MOTOROLA
intelligence everywhere™

digital dna™

MPCFPE32B/AD
12/2001
REV 2

# Programming Environments Manual

## For 32-Bit Implementations of the PowerPC Architecture

MOTOROLA

# Contents

# Contents

# Contents

# Contents

# Contents

| Paragraph Number | Title | Page Number |
|---|---|---|

# Contents

**Paragraph Number** | **Title** | **Page Number**

# Contents

# Contents

**Paragraph Number** | **Title** | **Page Number**

# Contents

# Contents

# Contents

# Contents

**Paragraph Number** | **Title** | **Page Number**

# Figures

# Figures

# Figures

# Figures

# Tables

# Tables

# Tables

# Tables

# Tables

# Tables

# About This Book

The primary objective of this manual is to help programmers provide software that is compatible across a variety of implementations. 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 these processors and indicates those features that are optional or that may be implemented differently in the design of each processor.

This revision of this book describes only the 32-bit portion of the PowerPC architecture in detail.

Locate errata or updates for this document at http://www.motorola.com/semiconductors.

For designers working with a specific processor, this book should be used in conjunction with the user's manual for that processor.

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

- User instruction set architecture (UISA)—The UISA defines 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, memory conventions, and the memory and programming models seen by application programmers.
- 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. The VEA describes the memory model for an environment in which multiple processors or other devices can access external memory and defines aspects of the cache model and cache control instructions from a user-level perspective. VEA resources are particularly useful for optimizing memory accesses and for managing resources in an environment in which other processors and other devices can access external memory.

  Implementations that conform to the VEA also conform to the UISA but may not necessarily adhere to the OEA.
- Operating environment architecture (OEA)—The OEA defines supervisor-level resources typically required by an operating system. It defines the memory management model, supervisor-level registers, and the exception model.

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

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

Because it is important to distinguish between the levels of the architecture to ensure compatibility across multiple platforms, those distinctions are shown clearly throughout this book. The level of the architecture to which text refers is indicated in the outer margin, using the conventions shown in “Conventions,” on Page 5.

For ease in reference, topics in the user’s manuals are presented in the same order in this book. Topics build upon one another, beginning with a description and complete summary of the MPC750 programming model (registers and instructions) and progressing to more specific, architecture-based topics regarding 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.

*The PowerPC Architecture: A Specification for a New Family of RISC Processors* defines the architecture from the perspective of the three programming environments and remains the defining document for the PowerPC architecture. .

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

## Audience

It is assumed that the reader 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, “Register Set,” is useful for software engineers who need to understand the PowerPC programming model for the three programming environments and the functionality of each register.

- Chapter 3, “Operand Conventions,” describes 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 addressing modes and a description of the 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 address translation and memory protection mechanism as defined by the OEA.
- Chapter 8, “Instruction Set,” functions as a handbook for the instruction set. Instructions are sorted by mnemonic. Each instruction description includes the instruction formats and an individualized legend that provides such information as the level or levels of the architecture in which the instruction may be found and the privilege level of the instruction.
- Appendix A, “Instruction Set Listings,” describes each instruction in detail. Instructions are grouped according to mnemonic, opcode, function, and form.
- Appendix B, “POWER Architecture Cross-Reference,” identifies the differences that must be managed in migration from the POWER architecture to the architecture.
- Appendix C, “Multiple-Precision Shifts,” describes how multiple-precision shift operations can be programmed as defined by the UISA.
- Appendix D, “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 E, “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 F, “Simplified Mnemonics,” provides a set of simplified mnemonic examples and symbols.
- Appendix G, “Programming Environments Manual (32-Bit) Revision History,” describes major changes since the previous revision of this document.
- This manual also includes a glossary and an index.

## Suggested Reading

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

## General Information

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

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

  For updates to the specification, see http://www.austin.ibm.com/tech/ppc-chg.html.
- *PowerPC Microprocessor Common Hardware Reference Platform: A System Architecture*, by Apple Computer, Inc., International Business Machines, Inc., and Motorola, Inc.
- *Computer Architecture: A Quantitative Approach*, Second Edition, by John L. Hennessy and David A. Patterson
- *Computer Organization and Design: The Hardware/Software Interface*, Second Edition, David A. Patterson and John L. Hennessy

## Related Documentation

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

- User's manuals—These books provide details about individual implementations and are intended for use with the *Programming Environments Manual*.
- Addenda/errata to user's manuals—Because some processors have follow-on parts an addendum is provided that describes the additional features and functionality changes. These addenda are intended for use with the corresponding user's manuals.
- Hardware specifications—Hardware specifications provide specific data regarding bus timing, signal behavior, and AC, DC, and thermal characteristics, as well as other design considerations.
- Technical summaries—Each device has a technical summary that provides an overview of its features. This document is roughly the equivalent to the overview (Chapter 1) of an implementation's user's manual.
- *The Programmer's Reference Guide for the PowerPC Architecture*: MPCPRG/D—This concise reference includes the register summary, memory control model, exception vectors, and the instruction set.
- *The Programmer's Pocket Reference Guide for the PowerPC Architecture*: MPCPRGREF/D—This foldout card provides an overview of registers, instructions, and exceptions for 32-bit implementations.
- Application notes—These short documents address specific design issues useful to programmers and engineers working with Motorola processors.

Additional literature is published as new processors become available. For a current list of documentation, refer to http://www.motorola.com/semiconductors.

## Conventions

This document uses the following notational conventions:

| | |
|---|---|
| cleared/set | When a bit takes the value zero, it is said to be cleared; when it takes a value of one, it is said to be set. |
| **mnemonics** | Instruction mnemonics are shown in lowercase bold. |
| *italics* | Italics indicate variable command parameters, for example, **bcctr***x*. |
| | Book titles in text are set in italics |
| | Internal signals are set in italics, for example, $\overline{\textit{qual BG}}$ |
| 0x0 | Prefix to denote hexadecimal number |
| 0b0 | Prefix to denote binary number |
| **r**A, **r**B | Instruction syntax used to identify a source GPR |
| **r**D | Instruction syntax used to identify a destination GPR |
| **fr**A, **fr**B, **fr**C | Instruction syntax used to identify a source FPR |
| **fr**D | Instruction syntax used to identify a destination FPR |
| REG[FIELD] | Abbreviations for registers are shown in uppercase text. Specific bits, fields, or ranges appear in brackets. For example, MSR[LE] refers to the little-endian mode enable bit in the machine state register. |
| x | In some contexts, such as signal encodings, an unitalicized x indicates a don't care. |
| *x* | An italicized *x* indicates an alphanumeric variable. |
| *n* | An italicized *n* indicates an numeric variable. |
| ¬ | NOT logical operator |
| & | AND logical operator |
| \| | OR logical operator |
| ▨ 0000 | Indicates reserved bits or bit fields in a register. Although these bits can be written to as ones or zeros, they are always read as zeros. |
| U | This symbol identifies text that is relevant with respect to the user instruction set architecture (UISA). This symbol is used both for information that can be found in the UISA specification as well as for explanatory information related to that programming environment. |
| ▼V | This symbol identifies text that is relevant with respect to the virtual environment architecture (VEA). This symbol is used both for |

information that can be found in the VEA specification as well as for explanatory information related to that programming environment.

Ⓞ This symbol identifies text that is relevant with respect to the operating environment architecture (OEA). This symbol is used both for information that can be found in the OEA specification as well as for explanatory information related to that programming environment.

Additional conventions used with instruction encodings are described in Table 8-2. Conventions used for pseudocode examples are described in Table 8-3.

## Acronyms and Abbreviations

Table i contains acronyms and abbreviations that are used in this document. 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 |
| BAT | Block address translation |
| 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 |
| DBAT | Data BAT |
| 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 |
| FPECR | Floating-point exception cause register |
| FPR | Floating-point register |
| FPSCR | Floating-point status and control register |

**Table i. Acronyms and Abbreviated Terms (continued)**

| Term | Meaning |
|---|---|
| FPU | Floating-point unit |
| GPR | General-purpose register |
| IBAT | Instruction BAT |
| 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 |
| MESI | Modified/exclusive/shared/invalid—cache coherency protocol |
| MMU | Memory management unit |
| MSB | Most-significant byte |
| msb | Most-significant bit |
| MSR | Machine state register |
| NaN | Not a number |
| NIA | Next instruction address |
| No-op | No operation |
| OEA | Operating environment architecture |
| 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 |
| SIMM | Signed immediate value |
| SLB | Segment lookaside buffer |
| SPR | Special-purpose register |
| SPRG*n* | Registers available for general purposes |
| SR | Segment register |

**Table i. Acronyms and Abbreviated Terms (continued)**

| Term | Meaning |
|---|---|
| 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 |
| XATC | Extended address transfer code |
| 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 |
| Out of order memory accesses | Speculative memory accesses |
| Storage (locations) | Memory |
| Storage (the act of) | Access |

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) |

# Chapter 1
# Overview

The architecture provides a software model that ensures software compatibility among implementations. The term ‘implementation’ is used to refer to a hardware device (typically a microprocessor) that complies with the architecture specifications.

The PowerPC architecture is a 64-bit architecture with a 32-bit subset. This manual describes the architecture from a 32-bit perspective. Although some 64-bit resources are discussed, this manual does not completely describe details of the 64-bit–only features of the architecture, in particular with respect to the memory management model, registers, and instruction set.

The architecture defines the following major components:

- 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 (pages and blocks) of that address space. It also defines the ability to configure pages and blocks 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 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).

For updates to this document, refer to http://www.motorola.com/motorola.

## 1.1 PowerPC Architecture Overview

The PowerPC architecture, developed jointly by Motorola, IBM, and Apple Computer, is based on the POWER architecture implemented by RS/6000™ family of computers. 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 to provide 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.

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 greater need for the performance gained from implementing floating-point normalization and denormalization in hardware than a battery-powered, general-purpose computer might.

The 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. The architecture supports both 32-bit or 64-bit implementations.This document describes the 32-bit portion of the PowerPC architecture.

This chapter provides an overview of the major characteristics of the 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 The 64-Bit Architecture and the 32-Bit Subset

It is important to distinguish the following modes of operations:

- 64-bit implementations/64-bit mode—The 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. For example, memory management differs somewhat between 32- and 64-bit processors. The processor is configured to operate in 64-bit mode by setting a bit in the machine state register (MSR).

- Processors that implement only the 32-bit portion of the architecture provide 32-bit effective addresses, which is also the maximum size of integer data types.
- 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 MSR[SF]. 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 run without modification on 64-bit processors running in 32-bit mode.

## 1.1.2 The Levels of the PowerPC Architecture

The 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 may 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 architecture are defined as follows:

- 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. U
- 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. V

  Implementations that conform to the VEA also adhere to the UISA, but may not necessarily adhere to the OEA.

- 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 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 conform to the OEA also conform to the UISA and VEA.

Implementations that adhere to the VEA level are guaranteed to adhere to the UISA 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 distinctions between the levels of the PowerPC architecture are maintained clearly throughout this document, using the conventions described in the “Conventions” section of the Preface.

### 1.1.3 Latitude Within the Levels of the Architecture

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

- The architecture defines some facilities (such as registers, bits within registers, instructions, and exceptions) as optional.
- The 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 architecture allows implementations to define. For example, the 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.

Because of updates to the architecture specification, which are described in this document, variances may result between existing devices and the revised architecture specification. Those variances are included in *Implementation Variances Relative to Rev. 1 of The Programming Environments Manual*.

### 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 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 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.

### 1.1.5 Summary of Architectural Changes in this Revision

This revision reflects enhancements to the architecture that have been made since the publication of the *PowerPC Microprocessor Family: The Programming Environments*, Rev. 0.1. The primary difference described in this document is the addition of the **rfid** and **mtmsrd** instructions to the 64-bit portion of the architecture. The **rfi** and **mtmsr** instructions are now legal in 32-bit processors and illegal in 64-bit processors. Likewise, the **rfid** and **mtmsrd** are valid instructions only in 64-bit processors and are illegal in 32-bit processors.

In addition, this book reflects smaller changes and clarifications to the PowerPC architecture. For more information, see Section 1.3, “Changes in This Revision of The Programming Environments Manual.”

## 1.2 The Architectural Models

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

- Registers and programming model
- Operand conventions
- Instruction set and addressing modes
- Cache model
- Exception model
- Memory management model

### 1.2.1 Registers and Programming Model

The 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.

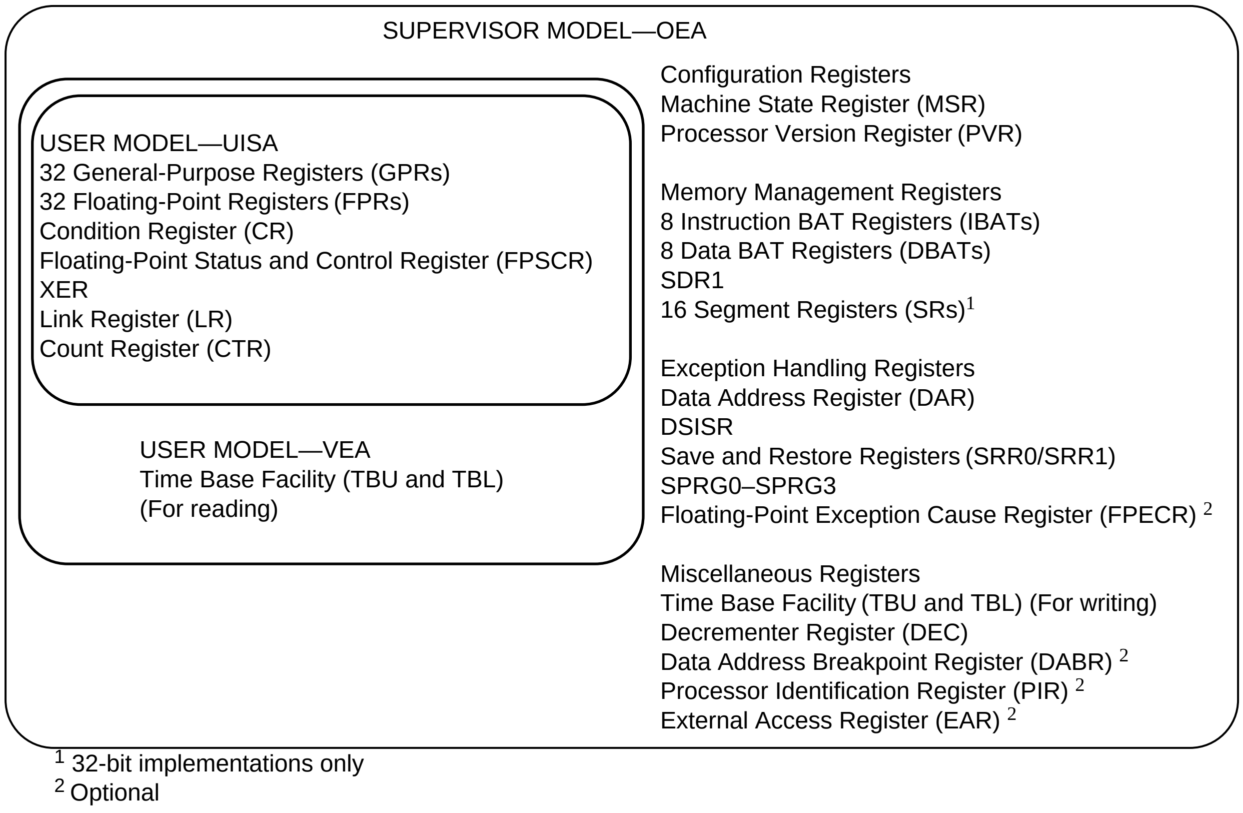

SUPERVISOR MODEL—OEA
USER MODEL—UISA
32 General-Purpose Registers (GPRs)
32 Floating-Point Registers (FPRs)
Condition Register (CR)
Floating-Point Status and Control Register (FPSCR)
XER
Link Register (LR)
Count Register (CTR)
USER MODEL—VEA
Time Base Facility (TBU and TBL)
(For reading)
Configuration Registers
Machine State Register (MSR)
Processor Version Register (PVR)
Memory Management Registers
8 Instruction BAT Registers (IBATs)
8 Data BAT Registers (DBATs)
SDR1
16 Segment Registers (SRs)1
Exception Handling Registers
Data Address Register (DAR)
DSISR
Save and Restore Registers (SRR0/SRR1)
SPRG0–SPRG3
Floating-Point Exception Cause Register (FPECR) 2
Miscellaneous Registers
Time Base Facility (TBU and TBL) (For writing)
Decrementer Register (DEC)
Data Address Breakpoint Register (DABR) 2
Processor Identification Register (PIR) 2
External Access Register (EAR) 2
1 32-bit implementations only
2 Optional

**Figure 1-1. Programming Model—PowerPC Registers**

The programming model incorporates 32 GPRs, 32 FPRs, special-purpose registers (SPRs), and several miscellaneous registers. Each implementation typically has 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 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.

U

- **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. V
- **Operating Environment Architecture Registers**—SPRs defined by the OEA are used for system-level operations such as memory management, exception handling, and time-keeping. O

The architecture also provides room in the SPR space for implementation-specific registers, which are not discussed in this manual.

## 1.2.2 Operand Conventions

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

### 1.2.2.1 Byte Ordering

The default mapping for 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. U

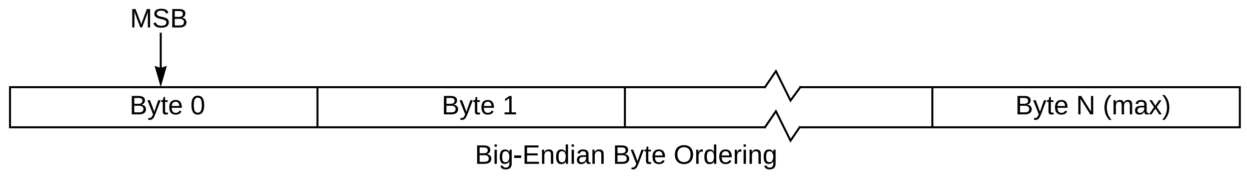


**Figure 1-2. Big-Endian Byte and Bit Ordering**

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

### 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, half words, words, or double words, 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

U The architecture adheres to the IEEE-754 standard for 64- and 32-bit 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 Instruction Set and Addressing Modes

All 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 Instruction Set

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

- Integer instructions—These instructions are defined by the UISA. They include computational and logical instructions. U
  - — Integer arithmetic instructions
  - — Integer compare instructions
  - — Logical instructions
  - — 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).
  - — Floating-point arithmetic instructions
  - — Floating-point multiply/add instructions
  - — Floating-point compare instructions
  - — Floating-point status and control instructions

  - Floating-point move instructions
  - Optional floating-point instructions
- Load/store instructions—These instructions, defined by the UISA, include integer and floating-point load and store instructions.
  - 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
  - Floating-point load and store instructions
- The UISA also provides a set of load/store with reservation instructions (**lwarx** and **stwcx**.) that can be used as primitives for constructing atomic memory operations. These are grouped under synchronization instructions.
- Synchronization instructions—The UISA and VEA define instructions for memory synchronizing, especially useful for multiprocessing:
  - 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 Synchronize instruction (**sync**)—This UISA-defined instruction is useful for synchronizing load and store operations on a memory bus that is shared by multiple devices.
  - Enforce In-Order Execution of I/O (**eieio**)—The **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. 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) (and segment registers in 32-bit implementations). These instructions include move to/from special-purpose register instructions (**mtspr** and **mfspr**).
- Memory/cache control instructions—These instructions provide control of caches, TLBs, and segment registers.
  - The VEA defines several cache control instructions.

  - — The OEA defines one cache control instruction and several memory control instructions.
- External control instructions—The VEA defines two optional instructions for use with special input/output devices.

Note that 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 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 that the architecture specification uses only the term effective address and not logical address.)

The 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 Cache Model

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

The architecture allows implementations to control the following memory access modes on a page or block 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, “Memory Management Model.”

The VEA 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. The OEA portion of the architecture defines one cache management instruction—the Data Cache Block Invalidate (**dcbi**) instruction. O

## 1.2.5 Exception Model

The 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 exception model is described in detail in Chapter 6, "Exceptions." Note also that 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 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 Memory Management Model O

The memory management unit (MMU) specifications are provided by the OEA. The primary functions of the MMU 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 that many aspects of memory management are implementation-dependent. The description in Chapter 7, "Memory Management," describes the conceptual model of a MMU; however, processors may differ in the specific hardware used to implement the MMU model of the OEA.

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 OEA includes models for both 64- and 32-bit implementations. The MMU of a 32-bit processor provides $2^{32}$ bytes of logical address space accessible to supervisor and user programs with a 4-Kbyte page size and 256-Mbyte segment size.

In 32-bit implementations, the entire 4-Gbyte memory space is defined by sixteen 256-Mbyte segments. Segments are configured through the 16 segment registers. In 64-bit implementations there are more segments than can be maintained in architecture-defined registers, so segment descriptors are maintained in segment table entries (STEs) in memory and are accessed through the use of a hashing algorithm much like that used for accessing page table entries (PTEs).

The block address translation (BAT) mechanism maps large blocks of memory. Block sizes range from 128 Kbytes to 256 Mbytes and are software-selectable. In addition, the MMU of 32-bit processors uses an interim virtual address (52 bits) and hashed page tables in the generation of 32-bit physical addresses.

Two types of processor-generated accesses 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 (or segment registers in 32-bit implementations) and page tables used 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 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 may contain segment lookaside buffers (SLBs) on-chip that contain recently-used segment table entries, but for which the architecture does not define the exact characteristics.

The block address translation (BAT) mechanism is a software-controlled array that stores the available block address translations on-chip. BAT array entries are implemented as pairs of BAT registers that are accessible as supervisor special-purpose registers (SPRs); refer to Chapter 7, “Memory Management,” for more information.

## 1.3 Changes in This Revision of *The Programming Environments Manual*

This book reflects changes made to the architecture after the publication of Rev. 0 of *The Programming Environments Manual* and before Dec. 13, 1994 (Rev. 0.1). In addition, it reflects changes made to the architecture after the publication of Rev. 0.1 of *The Programming Environments Manual* and before Aug. 6, 1996 (Rev. 1). Although there are many changes in this revision, this section summarizes only the most significant changes and clarifications to the architecture specification.

The main substantive change from Rev. 0 to Rev. 1 for 32-bit processors is the phasing out of the direct-store facility. This facility defined segments that were used to generate direct-store interface accesses on the external bus to communicate with specialized I/O devices; it was not optimized for performance in the architecture and was present for compatibility with older devices only. As of this revision of the architecture (Rev. 1), direct-store segments are an optional processor feature. However, they are not likely to be supported in future implementations and new software should not use them.

Table 1-1 and Table 1-2 list changes made to the UISA that are reflected in this book and identify the chapters affected by those changes. Note that many of the changes made in the UISA are reflected in both the VEA and OEA portions of the architecture as well.

**Table 1-1. UISA Changes—Rev. 0 to Rev. 0.1**

| Change | Chapter(s) Affected |
|---|---|
| The rules for handling of reserved bits in registers are clarified. | 2 |
| Clarified that **isync** does not wait for memory accesses to be performed. | 4, 8 |
| CR0[0–2] are undefined for some instructions in 64-bit mode. | 4, 8 |
| Clarified intermediate result with respect to floating-point operations (the intermediate result has infinite precision and unbounded exponent range). | 3 |
| Clarified the definition of rounding such that rounding always occurs (specifically, FR and FI flags are always affected) for arithmetic, rounding, and conversion instructions. | 3 |
| Clarified the definition of the term 'tiny' (detected before rounding). | 3 |
| In Section D.3.2, "Conversion from Floating-Point Number to Unsigned Fixed-Point Integer Word," changed value in FPR 3 from $2^{32}$ to $2^{32} - 1$ (in 32-bit implementation description). | D |
| Noted additional POWER incompatibility for Store Floating-Point Single (**stfs**) instruction. | B |

**Table 1-2. UISA Changes—Rev. 0.1 to Rev. 1.0**

| Change | Chapter(s) Affected |
|---|---|
| Although the **stfiwx** instruction is an optional instruction, it will likely be required for future processors. | 4, 8, A |
| Added the new Data Cache Block Allocate (**dcba**) instruction. | 4, 5, 8, A |
| Deleted some warnings about generating misaligned little-endian access. | 3 |

Table 1-3 and Table 1-4 list changes made to the VEA that are reflected in this book and the chapters that are affected by those changes. Note that some changes to the UISA are reflected in the VEA and in turn, some changes to the VEA affect the OEA as well.

**Table 1-3. VEA Changes—Rev. 0 to Rev. 0.1**

| Change | Chapter(s) Affected |
|---|---|
| Clarified conditions under which a cache block is considered modified. | 5 |
| WIMG bits have meaning only when the effective address is translated. | 2, 5, 7 |
| Clarified that **isync** does not wait for memory accesses to be performed. | 4, 5, 7, 8 |
| Clarified paging implications of **eciwx** and **ecowx**. | 4, 5, 7, 8 |

**Table 1-4. VEA Changes—Rev. 0.1 to Rev. 1.0**

| Change | Chapter(s) Affected |
|---|---|
| Added the requirement that caching-inhibited guarded store operations are ordered. | 5 |
| Clarified use of the **dcbf** instruction in keeping instruction cache coherency in the case of a combined instruction/data cache in a multiprocessor system. | 5 |

Table 1-5 and Table 1-6 list changes made to the OEA that are reflected in this book and the chapters that are affected by those changes. Note that some changes to the UISA and VEA are reflected in the OEA as well.

**Table 1-5. OEA Changes—Rev. 0 to Rev. 0.1**

| Change | Chapter(s) Affected |
|---|---|
| Restricted several aspects of out-of-order operations. | 2, 4, 5, 6, 7 |
| Clarified instruction fetching and instruction cache paradoxes. | 4, 5 |
| Specified that IBATs contain W and G bits and that software must not write 1s to them. | 2, 7 |
| Corrected the description of coherence when the W bit differs among processors. | 5 |
| Clarified that referenced and changed bits are set for virtual pages. | 7 |
| Revised the description of changed bit setting to avoid depending on the TLB. | 7 |
| Tightened the rules for setting the changed bit out of order. | 5, 7 |
| Specified which multiple DSISR bits may be set due to simultaneous DSI exceptions. | 6 |
| Removed software synchronization requirements for reading the TB and DEC. | 2 |
| More flexible DAR setting for a DABR exception. | 6 |

**Table 1-6. OEA Changes—Rev. 0.1 to Rev. 1.0**

| Change | Chapter(s) Affected |
|---|---|
| Changed definition of direct-store segments to an optional processor feature that is not likely to be supported in future implementations and new software should not use it. | 2, 6, 7 |
| Changed the ranges of bits saved from MSR to SRR1 (and restored from SRR1 to MSR on **rfi**) on an exception. | 2, 6 |
| Clarified the definition of execution synchronization. Also clarified that the **mtmsr** and **mtmsrd** instructions are not execution synchronizing. | 2, 4, 8 |
| Clarified the use of memory allocated for predefined uses (including the exception vectors). | 6, 7 |
| Revised the page table update synchronization requirements and recommended code sequences. | 7 |

# Chapter 2
# Register Set

This chapter describes the register organization defined by the three levels of the architecture—user instruction set architecture (UISA), virtual environment architecture (VEA), and operating environment architecture (OEA). The 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, preserving the original data for use by other instructions and reducing the number of instructions for some operations. Data is transferred between memory and registers with explicit load and store instructions only.

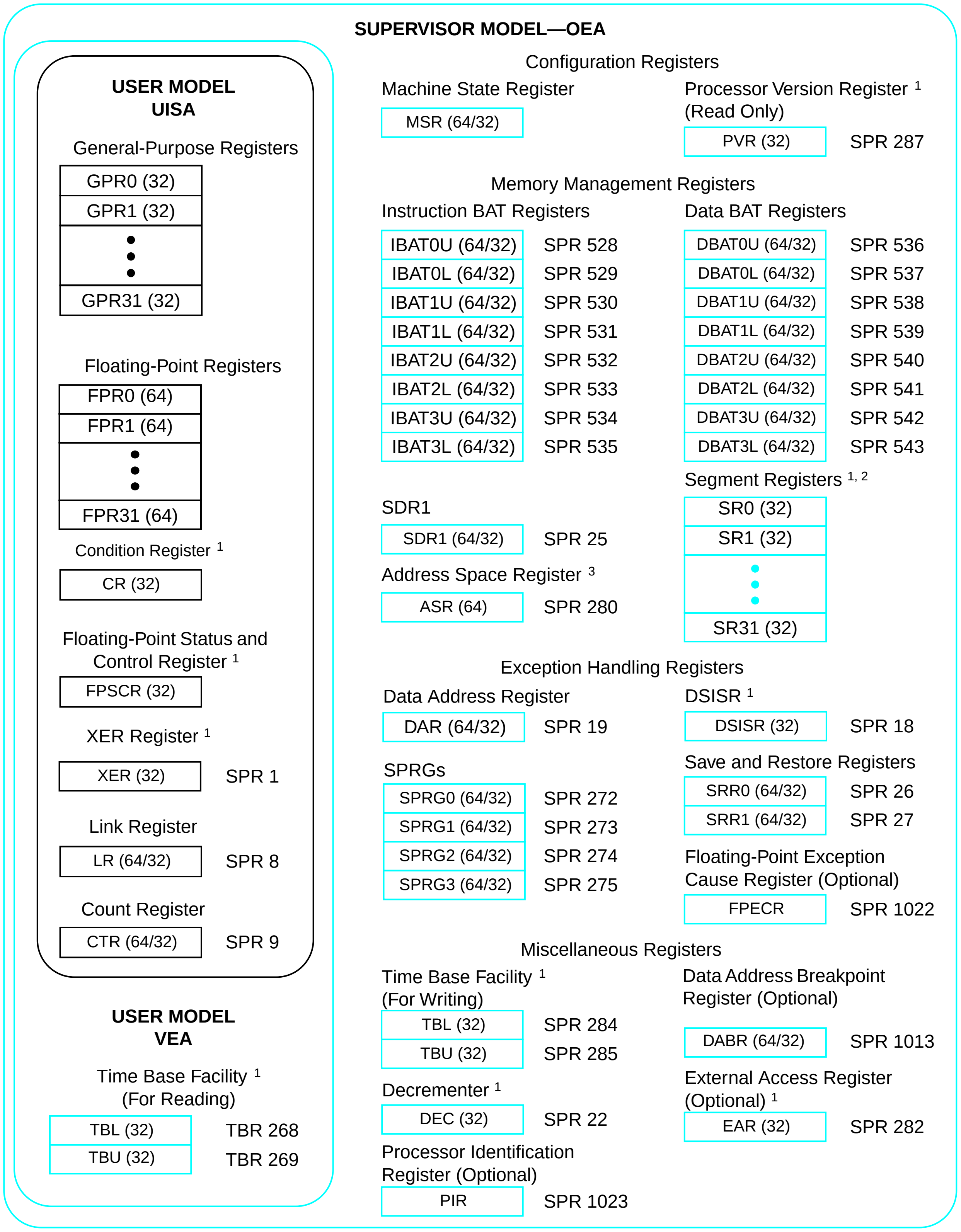
Note that the handling of reserved bits in any register is implementation-dependent. Software is permitted to write any value to a reserved bit in a register. However, a subsequent reading of the reserved bit returns 0 if the value last written to the bit was 0 and returns an undefined value (may be 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 UISA Register Set

The 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 names 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).

Note that the GPRs, LR, and CTR are 64 bits wide on 64-bit implementations and 32 bits wide on 32-bit implementations.



[1] These registers are 32-bit registers only.
[2] These registers are on 32-bit implementations only.
[3] These registers are on 64-bit implementations only.

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

The user-level registers can be accessed by all software with either user or supervisor privileges. The user-level register set includes the following:

- General-purpose registers (GPRs). The general-purpose register file consists of 32 GPRs designated as GPR0–GPR31. The GPRs serve as data source or destination registers for all integer instructions and provide data for generating addresses. See Section 2.1.1, "General-Purpose Registers (GPRs)."
- Floating-point registers (FPRs). The floating-point register file consists of 32 FPRs designated as FPR0–FPR31; FPRs serve as the data source or destination for all floating-point instructions. The floating-point model includes data objects of either single- or double-precision floating-point format, but the FPRs only contain data in double-precision format. See Section 2.1.2, "Floating-Point Registers (FPRs)."
- Condition register (CR). The 32-bit CR has eight 4-bit fields, CR0–CR7, that reflect the results of certain arithmetic operations and provides a mechanism for testing and branching. See Section 2.1.3, "Condition Register (CR)."
- Floating-point status and control register (FPSCR). The FPSCR contains all signal, summary and enable bits for floating-point exceptions and rounding control bits for compliance with the IEEE 754 standard. See Section 2.1.4, "Floating-Point Status and Control Register (FPSCR)."
- XER register (XER). The XER indicates overflows and carry conditions for integer operations and the number of bytes to be transferred by the load/store string indexed instructions. See Section 2.1.5, "XER Register (XER)."
- Link register (LR). The LR 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)."
- Count register (CTR). The CTR 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)."

### 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 in 64-bit implementations and 32-bit registers in 32-bit implementations. The GPRs are accessed as source and destination registers in the instruction syntax.

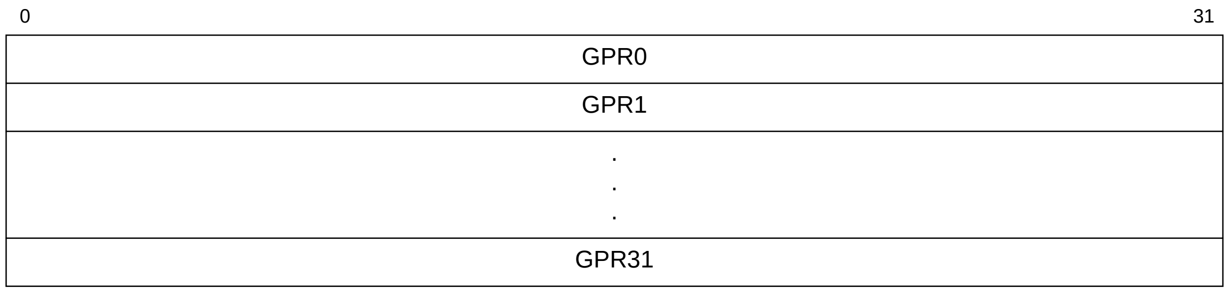


**Figure 2-2. General-Purpose Registers (GPRs)**

## 2.1.2 Floating-Point Registers (FPRs)

The architecture provides thirty-two 64-bit FPRs as shown in Figure 2-3. These registers are accessed as source and 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. Note that FPRs are 64 bits on both 64-bit and 32-bit processor implementations.

All floating-point arithmetic instructions operate on data located in FPRs and, with the exception of compare instructions, place the result into an 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)."

Load and store double-word instructions transfer 64 bits of data between memory and the FPRs with no conversion. Load single instructions are provided to read a single-precision floating-point value from memory, convert it to double-precision floating-point format, and place it in the target floating-point register. Store single-precision instructions are provided to read a double-precision floating-point value from a floating-point register, convert it to single-precision floating-point format, and place it in the target memory location.

Single- and double-precision arithmetic instructions accept values from the FPRs in double-precision format. For single-precision arithmetic and store instructions, all input values must be representable in single-precision format; otherwise, the result 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," for more information.

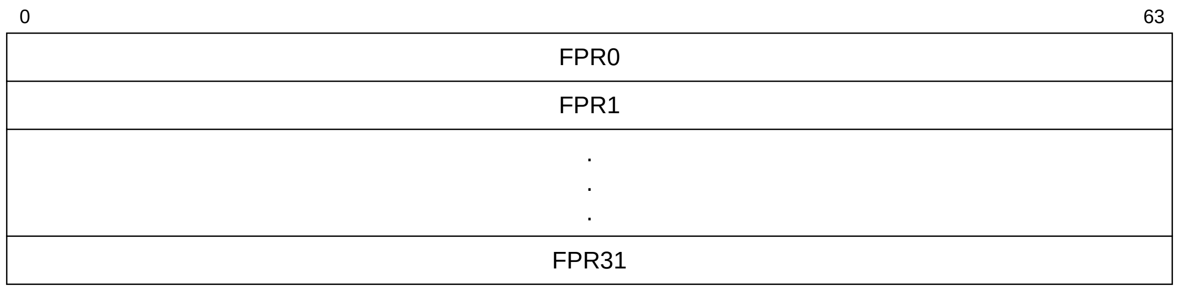


**Figure 2-3. Floating-Point Registers (FPRs)**

## 2.1.3 Condition Register (CR)

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

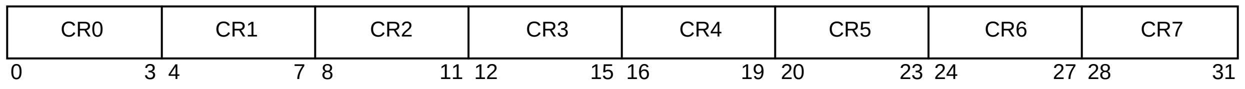


**Figure 2-4. Condition Register (CR)**

The CR fields can be set by using the following instructions:

- Specified CR fields are set from a GPR by using **mtcrf**.
- The contents of one CR field are copied into another CR field by using **mcrf**. All other condition register fields remain unchanged.
- The contents of XER[0–3] is moved to another CR field by using **mcrxr**.
- A specified FPSCR field is copied to a specified field of the CR by using **mcrfs**.
- CR logical instructions perform logical operations on specified CR bits.
- 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 that 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.

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

| Bits | 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. |
| 3 | Summary overflow (SO)—This is a copy of the final state of XER[SO] at the completion of the instruction. |

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

#### 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 (that is, when the instruction's record bit, Rc, is set), CR1 (bits 4–7 of the CR) is copied from FPSCR[0–3] and indicates the floating-point exception status. See Section 2.1.4, "Floating-Point Status and Control Register (FPSCR)." Table 2-2 shows CR1 bit settings.

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

| Bits | Description |
|---|---|
| 4 | Floating-point exception (FX). Copy of the final state of FPSCR[FX] at the completion of the instruction. |
| 5 | Floating-point enabled exception (FEX). Copy of the final state of FPSCR[FEX] at the completion of the instruction. |
| 6 | Floating-point invalid exception (VX). Copy of the final state of FPSCR[VX] at the completion of the instruction. |
| 7 | Floating-point overflow exception (OX). Copy of the final state of FPSCR[OX] at the completion of the instruction. |

#### 2.1.3.3 Condition Register CR*n* 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. CR*n* Field Bit Settings for Compare Instructions**

| Bits [1] | Description [2] |
|---|---|
| 0 | Less than or floating-point less than (LT, FL).<br>For integer compare instructions:<br>**r**A < SIMM or **r**B (signed comparison) or **r**A < UIMM or **r**B (unsigned comparison).<br>For floating-point compare instructions:**fr**A < **fr**B. |
| 1 | Greater than or floating-point greater than (GT, FG).<br>For integer compare instructions:<br>**r**A > SIMM or **r**B (signed comparison) or **r**A > UIMM or **r**B (unsigned comparison).<br>For floating-point compare instructions:**fr**A > **fr**B. |

Table 2-3. CR*n* Field Bit Settings for Compare Instructions (continued)

| Bits [1] | Description [2] |
|---|---|
| 2 | Equal or floating-point equal (EQ, FE).<br>For integer compare instructions: **r**A = SIMM, UIMM, or **r**B.<br>For floating-point compare instructions: **fr**A = **fr**B. |
| 3 | Summary overflow or floating-point unordered (SO, FU).<br>For integer compare instructions, this is a copy of the final state of XER[SO] at the completion of the instruction.<br>For floating-point compare instructions, one or both of **fr**A and **fr**B is a Not a Number (NaN). |

[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.

## 2.1.4 Floating-Point Status and Control Register (FPSCR)

The FPSCR, shown in Figure 2-5, contains bits that do the following:

- Record exceptions generated by floating-point operations
- Record the type of the result produced by a floating-point operation
- Control the rounding mode used by floating-point operations
- Enable or disable the reporting of exceptions (invoking the exception handler)

Bits 0–23 are status bits, which are updated at the completion of the instruction execution. Bits 24–31 are control bits.

Except for the floating-point enabled exception summary (FEX) and floating-point invalid operation exception summary (VX), the exception condition bits, FPSCR[0–12,21–23], are sticky. Once set, sticky bits remain set until they are cleared by an **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.

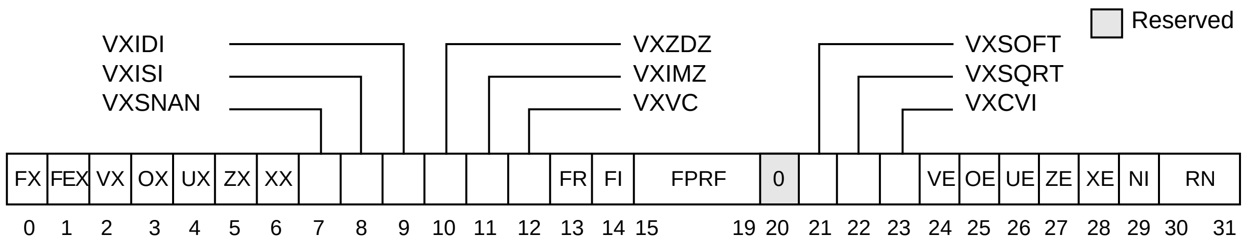


Figure 2-5. Floating-Point Status and Control Register (FPSCR)

FPSCR bits are decribed in Table 2-4

**Table 2-4. FPSCR Bit Settings**

| Bits | Name | Description |
|---|---|---|
| 0 | FX | Floating-point exception summary. Every floating-point instruction, except **mtfsfi** and **mtfsf**, implicitly sets FX if that instruction causes any FPSCR floating-point exception bit to transition from 0 to 1. The **mcrfs**, **mtfsfi**, **mtfsf**, **mtfsb0**, and **mtfsb1** instructions can alter FX explicitly. This is a sticky bit. |
| 1 | FEX | Floating-point enabled exception summary. Signals the occurrence of any 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 exception bits as described in Section 3.3.6.1.1, “Invalid Operation Exception Condition.” 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.” |
| 4 | UX | Floating-point underflow exception. This is a sticky bit. See Section 3.3.6.2.2, “Underflow Exception Condition.” |
| 5 | ZX | Floating-point zero divide exception. This is a sticky bit. See Section 3.3.6.1.2, “Zero Divide Exception Condition.” |
| 6 | XX | Floating-point inexact exception. This is a sticky bit. See Section 3.3.6.2.3, “Inexact Exception Condition.” XX is the sticky version of FPSCR[FI]. A given instruction sets XX as follows:<br>• 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].<br>• 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 Section 3.3.6.1.1, “Invalid Operation Exception Condition.” |
| 8 | VXISI | Floating-point invalid operation exception for $\infty - \infty$. This is a sticky bit. See Section 3.3.6.1.1, “Invalid Operation Exception Condition.” |
| 9 | VXIDI | Floating-point invalid operation exception for $\infty \div \infty$. This is a sticky bit. See Section 3.3.6.1.1, “Invalid Operation Exception Condition.” |
| 10 | VXZDZ | Floating-point invalid operation exception for $0 \div 0$. This is a sticky bit. See Section 3.3.6.1.1, “Invalid Operation Exception Condition.” |
| 11 | VXIMZ | Floating-point invalid operation exception for $\infty * 0$. This is a sticky bit. See Section 3.3.6.1.1, “Invalid Operation Exception Condition.” |
| 12 | VXVC | Floating-point invalid operation exception for invalid compare. This is a sticky bit. See Section 3.3.6.1.1, “Invalid Operation Exception Condition.” |
| 13 | FR | Floating-point fraction rounded. The last arithmetic, rounding, or conversion instruction incremented the fraction. See Section 3.3.5, “Rounding.” This bit is not sticky. |
| 14 | FI | Floating-point fraction inexact. The last arithmetic, rounding, or conversion instruction either produced an inexact result during rounding 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 2-4. FPSCR Bit Settings (continued)

| Bits | Name | Description |
|---|---|---|
| 15–19 | FPRF | Floating-point result flags. For arithmetic, rounding, and conversion instructions, FPRF 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.<br>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.<br>Bits 16–19 comprise the 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.<br>16 Floating-point less than or negative (FL or <)<br>17 Floating-point greater than or positive (FG or >)<br>18 Floating-point equal or zero (FE or =)<br>19 Floating-point unordered or NaN (FU or ?)<br>Note that 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 Section 3.3.6.1.1, “Invalid Operation Exception Condition.” |
| 22 | VXSQRT | Floating-point invalid operation exception for invalid square root. This is a sticky bit. For more detailed information, refer to Section 3.3.6.1.1, “Invalid Operation Exception Condition.” |
| 23 | VXCVI | Floating-point invalid operation exception for invalid integer convert. This is a sticky bit. See Section 3.3.6.1.1, “Invalid Operation Exception Condition.” |
| 24 | VE | Floating-point invalid operation exception enable. See Section 3.3.6.1.1, “Invalid Operation Exception Condition.” |
| 25 | OE | IEEE floating-point overflow exception enable. See Section 3.3.6.2, “Overflow, Underflow, and Inexact Exception Conditions.” |
| 26 | UE | IEEE floating-point underflow exception enable. See Section 3.3.6.2.2, “Underflow Exception Condition.” |
| 27 | ZE | IEEE floating-point zero divide exception enable. See Section 3.3.6.1.2, “Zero Divide Exception Condition.” |
| 28 | XE | Floating-point inexact exception enable. See Section 3.3.6.2.3, “Inexact Exception Condition.” |
| 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 *i*mplementation-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.<br>Effects of the setting of this bit are implementation-dependent. |
| 30–31 | RN | Floating-point rounding control. See Section 3.3.5, “Rounding.”<br>00 Round to nearest<br>01 Round toward zero<br>10 Round toward +infinity<br>11 Round toward –infinity |

Table 2-5 describes the floating-point result flags, which correspond to FPSCR[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 XER register is a 32-bit, user-level register shown in Figure 2-6.

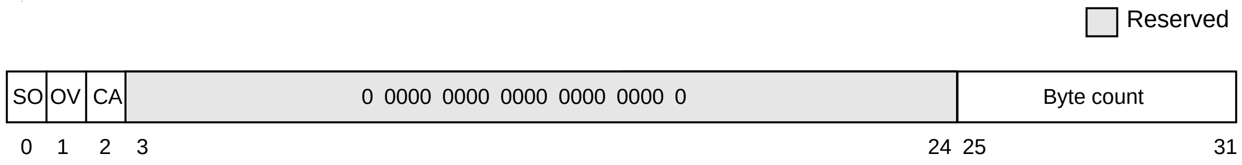


Figure 2-6. XER Register

The XER bit definitions, 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 XER bits based on the entire operation, not on an intermediate sum.

Table 2-6. XER Bit Definitions

| Bits | Name | Description |
|---|---|---|
| 0 | 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. |
| 1 | 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 that cannot overflow (except **mtspr** to the XER, and **mcrxr**). |

Table 2-6. XER Bit Definitions (continued)

| Bits | Name | Description |
|---|---|---|
| 2 | CA | Carry. Set during execution of the following instructions:<br>• 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.<br>• Shift right algebraic instructions set CA if any 1 bits have been shifted out of a negative operand, and clear it otherwise.<br>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**). |
| 3–24 | — | Reserved |
| 25–31 | Byte count | 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 in 64-bit implementations and a 32-bit register in 32-bit implementations. The LR 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.

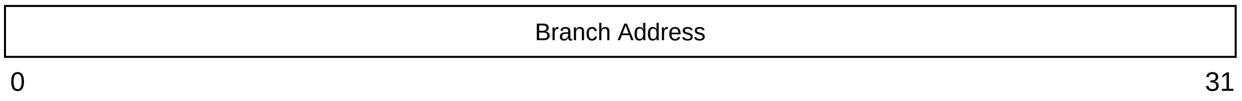


Figure 2-7. Link Register (LR)

Note that 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 LR is loaded sufficiently ahead of the branch instruction (so that any branch prediction hardware can calculate the branch address). Additionally, processors can prefetch along a target path loaded by a branch and link instruction.

Note that some 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 LR 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 in 64-bit implementations and a 32-bit register in 32-bit implementations. The CTR 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 0xFFFF_FFFF ($2^{32}$–1) afterward. 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.

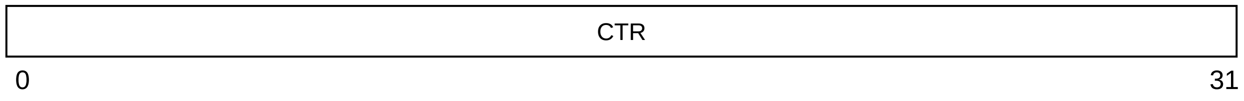


**Figure 2-8. Count Register (CTR)**

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 2-7.

**Table 2-7. BO Operand Encodings**

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

**Notes**: The *y* bit provides a hint about whether a conditional branch is likely to be taken and is used by some implementations to improve performance. Other implementations may ignore the *y* bit.

The *z* indicates a bit that is ignored. The *z* bits should be cleared (zero), as they may be assigned a meaning in a future version of the UISA.

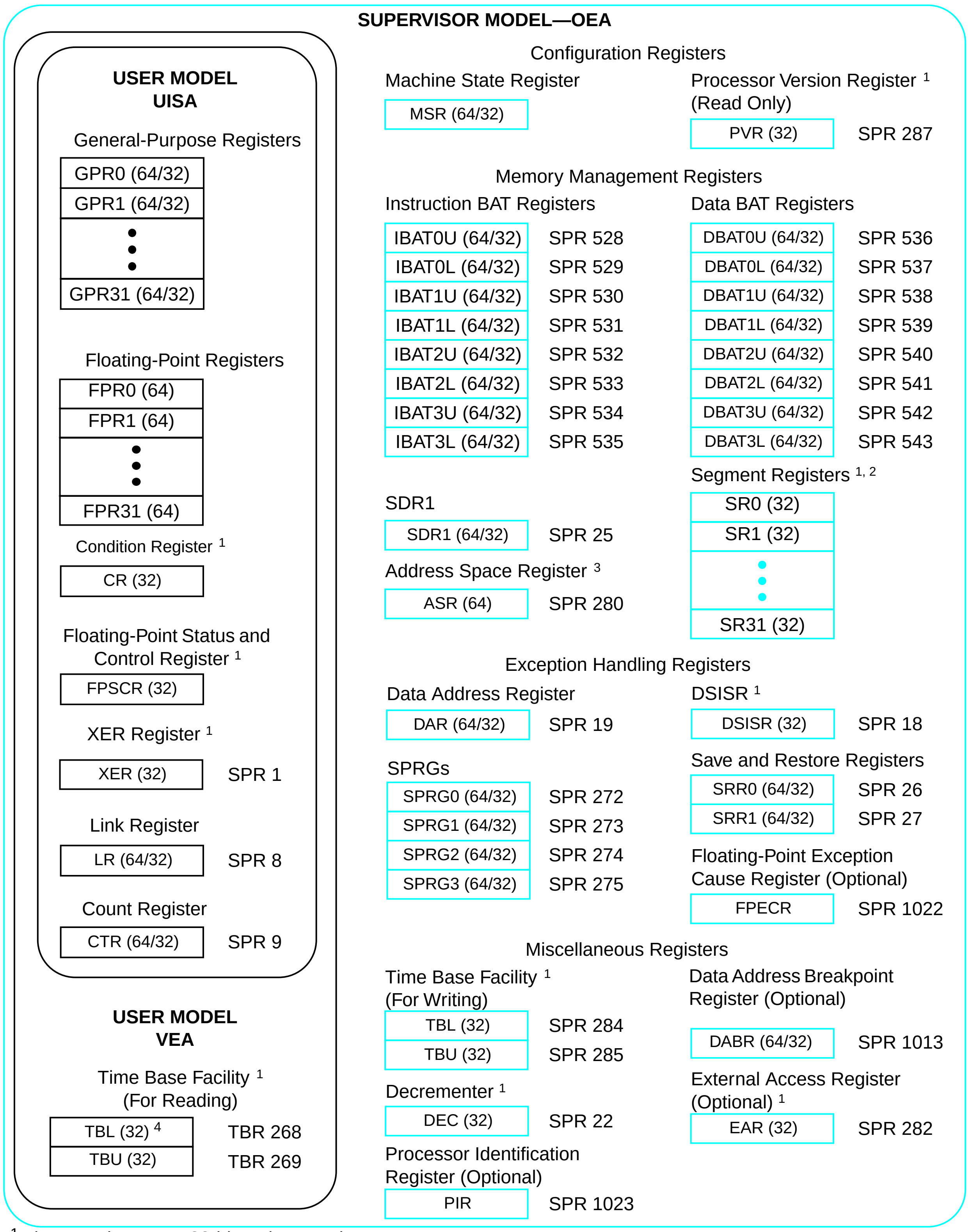
## 2.2 VEA Register Set—Time Base

The virtual environment architecture (VEA) defines registers in addition to those defined by the UISA. The VEA register set can be accessed by all software with either user- or supervisor-level privileges. Figure 2-9 shows the VEA register set. Note that the following programming model is similar to that found in Figure 2-1, however, the VEA registers are now included.

The 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 that 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 2.3.12, “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).

Note that the GPRs, LR, and CTR are 64 bits on 64-bit implementations and 32 bits on 32-bit implementations. These registers are described fully in Section 2.1, “UISA Register Set.”



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

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

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.

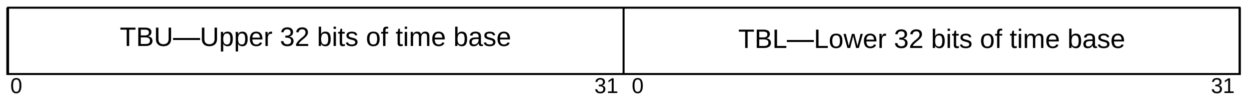

TBU—Upper 32 bits of time base
TBL—Lower 32 bits of time base
0
31
0
31

**Figure 2-10. Time Base (TB)**

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. Note that there is no explicit indication that this has occurred (that is, no exception is generated).

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 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 that 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. For specific details on using the **mftb** instruction, see Chapter 8, “Instruction Set.” For information on writing the time base, see Section 2.3.12.1, “Writing to the Time Base.”

On 32-bit implementations, it is not possible to read the entire 64-bit time base in a single instruction. 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.

Because of the possibility of a carry from TBL to TBU occurring between reads of the TBL and TBU, a sequence such as the following example is necessary to read the time base on 32-bit implementations:

```
loop:
        mftbu   rx      #load from TBU
        mftb    ry      #load from TBL
        mftbu   rz      #load from TBU
        cmpw    rz,rx   #see if ‘old’ = ‘new’
        bne     loop    #loop if carry occurred
```

The comparison and loop are necessary to ensure that a consistent pair of values has been obtained. The previous example will also work on 64-bit implementations running in either 64-bit or 32-bit mode.

## 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 system-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 system in which the time base update frequency does not vary, the first three inputs would be constant.

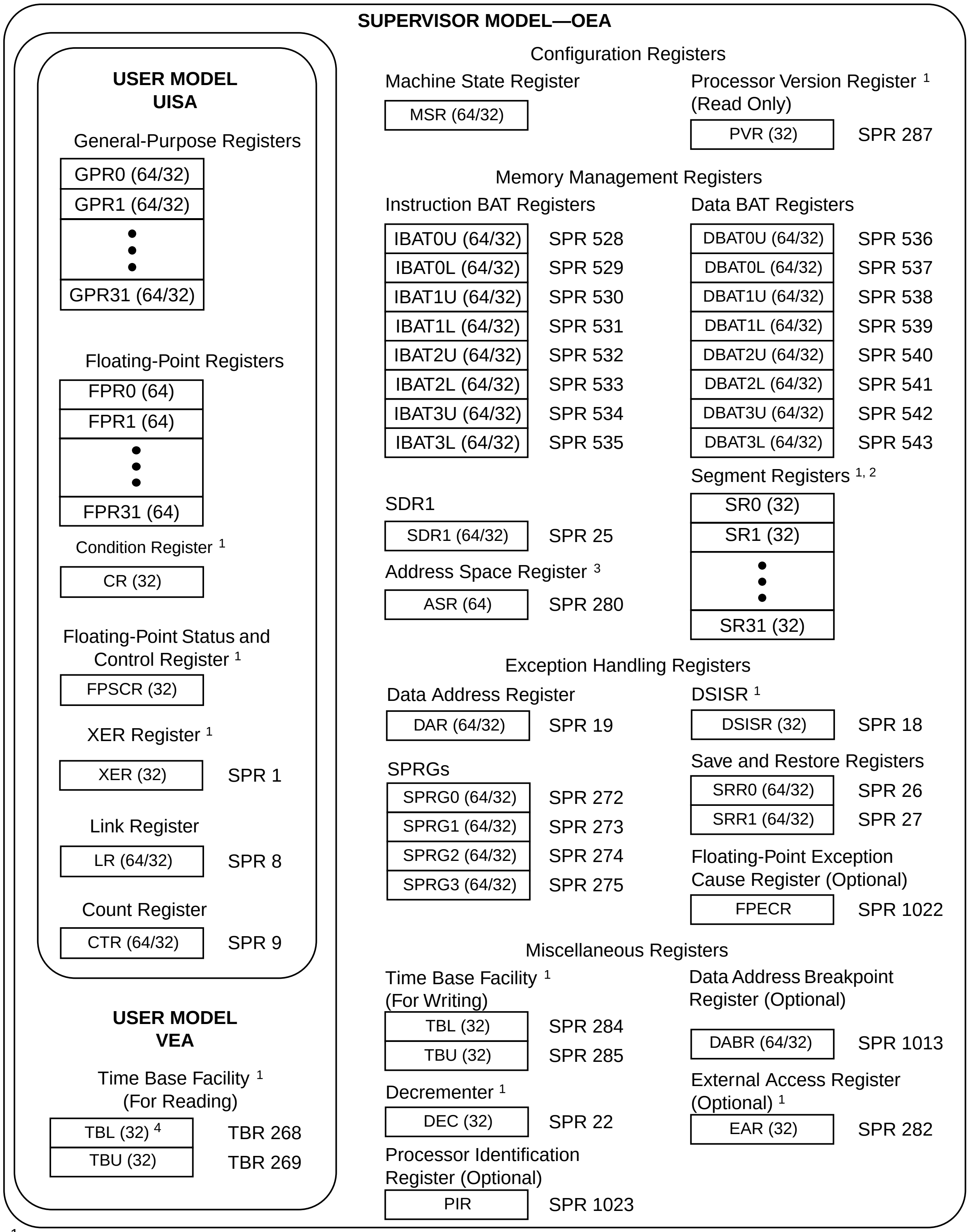
## 2.3 OEA Register Set

The operating environment architecture (OEA) completes the discussion of registers. Figure 2-11. shows the entire 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 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 6.4.7, "Program Exception (0x00700)," for more information.

