribute pertains to out-of-order execution. When a page is designated as guarded, instructions and data cannot be accessed out-of-order.

H

Harvard architecture. An architectural model featuring separate caches for instruction and data.

Hashing. An algorithm used in the *page table* search process.

I

IEEE 754. A standard written by the Institute of Electrical and Electronics Engineers that defines operations and representations of binary floating-point arithmetic.

Illegal instructions. A class of instructions that are not implemented for a particular PowerPC processor. These include instructions not defined by the PowerPC Architecture. In addition, for 32-bit implementations, instructions that are defined only for 64-bit implementations are considered to be illegal instructions. For 64-bit implementations instructions that are defined only for 32-bit implementations are considered to be illegal instructions.

Implementation. A particular processor that conforms to the PowerPC Architecture, but may differ from other architecture-compliant implementations for example in design, feature set, and implementation of *optional* features. The PowerPC Architecture has many different implementations.

Implementation-dependent. An aspect of a feature in a processor's design that is defined by a processor's design specifications rather than by the PowerPC Architecture.

Implementation-specific. An aspect of a feature in a processor's design that is not required by the PowerPC Architecture, but for which the PowerPC Architecture may provide concessions to ensure that processors that implement the feature do so consistently.

Imprecise exception. A type of *synchronous exception* that is allowed not to adhere to the precise exception model (*see* Precise exception). The PowerPC Architecture allows only floating-point exceptions to be handled imprecisely.

Inexact. Loss of accuracy in an arithmetic operation when the rounded result differs from the infinitely precise value with unbounded range.

In-order. An aspect of an operation that adheres to a sequential model. An operation is said to be performed in-order if, at the time that it is performed, it is known to be required by the sequential execution model. *See* Out-of-order.

Instruction latency. The total number of clock cycles necessary to execute an instruction and make ready the results of that instruction.

Instruction parallelism. A feature of PowerPC processors that allows instructions to be processed in parallel.

Interrupt. An *asynchronous exception*. On PowerPC processors, interrupts are a special case of exceptions. *See also* asynchronous exception.

Invalid state. State of a cache entry that does not currently contain a valid copy of a cache block from memory.

K

Key bits. A set of key bits referred to as Ks and Kp in each SLB entry. The key bits determine whether supervisor or user programs can access a *page* within that *segment*.

Kill. An operation that causes a *cache block* to be invalidated.

L

L2 cache. *See* Secondary cache.

Least-significant bit (lsb). The bit of least value in an address, register, data element, or instruction encoding.

Least-significant byte (LSB). The byte of least value in an address, register, data element, or instruction encoding.

Little-endian. A byte-ordering method in memory where the address *n* of a word corresponds to the *least-significant byte*. In an addressed memory word, the bytes are ordered (left to right) 3, 2, 1, 0, with 3 being the *most-significant byte*. *See* Big-endian.

Loop unrolling. Loop unrolling provides a way of increasing performance by allowing more instructions to be issued in a clock cycle. The compiler replicates the loop body to increase the number of instructions executed between a loop branch.

M

MESI (modified/exclusive/shared/invalid). *Cache coherency* protocol used to manage caches on different devices that share a memory system. Note that the PowerPC Architecture does not specify the implementation of a MESI protocol to ensure cache coherency.

Memory access ordering. The specific order in which the processor performs load and store memory accesses and the order in which those accesses complete.

Memory-mapped accesses. Accesses whose addresses use the page address translation mechanisms provided by the MMU and that occur externally with the bus protocol defined for memory.

Memory coherency. An aspect of caching in which it is ensured that an accurate view of memory is provided to all devices that share system memory.

Memory consistency. Refers to agreement of levels of memory with respect to a single processor and system memory (for example, on-chip cache, secondary cache, and system memory).

Memory management unit (MMU). The functional unit that is capable of translating an *effective* (logical) *address* to a physical address, providing protection mechanisms, and defining caching methods.

Microarchitecture. The hardware details of a microprocessor's design. Such details are not defined by the PowerPC Architecture.