Note that the GPRs, LR, CTR, TBL, MSR, DAR, SDR1, SRR0, SRR1, and SPRG0–SPRG3 are 64 bits wide on 64-bit implementations and 32 bits wide on 32-bit implementations.



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

**Figure 2-11. OEA Programming Model—All Registers**

A description of the OEA supervisor-level registers follows:

- **Configuration registers**
  - Machine state register (MSR). The MSR defines the state of the processor. The MSR can be modified by the Move to Machine State Register (**mtmsr**), System Call (**sc**), and Return from Interrupt (**rfi**) 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). This register is a read-only register that identifies the version (model) and revision level of the processor. For more information, see Section 2.3.2, “Processor Version Register (PVR).”
- **Memory management registers**
  - Block-address translation (BAT) registers. The OEA defines four pairs of instruction BATs (IBAT0U–IBAT3U and IBAT0L–IBAT3L) and four pairs of data BATs (DBAT0U–DBAT3U and DBAT0L–DBAT3L). Figure 2-11 shows SPR numbers for BAT registers. See Section 2.3.3, “BAT Registers,” for more information.
  - SDR1. The SDR1 register specifies the page table base address used in virtual-to-physical address translation. For more information, see Section 2.3.4, “SDR1.” (Note that physical address is referred to as real address in the architecture specification.)
  - Segment registers (SR). The OEA defines sixteen 32-bit segment registers (SR0–SR15). Note that the SRs are implemented on 32-bit implementations only. The fields in the segment register are interpreted differently depending on the value of bit 0. For more information, see Section 2.3.5, “Segment Registers.”
- **Exception handling registers**
  - Data address register (DAR). After a DSI or an alignment exception, DAR is set to the effective address generated by the faulting instruction. For more information, see Section 2.3.6, “Data Address Register (DAR).”
  - SPRG0–SPRG3. The SPRG0–SPRG3 registers are provided for operating system use. For more information, see Section 2.3.7, “SPRG0–SPRG3.”
  - DSISR. The DSISR defines the cause of DSI and alignment exceptions. For more information, refer to Section 2.3.8, “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 **rfi** instruction is executed. For more information, see Section 2.3.9, “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 **rfi** instruction is executed. For more information, see Section 2.3.10, “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

— 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 2.3.12, “Time Base Facility (TB)—OEA,” and Section 2.2, “VEA Register Set—Time Base.”

— Decrementer register (DEC). This 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 2.3.13, “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 architecture and may not be supported in all 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 architecture and may not be supported in all 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 on 64-bit implementations and a 32-bit register in 32-bit implementations (see Figure 2-12). The MSR defines the state of the processor. When an exception occurs, MSR bits, as described in Table 2-8, are altered as determined by the exception. The MSR can also be modified by the **mtmsr**, **sc**, and **rfi** instructions. It can be read by the **mfmsr** instruction.

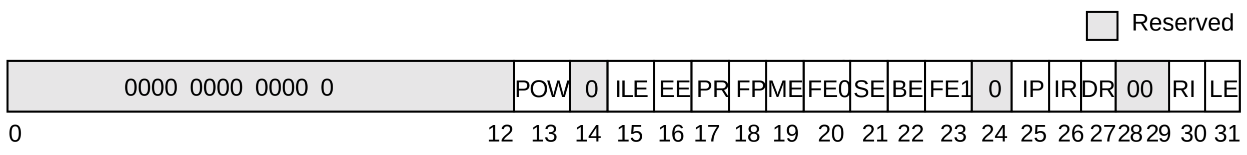


**Figure 2-12. Machine State Register (MSR)**

Table 2-9 shows the bit definitions for the MSR.

**Table 2-9. MSR Bit Settings**

| Bits | Name | Description |
|---|---|---|
| 0–12 | — | Reserved |
| 13 | POW | Power management enable<br>0 Power management disabled (normal operation mode)<br>1 Power management enabled (reduced power mode)<br>Note: Power management functions are implementation-dependent. If the function is not implemented, this bit is treated as reserved. |
| 14 | — | Reserved |
| 15 | ILE | Exception little-endian mode. When an exception occurs, this bit is copied into MSR[LE] to select the endian mode for the context established by the exception. |
| 16 | EE | External interrupt enable<br>0 While the bit is cleared, the processor delays recognition of external interrupts and decrementer exception conditions.<br>1 The processor is enabled to take an external interrupt or the decrementer exception. |
| 17 | PR | Privilege level<br>0 The processor can execute both user- and supervisor-level instructions.<br>1 The processor can only execute user-level instructions. |
| 18 | FP | Floating-point available<br>0 The processor prevents dispatch of floating-point instructions, including floating-point loads, stores, and moves.<br>1 The processor can execute floating-point instructions. |
| 19 | ME | Machine check enable<br>0 Machine check exceptions are disabled.<br>1 Machine check exceptions are enabled. |
| 20 | FE0 | Floating-point exception mode 0 (see Table 2-10). |
| 21 | SE | Single-step trace enable (Optional)<br>0 The processor executes instructions normally.<br>1 The processor generates a single-step trace exception upon the successful execution of the next instruction.<br>Note: If the function is not implemented, this bit is treated as reserved. |
| 22 | BE | Branch trace enable (Optional)<br>0 The processor executes branch instructions normally.<br>1 The processor generates a branch trace exception after completing the execution of a branch instruction, regardless of whether the branch was taken.<br>Note: If the function is not implemented, this bit is treated as reserved. |
| 23 | FE1 | Floating-point exception mode 1 (See Table 2-10). |
| 24 | — | Reserved |
| 25 | IP | Exception prefix. The setting of this bit specifies whether an exception vector offset is prepended with Fs or 0s. In the following description, *nnnnn* is the offset of the exception vector. See Table 6-2.<br>0 Exceptions are vectored to the physical address 0x000*n_nnnn* in 32-bit implementations and 0x0000_0000_000*n_nnnn* in 64-bit implementations.<br>1 Exceptions are vectored to the physical address 0xFFF*n_nnnn* in 32-bit implementations and 0x0000_0000_FFF*n_nnnn* in 64-bit implementations.<br>In most systems, IP is set during system initialization and then cleared when initialization is complete. |

**Table 2-9. MSR Bit Settings (continued)**

| Bits | Name | Description |
|---|---|---|
| 26 | IR | Instruction address translation<br>0 Instruction address translation is disabled.<br>1 Instruction address translation is enabled.<br>For more information, see Chapter 7, “Memory Management.” |
| 27 | DR | Data address translation<br>0 Data address translation is disabled.<br>1 Data address translation is enabled.<br>For more information, see Chapter 7, “Memory Management.” |
| 28–29 | — | Reserved |
| 30 | RI | Recoverable exception (for system reset and machine check exceptions).<br>0 Exception is not recoverable.<br>1 Exception is recoverable.<br>For more information, see Chapter 6, “Exceptions.” |
| 31 | LE | Little-endian mode enable<br>0 The processor runs in big-endian mode.<br>1 The processor runs in little-endian mode. |

The floating-point exception mode bits (FE0–FE1) are interpreted as shown in Table 2-10.

**Table 2-10. 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 2-11 indicates the initial state of the MSR at power up.

**Table 2-11. State of MSR at Power Up**

| Bits | Name | 32-Bit Default Value |
|---|---|---|
| 0–12 | — | Unspecified [1] |
| 13 | POW | 0 |
| 14 | — | Unspecified[1] |
| 15 | ILE | 0 |
| 16 | EE | 0 |
| 17 | PR | 0 |
| 18 | FP | 0 |
| 19 | ME | 0 |
| 20 | FE0 | 0 |
| 21 | SE | 0 |

**Table 2-11. State of MSR at Power Up**

| Bits | Name | 32-Bit Default Value |
|---|---|---|
| 22 | BE | 0 |
| 23 | FE1 | 0 |
| 24 | — | Unspecified[1] |
| 25 | IP | 1 [2] |
| 26 | IR | 0 |
| 27 | DR | 0 |
| 28–29 | — | Unspecified [1] |
| 30 | RI | 0 |
| 31 | LE | 0 |

[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 that contains a value identifying the specific version (model) and revision level of the processor. The contents of the PVR can be copied to a GPR by the **mfspr** instruction. Read access to the PVR is supervisor-level only; write access is not provided.

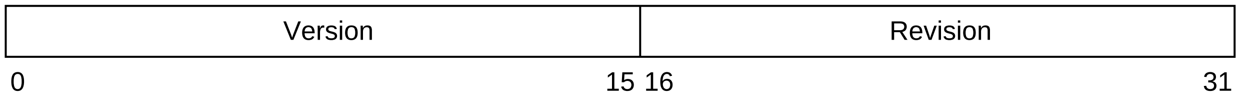


**Figure 2-13. Processor Version Register (PVR)**

The PVR consists of two 16-bit fields, described in Table 2-12.

**Table 2-12. PVR Field Descriptions**

| Bits | Name | Description |
|---|---|---|
| 0–15 | Version | A 16-bit number that uniquely identifies a particular processor version. This number can be used to determine the version of a processor; it may not distinguish between different end product models if more than one model uses the same processor. |
| 16–31 | Revision | 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 BAT Registers

The BAT registers (BATs) maintain the address translation information for eight blocks of memory. The BATs are maintained by the system software and are implemented as eight pairs of special-purpose registers (SPRs). Each block is defined by a pair of SPRs called upper and lower BAT registers. These BAT registers define the starting addresses and sizes of BAT areas.

The OEA defines eight instruction block-address translation (IBAT) registers, consisting of four pairs of instruction BATs (IBAT0U–IBAT3U and IBAT0L–IBAT3L) and eight data BATs (DBAT0U–DBAT3U and DBAT0L–DBAT3L). Figure 2-11 lists SPR numbers for BAT registers.

Figure 2-14 and Figure 2-15 show the format of the upper and lower BATs for 32-bit processors.

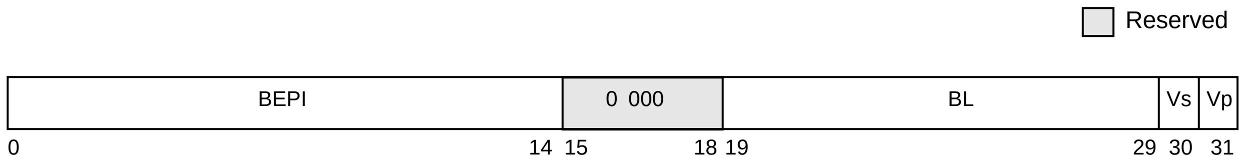
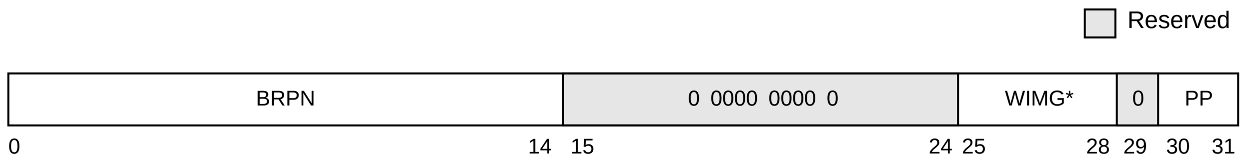


**Figure 2-14. Upper BAT Register**



*W and G bits are not defined for IBAT registers. Attempting to write to these bits causes boundedly-undefined results.

**Figure 2-15. Lower BAT Register**

Figure 2-13 describes the bits in the BAT registers.

**Table 2-14. BAT Registers—Field and Bit Descriptions**

| Upper/Lower BAT | Bits | Name | Description |
|---|---|---|---|
| Upper BAT Register | 0–14 | BEPI | Block effective page index. This field is compared with high-order bits of the logical address to determine if there is a hit in that BAT array entry. (Note that the architecture specification refers to logical address as effective address.) |
| | 15–18 | — | Reserved |
| | 19–29 | BL | Block length. BL is a mask that encodes the size of the block. Values for this field are listed in Table 2-15. |
| | 30 | Vs | Supervisor mode valid bit. This bit interacts with MSR[PR] to determine if there is a match with the logical address. For more information, see Section 7.5.2, “Recognition of Addresses in BAT Arrays." |
| | 31 | Vp | User mode valid bit. This bit also interacts with MSR[PR] to determine if there is a match with the logical address. For more information, see Section 7.5.2, “Recognition of Addresses in BAT Arrays.” |

**Table 2-14. BAT Registers—Field and Bit Descriptions (continued)**

| Upper/Lower BAT | Bits | Name | Description |
|---|---|---|---|
| Lower BAT Register | 0–14 | BRPN | This field is used in conjunction with the BL field to generate high-order bits of the physical address of the block. |
| | 15–24 | — | Reserved |
| | 25–28 | WIMG | Memory/cache access mode bits<br>W Write-through<br>I Caching-inhibited<br>M Memory coherence<br>G Guarded<br>Attempting to write to the W and G bits in IBAT registers causes boundedly-undefined results. For detailed information about the WIMG bits, see Section 5.3.1, “Memory/Cache Access Attributes." |
| | 29 | — | Reserved |
| | 30–31 | PP | Protection bits for block. This field determines the protection for the block as described in Section 7.5.4, “Block Memory Protection." |

Figure 2-15 lists the BAT area lengths encoded in BAT[BL].

**Table 2-15. BAT Area Lengths**

| BAT Area Length | BL Encoding |
|---|---|
| 128 Kbytes | 000 0000 0000 |
| 256 Kbytes | 000 0000 0001 |
| 512 Kbytes | 000 0000 0011 |
| 1 Mbyte | 000 0000 0111 |
| 2 Mbytes | 000 0000 1111 |
| 4 Mbytes | 000 0001 1111 |
| 8 Mbytes | 000 0011 1111 |
| 16 Mbytes | 000 0111 1111 |
| 32 Mbytes | 000 1111 1111 |
| 64 Mbytes | 001 1111 1111 |
| 128 Mbytes | 011 1111 1111 |
| 256 Mbytes | 111 1111 1111 |

Only the values in Table 2-15 are valid for the BL field. The rightmost BL bit is aligned with bit 14 of the logical address. A logical address is determined to be within a BAT area if the logical address matches the value in the BEPI field.

The boundary between the cleared bits and set bits in BL determines the bits of logical address that participate in the comparison with BEPI. Bits in the logical address corresponding to set bits in BL are cleared for this comparison. Bits in the logical address corresponding to set bits in the BL field, concatenated with the 17 bits of the logical address to the right (less significant bits) of BL, form the offset within the BAT area. This is described in detail in Chapter 7, “Memory Management.”

The value loaded into BL determines both the length of the BAT area and the alignment of the area in both logical and physical address space. The values loaded into BEPI and BRPN must have at least as many low-order zeros as there are ones in BL.

Use of BAT registers is described in Chapter 7, “Memory Management.”

## 2.3.4 SDR1

The SDR1 is a 64-bit register in 64-bit implementations and a 32-bit register in 32-bit implementations. The 32-bit implementation of SDR1 is shown in Figure 2-16.

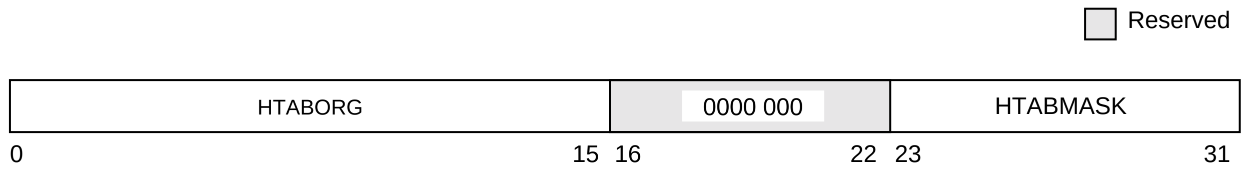


**Figure 2-16. SDR1**

The bits of the 32-bit implementation of SDR1 are described in Table 2-16.

**Table 2-16. SDR1 Bit Settings**

| Bits | Name | Description |
|---|---|---|
| 0–15 | HTABORG | The high-order 16 bits of the 32-bit physical address of the page table |
| 16–22 | — | Reserved |
| 23–31 | HTABMASK | Mask for page table address |

In 32-bit implementations, the HTABORG field in SDR1 contains the high-order 16 bits of the 32-bit physical address of the page table. Therefore, the page table is constrained to lie on a $2^{16}$-byte (64 Kbytes) boundary at a minimum. At least 10 bits from the hash function are used to index into the page table. The page table must consist of at least 64 Kbytes ($2^{10}$ PTEGs of 64 bytes each).

The page table can be any size $2^n$ where $16 \le n \le 25$. 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 HTABMASK field in SDR1 contains a mask value that determines how many bits from the hash are used in the page table index. This mask must be of the form 0b00...011...1; that is, a string of 0 bits followed by a string of 1bits. The 1 bits determine how many additional bits (at least 10) from the hash are used in the index; HTABORG must have this same number of low-order bits equal to 0.

For example, suppose that the page table is 8,192 ($2^{13}$), 64-byte PTEGs, for a total size of $2^{19}$ bytes (512 Kbytes). Note that a 13-bit index is required. Ten bits are provided from the hash initially, so 3 additional bits form the hash must be selected. The value in

HTABMASK must be 0x007 and the value in HTABORG must have its low-order 3 bits (bits 13–15 of SDR1) equal to 0. This means that the page table must begin on a $2^{3 + 10 + 6} = 2^{19}$ = 512 Kbytes boundary.

For more information, refer to Chapter 7, “Memory Management.”

## 2.3.5 Segment Registers

The segment registers contain the segment descriptors for 32-bit implementations. For 32-bit processors, the OEA defines a segment register file of sixteen 32-bit registers. Segment registers can be accessed by using the **mtsr/mfsr** and **mtsrin/mfsrin** instructions. The value of bit 0, the T bit, determines how the remaining register bits are interpreted. Figure 2-17 shows the format of a segment register when T = 0.

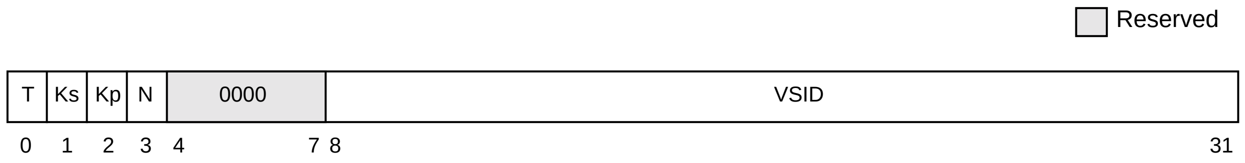


**Figure 2-17. Segment Register Format (T = 0)**

Segment register bit settings when T = 0 are described in Table 2-17.

**Table 2-17. Segment Register Bit Settings (T = 0)**

| Bits | Name | Description |
|---|---|---|
| 0 | T | T = 0 selects this format |
| 1 | Ks | Supervisor-state protection key |
| 2 | Kp | User-state protection key |
| 3 | N | No-execute protection |
| 4–7 | — | Reserved |
| 8–31 | VSID | Virtual segment ID |

Figure 2-18 shows the bit definition when T = 1.

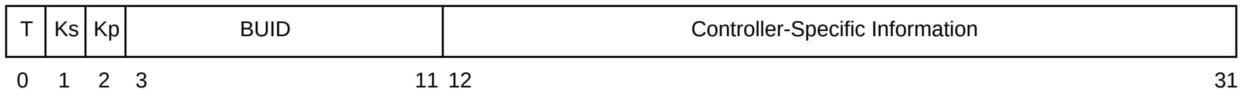


**Figure 2-18. Segment Register Format (T = 1)**

The bits in the segment register when T = 1 are described in Table 2-18.

**Table 2-18. Segment Register Bit Settings (T = 1)**

| Bits | Name | Description |
|---|---|---|
| 0 | T | T = 1 selects this format. |
| 1 | Ks | Supervisor-state protection key |

**Table 2-18. Segment Register Bit Settings (T = 1) (continued)**

| Bits | Name | Description |
|---|---|---|
| 2 | Kp | User-state protection key |
| 3–11 | BUID | Bus unit ID |
| 12–31 | CNTLR_SPEC | Device-specific data for I/O controller |

If an access is translated by the block address translation (BAT) mechanism, the BAT translation takes precedence and the results of translation using segment registers are not used. However, if an access is not translated by a BAT, and T = 0 in the selected segment register, the effective address is a reference to a memory-mapped segment. In this case, the 52-bit virtual address (VA) is formed by concatenating the following:

- The 24-bit VSID field from the segment register
- The 16-bit page index, EA[4–19]
- The 12-bit byte offset, EA[20–31]

The VA is then translated to a physical address as described in Section 7.6, “Memory Segment Model.”

If T = 1 in the selected segment register (and the access is not translated by a BAT), the effective address is a reference to a direct-store segment, defined by the architecture but not supported. No reference is made to the page tables.

## 2.3.6 Data Address Register (DAR)

The DAR is a 64-bit register in 64-bit implementations and a 32-bit register in 32-bit implementations. The DAR is shown in Figure 2-19.

**Figure 2-19. 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). For information, see Chapter 6, “Exceptions.”

## 2.3.7 SPRG0–SPRG3

SPRG0–SPRG3 are 64-bit or 32-bit registers, depending on the type of processor. They 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 2-20.

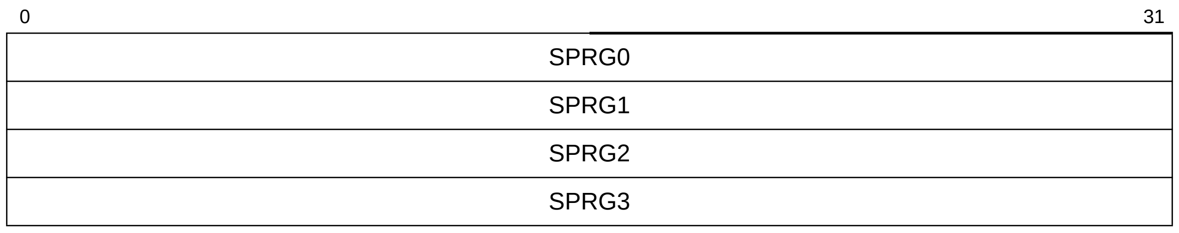


**Figure 2-20. SPRG0–SPRG3**

Table 2-19 describes typical uses of SPRG0 through SPRG3.

**Table 2-19. 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 | SPRG1 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 | SPRG2 may be used by the operating system as needed. |
| SPRG3 | SPRG3 may be used by the operating system as needed. |

## 2.3.8 DSISR

The DSISR, shown in Figure 2-21, identifies the cause of DSI and alignment exceptions.

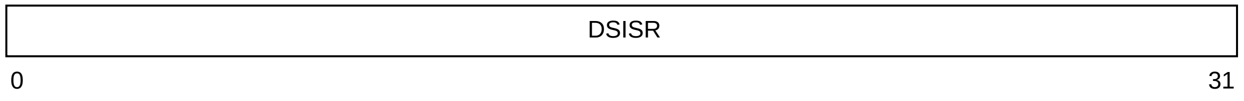


**Figure 2-21. DSISR**

For information about bit settings, see Section 6.4.3, “DSI Exception (0x00300),” and Section 6.4.6, “Alignment Exception (0x00600).”

## 2.3.9 Machine Status Save/Restore Register 0 (SRR0)

The SRR0 is used to save machine status on exceptions and restore machine status when an **rfi** 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 2-22.

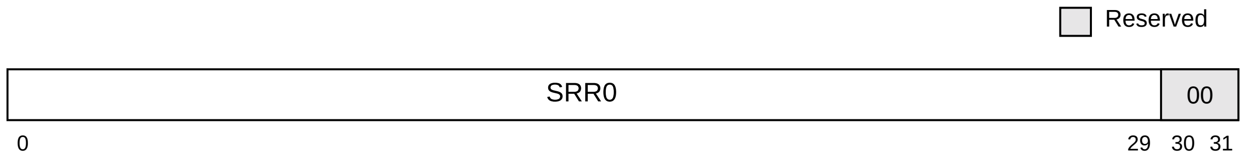


**Figure 2-22. Machine Status Save/Restore Register 0 (SRR0)**

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.

When an **rfi** 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.

Note that 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.”

## 2.3.10 Machine Status Save/Restore Register 1 (SRR1)

The SRR1 is a 64-bit register in 64-bit implementations and a 32-bit register in 32-bit implementations. SRR1 is used to save machine status on exceptions and to restore machine status when an **rfi** instruction is executed. Figure 2-23 shows the SRR1 format.

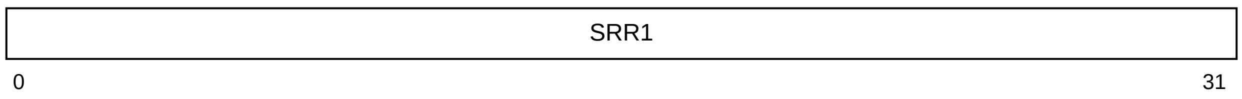


**Figure 2-23. Machine Status Save/Restore Register 1 (SRR1)**

When an exception occurs, SRR1[1–4,10–15] are loaded with exception-specific information and MSR[16–23,25–27,30–31] are placed in corresponding SRR1 bit positions. When **rfi** executes, MSR[16–23,25–27,30–31] are loaded from SRR1[16–23,25–27,30–31].

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 an **rfi**.

Note that, 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.”

## 2.3.11 Floating-Point Exception Cause Register (FPECR)

FPECR may be used to identify the cause of a floating-point exception. Note that the FPECR is an optional register in the 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.

## 2.3.12 Time Base Facility (TB)—OEA

As described in Section 2.2, “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 at start-up.

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

### 2.3.12.1 Writing to the Time Base

Note that 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 F, “Simplified Mnemonics.” 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 64- or 32-bit mode.

The TB can be written by a sequence such as the following:

```
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.”

## 2.3.13 Decrementer Register (DEC)

The decrementer register (DEC), shown in Figure 2-24, 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.

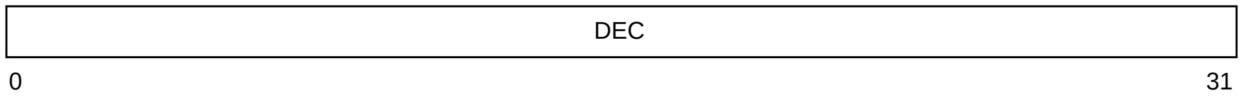


**Figure 2-24. Decrementer Register (DEC)**

### 2.3.13.1 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.

### 2.3.13.2 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
```

## 2.3.14 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 in 64-bit implementations and a 32-bit register in 32-bit implementations. The data address breakpoint facility is optional to the architecture. 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 double word. The address comparison is done on an effective address, and it applies to data accesses only. It does not apply to instruction fetches.

The DABR is shown in Figure 2-25.

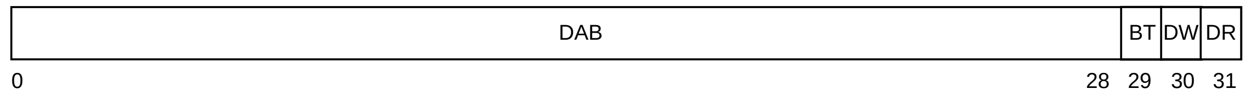


**Figure 2-25. Data Address Breakpoint Register (DABR)**

Table 2-20 describes the fields in the DABR.

**Table 2-20. DABR—Bit Settings**

| Bits | Name | Description |
|---|---|---|
| 0–28 | DAB | Data address breakpoint |
| 29 | BT | Breakpoint translation enable |
| 30 | DW | Data write enable |
| 31 | 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[0–28] = 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.

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

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

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

A match generates a DSI exception. Section 6.4.3, “DSI Exception (0x00300),” gives more information on the data address breakpoint facility.

## 2.3.15 External Access Register (EAR)

The 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 2-26.

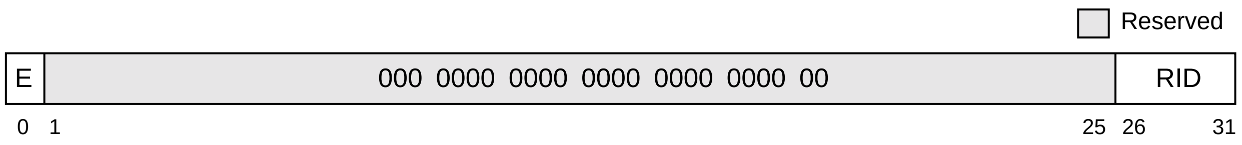


**Figure 2-26. External Access Register (EAR)**

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. EAR bit settings described in Table 2-21. 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 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 supports a graphics adapter, **ecowx** could be used to send the translated physical address of a buffer containing graphics data to the graphics device; **eciwx** could be used to load status information from the graphics adapter.

**Table 2-21. External Access Register (EAR) Bit Settings**

| Bits | Name | Description |
|---|---|---|
| 0 | E | Enable bit<br>0 Disabled. **eciwx** or **ecowx** causes a DSI exception.<br>1 Enabled. **eciwx** and **ecowx** can perform the specified external operation. |
| 1–25 | — | Reserved |
| 26–31 | RID | Resource ID |

EAR can be accessed by using the **mtspr** and **mfspr**. Table 2-23 and Table 2-24 show EAR synchronization requirements.

## 2.3.16 Processor Identification Register (PIR)

The optional, 32-bit processor identification register (PIR) is a read-only register that contains a value that can be used to distinguish the processor from other processors in the system. The contents of the PIR can be copied to a GPR by the **mfspr** instruction.

Read access to the PIR is privileged; write access, if provided, is implementation dependent.

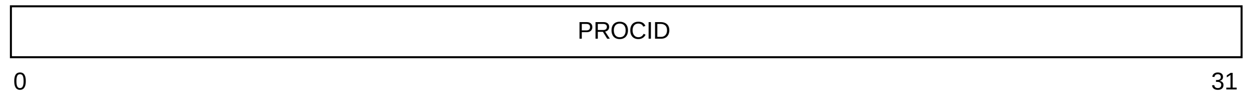


**Figure 2-27. Processor Identification Register**

**Table 2-22. PID Field Description**

| Bits | Name | Description |
|---|---|---|
| 0–31 | PROCID | Processor ID |

## 2.3.17 Synchronization Requirements for Special Registers and for Lookaside Buffers

Changing the value in certain system registers, and invalidating 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 2-23 for data access and Table 2-24 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 an instruction sequence alters the context but contains no instructions affected the alterations, no software synchronization is required within the sequence.

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

No software synchronization is needed before altering the MSR (except when altering MSR[POW] or MSR[LE]; see Table 2-23 and Table 2-24), because **mtmsr** is execution synchronizing. No software synchronization is required before most of the other alterations shown in Table 2-24, because 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 2-23 provides information on data access synchronization requirements.

**Table 2-23. Data Access Synchronization**

| Instruction/Event | Required Prior | Required After |
|---|---|---|
| Exception [1] | None | None |
| **rfi** [1] | None | None |
| sc [1] | None | None |
| Trap [1] | None | None |
| **mtmsr** (ILE) | None | None |
| **mtmsr** (PR) | None | Context-synchronizing instruction |
| **mtmsr** (ME) [2] | None | Context-synchronizing instruction |
| **mtmsr** (DR) | None | Context-synchronizing instruction |
| **mtmsr** (LE) [3] | — | — |
| mtsr [or mtsrin] | Context-synchronizing instruction | Context-synchronizing instruction |
| mtspr (SDR1) [4, 5] | **sync** | Context-synchronizing instruction |
| mtspr (DBAT) | Context-synchronizing instruction | Context-synchronizing instruction |
| mtspr (DABR) [6] | — | — |
| mtspr (EAR) | Context-synchronizing instruction | Context-synchronizing instruction |
| tlbie [7, 8] | Context-synchronizing instruction | Context-synchronizing instruction or **sync** |
| tlbia [7, 8] | Context-synchronizing instruction | Context-synchronizing instruction or **sync** |

[1] Synchronization requirements for changing the power conserving mode are implementation-dependent.

[2] A context synchronizing instruction is required after modification of the MSR[ME] bit to ensure that the modification takes effect for subsequent machine check exceptions, which may not be recoverable and therefore may not be context synchronizing.

[3] Synchronization requirements for changing from one endian mode to the other are implementation-dependent.

[4] SDR1 must not be altered when MSR[DR] = 1 or MSR[IR] = 1; if it is, the results are undefined.

[5] A **sync** instruction is required before the **mtspr** instruction because SDR1 identifies the page table and thereby the location of the referenced and changed (R and C) bits. To ensure that R and C bits are updated in the correct page table, SDR1 must not be altered until all R and C bit updates due to instructions before the **mtspr** have completed. A **sync** instruction guarantees this synchronization of R and C bit updates, while neither a context synchronizing operation nor the instruction fetching mechanism does so.

[6] Synchronization requirements for changing the DABR are implementation-dependent.

[7] For data accesses, the context synchronizing instruction before the **tlbie**, or **tlbia** instruction ensures that all memory accesses, due to preceding instructions, have completed to a point at which they have reported all exceptions that may be caused. The context synchronizing instruction after the **tlbie**, or **tlbia** ensures that subsequent memory accesses will not use the TLB entry(s) being invalidated. It does not ensure that all memory accesses previously translated by the TLB entry(s) being invalidated have completed with respect to memory or, for **tlbie** or **tlbia**, that R and C bit updates associated with those memory accesses have completed; if these completions must be ensured, the **tlbie**, or **tlbia** must be followed by a **sync** instruction rather than by a context synchronizing instruction.

[8] Multiprocessor systems have other requirements to synchronize TLB invalidate.

For information on instruction access synchronization requirements, see Table 2-24.

**Table 2-24. Instruction Access Synchronization**

| Instruction/Event | Required Prior | Required After |
|---|---|---|
| Exception [1] | None | None |
| **rfi** [1] | None | None |
| **sc** [1] | None | None |
| Trap [1] | None | None |
| **mtmsr** (POW) [1] | — | — |
| **mtmsr** (ILE) | None | None |
| **mtmsr** (EE) [2] | None | None |
| **mtmsr** (PR) | None | Context-synchronizing instruction |
| **mtmsr** (FP) | None | Context-synchronizing instruction |
| **mtmsr** (ME) [3] | None | Context-synchronizing instruction |
| **mtmsr** (FE0, FE1) | None | Context-synchronizing instruction |
| **mtmsr** (SE, BE) | None | Context-synchronizing instruction |
| **mtmsr** (IP) | None | None |
| **mtmsr** (IR) [4] | None | Context-synchronizing instruction |
| **mtmsr** (RI) | None | None |
| **mtmsr** (LE) [5] | — | — |
| **mtsr** [or **mtsrin**] [4] | None | Context-synchronizing instruction |
| **mtspr** (SDR1) [6, 7] | **sync** | Context-synchronizing instruction |
| **mtspr** (IBAT) [4] | None | Context-synchronizing instruction |
| **mtspr** (DEC) [8] | None | None |
| **tlbie** [9, 10] | None | Context-synchronizing instruction or **sync** |
| **tlbia** [9, 10] | None | Context-synchronizing instruction or **sync** |

[1] Synchronization requirements for changing the power conserving mode are implementation-dependent.

[2] The effect of altering the EE bit is immediate as follows:

- If an **mtmsr** clears EE, neither an external interrupt nor a decrementer exception can occur after the instruction is executed.
- If an **mtmsr** sets EE, when an external interrupt, decrementer exception, or higher priority exception exists, the corresponding exception occurs immediately after the **mtmsr** is executed, and before the next instruction is executed in the program that set MSR[EE].

[3] A context synchronizing instruction is required after modification of MSR[ME] to ensure that the modification takes effect for subsequent machine check exceptions, which may not be recoverable and therefore may not be context synchronizing.

[4] The alteration must not cause an implicit branch in physical address space. The physical 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.

[5] Synchronization requirements for changing from one endian mode to the other are implementation-dependent.

[6] SDR1 must not be altered when MSR[DR] = 1 or MSR[IR] = 1; if it is, the results are undefined.

[7] A **sync** instruction is required before the **mtspr** instruction because SDR1 identifies the page table and thereby the location of the referenced and changed (R and C) bits. To ensure that R and C bits are updated in the correct page table, SDR1 must not be altered until all R and C bit updates due to instructions before the **mtspr** have completed. A **sync** instruction guarantees this synchronization of R and C bit updates, while neither a context synchronizing operation nor the instruction fetching mechanism does so.

[8] The elapsed time between the content of the decrementer becoming negative and the signaling of the decrementer exception is not defined.

[9] For data accesses, the context synchronizing instruction before the **tlbie**, or **tlbia** instruction ensures that all memory accesses, due to preceding instructions, have completed to a point at which they have reported all exceptions that may be caused. The context synchronizing instruction after the **tlbie**, or **tlbia** ensures that subsequent memory accesses will not use the TLB entry(s) being invalidated. It does not ensure that all memory accesses previously translated by the TLB entry(s) being invalidated have completed with respect to memory or, for **tlbie** or **tlbia**, that R and C bit updates associated with those memory accesses have completed; if these completions must be ensured, the **tlbie**, or **tlbia** must be followed by a **sync** instruction rather than by a context synchronizing instruction.

[10] Multiprocessor systems have other requirements to synchronize TLB invalidate.

# Chapter 3
# Operand Conventions

This chapter describes the operand conventions as they are represented in the 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 registers and representing data in these registers in both big- and little-endian modes. The floating-point data formats and exception conditions are also described. Refer to Appendix D, "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

U

Bytes in memory are numbered consecutively starting with 0. Each number is the address of the corresponding byte. Memory operands may be bytes, half words, words, or double words, 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, its lowest-numbered byte). Operand length is implicit for each instruction.

### 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. That is, the natural address of an operand is an integral multiple of its length. An operand not aligned at its natural boundary is considered misaligned. Instructions are always 4 bytes long and word-aligned. Table 3-1 shows operand characteristics for single-register memory access instructions.

**Table 3-1. Memory Operand Alignment**

| Operand | Length | Aligned Addr(60–63) [1] |
|---|---|---|
| Byte | 8 bits | xxxx |
| Half word | 2 bytes | xxx0 |
| Word | 4 bytes | xx00 |
| Double word | 8 bytes | x000 |
| Quad word (Although not permitted as operands, quad-word alignment is desirable for certain memory operands.) | 16 bytes | 0000 |

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

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.

The smallest addressable memory unit is the byte (8 bits), and scalars are composed of one or more sequential bytes. When a 32-bit scalar is moved from a register to memory, it occupies four consecutive bytes in memory, and a decision must be made regarding the order of these bytes in these four addresses.

Both big- and little-endian byte ordering are supported; the default is big-endian. Big- and little-endian byte orderings are described as follows:

- Big-endian byte ordering (default). 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.
- Little-endian byte ordering. For little-endian scalars, the LSB is stored at the lowest (or starting) address while the MSB 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 defines data structure *S* is used in this section to demonstrate how the bytes that comprise each element (*a*, *b*, *c*, *d*, *e*, and *f*) are mapped into memory. The structure contains scalars (shown in hexadecimal in the comments) and a sequence of characters, shown in single quote marks.

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

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

### 3.1.3.1 Big-Endian Mapping

Figure 3-2 shows the big-endian mapping of the structure. Note that the MSB of each scalar is at the lowest address. The mapping uses padding (indicated by (x)) to align the scalars—4 bytes between elements *a* and *b*, 1 byte between *d* and *e*, and 2 bytes between *e* and *f*. Note that the padding is determined by the compiler, not the architecture.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 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 |

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

### 3.1.3.2 Little-Endian Mapping

Figure 3-3 shows the structure using little-endian mapping. Note that the LSB of each scalar is at the lowest address.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 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. Little-Endian Mapping of Structure *S***

Figure 3-3 shows the sequence of double words laid out with addresses increasing from left to right. Programmers familiar with little-endian byte ordering may be more accustomed to viewing double words 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.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 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 |

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

## 3.1.4 Byte Ordering

The architecture supports both big- and little-endian byte ordering; however, the code sequence to switch modes may differ among processors. Byte ordering is specified through two MSR bits. MSR[LE] (little-endian mode) indicates the endian mode in which the processor is currently operating; MSR[ILE] (exception little-endian mode) specifies the mode to be used when an exception handler is invoked. When an exception occurs, MSR[ILE] (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 architecture 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.”

### 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 3 low-order effective address bits are exclusive-ORed (XOR) with a 3-bit value that depends on operand length, as shown in Table 3-2. This modification is sometimes called munging.

**Table 3-2. EA Modifications**

| Data Width (Bytes) | EA Modification |
|---|---|
| 8 | No change |
| 4 | XOR with 0b100 |
| 2 | XOR with 0b110 |
| 1 | XOR with 0b111 |

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

Modifying the address 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 double words. Only the address is modified, not the byte order.

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

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 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 |

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

Note that the mapping shown in Figure 3-5 is not a true little-endian mapping of the structure *S*. However, because the processor modifies 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.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 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 |

**Figure 3-6. Modified Little-Endian Structure *S* as Seen by the Processor**

Note that the mapping in Figure 3-6 is identical to the little-endian mapping 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 considered when performing I/O operations in little-endian mode, as described in Section 3.1.4.5, “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. Table 3-7 shows a true little-endian mapping of the four-byte word 0x1112_1314, 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 |

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

For the true little-endian example in Figure 3-7, the LSB (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 MSB (0x11) is stored at address 0x08.

When a 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, memory synchronization (excluding **sync**), or floating-point load/store (including **stfiwx**) instructions. 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 architecture requires that half words, words, and double words 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. However, because processors in little-endian mode modify the effective address, the byte order 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 LSB (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 MSB (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.

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

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

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

Note that the misaligned word in this example spans two double words. 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 double word may be supported, while one that spans double words may cause an alignment exception.

### 3.1.4.3 Nonscalars

Two types of instructions handle nonscalars (multiple instances of scalars):

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

Address modification cannot be used because these instructions typically operate on more than one word-length scalar. These instructions cause alignment exception conditions when the processor executes in little-endian mode. String accesses are not supported, but they are inherently byte-based operations and can be broken into a series of word-aligned accesses.

#### 3.1.4.4 Instruction Addressing in Little-Endian Mode

Instructions are word-aligned in memory. They are fetched as if the current instruction address is incremented by four for each sequential instruction. In little-endian mode, the instruction address is XORed with 0b100 as described in Section 3.1.4.1, “Aligned Scalars in Little-Endian Mode.” A program is thus an array of little-endian words with each word fetched and executed in order (not including branches).

Instruction addresses visible to an executing program are the effective addresses 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 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.
- Memory locations that contain the first set of instructions executed by each kind of exception handler must be set consistently with the endian mode in which the exception handler is invoked. Thus, if the 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 double word reversed from the order in which they are to be executed.

#### 3.1.4.5 Input/Output Data Transfer Addressing in Little-Endian Mode

In big-endian mode, the processor and memory subsystem recognize the same byte as byte 0. However, this is not true in little-endian mode because of the address bits modified when the processor accesses memory.

I/O transfers in little-endian mode 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 systems must be performed using only 1-byte transfers. Transfers can be as wide as desired, but the byte order within double words must be as if they were fetched or stored one at a time. That is, in a true little-endian I/O device, the system must provide a way to modify and unmodify addresses and reverse the bytes within a double word (MSB to LSB).

## 3.2 Operand Placement and Performance—VEA

The 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 5.2, “The Virtual Environment.”

### 3.2.1 Summary of Performance Effects

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

The performance of accesses varies depending on the following:

- Operand size and alignment
- Endian mode (big-endian or little-endian)
- Whether a cache block, page, BAT, or segment boundary is crossed

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 | | | |
|---|---|---|---|---|---|---|
| **Type** | **Size** | **Byte Alignment** | **None** | **Cache Block** | **Page** | **BAT/Segment** |
| Integer | 8 byte | 8<br>4<br><4 | Optimal<br>Good<br>Poor | —<br>Good<br>Poor | —<br>Poor<br>Poor | —<br>Poor<br>Poor |
| | 4 byte | 4<br><4 | Optimal<br>Good | —<br>Good | —<br>Poor | —<br>Poor |
| | 2 byte | 2<br><2 | Optimal<br>Good | —<br>Good | —<br>Poor | —<br>Poor |
| | 1 byte | 1 | Optimal | — | — | — |
| | **lmw**, **stmw** | 4 | Good | Good | Good [1] | Poor |
| | String | — | Good | Good | Poor | Poor |
| Floating point | 8 byte | 8<br>4<br><4 | Optimal<br>Good<br>Poor | —<br>Good<br>Poor | —<br>Poor<br>Poor | —<br>Poor<br>Poor |
| | 4 byte | 4<br><4 | Optimal<br>Poor | —<br>Poor | —<br>Poor | —<br>Poor |

[1] Crossing a page boundary where the memory/cache access attributes of the two pages differ is equivalent to crossing a segment boundary and thus has poor performance.

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 | | | |
|---|---|---|---|---|---|---|
| **Type** | **Size** | **Byte Alignment** | **None** | **Cache Block** | **Page** | **BAT/Segment** |
| Integer | 8 byte | 8<br><8 | Optimal<br>Poor | —<br>Poor | —<br>Poor | —<br>Poor |
| | 4 byte | 4<br><4 | Optimal<br>Poor | —<br>Poor | —<br>Poor | —<br>Poor |
| | 2 byte | 2<br><2 | Optimal<br>Poor | —<br>Poor | —<br>Poor | —<br>Poor |
| | 1 byte | 1 | Optimal | — | — | — |
| Floating point | 8 byte | 8<br><8 | Optimal<br>Poor | —<br>Poor | —<br>Poor | —<br>Poor |
| | 4 byte | 4<br><4 | Optimal<br>Poor | —<br>Poor | —<br>Poor | —<br>Poor |

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

## 3.2.2 Instruction Restart

The execution of a memory access instruction may abort after part of an access is performed for several reasons. For example, if a program attempts to access a page for the first time or when the processor must check for a change in the memory and cache access attributes when an access crosses a page boundary. When this occurs, the processor or operating system may restart the instruction, in which case, some bytes at that location may be loaded from or stored to the target location a second time.

The following rules apply to memory accesses with regard to restarting the instruction:

- Aligned accesses—A single-register instruction that accesses an aligned operand is never restarted (that is, it is not partially executed).
- Misaligned accesses—A single-register instruction that accesses a misaligned operand may be restarted if the access crosses a page, BAT, or segment boundary, or if the processor is in little-endian mode.
- Load/store multiple, load/store string instructions—These instructions may be restarted if, in accessing the locations specified by the instruction, a page, BAT, or segment boundary is crossed.

Programmers should assume that any misaligned access in a segment might be restarted. When the processor is in big-endian mode, software can ensure that misaligned accesses are not restarted by placing the misaligned data in BAT areas, as BAT areas have no internal protection boundaries. See Section 7.5, “Block Address Translation.”

## 3.3 Floating-Point Execution Models—UISA

The architecture supports the two following types of floating-point instructions:

U

- Computational instructions including IEEE-754 defined operations for 64- and 32-bit arithmetic (addition, subtraction, multiplication, division, extracting the square root, rounding conversion, comparison, and combinations of these) and architecture-defined multiply-add and reciprocal estimate instructions.
- Noncomputational instructions—floating-point load, store, and move instructions.

Although computational and noncomputational instructions are governed by MSR[FP] (that allows floating-point instructions to be executed), only 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, double- to single-precision conversion must be done explicitly by software; single- to double-precision conversion is done implicitly by the processor.

All 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 D, “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 UISA defines a 32-bit format for a single-precision floating-point value and a 64-bit format for a double-precision floating-point value. Floating-point data in memory may be in single- or double-precision format, however both single- and double-precision values in floating-point registers (FPRs) are stored in double-precision format.

Figure 3-9 shows single-precision format; Figure 3-10 shows double-precision format.

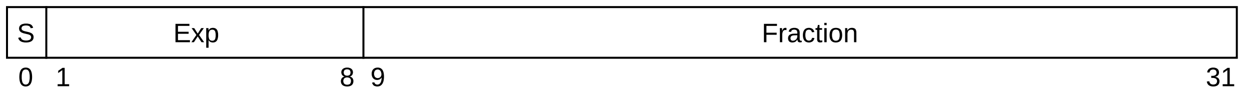


**Figure 3-9. Floating-Point Single-Precision Format**

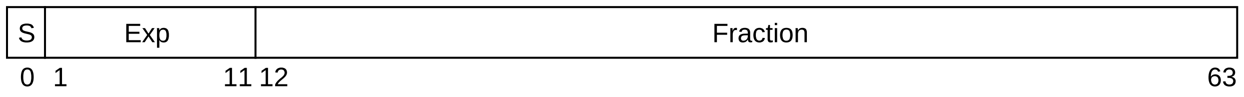


**Figure 3-10. Floating-Point Double-Precision Format**

Both formats 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 half word (or word in the case of floating-point double-precision format), the value affected depends on whether the system is using big- or little-endian byte ordering, as described in Section 3.1.2, “Byte Ordering.” 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 (the first bit to the left of the binary point) is 1 for normalized numbers and 0 for denormalized numbers. Values representable in the two floating-point formats can be specified by the parameters 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 |

As Table 3-6 shows, an exponent's true value can be determined by subtracting 127 for single-precision numbers and 1023 for double-precision numbers. Note that an exponent of all ones indicates an infinity or NaN; all zeros indicates zero or a denormalized number.

**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 UISA defines numerical and nonnumerical values representable in single- and double-precision formats. Numerical values are approximations to real numbers, including normalized numbers, denormalized numbers, and zero values. Representable nonnumerical values are positive and negative infinities and NaNs. 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. Figure 3-11 shows the relative location of defined numerical entities on a real number line. Tiny values include denormalized numbers and numbers that too small to be represented for a particular precision format, but do not include ±0.

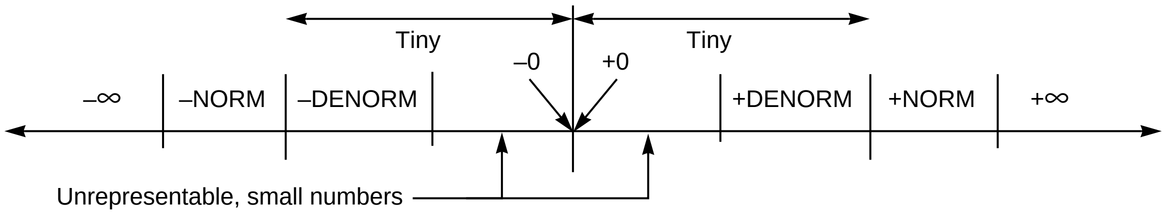


**Figure 3-11. Approximation to Real Numbers**

Positive and negative NaNs convey diagnostic information such as 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 approximations of real numbers. Three categories are supported—normalized numbers, denormalized numbers, and zeros.

### 3.3.1.3 Normalized Numbers (±NORM)

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

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

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

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

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. Figure 3-12 shows the format for normalized numbers.

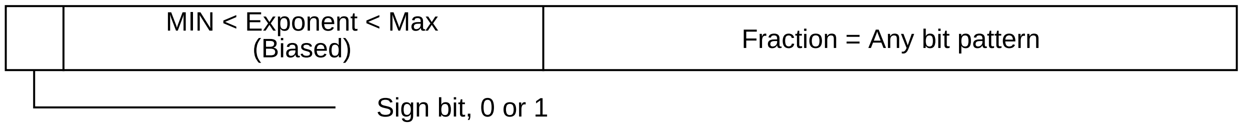


Figure 3-12. Format for Normalized Numbers

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, Figure 3-13, have a biased exponent value of zero and fraction of zero. Zeros can have a positive or negative sign. Comparison operations ignore the sign (that is, +0 = –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 Section 3.3.6.1.1, “Invalid Operation Exception Condition.” Rounding a zero result affects only the sign (±0).

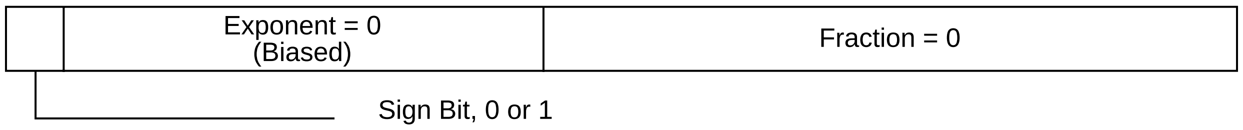


**Figure 3-13. Format for Zero Numbers**

### 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.

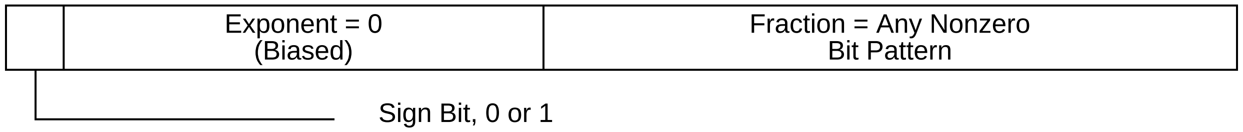


**Figure 3-14. Format for Denormalized Numbers**

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 (±∞)

Infinities have a maximum biased exponent value of 255 in single-precision format, 2047 in double-precision format, and a zero fraction value. Infinities 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 real numbers can be related by ordering in the affine sense:

–∞ < every finite number < +∞

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

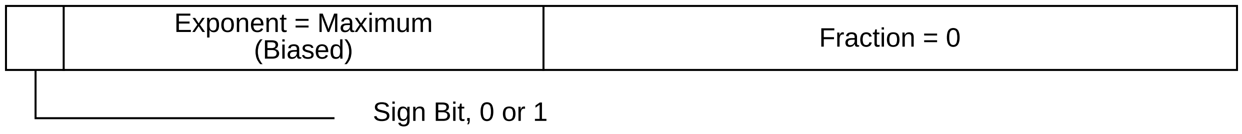


**Figure 3-15. Format for Positive and Negative Infinities**

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 Section 3.3.6.1.1, “Invalid Operation Exception Condition.”

### 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).

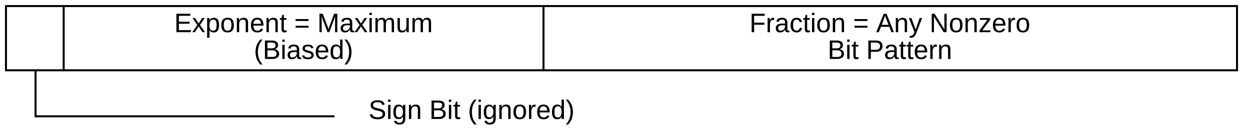


**Figure 3-16. Format for NaNs**

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

QNaNs 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). QNaNs propagate through all operations, except floating-point round to single-precision, ordered comparison, and conversion to integer operations. They 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. 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. This is shown in Figure 3-17.

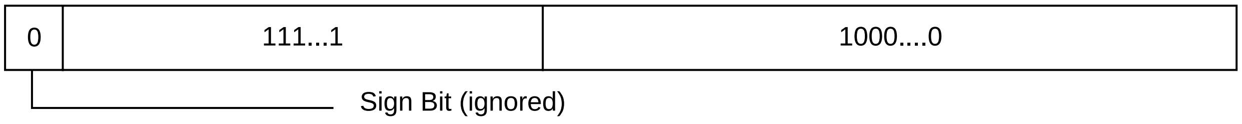


**Figure 3-17. Representation of Generated QNaN**

## 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 ±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 instructions, these rules are applied first to the multiplication operation and then to the addition or subtraction operation (one of the source operands to the addition or 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 Section D.1, “Execution Model for IEEE Operations,” for information about the guard and round bits. During normalization, the exponent is regarded as if its range were 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 Section 3.3.6.2.2, “Underflow Exception Condition.” 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 **frsp***x* instruction
- 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.

### 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 FPSCR enable bits. 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 CR 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, FPSCR and CR 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.

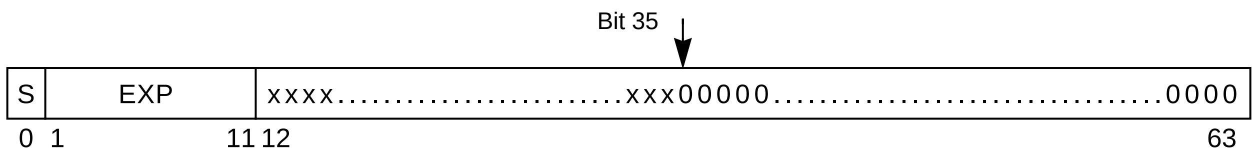

Bit 35
S
EXP
xxxx.......................xxx00000...............................0000
0
1
11
12
63

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

The **frsp***x* instruction allows conversion from double- to single-precision with appropriate exception checking and rounding. It is used to convert double-precision floating-point values (produced by double-precision load and arithmetic instructions) 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. If double-precision accuracy is not required, using single-precision data and instructions may speed operations in some implementations.

## 3.3.5 Rounding

All arithmetic, rounding, and conversion instructions defined by the 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 how loss of accuracy is detected.

Let Z be the intermediate result (with infinite precision and unbounded range) or the operand of a conversion operation. If Z can be represented exactly in the target format, 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; 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.

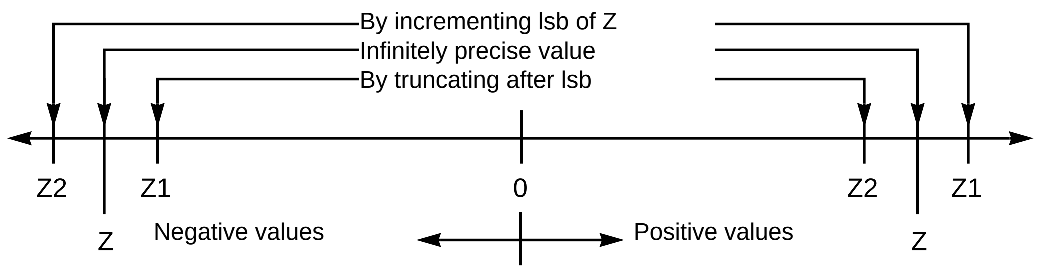


**Figure 3-19. Relation of Z1 and Z2**

Table 3-8 describes the four rounding modes available through FPSCR[RN].

**Table 3-8. FPSCR[RN] Setting**

| RN | Rounding Mode | Rules |
|---|---|---|
| 00 | Round to nearest | Choose the best approximation (Z1 or Z2). If a tie, choose the one that is even (llsb 0). |
| 01 | Round toward zero | Choose the smaller in magnitude (Z1 or Z2). |
| 10 | Round toward +∞ | Choose Z1. |
| 11 | Round toward –∞ | Choose Z2. |

See Section D.1, “Execution Model for IEEE Operations,” 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 Section 3.3.6.2.2, “Underflow Exception Condition.” Figure 3-20 shows the selection of Z1 and Z2 for the four possible rounding modes provided by FPSCR[RN].

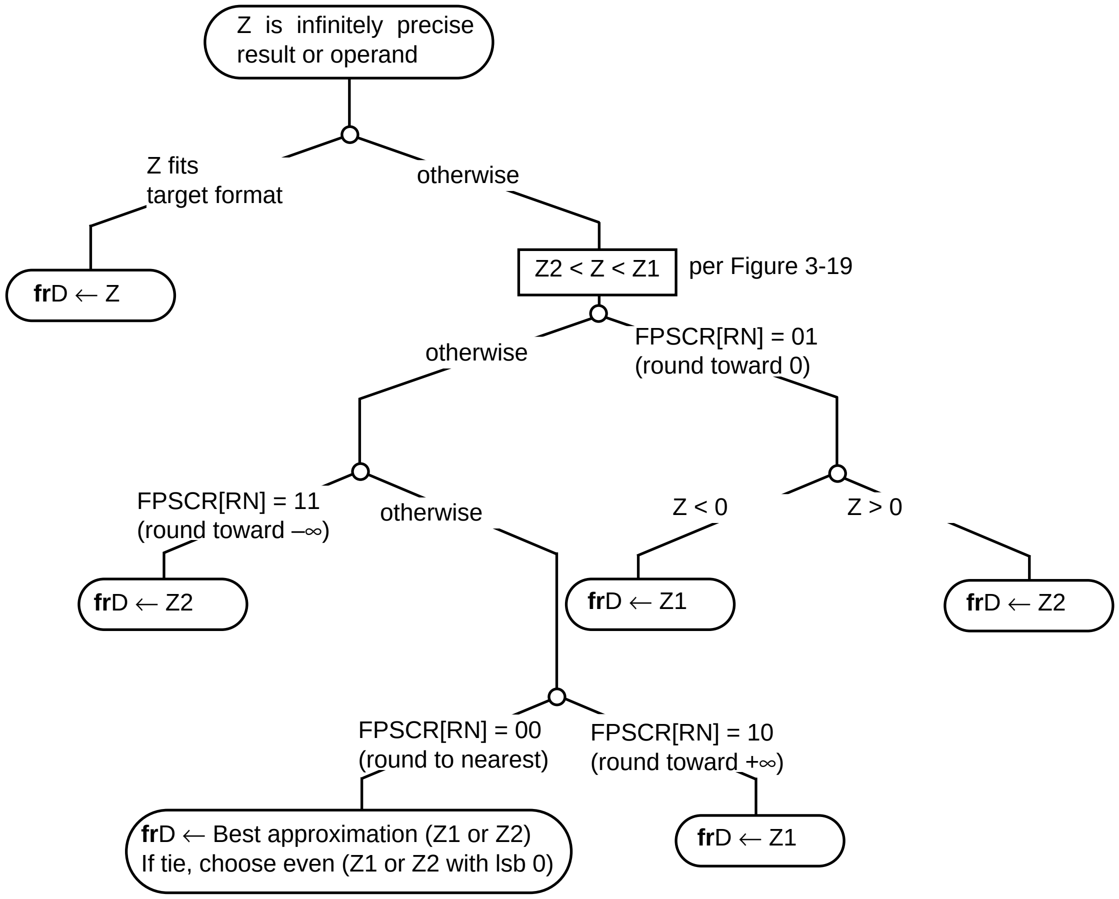

Z is infinitely precise result or operand
Z fits target format
otherwise
frD ← Z
Z2 < Z < Z1
per Figure 3-19
otherwise
FPSCR[RN] = 01 (round toward 0)
FPSCR[RN] = 11 (round toward –∞)
otherwise
Z < 0
Z > 0
frD ← Z2
frD ← Z1
frD ← Z2
FPSCR[RN] = 00 (round to nearest)
FPSCR[RN] = 10 (round toward +∞)
frD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

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

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.

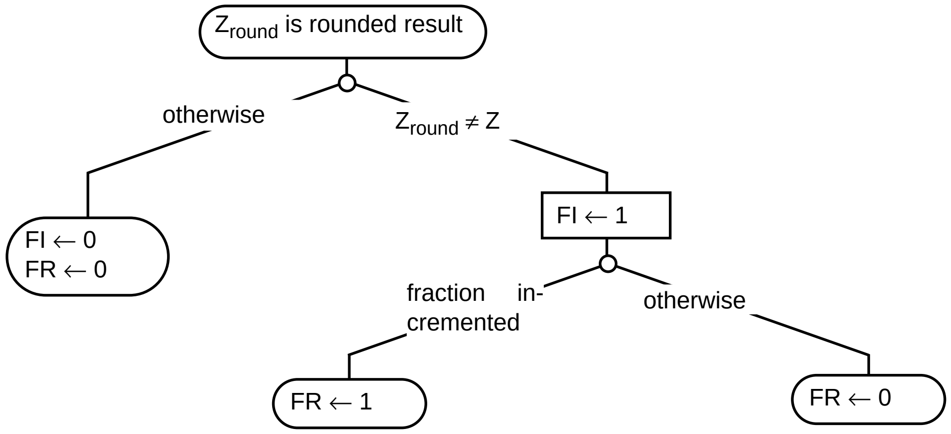


**Figure 3-21. Rounding Flags in FPSCR**

## 3.3.6 Floating-Point Program Exceptions

Only computational instructions can cause floating-point enabled exceptions (subsets of the program exception). Floating-point program exceptions are signaled by FPSCR condition bits described here and in Chapter 2, “Register Set.” These bits correspond to 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 6.4.7, “Program Exception (0x00700).”

The FPSCR is shown in Figure 3-22.

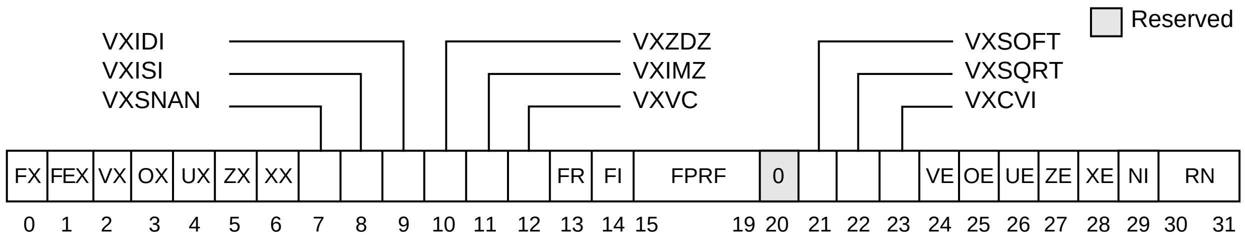


**Figure 3-22. Floating-Point Status and Control Register (FPSCR)**

Table 3-9 describes FPSCR bit settings.

**Table 3-9. FPSCR Bit Settings**

| Bits | 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. |

**Table 3-9. FPSCR Bit Settings (continued)**

| Bits | Name | Description |
|---|---|---|
| 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 exception bits as described in Section 3.3.6.1.1, “Invalid Operation Exception Condition.” 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.” |
| 4 | UX | Floating-point underflow exception. This is a sticky bit. See Section 3.3.6.2.2, “Underflow Exception Condition.” |
| 5 | ZX | Floating-point zero divide exception. This is a sticky bit. See Section 3.3.6.1.2, “Zero Divide Exception Condition.” |
| 6 | XX | Floating-point inexact exception. This is a sticky bit. See Section 3.3.6.2.3, “Inexact Exception Condition.” XX is the sticky version of FPSCR[FI]. The following describes how XX is set by a given instruction:<br>• 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].<br>• 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 Section 3.3.6.1.1, “Invalid Operation Exception Condition.” |
| 8 | VXISI | Floating-point invalid operation exception for $\infty - \infty$. This is a sticky bit. See Section 3.3.6.1.1, “Invalid Operation Exception Condition.” |
| 9 | VXIDI | Floating-point invalid operation exception for $\infty \div \infty$. This is a sticky bit. See Section 3.3.6.1.1, “Invalid Operation Exception Condition.” |
| 10 | VXZDZ | Floating-point invalid operation exception for $0 \div 0$. This is a sticky bit. See Section 3.3.6.1.1, “Invalid Operation Exception Condition.” |
| 11 | VXIMZ | Floating-point invalid operation exception for $\infty * 0$. This is a sticky bit. See Section 3.3.6.1.1, “Invalid Operation Exception Condition.” |
| 12 | VXVC | Floating-point invalid operation exception for invalid compare. This is a sticky bit. See Section 3.3.6.1.1, “Invalid Operation Exception Condition.” |
| 13 | FR | Floating-point fraction rounded. The last arithmetic, rounding, or conversion instruction incremented the fraction. See Section 3.3.5, “Rounding.” This bit is not sticky. |
| 14 | FI | Floating-point fraction inexact. The last arithmetic, rounding, or conversion instruction either produced an inexact result during rounding 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)

| Bits | Name | Description |
|---|---|---|
| 15–19 | FPRF | Floating-point result flags. For arithmetic, rounding, and conversion instructions, FPRF 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.<br>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.<br>Bits 16–19 comprise the 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.<br>16 Floating-point less than or negative (FL or <)<br>17 Floating-point greater than or positive (FG or >)<br>18 Floating-point equal or zero (FE or =)<br>19 Floating-point unordered or NaN (FU or ?)<br>Note that 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 Section 3.3.6.1.1, “Invalid Operation Exception Condition.” |
| 22 | VXSQRT | Floating-point invalid operation exception for invalid square root. This is a sticky bit. For more detailed information, refer to Section 3.3.6.1.1, “Invalid Operation Exception Condition.” |
| 23 | VXCVI | Floating-point invalid operation exception for invalid integer convert. This is a sticky bit. See Section 3.3.6.1.1, “Invalid Operation Exception Condition.” |
| 24 | VE | Floating-point invalid operation exception enable. See Section 3.3.6.1.1, “Invalid Operation Exception Condition.” |
| 25 | OE | IEEE floating-point overflow exception enable. See Section 3.3.6.2, “Overflow, Underflow, and Inexact Exception Conditions.” |
| 26 | UE | IEEE floating-point underflow exception enable. See Section 3.3.6.2.2, “Underflow Exception Condition.” |
| 27 | ZE | IEEE floating-point zero divide exception enable. See Section 3.3.6.1.2, “Zero Divide Exception Condition.” |
| 28 | XE | Floating-point inexact exception enable. See Section 3.3.6.2.3, “Inexact Exception Condition.” |
| 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 *i*mplementation-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.<br>Effects of the setting of this bit are implementation-dependent. |
| 30–31 | RN | Floating-point rounding control. See Section 3.3.5, “Rounding.”<br>00 Round to nearest<br>01 Round toward zero<br>10 Round toward +infinity<br>11 Round toward –infinity |

Table 3-10 describes the floating-point result flags, FPSCR[FPRF].

**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 Section 3.3.6.1.1, “Invalid Operation Exception Condition.”
- Zero divide exception condition (ZX). These exception conditions are described in Section 3.3.6.1.2, “Zero Divide Exception Condition.”
- Overflow Exception Condition (OX). These exception conditions are described in Section 3.3.6.2.1, “Overflow Exception Condition.”
- Underflow Exception Condition (UX). These exception conditions are described in Section 3.3.6.2.2, “Underflow Exception Condition.”
- Inexact Exception Condition (XX). These exception conditions are described in Section 3.3.6.2.3, “Inexact Exception Condition.”

Each floating-point exception condition and each category of invalid IEEE floating-point operation exception condition has a corresponding exception bit in the FPSCR. 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 corresponding IEEE floating-point enable bits in the FPSCR. If no governing enable bit is set, the instruction delivers a default result. Otherwise, specific condition bits and FPSCR[FX] are set and instruction execution is completed by suppressing or delivering a result. Finally, after instruction execution completes, a nonzero FPSCR[FX] causes a program exception if either MSR[FE0] or MSR[FE1] is set (invoking the system error handler). The FPR values immediately after the occurrence of an enabled exception do not depend on MSR[FE0,FE1].

FPSCR[FX] is set by any floating-point instruction (except **mtfsfi** and **mtfsf**) that causes any FPSCR exception bit 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 an exception condition bits is set and the exception enable bit is one.

A single instruction can set multiple exception condition bits 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 (∞*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. 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 revises 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 is not detected by software, which uses 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 excepting instruction or the data that caused the exception. Results from the excepting 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 excepting instruction and correct any faulty results. In this mode, no results caused by the excepting instruction have been used by or affected subsequent instructions that are executed before the exception handler is invoked. |
| 1 | 1 | Precise mode—The system floating-point enabled exception error handler is invoked precisely at the instruction that caused the enabled exception. |

In precise mode, whenever the system floating-point enabled exception error handler is invoked, the architecture ensures that all instructions logically residing before the excepting instruction have completed and no instruction after the excepting instruction has been executed. In an imprecise mode, the instruction flow may not be interrupted at the point of the instruction that caused the exception. The instruction at which the system floating-point exception handler is invoked has not been executed unless it is the excepting instruction and the exception is not suppressed.

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.

Instead of using an FPSCR instruction, an execution synchronizing instruction or event can be used to force exceptions and set bits in the FPSCR; however, for the best performance across the widest range of implementations, an FPSCR instruction should be used to achieve these effects.

For the best performance across the widest range of implementations, the following guidelines should be considered:

- If IEEE default results are acceptable, FE0 and FE1 should be cleared (ignore exceptions mode). All FPSCR exception enable bits should be cleared.
- If IEEE default results are unacceptable, 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 substantially degrade performance in some implementations and should be used only for debugging or 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). If any of these conditions occur, if FPSCR[FEX] is set (implicitly) and MSR[FE0–FE1] ≠ 00, the processor takes a program exception (floating-point enabled exception type).

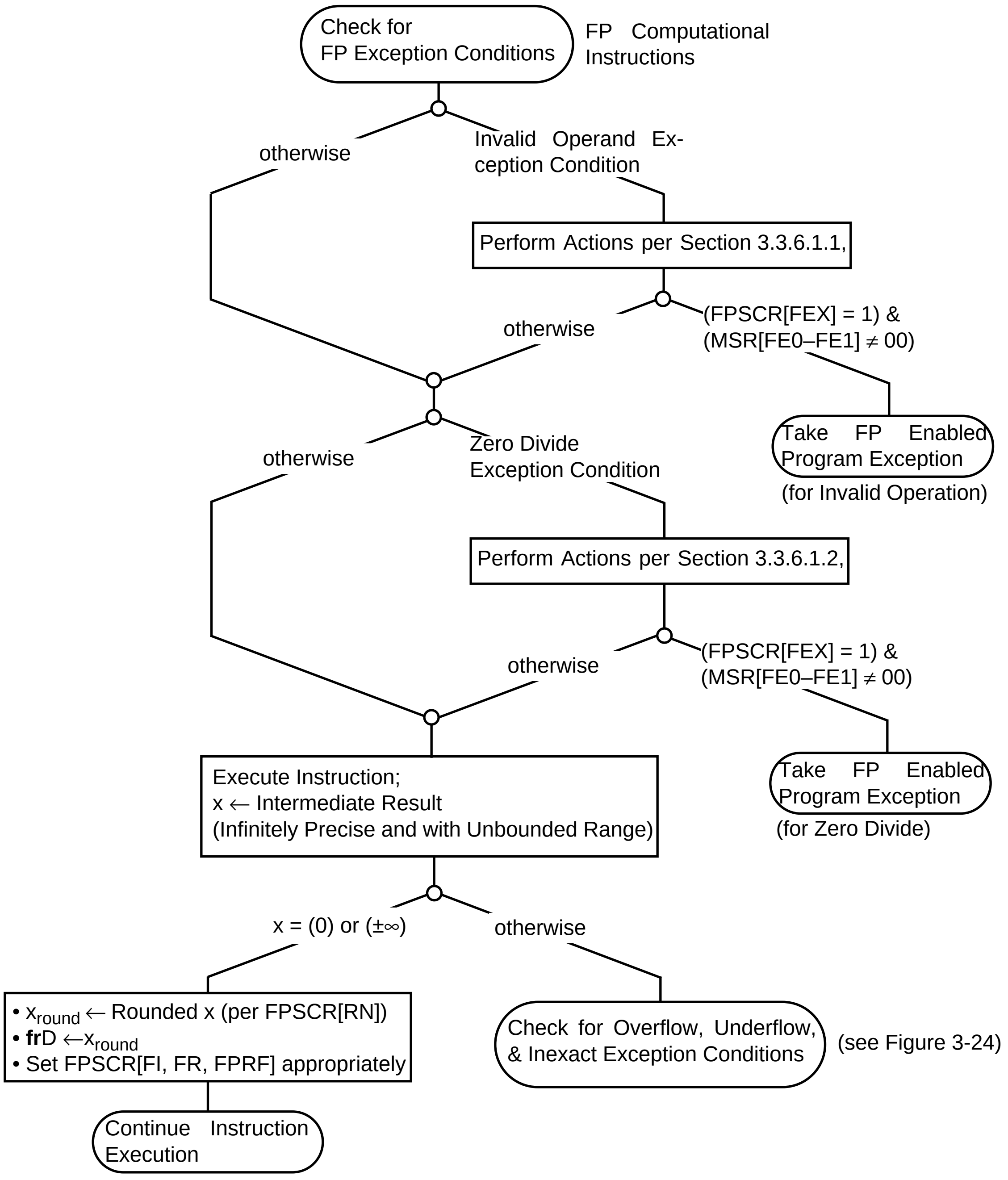

Check for FP Exception Conditions
FP Computational Instructions
otherwise
Invalid Operand Exception Condition
Perform Actions per Section 3.3.6.1.1,
otherwise
(FPSCR[FEX] = 1) & (MSR[FE0–FE1] ≠ 00)
Take FP Enabled Program Exception
(for Invalid Operation)
otherwise
Zero Divide Exception Condition
Perform Actions per Section 3.3.6.1.2,
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])
• frD ← x_round
• Set FPSCR[FI, FR, FPRF] appropriately
Check for Overflow, Underflow, & Inexact Exception Conditions
(see Figure 3-24)
Continue Instruction Execution

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

See Chapter 6, “Exceptions,” for information about exception handling. The actions performed for floating-point exception conditions are described in the following sections.

#### 3.3.6.1.1 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 (∞ – ∞)
- Division of infinity by infinity (∞ ÷ ∞)
- Division of zero by zero (0 ÷ 0)
- Multiplication of infinity by zero (∞ * 0)
- Ordered comparison involving a NaN (invalid compare)
- Square root or reciprocal square root of a negative, nonzero number (invalid square root). Note that 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 FPSCR[VXSQRT] 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 instructions not implemented in hardware to be emulated.

If an invalid operation occurs or software explicitly requests the exception using FPSCR[VXSOFT], (regardless of the value of FPSCR[VE]), the following occurs:

- One or two invalid operation exception condition bits is set

| | |
|---|---|
| FPSCR[VXSNAN] | (if SNaN) |
| FPSCR[VXISI] | (if ∞ – ∞) |
| FPSCR[VXIDI] | (if ∞ ÷ ∞) |
| FPSCR[VXZDZ] | (if 0 ÷ 0) |
| FPSCR[VXIMZ] | (if ∞ * 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.

Table 3-12 describes additional actions performed that depend on the value of FPSCR[VE].

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

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

### 3.3.6.1.2 Zero Divide Exception Condition

A zero divide exception condition occurs when a divide instruction is executed with a zero divisor and a finite, nonzero dividend or when an **fres** or **frsqrte** is executed with a zero operand. This condition indicates an exact infinite result from finite operands exception condition corresponding to a mathematical pole (divide or **fres**) or a branch point singularity (**frsqrte**). 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.

Table 3-13 describes additional actions depending on the value of FPSCR[ZE].

**Table 3-13. Additional Actions Performed for Zero Divide**

| Result Category | Action if FPSCR[ZE] = 1 | Action if FPSCR[ZE] = 0 |
|---|---|---|
| **fr**D | 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

Overflow, underflow, and inexact exception conditions are detected after the instruction executes and an infinitely precise result with unbounded range is computed. Figure 3-24 shows the flow for the detection of these conditions. It is a continuation of Figure 3-23.

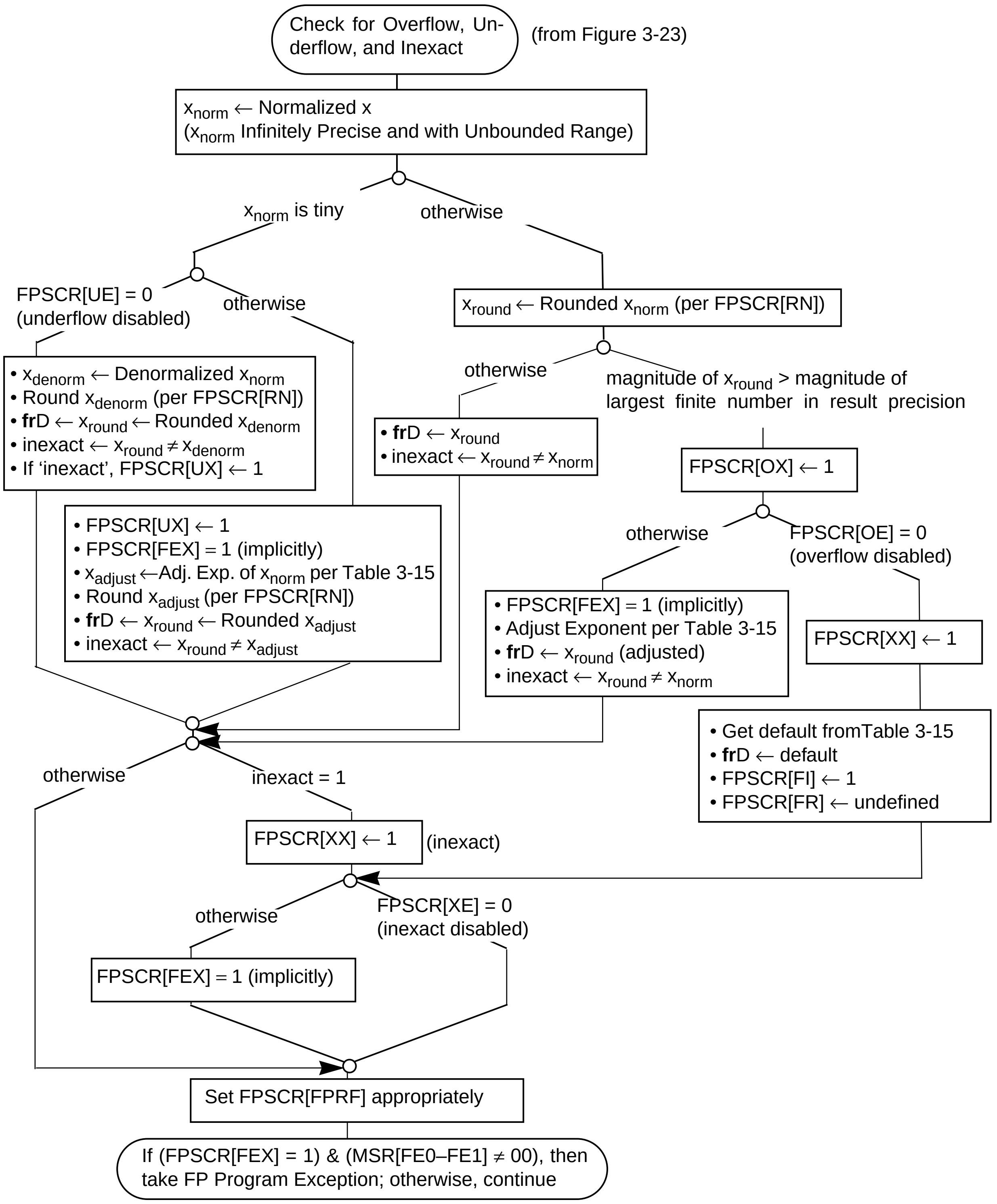

Check for Overflow, Underflow, and Inexact
(from Figure 3-23)
$x_{norm}$ ← Normalized x
($x_{norm}$ Infinitely Precise and with Unbounded Range)
$x_{norm}$ is tiny
otherwise
FPSCR[UE] = 0 (underflow disabled)
otherwise
• $x_{denorm}$ ← Denormalized $x_{norm}$
• Round $x_{denorm}$ (per FPSCR[RN])
• **fr**D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

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

As in the cases of invalid operation, or zero divide conditions, if FPSCR[FEX] is set implicitly 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.

#### 3.3.6.2.1 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. FPSCR[OX] is set, regardless of the FPSCR[OE] value.

Table 3-14 describes additional actions taken that depend on the setting of FPSCR[OE].

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

| Condition | Result Category | Action if FPSCR[OE] = 1 | Action if 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 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 |

**Table 3-15. Target Result for Overflow Exception Disabled Case (continued)**

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

#### 3.3.6.2.2 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, FPSCR[UE], is zero, 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 | — |

Table 3-16. Actions Performed for Underflow Conditions (continued)

| Condition | Result Category | Action Performed | |
|---|---|---|---|
| | | FPSCR[UE] = 1 | FPSCR[UE] = 0 |
| 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 that FPSCR[FR,FI] 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, FR and FI allow the system floating-point enabled exception error handler to unround the result, thus allowing the result to be denormalized.

#### 3.3.6.2.3 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 independently 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.

If the inexact exception condition enable bit, FPSCR[XE], is set and an inexact condition exists, then FPSCR[FEX] is implicitly set, causing the processor to take a floating-point enabled program exception. Running with inexact exception conditions enabled may have greater latency than enabling other types of floating-point exception conditions.

# Chapter 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 that 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, half-word, and word operands. Floating-point instructions operate on single-precision and double-precision floating-point operands. The PowerPC architecture uses instructions that are 4 bytes long and word-aligned. It provides for byte, half-word, and word operand fetches and stores between memory and a set of 32 general-purpose registers (GPRs). It also provides for word and double-word operand fetches and stores between memory and a set of 32 floating-point registers (FPRs). FPRs are 64 bits wide in all implementations. GPRs are 32 bits wide in 32-bit implementations and 64 bits wide in 64-bit 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. that 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 F, "Simplified Mnemonics," for a complete list of simplified mnemonics.

The instructions are organized by functional categories while maintaining the delineation of the three levels as described in Section 1.1.2, "The Levels of the PowerPC Architecture."

## 4.1 Conventions

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

### 4.1.1 Sequential Execution Model

Processors that implement the PowerPC architecture 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 that 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 5.2.5.2, “Instruction Cache Instructions.” For information regarding instruction fetching, and for information about guarded memory refer to Section 5.3.1.5, “The Guarded Attribute (G).”

## 4.1.2 Computation Modes

The architecture allows for the following types of implementations:

- 64-bit implementations, in which all general-purpose and floating-point registers, and 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. U

This chapter describes only the instructions defined for 32-bit implementations. Instructions defined only for 64-bit implementations are illegal in 32-bit implementations, and vice versa.

## 4.1.3 Classes of Instructions

Instructions belong to one of the following three classes:

- Defined
- Illegal
- Reserved

Note that while the definitions of these terms are consistent among these 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 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 UISA, VEA, and OEA. Defined instructions are guaranteed to be supported in all 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 processor may invoke the illegal instruction error handler (part of the program exception handler) when an unimplemented instruction is encountered so that it may be emulated in software, as required.

A defined instruction can have invalid forms, as described in Section 4.1.3.2.2, “Invalid Instruction Forms.”

##### 4.1.3.2.1 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:

- Load/store multiple instructions
- Load/store string instructions
- Or immediate instruction (preferred form of no-op)

##### 4.1.3.2.2 Invalid Instruction Forms

A defined instruction may have an instruction form that is invalid if one or more operands, excluding opcodes, 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
- Integer compare instructions (in 32-bit implementations only)
- Load/store floating-point with update instructions

#### 4.1.3.2.3 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** V
- Lookaside buffer management instructions—**tlbia**, **tlbie**, and **tlbsync** (with conditions, see Chapter 8, “Instruction Set,” for more information)

Note that the **stfiwx** instruction is defined as optional by the architecture to ensure U backwards compatibility with earlier processors; however, it will likely be required for subsequent processors.

Also, note that additional categories may be defined in future implementations. If an implementation claims to support a given category, it implements all the instructions in that category.

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 architecture. These opcodes are available for future extensions of the architecture; that is, future versions of the 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

- Instructions that are implemented in the architecture but are not implemented in a specific implementation. For example, instructions specific to 64-bit processors are illegal for 32-bit processors.

  The following primary opcodes are defined for 64-bit implementations only and are illegal on 32-bit implementations:

  2, 30, 58, 62
- All unused extended opcodes are illegal. The unused extended opcodes can be determined from information in Section A.4, "Instructions Sorted by Opcode (Binary)," and Section 4.1.3.4, "Reserved Instructions." Notice that extended opcodes for instructions that are defined only for 64-bit implementations are illegal in 32-bit implementations. The following primary opcodes have unused extended opcodes.

  19, 31, 59, 63 (primary opcodes 30 and 62 are illegal for 32-bit implementations, but as 64-bit opcodes they have some unused extended opcodes)
- 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 that 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 6.4.7, "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 architecture.

#### 4.1.3.4 Reserved Instructions

Reserved instructions are allocated to specific implementation-dependent purposes not defined by the architecture. An attempt to execute an unimplemented reserved instruction invokes the illegal instruction program exception. See Section 6.4.7, "Program Exception (0x00700)."

The following types of instructions are reserved:

- POWER architecture instructions not included in the architecture.
- Implementation-specific instructions not defined in the UISA, VEA, or OEA, including those used to conform to the architecture specifications (for example, Load Data TLB Entry (**tlbld**) and Load Instruction TLB Entry (**tlbli**) instructions implemented in several processors).
- The instruction with primary opcode 0 when the instruction does not consist entirely of binary zeros

## 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.

Memory operands may be bytes, half words, words, or double words, 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 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-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 32-bit unsigned binary arithmetic. A carry from bit 0 is ignored.

In all 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 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 LR with the next sequential instruction address (current instruction address + 4).

## 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 (**rfi**), and Instruction Synchronize (**isync**) instructions perform context synchronization by allowing previously issued instructions to complete before performing a context switch. Execution of one of these instructions ensures the following:

1. No higher priority exception exists (**sc**) and instruction 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 following the **sc**, **rfi**, or **isync** instruction execute in the context established by these instructions.

### 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** instruction is 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 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

The exception mechanism handles 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 architecture provides the following supervisor-level instructions: **dcbi**, **mfmsr**, **mfspr**, **mfsr**, **mfsrin**, **mtmsr**, **mtspr**, **mtsr**, **mtsrin**, **rfi**, **tlbia**, **tlbie**, and **tlbsync** (defined by OEA). Note that 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.
- The execution of a floating-point instruction that requires system software assistance causes the floating-point assist exception handler to be invoked. The conditions under which such software assistance is required are implementation-dependent.

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

## 4.2 UISA Instructions

The 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. CR0 is set to reflect a signed comparison 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. 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 result.

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

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

#### 4.2.1.1 Integer Arithmetic Instructions

Table 4-1 lists the integer arithmetic instructions.

**Table 4-1. Integer Arithmetic Instructions**

| Name | Mnemonic | 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**<br>**add.**<br>**addo**<br>**addo.** | **r**D,**r**A,**r**B | The sum (**r**A) + (**r**B) is placed into **r**D.<br>**add** Add<br>**add.** Add with CR Update. The dot suffix enables the update of the CR.<br>**addo** Add with Overflow Enabled. The o suffix enables the overflow bit (OV) in the XER.<br>**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**<br>**subf.**<br>**subfo**<br>**subfo.** | **r**D,**r**A,**r**B | The sum ¬ (**r**A) + (**r**B) +1 is placed into **r**D.<br>**subf** Subtract From<br>**subf.** Subtract from with CR Update. The dot suffix enables the update of the CR.<br>**subfo** Subtract from with Overflow Enabled. The o suffix enables the overflow bit (OV) in the XER.<br>**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** | **r**D,**r**A,SIMM | The sum (**r**A) + SIMM is placed into **r**D. |
| Add Immediate Carrying and Record | **addic.** | **r**D,**r**A,SIMM | The sum (**r**A) + SIMM is placed into **r**D. The CR is updated. |
| Subtract from Immediate Carrying | **subfic** | **r**D,**r**A,SIMM | The sum ¬ (**r**A) + SIMM + 1 is placed into **r**D. |
| Add Carrying | **addc**<br>**addc.**<br>**addco**<br>**addco.** | **r**D,**r**A,**r**B | The sum (**r**A) + (**r**B) is placed into **r**D.<br>**addc** Add Carrying<br>**addc.** Add Carrying with CR Update. The dot suffix enables the update of the CR.<br>**addco** Add Carrying with Overflow Enabled. The o suffix enables the overflow bit (OV) in the XER.<br>**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. |

**Table 4-1. Integer Arithmetic Instructions (continued)**

| Name | Mnemonic | Syntax | Operation |
|---|---|---|---|
| Subtract from Carrying | **subfc**<br>**subfc.**<br>**subfco**<br>**subfco.** | **r**D,**r**A,**r**B | The sum ¬ (**r**A) + (**r**B) + 1 is placed into **r**D.<br>**subfc** Subtract from Carrying<br>**subfc.** Subtract from Carrying with CR Update. The dot suffix enables the update of the CR.<br>**subfco** Subtract from Carrying with Overflow. The o suffix enables the overflow bit (OV) in the XER.<br>**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. |
| Add Extended | **adde**<br>**adde.**<br>**addeo**<br>**addeo.** | **r**D,**r**A,**r**B | The sum (**r**A) + (**r**B) + XER[CA] is placed into **r**D.<br>**adde** Add Extended<br>**adde.** Add Extended with CR Update. The dot suffix enables the update of the CR.<br>**addeo** Add Extended with Overflow. The o suffix enables the overflow bit (OV) in the XER.<br>**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**<br>**subfe.**<br>**subfeo**<br>**subfeo.** | **r**D,**r**A,**r**B | The sum ¬ (**r**A) + (**r**B) + XER[CA] is placed into **r**D.<br>**subfe** Subtract from Extended<br>**subfe.** Subtract from Extended with CR Update. The dot suffix enables the update of the CR.<br>**subfeo** Subtract from Extended with Overflow. The o suffix enables the overflow bit (OV) in the XER.<br>**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**<br>**addme.**<br>**addmeo**<br>**addmeo.** | **r**D,**r**A | The sum (**r**A) + XER[CA] added to 0xFFFF_FFFF is placed into **r**D.<br>**addme** Add to Minus One Extended<br>**addme.** Add to Minus One Extended with CR Update. The dot suffix enables the update of the CR.<br>**addmeo** Add to Minus One Extended with Overflow. The o suffix enables the overflow bit (OV) in the XER.<br>**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**<br>**subfme.**<br>**subfmeo**<br>**subfmeo.** | **r**D,**r**A | The sum ¬ (**r**A) + XER[CA] added to 0xFFFF_FFFF is placed into **r**D.<br>**subfme** Subtract from Minus One Extended<br>**subfme.** Subtract from Minus One Extended with CR Update. The dot suffix enables the update of the CR.<br>**subfmeo** Subtract from Minus One Extended with Overflow. The o suffix enables the overflow bit (OV) in the XER.<br>**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. |

**Table 4-1. Integer Arithmetic Instructions (continued)**

| Name | Mnemonic | Syntax | Operation |
|---|---|---|---|
| Add to Zero Extended | **addze**<br>**addze.**<br>**addzeo**<br>**addzeo.** | **r**D,**r**A | The sum (**r**A) + XER[CA] is placed into **r**D.<br>**addze** Add to Zero Extended<br>**addze.** Add to Zero Extended with CR Update. The dot suffix enables the update of the CR.<br>**addzeo** Add to Zero Extended with Overflow. The o suffix enables the overflow bit (OV) in the XER.<br>**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. |
| Subtract from Zero Extended | **subfze**<br>**subfze.**<br>**subfzeo**<br>**subfzeo.** | **r**D,**r**A | The sum ¬ (**r**A) + XER[CA] is placed into **r**D.<br>**subfze** Subtract from Zero Extended<br>**subfze.** Subtract from Zero Extended with CR Update. The dot suffix enables the update of the CR.<br>**subfzeo** Subtract from Zero Extended with Overflow. The o suffix enables the overflow bit (OV) in the XER.<br>**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**<br>**neg.**<br>**nego**<br>**nego.** | **r**D,**r**A | The sum ¬ (**r**A) + 1 is placed into **r**D.<br>**neg** Negate<br>**neg.** Negate with CR Update. The dot suffix enables the update of the CR.<br>**nego** Negate with Overflow. The o suffix enables the overflow bit (OV) in the XER.<br>**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 32 bits of the product (**r**A) ∗ SIMM are placed into **r**D.<br>This instruction can be used with **mulhd***x* or **mulhw***x* to calculate a full 64-bit product. |
| Multiply Low | **mullw**<br>**mullw.**<br>**mullwo**<br>**mullwo.** | **r**D,**r**A,**r**B | The 32-bit product (**r**A) ∗ (**r**B) is placed into register **r**D.<br>This instruction can be used with **mulhw***x* to calculate a full 64-bit product.<br>**mullw** Multiply Low<br>**mullw.** Multiply Low with CR Update. The dot suffix enables the update of the CR.<br>**mullwo** Multiply Low with Overflow. The o suffix enables the overflow bit (OV) in the XER.<br>**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 High Word | **mulhw**<br>**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 **r**D.<br>**mulhw** Multiply High Word<br>**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 | Syntax | Operation |
|---|---|---|---|
| Multiply High Word Unsigned | **mulhwu**<br>**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 **r**D.<br>**mulhwu** Multiply High Word Unsigned<br>**mulhwu.** Multiply High Word Unsigned with CR Update. The dot suffix enables the update of the CR. |
| Divide Word | **divw**<br>**divw.**<br>**divwo**<br>**divwo.** | **r**D,**r**A,**r**B | The dividend is the signed value of **r**A. The divisor is the signed value of **r**B. The quotient is placed into **r**D. The remainder is not supplied as a result.<br>**divw** Divide Word<br>**divw.** Divide Word with CR Update. The dot suffix enables the update of the CR.<br>**divwo** Divide Word with Overflow. The o suffix enables the overflow bit (OV) in the XER.<br>**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. |
| Divide Word Unsigned | **divwu**<br>**divwu.**<br>**divwuo**<br>**divwuo.** | **r**D,**r**A,**r**B | The dividend is the zero-extended value in **r**A. The divisor is the zero-extended value in **r**B. The quotient is placed into **r**D. The remainder is not supplied as a result.<br>**divwu** Divide Word Unsigned<br>**divwu.** Divide Word Unsigned with CR Update. The dot suffix enables the update of the CR.<br>**divwuo** Divide Word Unsigned with Overflow. The o suffix enables the overflow bit (OV) in the XER.<br>**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. |

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 F, “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.

For 32-bit implementations, the L field must be cleared, otherwise the instruction form is invalid.

The integer compare instructions (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 | Syntax | Operation |
|---|---|---|---|
| Compare Immediate | **cmpi** | **crf**D,L,**r**A,SIMM | The value in register **r**A 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 is compared with the value in register **r**B, 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 is compared with 0x0000 \|\| 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 is compared with the value in register **r**B, 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 F, "Simplified Mnemonics."

### 4.2.1.3 Integer Logical Instructions

The logical instructions shown in Table 4-3 perform bit-parallel operations on 32-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. Logical instructions without CR update and the remaining logical instructions do not modify the CR. Logical instructions do not affect XER[SO,OV,CA].

See Appendix F, "Simplified Mnemonics," for simplified mnemonic examples for integer logical operations.

**Table 4-3. Integer Logical Instructions**

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

Table 4-3. Integer Logical Instructions (continued)

| Name | Mnemonic | Syntax | Operation |
|---|---|---|---|
| OR Immediate Shifted | **oris** | **r**A,**r**S,UIMM | The contents of **r**S are ORed with 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 \|\| 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 UIMM \|\| 0x0000 and the result is placed into **r**A. |
| AND | **and**<br>**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.<br>**and** AND<br>**and.** AND with CR Update**.** The dot suffix enables the update of the CR. |
| OR | **or**<br>**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.<br>**or** OR<br>**or.** OR with CR Update**.** The dot suffix enables the update of the CR. |
| XOR | **xor**<br>**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.<br>**xor** XOR<br>**xor.** XOR with CR Update**.** The dot suffix enables the update of the CR. |
| NAND | **nand**<br>**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.<br>**nand** NAND<br>**nand.** NAND with CR Update**.** The dot suffix enables the update of CR.<br>Note that **nand***x*, with **r**S = **r**B, can be used to obtain the one's complement. |
| NOR | **nor**<br>**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.<br>**nor** NOR<br>**nor.** NOR with CR Update**.** The dot suffix enables the update of the CR.<br>Note that **nor***x*, with **r**S = **r**B, can be used to obtain the one's complement. |
| Equivalent | **eqv**<br>**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.<br>**eqv** Equivalent<br>**eqv.** Equivalent with CR Update**.** The dot suffix enables the update of the CR. |
| AND with Complement | **andc**<br>**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.<br>**andc** AND with Complement<br>**andc.** AND with Complement with CR Update**.** The dot suffix enables the update of the CR. |
| OR with Complement | **orc**<br>**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.<br>**orc** OR with Complement<br>**orc.** OR with Complement with CR Update**.** The dot suffix enables the update of the CR. |

**Table 4-3. Integer Logical Instructions (continued)**

| Name | Mnemonic | Syntax | Operation |
|---|---|---|---|
| Extend Sign Byte | **extsb**<br>**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 24 of **r**S is placed into the remaining high-order bits of **r**A.<br>**extsb** Extend Sign Byte<br>**extsb.** Extend Sign Byte with CR Update**.** The dot suffix enables the update of the CR. |
| Extend Sign Half Word | **extsh**<br>**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 16 of **r**S is placed into the remaining high-order bits of **r**A.<br>**extsh** Extend Sign Half Word<br>**extsh.** Extend Sign Half Word with CR Update**.** The dot suffix enables the update of the CR. |
| Count Leading Zeros Word | **cntlzw**<br>**cntlzw.** | **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 32, inclusive.<br>If Rc = 1 (dot suffix), LT is cleared in CR0.<br>**cntlzw** Count Leading Zeros Word<br>**cntlzw.** Count Leading Zeros Word 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 32-bit quantity left by a specified number of bit positions. Bits that exit from position 0 enter at position 31.

The rotate and shift instructions employ a mask generator. The mask is 32 bits long and consists of ‘1’ bits from a start bit, Mstart, through and including a stop bit, Mstop, and ‘0’ bits elsewhere. The values of Mstart and Mstop range from 0 to 31. If Mstart > Mstop, the ‘1’ bits wrap around from position 31 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–31] = 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 F, “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.

### 4.2.1.4.1 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 | Syntax | Operation |
|---|---|---|---|
| Rotate Left Word Immediate then AND with Mask | **rlwinm**<br>**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 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.<br>**rlwinm** Rotate Left Word Immediate then AND with Mask<br>**rlwinm.** Rotate Left Word Immediate then AND with Mask with CR Update. The dot suffix enables the update of the CR. |
| Rotate Left Word then AND with Mask | **rlwnm**<br>**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 through the bit specified by operand ME and 0 bits elsewhere. The rotated word is ANDed with the generated mask and the result is placed into **r**A.<br>**rlwnm** Rotate Left Word then AND with Mask<br>**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**<br>**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 through the bit specified by operand ME and 0 bits elsewhere. The rotated word is inserted into **r**A under control of the generated mask.<br>**rlwimi** Rotate Left Word Immediate then Mask<br>**rlwimi.** Rotate Left Word Immediate then Mask Insert with CR Update. The dot suffix enables the update of the CR. |

### 4.2.1.4.2 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 F, “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 C, “Multiple-Precision Shifts.”

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

**Table 4-5. Integer Shift Instructions**

| Name | Mnemonic | Syntax | Operation |
|---|---|---|---|
| Shift Left Word | **slw**<br>**slw.** | **r**A,**r**S,**r**B | The contents 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 0 are lost. Zeros are supplied to the vacated positions on the right. The 32-bit result is placed into **r**A.<br>**slw** Shift Left Word<br>**slw.** Shift Left Word with CR Update. The dot suffix enables the update of the CR. |
| Shift Right Word | **srw**<br>**srw.** | **r**A,**r**S,**r**B | The contents of **r**S are shifted right the number of bits specified by the low-order 6 **r**B bits. Bits shifted out of position 31 are lost. Zeros are supplied to the vacated positions on the left. The 32-bit result is placed into **r**A.<br>**srw** Shift Right Word<br>**srw.** Shift Right Word with CR Update. The dot suffix enables the update of the CR. |
| Shift Right Algebraic Word Immediate | **srawi**<br>**srawi.** | **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 31are lost. The result is sign extended and placed into **r**A.<br>**srawi** Shift Right Algebraic Word Immediate<br>**srawi.** Shift Right Algebraic Word Immediate with CR Update. The dot suffix enables the update of the CR. |
| Shift Right Algebraic Word | **sraw**<br>**sraw.** | **r**A,**r**S,**r**B | The contents 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 31 are lost. The result is placed into **r**A.<br>**sraw** Shift Right Algebraic Word<br>**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 that 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 6.4.8, “Floating-Point Unavailable Exception (0x00800)”). See Section 4.2.3, “Load and Store Instructions,” for information about floating-point loads and stores.

The architecture supports the IEEE-754 floating-point standard, but requires software support to conform with that standard. Floating-point operations conform to the standard, except for operations performed with the **fmadd**, **fres**, **fsel**, and **frsqrte** instructions, or if software sets the non-IEEE mode bit, FPSCR[NI]. Section 3.3, “Floating-Point Execution Models—UISA,” gives detailed information about the floating-point formats and exception conditions. Also, see Appendix D, “Floating-Point Models.”

### 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 | Syntax | Operation |
|---|---|---|---|
| Floating Add (Double-Precision) | **fadd**<br>**fadd.** | **fr**D,**fr**A,**fr**B | 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 target precision under control of the floating-point rounding control field RN of the FPSCR and placed into **fr**D.<br>**fadd** Floating Add (Double-Precision)<br>**fadd.** Floating Add (Double-Precision) with CR Update. The dot suffix enables the update of the CR. |
| Floating Add Single | **fadds**<br>**fadds.** | **fr**D,**fr**A,**fr**B | 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 target precision under control of the floating-point rounding control field RN of the FPSCR and placed into **fr**D.<br>**fadds** Floating Add Single<br>**fadds.** Floating Add Single with CR Update. The dot suffix enables the update of the CR. |
| Floating Subtract (Double-Precision) | **fsub**<br>**fsub.** | **fr**D,**fr**A,**fr**B | The floating-point operand in **fr**B is subtracted from the floating-point operand in **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.<br>**fsub** Floating Subtract (Double-Precision)<br>**fsub.** Floating Subtract (Double-Precision) with CR Update. The dot suffix enables the update of the CR. |
| Floating Subtract Single | **fsubs**<br>**fsubs.** | **fr**D,**fr**A,**fr**B | The floating-point operand in **fr**B is subtracted from the floating-point operand in **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.<br>**fsubs** Floating Subtract Single<br>**fsubs.** Floating Subtract Single with CR Update. The dot suffix enables the update of the CR. |

**Table 4-6. Floating-Point Arithmetic Instructions (continued)**

| Name | Mnemonic | Syntax | Operation |
|---|---|---|---|
| Floating Multiply (Double-Precision) | **fmul**<br>**fmul.** | **fr**D,**fr**A,**fr**C | The floating-point operand in **fr**A is multiplied by the floating-point operand in **fr**C.<br>**fmul** Floating Multiply (Double-Precision)<br>**fmul.** Floating Multiply (Double-Precision) with CR Update. The dot suffix enables the update of the CR. |
| Floating Multiply Single | **fmuls**<br>**fmuls.** | **fr**D,**fr**A,**fr**C | The floating-point operand in **fr**A is multiplied by the floating-point operand in **fr**C.<br>**fmuls** Floating Multiply Single<br>**fmuls.** Floating Multiply Single with CR Update. The dot suffix enables the update of the CR. |
| Floating Divide (Double-Precision) | **fdiv**<br>**fdiv.** | **fr**D,**fr**A,**fr**B | The floating-point operand in **fr**A is divided by the floating-point operand in **fr**B. No remainder is preserved.<br>**fdiv** Floating Divide (Double-Precision)<br>**fdiv.** Floating Divide (Double-Precision) with CR Update. The dot suffix enables the update of the CR. |
| Floating Divide Single | **fdivs**<br>**fdivs.** | **fr**D,**fr**A,**fr**B | The floating-point operand in **fr**A is divided by the floating-point operand in **fr**B. No remainder is preserved.<br>**fdivs** Floating Divide Single<br>**fdivs.** Floating Divide Single with CR Update. The dot suffix enables the update of the CR. |
| Floating Square Root (Double-Precision) | **fsqrt**<br>**fsqrt.** | **fr**D,**fr**B | The square root of the floating-point operand in **fr**B is placed into **fr**D.<br>**fsqrt** Floating Square Root (Double-Precision)<br>**fsqrt.** Floating Square Root (Double-Precision) with CR Update. The dot suffix enables the update of the CR.<br>This instruction is optional. |
| Floating Square Root Single | **fsqrts**<br>**fsqrts.** | **fr**D,**fr**B | The square root of the floating-point operand in **fr**B is placed into **fr**D.<br>**fsqrts** Floating Square Root Single<br>**fsqrts.** Floating Square Root Single with CR Update. The dot suffix enables the update of the CR.<br>This instruction is optional. |
| Floating Reciprocal Estimate Single | **fres**<br>**fres.** | **fr**D,**fr**B | A single-precision estimate of the reciprocal of the floating-point operand in **fr**B is placed into **fr**D. The estimate placed into **fr**D is correct to a precision of one part in 256 of the reciprocal of **fr**B.<br>**fres** Floating Reciprocal Estimate Single<br>**fres.** Floating Reciprocal Estimate Single with CR Update. The dot suffix enables the update of the CR.<br>This instruction is optional. |

**Table 4-6. Floating-Point Arithmetic Instructions (continued)**

| Name | Mnemonic | Syntax | Operation |
|---|---|---|---|
| Floating Reciprocal Square Root Estimate | **frsqrte**<br>**frsqrte.** | **fr**D,**fr**B | A double-precision estimate of the reciprocal of the square root of the floating-point operand in **fr**B is placed into **fr**D. The estimate placed into **fr**D is correct to a precision of one part in 32 of the reciprocal of the square root of **fr**B.<br>**frsqrte** Floating Reciprocal Square Root Estimate<br>**frsqrte.** Floating Reciprocal Square Root estimate with CR Update. The dot suffix enables the update of the CR.<br>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).<br>**fsel** Floating Select<br>**fsel.** Floating Select with CR Update. The dot suffix enables the update of the CR.<br>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, FPSCR[FR,FI,FPRF] 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 | Syntax | Operation |
|---|---|---|---|
| Floating Multiply-Add (Double-Precision) | **fmadd**<br>**fmadd.** | **fr**D,**fr**A,**fr**C,**fr**B | The floating-point operand in **fr**A is multiplied by the floating-point operand in **fr**C. The floating-point operand in **fr**B is added to this intermediate result.<br>**fmadd** Floating Multiply-Add (Double-Precision)<br>**fmadd.** Floating Multiply-Add (Double-Precision) with CR Update. The dot suffix enables the update of the CR. |
| Floating Multiply-Add Single | **fmadds**<br>**fmadds.** | **fr**D,**fr**A,**fr**C,**fr**B | The floating-point operand in **fr**A is multiplied by the floating-point operand in **fr**C. The floating-point operand in **fr**B is added to this intermediate result.<br>**fmadds** Floating Multiply-Add Single<br>**fmadds.** Floating Multiply-Add Single with CR Update. The dot suffix enables the update of the CR. |

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

| Name | Mnemonic | Syntax | Operation |
|---|---|---|---|
| Floating Multiply-Subtract (Double-Precision) | **fmsub**<br>**fmsub.** | **fr**D,**fr**A,**fr**C,**fr**B | The floating-point operand in **fr**A is multiplied by the floating-point operand in **fr**C. The floating-point operand in **fr**B is subtracted from this intermediate result.<br>**fmsub** Floating Multiply-Subtract (Double-Precision)<br>**fmsub.** Floating Multiply-Subtract (Double-Precision) with CR Update. The dot suffix enables the update of the CR. |
| Floating Multiply-Subtract Single | **fmsubs**<br>**fmsubs.** | **fr**D,**fr**A,**fr**C,**fr**B | The floating-point operand in **fr**A is multiplied by the floating-point operand in **fr**C. The floating-point operand in **fr**B is subtracted from this intermediate result.<br>**fmsubs** Floating Multiply-Subtract Single<br>**fmsubs.** Floating Multiply-Subtract Single with CR Update. The dot suffix enables the update of the CR. |
| Floating Negative Multiply-Add (Double-Precision) | **fnmadd**<br>**fnmadd.** | **fr**D,**fr**A,**fr**C,**fr**B | The floating-point operand in **fr**A is multiplied by the floating-point operand in **fr**C. The floating-point operand in **fr**B is added to this intermediate result.<br>**fnmadd** Floating Negative Multiply-Add (Double-Precision)<br>**fnmadd.** Floating Negative Multiply-Add (Double-Precision) with CR Update. The dot suffix enables update of the CR. |
| Floating Negative Multiply-Add Single | **fnmadds**<br>**fnmadds.** | **fr**D,**fr**A,**fr**C,**fr**B | The floating-point operand in **fr**A is multiplied by the floating-point operand in **fr**C. The floating-point operand in **fr**B is added to this intermediate result.<br>**fnmadds** Floating Negative Multiply-Add Single<br>**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**<br>**fnmsub.** | **fr**D,**fr**A,**fr**C,**fr**B | The floating-point operand in **fr**A is multiplied by the floating-point operand in **fr**C. The floating-point operand in **fr**B is subtracted from this intermediate result.<br>**fnmsub** Floating Negative Multiply-Subtract (Double-Precision)<br>**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**<br>**fnmsubs.** | **fr**D,**fr**A,**fr**C,**fr**B | The floating-point operand in **fr**A is multiplied by the floating-point operand in **fr**C. The floating-point operand in **fr**B is subtracted from this intermediate result.<br>**fnmsubs** Floating Negative Multiply-Subtract Single<br>**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 Section D.2, “Multiply-Add Type Instruction Execution Model.”

### 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 architecture defines **fr**D[0–31] as undefined when executing the Floating Convert to Integer Word (**fctiw**) and Floating Convert to Integer Word with Round toward Zero (**fctiwz**) instructions. Floating-point rounding instructions are described in Table 4-8.

Examples of uses of these instructions to perform various conversions can be found in Appendix D, “Floating-Point Models.”

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

| Name | Mnemonic | Syntax | Operation |
|---|---|---|---|
| Floating Round to Single-Precision | **frsp**<br>**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.<br>**frsp** Floating Round to Single-Precision<br>**frsp.** Floating Round to Single-Precision with CR Update. The dot suffix enables the update of the CR. |
| Floating Convert to Integer Word | **fctiw**<br>**fctiw.** | **fr**D,**fr**B | The floating-point operand in **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.<br>**fctiw** Floating Convert to Integer Word<br>**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**<br>**fctiwz.** | **fr**D,**fr**B | The floating-point operand in **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.<br>**fctiwz** Floating Convert to Integer Word with Round toward Zero<br>**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 FPRs 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 FPCC (floating-point condition code) FPSCD[16–19] 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) |

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

**Table 4-10. Floating-Point Compare Instructions**

| Name | Mnemonic | 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 | Syntax | Operation |
|---|---|---|---|
| Move from FPSCR | **mffs**<br>**mffs.** | **fr**D | The contents of the FPSCR are placed into bits 32–63 of **fr**D. **fr**D[0–31] are undefined.<br>**mffs** Move from FPSCR<br>**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) are cleared in the FPSCR. |
| Move to FPSCR Field Immediate | **mtfsfi**<br>**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.<br>**mtfsfi** Move to FPSCR Field Immediate<br>**mtfsfi.** Move to FPSCR Field Immediate with CR Update. The dot suffix enables the update of the CR. |

**Table 4-11. Floating-Point Status and Control Register Instructions (continued)**

| Name | Mnemonic | Syntax | Operation |
|---|---|---|---|
| Move to FPSCR Fields | **mtfsf**<br>**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.<br>The contents of FPSCR[FX] are altered only if FM[0] = 1.<br>**mtfsf** Move to FPSCR Fields<br>**mtfsf.** Move to FPSCR Fields with CR Update. The dot suffix enables the update of the CR. |
| Move to FPSCR Bit 0 | **mtfsb0**<br>**mtfsb0.** | **crb**D | The FPSCR bit location specified by operand **crb**D is cleared.<br>Bits 1 and 2 (FEX and VX) cannot be reset explicitly.<br>**mtfsb0** Move to FPSCR Bit 0<br>**mtfsb0.** Move to FPSCR Bit 0 with CR Update. The dot suffix enables the update of the CR. |
| Move to FPSCR Bit 1 | **mtfsb1**<br>**mtfsb1.** | **crb**D | The FPSCR bit location specified by operand **crb**D is set.<br>Bits 1 and 2 (FEX and VX) cannot be set explicitly.<br>**mtfsb1** Move to FPSCR Bit 1<br>**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; **fneg**, **fabs**, and **fnabs** may alter the sign bit of a NaN. 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 | Syntax | Operation |
|---|---|---|---|
| Floating Move Register | **fmr**<br>**fmr.** | **fr**D,**fr**B | The contents of **fr**B are placed into **fr**D.<br>**fmr** Floating Move Register<br>**fmr.** Floating Move Register with CR Update. The dot suffix enables the update of the CR. |
| Floating Negate | **fneg**<br>**fneg.** | **fr**D,**fr**B | The contents of **fr**B with bit 0 inverted are placed into **fr**D.<br>**fneg** Floating Negate<br>**fneg.** Floating Negate with CR Update. The dot suffix enables the update of the CR. |

**Table 4-12. Floating-Point Move Instructions (continued)**

| | | | |
|---|---|---|---|
| Floating Absolute Value | **fabs**<br>**fabs.** | **fr**D,**fr**B | The contents of **fr**B with bit 0 cleared are placed into **fr**D.<br>**fabs** Floating Absolute Value<br>**fabs.** Floating Absolute Value with CR Update. The dot suffix enables the update of the CR. |
| Floating Negative Absolute Value | **fnabs**<br>**fnabs.** | **fr**D,**fr**B | The contents of **fr**B with bit 0 set are placed into **fr**D.<br>**fnabs** Floating Negative Absolute Value<br>**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 indirect with index mode, or register indirect mode. See Section 4.1.4.2, “Effective Address Calculation,” for information about calculating effective addresses. Note that in some implementations, operations that are not naturally aligned may suffer performance degradation. Section 6.4.6.1, “Integer Alignment Exceptions,” gives additional information about load and store address alignment exceptions.