Mnemonic. The abbreviated name of an instruction used for coding.

Modified state. When a cache block is in the modified state, it has been modified by the processor since it was copied from memory. *See* MESI.

Munging. A modification performed on an *effective address* that allows it to appear to the processor that individual aligned scalars are stored as *little-endian* values, when in fact it is stored in *big-endian* order, but at different byte addresses within doublewords.

Note: Munging affects only the effective address and not the byte order. This term is not used by the PowerPC Architecture.

Multiprocessing. The capability of software, especially operating systems, to support execution on more than one processor at the same time.

Most-significant bit (msb). The highest-order bit in an address, registers, data element, or instruction encoding.

Most-significant byte (MSB). The highest-order byte in an address, registers, data element, or instruction encoding.

N

NaN. An abbreviation for 'Not a Number'; a symbolic entity encoded in floating-point format. There are two types of NaNs—signaling NaNs (SNaNs) and quiet NaNs (QNaNs).

No-op. No-operation. A single-cycle operation that does not affect registers or generate bus activity.

Normalization. A process by which a floating-point value is manipulated such that it can be represented in the format for the appropriate precision (single or double-precision). For a floating-point value to be representable in the single or double-precision format, the leading implied bit must be a 1.

O

OEA (Operating Environment Architecture). The level of the architecture that describes the PowerPC memory management model, supervisor-level registers, synchronization requirements, and the exception model. It also defines the time-base feature from a supervisor-level perspective. Implementations that conform to the PowerPC OEA also conform to the PowerPC UISA and VEA.

Optional. A feature, such as an instruction, a register, or an exception, that is defined by the PowerPC Architecture but not required to be implemented.

Out-of-order. An aspect of an operation that allows it to be performed ahead of one that may have preceded it in the sequential model, for example, speculative operations. An operation is said to be performed out-of-order if, at the time that it is performed, it is not known to be required by the sequential execution model. *See* In-order.

Out-of-order execution. A technique that allows instructions to be issued and completed in an order that differs from their sequence in the instruction stream.

Overflow. An error condition that occurs during arithmetic operations when the result cannot be stored accurately in the destination register(s). For example, if two 32-bit numbers are multiplied, the result may not be representable in 32 bits.

P

Page. A region in memory. The OEA defines a page as a 4-Kbyte area of memory, aligned on a 4-Kbyte boundary or a large page size which is implementation dependent.

Page access history bits. The *changed* and *referenced* bits in the PTE keep track of the access history within the page. The referenced bit is set by the MMU whenever the page is accessed for a read or write operation. The processor sets the changed bit if any store is performed into the *page*. *See* Changed bit and Referenced bit.

Page fault. A page fault is a condition that occurs when the processor attempts to access a memory location that does not reside within a *page* not currently resident in *physical memory*. On PowerPC processors, a page fault exception condition occurs when a matching, valid *page table entry* (PTE[V] = 1) cannot be located.

Page table. A table in memory is comprised of *page table entries*, or PTEs. It is further organized into eight PTEs per PTEG (page table entry group). The number of PTEGs in the page table depends on the size of the page table (as specified in the SDR1 register).

Page table entry (PTE). A 16-byte data structure containing information used to translate a virtual page address to a physical page address. A page is either 4 KB or an implementation-specific sized large page.

Physical memory. The actual memory that can be accessed through the system's memory bus.

Pipelining. A technique that breaks operations, such as instruction processing or bus transactions, into smaller distinct stages or tenures (respectively) so that a subsequent operation can begin before the previous one has completed.

Precise exceptions. A category of exception for which the pipeline can be stopped so instructions that preceded the faulting instruction can complete, and subsequent instructions can be flushed and redispatched after exception handling has completed. *See* Imprecise exceptions.

Primary opcode. The most-significant 6 bits (bits [0–5]) of the instruction encoding that identifies the type of instruction. S*ee* Secondary opcode.

Protection boundary. A boundary between *protection domains*.