#### 4.2.3.1.1 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.

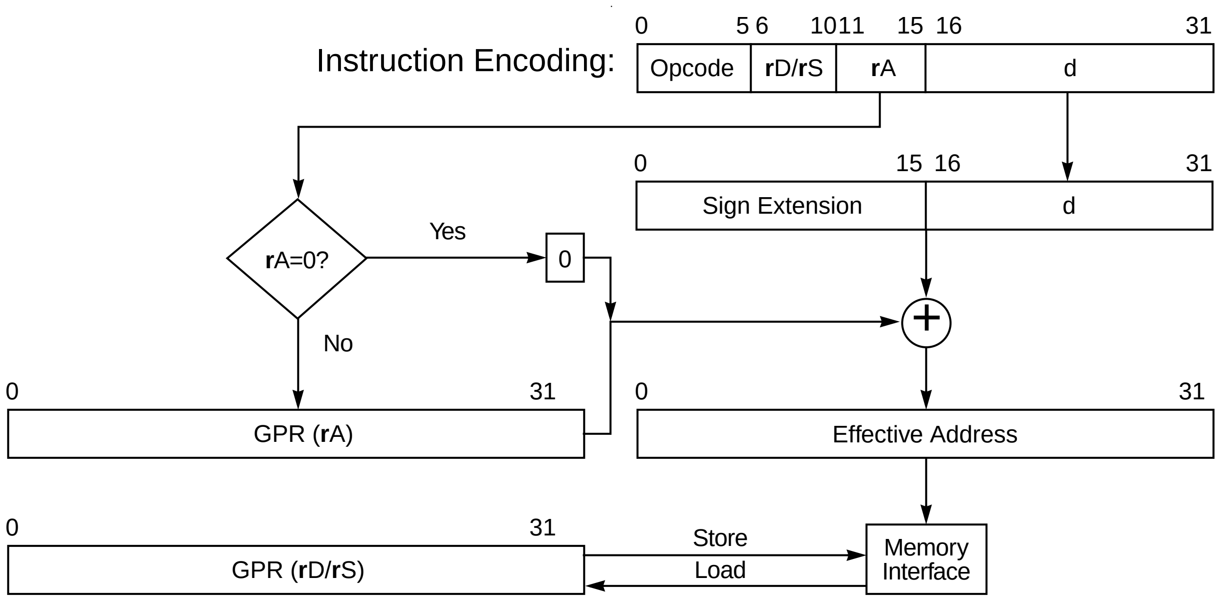


**Figure 4-1. Register Indirect with Immediate Index Addressing for Integer Loads/Stores**

#### 4.2.3.1.2 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.

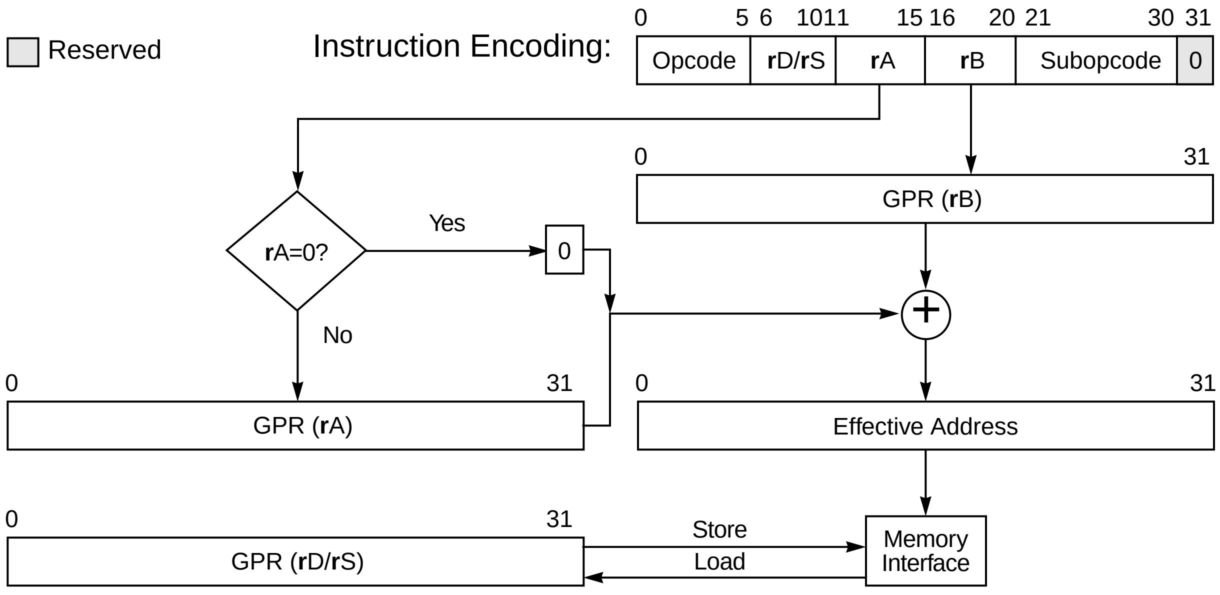


**Figure 4-2. Register Indirect with Index Addressing for Integer Loads/Stores**

### 4.2.3.1.3 Register Indirect Addressing for Integer Loads and Stores

Instructions using this addressing mode use the contents of the GPR specified by the **r**A operand as the 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.

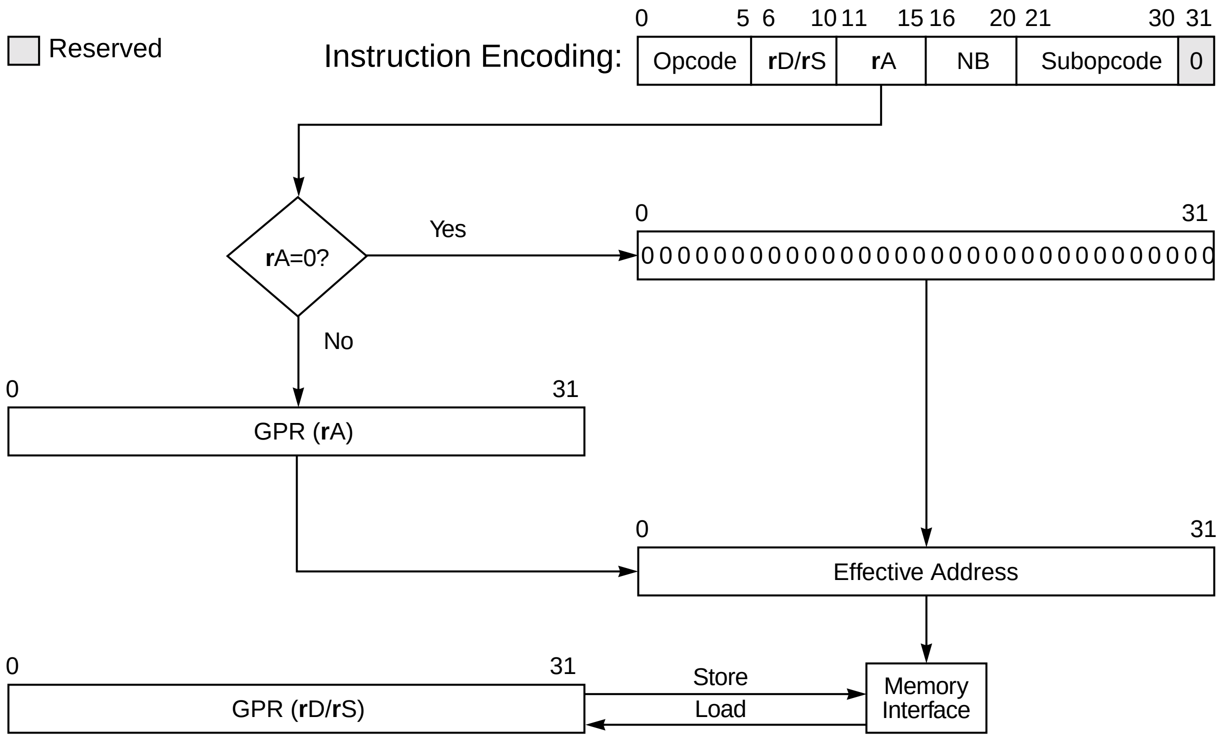


**Figure 4-3. Register Indirect Addressing for Integer Loads/Stores**

### 4.2.3.2 Integer Load Instructions

For integer load instructions, the byte, half word, word, or double word addressed by the EA (effective address) 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, half word, word, or double word) addressed by the EA is loaded into **r**D. Note that the 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 architecture; see Section 3.1.2, “Byte Ordering,” for information about little-endian byte ordering.

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

Table 4-13 summarizes the integer load instructions.

**Table 4-13. Integer Load Instructions**

| Name | Mnemonic | 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 Half Word and Zero | **lhz** | **r**D,d(**r**A) | The EA is the sum (**r**A\|0) + d. The half word 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 Half Word and Zero Indexed | **lhzx** | **r**D,**r**A,**r**B | The EA is the sum (**r**A\|0) + (**r**B). The half word 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 Half Word and Zero with Update | **lhzu** | **r**D,d(**r**A) | The EA is the sum (**r**A) + d. The half word 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 Half Word and Zero with Update Indexed | **lhzux** | **r**D,**r**A,**r**B | The EA is the sum (**r**A) + (**r**B). The half word 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 Half Word Algebraic | **lha** | **r**D,d(**r**A) | The EA is the sum (**r**A\|0) + d. The half word 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 half word. |
| Load Half Word Algebraic Indexed | **lhax** | **r**D,**r**A,**r**B | The EA is the sum (**r**A\|0) + (**r**B). The half word 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 half word. |
| Load Half Word Algebraic with Update | **lhau** | **r**D,d(**r**A) | The EA is the sum (**r**A) + d. The half word 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 half word. The EA is placed into **r**A. |
| Load Half Word Algebraic with Update Indexed | **lhaux** | **r**D,**r**A,**r**B | The EA is the sum (**r**A) + (**r**B). The half word 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 half word. The EA is placed into **r**A. |
| Load Word and Zero | **lwz** | **r**D,d(**r**A) | The EA is the sum (**r**A\|0) + d. The word in memory addressed by the EA is loaded into **r**D. |
| Load Word and Zero Indexed | **lwzx** | **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 **r**D. |

**Table 4-13. Integer Load Instructions (continued)**

| Name | Mnemonic | Syntax | Operation |
|---|---|---|---|
| Load Word and Zero with Update | **lwzu** | **r**D,d(**r**A) | The EA is the sum (**r**A) + d. The word in memory addressed by the EA is loaded into **r**D. The EA is placed into **r**A. |
| Load Word and Zero with Update Indexed | **lwzux** | **r**D,**r**A,**r**B | The EA is the sum (**r**A) + (**r**B). The word 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, half word, word or double word 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 architecture defines a sequential execution model. However, when a store instruction modifies a location that contains an instruction, software synchronization is required to ensure that subsequent instruction fetches from that location obtain the modified version of the instruction.

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 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 | Syntax | Operation |
|---|---|---|---|
| Store Byte | **stb** | **r**S,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** | **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 byte in memory addressed by the EA. |
| Store Byte with Update | **stbu** | **r**S,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** | **r**S,**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. |

**Table 4-14. Integer Store Instructions (continued)**

| Name | Mnemonic | Syntax | Operation |
|---|---|---|---|
| Store Half Word | **sth** | **r**S,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 half word in memory addressed by the EA. |
| Store Half Word Indexed | **sthx** | **r**S,**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 half word in memory addressed by the EA. |
| Store Half Word with Update | **sthu** | **r**S,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 half word in memory addressed by the EA. The EA is placed into **r**A. |
| Store Half Word with Update Indexed | **sthux** | **r**S,**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 half word in memory addressed by the EA. The EA is placed into **r**A. |
| Store Word | **stw** | **r**S,d(**r**A) | The EA is the sum (**r**A\|0) + d. The contents of **r**S are stored into the word in memory addressed by the EA. |
| Store Word Indexed | **stwx** | **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 word in memory addressed by the EA. |
| Store Word with Update | **stwu** | **r**S,d(**r**A) | The EA is the sum (**r**A) + d. The contents of **r**S are stored into the word in memory addressed by the EA. The EA is placed into **r**A. |
| Store Word with Update Indexed | **stwux** | **r**S,**r**A,**r**B | The EA is the sum (**r**A) + (**r**B). The contents of **r**S are stored into the word 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 implementations, load byte-reverse instructions may have greater latency than other load instructions.

When used in a 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 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 | Syntax | Operation |
|---|---|---|---|
| Load Half Word 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 half word addressed by the EA are loaded into the low-order eight bits of **r**D. The next eight higher-order bits of the half word 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 16–23 of **r**D. Bits 16–23 of the word in memory addressed by the EA are loaded into bits 8–15. Bits 24–31 of the word in memory addressed by the EA are loaded into bits 0–7. The remaining bits in **r**D are cleared. |

**Table 4-15. Integer Load and Store with Byte-Reverse Instructions (continued)**

| Name | Mnemonic | Syntax | Operation |
|---|---|---|---|
| Store Half Word 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 half word 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 half word 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 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 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 | 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 = 5, and the last register loaded or stored is less than or equal to 12.

In some 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 | 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 6.4.6, “Alignment Exception (0x00600),” a misaligned string operation suffers a performance penalty compared to an aligned operation of the same type. A non–word-aligned string operation that crosses a double-word 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.

#### 4.2.3.7.1 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 32 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.

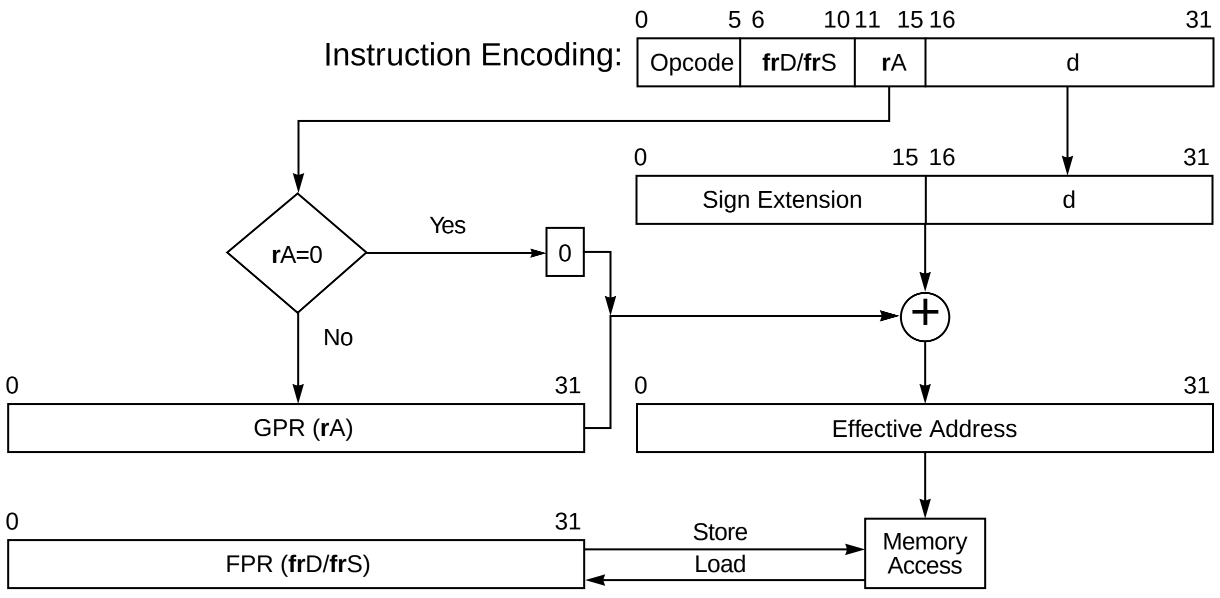


**Figure 4-4. Register Indirect with Immediate Index Addressing for Floating-Point Loads/Stores**

### 4.2.3.7.2 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.

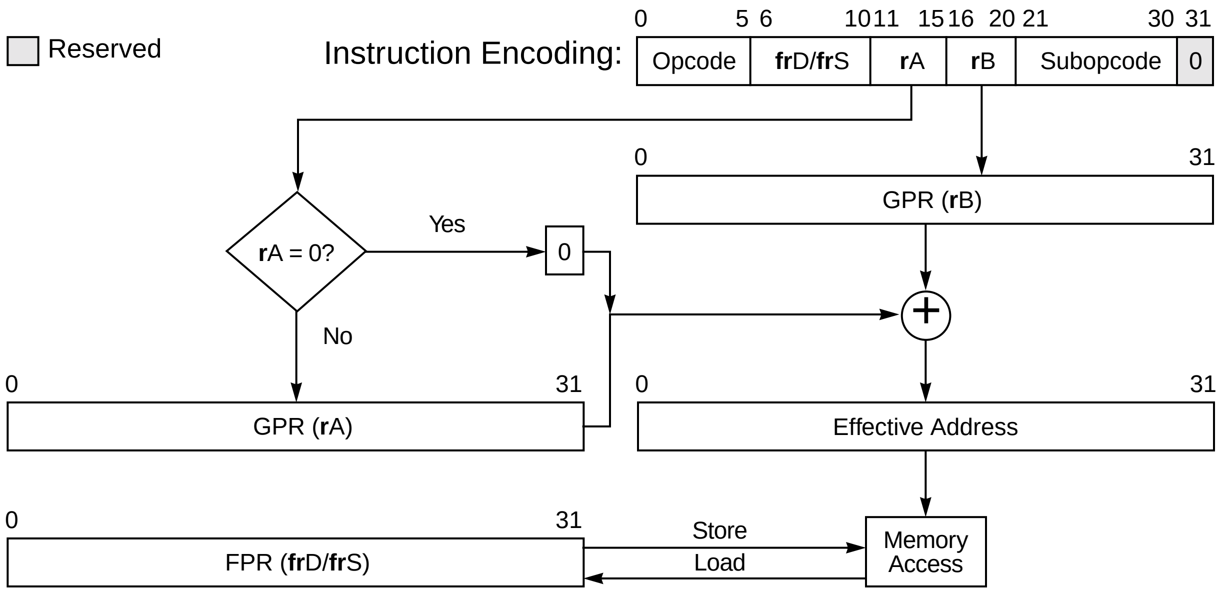


**Figure 4-5. Register Indirect with Index Addressing for Floating-Point Loads/Stores**

The 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 architecture defines that FPSCR[UE] 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 Section D.6, “Floating-Point Load Instructions.” Table 4-18 provides a summary of the floating-point load instructions.

Note that the architecture defines load with update instructions with **r**A = 0 as an invalid form.

**Table 4-18. Floating-Point Load Instructions**

| Name | Mnemonic | Syntax | Operation |
|---|---|---|---|
| Load Floating-Point Single | **lfs** | **fr**D,d**(r**A**)** | The EA is the sum (**r**A\|0) + d.<br>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).<br>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.<br>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.<br>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).<br>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.<br>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.<br>The double word in memory addressed by the EA is placed into **fr**D. |
| Load Floating-Point Double Indexed | **lfdx** | **fr**D**,r**A**,r**B | The EA is the sum (**r**A\|0) + (**r**B).<br>The double word in memory addressed by the EA is placed into **fr**D. |
| Load Floating-Point Double with Update | **lfdu** | **fr**D,d**(r**A**)** | The EA is the sum (**r**A) + d.<br>The double word in memory addressed by the EA is placed into **fr**D.<br>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).<br>The double word in memory addressed by the EA is placed into **fr**D.<br>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. (Note that the **stfiwx** instruction is defined as optional by the architecture to ensure backwards compatibility with earlier processors; however, it will likely be required for subsequent processors.) 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 Section D.7, “Floating-Point Store Instructions.” Table 4-19 provides a summary of the floating-point store instructions.

Note that the architecture defines store with update instructions with **r**A = 0 as an invalid form.

Table 4-19 provides the floating-point store instructions.

**Table 4-19. Floating-Point Store Instructions**

| Name | Mnemonic | Syntax | Operation |
|---|---|---|---|
| Store Floating-Point Single | **stfs** | **fr**S,d**(r**A**)** | The EA is the sum (**r**A\|0) + d.<br>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).<br>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.<br>The contents of **fr**S are converted to single-precision and stored into the word in memory addressed by the EA.<br>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).<br>The contents of **fr**S are converted to single-precision and stored into the word in memory addressed by the EA.<br>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.<br>The contents of **fr**S are stored into the double word 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).<br>The contents of **fr**S are stored into the double word 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.<br>The contents of **fr**S are stored into the double word in memory addressed by the EA.<br>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).<br>The contents of **fr**S are stored into the double word in memory addressed by EA.<br>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).<br>The contents of the low-order 32 bits of **fr**S are stored, without conversion, into the word in memory addressed by the EA.<br>**Note**: The **stfiwx** instruction is defined as optional by the architecture to ensure backwards compatibility with earlier processors; however, it will likely be required for subsequent processors. |

### 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 scans the execution pipelines to determine whether an instruction in progress may affect the particular CR bit. If no interlock is found, the branch can be resolved immediately by checking the bit in the CR and taking the action defined for the branch instruction.

If an interlock is detected, the branch is considered unresolved and the direction of the branch may either be predicted using the *y* bit (as described in Table 4-20) or by using dynamic prediction. The interlock is monitored while instructions are fetched for the predicted branch. When the interlock is cleared, the processor determines whether the prediction was correct based on the value of the CR bit. If the prediction is correct, the branch is considered completed and instruction fetching continues. 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 two low-order bits of the generated branch target address are ignored.

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.

##### 4.2.4.1.1 Branch Relative Addressing Mode

Instructions that use branch relative addressing generate the next instruction address by sign extending and appending 0b00 to the immediate displacement operand LI, and adding the resultant value to the current instruction address. Branches using this 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.

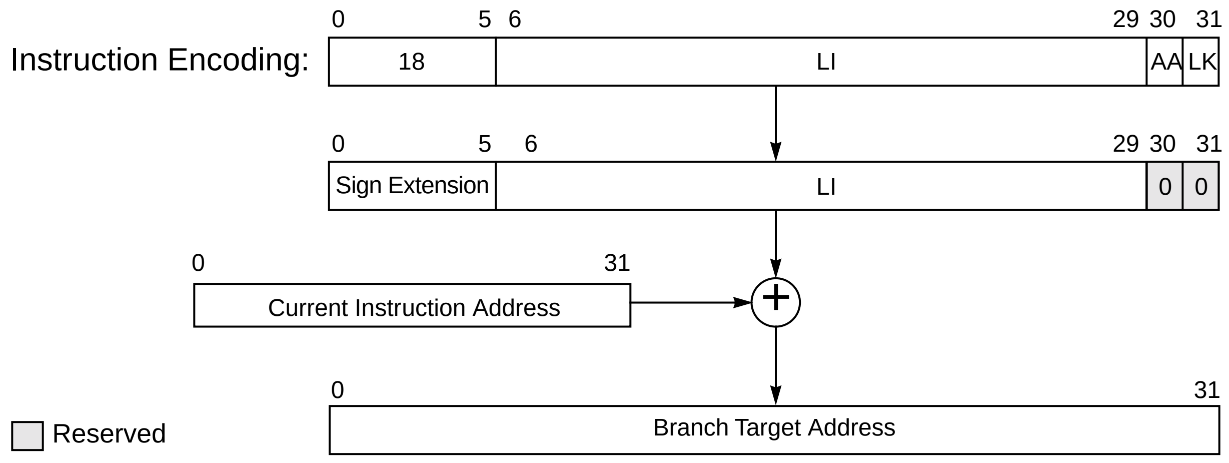


**Figure 4-6. Branch Relative Addressing**

### 4.2.4.1.2 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 0b00 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.

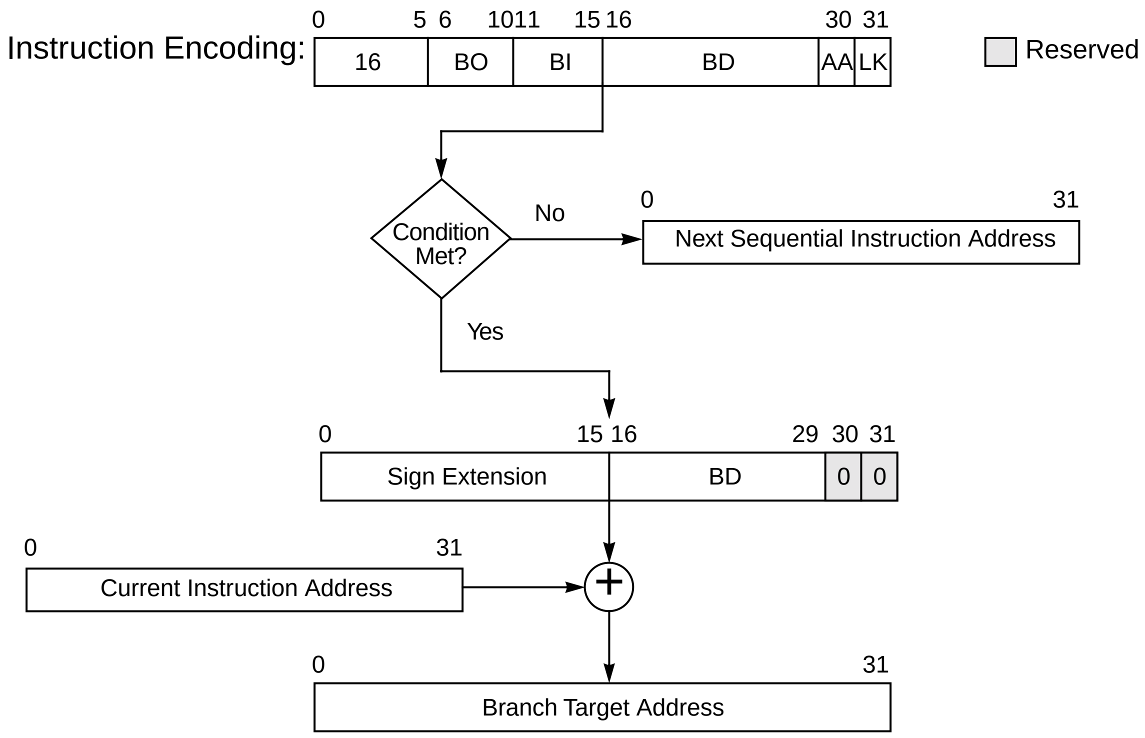


**Figure 4-7. Branch Conditional Relative Addressing**

### 4.2.4.1.3 Branch to Absolute Addressing Mode

Instructions that use branch to absolute addressing mode generate the next instruction address by sign extending and appending 0b00 to the LI operand. Branches using this addressing mode have the absolute addressing option enabled (AA field, bit 30, in the instruction encoding = 1). The 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.

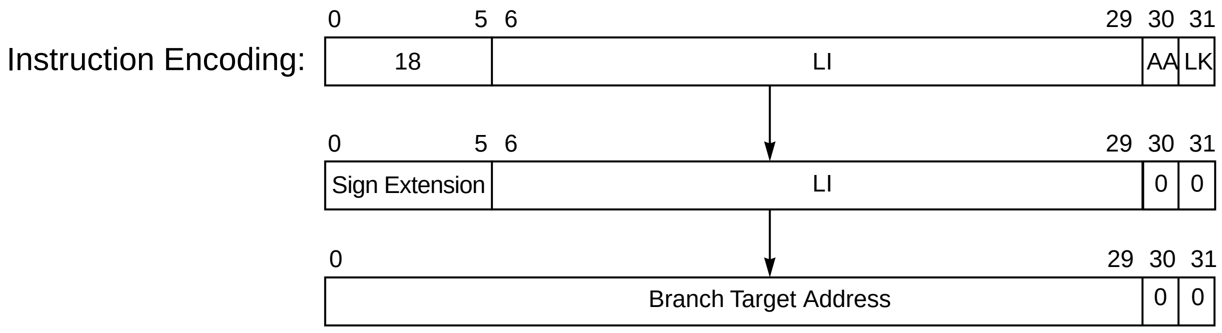


**Figure 4-8. Branch to Absolute Addressing**

#### 4.2.4.1.4 Branch Conditional to Absolute Addressing Mode

If the branch conditions are met, instructions that use the branch conditional to absolute addressing mode generate the next instruction address by sign extending and appending 0b00 to the BD operand. Branches using this addressing mode have the absolute addressing option enabled (AA field, bit 30, in the instruction encoding = 1). The 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.

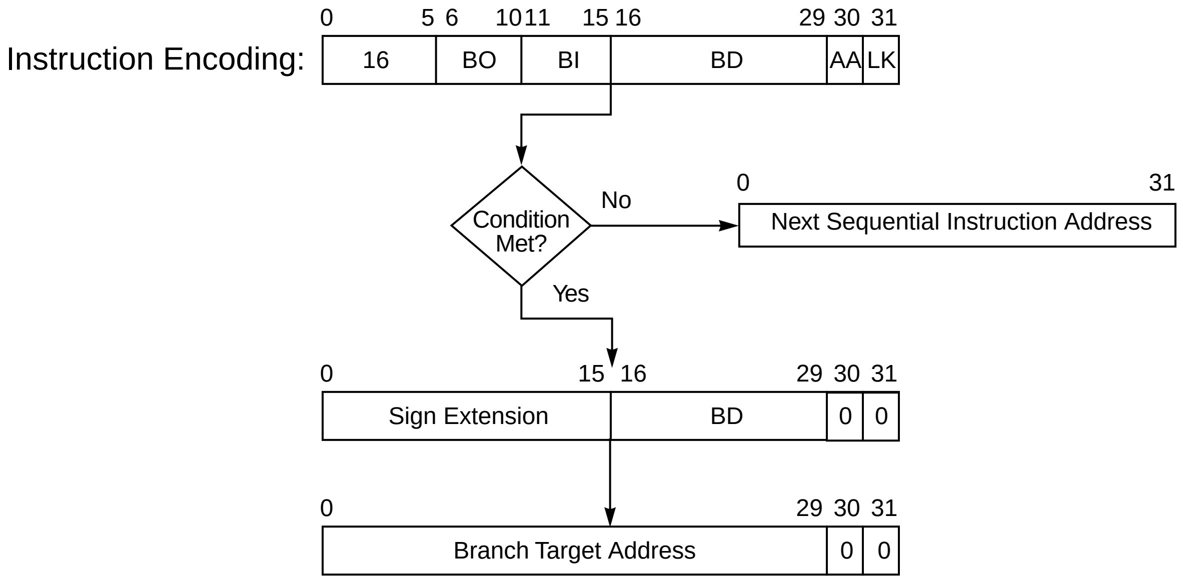


**Figure 4-9. Branch Conditional to Absolute Addressing**

#### 4.2.4.1.5 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 fetching the contents of the LR and clearing the two low-order bits to zero. 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 4-10 shows how the branch target address is generated when using the branch conditional to link register addressing mode.

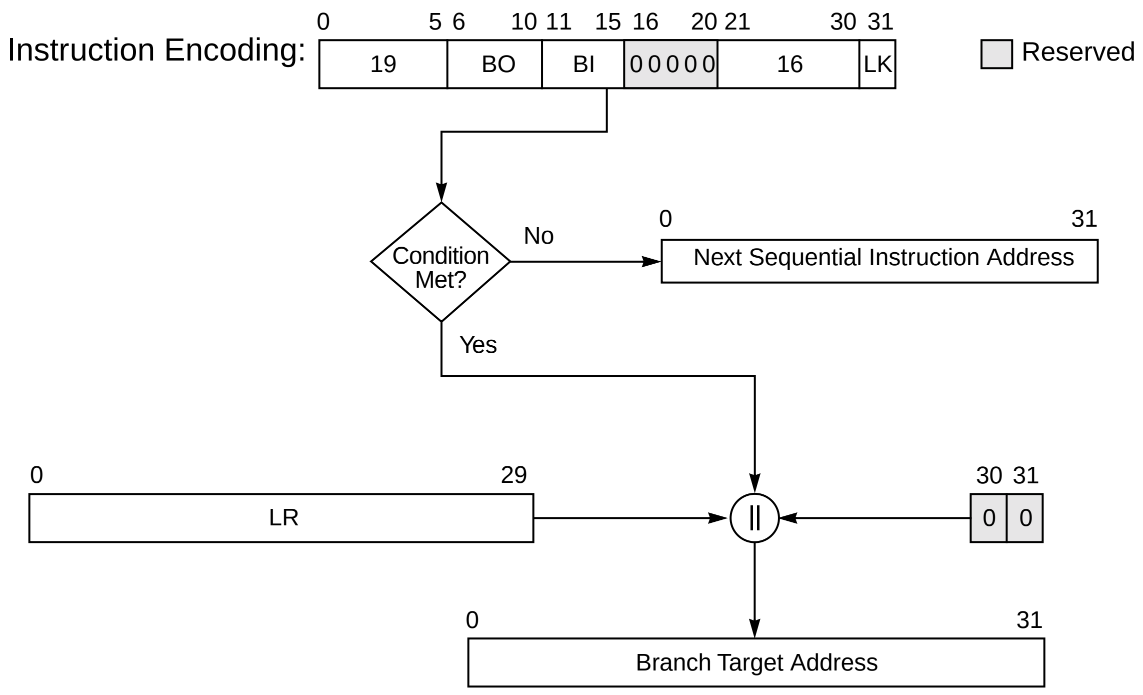


**Figure 4-10. Branch Conditional to Link Register Addressing**

#### 4.2.4.1.6 Branch Conditional to Count Register Addressing Mode

If the branch conditions are met, the branch conditional to count register instruction generates the next instruction address by fetching the contents of the count register (CTR) and clearing the two low-order bits to zero. The 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 4-11 shows how the branch target address is generated when using the branch conditional to count register addressing mode.

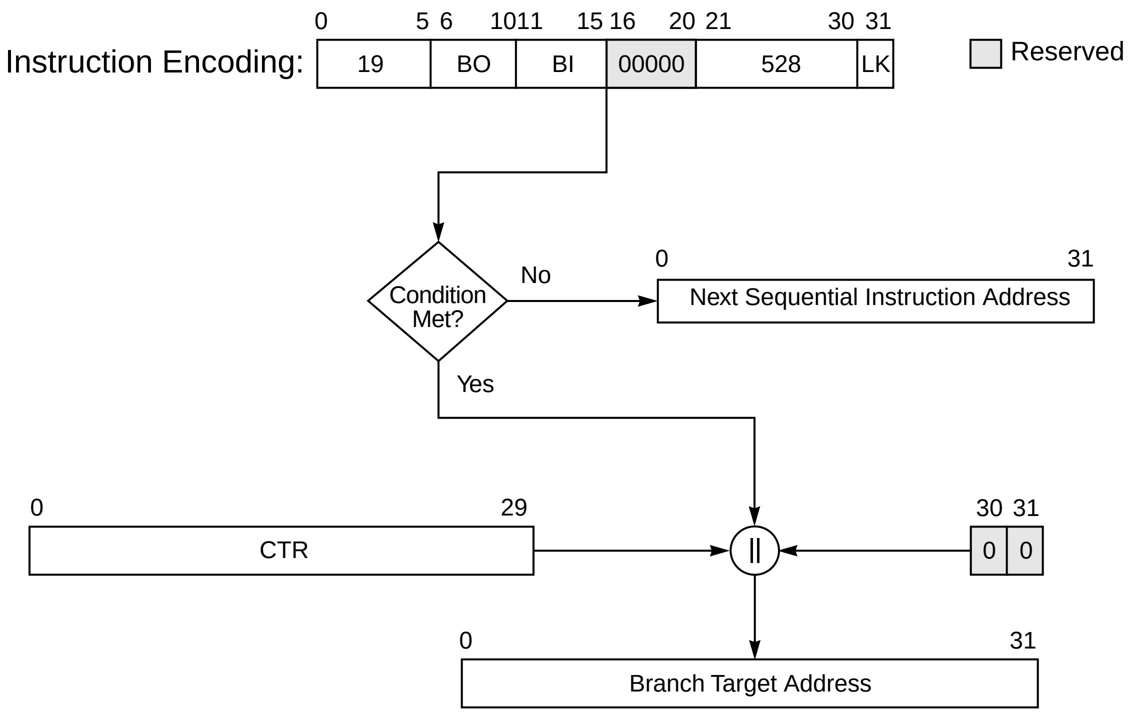


**Figure 4-11. Branch Conditional to Count Register Addressing**

### 4.2.4.2 Conditional Branch Control

For branch conditional instructions, the BO operand specifies the conditions under which the branch is taken. The first four bits of the BO operand specify how the branch is affected by or affects the condition and count registers. The fifth bit, shown in Table 4-20 as having the value *y*, is used by some implementations for branch prediction as described below.

The encodings for the BO operands are shown in Table 4-20.

**Table 4-20. BO Operand Encodings**

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

**Table 4-20. BO Operand Encodings (continued)**

| BO | Description |
|---|---|
| 1*z*1*zz* | Branch always |
| In this table, *z* indicates a bit that is ignored. Note that the *z* bits should be cleared, as they may be assigned a meaning in some future version of the architecture. The *y* bit provides a hint about whether a conditional branch is likely to be taken, and may be used by some implementations to improve performance. | |

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

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

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

Setting the *y* bit reverses the preceding indications.

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

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

((BO[0] & BO[2]) | S) ≈ BO[4]

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

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

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

Branching can be conditional or unconditional, and optionally a branch return address is created by the access of the effective address of the instruction following the branch instruction in the LR after the branch target address has been computed. This is done regardless of whether the branch is taken. 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 them in the count register, and use one of the branch conditional instructions 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) 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 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 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).

### 4.2.4.3 Branch Instructions

Table 4-21 describes the branch instructions provided by the processors.

**Table 4-21. Branch Instructions**

| Name | Mnemonic | Syntax | Operation |
|---|---|---|---|
| Branch | **b**<br>**ba**<br>**bl**<br>**bla** | target_addr | **b** Branch. Branch to the address computed as the sum of the immediate address and the address of the current instruction.<br>**ba** Branch Absolute. Branch to the absolute address specified.<br>**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).<br>**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**<br>**bca**<br>**bcl**<br>**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.<br>**bc** Branch Conditional. Branch conditionally to the address computed as the sum of the immediate address and the address of the current instruction.<br>**bca** Branch Conditional Absolute. Branch conditionally to the absolute address specified.<br>**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.<br>**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**<br>**bclrl** | BO**,**BI | 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.<br>**bclr** Branch Conditional to Link Register. Branch conditionally to the address in the LR.<br>**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**<br>**bcctrl** | BO**,**BI | 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.<br>**bcctr** Branch Conditional to Count Register. Branch conditionally to the address specified in the count register.<br>**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.<br>**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 F, “Simplified Mnemonics,” for a list of simplified mnemonic examples.

### 4.2.4.5 Condition Register Logical Instructions

Condition register logical instructions, shown in Table 4-22, and the Move Condition Register Field (**mcrf**) instruction are also defined as flow control instructions.

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

**Table 4-22. Condition Register Logical Instructions**

| Name | Mnemonic | 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 4-23 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 F, “Simplified Mnemonics,” for a complete set of simplified mnemonics.

**Table 4-23. Trap Instructions**

| Name | Mnemonic | Syntax | Description |
|---|---|---|---|
| Trap Word Immediate | **twi** | 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 | **tw** | 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. |

### 4.2.4.7 System Linkage Instruction—UISA

Table 4-24 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 4-24. System Linkage Instruction—UISA**

| Name | Mnemonic | 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.”<br>See Section 4.4.1, “System Linkage Instructions—OEA,” for a complete description of the **sc** instruction. |

## 4.2.5 Processor Control Instructions—UISA

U V O 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

U Table 4-25 summarizes the instructions for reading from or writing to the condition register.

**Table 4-25. Move to/from Condition Register Instructions**

| Name | Mnemonic | Syntax | Operation |
|---|---|---|---|
| Move to Condition Register Fields | **mtcrf** | CRM,**r**S | The contents 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 **r**S. |
| Move to Condition Register from XER | **mcrxr** | **crf**D | The contents of XER[0–3] are copied into the condition register field designated by **crf**D. All other CR fields remain unchanged. The contents of XER[0–3] are cleared. |
| Move from Condition Register | **mfcr** | **r**D | The contents of the CR are placed into **r**D. |

### 4.2.5.2 Move to/from Special-Purpose Register Instructions (UISA)

Table 4-26 provides a brief description of the **mtspr** and **mfspr** instructions. For more detailed information refer to Chapter 8, “Instruction Set.”

**Table 4-26. Move to/from Special-Purpose Register Instructions (UISA)**

| Name | Mnemonic | Syntax | Operation |
|---|---|---|---|
| Move to Special-Purpose Register | **mtspr** | SPR,**r**S | The value specified by **r**S are placed in the specified SPR. |
| Move from Special-Purpose Register | **mfspr** | **r**D,SPR | The contents of the specified SPR are placed in **r**D. |

## 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 architecture defines the **sync** instruction with CR update enabled (Rc field, bit 31 = 1) to be an invalid form.

The proper paired use of the **lwarx** with **stwcx.** instructions allows programmers to emulate common semaphore operations such as test and set, compare and swap, exchange memory, and fetch and add. Examples of these semaphore operations can be found in Appendix E, “Synchronization Programming Examples.” The **lwarx** instruction must be paired with an **stwcx.** instruction with the same effective address specified by both instructions of the pair. 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. Note that the reservation granularity is implementation-dependent.

U
V

The concept behind the use of the **lwarx** and **stwcx.** 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. The conditional store is performed based upon the existence of a reservation established by the preceding **lwarx** 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.

The **lwarx** and **stwcx.** 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** and **stwcx.** instructions require the EA to be aligned.

In general, the **lwarx** and **stwcx.** 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** instruction. The conditional store is performed based upon the existence of a reservation established by the preceding **lwarx** 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** instruction; this clears the first reservation and establishes a new one.
- The processor holding the reservation executes any **stwcx.** instruction 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**, **dcbf**, or **dcbi** to the same reservation granule; whether the reservation is cleared is undefined.
- Some other processor executes a **dcba** to the same reservation granule. The reservation is cleared if the instruction causes the target block to be newly established in the data cache or to be modified; otherwise, whether the reservation is cleared is undefined.
- Some other mechanism modifies a memory location in the same reservation granule.

  Note that exceptions do not clear reservations; however, system software invoked by exceptions may clear reservations.

Table 4-27 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 4-27. Memory Synchronization Instructions—UISA**

| Name | Mnemonic | Syntax | Operation |
|---|---|---|---|
| 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 **r**D. |
| Store Word Conditional Indexed | **stwcx.** | **r**S,**r**A,**r**B | The EA is the sum (**r**A\|0) + (**r**B).<br>If a reservation exists and the EA specified by **stwcx.** 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 word addressed by the EA and the reservation is cleared.<br>If a reservation exists but the EA 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 **r**S are stored into the word in memory addressed by the EA.<br>If a reservation does not exist, the instruction completes without altering memory or the contents of the cache. |
| Synchronize | **sync** | — | 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.<br>See Chapter 8, “Instruction Set,” for more information. |

### 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 Section F.9, “Recommended Simplified Mnemonics.” Programs written to be portable across the various assemblers for the architecture should not assume the existence of mnemonics not described in this document.

For a complete list of simplified mnemonics, see Appendix F, “Simplified Mnemonics.”

## 4.3 VEA Instructions

U V O The 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

V 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 4-28 describes the **mftb** instruction.

Simplified mnemonics are provided (See Section F.8, “Simplified Mnemonics for SPRs”) 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.

On 32-bit implementations, it is not possible to read the entire 64-bit time base register in a single instruction. 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 4-28. Move from Time Base Instruction**

| Name | Mnemonic | 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 4-29 and Table 4-30. The contents of the designated register are copied to **r**D. |

Table 4-29 summarizes the time base (TBL/TBU) register encodings to which user-level access (using **mftb**) is permitted (as specified by the VEA).

**Table 4-29. 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 4-30 summarizes the TBL and TBU register encodings to which supervisor-level access (using **mtspr**) is permitted.

**Table 4-30. 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) |

[1]Moving from the time base (TBL and TBU) can also be accomplished with the **mftb** instruction.

## 4.3.2 Memory Synchronization Instructions—VEA

U

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.

V

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 4-31. 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 (which has the effect of clearing the pipeline behind the **isync** instruction).

**Table 4-31. Memory Synchronization Instructions—VEA**

| Name | Mnemonic | 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

V

Memory control instructions include the following types:

O

- Cache management instructions (user-level and supervisor-level)
- Segment register manipulation instructions
- Segment lookaside buffer management 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

V 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.

O Note that when data address translation is disabled (MSR[DR] = 0), the Data Cache Block Clear to Zero (**dcbz**) and the Data Cache Block Allocate (**dcba**) instructions allocate a cache block in the cache and may 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.

Any cache control instruction that generates an effective address for which SR[T] = 1 is treated as a no-op.

Table 4-32 summarizes the cache instructions defined by the VEA. Note that these instructions are accessible to user-level programs. V

**Table 4-32. User-Level Cache Instructions**

| Name | Mnemonic | Syntax | Operation |
|---|---|---|---|
| Data Cache Block Touch | **dcbt** | **r**A,**r**B | The EA is the sum (**r**A\|0) + (**r**B).<br>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. |
| Data Cache Block Touch for Store | **dcbtst** | **r**A,**r**B | The EA is the sum (**r**A\|0) + (**r**B).<br>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. |

**Table 4-32. User-Level Cache Instructions (continued)**

| Name | Mnemonic | Syntax | Operation |
|---|---|---|---|
| Data Cache Block Allocate | **dcba** | **r**A,**r**B | The EA is the sum (**r**A\|0) + (**r**B).<br>If the cache block containing the byte addressed by the EA is in the data cache, all bytes of the cache block are made undefined, but the cache block is still considered valid. Note that programming errors can occur if the data in this cache block is subsequently read or used inadvertently.<br>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 allocated (and made valid) in the data cache without fetching the block from main memory, and the value of all bytes of the cache block is undefined.<br>If the page containing the byte addressed by the EA is marked caching inhibited (WIM = x1x), this instruction is treated as a no-op.<br>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.<br>The **dcba** 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.<br>This instruction is optional in the architecture.<br>(In the OEA, the **dcba** instruction is additionally defined to clear all bytes of a newly established block to zero in the case that the block did not already exist in the cache.) |
| Data Cache Block Clear to Zero | **dcbz** | **r**A,**r**B | The EA is the sum (**r**A\|0) + (**r**B).<br>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.<br>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.<br>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.<br>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.<br>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 4-32. User-Level Cache Instructions (continued)**

| Name | Mnemonic | Syntax | Operation |
|---|---|---|---|
| Data Cache Block Store | **dcbst** | **r**A,**r**B | The EA is the sum(**r**A\|0) + (**r**B).<br>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.<br>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.<br>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.<br>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**.** |

**Table 4-32. User-Level Cache Instructions (continued)**

| Name | Mnemonic | Syntax | Operation |
|---|---|---|---|
| Data Cache Block Flush | **dcbf** | **r**A,**r**B | The EA is the sum (**r**A\|0) + (**r**B).<br>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.<br>• •Coherency required (WIM = xx1)<br>— Unmodified block—Invalidates copies of the block in the caches of all processors.<br>— Modified block—Copies the block to memory. Invalidates copies of the block in the caches of all processors.<br>— 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.<br>• •Coherency not required (WIM = xx0)<br>— Unmodified block—Invalidates the block in the processor's cache.<br>— Modified block—Copies the block to memory. Invalidates the block in the processor's cache.<br>— Absent block—Does nothing.<br>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.<br>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. |
| Instruction Cache Block Invalidate | **icbi** | **r**A,**r**B | The EA is the sum (**r**A\|0) + (**r**B).<br>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.<br>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.<br>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.<br>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. |

## 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 4-33.

**Table 4-33. External Control Instructions**

| Name | Mnemonic | Syntax | Operation |
|---|---|---|---|
| External Control In Word Indexed | **eciwx** | **r**D,**r**A,**r**B | The EA is the sum (**r**A\|0) + (**r**B).<br>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 **r**D. The EA sent to the device must be word-aligned.<br>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**.<br>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).<br>A store word request for the physical address corresponding to the EA and the contents 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.<br>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.<br>This instruction is optional. |

## 4.4 OEA Instructions

The 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 4-34). 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 **rfi** instruction is a supervisor-level instruction that is useful for returning from an exception handler.

**Table 4-34. System Linkage Instructions—OEA**

| Name | Mnemonic | Syntax | Operation |
|---|---|---|---|
| System Call | **sc** | — | When executed, the effective address of the instruction following the **sc** instruction is placed into SRR0. Bits 1–4, and 10–15 of SRR1 are cleared. Additionally, bits 16–23, 25–27, and 30–31of 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.”<br>The exception causes the next instruction to be fetched from offset 0xC00 from the base physical address indicated by the new setting of MSR[IP].<br>This instruction is context synchronizing. |
| Return from Interrupt **(32-bit only)** | **rfi** | — | Bits 16–23, 25–27, and 30–31 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[0–29] \|\| 0b00.<br>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.<br>This is a supervisor-level instruction and is context-synchronizing.<br>This instruction is defined only for 32-bit implementations. The use of the **rfi** instruction on a 64-bit implementation will invoke the system exception handler. |

## 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 4-35 summarizes the instructions used for reading from and writing to the MSR.

**Table 4-35. Move to/from Machine State Register Instructions**

| Name | Mnemonic | Syntax | Operation |
|---|---|---|---|
| Move to Machine State Register **(32-bit only)** | **mtmsr** | **r**S | The contents of **r**S are placed into the MSR.<br>This instruction is a supervisor-level instruction and is context synchronizing except with respect to alterations to the POW and LE bits. Refer to Section 2.3.17, “Synchronization Requirements for Special Registers and for Lookaside Buffers,” for more information. |
| 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 4-36). For more detailed information, see Chapter 8, “Instruction Set.” Simplified mnemonics are provided for the **mtspr** and **mfspr** instructions in Appendix F, “Simplified Mnemonics.” For a discussion of context synchronization requirements when altering certain SPRs, refer to Appendix E, “Synchronization Programming Examples.”

**Table 4-36. Move to/from Special-Purpose Register Instructions (OEA)**

| Name | Mnemonic | 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 **r**S are placed into the SPR.<br>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 that there are additional SPRs specific to each implementation; for implementation-specific SPRs, see the user’s manual for that 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 Supervisor-Level Cache Management Instruction

Table 4-37 summarizes the operation of the only supervisor-level cache management instruction. See Section 4.3.3.1, “User-Level Cache Instructions—VEA,” for cache instructions that provide user-level programs the ability to manage the on-chip caches.

Note that any cache control instruction that generates an effective address for which SR[T] = 1 is treated as a no-op.

**Table 4-37. Cache Management Supervisor-Level Instruction**

| Name | Mnemonic | Syntax | Operation |
|---|---|---|---|
| Data Cache Block Invalidate | **dcbi** | **r**A,**r**B | The EA is the sum (**r**A\|0) + (**r**B).<br>The action taken depends on the memory mode associated with the target, and the state (modified, unmodified) of the cache block. The following list describes the action to take if the cache block containing the byte addressed by the EA is or is not in the cache.<br>• •Coherency required (WIM = xx1)<br>— Unmodified cache block—Invalidates copies of the cache block in the caches of all processors.<br>— Modified cache block—Invalidates copies of the cache block in the caches of all processors. (Discards the modified contents.)<br>— Absent cache block—If copies are in the caches of any other processor, causes the copies to be invalidated. (Discards any modified contents.)<br>• •Coherency not required (WIM = xx0)<br>— Unmodified cache block—Invalidates the cache block in the local cache.<br>— Modified cache block—Invalidates the cache block in the local cache. (Discards the modified contents.)<br>— Absent cache block—No action is taken.<br>When data address translation is enabled, MSR[DT]=1, and the logical (effective) address has no translation, a data access exception occurs.<br>The function of this instruction is independent of the write-through and cache-inhibited/allowed modes determined by the WIM bit settings of the block containing the byte addressed by the EA.<br>This instruction is treated as a store to the addressed byte with respect to address translation and protection, except that the change bit need not be set, and if the change bit is not set then the reference bit need not be set. |

## 4.4.3.2 Segment Register Manipulation Instructions

The instructions listed in Table 4-38 provide access to the segment registers for 32-bit implementations. These instructions operate completely independently of the MSR[IR] and MSR[DR] bit settings. Refer to Section 2.3.17, “Synchronization Requirements for Special Registers and for Lookaside Buffers,” for serialization requirements and other recommended precautions to observe when manipulating the segment registers.

**Table 4-38. Segment Register Manipulation Instructions**

| Name | Mnemonic | Syntax | Operation |
|---|---|---|---|
| Move to Segment Register (**32-bit only**) | **mtsr** | SR,**r**S | The contents of **r**S are placed into segment register specified by operand SR.<br>This is a supervisor-level instruction. |
| Move to Segment Register Indirect (**32-bit only**) | **mtsrin** | **r**S,**r**B | The contents of **r**S are copied to the segment register selected by bits 0–3 of **r**B.<br>This is a supervisor-level instruction. |

**Table 4-38. Segment Register Manipulation Instructions (continued)**

| Name | Mnemonic | Syntax | Operation |
|---|---|---|---|
| Move from Segment Register (**32-bit only**) | **mfsr** | **r**D,SR | The contents of the segment register specified by operand SR are placed into **r**D.<br>This is a supervisor-level instruction. |
| Move from Segment Register Indirect (**32-bit only**) | **mfsrin** | **r**D,**r**B | The contents of the segment register selected by bits 0–3 of **r**B are copied into **r**D.<br>This is a supervisor-level instruction. |

## 4.4.3.3 Translation Lookaside Buffer Management Instructions

The address translation mechanism is defined in terms of segment descriptors and page table entries (PTEs) used 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.

For performance reasons, many processors implement one or more translation lookaside buffers on-chip. These are caches of portions of the page table. As changes are made to the address translation tables, it is necessary to maintain coherency between the TLB and the updated tables. This is done by invalidating TLB entries, or occasionally by invalidating the entire TLB, and allowing the translation caching mechanism to refetch from the tables.

Each implementation that has a TLB provides means for invalidating an individual TLB entry and invalidating the entire TLB.

If a processor does not implement a TLB, it treats the corresponding instructions (**tlbie**, **tlbia**, and **tlbsync**) either as no-ops or as illegal instructions.

Refer to Chapter 7, “Memory Management,” for more information about TLB operation. Table 4-39 summarizes the operation of the SLB and TLB instructions.

**Table 4-39. Translation Lookaside Buffer Management Instructions**

| Name | Mnemonic | Syntax | Operation |
|---|---|---|---|
| TLB Invalidate Entry | **tlbie** | **r**B | The EA is the contents of **r**B. If the TLB contains an entry corresponding to the EA, that entry is removed from the TLB. The TLB search is performed regardless of the settings of MSR[IR,DR]. Any block address translation for the EA is ignored.<br>This instruction causes the target TLB entry to be invalidated in all processors.<br>The operation performed by this instruction is treated as a caching inhibited and guarded data access with respect to the ordering performed by **eieio**.<br>This is a supervisor-level instruction and optional in the architecture. |

**Table 4-39. Translation Lookaside Buffer Management Instructions (continued)**

| Name | Mnemonic | Syntax | Operation |
|---|---|---|---|
| TLB Invalidate All | **tlbia** | — | All TLB entries are made invalid. The TLB is invalidated regardless of the MSR[IR,DR] settings. This instruction does not cause the entries to be invalidated in other processors. This is a supervisor-level instruction and optional in the architecture. |
| TLB Synchronize | **tlbsync** | — | Executing a **tlbsync** instruction ensures that all **tlbie** instructions previously executed by the processor executing **tlbsync** 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**.<br>This is a supervisor-level instruction and optional in the architecture. |

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.

# Chapter 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.

## 5.1 Overview

The PowerPC architecture provides for relaxed memory coherency. Features such as write-back caching and memory access reordering 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. This chapter uses the term ‘cache block’ to distinguish it from the unit of memory addressed by the block address translation (BAT) mechanism. The cache block size can vary by implementation. In addition, the unit of memory at which coherency is maintained is called the coherence block, the size of which is also implementation-specific, although it is typically the same size as a cache block.

## 5.2 The Virtual Environment

The user instruction set architecture (UISA) relies upon a memory space of $2^{32}$ 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."

## 5.2.1 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 5.2.4, "Memory Coherency," and Section 5.3.1, "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 8, "Instruction Set," contains additional information about **sync** and **eieio**.

### 5.2.1.1 Enforce In-Order Execution of I/O Instruction

The **eieio** instruction creates a memory barrier, which provides an ordering function for the memory accesses caused by load, store, **dcbz**, and **dcba** instructions executed by the processor executing the **eieio** instruction. These accesses are divided into two sets, which are ordered separately. The access caused by **dcbz** or **dcba** is ordered as a store.

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** have been performed.

If stronger ordering is desired, **sync** must be used.

#### 5.2.1.2 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**, 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 is described in Chapter 8, “Instruction Set.”

### 5.2.2 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)
- Half-word accesses aligned on half-word boundaries
- Word accesses aligned on word 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
- Floating-point double-word accesses in 32-bit implementations
- Any cache management instructions

The **lwarx**/**stwcx.** instruction combinations can be used to perform atomic memory references. The **lwarx** is a load from a word-aligned location that has two side effects:

1. A reservation for a subsequent **stwcx.** is created.

2. The memory coherence mechanism is notified that a reservation exists for the memory location accessed by the **lwarx**.

The **stwcx.** is a store to a word-aligned location that is conditioned on the existence of the reservation created by **lwarx** and on whether the same memory location is specified by both instructions and whether the instructions are issued by the same processor.

In a multiprocessor system, every processor (other than the one executing **lwarx**/**stwcx.**) that might update the location must configure the addressed page as memory coherency required. The **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. (Note that exceptions are referred to as interrupts in the architecture specification.)

The **lwarx**/**stwcx.** combination is described in Section 4.3.2, “Memory Synchronization Instructions—VEA,” and Chapter 8, “Instruction Set.”

### 5.2.3 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 does not halt execution (note that **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.

### 5.2.4 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 **lwarx**/**stwcx.** pair. 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. It is important to note that the VEA (user level) does not specify the page size.

#### 5.2.4.1 Memory/Cache Access Modes

The OEA defines the set of memory/cache access modes and the mechanism to implement these modes. Refer to Section 5.3.1, “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.

##### 5.2.4.1.1 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, **dcbst** and **dcbf** can explicitly force an update of main memory.

The write-through attribute is meaningless for locations designated as caching-inhibited.

##### 5.2.4.1.2 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.

#### 5.2.4.1.3 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 a **sync** are the previous accesses it executed guaranteed to have been performed with respect to all other processors and mechanisms.

#### 5.2.4.1.4 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** 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.

#### 5.2.4.1.5 Pages Designated as Guarded

The guarded attribute pertains to speculative execution. Refer to Section 5.3.1.5.4, “Speculative Accesses to Guarded Memory,” for more information about speculative execution. Note that the term ‘speculative’ is referred to as out-of-order in the architecture specification. The use of these terms in this manual is described in Section 5.3.1.5.1, “Definition of Speculative and Out-of-Order Memory Accesses.”

Instructions and data cannot be accessed speculatively from a page designated as guarded. Additionally, if separate store instructions access memory that is caching-inhibited and guarded, accesses are performed in the order specified by the program. When a page is designated as not guarded, speculative fetches and accesses are allowed.

#### 5.2.4.2 Coherency Precautions

Mismatched memory/cache attributes cause coherency paradoxes in both single- and multiple-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 block or 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 such conditions to minimize the negative effects on memory coherency. Bus operations generated for specific instructions and state conditions are not defined by the architecture.

### 5.2.5 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.

Note that any cache control instruction that generates an EA that corresponds to a direct-store segment (SR[T] = 1) is treated as a no-op. However, the direct-store facility is being phased out of the architecture and will not likely be supported in future devices. Thus, software should not depend on its effects.

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.

#### 5.2.5.1 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 that physical address is referred to as real address in the architecture specification.

##### 5.2.5.1.1 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 **dcbt** 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 **dcbt**. 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 like 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 EA into their cache. For information about the R and C bits, see Section 7.6.3, “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.

#### 5.2.5.1.2 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.

#### 5.2.5.1.3 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(s) of any processor(s) and has been modified, the data is written to main memory.

- 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 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 **dcbst** 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).

#### 5.2.5.1.4 Data Cache Block Flush (dcbf) Instruction

The action taken depends on the memory/cache access mode associated with the addressed byte 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 copies of the cache block in the data caches of all processors.

  Target block not in cache—If a modified copy of the cache block is in the data cache(s) of any processor(s), **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 **dcbf** 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).

### 5.2.5.2 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 that most operating systems will provide a system service for this function. These operations are necessary because the memory may be designated as write-back. Because 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 fetch completes.

For implementations used in multiprocessor systems, variations on this sequence may be recommended. For example, in a multiprocessor system with a unified cache (at any level), if instructions are fetched without coherency being enforced, the preceding instruction sequence is inadequate. Because **icbi** does not invalidate blocks in a unified cache, a **dcbf** should be used instead of a **dcbst** for this case.

##### 5.2.5.2.1 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.

A memory access caused by an **icbi** is not necessarily recorded in the page tables. If it is recorded, it is treated as a load operation. Implementations that have a unified cache treat **icbi** as a no-op except that they may invalidate the target cache block in the instruction caches of other processors (in coherency required mode).

#### 5.2.5.2.2 Instruction Synchronize Instruction (isync)

The **isync** instruction provides an ordering function for the effects of all instructions executed by a processor. Executing an **isync** ensures that all instructions preceding the **isync** have completed before the **isync** 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** 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 **isync**. The **isync** has no effect on other processors or on their caches.

## 5.2.6 Shared Memory

The architecture supports sharing memory between programs, between different instances of the same program, and between processors and other mechanisms. It also supports access to a memory location by one or more programs using different effective addresses. In these cases memory is shared in blocks that are an integral number of pages. When one physical memory location has different effective addresses, the addresses are said to be aliases. Each application can be granted separate access privileges to aliased pages.

Section 5.2.6.2, “Lock Acquisition and Import Barriers,” gives examples of how **sync** and **eieio** are used to control memory access ordering when memory is shared among programs.

### 5.2.6.1 Memory Access Ordering

The memory model for memory access ordering is weakly consistent. This provides an opportunity for improved performance over a model with stronger consistency rules, but places the responsibility on the program to ensure that ordering or synchronization instructions are properly placed for the correct execution of the program.

The order in which the processor accesses memory, the order in which those accesses are performed with respect to other processors or mechanisms, and the order in which they are performed in main memory may all be different. The following ways to enforcing accesses ordering are provided to allow programs to share memory with other programs or with mechanisms such as I/O devices.

- If two store instructions specify memory locations that are both caching inhibited and guarded, the corresponding memory accesses are performed in program order with respect to any processor or mechanism.
- If a load instruction depends on the value returned by a preceding load (because the value is used to compute the effective address specified by the second load), the corresponding memory accesses are performed in program order with respect to any processor or mechanism to the extent required by the associated memory coherence required attributes (that is, the memory coherence required attribute, if any, associated with each access). This applies even if the dependency does not affect

program logic (for example, the value returned by the first load is ANDed with zero and then added to the effective address specified by the second load).

- When a processor (P1) executes **sync** or **eieio**, a memory barrier is created that separates applicable memory accesses into two groups, G1 and G2. G1 includes all applicable memory accesses associated with instructions preceding the barrier-creating instruction, and G2 includes all applicable memory accesses associated with instructions following the barrier-creating instruction.

Figure 5-1 shows an example using a two-processor system.

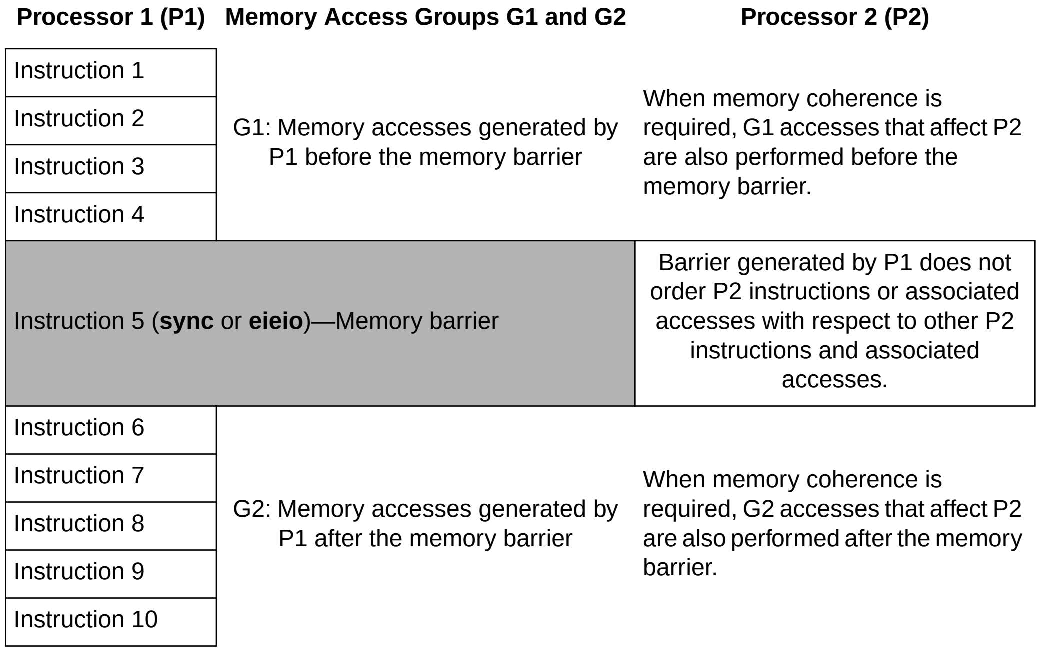

Processor 1 (P1)
Memory Access Groups G1 and G2
Processor 2 (P2)
Instruction 1
Instruction 2
Instruction 3
Instruction 4
G1: Memory accesses generated by P1 before the memory barrier
When memory coherence is required, G1 accesses that affect P2 are also performed before the memory barrier.
Instruction 5 (sync or eieio)—Memory barrier
Barrier generated by P1 does not order P2 instructions or associated accesses with respect to other P2 instructions and associated accesses.
Instruction 6
Instruction 7
Instruction 8
Instruction 9
Instruction 10
G2: Memory accesses generated by P1 after the memory barrier
When memory coherence is required, G2 accesses that affect P2 are also performed after the memory barrier.

**Figure 5-1. Memory Barrier when Coherency is Required (M = 1)**

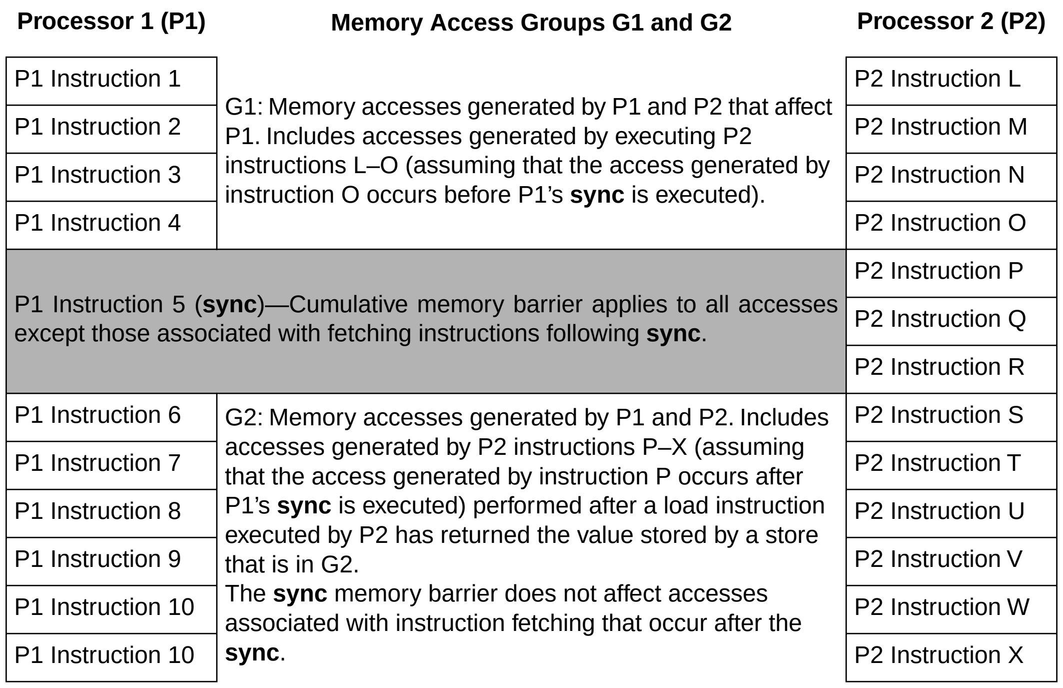
The memory barrier ensures that all memory accesses in G1 are performed with respect to any processor or mechanism, to the extent required by the associated memory coherence required attributes (that is, the memory coherence required attribute, if any, associated with each access), before any memory accesses in G2 are performed with respect to that processor or mechanism.

The ordering done by a memory barrier is said to be cumulative if it also orders memory accesses that are performed by processors and mechanisms other than P1, as follows:

— G1 includes all applicable memory accesses by any such processor or mechanism that have been performed with respect to P1 before the memory barrier is created.

— G2 includes all applicable memory accesses by any such processor or mechanism that are performed after a load instruction executed by that processor or mechanism has returned the value stored by a store that is in G2.

Figure 5-2 shows an example of a cumulative memory barrier in a two-processor system.



| Processor 1 (P1) | Memory Access Groups G1 and G2 | Processor 2 (P2) |
|---|---|---|
| P1 Instruction 1 | G1: Memory accesses generated by P1 and P2 that affect P1. Includes accesses generated by executing P2 instructions L–O (assuming that the access generated by instruction O occurs before P1's **sync** is executed). | P2 Instruction L |
| P1 Instruction 2 | | P2 Instruction M |
| P1 Instruction 3 | | P2 Instruction N |
| P1 Instruction 4 | | P2 Instruction O |
| P1 Instruction 5 (**sync**)—Cumulative memory barrier applies to all accesses except those associated with fetching instructions following **sync**. | | P2 Instruction P |
| | | P2 Instruction Q |
| | | P2 Instruction R |
| P1 Instruction 6 | G2: Memory accesses generated by P1 and P2. Includes accesses generated by P2 instructions P–X (assuming that the access generated by instruction P occurs after P1's **sync** is executed) performed after a load instruction executed by P2 has returned the value stored by a store that is in G2. The **sync** memory barrier does not affect accesses associated with instruction fetching that occur after the **sync**. | P2 Instruction S |
| P1 Instruction 7 | | P2 Instruction T |
| P1 Instruction 8 | | P2 Instruction U |
| P1 Instruction 9 | | P2 Instruction V |
| P1 Instruction 10 | | P2 Instruction W |
| P1 Instruction 10 | | P2 Instruction X |

**Figure 5-2. Cumulative Memory Barrier**

A memory barrier created by **sync** is cumulative and applies to all accesses except those associated with fetching instructions following the **sync**. See the definition of **eieio** in Chapter 8, "Instruction Set," for a description of the corresponding properties of the memory barrier created by that instruction.

### 5.2.6.1.1 Programming Considerations

Because stores cannot be performed out of program order, as described in the OEA, if a store instruction depends on the value returned by a preceding load (because the value the load returns is needed to compute either the effective address specified by the store or the value to be stored), the corresponding accesses are performed in program order. The same applies if whether the store instruction executes depends on a conditional branch that in turn depends on the value returned by a preceding load. For example, if a conditional branch depends on a preceding load and that branch chooses between a path that includes a store instruction if the condition is met, that dependent store is not performed unless and until the condition determined by the load is met.

Because instructions following an **isync** cannot execute until all instructions preceding **isync** have completed, if an **isync** follows a conditional branch instruction that depends on the value returned by a preceding load instruction, that load is performed before any loads caused by instructions following the **isync**. This is true even if the effects of the dependency are independent of the value loaded (for example, the value is compared to itself and the branch tests CR*n*[EQ]), and even if the branch target is the next sequential instruction.

Except for the cases described above and earlier in this section, data and control dependencies do not order memory accesses. Examples include the following:

- If a load specifies the same memory location as a preceding store and the location is not caching inhibited, the load may be satisfied from a store queue (a buffer into which the processor places stored values before presenting them to the memory subsystem) and not be visible to other processors and mechanisms. As a result, if a subsequent store depends on the value returned by the load, the two stores need not be performed in program order with respect to other processors and mechanisms.
- Because a store conditional instruction may complete before its store is performed, a conditional branch instruction that depends on the CR0 value set by a store conditional instruction does not order the store conditional's store with respect to memory accesses caused by instructions that follow the branch.

  For example, in the following sequence, the **stw** instruction is the **bc** instruction’s target:

  **stwcx.**
  **bc**
  **stw**

  For the **stwcx.** to complete, it must update the architected CR0 value, even though its store may not have performed. The architecture does not require that the store generated by the **stwcx.** must be performed before the store generated by the **stw**.
- Because processors may predict branch target addresses and branch condition resolution, control dependencies (for example, branches) do not order memory accesses except as described above. For example, when a subroutine returns to its caller, the return address may be predicted, with the result that loads caused by instructions at or after the return address may be performed before the load that obtains the return address is performed.

Some processors implement nonarchitected duplicates of architected resources such as GPRs, CR fields, and the LR, so resource dependencies (for example, specification of the same target register for two load instructions) do not order memory accesses.

Examples of correct uses of dependencies, **sync**, and **eieio** to order memory accesses can be found in Appendix E, “Synchronization Programming Examples.”

Because the memory model is weakly consistent, the sequential execution model as applied to instructions that cause memory accesses guarantees only that those accesses appear to be performed in program order with respect to the processor executing the instructions. For example, an instruction may complete, and subsequent instructions may be executed, before memory accesses caused by the first instruction have been performed. However, for a sequence of atomic accesses to the same memory location for which memory coherence is required, the definition of coherence guarantees that the accesses are performed in program order with respect to any processor or mechanism that accesses the location coherently, and similarly if the location is one for which caching is inhibited.

Because accesses to caching inhibited memory are performed in main memory, memory barriers and dependencies on load instructions order such accesses with respect to any processor or mechanism even if the memory is not memory coherence required.

#### 5.2.6.1.2 Programming Examples

Example 1 shows cumulative ordering of memory accesses preceding a memory barrier, and the second illustrates cumulative ordering of memory accesses following a memory barrier. Assume that locations X, Y, and Z initially contain the value 0.

Example 1:

- Processor A stores the value 1 to location X.
- Processor B loads from location X obtaining the value 1, executes a **sync**, then stores the value 2 to location Y.
- Processor C loads from location Y obtaining the value 2, executes a **sync**, then loads from location X.

Example 2:

- Processor A stores the value 1 to location X, executes a **sync**, then stores the value 2 to location Y.
- Processor B loops loading from location Y until the value 2 is obtained, then stores the value 3 to location Z.
- Processor C loads from location Z obtaining the value 3, executes a **sync**, then loads from location X. In both cases, cumulative ordering dictates that the value loaded from location X by processor C is 1.

### 5.2.6.2 Lock Acquisition and Import Barriers

An import barrier is an instruction or instruction sequence that prevents memory accesses caused by instructions following the barrier from being performed before memory accesses that acquire a lock have been performed. An import barrier can be used to ensure that a shared data structure protected by a lock is not accessed until the lock has been acquired. A **sync** instruction can always be used as an import barrier, but the approaches shown below generally yield better performance because they order only the relevant memory accesses.

#### 5.2.6.2.1 Acquire Lock and Import Shared Memory

If **lwarx** and **stwcx.** are used to obtain the lock, an import barrier can be constructed by placing an **isync** immediately following the loop containing the **lwarx** and **stwcx.**. The following example uses the Compare and Swap primitive (see Section E.2, "Synchronization Primitives") 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.

#### 5.2.6.2.2 Obtain Pointer and Import Shared Memory

If **lwarx** and **stwcx.** are used to obtain a pointer into a shared data structure, an import barrier is not needed if all the accesses to the shared data structure depend on the value obtained for the pointer. The following example uses the Fetch and Add primitive (see Section E.2, "Synchronization Primitives") to obtain and increment the pointer.

In this example it is assumed that the address of the pointer is in GPR 3, the value to be added to the pointer is in GPR 4, and the old value of the pointer is returned in GPR 5.

```
loop:   lwarx   r5,0,r3          # load pointer and reserve
        add     r0,r4,r5         # increment the pointer
        stwcx.  r0,0,r3          # try to store new value
        bne-    loop             # loop if lost reservation
        lwz     r7,data1(r5)     # load shared data
```

The load from data1 cannot be performed until the **lwarx** loads the pointer value into GPR 5. The load from data1 may be performed out-of-order before the **stwcx.**. But if the **stwcx.** fails, the branch is taken and the value returned by the load from data1 is discarded. If the **stwcx.** succeeds, the value returned by the load from data1 is valid even if the load is performed out-of-order, because the load uses the pointer value returned by the instance of the **lwarx** that created the reservation used by the successful **stwcx.**.

An **isync** could be placed between the **bne-** and the subsequent **lwz**, but no **isync** is needed if all accesses to the shared data structure depend on the value returned by the **lwarx**.

## 5.3 The Operating Environment

O The OEA defines the mechanism for controlling the memory/cache access modes introduced in Section 5.2.4.1, “Memory/Cache Access Modes.” This section describes the cache-related aspects of the OEA including the memory/cache access attributes, the **dcbi**, and speculative execution. Note that the terms ‘speculative’ and ‘out-of-order’ are used here as defined in Section 5.3.1.5.1, “Definition of Speculative and Out-of-Order Memory Accesses.”

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 **dcbst**, **dcbf**, **icbi**, **isync**, **sync**, **eieio**, **lwarx**, and **stwcx.** instructions allow the programmer to ensure a consistent memory state.

### 5.3.1 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 programmed in the PTEs and BATs by the operating system for each page and block respectively. 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 speculative (referred to the as out-of-order in the architecture specification) loading and prefetching from the addressed memory location.

Note that the memory/cache access attributes are relevant only when an effective address is translated by the processor performing the access. Note also that not all combinations of settings of these bits is 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.

The operating system programs the memory/cache access attribute for each page or block as required. The WIMG attributes occupy four bits in the BAT registers for block address translation and in the PTEs for page address translation. The WIMG bits are programmed as follows:

- The operating system uses the **mtspr** to program the WIMG bits in the BAT registers for block address translation. The IBAT register pairs implement the W or G bits; however, attempting to set either bit in IBAT registers causes boundedly-undefined results.
- The operating system writes the WIMG bits for each page into the PTEs in system memory as it sets up the page tables.

Note that for data accesses performed in real addressing mode (MSR[DR] = 0), the WIMG bits are assumed to be 0b0011 (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 0b0001 (the data is write-back, caching is enabled, memory coherency is not enforced, and memory is guarded).

#### 5.3.1.1 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**. 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.

#### 5.3.1.2 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**, or by an **eieio** when the page or block 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.

#### 5.3.1.3 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.

#### 5.3.1.4 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 that either a zero or one 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 | Meaning |
|---|---|
| 000 | The processor may cache data (or instructions).<br>A load or store operation whose target hits in the cache can use that entry in the cache.<br>The processor does not need to enforce memory coherency for accesses it initiates. |
| 001 | Data (or instructions) may be cached.<br>A load or store operation whose target hits in the cache may use that entry in the cache.<br>The processor enforces memory coherency for accesses it initiates. |
| 010 | Caching is inhibited.<br>The access is performed to memory, bypassing the cache.<br>The processor does not need to enforce memory coherency for accesses it initiates. |
| 011 | Caching is inhibited.<br>The access is performed to memory, bypassing the cache.<br>The processor enforces memory coherency for accesses it initiates. |

**Table 5-1. Combinations of W, I, and M Bits (continued)**

| WIM | Meaning |
|---|---|
| 100 | Data (or instructions) may be cached.<br>A load operation whose target hits in the cache may use that entry in the cache.<br>Store operations are written to memory. The target location of the store may be cached and is updated on a hit.<br>The processor does not need to enforce memory coherency for accesses it initiates. |
| 101 | Data (or instructions) may be cached.<br>A load operation whose target hits in the cache may use that entry in the cache.<br>Store operations are written to memory. The target location of the store may be cached and is updated on a hit.<br>The processor enforces memory coherency for accesses it initiates. |

### 5.3.1.5 The Guarded Attribute (G)

When the guarded bit is set, the memory area (block or page) is designated as guarded. This setting can be used to protect certain memory areas from read accesses made by the processor that are not dictated directly by the program. If 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 speculative (referred to as out-of-order in the architecture specification) 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 speculative load operations or prefetches from occurring to certain peripheral devices that produce undesired results when accessed in this way.

#### 5.3.1.5.1 Definition of Speculative and Out-of-Order Memory Accesses

In the architecture definition, the term ‘out-of-order’ replaced the term ‘speculative’ with respect to memory accesses to avoid a conflict between the word’s meaning in the context of execution of instructions past unresolved branches. The architecture’s use of out-of-order in this context could in turn be confused with the notion of loads and stores being reordered in a weakly ordered memory system.

To address the need for these distinctions, in the context of memory accesses this document uses the terms ‘speculative’ and ‘out-of-order’ as follows:

- Speculative memory access. An access to memory that occurs before it is known to be required by the sequential execution model.
- Out-of-order memory access. A memory access performed ahead of one that may have preceded it in the sequential model, such as is allowed by a weakly-ordered memory model.

#### 5.3.1.5.2 Performing Operations Speculatively

An operation is said to be nonspeculative if it is guaranteed to be required by the sequential execution model. Any other operation is said to be performed speculatively, which the architecture specification refers to as out of order.

Operations are performed speculatively by hardware on the expectation that the results will be needed by an instruction that will be required by the sequential execution model. Whether the results are really needed depends on anything that might divert the control flow away from the instruction, such as exceptions, branch, trap, system call, and **rfi** instructions, and anything that might change the context in which the instruction is executed.

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

Most operations can be performed speculatively, as long as the machine appears to follow the sequential execution model. Certain speculative operations are restricted, as follows.

- Stores—A store instruction may not be executed speculatively in a manner such that the alteration of the target location can be observed by other processors or mechanisms.
- Accessing guarded memory—The restrictions for this case are given in Section 5.3.1.5.4, “Speculative Accesses to Guarded Memory.”

No error of any kind other than a machine check exception may be reported due to an operation that is performed speculatively, until such time as it is known that the operation is required by the sequential execution model. The only other permitted side effects (other than machine check) of performing an operation speculatively are the following:

- Referenced and changed bits may be set as described in Section 7.6.3, “Page History Recording.”
- Nonguarded memory locations that could be fetched into a cache by nonspeculative execution may be fetched speculatively into that cache.

#### 5.3.1.5.3 Guarded Memory

Memory is said to be well behaved if the corresponding physical memory exists and is not defective, and if the effects of a single access to it are indistinguishable from the effects of multiple identical accesses to it. Data and instructions can be fetched speculatively from well-behaved memory without causing undesired side effects.

Memory is said to be guarded if either of the following cases:

- The G bit is 1 in the relevant PTE or DBAT register
- The processor is in real addressing mode (MSR[IR] = 0 or MSR[DR] = 0 for instruction fetches or data accesses respectively). In this case, 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 non-existent locations, a speculative access to such memory may cause an I/O device to perform incorrect operations or may cause a machine check.

Note that if separate store instructions access memory that is both caching-inhibited and guarded, the accesses are performed in the order specified by the program. If an aligned, load or store that is not a string or multiple access to caching-inhibited, guarded memory has accessed main memory and an external, decrementer, or imprecise-mode floating-point enabled exception is pending, the load or store is completed before the exception is taken.

#### 5.3.1.5.4 Speculative Accesses to Guarded Memory

The circumstances in which guarded memory may be accessed speculatively 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.

  Additionally, instructions are not fetched from direct-store segments (only applies when MSR[IR] = 1). If an instruction fetch is attempted from a direct-store segment, an ISI exception is generated. Note that the direct-store facility is being phased out of the architecture and will not likely be supported in future devices. Thus, software should not depend on its effects.

Note that software should ensure that only well-behaved memory is loaded into a cache, either by marking as caching-inhibited (and guarded) all memory that may not be well-behaved, or by marking such memory caching-allowed (and guarded) and referring only to cache blocks that are well-behaved.

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.

### 5.3.2 I/O Interface Considerations

The PowerPC architecture defines two mechanisms for accessing I/O:

- Memory-mapped I/O interface operations. SR[T] = 0. These 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.
- Direct-store segment operations. SR[T] = 1. These operations are considered to address the noncoherent and noncacheable direct-store segment space; therefore, hardware need not maintain coherency for these operations, and the cache is bypassed completely. Although the architecture defines this direct-store functionality, it is being phased out of the architecture and will not likely be supported in future devices. Thus, its use is discouraged, and new software should not use it or depend on its effects.

### 5.3.3 OEA Cache Management Instruction—Data Cache Block Invalidate (dcbi)

As described in Section 5.2.5, “VEA Cache Management Instructions,” the VEA defines instructions for controlling both the instruction and data caches, The OEA defines one instruction, the data cache block invalidate (**dcbi**) instruction, for controlling the data cache. This section briefly describes the cache management instruction available to programs at the supervisor privilege level. Additional descriptions of coding **dcbi** are provided in Chapter 4, “Addressing Modes and Instruction Set Summary,” and Chapter 8, “Instruction Set.” In the following description, the target is the cache block containing the byte addressed by the effective address.

Any cache management instruction that generates an EA that corresponds to a direct-store segment (SR[T] = 1) is treated as a no-op. However, note that the direct-store facility is being phased out of the architecture and will not likely be supported in future devices. Thus, software should not depend on its effects.

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—Invalidates copies of the cache block in the data caches of all processors. (Discards the modified data in the cache block.)

Target block not in cache—If copies of the target are in the data caches of other processors, **dcbi** causes those copies to be invalidated, regardless of whether the data is modified or unmodified.

- Coherency not required

  Unmodified cache block—Invalidates the cache block in the executing processor's data cache.

  Modified cache block—Invalidates the cache block in the executing processor's data cache. (Discards the modified data in the cache block.)

  Target block not in cache—No action is taken.

The processor treats **dcbi** as a store to the addressed byte with respect to address translation and protection. It is not necessary to set the referenced and changed bits.

The function of this instruction is independent of the write-through/write-back and caching-inhibited/caching-allowed attributes of the target. To ensure coherency, aliased effective addresses (two effective addresses that map to the same physical address) must have the same page offset.

# Chapter 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 that 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<br>System Reset |
| Asynchronous/maskable | External interrupt<br>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, depending on the value of MSR[IP]. Refer to Section 7.3.1.2, “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 that not all processors provide the same level of error checking.<br>The machine check exception is disabled when MSR[ME] = 0. If a machine check exception condition exists and ME is cleared, the processor goes into checkstop state.<br>If the conditions that cause the exception also corrupt the processor state such that the SRR0 and SRR1 contents 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.<br>(Note that physical address is referred to as real address in the architecture specification.) |
| DSI | 00300 | 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 and by certain memory control and cache control instructions. |
| ISI | 00400 | Occurs when an instruction fetch cannot be performed for a variety of reasons described in Section 6.4.4, “ISI Exception (0x00400).” |
| External interrupt | 00500 | 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 | May occur when the processor cannot perform a memory access for reasons described in Section 6.4.6, “Alignment Exception (0x00600).” Note that an implementation is allowed to perform the operation correctly and not cause an alignment exception. |

**Table 6-2. Exceptions and Conditions—Overview (continued)**

| Exception Type | Vector Offset (hex) | Causing Conditions |
|---|---|---|
| Program | 00700 | Caused by one of the following exception conditions, which correspond to bit settings in SRR1 and arise during execution of an instruction:<br>• Floating-point enabled exception—A floating-point enabled exception condition is generated when MSR[FE0–FE1] $\neq$ 00 and FPSCR[FEX] is set. The settings of FE0 and FE1 are described in Table 6-3.<br>• 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.”<br>• 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 instruction set is described in Chapter 4, “Addressing Modes and Instruction Set Summary.” See Section 6.4.7, “Program Exception (0x00700),” for a complete list of causes for an illegal instruction program exception.<br>• 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.<br>• Trap—A trap type program exception is generated when any of the conditions specified in a trap instruction is met.<br>• For more information, refer to Section 6.4.7, “Program Exception (0x00700).” |
| Floating-point unavailable | 00800 | 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 | 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 | Reserved for implementation-specific exceptions. |
| Reserved | 00B00 | — |
| System call | 00C00 | Occurs when a System Call (**sc**) instruction is executed. |
| Trace | 00D00 | Optional. If implemented, a trace exception occurs if either MSR[SE] = 1 and almost any instruction successfully completed or MSR[BE] = 1 and a branch instruction is completed. See Section 6.4.11, “Trace Exception (0x00D00),” for more information. |
| Floating-point assist | 00E00 | Optional. This exception can be used to provide software assistance for infrequent and complex floating-point operations such as denormalization. |
| Reserved | 00E10–00FFF | — |
| Reserved | 01000–02FFF | Reserved for implementation-specific purposes. May be used for implementation-specific exception vectors or other uses. |

## 6.1.1 Precise Exceptions

When any precise exceptions occur, SRR0 is set to point to an instruction such that all prior instructions in the instruction stream have completed execution and no subsequent instruction has begun execution. However, depending on the exception type, the instruction addressed by SRR0 may 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 report all exceptions they will cause. However, some memory accesses associated with these preceding instructions may not have been performed with respect to other processors and mechanisms.
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 and the software handler is 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 the following section.

## 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. Context-synchronizing operations include the **sc** and **rfi** instructions and most exceptions and have the following characteristics:

1. The operation causes instruction dispatching 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. Instructions that precede the operation complete execution in the context (for example, the privilege, translation mode, and memory protection) in which they were initiated.
4. If the operation either directly causes an exception (for example, the **sc** instruction causes a system call exception) or is an exception, the operation is not initiated until no exception exists having higher priority than the exception associated with the context-synchronizing operation.

A context-synchronizing operation is necessarily execution synchronizing. Unlike the **sync** instruction, these operations need not wait for memory-related operations to complete on 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** instruction is 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 that instruction. 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:

- Depending on the type of exception, 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, which are defined in the description of each exception.
- All instructions that precede the excepting instruction complete 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.

#### 6.1.2.4.1 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 that 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. Note that these exceptions are context-synchronizing if they are recoverable (MSR[RI] is copied from the MSR to SRR1 if the exception does not cause loss of state.) If the RI bit is clear (nonrecoverable), the exception is context-synchronizing only with respect to subsequent instructions.

These exceptions cannot be masked by using MSR[EE]. However, 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. 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 that MSR[RI] indicates whether enough of the machine state was saved to allow the processor to resume processing.

#### 6.1.2.4.2 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] 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 instruction that would have been executed had the exception not occurred.

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. EE is cleared automatically when an exception is taken, to delay recognition of subsequent exception conditions. 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 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 contains the address of either the instruction that caused the exception or of some instruction following that instruction.
- 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** or **isync**, 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** or **isync** instruction, SRR0 may address either the **sync** or **isync** instruction, or the following instruction.
- If the imprecise exception is not forced by either the context- or execution-synchronizing mechanism, the instruction addressed by SRR0 is considered not to have begun execution if it did not cause 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, MSR[FE0,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. 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 excepting instruction or the data that caused the exception. Results from the excepting instruction may have been used by or affected subsequent instructions executed before the exception handler was invoked. |
| 1 | 0 | Floating-point imprecise recoverable. When an 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 excepting instruction and correct any faulty results. In this mode, no incorrect results caused by the excepting instruction have been used by or affected subsequent instructions that are executed before the exception handler is invoked. |
| 1 | 1 | Floating-point precise mode |

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.**), 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 unchanged.

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.

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. |

**Table 6-4. Exception Priorities (continued)**

| Exception Class | Priority | 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.<br>Note that a single instruction can cause multiple exceptions. When this occurs, those exceptions are ordered in priority as indicated in the following:<br>A. Integer loads and stores<br>a. Alignment<br>b. DSI<br>c. Trace (if implemented)<br>B. Floating-point loads and stores<br>a. Floating-point unavailable<br>b. Alignment<br>c. DSI<br>d. Trace (if implemented)<br>C. Other floating-point instructions<br>a. Floating-point unavailable<br>b. Program—Precise-mode floating-point enabled exception<br>c. Floating-point assist (if implemented)<br>d. Trace (if implemented)<br>D.**rfi** and **mtmsr**<br>a. Program—Privileged Instruction<br>b. Program—Precise-mode floating-point enabled exception<br>c. Trace (if implemented), for **mtmsr** only<br>If precise-mode IEEE floating-point enabled exceptions are enabled and FPSCR[FEX] is set, a program exception occurs no later than the next synchronizing event.<br>E. Other instructions<br>a. These exceptions are mutually exclusive and have the same priority:<br>—Program: Trap<br>—System call (**sc**)<br>—Program: Privileged Instruction<br>—Program: Illegal Instruction<br>b. Trace (if implemented)<br>F. ISI exception<br>The ISI 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. |
| 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] $\neq$ MSR[FE1]. For more information see, Section 6.1.3, “Imprecise Exceptions.” |

**Table 6-4. Exception Priorities (continued)**

| Exception Class | Priority | Exception |
|---|---|---|
| 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. |
| | 6 | 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 (system reset or machine check exceptions) can occur anytime; they 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 may 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

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 begun 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.

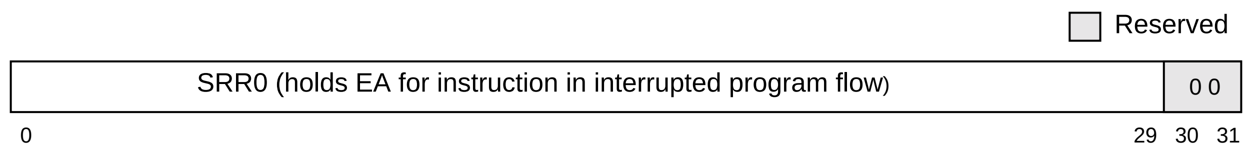


Figure 6-1. Machine Status Save/Restore Register 0

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 **rfi** is executed. SRR1 is shown in Table 6-2.

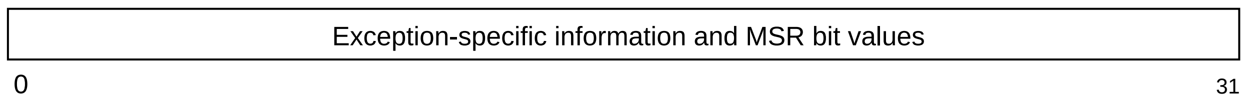


Figure 6-2. Machine Status Save/Restore Register 1

When an exception occurs, SRR1 bits 1–4 and 10–15 are loaded with exception-specific information and MSR bits 16–23, 25–27, and 30–31 are placed into the corresponding bit positions of SRR1. Depending on the implementation, additional MSR bits may be copied to SRR1.

In some implementations, every instruction fetch when MSR[IR] = 1 and every data access requiring address translation when MSR[DR] = 1, may modify SRR0 and SRR1.

The MSR is 32 bits wide as shown in Figure 6-3. Note that the 32-bit implementation of the MSR is comprised of the 32 least-significant bits of the 64-bit MSR.

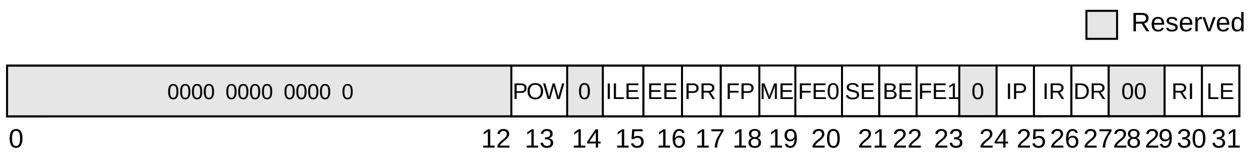


Figure 6-3. Machine State Register (MSR)

Table 6-6 shows the bit definitions for the MSR

Table 6-6. MSR Bit Settings

| Bits | Name | Description |
|---|---|---|
| 0–12 | — | Reserved |
| 13 | POW | Power management enable<br>0 Power management disabled (normal operation mode).<br>1 Power management enabled (reduced power mode).<br>**Note**: Power management functions are implementation-dependent. If the function is not implemented, this bit is treated as reserved. |
| 14 | — | Reserved |

Table 6-6. MSR Bit Settings (continued)

| Bits | Name | Description |
|---|---|---|
| 15 | ILE | Exception little-endian mode. When an exception occurs, this bit is copied into MSR[LE] to select the endian mode for the context established by the exception. |
| 16 | EE | External interrupt enable<br>0 While the bit is cleared the processor delays recognition of external interrupts and decrementer exception conditions.<br>1 The processor is enabled to take an external interrupt or the decrementer exception. |
| 17 | PR | Privilege level<br>0 The processor can execute both user- and supervisor-level instructions.<br>1 The processor can only execute user-level instructions. |
| 18 | FP | Floating-point available<br>0 The processor prevents dispatch of floating-point instructions, including floating-point loads, stores, and moves.<br>1 The processor can execute floating-point instructions. |
| 19 | ME | Machine check enable<br>0 Machine check exceptions are disabled.<br>1 Machine check exceptions are enabled. |
| 20 | FE0 | Floating-point exception mode 0 (see Table 2-10). |
| 21 | SE | Single-step trace enable (Optional)<br>0 The processor executes instructions normally.<br>1 The processor generates a single-step trace exception upon the successful execution of the next instruction.<br>Note: If the function is not implemented, this bit is treated as reserved. |
| 22 | BE | Branch trace enable (Optional)<br>0 The processor executes branch instructions normally.<br>1 The processor generates a branch trace exception after completing the execution of a branch instruction, regardless of whether or not the branch was taken.<br>Note: If the function is not implemented, this bit is treated as reserved. |
| 23 | FE1 | Floating-point exception mode 1 (See Table 2-10). |
| 24 | — | Reserved |
| 25 | IP | Exception prefix. The setting of this bit specifies whether an exception vector offset is prepended with Fs or 0s. In the following description, *nnnnn* is the offset of the exception vector. See Table 6-2.<br>0 Exceptions are vectored to the physical address 0x000*n_nnnn* .<br>1 Exceptions are vectored to the physical address 0xFFF*n_nnnn*.<br>In most systems, IP is set to 1 during system initialization, and then cleared to 0 when initialization is complete. |
| 26 | IR | Instruction address translation<br>0 Instruction address translation is disabled.<br>1 Instruction address translation is enabled.<br>For more information see Chapter 7, “Memory Management.” |
| 27 | DR | Data address translation<br>0 Data address translation is disabled.<br>1 Data address translation is enabled.<br>For more information see Chapter 7, “Memory Management.” |
| 28–29 | — | Reserved |

**Table 6-6. MSR Bit Settings (continued)**

| Bits | Name | Description |
|---|---|---|
| 30 | RI | Recoverable exception (for system reset and machine check exceptions).<br>0 Exception is not recoverable.<br>1 Exception is recoverable.<br>For more information see Section 6.4.1, "System Reset Exception (0x00100),"and Section 6.4.2, "Machine Check Exception (0x00200)." |
| 31 | LE | Little-endian mode enable<br>0 The processor runs in big-endian mode.<br>1 The processor runs in little-endian mode. |

Those MSR bits that are written to SRR1 are written when the first instruction of the exception handler is encountered. The data address register (DAR) is 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 MSR[EE]. 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 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.
2. SRR1 bits 1–4 and 10–15 are loaded with exception-specific information.
3. MSR bits 16–23, 25–27, and 30–31 are loaded with a copy of the corresponding MSR bits. Note that some implementations save additional MSR bits to SRR1.

4. The MSR is set as described in Table 6-7. 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 that 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.

   Also, note that the MSR[ILE] setting at the time of the exception is copied to MSR[LE] when the exception is taken (as shown in Table 6-7).

5. Instruction fetch and execution resumes, using the new MSR value, at a location specific to the exception type. The location is determined by adding the exception's vector offset (see Table 6-2) to the base address determined by MSR[IP]. If IP is cleared, exceptions are vectored to the physical address 0x000*n_nnnn*. If IP is set, exceptions are vectored to the physical address 0xFFF*n_nnnn*. 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 may cause SRR0 and SRR1 to be modified.

### 6.2.3 Returning from an Exception Handler

The Return from Interrupt (**rfi**) instruction performs context synchronization by allowing previously issued instructions to complete before returning to the interrupted process. Execution of the **rfi** 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 **rfi** instruction copies SRR1 bits back into the MSR.
- Subsequent instructions 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** 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.** instruction clears any outstanding reservations, which ensures that an **lwarx** instruction in the old process is not paired with an **stwcx.** instruction in the new process.

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 is 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 SRR0 and SRR1 appropriately, and then execute **rfi**.

Note that 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 6-8. shows all exception types and certain MSR bit settings when the exception handler is invoked. Depending on the exception, some of these bits are stored in SRR1 when an exception is taken. The following subsections describe each exception in detail.

**Table 6-8. MSR Setting Due to Exception**

| Exception Type | MSR Bit | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | POW | ILE | EE | PR | FP | ME | FE0 | SE | BE | FE1 | IP | IR | DR | RI | LE |
| System reset | 0 | — | 0 | 0 | 0 | — | 0 | 0 | 0 | 0 | — | 0 | 0 | 0 | ILE |
| Machine check | 0 | — | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | — | 0 | 0 | 0 | ILE |
| Data access | 0 | — | 0 | 0 | 0 | — | 0 | 0 | 0 | 0 | — | 0 | 0 | 0 | ILE |
| Instruction access | 0 | — | 0 | 0 | 0 | — | 0 | 0 | 0 | 0 | — | 0 | 0 | 0 | ILE |
| External | 0 | — | 0 | 0 | 0 | — | 0 | 0 | 0 | 0 | — | 0 | 0 | 0 | ILE |
| Alignment | 0 | — | 0 | 0 | 0 | — | 0 | 0 | 0 | 0 | — | 0 | 0 | 0 | ILE |
| Program | 0 | — | 0 | 0 | 0 | — | 0 | 0 | 0 | 0 | — | 0 | 0 | 0 | ILE |
| Floating-point unavailable | 0 | — | 0 | 0 | 0 | — | 0 | 0 | 0 | 0 | — | 0 | 0 | 0 | ILE |
| Decrementer | 0 | — | 0 | 0 | 0 | — | 0 | 0 | 0 | 0 | — | 0 | 0 | 0 | ILE |
| System call | 0 | — | 0 | 0 | 0 | — | 0 | 0 | 0 | 0 | — | 0 | 0 | 0 | ILE |

**Table 6-8. MSR Setting Due to Exception (continued)**

| Exception Type | MSR Bit | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | POW | ILE | EE | PR | FP | ME | FE0 | SE | BE | FE1 | IP | IR | DR | RI | LE |
| Trace exception | 0 | — | 0 | 0 | 0 | — | 0 | 0 | 0 | 0 | — | 0 | 0 | 0 | ILE |
| Floating-point assist exception | 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.

## 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 6-9.

**Table 6-9. 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 | 1–4 Cleared<br>10–15 Cleared<br>16–23 Loaded with equivalent bits from the MSR<br>25–27 Loaded with equivalent bits from the MSR<br>30 Loaded from the equivalent MSR bit, MSR[RI], if the exception is recoverable; otherwise cleared.<br>31 Loaded with equivalent bit from the MSR<br><br>Note that depending on the implementation, additional MSR bits may be copied to SRR1.<br>If the processor state is corrupted to the extent that execution cannot resume reliably, the bit corresponding to MSR[RI], (SRR1[30]), is cleared. |
| MSR | POW 0<br>ILE —<br>EE 0<br>PR 0<br>FP 0<br>ME —<br>FE0 0<br>SE 0<br>BE 0<br>FE1 0<br>IP —<br>IR 0<br>DR 0<br>RI 0<br>LE Set to value of ILE |

When a system reset exception is taken, instruction execution continues at offset 0x00100 from the physical base address determined by MSR[IP].

If the exception is recoverable, the value of MSR[RI] is copied to the corresponding SRR1 bit. The exception functions as a context-synchronizing operation. The exception is not recoverable if a reset exception causes the loss 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. Note that 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 that 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 that 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 that 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** or **dcba** 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 6-10.

**Table 6-10. 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 30 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 | POW | 0 | FP | 0 | BE | 0 | DR | 0 |
| | ILE | — | ME [1] | — | FE1 | 0 | RI | 0 |
| | EE | 0 | FE0 | 0 | IP | — | LE | Set to value of ILE |
| | PR | 0 | SE | 0 | IR | 0 | | |

[*] Note that 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 cannot resume in the same context that existed before the exception.

When a machine check exception is taken, instruction execution resumes at offset 0x00200 from the physical base address determined by MSR[IP].

## 6.4.3 DSI Exception (0x00300)

A DSI 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) that can be read by using the **mfspr** instruction. Bit settings are provided in Table 6-11. Table 6-11 also 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 translation, for example from a storage device such as a hard disk drive.
- The instruction is not supported for the type of memory addressed.
  - For **lwarx**/**stwcx.** 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 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[0–28] = DABR[DAB]
  - MSR[DR] = DABR[BT]
  - The instruction is a store and DABR[DW] = 1, or it is a load and DABR[DR] = 1.

  The DABR is described in Section 2.3.14, "Data Address Breakpoint Register (DABR)." DAR settings are described in Table 6-11. 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.** instruction has an EA for which a normal store operation would cause a DSI exception but the processor does not have the reservation from **lwarx**, whether a DSI exception is taken is 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 6-11, this exception also sets the data address register (DAR).

**Table 6-11. DSI Exception—Register Settings**

| Register | Setting Description | |
|---|---|---|
| SRR0 | Set to the effective address of the instruction that caused the exception. | |
| SRR1 | 1–4 | Cleared |
| | 10–15 | Cleared |
| | 16–23 | Loaded with equivalent bits from the MSR |
| | 25–27 | Loaded with equivalent bits from the MSR |
| | 30–31 | Loaded with equivalent bits from the MSR |
| | Note that depending on the implementation, additional MSR bits may be copied to SRR1. | |

**Table 6-11. DSI Exception—Register Settings (continued)**

| Register | Setting Description |
|---|---|
| MSR | POW 0 · ILE — · EE 0 · PR 0 · FP 0 · ME — · FE0 0 · SE 0 · BE 0 · FE1 0 · IP — · IR 0 · DR 0 · RI 0 · LE Set to value of ILE |
| DSISR | 0 Set if a load or store instruction results in a direct-store error exception; otherwise cleared.<br>1 Set if the translation of an attempted access is not found in the primary hash table entry group (HTEG), or in the rehashed secondary HTEG, or in the range of a DBAT register (page fault condition); otherwise cleared.<br>2–3 Cleared<br>4 Set if a memory access is not permitted by the page or DBAT protection mechanism; otherwise cleared.<br>5 Set if the **eciwx**, **ecowx**, **lwarx,** or **stwcx.** instruction is attempted to direct-store interface space, or if the **lwarx** or **stwcx** instruction is used with addresses that are marked as write-through. Otherwise cleared.<br>6 Set for a store operation and cleared for a load operation.<br>7–8 Cleared<br>9 Set if a DABR match occurs. Otherwise cleared.<br>10 Cleared<br>11 Set if the instruction is an **eciwx** or **ecowx** and EAR[E] = 0; otherwise cleared.<br>12–31Cleared<br>Due to the multiple exception conditions possible from the execution of a single instruction, the following combinations of bits of DSISR may be set concurrently:<br>• Bits 1 and 11<br>• Bits 4 and 5<br>• Bits 4 and 11<br>• Bits 5 and 11<br>Additonally, bit 6 is set if the instruction that caused the exception is a store, **ecowx**, **dcbz**, **dcba**, or **dcbi** and bit 6 would otherwise be cleared. Also, bit 9 (DABR match) may be set alone, or in combination with any other bit, or with any of the other combinations shown above. |
| DAR | Set to the effective address of a memory element as described in the following list:<br>• A byte in the first word accessed in the segment or BAT area that caused the DSI exception, for a byte, half word, or word memory access (to a segment or BAT area).<br>• A byte in the first double word accessed in the segment or BAT area that caused the DSI exception, for a double-word memory access (to a segment or BAT area).<br>• A byte in the block that caused the exception for a cache management instruction.<br>• The EA computed by the instruction for the attempted execution of an **eciwx** or **ecowx** instruction when EAR[E] is cleared.<br>If the exception is caused by a DABR match, the DAR is set to the effective address of any byte in the range from A to B inclusive, where A is the effective address of the word (for a byte, half word,or word access) or double word (for a double word access) specified by the EA computed by the instruction, and B is the EA of the last byte in the word or double word in which the match occurred. |

When a DSI exception is taken, instruction execution resumes at offset 0x00300 from the physical base address determined by MSR[IP].

### 6.4.4 ISI Exception (0x00400)

An 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:

- The effective address cannot be translated. For example, when there is a page fault for this portion of the translation, an ISI exception must be taken to retrieve the page (and possibly the translation), typically from a storage device.
- An attempt is made to fetch an instruction from a no-execute segment.
- An attempt is made to fetch an instruction from guarded memory and MSR[IR] = 1.
- The fetch access violates memory protection.

Register settings for ISI exceptions are shown in Table 6-12.

**Table 6-12. 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 | 1 | Set if the translation of an attempted access is not found in the primary hash table entry group (HTEG), or in the rehashed secondary HTEG, or in the range of an IBAT register (page fault condition); otherwise cleared. | | | | | | |
| | 2 | Cleared | | | | | | |
| | 3 | Set if the fetch access occurs to a direct-store segment (SR[T] = 1), to a no-execute segment (N bit set in segment descriptor), or to guarded memory when MSR[IR] = 1. Otherwise, cleared. Note that the direct-store facility is being phased out of the architecture and is not likely to be supported in future devices. | | | | | | |
| | 4 | Set if a memory access is not permitted by the page or IBAT protection mechanism, described in Chapter 7, “Memory Management”; otherwise cleared. | | | | | | |
| | 10–15 | Cleared | | | | | | |
| | 16–23 | Loaded with equivalent bits from the MSR | | | | | | |
| | 25–27 | Loaded with equivalent bits from the MSR | | | | | | |
| | 30–31 | Loaded with equivalent bits from the MSR | | | | | | |
| | Note that only one of bits 1, 3, and 4 can be set . Also, note that depending on the implementation, additional MSR bits may be copied to SRR1. | | | | | | | |
| MSR | POW | 0 | FP | 0 | BE | 0 | DR | 0 |
| | ILE | — | ME | — | FE1 | 0 | RI | 0 |
| | EE | 0 | FE0 | 0 | IP | — | LE | Set to value of ILE |
| | PR | 0 | SE | 0 | IR | 0 | | |

When an ISI exception is taken, instruction execution resumes at offset 0x00400 from the physical base address determined by MSR[IP].

## 6.4.5 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 MSR[EE] is zero when the exception is detected. Note that the occurrance 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 | 1–4 Cleared<br>10–15 Cleared<br>16–23 Loaded with equivalent bits from the MSR<br>25–27 Loaded with equivalent bits from the MSR<br>30–31 Loaded with equivalent bits from the MSR<br><br>Note that depending on the implementation, additional MSR bits may be copied to SRR1. |
| MSR | POW 0 FP 0 BE 0 DR 0<br>ILE — ME — FE1 0 RI 0<br>EE 0 FE0 0 IP — LE Set to value of ILE<br>PR 0 SE 0 IR 0 |

When an external interrupt exception is taken, instruction execution resumes at offset 0x00500 from the physical base address determined by MSR[IP].

## 6.4.6 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:

- An operand of a floating-point load or store instruction is not word-aligned.
- The operand of **lmw**, **stmw**, **lwarx**, **stwcx.**, **eciwx**, or **ecowx** is not aligned.
- The instruction is **lmw**, **stmw**, **lswi**, **lswx**, **stswi**, or **stswx** and the processor is in little-endian mode.
- An operand of an elementary or string load or store crosses a protection boundary.
- An operand of **lmw** or **stmw** crosses a segment or BAT boundary.
- An operand of **dcbz** is in memory that is write-through-required or caching inhibited, or **dcbz** is executed in an implementation that has either no data cache or a write-through data cache.

For **lmw**, **stmw**, **lswi**, **lswx**, **stswi**, and **stswx** instructions in little-endian mode, an alignment exception always occurs. For **lmw** and **stmw** instructions with an operand that is not aligned in big-endian mode, and for **lwarx**, **stwcx.**, **eciwx**, and **ecowx** with an operand that is not aligned in either endian mode, an implementation may yield boundedly-undefined results instead of causing an alignment exception (for **eciwx** and **ecowx** when EAR[E] = 0, a third alternative is to cause a DSI exception). For all other cases listed above, an implementation may execute the instruction correctly instead of causing an alignment exception. For the **dcbz** instruction, correct execution means clearing each byte of the block in main memory. See Section 3.1, “Data Organization in Memory and Data Transfers,” for a complete definition of alignment in the PowerPC architecture.

The term, ‘protection boundary’, refers to the boundary between protection domains. A protection domain is a segment, a block of memory defined by a BAT entry, a virtual 4-Kbyte page, 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 | 1–4 Cleared<br>10–15 Cleared<br>16–23 Loaded with equivalent bits from the MSR<br>25–27 Loaded with equivalent bits from the MSR<br>30–31 Loaded with equivalent bits from the MSR<br><br>Note that depending on the implementation, additional MSR bits may be copied to SRR1. |
| MSR | POW 0   FP 0   BE 0   DR 0<br>ILE —   ME —   FE1 0   RI 0<br>EE 0   FE0 0   IP —   LE Set to value of ILE<br>PR 0   SE 0   IR 0 |

**Table 6-14. Alignment Exception—Register Settings (continued)**

| Register | Setting Description |
|---|---|
| DSISR | 0–14 Cleared<br>15–16 For instructions that use register indirect with index addressing—set to bits 29–30 of the instruction encoding.<br>For instructions that use register indirect with immediate index addressing—cleared<br>17 For instructions that use register indirect with index addressing—set to bit 25 of the instruction encoding.<br>For instructions that use register indirect with immediate index addressing— set to bit 5 of the instruction encoding.<br>18–21 For instructions that use register indirect with index addressing—set to bits 21–24 of the instruction encoding.<br>For instructions that use register indirect with immediate index addressing—set to bits 1–4 of the instruction encoding.<br>22–26 Set to bits 6–10 (identifying either the source or destination) of the instruction encoding. Undefined for **dcbz**.<br>27–31 Set to bits 11–15 of the instruction encoding (**r**A) for update-form instructions<br>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.<br>Note that 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:<br>000000000000 00 0 01 0 0101 ttttt ?????<br>The value ttttt refers to the destination and ????? indicates undefined bits.<br>However, this register may be set as if the instruction were **lwa**, as follows:<br>000000000000 10 0 00 0 1101 ttttt ?????<br>If there is no corresponding instruction, no alternative value can be specified.<br>The instruction pairs that can use the same DSISR values are as follows:<br>**lbz/lbzxlbzu/lbzuxlhz/lhzxlhzu/lhzuxlha/lhaxlhau/lhaux lwz/lwzxlwzu/lwzuxlwa/lwaxstb/stbx stbu/stbuxsth/sthxsthu/sthuxstw/stwxstwu/stwuxlfs/lfsxlfsu/lfsuxstfs/stfsxstfsu/stfsux** |
| DAR | Set to the EA of the data access as computed by the instruction causing the alignment exception. |

The architecture does not support the use of a misaligned EA by load/store with reservation instructions or by the **eciwx** and **ecowx** instructions. If one of these instructions specifies a misaligned EA, the exception handler should not emulate the instruction but should treat the occurrence as a programming error.