Protection domain. A protection domain is a segment, a virtual page, or a range of unmapped effective addresses. It is defined only when the appropriate relocate bit in the MSR ([IR] or [DR]) is '1'.

Q

Quad word. A group of 16 contiguous locations starting at an address divisible by 16.

Quiet NaN. A type of *NaN* that can propagate through most arithmetic operations without signaling exceptions. A quiet NaN is used to represent the results of certain invalid operations, such as invalid arithmetic operations on infinities or on NaNs, when invalid. *See* Signaling NaN.

R

rA. The **r**A instruction field is used to specify a GPR to be used as a source or destination.

rB. The **r**B instruction field is used to specify a GPR to be used as a source.

rD. The **r**D instruction field is used to specify a GPR to be used as a destination.

rS. The **r**S instruction field is used to specify a GPR to be used as a source.

Real address mode. An MMU mode when no address translation is performed and the *effective address* specified is the same as the physical address. The processor's MMU is operating in real address mode if its ability to perform address translation has been disabled through the MSR registers IR and/or DR bits.

Record bit. Bit [31] (or the Rc bit) in the instruction encoding. When it is set, updates the condition register (CR) to reflect the result of the operation.

Referenced bit. One of two *page history bits* found in each *page table entry* (PTE). The processor sets the *referenced bit* whenever the page is accessed for a read or write. *See also* Page access history bits.

Register indirect addressing. A form of addressing that specifies one GPR that contains the address for the load or store.

Register indirect with immediate index addressing. A form of addressing that specifies an immediate value to be added to the contents of a specified GPR to form the target address for the load or store.

Register indirect with index addressing. A form of addressing that specifies that the contents of two GPRs be added together to yield the target address for the load or store.

Reservation. The processor establishes a reservation on a *cache block* of memory space when it executes an **lwarx** or **ldarx** instruction to read a memory semaphore into a GPR.

Reserved field. In a register, a reserved field is one that is not assigned a function. A reserved field may be a single bit. The handling of reserved bits is *implementation-dependent*. Software is permitted to write any value to such a bit. A subsequent reading of the bit returns 0 if the value last written to the bit was 0 and returns an undefined value ('0' or '1') otherwise.

RISC (Reduced Instruction Set Computing). An *architecture* characterized by fixed-length instructions with nonoverlapping functionality and by a separate set of load and store instructions that perform memory accesses.

S

SLB (Segment Lookaside Buffer). An optional cache that holds recently-used *segment table entries*.

Scalability. The capability of an architecture to generate *implementations* specific for a wide range of purposes, and in particular implementations of significantly greater performance and/or functionality than at present, while maintaining compatibility with current implementations.

Secondary cache. A cache memory that is typically larger and has a longer access time than the primary cache. A secondary cache may be shared by multiple devices. Also referred to as L2, or level-2 cache.

Segment. A 256-Mbyte area of *virtual memory* that is the most basic memory space defined by the PowerPC Architecture. Each segment is configured through a unique *segment descriptor*.

Segment descriptors. Information used to generate the interim *virtual address*. The segment descriptors reside as segment table entries in a hashed segment table in memory.

Segment table entry (STE). Data structures containing information used to translate *effective address* to physical address. STEs are implemented on 64-bit processors only.

Set (*v*). To write a nonzero value to a bit or bit field; the opposite of *clear*. The term 'set' may also be used to generally describe the updating of a bit or bit field.

Set (*n*). A subdivision of a *cache*. Cacheable data can be stored in a given location in any one of the sets, typically corresponding to its lower-order address bits. Because several memory locations can map to the same location, cached data is typically placed in the set whose *cache block* corresponding to that address was used least recently. *See* Set-associative.

Set-associative. Aspect of cache organization in which the cache space is divided into sections, called *sets*. The cache controller associates a particular main memory address with the contents of a particular set, or region, within the cache.

Signaling NaN. A type of *NaN* that generates an invalid operation program exception when it is specified as arithmetic operands. *See* Quiet NaN.