### 6.4.6.1 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.

#### 6.4.6.1.1 Page Address Translation Access Considerations

A page address translation access occurs when MSR[DR] is set, SR[T] is cleared, and there is no BAT match. Note that a **dcbz** instruction causes an alignment exception if the access is to a page or block with the W (write-through) or I (cache-inhibit) 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.

### 6.4.6.2 Little-Endian Mode Alignment Exceptions

The OEA allows implementations to take alignment exceptions on misaligned accesses (as described in Section 3.1.4, “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.

### 6.4.6.3 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 (half word or word)
- 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 DSISR bits 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.

**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 | — |
| 00 0 0010 | — | 01 1 0101 | **lwaux** |
| 00 0 0010 | **stw** | 10 0 0010 | **stwcx.** |
| 00 0 0100 | **lhz** | 10 0 0011 | — |
| 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 | — | 10 1 0100 | **eciwx** |
| 00 0 1010 | **stfs** | 10 1 0110 | **ecowx** |
| 00 0 1011 | — | 10 1 1111 | **dcbz** |
| 00 0 1101 | **ld**,**lwa** [2] | 11 0 0000 | **lwzx** |
| 00 0 1111 | **std** | 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 | — |
| 00 1 0111 | **stmw** | 11 0 1010 | **stfsx** |
| 00 1 1000 | **lfsu** | 11 0 1011 | — |
| 00 1 1001 | — | 11 0 1111 | **stfiwx** |
| 00 1 1010 | **stfsu** | 11 1 0000 | **lwzux** |
| 00 1 1011 | — | 11 1 0010 | **stwux** |
| 01 0 0000 | — | 11 1 0100 | **lhzux** |
| 01 0 0010 | — | 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 | — |
| 01 0 1010 | **stswx** | 11 1 1010 | **stfsux** |
| 01 0 1011 | **stswi** | 11 1 1011 | — |

**Table 6-15. DSISR(15–21) Settings to Determine Misaligned Instruction (continued)**

| DSISR[15–21] | Instruction | DSISR[15–21] | Instruction |
|---|---|---|---|
| 01 1 0000 | — | — | — |

[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 indistiguishable to the exception handler from **lwarx** and **lwz**.

[2]These instructions are distinguished by DSISR[12–13], which are not shown in this table.

### 6.4.7 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) MSR[FE0] or MSR[FE1] 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.

  Note that 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)
  - A move to/from SPR instruction with an SPR field that does not contain one of the defined values
    - MSR[PR] = 1 and spr[0] = 1 (this can cause a privileged instruction program exception)
    - MSR[PR] = 0 or spr[0] = 0 (may cause boundedly-undefined results.)

  — An unimplemented floating-point instruction that is not optional (may cause a floating-point assist exception)

- 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:<br>• For all program exceptions except floating-point enabled exceptions when operating in imprecise mode (MSR[FE0] ≠ MSR[FE1]), SRR0 contains the EA of the excepting instruction.<br>• When the processor is in floating-point imprecise mode, SRR0 may contain the EA of the excepting instruction or that of a subsequent unexecuted instruction. If the subsequent instruction is **sync** or **isync**, SRR0 points no more than four bytes beyond the **sync** or **isync** instruction.<br>• 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 | 1–4 Cleared<br>10 Cleared<br>11 Set for an IEEE floating-point enabled program exception; otherwise cleared.<br>12 Set for an illegal instruction program exception; otherwise cleared.<br>13 Set for a privileged instruction program exception; otherwise cleared.<br>14 Set for a trap program exception; otherwise cleared.<br>15 Cleared if SRR0 contains the address of the instruction causing the exception, and set if SRR0 contains the address of a subsequent instruction.<br>16–23 Loaded with equivalent bits from the MSR<br>25–27 Loaded with equivalent bits from the MSR<br>30–31 Loaded with equivalent bits from the MSR<br><br>Note that depending on the implementation, additional MSR bits may be copied to SRR1. | | | |
| MSR | POW 0<br>ILE —<br>EE 0<br>PR 0 | FP 0<br>ME —<br>FE0 0<br>SE 0 | BE 0<br>FE1 0<br>IP —<br>IR 0 | DR 0<br>RI 0<br>LE Set to value of ILE |

When a program exception is taken, instruction execution resumes at offset 0x00700 from the physical base address determined by MSR[IP].

## 6.4.8 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 | 1–4 | Cleared | | |
| | 10–15 | Cleared | | |
| | 16–23 | Loaded with equivalent bits from the MSR | | |
| | 25–27 | Loaded with equivalent bits from the MSR | | |
| | 30–31 | Loaded with equivalent bits from the MSR | | |
| | Note that depending on the implementation, additional MSR bits may be copied to SRR1. | | | |
| MSR | POW 0 | FP 0 | BE 0 | DR 0 |
| | ILE — | ME — | FE1 0 | RI 0 |
| | EE 0 | FE0 0 | IP — | LE Set to value of ILE |
| | PR 0 | SE 0 | IR 0 | |

When a floating-point unavailable exception is taken, instruction execution resumes at offset 0x00800 from the physical base address determined by MSR[IP].

## 6.4.9 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 | 1–4 Cleared<br>10–15 Cleared<br>16–23 Loaded with equivalent bits from the MSR<br>25–27 Loaded with equivalent bits from the MSR<br>30–31 Loaded with equivalent bits from the MSR<br><br>Note that depending on the implementation, additional MSR bits may be copied to SRR1. |
| MSR | POW 0 · ILE — · EE 0 · PR 0<br>FP 0 · ME — · FE0 0 · SE 0<br>BE 0 · FE1 0 · IP — · IR 0<br>DR 0 · RI 0 · LE Set to value of ILE |

When a decrementer exception is taken, instruction execution resumes at offset 0x00900 from the physical base address determined by MSR[IP].

## 6.4.10 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.. Then a system call exception is generated.

The system call exception causes the next instruction to be fetched from offset 0x00C00 from the physical base address determined by the new setting of MSR[IP]. As with most other exceptions, this exception is context-synchronizing. Refer to Section 6.1.2.1, “Context Synchronization,” 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 | 1–4 Cleared<br>10–15 Cleared<br>16–23 Loaded with equivalent bits from the MSR<br>25–27 Loaded with equivalent bits from the MSR<br>30–31 Loaded with equivalent bits from the MSR<br><br>Note that depending on the implementation, additional MSR bits may be copied to SRR1. |

**Table 6-19. System Call Exception—Register Settings (continued)**

| Register | Setting Description | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| MSR | POW | 0 | FP | 0 | BE | 0 | DR | 0 |
| | ILE | — | ME | — | FE1 | 0 | RI | 0 |
| | EE | 0 | FE0 | 0 | IP | — | LE | Set to value of ILE |
| | PR | 0 | SE | 0 | IR | 0 | | |

When a system call exception is taken, instruction execution resumes at offset 0x00C00 from the physical base address determined by MSR[IP].

## 6.4.11 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[SE,BE] as follows:

- MSR[SE] = 1: the processor generates a single-step type trace exception after each instruction that completes without causing an exception or context change (such as occurs when an **sc**, **rfi**, or a load instruction that causes an exception, for example, is executed).
- 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:

- **rfi** 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 **rfi** 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 | 1–4 Cleared<br>10–15 Cleared<br>16–23 Loaded with equivalent bits from the MSR<br>25–27 Loaded with equivalent bits from the MSR<br>30–31 Loaded with equivalent bits from the MSR<br><br>Note that depending on the implementation, additional MSR bits may be copied to SRR1. |
| MSR | POW 0   FP 0   BE 0   DR 0<br>ILE —   ME —   FE1 0   RI 0<br>EE 0   FE0 0   IP —   LE Set to value of ILE<br>PR 0   SE 0   IR 0 |

When a trace exception is taken, instruction execution resumes at offset 0x00D00 from the base address determined by MSR[IP].

## 6.4.12 Floating-Point Assist Exception (0x00E00)

The floating-point assist exception is optional to the PowerPC architecture. It can be used to allow software to assist in the following situations:

- Execution of floating-point instructions for which an implementation uses software routines to perform certain operations, such as those involving denormalization.
- Execution of floating-point instructions that are not optional and are not implemented in hardware. In this case, the processor may generate an illegal instruction type program exception instead.

Register settings for the floating-point assist exceptions are described in Table 6-21.

**Table 6-21. Floating-Point Assist Exception—Register Settings**

| Register | Setting Description |
|---|---|
| SRR0 | Set to the address of the next instruction to be executed in the program for which the floating-point assist exception was generated. |
| SRR1 | 1–4 Implementation-specific information<br>10–15 Implementation-specific information<br>16–23 Loaded with equivalent bits from the MSR<br>25–27 Loaded with equivalent bits from the MSR<br>30–31 Loaded with equivalent bits from the MSR<br><br>Note that depending on the implementation, additional MSR bits may be copied to SRR1. |
| MSR | POW 0   FP 0   BE 0   DR 0<br>ILE —   ME —   FE1 0   RI 0<br>EE 0   FE0 0   IP —   LE Set to value of ILE<br>PR 0   SE 0   IR 0 |

When a floating-point assist exception is taken, instruction execution resumes as offset 0x00E00 from the base address determined by MSR[IP].

# Chapter 7
# Memory Management

This chapter gives operating system designers an understanding of the memory management model, which defines how addresses are translated and how memory is protected.

## 7.1 Overview

Two general types of processor-generated memory accesses require address translation—instruction accesses and data accesses 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 the segment descriptors and page tables used to locate the effective-to-physical address mapping for memory 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 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.

The segment information, used to generate the interim virtual addresses, is stored as segment descriptors. These descriptors may reside in on-chip segment registers (32-bit implementations) or as segment table entries (STEs) in memory (64-bit implementations). 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.

The block address translation (BAT) mechanism is a software-controlled array that stores the available block address translations on-chip. BAT array entries are implemented as pairs of BAT registers that are accessible as supervisor special-purpose registers (SPRs).

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. (Note that the architecture specification refers to exceptions as interrupts.)

## 7.2 MMU Features

The MMU of a 32-bit processor provides 4 Gbytes of effective address space, a 52-bit interim virtual address, and physical addresses that are $\leq$ 32 bits in length. Note that 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 and are invisible to the programming model; therefore, they are not described in this document. 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.

## 7.3 MMU Overview

The 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 are 32 bits wide. In the address translation process, the processor converts an effective address to a 52-bit virtual address, 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.

Note that in the cases that implementations support a physical address range that is smaller than 32 bits, the high-order bits of the effective address may be ignored in the address translation process. The remainder of this chapter assumes that implementations support the maximum physical address range.

The operating system manages the system’s physical memory resources. Consequently, the operating system initializes the MMU registers (segment registers, BAT registers, and SDR1 register) and sets up page 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 or into other large regions called blocks (128 Kbyte–256 Mbyte). Segments that correspond to memory-mapped areas can be further subdivided into 4-Kbyte pages. For each block or page, the operating system creates an address descriptor (page table entry (PTE) or BAT array entry); 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 block or page is accessed. Address descriptors for pages reside in tables (as PTEs) in physical memory; for faster accesses, the MMU often caches on-chip copies of recently-used PTEs in an on-chip TLB. The MMU keeps the block information on-chip in the BAT array (comprised of the BAT registers).

This section is an overview of the high-level organization and operational concepts of the MMU and a summary of all MMU control registers. For more information, see Section 2.3.1, “Machine State Register (MSR).” Section 7.5.3, “BAT Register Implementation of BAT Array,” Section 7.6.2.1, “Segment Descriptor Definitions,” and Section 7.7.1.1, “SDR1 Register Definitions.”

## 7.3.1 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 address according to the procedures described throughout this chapter. The memory subsystem uses the physical address for the access.

### 7.3.1.1 Effective Addresses in 32-Bit Mode

In addition to the 64-and 32-bit memory management models defined by the OEA, the 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 EA 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 MSR[IR] and MSR[DR], which enable instruction and data address translation. The truncation of the EA is the only way in which memory accesses are affected by the 32-bit mode of operation.See Section 4.1.4.2, “Effective Address Calculation.”

### 7.3.1.2 Predefined Physical Memory Locations

There are four areas of the physical memory map that have predefined uses. The first 256 bytes of physical memory (or if MSR[IP] = 1, the first 256 bytes of memory located at physical address 0xFFF0_0000) are assigned for arbitrary use by the operating system. The rest of that first page of physical memory defined by the vector base address (determined by MSR[IP]) is either used for exception vectors, or reserved for future exception vectors. The third predefined area of memory consists of the second and third physical pages of the memory map, which are used for implementation-specific purposes. In some implementations, the second and third pages located at physical address 0xFFF0_1000when MSR[IP] = 1 are also used for implementation-specific purposes. Fourthly, the system software defines the locations in physical memory that contain the page address translation tables. These predefined memory areas are summarized in Table 7-1. in terms of the variable 'Base'.

**Table 7-1. Predefined Physical Memory Locations**

| Memory Area | Physical Address Range | Predefined Use |
|---|---|---|
| 1 | Base \|\| 0x0_0000–Base \|\| 0x0_00FF | Operating system |
| 2 | Base \|\| 0x0_0100–Base \|\| 0x0_0FFF | Exception vectors |
| 3 | Base \|\| 0x0_1000–Base \|\| 0x0_2FFF | Implementation-specific [1] |
| 4 | Software-specified—contiguous sequence of physical pages | Page table |

[1] Only valid for MSR[IP] = 1 on some implementations

Table 7-2 decodes the actual value of Base. Refer to Chapter 6, "Exceptions," for more detailed information on the assignment of the exception vector offsets.

**Table 7-2. Value of Base for Predefined Memory Use**

| MSR[IP] | Value of Base |
|---|---|
| 0 | Base = 0x000 |
| 1 | Base = 0xFFF |

## 7.3.2 MMU Organization

Figure 7-1 shows a conceptual block diagram of the MMU in a 32-bit implementation. The 32-bit MMU implementation differs from the 64-bit implementation in that after an address is generated, the high-order bits of the effective address, EA0–EA19 (or a smaller set of address bits, EA0–EA*n*, in the cases of blocks), are translated into physical address bits PA0–PA19. The low-order address bits, A20–A31 are untranslated and therefore identical for both effective and physical addresses. After translating the address, the MMU passes the resulting 32-bit physical address to the memory subsystem.

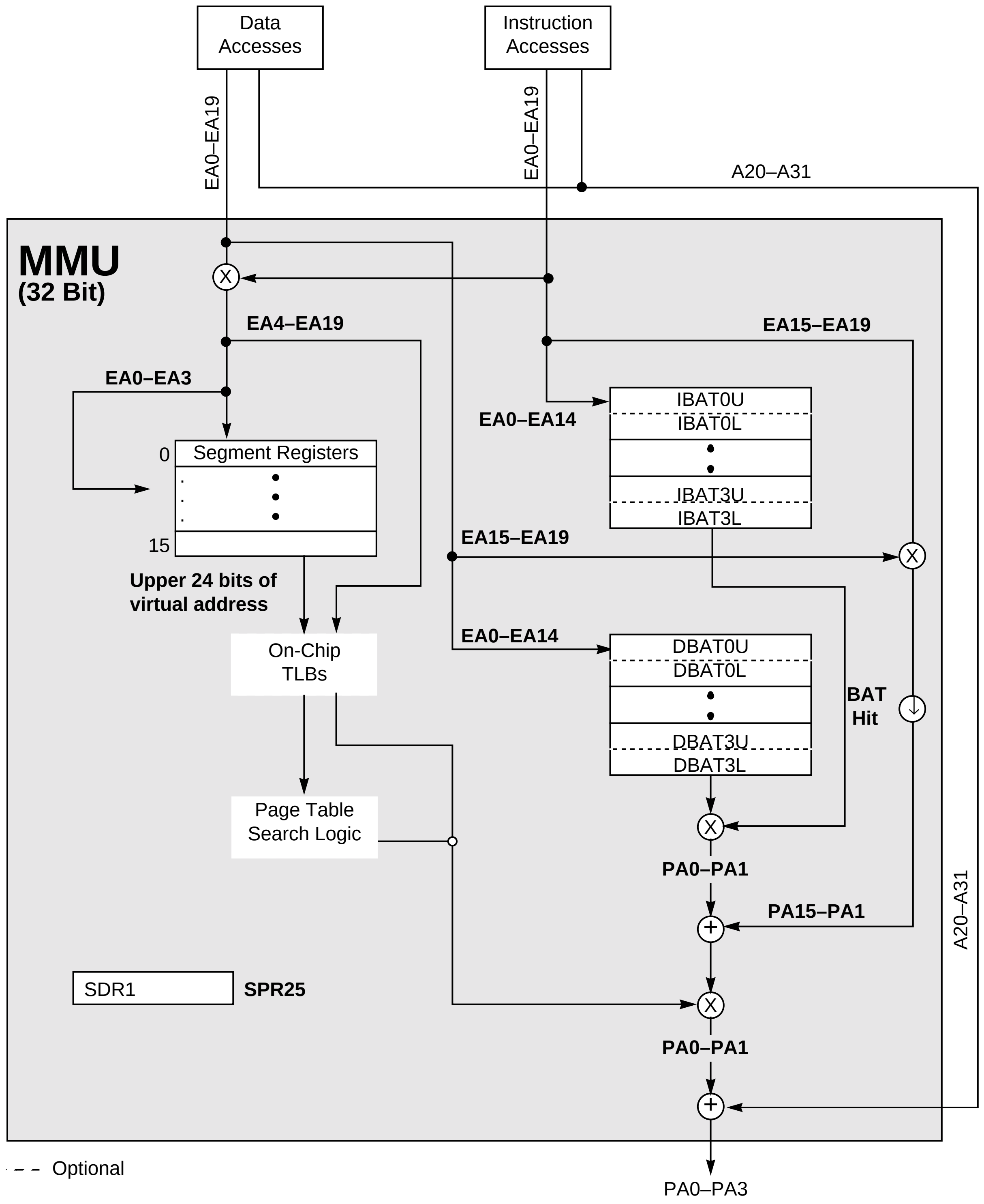


Figure 7-1. MMU Conceptual Block Diagram

## 7.3.3 Address Translation Mechanisms

The following types of address translation are supported:

- Page address translation—translates the page frame address for a 4-Kbyte page size

- Block address translation—translates the block number for blocks that range in size from 128 Kbyte to 256 Mbyte
- Real addressing mode address translation—when address translation is disabled, the physical address is identical to the effective address.

In addition, earlier processors implement a direct-store facility that is used to generate direct-store interface accesses on the external bus. Note that this facility is not optimized for performance and was present for compatibility with POWER devices. Future devices are not likely to support it; software should not depend on its effects and new software should not use it.

Figure 7-2 shows the address translation mechanisms provided by the MMU. The segment descriptors shown in the figure control both the page and direct-store segment address translation mechanisms. When an access uses the page or direct-store segment address translation, the appropriate segment descriptor is required. One of the 16 on-chip segment registers (which contain the segment descriptors) is selected by the highest-order effective address bits.

A control bit in the corresponding segment descriptor then determines if the access is to memory (memory-mapped) or to a direct-store segment. Note that the direct-store interface is present to allow certain older I/O devices to use this interface. When an access is determined to be to the direct-store interface space, the implementation invokes an elaborate hardware protocol for communication with these devices. The direct-store interface protocol is not optimized for performance, and therefore, its use is discouraged. The most efficient method for accessing I/O is by memory-mapping the I/O areas.

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 32-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.

Because blocks are larger than pages, there are fewer upper-order effective address bits to be translated into physical address bits (more low-order address bits (at least 17) are untranslated to form the offset into a block) for block address translation. Also, instead of segment descriptors and a page table, block address translations use the on-chip BAT registers as a BAT array. If an effective address matches the corresponding field of a BAT register, the information in the BAT register is used to generate the physical address; in this case, the results of the page translation (occurring in parallel) are ignored. Note that a matching BAT array entry takes precedence over a translation provided by the segment descriptor in all cases (even if the segment is a direct-store segment).

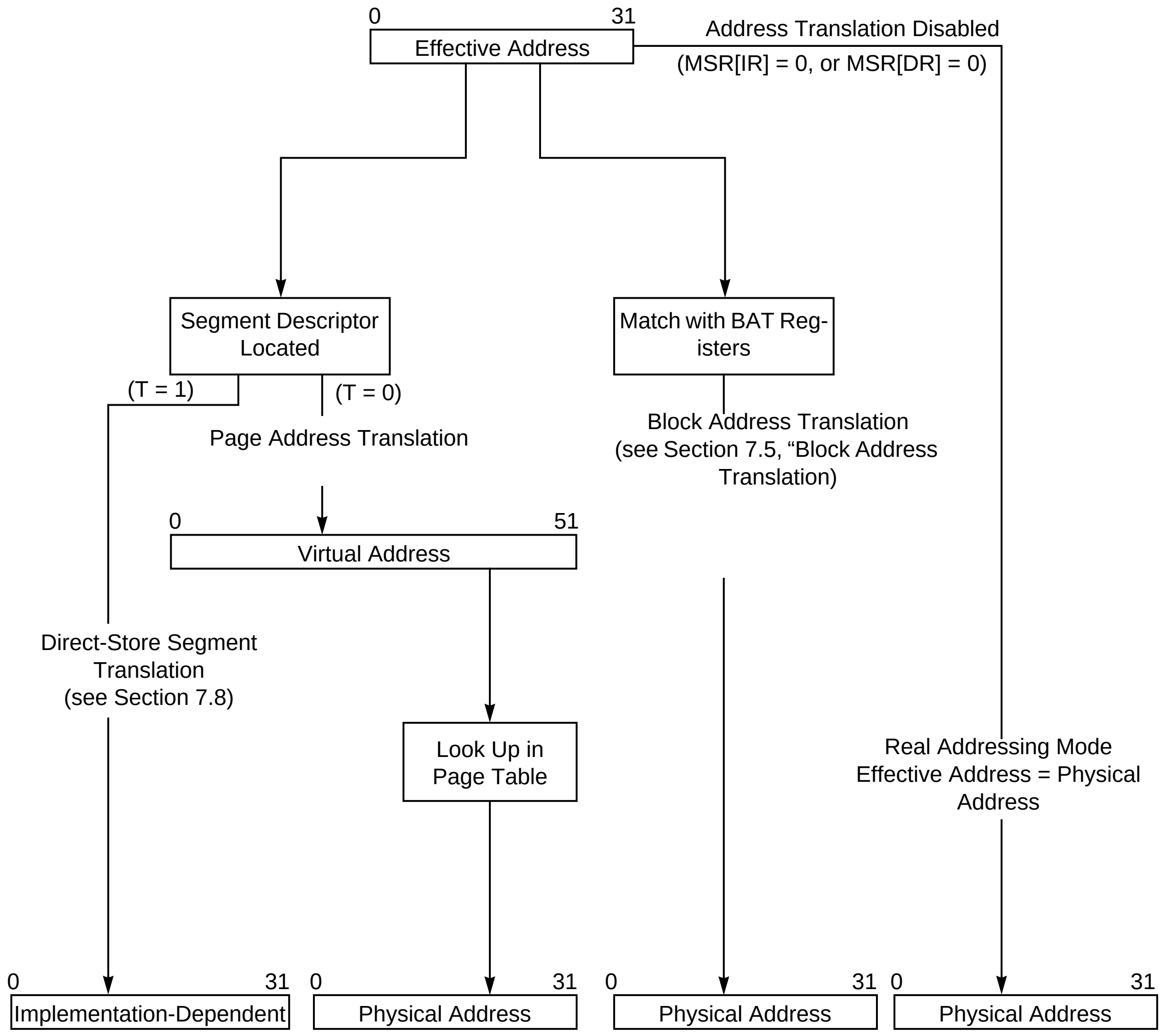

0
31
Effective Address
Address Translation Disabled
(MSR[IR] = 0, or MSR[DR] = 0)
Segment Descriptor Located
Match with BAT Registers
(T = 1)
(T = 0)
Page Address Translation
Block Address Translation (see Section 7.5, "Block Address Translation)
0
51
Virtual Address
Direct-Store Segment Translation (see Section 7.8)
Look Up in Page Table
Real Addressing Mode Effective Address = Physical Address
Implementation-Dependent
Physical Address
Physical Address
Physical Address

**Figure 7-2. Address Translation Types**

Direct-store address translation is used when the optional direct-store translation control bit (T bit) in the corresponding segment descriptor is set. In this case, the remaining information in the segment descriptor is interpreted as identifier information that is used with the remaining effective address bits to generate the protocol used in a direct-store interface access on the external interface; additionally, no TLB lookup or page table search is performed. Note that this facility is not likely to be supported in future processors.

When the processor generates an access, and the corresponding address translation enable bit in MSR is cleared, the resulting physical address is identical to the effective address and all other translation mechanisms are ignored. Instruction and data address translation is enabled by setting MSR[IR] and MSR[DR], respectively. See Section 7.3.6.1, "Real Addressing Mode and Block Address Translation Selection."

### 7.3.4 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. Figure 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 | — | — | — | √ | √ | √ |
| Supervisor-only-no-execute | — | — | — | — | √ | √ |
| Supervisor-write-only | √ | √ | — | √ | √ | √ |
| Supervisor-write-only-no-execute | — | √ | — | — | √ | √ |
| Both user/supervisor | √ | √ | √ | √ | √ | √ |
| Both (user/supervisor)-no-execute | — | √ | √ | — | √ | √ |
| Both (user/supervisor) read-only | √ | √ | — | √ | √ | — |
| Both (user/supervisor) read-only-no-execute | — | √ | — | — | √ | — |

√ Access permitted
— Protection violation

The no-execute option provided in the segment descriptor lets the operating system program whether or not instructions can be fetched from an area of memory. 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 (MSR[PR] = 0) to access the page. User accesses that map into a supervisor-only page cause an exception.

Note that independently 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. Section 5.2.5.2, “Instruction Cache Instructions,” describes 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).

For areas of memory that are translated by the block address translation mechanism, the protection options are similar, except that blocks are translated by separate mechanisms for instruction and data, blocks do not have a no-execute option, and blocks can be designated as enabled for user and supervisor accesses independently. Therefore, a block can be designated as supervisor-only, for example, but this block can be programmed such that all user accesses simply ignore the block translation, rather than take an exception in the case of a match. This allows a flexible way for supervisor and user programs to use overlapping effective address space areas that map to unique physical address areas (without exceptions occurring).

For direct-store segments, the MMU calculates a key bit based on the protection values programmed in the segment descriptor and the specific user/supervisor and read/write information for the particular access. However, this bit is merely passed on to the system interface to be transmitted in the context of the direct-store interface protocol. The MMU does not itself enforce any protection or cause any exception based on the state of the key bit for these accesses. The I/O controller device or other external hardware can optionally use this bit to enforce any protection required. Note that future devices are not likely to implement the direct-store facility.

Finally, a facility in the VEA and OEA allows pages or blocks to be designated as guarded, preventing speculative (refered to as out-of-order in the architecture) accesses that may cause undesired side effects. For example, areas of the memory map used to control I/O devices can be marked as guarded so accesses do not occur unless they are explicitly required by the program. Note that the terms ‘speculative’ and ‘out-of-order’ are used here as defined in Section 5.3.1.5.1, “Definition of Speculative and Out-of-Order Memory Accesses.” Refer to Section 5.3.1.5.4, “Speculative Accesses to Guarded Memory,” for a complete description of how accesses to guarded memory are restricted.

## 7.3.5 Page History Information

The MMU model also defines referenced (R) and changed (C) bits in the page address translation mechanism that can be used as history information relevant to the page. The operating system can use these bits to determine which areas of memory to write back to disk when new pages must be allocated in main memory. While these bits are initially programmed by the operating system into the page table, the architecture specifies that the processor maintains the R and C bits and updates them bits when required.

## 7.3.6 General Flow of MMU Address Translation

The following sections describe the general flow used to translate effective addresses to virtual and then physical addresses. Note that although there are references to the concept of an on-chip TLB, these entities may not be present in a particular hardware implementation for performance enhancement (and a particular implementation may have one or more TLBs). Thus, they are shown here as optional and only the software ramifications of the existence of a TLB are discussed.

#### 7.3.6.1 Real Addressing Mode and Block Address Translation 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 7.4, “Real Addressing Mode.”

Figure 7-3 shows the flow the MMU uses in determining whether to select real addressing mode, block address translation, or the segment descriptor (to select either direct-store or page address translation).

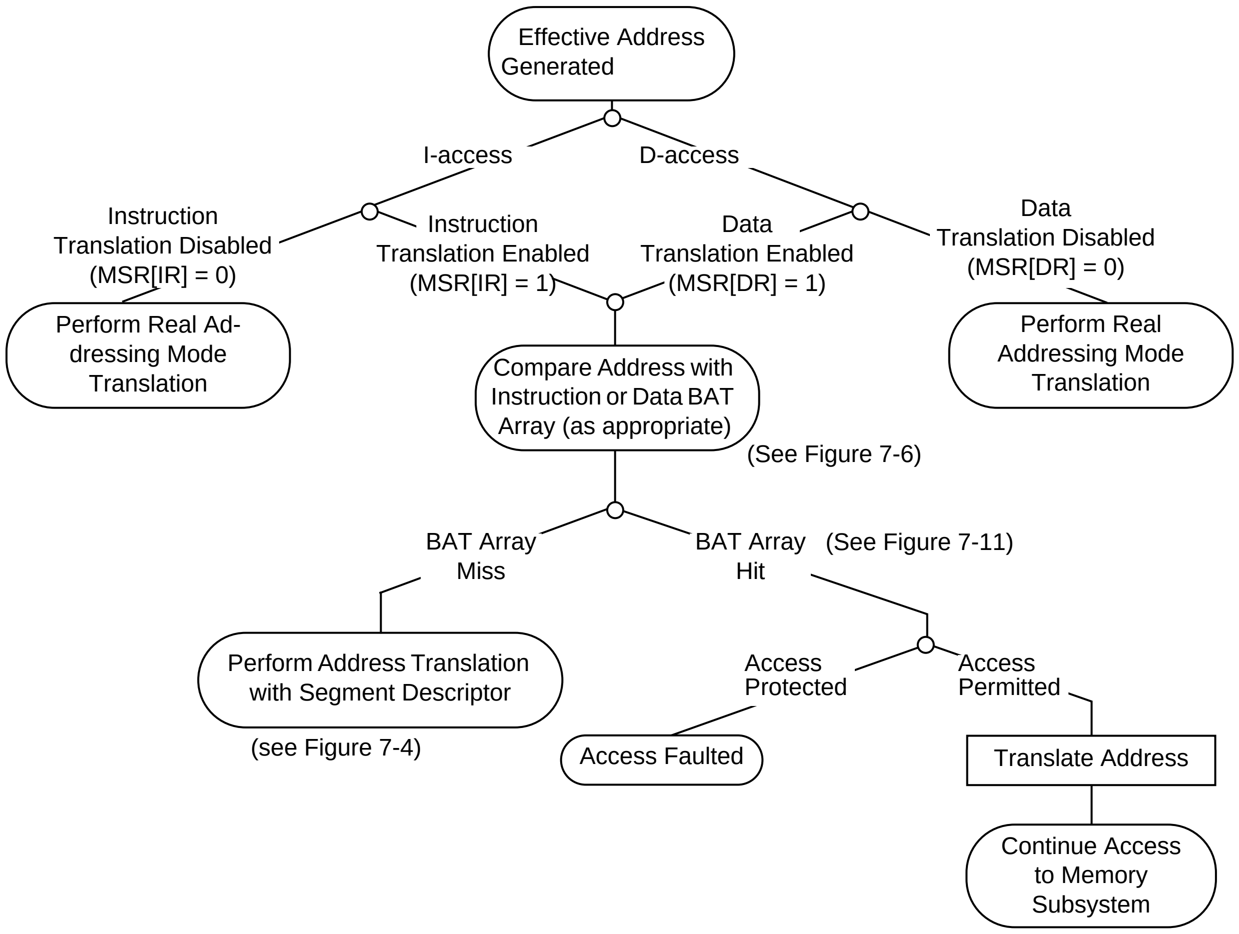

Effective Address Generated
I-access
D-access
Instruction Translation Disabled (MSR[IR] = 0)
Instruction Translation Enabled (MSR[IR] = 1)
Data Translation Enabled (MSR[DR] = 1)
Data Translation Disabled (MSR[DR] = 0)
Perform Real Addressing Mode Translation
Compare Address with Instruction or Data BAT Array (as appropriate)
Perform Real Addressing Mode Translation
(See Figure 7-6)
BAT Array Miss
BAT Array Hit
(See Figure 7-11)
Perform Address Translation with Segment Descriptor
(see Figure 7-4)
Access Protected
Access Permitted
Access Faulted
Translate Address
Continue Access to Memory Subsystem

**Figure 7-3. General Flow of Address Translation (Real Addressing Mode and Block)**

Note that if the BAT array search results in a hit, 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.

#### 7.3.6.2 Page and Direct-Store Address Translation Selection

If address translation is enabled (real addressing mode translation not selected) and the effective address information does not match a BAT array entry, the segment descriptor must be located. When the segment descriptor is located, the T bit in the segment descriptor selects whether the translation is to a page or to a direct-store segment as shown in Figure 7-4. In addition, 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.

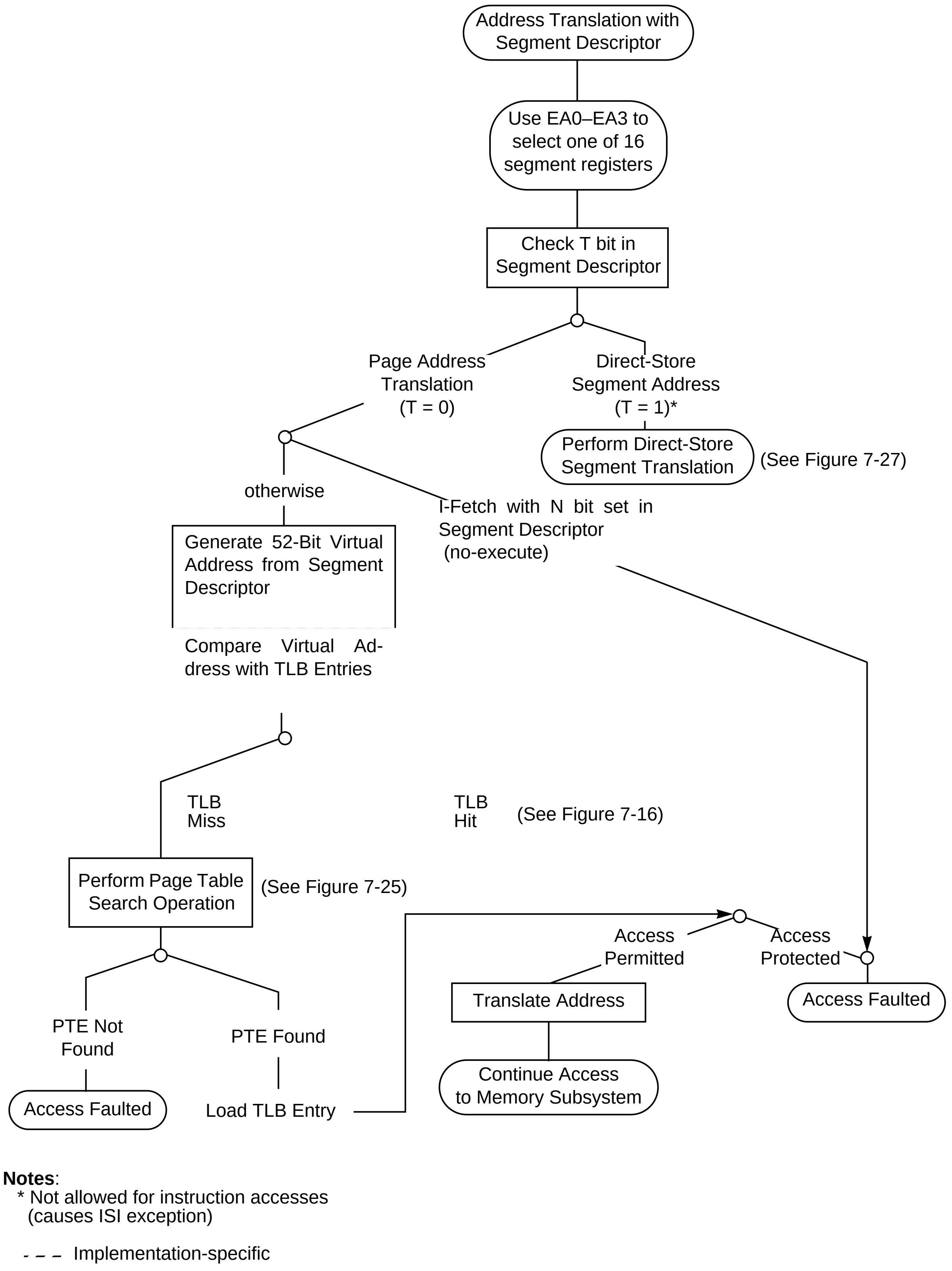

Address Translation with Segment Descriptor
Use EA0–EA3 to select one of 16 segment registers
Check T bit in Segment Descriptor
Page Address Translation (T = 0)
Direct-Store Segment Address (T = 1)*
Perform Direct-Store Segment Translation
(See Figure 7-27)
otherwise
I-Fetch with N bit set in Segment Descriptor (no-execute)
Generate 52-Bit Virtual Address from Segment Descriptor
Compare Virtual Address with TLB Entries
TLB Miss
TLB Hit
(See Figure 7-16)
Perform Page Table Search Operation
(See Figure 7-25)
PTE Not Found
PTE Found
Access Faulted
Load TLB Entry
Access Permitted
Access Protected
Translate Address
Continue Access to Memory Subsystem
Access Faulted
Notes:
* Not allowed for instruction accesses (causes ISI exception)
- – – Implementation-specific

**Figure 7-4. General Flow of Page and Direct-Store Address Translation**

For 32-bit implementations, the segment descriptor for an access is contained in one of 16 on-chip segment registers; effective address bits EA0–EA3 select one of the 16 SRs.

#### 7.3.6.2.1 Selection of Page Address Translation

If SR[T] = 0, page address translation is selected. The information in the segment descriptor is then used to generate the 52-bit 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). Once again, 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.

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.

## 7.3.7 MMU Exceptions Summary

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) and there is no valid BAT translation.
- There is no valid segment descriptor and there is no valid BAT translation.
- 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 Figure 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 (and no matching BAT array entry) | I access: ISI exception<br>SRR1[1] = 1 |
| | | D access: DSI exception<br>DSISR[1] = 1 |
| Block protection violation | Conditions described in Table 7-12 for block | I access: ISI exception<br>SRR1[4] = 1 |
| | | D access: DSI exception<br>DSISR[4] = 1 |
| Page protection violation | Conditions described in Table 7-19 for page | I access: ISI exception<br>SRR1[4] = 1 |
| | | D access: DSI exception<br>DSISR[4] = 1 |
| No-execute protection violation | Attempt to fetch instruction when SR[N] = 1 | ISI exception<br>SRR1[3] = 1 |
| Instruction fetch from direct-store segment—note that the direct-store facility is optional and being removed from the architecture. | Attempt to fetch instruction when SR[T] = 1 | ISI exception<br>SRR1[3] = 1 |
| Instruction fetch from guarded memory | Attempt to fetch instruction when MSR[IR] = 1 and either:<br>matching xBAT[G] = 1, or<br>no matching BAT entry and PTE[G] = 1 | ISI exception<br>SRR1[3] = 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 those 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 6.4.6, “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 (may cause exception or operation may be performed to memory) | **dcbz** instruction to write-through or cache-inhibited segment or block | Alignment exception (implementation-dependent) |
| **lwarx** or **stwcx.** with W = 1 (may cause exception or execute correctly) | Reservation instruction to write-through segment or block | DSI exception (implementation-dependent)<br>DSISR[5] = 1 |

**Table 7-5. Other MMU Exception Conditions (continued)**

| Condition | Description | Exception |
|---|---|---|
| **lwarx**, **stwcx.**, **eciwx**, or **ecowx** instruction to direct-store segment (may cause exception or may produce boundedly-undefined results)—note that the direct-store facility is optional and being removed from the architecture | Reservation instruction or external control instruction when SR[T] = 1 | DSI exception (implementation-dependent) DSISR[5] = 1 |
| Floating-point load or store to direct-store segment (may cause exception or instruction may execute correctly)—note that the direct-store facility is optional and being removed from the architecture | Floating-point memory access when SR[T] = 1 | Alignment exception (implementation-dependent) |
| Load or store operation that causes a direct-store error—note that the direct-store facility is optional and being removed from the architecture | Direct-store interface protocol signalled with an error condition | DSI exception DSISR[0] = 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) |

## 7.3.8 MMU Instructions and Register Summary

The MMU instructions and registers allow the operating system to set up the segment descriptors. Additionally, the operating system has the resources to set up the block address translation areas and the page tables in memory.

Note that because the implementation of TLBs is optional, instructions that refer to TLBs are also optional. However, because TLBs 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 the tables in memory are modified. Therefore, the OEA specifies that a processor implementing a TLB is guaranteed to have a means for doing the following:

- Invalidating an individual TLB entry
- Invalidating the entire TLB

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 (a loop invalidating individual TLB entries may be used to invalidate the entire TLB, for example), 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 so as to purge all TLB entries in a congruence class (that is, all TLB entries indexed by the specified EA which can include corresponding entries in data and instruction TLBs) or the entire TLB.

Note that 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 segment register and TLB concepts described here are conceptual; that is, a processor may implement parallel sets of segment registers (and even TLBs) for instructions and data.

Because the MMU specification 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 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 | Syntax | Description |
|---|---|---|
| Move to Segment Register | **mtsr** SR**,r**S | SR[SR]⟵ **r**S<br>32-bit implementations only |
| Move to Segment Register Indirect | **mtsrin r**S**,r**B | SR[**r**B[0–3]]⟵**r**S<br>32-bit implementations only |
| Move from Segment Register | **mfsr r**D**,**SR | **r**D⟵SR[SR]<br>32-bit implementations only |
| Move from Segment Register Indirect | **mfsrin r**D**,r**B | **r**D⟵SR[**r**B[0–3]]<br>32-bit implementations only |
| Translation Lookaside Buffer Invalidate All (optional) | **tlbia** | For all TLB entries, TLB[V]⟵0<br>Causes invalidation of TLB entries only for processor that executed the **tlbia** |
| Translation Lookaside Buffer Invalidate Entry (optional) | **tlbie r**B | If TLB hit (for effective address specified as **r**B), TLB[V]⟵0<br>Causes TLB invalidation of entry in all processors in system |
| Translation Lookaside Buffer Synchronize (optional) | **tlbsync** | Ensures that all **tlbie** instructions previously executed by the processor executing the **tlbsync** instruction have completed on all processors |

Table 7-7 summarizes the registers that the operating system uses to program the MMU. These registers are accessible to supervisor-level software only (supervisor level is referred to as privileged state in the architecture specification). These registers are described in detail in Chapter 2, “Register Set.”

**Table 7-7. MMU Registers**

| Register | Description |
|---|---|
| Segment registers (SR0–SR15) | The sixteen 32-bit segment registers are present only in 32-bit implementations of the PowerPC architecture. Figure 7-13 shows the format of a segment register. The fields in the segment register are interpreted differently depending on the value of bit 0. The segment registers are accessed by the **mtsr**, **mtsrin**, **mfsr**, and **mfsrin** instructions. |
| BAT registers (IBAT0U–IBAT3U, IBAT0L–IBAT3L, DBAT0U–DBAT3U, and DBAT0L–DBAT3L) | There are 16 BAT registers, organized as four pairs of instruction BAT registers (IBAT0U–IBAT3U paired with IBAT0L–IBAT3L) and four pairs of data BAT registers (DBAT0U–DBAT3U paired with DBAT0L–DBAT3L). The BAT registers are defined as 32-bit registers in 32-bit implementations. These SPRs are accessed by the **mtspr** and **mfspr** instructions. |
| SDR1 register | The SDR1 register specifies the base and size of the page tables in memory. SDR1 is defined as a 32-bit register for 32-bit implementations. This is a special-purpose register that is accessed by the **mtspr** and **mfspr** instructions. |

### 7.3.9 TLB Entry Invalidation

Optionally, 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.

## 7.4 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 2.3.17, “Synchronization Requirements for Special Registers and for Lookaside Buffers,” describes 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 7.5.4, “Block Memory Protection,” and Section 7.6.4, “Page Memory Protection” and do not cause the recording of referenced and changed information (described in Section 7.6.3, “Page History Recording”).

For data accesses that use real addressing mode, the memory access mode bits (WIMG) are assumed to be 0b0011. 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. For instruction accesses in real addressing mode, the memory access mode bits (WIMG) are assumed to be either 0b0001 or 0b0011. That is, caching is enabled (I = 0) and the memory is guarded. 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 5.3.1, “Memory/Cache Access Attributes.”

Note that 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, “Register Set.”

Note that 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. Note that this can also occur when translation is enabled and SDR1 sets up the translation such that nonexistent memory is accessed. Section 6.4.2, “Machine Check Exception (0x00200),” gives more information on machine check exceptions.

## 7.5 Block Address Translation

The block address translation (BAT) mechanism in the OEA provides a way to map ranges of effective addresses larger than a single page into contiguous areas of physical memory. Such areas can be used for data that is not subject to normal virtual memory handling (paging), such as a memory-mapped display buffer or an extremely large array of numbers.

The following sections describe the implementation of block address translation, including the block protection mechanism, followed by a block translation summary with a detailed flow diagram.

### 7.5.1 BAT Array Organization

The block address translation mechanism is implemented as a software-controlled BAT array. The BAT array maintains the address translation information for eight blocks of memory. The BAT array is maintained by the system software and is implemented as a set of 16 special-purpose registers (SPRs). Each block is defined by a pair of SPRs called upper and lower BAT registers that contain the effective and physical addresses for the block.

The BAT registers can be read from or written to by the **mfspr** and **mtspr** instructions; access to the BAT registers is privileged. Section 7.5.3, “BAT Register Implementation of BAT Array,” gives more information about the BAT registers. Note that the BAT array entries are completely ignored for TLB invalidate operations detected in hardware and in the execution of the **tlbie** or **tlbia** instruction.

Figure 7-5 shows the organization of the BAT array. Four pairs of BAT registers are provided for translating instruction addresses and four pairs of BAT registers are used for translating data addresses. These eight pairs of BAT registers comprise two four-entry fully-associative BAT arrays (each BAT array entry corresponds to a pair of BAT registers). The BAT array is fully-associative in that any address can reside in any BAT. In addition, the effective address field of all four corresponding entries (instruction or data) is simultaneously compared with the effective address of the access to check for a match.

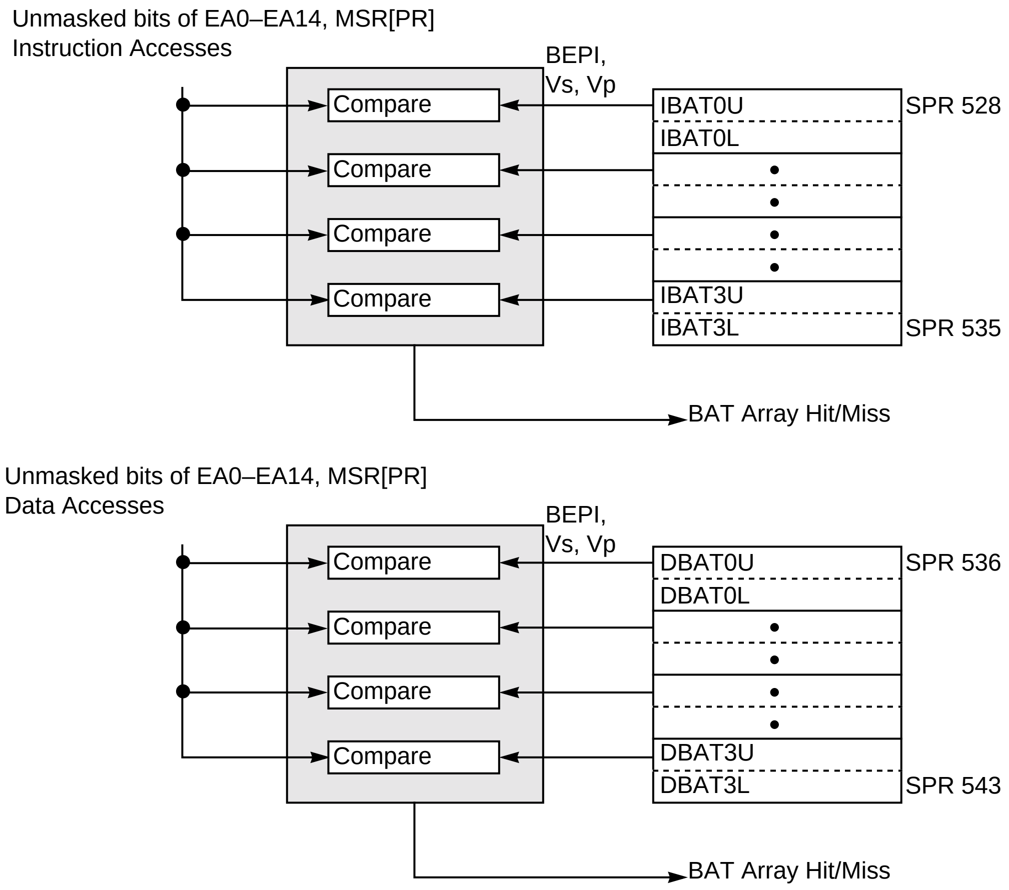


**Figure 7-5. BAT Array Organization**

Each pair of BAT registers defines the starting address of a block in the effective address space, the size of the block, and the start of the corresponding block in physical address space. If an effective address is within the range defined by a pair of BAT registers, its physical address is defined as the starting physical address of the block plus the low-order effective address bits.

Blocks are restricted to a finite set of sizes, from 128 Kbytes ($2^{17}$ bytes) to 256 Mbytes ($2^{28}$ bytes). The starting address of a block in both effective address space and physical address space is defined as a multiple of the block size.

It is an error for system software to program the BAT registers such that an effective address is translated by more than one valid IBAT pair or more than one valid DBAT pair. If this occurs, the results are undefined and may include a spurious violation of the memory protection mechanism, a machine check exception, or a checkstop condition.

The following equation determines whether a BAT entry is valid for a particular access:

BAT_entry_valid = (Vs & ¬MSR[PR]) | (Vp & MSR[PR])

If a BAT entry is not valid for a given access, it does not participate in address translation for that access. Two BAT entries may not map an overlapping effective address range and be valid at the same time.

Entries that have complementary settings of V[s] and V[p] may map overlapping effective address blocks. Complementary settings would be as follows:

BAT entry A: Vs = 1, Vp = 0

BAT entry B: Vs = 0, Vp = 1

## 7.5.2 Recognition of Addresses in BAT Arrays

The BAT arrays are accessed in parallel with segmented address translation to determine whether a particular effective address corresponds to a block defined by the BAT arrays. If an effective address is within a valid BAT area, the physical address for the memory access is determined as described in Section 7.5.5, “Block Physical Address Generation.”

Block address translation is enabled only when address translation is enabled (MSR[IR] = 1 and/or MSR[DR] = 1). Also, a matching BAT array entry always takes precedence over any segment descriptor translation, independent of the setting of SR[T], and the segment descriptor information is completely ignored.

Figure 7-6 shows the flow of the BAT array comparison used in block address translation. When an instruction fetch operation is required, the effective address is compared with the four instruction BAT array entries; similarly, the effective addresses of data accesses are compared with the four data BAT array entries. The BAT arrays are fully-associative in that any of the four instruction or data BAT array entries can contain a matching entry (for an instruction or data access, respectively).

Note that Figure 7-6 assumes that the protection bits, BATL[PP], allow an access to occur. If not, an exception is generated, as described in Section 7.5.4, “Block Memory Protection.”

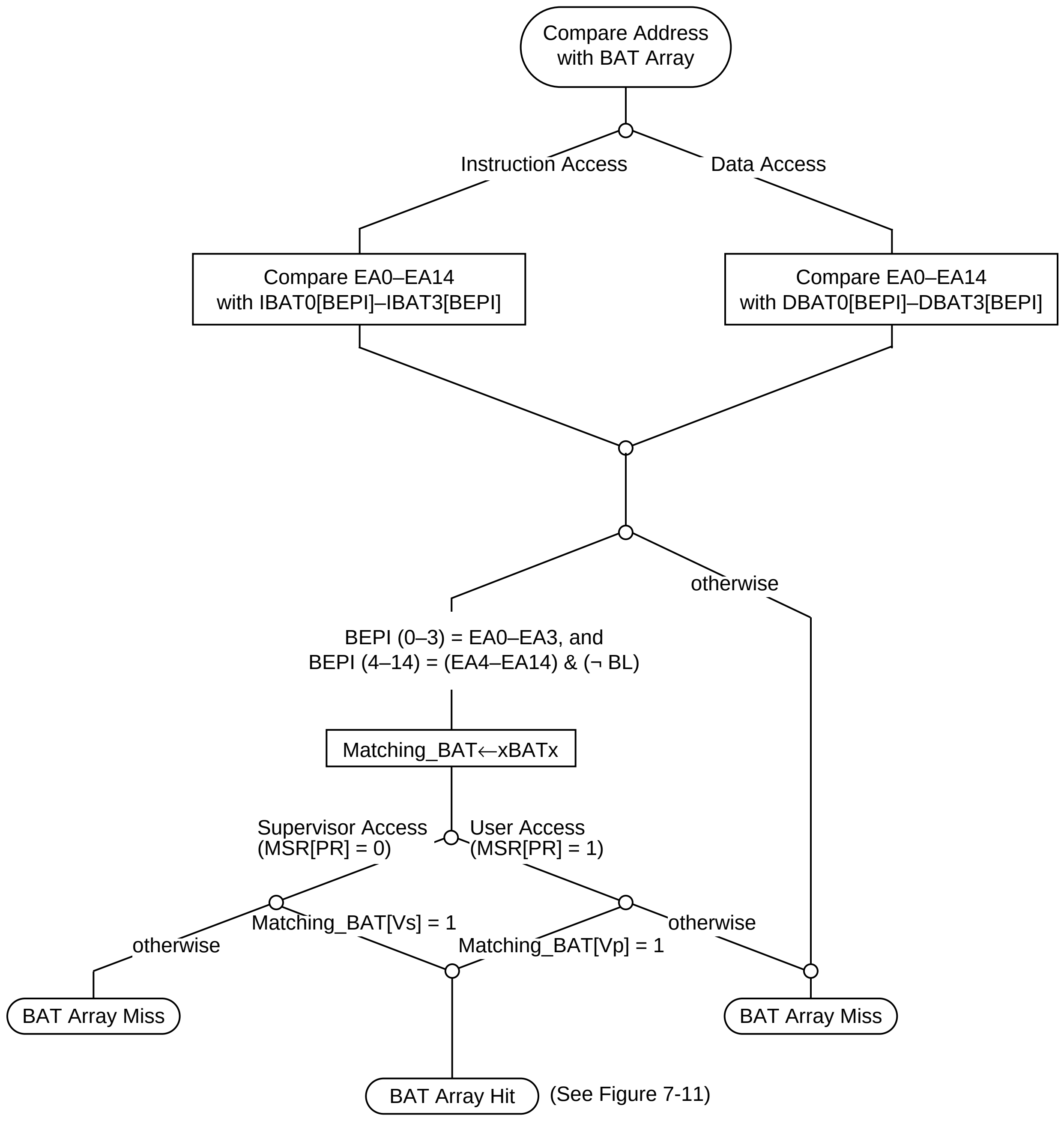

Compare Address with BAT Array
Instruction Access
Data Access
Compare EA0–EA14 with IBAT0[BEPI]–IBAT3[BEPI]
Compare EA0–EA14 with DBAT0[BEPI]–DBAT3[BEPI]
otherwise
BEPI (0–3) = EA0–EA3, and
BEPI (4–14) = (EA4–EA14) & (¬ BL)
Matching_BAT←xBATx
Supervisor Access (MSR[PR] = 0)
User Access (MSR[PR] = 1)
Matching_BAT[Vs] = 1
otherwise
Matching_BAT[Vp] = 1
otherwise
BAT Array Miss
BAT Array Miss
BAT Array Hit
(See Figure 7-11)

**Figure 7-6. BAT Array Hit/Miss Flow**

Two BAT array entry fields are compared to determine if there is a BAT array hit—a block effective page index (BEPI) field, which is compared with the high-order effective address bits, and one of two valid bits (Vs or Vp), which is evaluated relative to the value of MSR[PR]. Note that the figure assumes a block size of 128 Kbytes (all bits of BEPI are used in the comparison); the actual number of bits of the BEPI field that are used are masked by the BL field (block length) as described in Section 7.5.3, “BAT Register Implementation of BAT Array.”

Thus, the specific criteria for determining a BAT array hit are as follows:

- The upper-order 15 bits of the effective address, subject to a mask, must match the BEPI field of the BAT array entry.
- The appropriate valid bit in the BAT array entry must set to one as follows:
  - — MSR[PR] = 0 corresponds to supervisor mode; in this mode, Vs is checked.
  - — MSR[PR] = 1 corresponds to user mode; in this mode, Vp is checked.

The matching entry is then subject to the protection checking described in Section 7.5.4, “Block Memory Protection,” before it is used as the source for the physical address. Note that if a user mode program performs an access with an effective address that matches the BEPI field of a BAT area defined as valid only for supervisor accesses (Vp = 0 and Vs = 1) for example, the BAT mechanism does not generate a protection violation and the BAT entry is simply ignored. Thus, a supervisor program can use the block address translation mechanism to share a portion of the effective address space with a user program (that uses page address translation for this area).

If a memory area is to be mapped by the BAT mechanism for both instruction and data accesses, the mapping must be set up in both an IBAT and DBAT entry; this is the case even on implementations that do not have separate instruction and data caches.

Note that a block can be defined to overlay part of a segment such that the block portion is nonpaged although the rest of the segment can be paged. This allows nonpaged areas to be specified within a segment. Thus, if an area of memory is translated by an instruction BAT entry and data accesses are not also required to that same area of memory, PTEs are not required for that area of memory. Similarly, if an area of memory is translated by a data BAT entry, and instruction accesses are not also required to that same area of memory, PTEs are not required for that area of memory.

## 7.5.3 BAT Register Implementation of BAT Array

Recall that the BAT array is comprised of four entries used for instruction accesses and four entries used for data accesses. Each BAT array entry consists of a pair of BAT registers—an upper and a lower BAT register for each entry. The BAT registers are accessed with the **mtspr** and **mfspr** instructions and are only accessible to supervisor-level programs. See Appendix F, “Simplified Mnemonics,” for a list of simplified mnemonics for use with the BAT registers. (Note that simplified mnemonics are referred to as extended mnemonics in the architecture specification.)

The format and bit definitions of the upper and lower BAT registers are shown in Figure 7-7 and Figure 7-8, respectively.

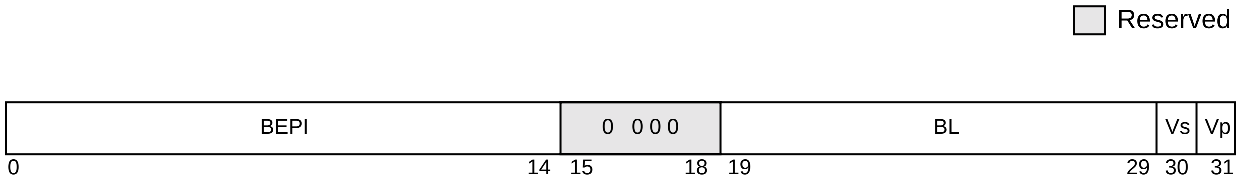
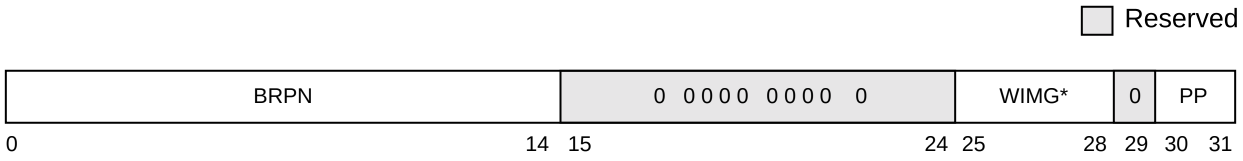


**Figure 7-7. Format of Upper BAT Registers**



*W and G bits are not defined for IBAT registers. Attempting to write to these bits causes boundedly-undefined results.

**Figure 7-8. Format of Lower BAT Registers**

The BAT registers contain the effective-to-physical address mappings for blocks of memory. This mapping includes the effective address bits that are compared with the effective address of the access, the memory/cache access mode bits (WIMG), and the protection bits for the block. The size of the block and the starting address of the block are defined by the physical block number (BRPN) and block size mask (BL) fields.

Figure 7-8 describes the bits in the upper and lower BAT registers. Note that the W and G bits are defined for BAT registers that translate data accesses (DBAT registers); attempting to write to the W and G bits in IBAT registers causes boundedly-undefined results.

**Table 7-9. BAT Registers—Field and Bit Descriptions**

| Upper/Lower BAT | Bits | Name | Description |
|---|---|---|---|
| Upper BAT Register | 0–14 | BEPI | Block effective page index. This field is compared with high-order bits of the logical address to determine if there is a hit in that BAT array entry. (Note that the architecture specification refers to logical address as effective address.) |
| | 15–18 | — | Reserved |
| | 19–29 | BL | Block length. BL is a mask that encodes the size of the block. Values for this field are listed in Figure 2-13. |
| | 30 | Vs | Supervisor mode valid bit. This bit interacts with MSR[PR] to determine if there is a match with the logical address. For more information, see Section 7.5.2, “Recognition of Addresses in BAT Arrays." |
| | 31 | Vp | User mode valid bit. This bit also interacts with MSR[PR] to determine if there is a match with the logical address. For more information, see Section 7.5.2, “Recognition of Addresses in BAT Arrays.” |

**Table 7-9. BAT Registers—Field and Bit Descriptions**

| Upper/Lower BAT | Bits | Name | Description |
|---|---|---|---|
| Lower BAT Register | 0–14 | BRPN | This field is used in conjunction with the BL field to generate high-order bits of the physical address of the block. |
| | 15–24 | — | Reserved |
| | 25–28 | WIMG | Memory/cache access mode bits<br>W Write-through<br>I Caching-inhibited<br>M Memory coherence<br>G Guarded<br>Attempting to write to the W and G bits in IBAT registers causes boundedly-undefined results. For detailed information about the WIMG bits, see Section 5.3.1, “Memory/Cache Access Attributes." |
| | 29 | — | Reserved |
| | 30–31 | PP | Protection bits for block. This field determines the protection for the block as described in Section 7.5.4, “Block Memory Protection." |

The BL field in the upper BAT register is a mask that encodes the size of the block. Table 7-10 defines the bit encodings for the BL field of the upper BAT register.

**Table 7-10. Upper BAT Register Block Size Mask Encodings**

| Block Size | BL Encoding |
|---|---|
| 128 Kbytes | 000 0000 0000 |
| 256 Kbytes | 000 0000 0001 |
| 512 Kbytes | 000 0000 0011 |
| 1 Mbyte | 000 0000 0111 |
| 2 Mbytes | 000 0000 1111 |
| 4 Mbytes | 000 0001 1111 |
| 8 Mbytes | 000 0011 1111 |
| 16 Mbytes | 000 0111 1111 |
| 32 Mbytes | 000 1111 1111 |
| 64 Mbytes | 001 1111 1111 |
| 128 Mbytes | 011 1111 1111 |
| 256 Mbytes | 111 1111 1111 |

Only the values in Table 7-10 are valid for BL. An effective address is lies within a BAT area if the appropriate bits (determined by the BL) of the effective address match the value in the BEPI field of the upper BAT register and if the appropriate valid bit (Vs or Vp) is set. Note that for an access to occur, the protection bits (PP) in the lower BAT register must be set appropriately, as described in Section 7.5.4, “Block Memory Protection.”

The number of zeros in the BL field determines the bits of the effective address used in the comparison with the BEPI field to determine if there is a hit in that BAT array entry. The rightmost bit of the BL field is aligned with bit 14 of the effective address; bits of the effective address corresponding to ones in the BL field are then cleared for the comparison.

The value loaded into BL determines both the block size and the alignment of the block in both effective address and physical address space. The values loaded into the BEPI and BRPN fields must have at least as many low-order zeros as there are ones in BL. Otherwise, the results are undefined. Also, if the processor does not support 32 bits of physical address, software should write zeros to those unsupported bits in the BRPN field (as the implementation treats them as reserved). Otherwise, a machine check exception can occur.

## 7.5.4 Block Memory Protection

After an effective address is determined to be within a block defined by the BAT array, the access is validated by the memory protection mechanism. If this mechanism prohibits the access, a block protection violation exception condition (DSI or ISI exception) is generated.

The 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 block protection mechanism provides protection at the granularity defined by the block size (128 Kbyte to 256 Mbyte).

As the memory protection mechanism used by the block and page address translation is different, refer to Section 7.6.4, “Page Memory Protection,” for specific information unique to page address translation.

For block address translation, the memory protection mechanism is controlled by the PP bits located in the lower BAT register. The PP bits define access options for the block. Table 7-11 shows the types of accesses allowed for the possible PP bit combinations.

**Table 7-11. Access Protection Control for Blocks**

| PP | Accesses Allowed |
|---|---|
| 00 | No access |
| x1 | Read only |
| 10 | Read/write |

Thus, any access attempted (read or write) when PP = 00 results in a protection violation exception condition. When PP = x1, an attempt to perform a write access causes a protection violation exception condition, and when PP = 10, all accesses are allowed. 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 bit 4 of DSISR is set.
- For instruction accesses, an ISI exception is generated and SRR1 bit 4 is set.

See Chapter 6, “Exceptions,” for more information about these exceptions.

Table 7-12 shows a summary of the conditions that cause exceptions for supervisor and user read and write accesses within a BAT area. Each BAT array entry is programmed to be either used or ignored for supervisor and user accesses via the BAT array entry valid bits, and the PP bits enforce the read/write protection options. Note that the valid bits (Vs and Vp) are used as part of the match criteria for a BAT array entry and are not explicitly part of the protection mechanism.

**Table 7-12. Access Protection Summary for BAT Array**

| Vs | Vp | PP Field | Block Type | User Read | User Write | Supervisor Read | Supervisor Write |
|---|---|---|---|---|---|---|---|
| 0 | 0 | xx | No BAT array match | Not used | Not used | Not used | Not used |
| 0 | 1 | 00 | User—no access | Exception | Exception | Not used | Not used |
| 0 | 1 | x1 | User-read-only | √ | Exception | Not used | Not used |
| 0 | 1 | 10 | User read/write | √ | √ | Not used | Not used |
| 1 | 0 | 00 | Supervisor—no access | Not used | Not used | Exception | Exception |
| 1 | 0 | x1 | Supervisor-read-only | Not used | Not used | √ | Exception |
| 1 | 0 | 10 | Supervisor read/write | Not used | Not used | √ | √ |
| 1 | 1 | 00 | Both—no access | Exception | Exception | Exception | Exception |
| 1 | 1 | x1 | Both-read-only | √ | Exception | √ | Exception |
| 1 | 1 | 10 | Both read/write | √ | √ | √ | √ |

**Note**: The term ‘Not used’ implies that the access is not translated by the BAT array and is translated by the page address translation mechanism described in Section 7.6, “Memory Segment Model,” instead.

Note that because access to the BAT registers is privileged, only supervisor programs can modify the protection and valid bits for the block.

Figure 7-9 expands on the actions taken by the processor in the case of a memory protection violation. Note that the **dcbt** and **dcbtst** instructions do not cause exceptions; in the case of a memory protection violation for the attempted execution of one of these instructions, the translation is aborted and the instruction executes as a no-op (no violation is reported). Refer to Chapter 6, “Exceptions,” for a complete description of the SRR1 and DSISR bit settings for the protection violation exceptions.

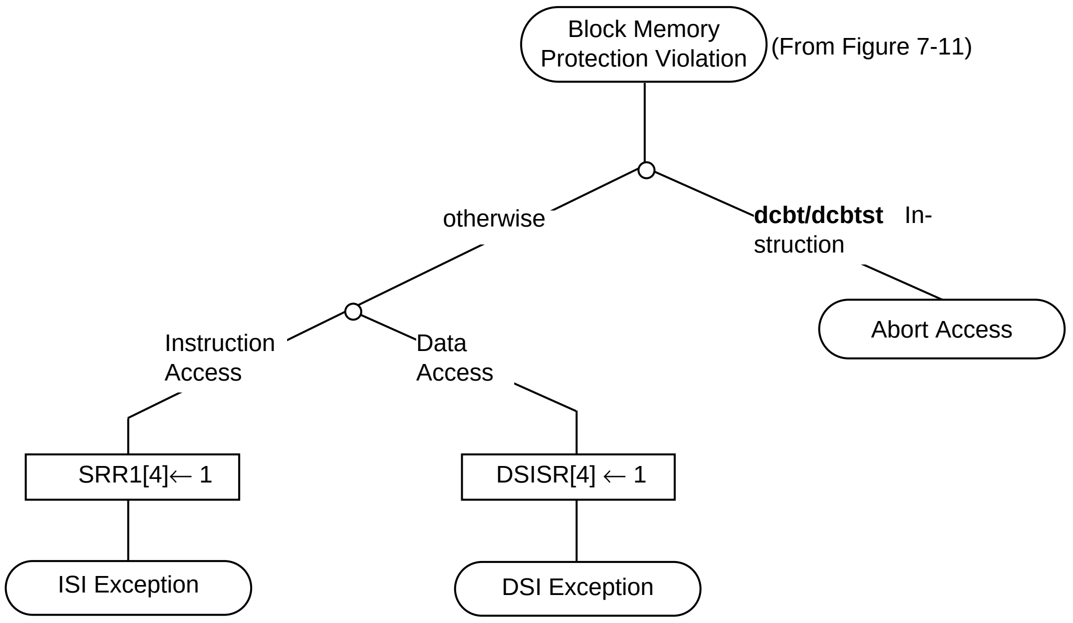


**Figure 7-9. Memory Protection Violation Flow for Blocks**

## 7.5.5 Block Physical Address Generation

Access to the physical memory within the block is made according to the memory/cache access mode defined by the WIMG bits in the lower BAT register. These bits apply to the entire block rather than to an individual page as described in Section 5.3.1, "Memory/Cache Access Attributes."

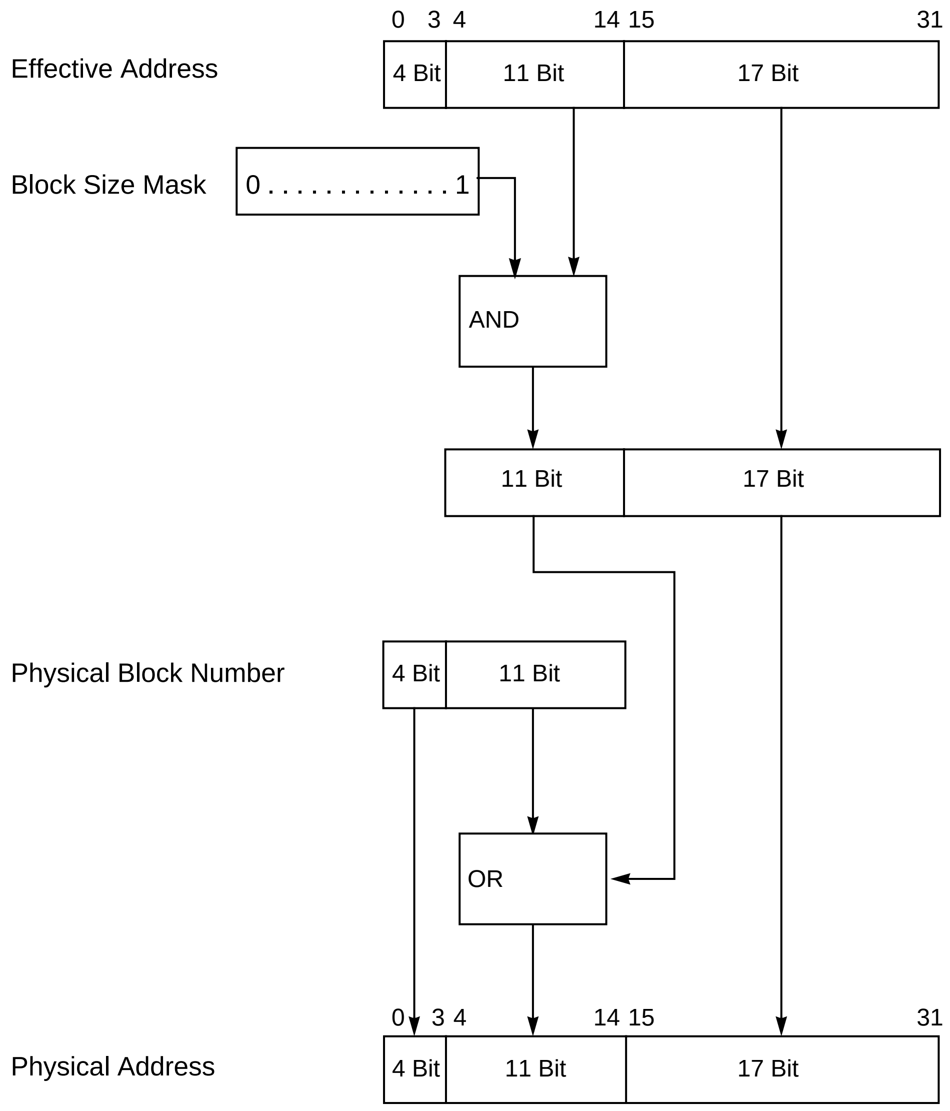


**Figure 7-10. Block Physical Address Generation**

## 7.5.6 Block Address Translation Summary

Figure 7-11 is an expansion of the BAT array hit branch of Figure 7-3 and shows the translation of address bits for 32-bit implementations. Note that the figure does not show when many of the exceptions in Figure 7-5 are detected or taken as this is implementation-specific.

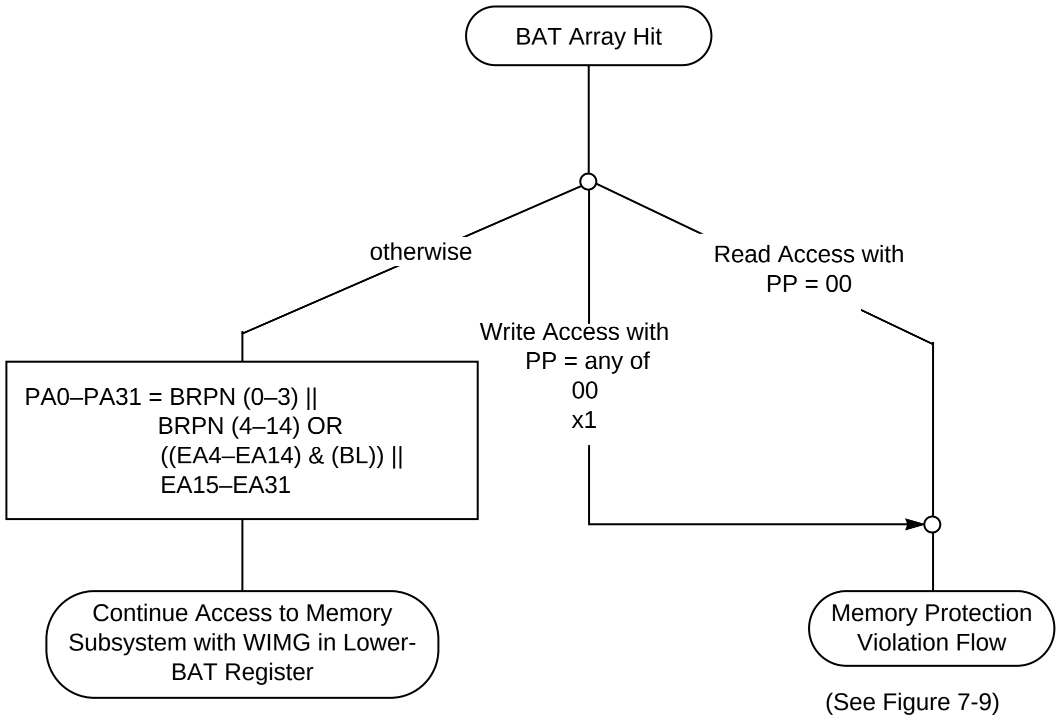

BAT Array Hit
otherwise
Write Access with PP = any of 00 x1
Read Access with PP = 00
PA0–PA31 = BRPN (0–3) || BRPN (4–14) OR ((EA4–EA14) & (BL)) || EA15–EA31
Continue Access to Memory Subsystem with WIMG in Lower-BAT Register
Memory Protection Violation Flow
(See Figure 7-9)

**Figure 7-11. Block Address Translation Flow**

## 7.6 Memory Segment Model

Memory in the OEA is divided into 256-Mbyte segments. This segmented memory model provides a way to map 4-Kbyte pages of effective addresses to 4-Kbyte pages in physical memory (page address translation), while providing the programming flexibility afforded by a large virtual address space (52 bits).

A page address translation may be superseded by a matching block address translation as described in Section 7.5, “Block Address Translation.” If not, the page translation proceeds in the following two steps:

1. From effective address to the virtual address (which never exists as a specific entity but can be considered to be the concatenation of the virtual page number and the byte offset within a page), 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.

### 7.6.1 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 one of the following two types:

- Memory segment—An effective address in these segments represents a virtual address that is used to define the physical address of the page.
- Direct-store segment—References made to direct-store segments do not use the virtual paging mechanism of the processor. Note that the direct-store facility is optional and being removed from the architecture. See Section 7.8, “Direct-Store Segment Address Translation,” for a complete description of the mapping of direct-store segments for those processors that implement it.

The T bit in the segment descriptor selects between memory segments and direct-store segments, as shown in Figure 7-13.

**Table 7-13. Segment Descriptor Types**

| Segment Descriptor T Bit | Segment Type |
|---|---|
| 0 | Memory segment |
| 1 | Direct-store segment—optional, being removed from the architecture. Its use is discouraged. |

### 7.6.1.1 Selection of Memory Segments

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 7.4, “Real Addressing Mode.” Otherwise, if T = 0 in the corresponding segment descriptor (and the address is not translated by the BAT mechanism), 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 that I/O devices can be easily mapped into memory space and used as memory-mapped I/O.

### 7.6.1.2 Selection of Direct-Store Segments

As described for memory segments, all accesses generated by the processor (with translation enabled) map to a segment descriptor. If T = 1 for the selected segment descriptor, the access maps to the direct-store interface space and the access proceeds as described in Section 7.8, “Direct-Store Segment Address Translation.” Because the direct-store interface is present only for compatibility with existing I/O devices that used this interface and because the direct-store interface protocol is not optimized for performance, its use is discouraged. Additionally, future devices are not likely to support it.

Thus, software should not depend on its results and new software should not use it. The most efficient method for accessing I/O is by mapping the I/O areas to memory segments.

## 7.6.2 Page Address Translation Overview

The translation of effective addresses to physical addresses is shown in Figure 7-12. The address translation is as follows:

- Bits 0–3 of the effective address comprise the segment register number used to select a segment descriptor, from which the virtual segment ID (VSID) is extracted.
- Bits 4–19 of the effective address correspond to the page number within the segment; these 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 (RPN).
- Bits 20–31 of the effective address are the byte offset within the page; these are concatenated with the RPN field of a PTE to form the physical address used to access memory.

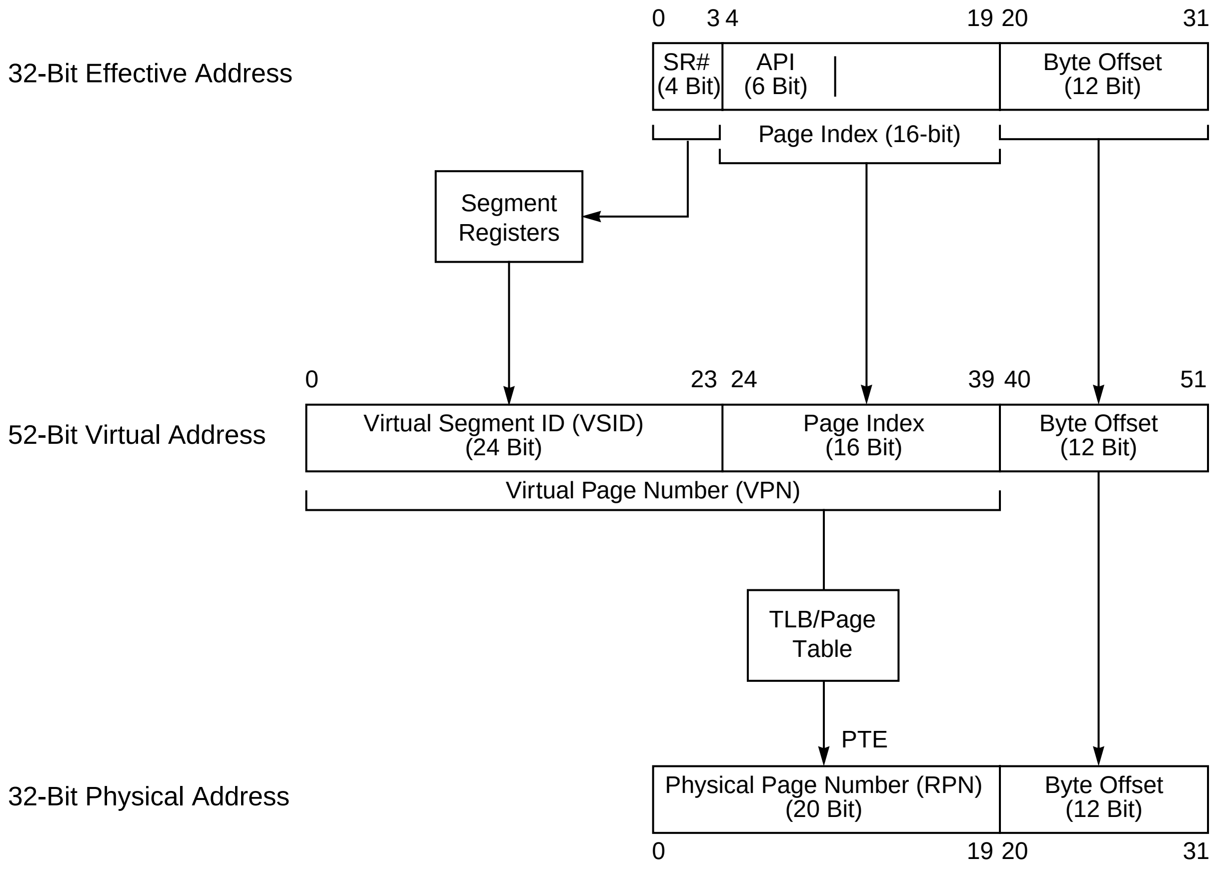


**Figure 7-12. Page Address Translation Overview**

### 7.6.2.1 Segment Descriptor Definitions

The fields in the segment descriptors are interpreted differently depending on the value of the T bit within the descriptor. When T = 1, the segment descriptor defines a direct-store segment, and the format is as described in Section 7.8.1, “Segment Descriptors for Direct-Store Segments.”

#### 7.6.2.1.1 Segment Descriptor Format

The segment descriptors are 32 bits long and reside in one of 16 on-chip segment registers. Figure 7-13 shows the format of a segment register used in page address translation (T = 0).

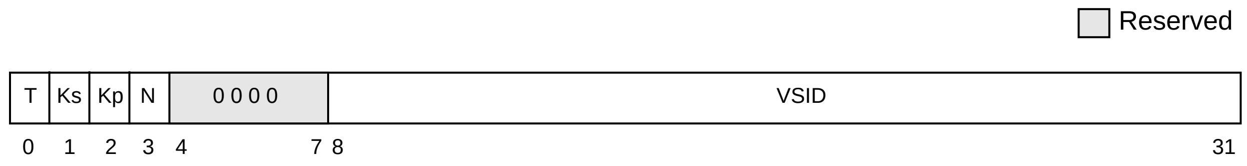


**Figure 7-13. Segment Register Format for Page Address Translation**

Figure 7-14 provides the corresponding bit definitions of the segment register.

**Table 7-14. Segment Register Bit Definition for Page Address Translation**

| Bit | Name | Description |
|---|---|---|
| 0 | T | T = 0 selects this format |
| 1 | Ks | Supervisor-state protection key |
| 2 | Kp | User-state protection key |
| 3 | N | No-execute protection bit |
| 4–7 | — | Reserved |
| 8–31 | VSID | Virtual segment ID |

The Ks and Kp bits partially define the access protection for the pages within the segment. The page protection provided in the OEA is described in Section 7.6.4, “Page Memory Protection.” The virtual segment ID field is used as the high-order bits of the virtual page number (VPN) as shown in Figure 7-12.

The segment registers are programmed with specific instructions that reference the segment registers. However, since the segment registers described here are merely a conceptual model, a processor may implement separate segment registers for instructions and for data, for example. In this case, it is the responsibility of the hardware to maintain the consistency between the multiple sets of segment registers.

The segment register instructions are summarized in Table 7-6. These instructions are privileged in that they are executable only while operating in supervisor mode. See Section 2.3.17, “Synchronization Requirements for Special Registers and for Lookaside Buffers,” for information about the synchronization requirements when modifying the segment registers. See Chapter 8, “Instruction Set,” for more detail on the encodings of these instructions.

#### 7.6.2.2 Page Table Entry (PTE) Definitions

Page table entries (PTEs) are generated and placed in page table in memory by the operating system using the hashing algorithm described in Section 7.7.1.3, “Page Table Hashing Functions.” The OEA defines PTEs that are 64 bits in length. 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 7.6.3, “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 is described in Section 7.6.4, “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.

##### 7.6.2.2.1 PTE Format

Figure 7-14 shows the format of the two words that comprise a PTE for 32-bit implementations.

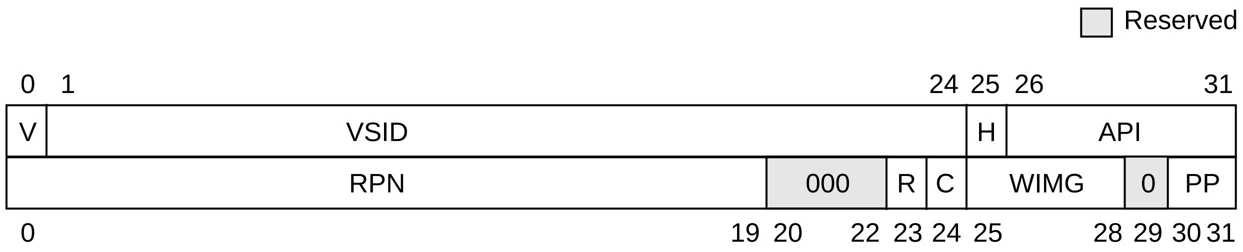


**Figure 7-14. Page Table Entry Format**

Figure 7-15 lists the corresponding bit definitions for each word in a PTE as defined above.

**Table 7-15. PTE Bit Definitions**

| Word | Bit | Name | Description |
|---|---|---|---|
| 0 | 0 | V | Entry valid (V = 1) or invalid (V = 0) |
| | 1–24 | VSID | Virtual segment ID |
| | 25 | H | Hash function identifier |
| | 26–31 | API | Abbreviated page index |
| 1 | 0–19 | RPN | Physical page number |
| | 20–22 | — | Reserved |
| | 23 | R | Referenced bit |
| | 24 | C | Changed bit |
| | 25–28 | WIMG | Memory/cache control bits |
| | 29 | — | Reserved |
| | 30–31 | PP | Page protection bits |

In this case, the PTE contains an abbreviated page index rather than the complete page index field because at least ten of the low-order bits of the page index are used in the hash function to select a PTEG address (PTEG addresses define the location of a PTE). Therefore, these ten low-order bits are not repeated in the PTEs of that PTEG.

## 7.6.3 Page History Recording

Referenced (R) and changed (C) bits in each PTE 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. Referenced and changed recording is performed only for accesses made with page address translation and not for translations made with the BAT mechanism or for accesses that correspond to direct-store (T = 1) segments. 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 7-16.

**Table 7-16. Table Search Operations to Update History Bits**

| R and C Bits | Processor Action |
|---|---|
| 00 | Read: Table search operation to update R<br>Write: Table search operation to update R and C |
| 01 | Combination doesn’t occur |
| 10 | Read: No special action<br>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 7.7.3, “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 as if address translation were disabled (real addressing mode address).

#### 7.6.3.1 Referenced Bit

The referenced bit for each virtual 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 an **stwcx.** instruction when no store is performed
- Accesses that cause exceptions and are not completed

#### 7.6.3.2 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, it is not updated. 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.** instruction is allowed by the memory protection mechanism but a store operation is not performed.
- The execution of a **stswx** instruction is allowed by the memory protection mechanism, but a store operation is not allowed because the specified length is zero.
- A **dcba** or **dcbi** instruction is executed.

No other cases cause the C bit to be set.

#### 7.6.3.3 Scenarios for Referenced and Changed Bit Recording

This section summarizes the OEA-defined model that automatically maintains referenced and changed bits 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 0b0010 (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 that this software should be contained in the area of memory reserved for implementation-specific use and should be invisible to the operating system.

Table 7-17 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 that in the table, load operations include those generated by load instructions, by the **eciwx** instruction, and by the cache management instructions that are treated as loads. 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.

Table 7-17. Model for Guaranteed R and C Bit Settings

| Priority | Scenario | Causes Setting of R Bit | Causes Setting of C Bit |
|---|---|---|---|
| 1 | No-execute protection violation | No | No |
| 2 | Page protection violation | Maybe | No |
| 3 | Speculative instruction fetch or load operation | Maybe | No |
| 4 | Speculative [1] 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 [2] | Maybe[2] |
| 5 | All other speculative[1] store operations | Maybe[2] | No |
| 6 | Zero-length load (**lswx**) | Maybe | No |
| 7 | Zero-length store (**stswx**) | Maybe[2] | Maybe[2] |
| 8 | Store conditional (**stwcx.**) that does not store | Maybe[2] | Maybe[2] |
| 9 | In-order instruction fetch | Yes [3] | No |
| 10 | Load instruction or **eciwx** | Yes | No |
| 11 | Store instruction, **ecowx**, **dcbz**, or **dcba** [4] instruction | Yes | Yes |
| 12 | **icbi**, **dcbt**, **dcbtst**, **dcbst**, or **dcbf** instruction | Maybe | No |
| 13 | **dcbi** instruction | Maybe[2] | Maybe[2] |

1 Refered to as out-of-order in the architecture.

2 If C is set, R is guaranteed to also be set.

3 This includes the case in which the instruction was fetched speculatively and R was not set.

4 For a **dcba** instruction that does not modify the target block, it is possible that neither bit is set.

#### 7.6.3.4 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.

### 7.6.4 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 block and page address translation mechanisms 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 specific information about block address translation, refer to Section 7.5.4, “Block Memory Protection.”

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
- Ks and Kp, the supervisor and user key bits, which define the key for the page
- The PP bits, which define the access options for the page

The key bits (Ks and Kp) and the PP bits are located as follows for page address translation:

- Ks and Kp 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 Ks bit is used and Kp is ignored
  - For user accesses (MSR[PR] = 1), the Kp bit is used and Ks is ignored

  That is, key = (Kp & MSR[PR]) | (Ks & ¬MSR[PR])
- The selected key is used with the PP bits to determine if instruction fetching, load access, or store access is allowed.

Table 7-18 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 7-18. 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 |

[1] Ks or Kp 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 7-19 and as a flow diagram in Figure 7-17. 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 7-19. 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 Ks, Kp, and PP bits is allowed. One example is if the Ks and Kp bits are programmed so that the value of the key bit for Table 7-18 directly matches the MSR[PR] bit for the access. In this case, the encoding of Ks = 0 and Kp = 1 is used for the PTE, and the PP bits then enforce the protection options shown in Table 7-20.

**Table 7-20. Access Protection Encoding of PP Bits for Ks = 0 and Kp = 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 | √ | √ |
| 01 | Supervisor-write-only | √ | Violation | √ | √ |
| 10 | Both user/supervisor | √ | √ | √ | √ |
| 11 | Both read-only | √ | Violation | √ | Violation |

However, if the setting Ks = 1 is used, supervisor accesses are treated as user reads and writes with respect to Table 7-20. Likewise, if the setting Kp = 0 is used, user accesses to the page are treated as supervisor accesses in relation to Table 7-20. 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, however, that only supervisor programs are allowed to modify the key bits for the segment descriptor. Access to the segment registers is privileged.

When the memory protection mechanism prohibits a reference, the flow of events is similar to that for a memory protection violation occurring with the block protection mechanism. As shown in Figure 7-15, 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[4] is set, or
  - an ISI exception is generated and SRR1[3] is set if the segment is designated as no-execute.

The only difference between the flow shown in Figure 7-15 and that of the block memory protection violation is the ISI exception that can be caused by an attempt to fetch an instruction from a segment that has been designated as no-execute (N bit set in the segment descriptor). See Chapter 6, “Exceptions,” for more information about these exceptions.

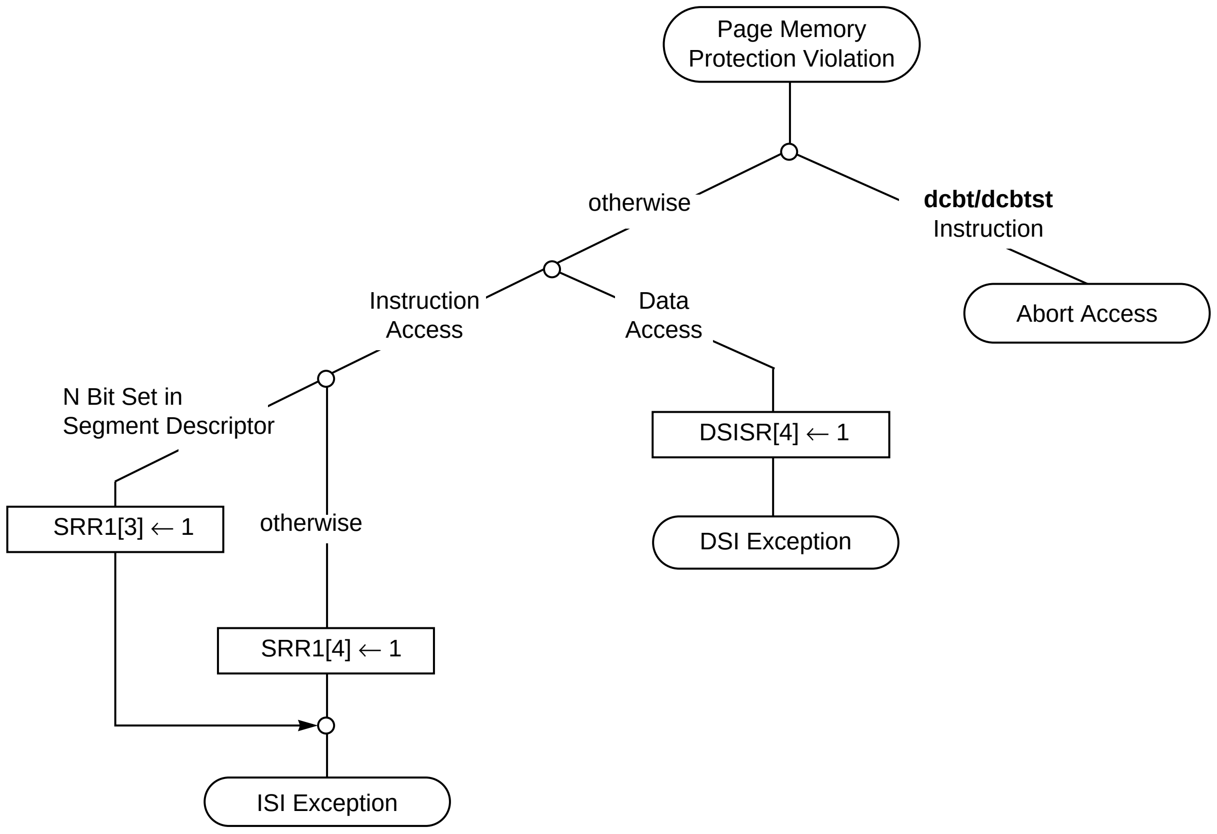
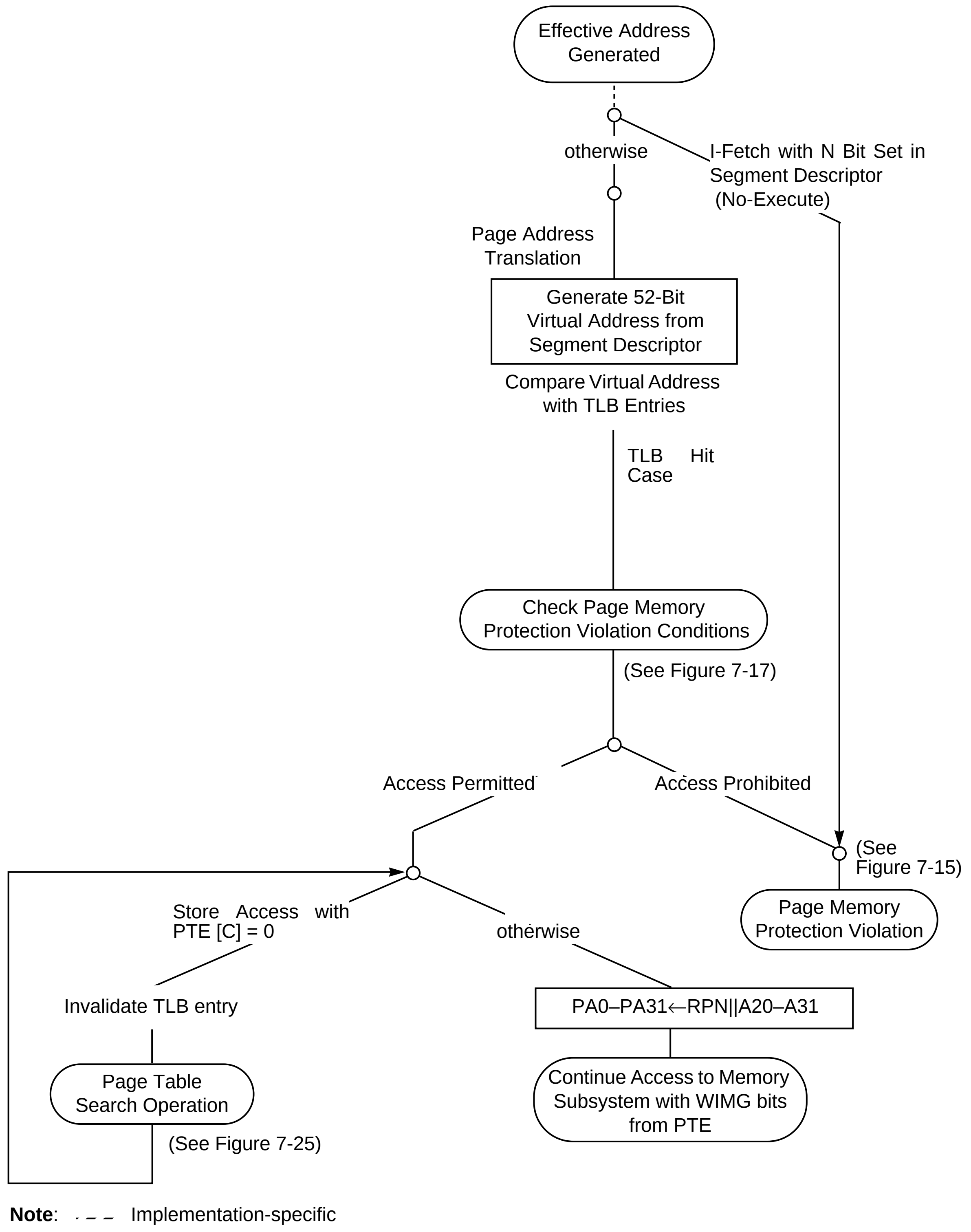


**Figure 7-15. Memory Protection Violation Flow for Pages**

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).

## 7.6.5 Page Address Translation Summary

Figure 7-16 provides the detailed flow for the page address translation mechanism. 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 7.7.2, "Page Table Search Operation." The checking of memory protection violation conditions for page address translation is shown in Figure 7-17. The 'Invalidate TLB Entry' box shown in Figure 7-16 is marked as implementation-specific as this level of detail for TLBs (and the existence of TLBs) is not dictated by the architecture. Note that the figure does not show the detection of all exception conditions shown in Table 7-4 and Figure 7-5; the flow for many of these exceptions is implementation-specific.



Effective Address Generated
otherwise
I-Fetch with N Bit Set in Segment Descriptor (No-Execute)
Page Address Translation
Generate 52-Bit Virtual Address from Segment Descriptor
Compare Virtual Address with TLB Entries
TLB Hit Case
Check Page Memory Protection Violation Conditions
(See Figure 7-17)
Access Permitted
Access Prohibited
(See Figure 7-15)
Page Memory Protection Violation
Store Access with PTE [C] = 0
otherwise
Invalidate TLB entry
PA0–PA31←RPN||A20–A31
Page Table Search Operation
Continue Access to Memory Subsystem with WIMG bits from PTE
(See Figure 7-25)
Note: – – – Implementation-specific

**Figure 7-16. Page Address Translation Flow—TLB Hit**

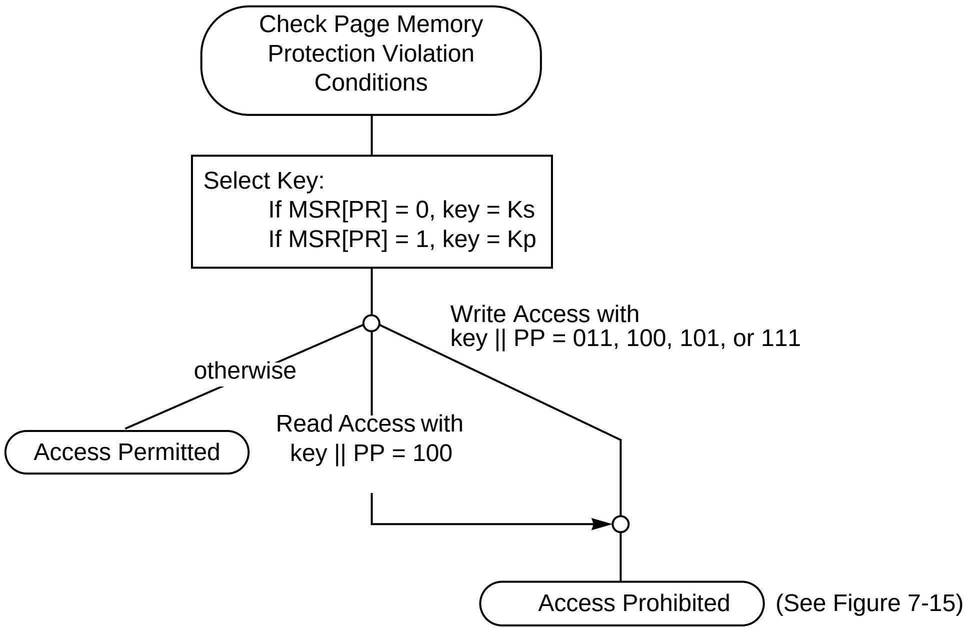

Check Page Memory Protection Violation Conditions
Select Key:
If MSR[PR] = 0, key = Ks
If MSR[PR] = 1, key = Kp
Write Access with key || PP = 011, 100, 101, or 111
otherwise
Read Access with key || PP = 100
Access Permitted
Access Prohibited
(See Figure 7-15)

**Figure 7-17. Page Memory Protection Violation Conditions for Page Address Translation**

## 7.7 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 system in main memory.

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.

### 7.7.1 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 0b0010.

The page table contains a number of page table entry groups (PTEGs). For 32-bit implementations, a PTEG contains eight PTEs of eight bytes each; therefore, each PTEG is 64 bytes long. PTEG addresses are entry points for table search operations. Figure 7-18 shows two PTEG addresses (PTEGaddr1 and PTEGaddr2) where a given PTE may reside.

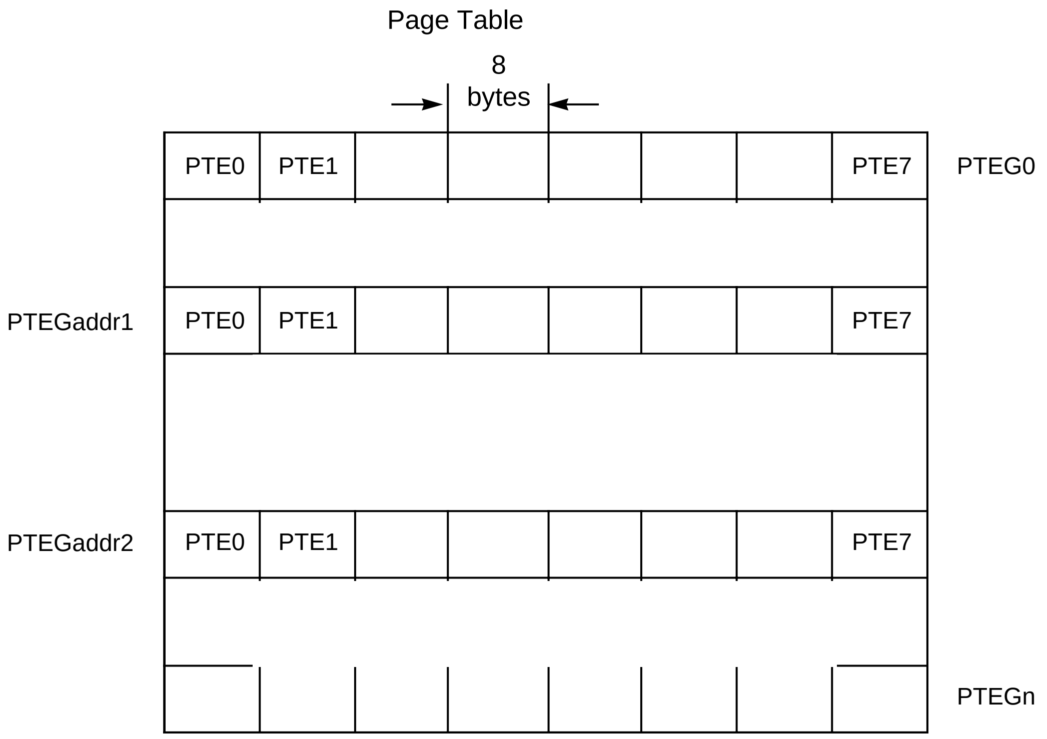


**Figure 7-18. Page Table Definitions**

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, however, that 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 7.7.1.6, "Page Table Structure Examples," illustrates how PTEs map into the 16 possible locations as primary and secondary PTEs.

### 7.7.1.1 SDR1 Register Definitions

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 2.3.17, "Synchronization Requirements for Special Registers and for Lookaside Buffers." The format of SDR1 is shown in the following sections.

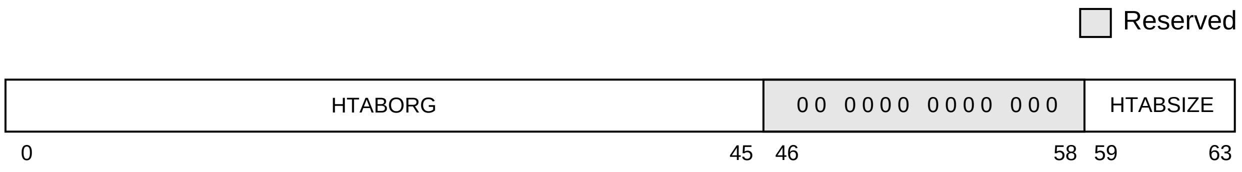


Figure 7-19 shows the format of the SDR1 register.

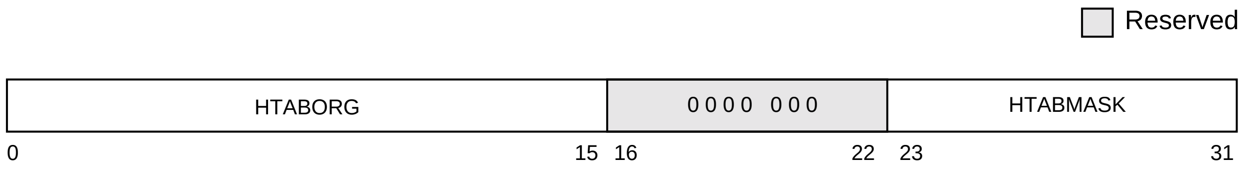


**Figure 7-19. SDR1 Register Format**

Bit settings are described in Figure 7-21.

**Table 7-21. SDR1 Register Bit Settings**

| Bits | Name | Description |
|---|---|---|
| 0–15 | HTABORG | Physical base address of page table |
| 16–22 | — | Reserved |
| 23–31 | HTABMASK | Mask for page table address |

The HTABORG field in SDR1 contains the high-order 16 bits of the 32-bit physical address of the page table. Therefore, the beginning of the page table lies on a $2^{16}$ byte (64 Kbyte) boundary at a minimum. If the processor does not support 32 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 $16 \le n \le 25$. The HTABMASK field in SDR1 contains a mask value that determines how many bits from the output of the hashing function are used as the page table index. This mask must be of the form 0b00...011...1 (a string of 0 bits followed by a string of 1 bits). 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 HTABMASK determine how many additional bits (beyond the minimum of 10) from the hash are used in the index; the HTABORG field must have the same number of low-order bits equal to 0 as the HTABMASK field has low-order bits equal to 1.

For example, suppose that the page table is 16,384 ($2^{14}$) 64-byte PTEGs, for a total size of $2^{20}$ bytes (1 Mbyte). A 14-bit index is required. Ten bits are provided from the hash to start with, so 4 additional bits from the hash must be selected. Thus the value in HTABMASK must be 15 and the value in HTABORG must have its low-order 4 bits (SDR1[12–15]) equal to 0. This means that the page table must begin on a $2^{<4 + 10 + 6>} = 2^{20}$ = 1-Mbyte boundary.

### 7.7.1.2 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.

In a 32-bit implementation, the minimum size for a page table is 64 Kbytes ($2^{10}$ PTEGs of 64 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 7-22 shows some example sizes for total main memory in a 32-bit system. The recommended minimum page table size for these example memory sizes are then outlined, along with their corresponding HTABORG and HTABMASK settings in SDR1. Note that systems with less than 8 Mbytes of main memory may be designed with 32-bit processors, but the minimum amount of memory that can be used for the page tables in these cases is 64 Kbytes.

**Table 7-22. 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 7–15) | HTABMASK |
| 8 Mbytes ($2^{23}$) | 64 Kbytes ($2^{16}$) | $2^{13}$ | $2^{10}$ | x xxxx xxxx | 0 0000 0000 |
| 16 Mbytes ($2^{24}$) | 128 Kbytes ($2^{17}$) | $2^{14}$ | $2^{11}$ | x xxxx xxx0 | 0 0000 0001 |
| 32 Mbytes ($2^{25}$) | 256 Kbytes ($2^{18}$) | $2^{15}$ | $2^{12}$ | x xxxx xx00 | 0 0000 0011 |
| 64 Mbytes ($2^{26}$) | 512 Kbytes ($2^{19}$) | $2^{16}$ | $2^{13}$ | x xxxx x000 | 0 0000 0111 |
| 128 Mbytes ($2^{27}$) | 1 Mbyte ($2^{20}$) | $2^{17}$ | $2^{14}$ | x xxxx 0000 | 0 0000 1111 |
| 256 Mbytes ($2^{28}$) | 2 Mbytes ($2^{21}$) | $2^{18}$ | $2^{15}$ | x xxx0 0000 | 0 0001 1111 |
| 512 Mbytes ($2^{29}$) | 4 Mbytes ($2^{22}$) | $2^{19}$ | $2^{16}$ | x xx00 0000 | 0 0011 1111 |
| 1 Gbytes ($2^{30}$) | 8 Mbytes ($2^{23}$) | $2^{20}$ | $2^{17}$ | x x000 0000 | 0 0111 1111 |
| 2 Gbytes ($2^{31}$) | 16 Mbytes ($2^{24}$) | $2^{21}$ | $2^{18}$ | x 0000 0000 | 0 1111 1111 |
| 4 Gbytes ($2^{32}$) | 32 Mbytes ($2^{25}$) | $2^{22}$ | $2^{19}$ | 0 0000 0000 | 1 1111 1111 |

As an example, if the physical memory size is $2^{29}$ bytes (512 Mbyte), then there are $2^{29} – 2^{12}$ (4 Kbyte page size) = $2^{17}$ (128 Kbyte) total page frames. If this number of page frames is divided by 2, the resultant minimum recommended page table size is $2^{16}$ PTEGs, or $2^{22}$ bytes (4 Mbytes) of memory for the page tables.

### 7.7.1.3 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 that 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 HTABMASK field determines how many of the high-order hash value bits are masked and how many are used in the generation of the physical address of the PTEG.

Figure 7-20 depicts the hashing functions defined by the OEA for 32-bit implementations. The inputs to the primary hashing function are the low-order 19 bits of the VSID field of the selected segment register (bits 5–23 of the 52-bit virtual address), and the page index field of the effective address (bits 24–39 of the virtual address) concatenated with three zero high-order bits. The XOR of these two values generates the output of the primary hashing function (hash value 1).

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.

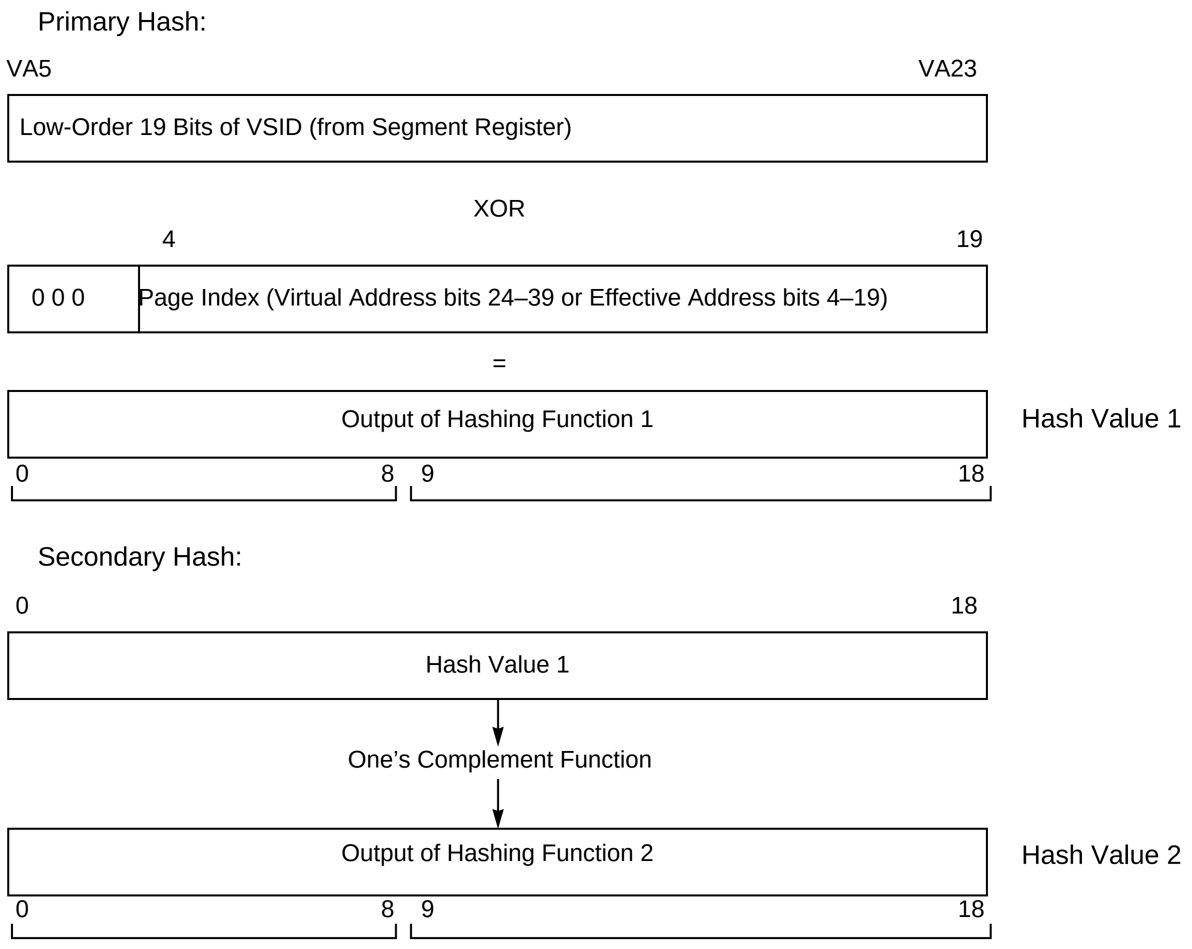

Primary Hash:
VA5
VA23
Low-Order 19 Bits of VSID (from Segment Register)
XOR
4
19
0 0 0
Page Index (Virtual Address bits 24–39 or Effective Address bits 4–19)
=
Output of Hashing Function 1
Hash Value 1
0
8
9
18
Secondary Hash:
0
18
Hash Value 1
One’s Complement Function
Output of Hashing Function 2
Hash Value 2
0
8
9
18

**Figure 7-20. Hashing Functions for Page Tables**

### 7.7.1.4 Page Table Addresses

The following sections illustrate 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 7.7.1.3, “Page Table Hashing Functions.”