Significand. The component of a binary floating-point number that consists of an explicit or implicit leading bit to the left of its implied binary point and a fraction field to the right.

Simplified mnemonics. Assembler mnemonics that represent a more complex form of a common operation.

Static branch prediction. Mechanism by which software (for example, compilers) can give a hint to the machine hardware about the direction a branch is likely to take.

Sticky bit. A bit that when set must be cleared explicitly.

Strong ordering. A memory access model that requires exclusive access to an address before making an update, to prevent another device from using stale data.

Superscalar machine. A machine that can issue multiple instructions concurrently from a conventional linear instruction stream.

Supervisor mode. The privileged operation state of a processor. In supervisor mode, software, typically the operating system, can access all control registers and can access the supervisor memory space, among other privileged operations.

Synchronization. A process to ensure that operations occur strictly *in order*. *See* Context synchronization and Execution synchronization.

Synchronous exception. An *exception* that is generated by the execution of a particular instruction or instruction sequence. There are two types of synchronous exceptions, *precise* and *imprecise*.

System memory. The physical memory available to a processor.

T

TLB (translation lookaside buffer) A cache that holds recently-used *page table entries*.

Throughput. The measure of the number of instructions that are processed per clock cycle.

Tiny. A floating-point value that is too small to be represented for a particular precision format, including *denormalized* numbers; they do not include ±0.

U

UISA (user instruction set architecture). The level of the architecture to which user-level software should conform. The UISA defines the base user-level instruction set, user-level registers, data types, floating-point memory conventions and exception model as seen by user programs, and the memory and programming models.

Underflow. An error condition that occurs during arithmetic operations when the result cannot be represented accurately in the destination register. For example, underflow can happen if two floating-point fractions are multiplied and the result requires a smaller *exponent* and/or mantissa than the single-precision format can provide. In other words, the result is too small to be represented accurately.

Unified cache. Combined data and instruction cache.

User mode. The unprivileged operating state of a processor used typically by application software. In user mode, software can only access certain control registers and can access only user memory space. No privileged operations can be performed. Also referred to as problem state.

V

VEA (virtual environment architecture). The level of the *architecture* that describes the memory model for an environment in which multiple devices can access memory, defines aspects of the cache model, defines cache control instructions, and defines the time-base facility from a user-level perspective. *Implementations* that conform to the PowerPC VEA also adhere to the UISA, but may not necessarily adhere to the OEA.

Virtual address. An intermediate address used in the translation of an *effective address* to a physical address.

Virtual memory. The address space created using the memory management facilities of the processor. Program access to virtual memory is possible only when it coincides with *physical memory*.

W

Weak ordering. A memory access model that allows bus operations to be reordered dynamically, which improves overall performance and in particular reduces the effect of memory latency on instruction throughput.

Word. A 32-bit data element.

Write-back. A cache memory update policy in which processor write cycles are directly written only to the cache. External memory is updated only indirectly, for example, when a modified cache block is *cast out* to make room for newer data.

Write-through. A cache memory update policy in which all processor write cycles are written to both the cache and memory.

Revision Log

Revision Date	Page Affected	Contents of Modification
July 15, 2005		Version 3.0 • Removed 32-bit implementation information • Removed obsolete instructions: **dcbi**, **mcrxr, mtsrd, mtsrdin**, **rfi**
March 31, 2005	486, 506, 544, 545, 546, 601, 386, , 512, 513, 529	Version 2.23 Updates to include changes to the PowerPC Architecture 2.01 (from PowerPC Architecture 1.10). This includes the addition of the following instructions: • **mfocrf**, **mtocrf**, **slbmfee**, **slbmfev**, **slbmte**, **tlbiel** The following instructions are considered obsolete in the PowerPC Architecture (2.01), however they are presented in this version: • **dcbi**, **mcrxr, mtsrd, mtsrdin**, **rfi** The following instruction is considered obsolete in the PowerPC Architecture (2.01) and has been deleted from this manual: • **dcba**