Note that unless all accesses to be performed by the processor can be translated by the BAT mechanism 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 7.3.1.2, “Predefined Physical Memory Locations”).

For 32-bit implementations, the base address of the page table is defined by the high-order bits of SDR1[HTABORG].

Effectively, bits 7–15 of the PTEG address are derived from the masking of the high-order bits of the hash value (as defined by SDR1[HTABMASK]) concatenated with (implemented as an OR function) the high-order bits of SDR1[HTABORG] as defined by HTABMASK. Bits 16–25 of the PTEG address are the 10 low-order bits of the hash value, and bits 26–31 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 7-21 provides a graphical description of the generation of the PTEG addresses for 32-bit implementations.

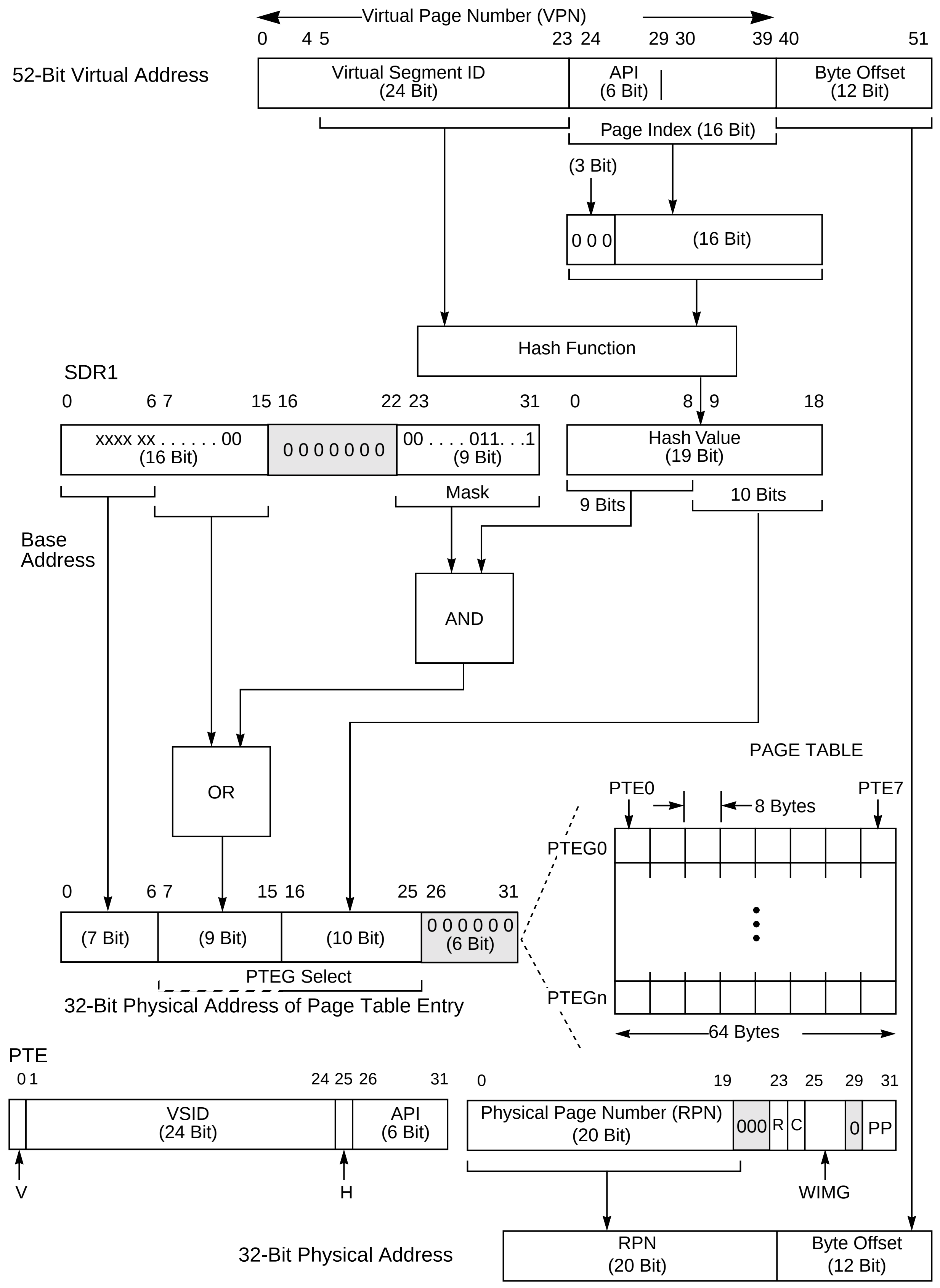


**Figure 7-21. Generation of Addresses for Page Tables**

#### 7.7.1.5 Page Table Structure Summary

In the process of searching for a PTE, the processor interprets the values read from memory as described in Section 7.6.2.2, “Page Table Entry (PTE) Definitions.” 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 7-21 and Figure 7-22, 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 that 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.

### 7.7.1.6 Page Table Structure Examples

Figure 7-22 shows the structure of an example page table. The page table base address is defined by SDR1[HTABORG] concatenated with 16 zero bits. In this example, the address is identified by bits 0–13 in SDR1[HTABORG]; note that bits 14 and 15 of HTABORG must be zero because the low-order two bits of HTABMASK are ones. The addresses for individual PTEGs within this page table are then defined by bits 14–25 as an offset from bits 0–13 of this base address. Thus, the size of the page table is defined as 4096 PTEGs.

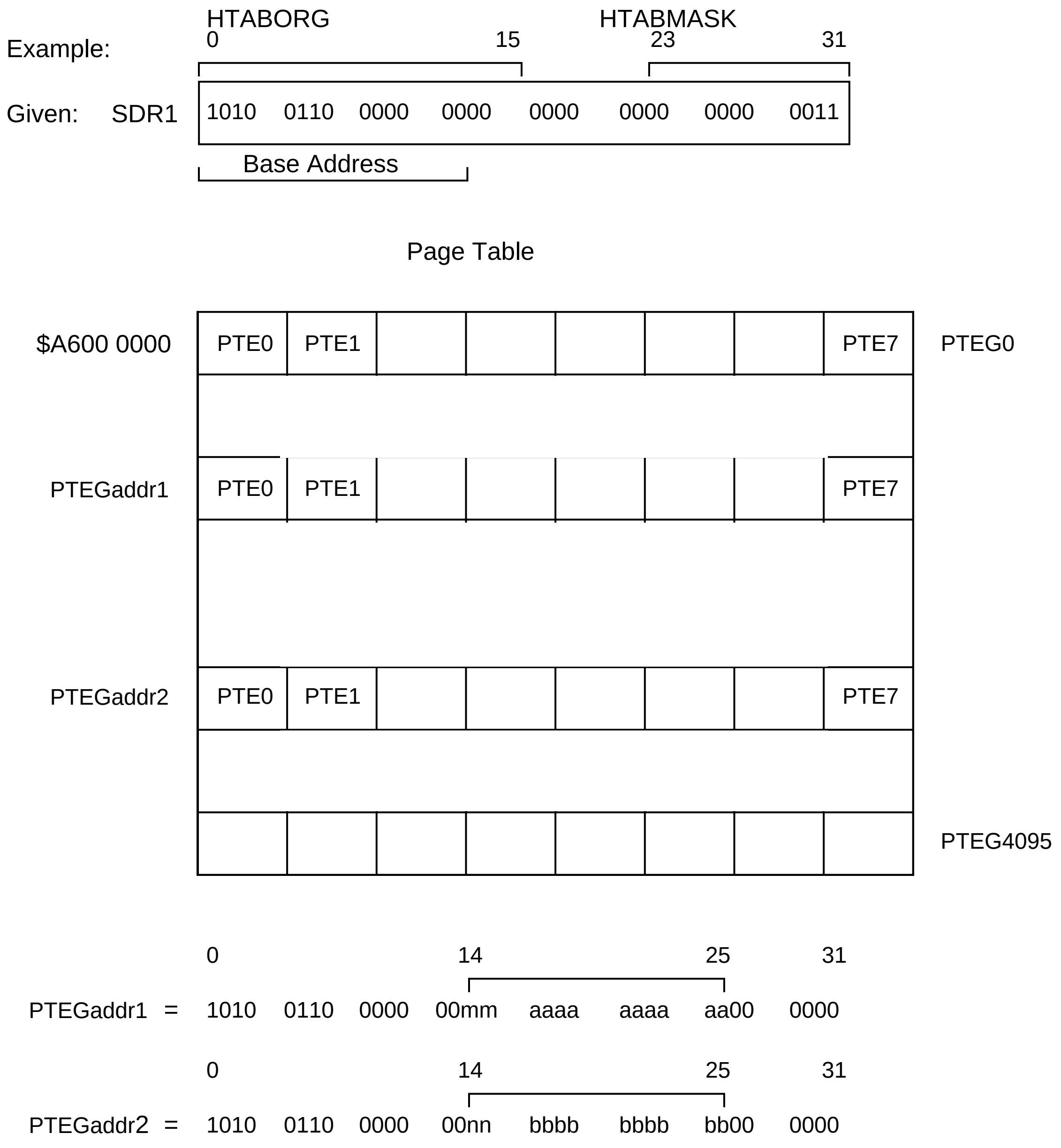


**Figure 7-22. Example Page Table Structure**

Two example PTEG addresses are shown in the figure as PTEGaddr1 and PTEGaddr2. Bits 14–25 of each PTEG address in this example page table are derived from the output of the hashing function (bits 26–31 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 two bits are generated from bits 7–8 of the output of the hashing function, logically ORed with bits 14–15 of the HTABORG field (which must be zero). If bits 14–25 of PTEGaddr1 were derived by using the primary hashing function, then PTEGaddr2 corresponds to the secondary PTEG.

Note, however, that bits 14–25 in PTEGaddr2 can also be derived from a combination of effective address bits, segment register 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 register bits), it may also correspond to the secondary PTEG for a different effective address (and segment register 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.

#### 7.7.1.7 PTEG Address Mapping Examples

This section contains two examples of an effective address and how its address translation (the PTE) maps into the primary PTEG in physical memory. The examples illustrate 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.

Figure 7-23 shows an example of PTEG address generation for a 32-bit implementation. In the example, the value in SDR1 defines a page table at address 0x0F98_0000 that contains 8192 PTEGs. The example effective address selects segment register 0 (SR0) with the highest order four bits. The contents of SR0 are then used along with bits 4–31 of the effective address to create the 52-bit virtual address.

To generate the address of the primary PTEG, bits 5–23, and bits 24–39 of the virtual address are then used as inputs into the primary hashing function (XOR) to generate hash value 1. The low-order 13 bits of hash value 1 are then concatenated with the high-order 13 bits of HTABORG and with six low-order 0 bits, defining the address of the primary PTEG (0x0F9F_F980).

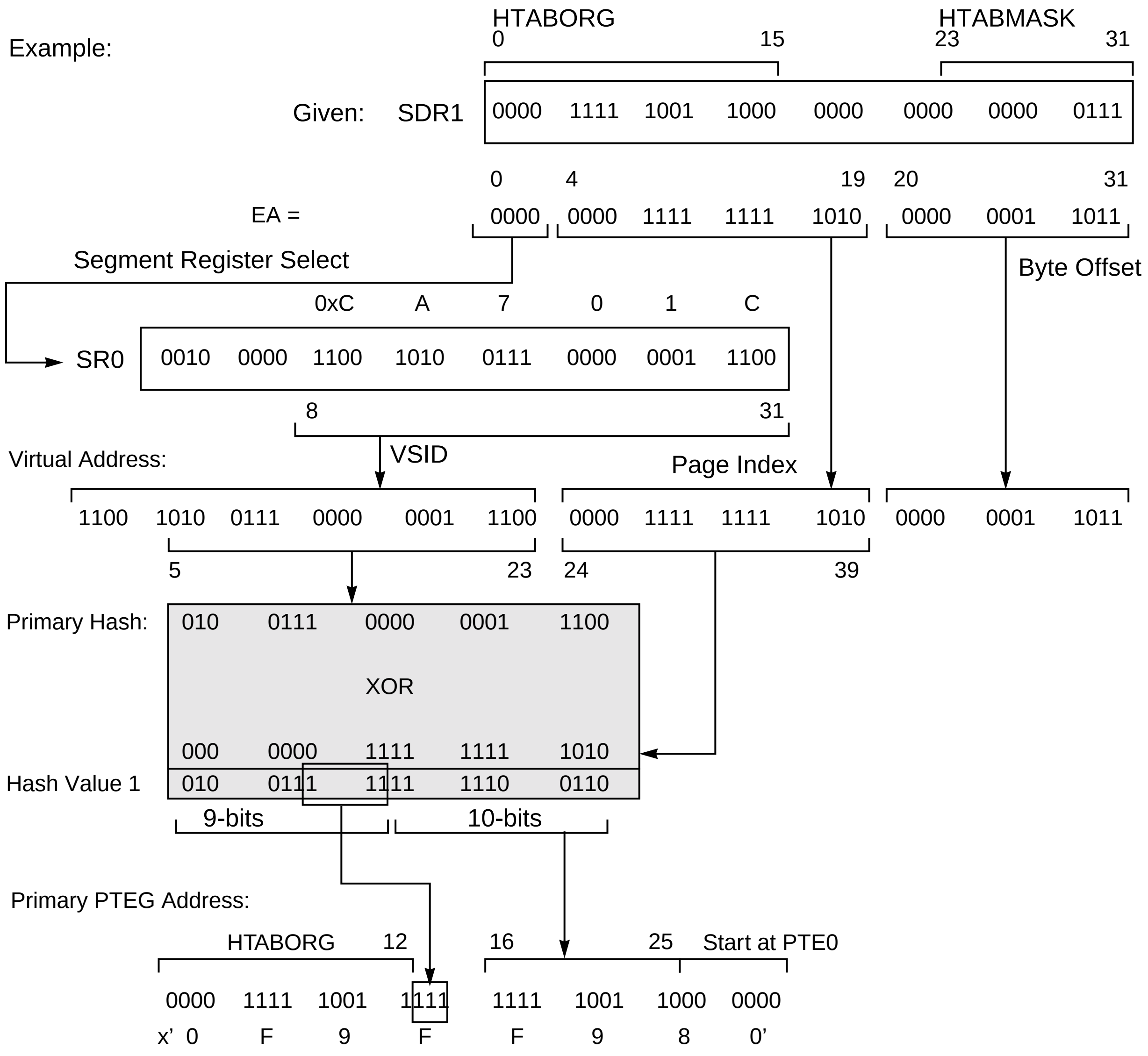


**Figure 7-23. Example Primary PTEG Address Generation**

Figure 7-24 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 13 bits of hash value 2 are then ORed with the high-order 16 bits of HTABORG (bits 13–15 should be zero), and concatenated with six low-order 0 bits, defining the address of the secondary PTEG (0x0F98_0640).

As described in Figure 7-21, the 10 low-order bits of the page index field are always used in the generation of a PTEG address (through the hashing function) for a 32-bit implementation. This is why only the 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 10 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 6 bits (the API field as defined for 32-bit implementations) 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.

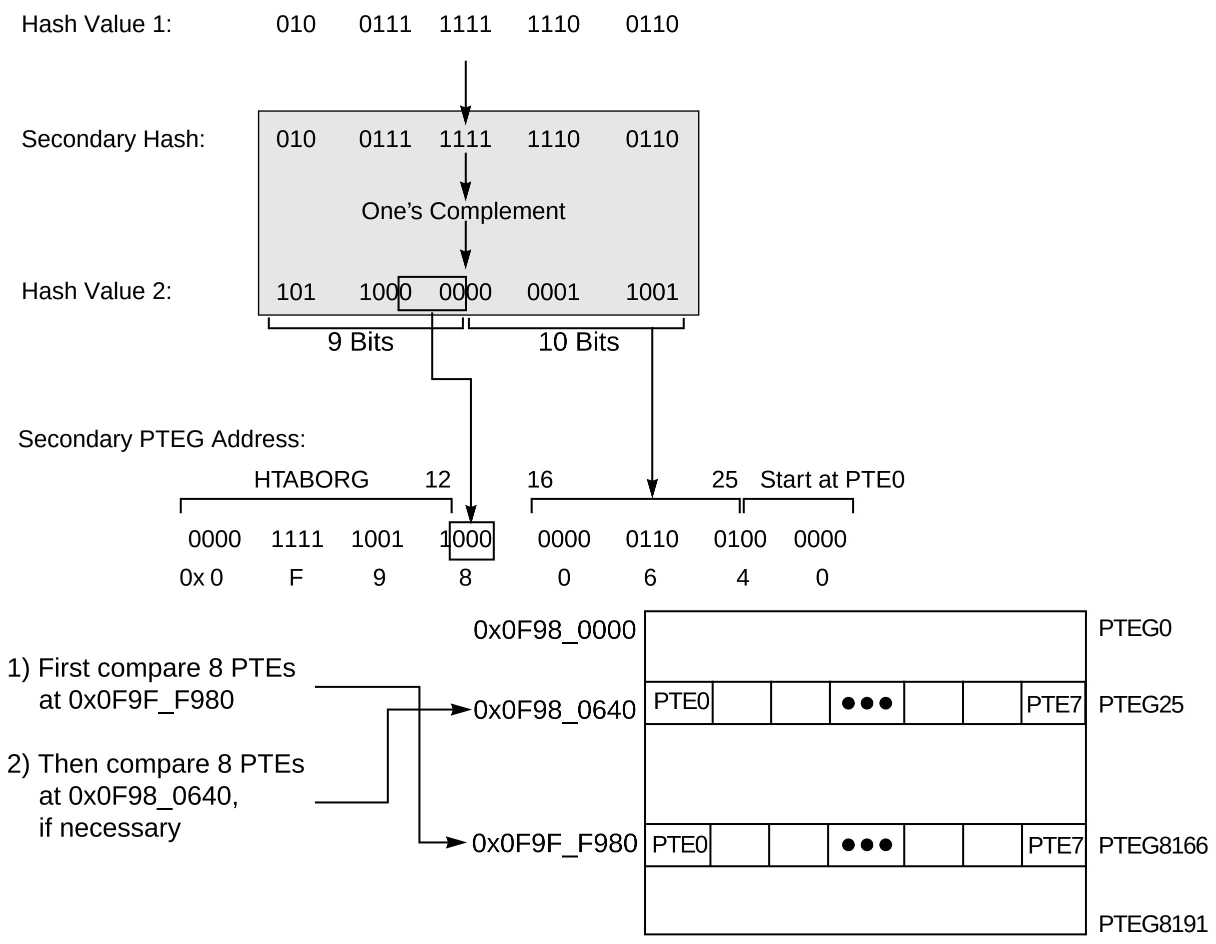


**Figure 7-24. Example Secondary PTEG Address Generation**

Note that 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 7.7.1.6, “Page Table Structure Examples”), but in this example, bits 24–26 of the page index field of the virtual address are not used to generate the PTEG address. Therefore, any of the eight combinations of these bits will map to the same primary PTEG address. (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 register 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.

### 7.7.2 Page Table Search Operation

An outline of the page table search process performed by a 32-bit implementation is as follows:

1. The 32-bit physical addresses of the primary and secondary PTEGs are generated as described in Section 7.7.1.4, “Page Table Addresses.”
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 0b001. 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[0–23]
   — PTE[API] = VA[24–29]
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 (that is, as unguarded cacheable operations in which coherency is required).

### 7.7.2.1 Flow for Page Table Search Operation

Figure 7-25 provides a detailed flow diagram of a page table search operation. Note that 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. Also, Figure 7-25 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 7-17.

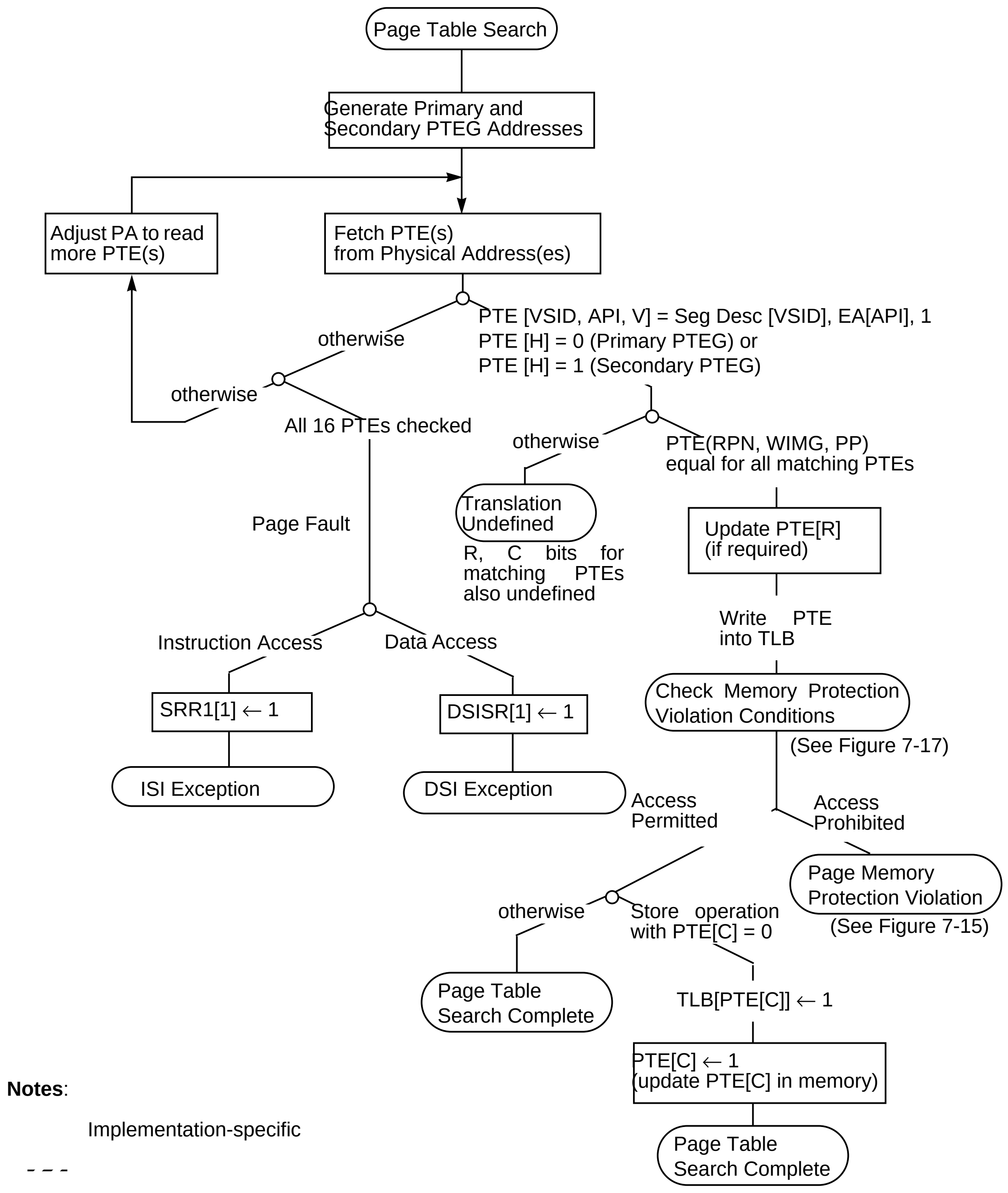

Page Table Search
Generate Primary and Secondary PTEG Addresses
Adjust PA to read more PTE(s)
Fetch PTE(s) from Physical Address(es)
PTE [VSID, API, V] = Seg Desc [VSID], EA[API], 1
PTE [H] = 0 (Primary PTEG) or
PTE [H] = 1 (Secondary PTEG)
otherwise
otherwise
All 16 PTEs checked
otherwise
PTE(RPN, WIMG, PP) equal for all matching PTEs
Translation Undefined
R, C bits for matching PTEs also undefined
Update PTE[R] (if required)
Page Fault
Write PTE into TLB
Instruction Access
Data Access
SRR1[1] ← 1
DSISR[1] ← 1
Check Memory Protection Violation Conditions
(See Figure 7-17)
ISI Exception
DSI Exception
Access Permitted
Access Prohibited
Page Memory Protection Violation
(See Figure 7-15)
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
Notes:
Implementation-specific
– – –

**Figure 7-25. Page Table Search Flow**

## 7.7.3 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
- 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 E, “Synchronization Programming Examples,” for more information about the use of the reservation instructions (such as the **lwarx** and **stwcx.** instructions) to perform software locking.

TLBs are implemented 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 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 to ensure proper synchronization with the other processors. Note that a **sync** 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 shown as a single line in the example are assumed to be performed with an atomic store instruction. Appropriate modifications must be made to these examples if this assumption is not satisfied.

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.

For a complete list of the synchronization requirements for executing the MMU instructions, see Section 2.3.17, “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. Page tables are modified by deleting and adding a page table entry.

#### 7.7.3.1 Adding a Page Table Entry

Adding a page table entry requires only a lock on the PTE in a multiprocessor system. The first bytes in the PTE are then written (this example assumes the old valid bit was cleared), the **eieio** instruction orders the update, and then the second update can be made. A **sync** instruction ensures that the updates have been made to memory.

```
lock(PTE)
PTE[RPN,R,C,WIMG,PP] ← new values
eieio                                        /* order 1st PTE update befor 2nd
PTE[VSID,H,API,V] ← new values (V = 1)
sync                                         /* ensure updates completed
unlock(PTE)
```

#### 7.7.3.2 Deleting a Page Table Entry

In this example, the entry is locked, marked invalid, invalidated in the TLB, and unlocked.

Again, note that the **tlbsync** and the **sync** instruction that follows it are only required if consistency must be maintained with other processors in a multiprocessor system (and the software is to be used in a multiprocessor environment).

```
lock(PTE)
PTE[V] ← 0      /* (other fields don’t matter)
sync            /* ensure update completed
tlbie(old_EA)   /* invalidate old translation
eieio           /* order tlbie before tlbsync
tlbsync         /* ensure tlbie completed on all processors
sync            /* ensure tlbsync completed
unlock(PTE)
```

### 7.7.4 Segment Register Updates

Synchronization requirements for using the move to segment register instructions are described in Section 2.3.17, “Synchronization Requirements for Special Registers and for Lookaside Buffers.”

## 7.8 Direct-Store Segment Address Translation

As described for memory segments, all accesses generated by the processor (with translation enabled) that do not map to a BAT area, map to a segment descriptor. If T = 1 for the selected segment descriptor, the access maps to the direct-store interface, invoking a specific bus protocol for accessing I/O devices.

Direct-store segments are provided for POWER compatibility. As the direct-store interface is present only for compatibility with existing I/O devices that used this interface and the direct-store interface protocol is not optimized for performance, its use is discouraged. This functionality is considered optional (to allow for those earlier devices that implemented it). However, future devices are not likely to support it. Thus, software should not depend on its results and new software should not use it. Applications that require low-latency load/store access to external address space should use memory-mapped I/O, rather than the direct-store interface.

## 7.8.1 Segment Descriptors for Direct-Store Segments

The format of many of the fields in the segment descriptors depends on the value of the T bit. In 32-bit implementations, the segment descriptors reside in one of 16 on-chip segment registers. Figure 7-26 shows the register format for the segment registers when the T bit is set.

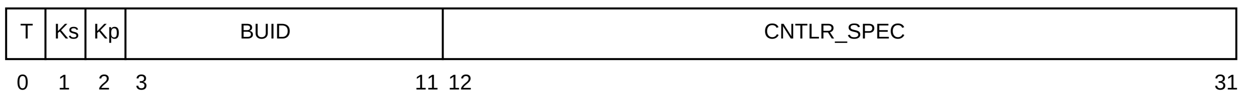


**Figure 7-26. Segment Register Format for Direct-Store Segments**

Table 7-23 shows the bit definitions for the segment registers when the T bit is set for 32-bit implementations.

**Table 7-23. Segment Register Bit Definitions for Direct-Store Segments**

| Bit | Name | Description |
|---|---|---|
| 0 | T | T = 1 selects this format. |
| 1 | Ks | Supervisor-state protection key |
| 2 | Kp | User-state protection key |
| 3–11 | BUID | Bus unit ID |
| 12–31 | CNTLR_SPEC | Device-specific data for I/O controller |

## 7.8.2 Direct-Store Segment Accesses

When the address translation process determines that the segment descriptor has T = 1, direct-store segment address translation is selected; no reference is made to the page tables and neither the referenced or changed bits are updated. These accesses are performed as if the WIMG bits were 0b0101; that is, caching is inhibited, the accesses bypass the cache, hardware-enforced coherency is not required, and the accesses are considered guarded.

The specific protocol invoked to perform these accesses involves the transfer of address and data information; however, the OEA does not define the exact hardware protocol used for direct-store accesses. Some instructions may cause multiple address/data transactions to occur on the bus. In this case, the address for each transaction is handled individually with respect to the MMU.

The following describes the data that is typically sent to the memory controller by processors that implement the direct-store function:

- One of the K*x* bits (Ks or Kp) is selected to be the key as follows:
  - For supervisor accesses (MSR[PR] = 0), the Ks bit is used and Kp is ignored.
  - For user accesses (MSR[PR] = 1), the Kp bit is used and Ks is ignored.
- An implementation-dependent portion of the segment descriptor.
- An implementation-dependent portion of the effective address.

### 7.8.3 Direct-Store Segment Protection

Page-level memory protection as described in Section 7.6.4, “Page Memory Protection,” is not provided for direct-store segments. The appropriate key bit (Ks or Kp) from the segment descriptor is sent to the memory controller, and the memory controller implements any protection required. Frequently, no such mechanism is provided; the fact that a direct-store segment is mapped into the address space of a process may be regarded as sufficient authority to access the segment.

### 7.8.4 Instructions Not Supported in Direct-Store Segments

The following instructions are not supported at all and cause either a DSI exception or boundedly-undefined results when issued with an effective address that selects a segment descriptor that has T = 1:

- **lwarx**
- **stwcx.**
- **eciwx**
- **ecowx**

### 7.8.5 Instructions with No Effect in Direct-Store Segments

The following instructions are executed as no-ops when issued with an effective address that selects a segment where T = 1:

- **dcba**
- **dcbt**
- **dcbtst**

- **dcbf**
- **dcbi**
- **dcbst**
- **dcbz**
- **icbi**

## 7.8.6 Direct-Store Segment Translation Summary Flow

Figure 7-27 shows the flow used by the MMU when direct-store segment address translation is selected. This figure expands the Direct-Store Segment Translation stub found in Figure 7-4 for both instruction and data accesses. In the case of a floating-point load or store operation to a direct-store segment, it is implementation-specific whether the alignment exception occurs. In the case of an **eciwx**, **ecowx**, **lwarx**, or **stwcx.** instruction, the implementation either sets the DSISR as shown and causes the DSI exception, or causes boundedly-undefined results.

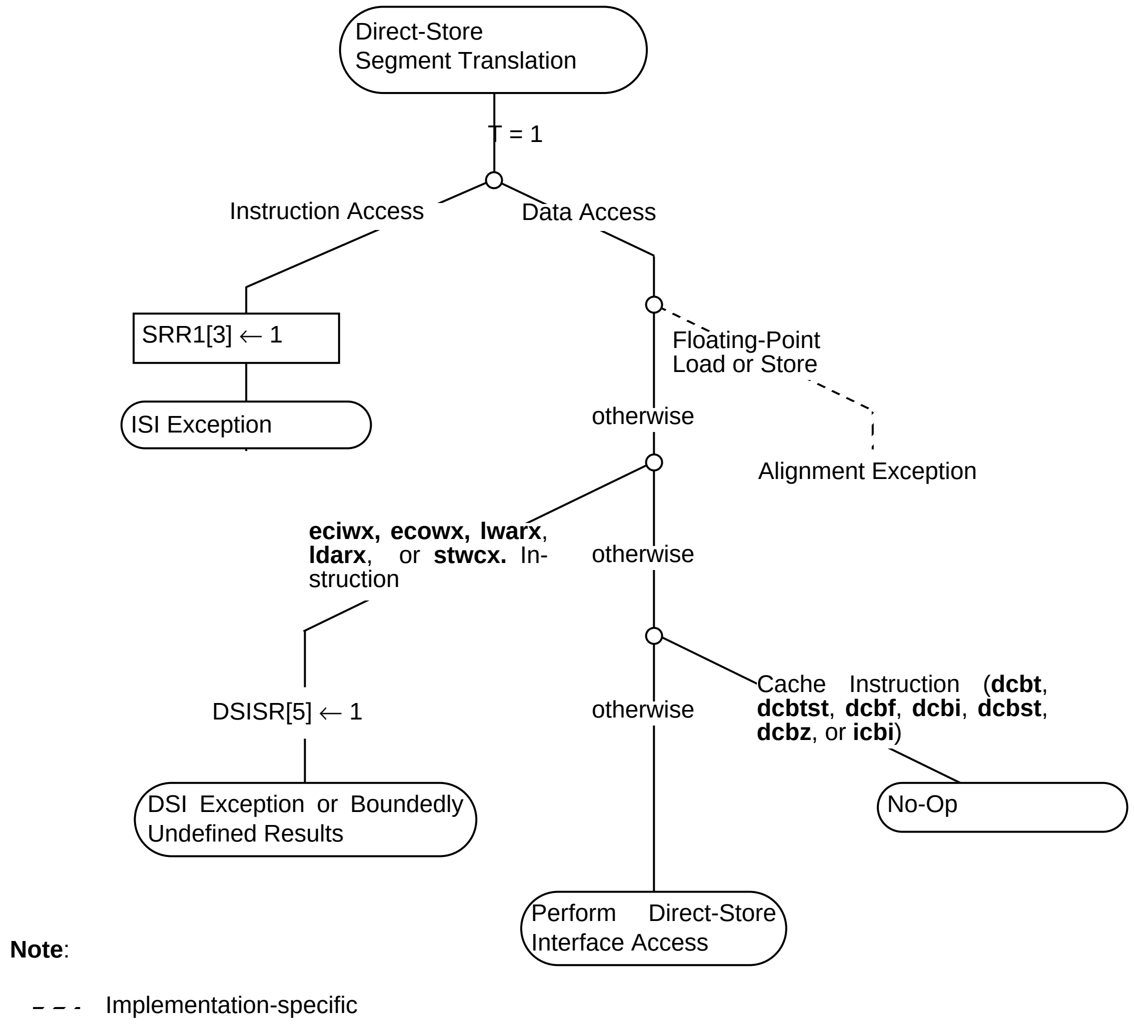

Direct-Store Segment Translation
T = 1
Instruction Access
Data Access
SRR1[3] ← 1
ISI Exception
Floating-Point Load or Store
otherwise
Alignment Exception
**eciwx, ecowx, lwarx, ldarx**, or **stwcx.** Instruction
otherwise
DSISR[5] ← 1
DSI Exception or Boundedly Undefined Results
Cache Instruction (**dcbt**, **dcbtst**, **dcbf**, **dcbi**, **dcbst**, **dcbz**, or **icbi**)
otherwise
No-Op
Perform Direct-Store Interface Access
**Note**:
– – - Implementation-specific

**Figure 7-27. Direct-Store Segment Translation Flow**

# Chapter 8
# Instruction Set

This chapter lists instructions in alphabetical order by mnemonic. Each entry includes the instruction formats and a legend that shows the level or levels of the architecture in which the instruction may be found (UISA, VEA, or OEA); the privilege level (user or supervisor); and the instruction formats. The format diagrams show all valid combinations of instruction fields. The legend also indicates if the instruction is 32-bit, and whether it is optional. Chapter 4, "Addressing Modes and Instruction Set Summary," gives a higher-level description of the instruction set.

## 8.1 Instruction Formats

Instructions are 4 bytes long and word-aligned, so when instruction addresses are presented to the processor (as in branch instructions) the 2 low-order bits are ignored. Similarly, whenever the processor develops an instruction address, its 2 low-order bits are zero. Bits 0–5 always specify the primary opcode. Many instructions also have an extended opcode. The remaining bits contain one or more fields for the different instruction formats.

Some instruction fields are reserved or must contain a predefined value. 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."

### 8.1.1 Split-Field Notation

Split fields occupy more than one contiguous sequence of bits or occupy a contiguous sequence of bits used in permuted order. Table 8-1 describes split fields that represent the concatenation of the sequences from left to right. They are shown in lowercase letters (spr and tbr). 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.

**Table 8-1. Split-Field Notation and Conventions**

| Field | Description |
|---|---|
| spr (11–20) | Specifies an spr for **mtspr** and **mfspr** instructions. |
| tbr (11–20) | Specifies either the time base lower (TBL) or time base upper (TBU). |

## 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.<br>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 32 bits and the address of the branch instruction or the sum of the BD field sign-extended to 32 bits and the address of the branch instruction.<br>1 The immediate field represents an absolute address. The effective address (EA) of the branch is the LI field sign-extended to 32 bits or the BD field sign-extended to 32 bits. |
| BD (16–29) | Immediate field specifying a 14-bit signed two's complement branch displacement that is concatenated on the right with 0b00 and sign-extended to 32 bits. |
| BI (11–15) | Used to specify a CR bit to be used as the condition of a branch conditional instruction. |
| BO (6–10) | Used to specify options for the branch conditional instructions. The encoding is described in Section 4.2.4.2, “Conditional Branch Control.” |
| **crb**A (11–15) | Used to specify a CR bit to be used as a source. |
| **crb**B (16–20) | Used to specify a CR bit to be used as a source. |
| **crb**D (6–10) | Used to specify a CR bit, or in the FPSCR, as the destination of the result of an instruction. |
| **crf**D (6–8) | Used to specify one of the CR fields, or one of the FPSCR fields, as a destination. |
| **crf**S (11–13) | 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 32 bits. |
| FM (7–14) | This field mask is used to identify the FPSCR fields that are to be updated by the **mtfsf** instruction. |
| **fr**A (11–15) | Used to specify an FPR as a source. |
| **fr**B (16–20) | Used to specify an FPR as a source. |
| **fr**C (21–25) | Used to specify an FPR as a source. |
| **fr**D (6–10) | Used to specify an FPR as the destination. |
| **fr**S (6–10) | 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. |
| LI (6–29) | Immediate field specifying a 24-bit signed two's complement integer that is concatenated on the right with 0b00 and sign-extended to 32 bits. |
| LK (31) | Link bit.<br>0 Does not update the link register (LR).<br>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) | Used in rotate instructions to specify a 32-bit mask as described in Section 4.2.1.4, “Integer Rotate and Shift Instructions.” |
| NB (16–20) | Used to specify the number of bytes to move in an immediate string load or store. |
| OE (21) | Used for extended arithmetic to enable setting OV and SO in the XER. |

Table 8-2. Instruction Syntax Conventions (continued)

| Field | Description |
|---|---|
| OPCD (0–5) | Primary opcode field |
| **r**A (11–15) | Used to specify a GPR to be used as a source or destination. |
| **r**B (16–20) | Used to specify a GPR to be used as a source. |
| Rc (31) | Record bit.<br>0 Does not update the condition register (CR).<br>1 Updates the CR to reflect the result of the operation.<br>For integer instructions, CR[0–2] reflects the result as a signed quantity and CR[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[4–7] reflect floating-point, floating-point enabled, floating-point invalid operation, and floating-point overflow exceptions. |
| **r**D (6–10) | Used to specify a GPR to be used as a destination. |
| **r**S (6–10) | Used to specify a GPR to be used as a source. |
| SH (16–20) | Used to specify a shift amount. |
| SIMM (16–31) | This immediate field is used to specify a 16-bit signed integer. |
| SR (12–15) | Used to specify one of the 16 segment registers. |
| TO (6–10) | Specifies 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 (21–30, 22–30, 26–30) | Extended opcode field. |

## 8.1.3 Notation and Conventions

The operation of some instructions is described by a semiformal language (pseudocode). Table 8-3 describes pseudocode notation and conventions.

Table 8-3. Notation and Conventions

| Notation/Convention | Meaning |
|---|---|
| ← | Assignment |
| $\leftarrow_{\text{iea}}$ | Assignment of an instruction effective address. |
| ¬ | 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. |

## Table 8-3. Notation and Conventions (continued)

| Notation/Convention | Meaning |
|---|---|
| c | Carry. When used as a character of an instruction mnemonic, a ‘c’ indicates a carry out in XER[CA]. |
| e | Extended Precision.<br>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 |
| ? | Unordered comparison relation |
| &, \| | AND, OR logical operators |
| \|\| | Used to describe the concatenation of two values (that is, 010 \|\| 111 is the same as 010111) |
| $\oplus$, $\equiv$ | Exclusive-OR, Equivalence logical operators (for example, $(a \equiv b) = (a \oplus \neg b)$) |
| 0b*nnnn* | A number expressed in binary format. |
| 0x*nnnn* | A number expressed in hexadecimal format. |
| (*n*)x | The replication of x, *n* times (that is, x concatenated to itself *n* – 1 times).<br>(*n*)0 and (*n*)1 are special cases. A description of the special cases follows:<br>• (*n*)0 means a field of *n* bits with each bit equal to 0. Thus (5)0 is equivalent to 0b00000.<br>• (*n*)1 means a field of *n* bits with each bit equal to 1. Thus (5)1 is equivalent to 0b11111. |
| (**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.<br>The 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. 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. |

**Table 8-3. Notation and Conventions (continued)**

| Notation/Convention | Meaning |
|---|---|
| Do | Do loop.<br>• Indenting shows range.<br>• “To” and/or “by” clauses specify incrementing an iteration variable.<br>• “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 double words; such an insertion requires more than one instruction.) This operation is used for rotate and shift instructions. (Note that simplified mnemonics are referred to as extended mnemonics in the architecture specification.) |
| 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. |
| NIA | Next instruction address, which is the32-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. Does not correspond to any architected register. |
| OEA | 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 | User instruction set architecture |
| VEA | Virtual environment architecture |

Table 8-4 describes instruction field notation conventions used throughout this chapter.

**Table 8-4. Instruction Field Conventions**

| 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) |

Some operators are applied before others, as shown in Table 8-5. 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.

**Table 8-5. Precedence Rules**

| Precedence | Operators | Associativity |
|---|---|---|
| Highest | x[*n*], function evaluation | Left to right |
| | (*n*)x or replication,<br>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 |
| Lowest | ←, $\leftarrow_{iea}$ | None |

### 8.1.4 Computation Modes

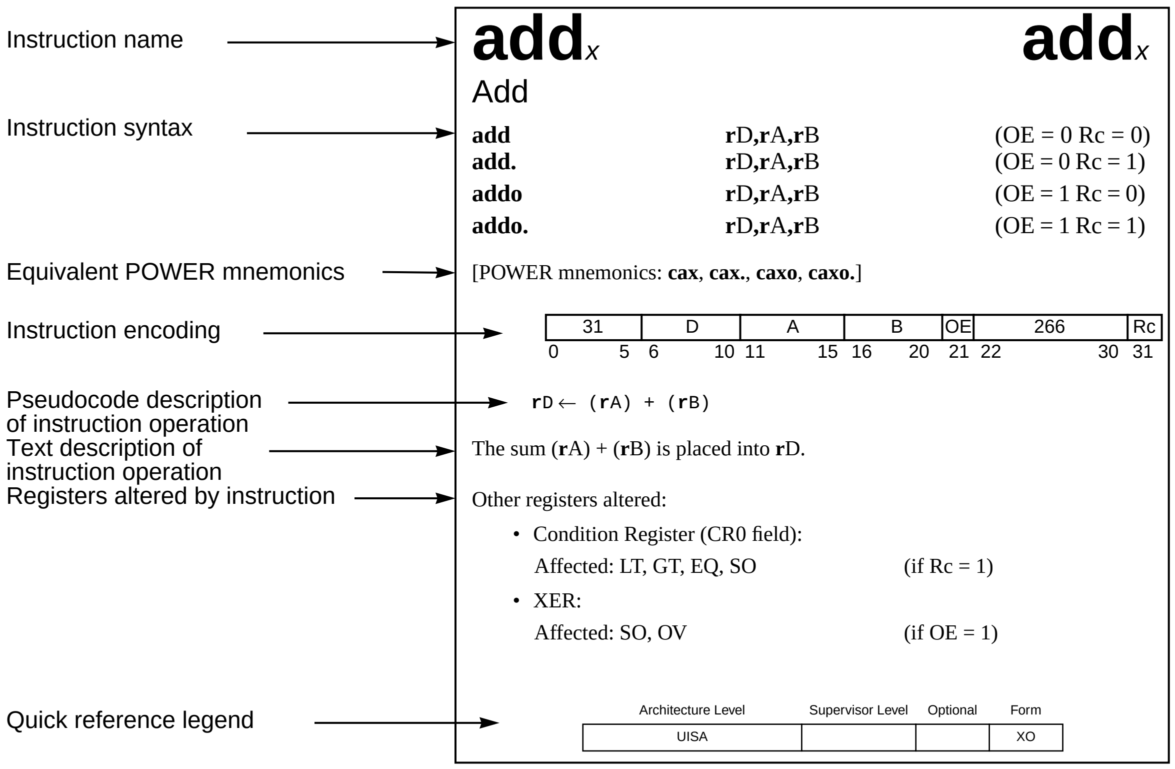
The architecture allows for the following types of implementations:

- 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.

Note that the all pseudocode examples provided in this chapter are for 32-bit implementations. For more information on 64-bit and 32-bit modes, refer to Section 1.1.1, “The 64-Bit Architecture and the 32-Bit Subset.”

## 8.2 Instruction Set

The remainder of this chapter lists and describes individual instructions, which are listed in alphabetical order by mnemonic. Figure 8-1 shows the format for each instruction description page.


Instruction name
Instruction syntax
Equivalent POWER mnemonics
Instruction encoding
Pseudocode description of instruction operation
Text description of instruction operation
Registers altered by instruction
Quick reference legend

**add**$_x$ **add**$_x$

Add

| | | |
|---|---|---|
| **add** | **r**D,**r**A,**r**B | (OE = 0 Rc = 0) |
| **add.** | **r**D,**r**A,**r**B | (OE = 0 Rc = 1) |
| **addo** | **r**D,**r**A,**r**B | (OE = 1 Rc = 0) |
| **addo.** | **r**D,**r**A,**r**B | (OE = 1 Rc = 1) |

[POWER mnemonics: **cax**, **cax.**, **caxo**, **caxo.**]

| 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.

Other registers altered:

- Condition Register (CR0 field):
  Affected: LT, GT, EQ, SO (if Rc = 1)
- XER:
  Affected: SO, OV (if OE = 1)

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | XO |

**Figure 8-1. Instruction Description**

Note that the execution unit that executes the instruction may not be the same for all processors.

## add*x* add*x*

Add

| | | |
|---|---|---|
| **add** | **r**D,**r**A,**r**B | (OE = 0 Rc = 0) |
| **add.** | **r**D,**r**A,**r**B | (OE = 0 Rc = 1) |
| **addo** | **r**D,**r**A,**r**B | (OE = 1 Rc = 0) |
| **addo.** | **r**D,**r**A,**r**B | (OE = 1 Rc = 1) |

[POWER mnemonics: **cax**, **cax.**, **caxo**, **caxo.**]

| 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 XER below).

- XER:

  Affected: SO, OV (if OE = 1)

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | XO |

## addc*x* addc*x*

Add Carrying

| | | |
|---|---|---|
| **addc** | **r**D,**r**A,**r**B | (OE = 0 Rc = 0) |
| **addc.** | **r**D,**r**A,**r**B | (OE = 0 Rc = 1) |
| **addco** | **r**D,**r**A,**r**B | (OE = 1 Rc = 0) |
| **addco.** | **r**D,**r**A,**r**B | (OE = 1 Rc = 1) |

[POWER mnemonics: **a**, **a.**, **ao**, **ao.**]

| 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)

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | XO |

## **adde***x* **adde***x*

Add Extended

| | | |
|---|---|---|
| **adde** | **r**D,**r**A,**r**B | (OE = 0 Rc = 0) |
| **adde.** | **r**D,**r**A,**r**B | (OE = 0 Rc = 1) |
| **addeo** | **r**D,**r**A,**r**B | (OE = 1 Rc = 0) |
| **addeo.** | **r**D,**r**A,**r**B | (OE = 1 Rc = 1) |

[POWER mnemonics: **ae**, **ae.**, **aeo**, **aeo.**]

| 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)

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | XO |

## addi addi

Add Immediate

**addi** **r**D**,r**A,SIMM

[POWER mnemonic: **cal**]

| 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) + SIMM is placed into **r**D.

The **addi** instruction is preferred for addition because it sets few status bits. Note that **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 |

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | D |

## addic addic

Add Immediate Carrying

**addic** **r**D**,r**A**,**SIMM

[POWER mnemonic: **ai**]

| 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

Simplified mnemonics:

| | | |
|---|---|---|
| **subic** **r**D**,r**A**,**value | equivalent to | **addic** **r**D**,r**A**,**–value |

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | D |

## addic. addic.

Add Immediate Carrying and Record

**addic.** **r**D**,r**A,SIMM

[POWER mnemonic: **ai.**]

| 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

Simplified mnemonics:

| | | |
|---|---|---|
| **subic. r**D**,r**A**,**value | equivalent to | **addic. r**D**,r**A**,**–value |

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | D |

## addis                                                                 addis

Add Immediate Shifted

**addis** **r**D**,r**A,SIMM

[POWER mnemonic: **cau**]

| 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 that **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 |

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | D |

## addme*x* addme*x*

Add to Minus One Extended

| | | |
|---|---|---|
| **addme** | **r**D,**r**A | (OE = 0 Rc = 0) |
| **addme.** | **r**D,**r**A | (OE = 0 Rc = 1) |
| **addmeo** | **r**D,**r**A | (OE = 1 Rc = 0) |
| **addmeo.** | **r**D,**r**A | (OE = 1 Rc = 1) |

[POWER mnemonics: **ame**, **ame.**, **ameo**, **ameo.**]

▨ 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 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)

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | XO |

## addze*x* addze*x*

Add to Zero Extended

| | | |
|---|---|---|
| **addze** | **r**D,**r**A | (OE = 0 Rc = 0) |
| **addze.** | **r**D,**r**A | (OE = 0 Rc = 1) |
| **addzeo** | **r**D,**r**A | (OE = 1 Rc = 0) |
| **addzeo.** | **r**D,**r**A | (OE = 1 Rc = 1) |

[POWER mnemonics: **aze**, **aze.**, **azeo**, **azeo.**]

▨ 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)

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | XO |

## and*x* and*x*

AND

| | | |
|---|---|---|
| **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)

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | X |

## andc*x* andc*x*

AND with Complement

| | | |
|---|---|---|
| **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)

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | X |

## andi. andi.

AND Immediate

**andi.** **r**A,**r**S,UIMM

[POWER mnemonic: **andil.**]

| | | | |
|---|---|---|---|
| 28 | S | A | UIMM |
| 0 5 | 6 10 | 11 15 | 16 31 |

```
rA ← (rS) & ((16)0 || UIMM)
```

The contents of **r**S are ANDed with 0x0000 || UIMM and the result is placed into **r**A.

Other registers altered:

- Condition Register (CR0 field):
  Affected: LT, GT, EQ, SO

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | D |

## andis. andis.

AND Immediate Shifted

**andis.** **r**A,**r**S,UIMM

[POWER mnemonic: **andiu.**]

| 29 | S | A | UIMM |
|---|---|---|---|
| 0 5 | 6 10 | 11 15 | 16 31 |

```
rA← (rS) + ( UIMM || (16)0)
```

The contents of **r**S are ANDed with UIMM || 0x0000 and the result is placed into **r**A.

Other registers altered:

- Condition Register (CR0 field):
  Affected: LT, GT, EQ, SO

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | D |

## **b***x* **b***x*

Branch

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

target_addr specifies the branch target address.

If AA = 0, then the branch target address is the sum of LI || 0b00 sign-extended and the address of this instruction.

If AA = 1, then the branch target address is the value LI || 0b00 sign-extended.

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)

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | I |

## bc*x* bc*x*

Branch Conditional

| | | |
|---|---|---|
| **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 LK then LR ←iea CIA + 4
if ¬ BO[2] then CTR ← CTR – 1
ctr_ok ← BO[2] | ((CTR ≠ 0) ⊕ BO[3])
cond_ok ← BO[0] | (CR[BI] ≡ BO[1])
if ctr_ok & cond_ok then
     if AA then NIA ←iea EXTS(BD || 0b00)
     else NIA ←iea CIA + EXTS(BD || 0b00)
```

The BI field specifies the CR bit used as the condition of the branch, as shown in Table 8-6.

**Table 8-6. BI Operand Settings for CR Fields**

| CR*n* Bits | CR Bits | BI | Description |
|---|---|---|---|
| CR0[0] | 0 | 00000 | Negative (LT)—Set when the result is negative. |
| CR0[1] | 1 | 00001 | Positive (GT)—Set when the result is positive (and not zero). |
| CR0[2] | 2 | 00010 | Zero (EQ)—Set when the result is zero. |
| CR0[3] | 3 | 00011 | Summary overflow (SO). Copy of XER[SO] at the instruction’s completion. |
| CR1[0] | 4 | 00100 | Copy of FPSCR[FX] at the instruction’s completion. |
| CR1[1] | 5 | 00101 | Copy of FPSCR[FEX] at the instruction’s completion. |
| CR1[2] | 6 | 00110 | Copy of FPSCR[VX] at the instruction’s completion. |
| CR1[3] | 7 | 00111 | Copy of FPSCR[OX] at the instruction’s completion. |
| CR*n*[0] | 8<br>12<br>16<br>20<br>24<br>28 | 01000<br>01100<br>10000<br>10100<br>11000<br>11100 | Less than or floating-point less than (LT, FL).<br>For integer compare instructions:<br>**r**A < SIMM or **r**B (signed comparison) or **r**A < UIMM or **r**B (unsigned comparison).<br>For floating-point compare instructions:**fr**A < **fr**B. |
| CR*n*[1] | 9<br>13<br>17<br>21<br>25<br>29 | 01001<br>01101<br>10001<br>10101<br>11001<br>11101 | Greater than or floating-point greater than (GT, FG).<br>For integer compare instructions:<br>**r**A > SIMM or **r**B (signed comparison) or **r**A > UIMM or **r**B (unsigned comparison).<br>For floating-point compare instructions:**fr**A > **fr**B. |
| CR*n*[2] | 10<br>14<br>18<br>22<br>26<br>30 | 01010<br>01110<br>10010<br>10110<br>11010<br>11110 | Equal or floating-point equal (EQ, FE).<br>For integer compare instructions: **r**A = SIMM, UIMM, or **r**B.<br>For floating-point compare instructions: **fr**A = **fr**B. |
| CR*n*[3] | 11<br>15<br>19<br>23<br>27<br>31 | 01011<br>01111<br>10011<br>10111<br>11011<br>11111 | Summary overflow or floating-point unordered (SO, FU).<br>For integer compare instructions, this is a copy of XER[SO] at the completion of the instruction.<br>For floating-point compare instructions, one or both of **fr**A and **fr**B is a NaN. |

Table 8-7 shows BO encodings. See also Section 4.2.4.2, “Conditional Branch Control.”

**Table 8-7. BO Operand Encodings**

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

*z* bits are ignored and should be cleared, as they may be assigned a meaning in a future version of the architecture.
*y* bits provides a hint about whether a conditional branch is likely to be taken and may be used by some implementations to improve performance.

target_addr specifies the branch target address.

If AA = 0, the branch target address is the sum of BD || 0b00 sign-extended and the address of this instruction.

If AA = 1, the branch target address is the value BD || 0b00 sign-extended.

If LK = 1, the EA of the instruction following the branch instruction is placed into the LR.

Other registers altered:

| | |
|---|---|
| Affected: Count Register (CTR) | (if BO[2] = 0) |
| Affected: 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 |

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | B |

## bcctr*x* bcctr*x*

Branch Conditional to Count Register

| | | |
|---|---|---|
| **bcctr** | BO,BI | (LK = 0) |
| **bcctrl** | BO,BI | (LK = 1) |

[POWER mnemonics: **bcc**, **bccl**]

▨ Reserved

| | | | | | |
|---|---|---|---|---|---|
| 19 | BO | BI | 0 0 0 0 0 | 528 | LK |
| 0 5 | 6 10 | 11 15 | 16 20 | 21 30 | 31 |

```
cond_ok ← BO[0] | (CR[BI] ≡ BO[1])
if cond_ok then
 NIA ←iea CTR || 0b00
 if LK then LR ←iea CIA + 4
```

The BI field specifies the CR bit used as the condition of the branch, as shown in Table 8-8.

**Table 8-8. BI Operand Settings for CR Fields**

| CR*n* Bits | CR Bits | BI | Description |
|---|---|---|---|
| CR0[0] | 0 | 00000 | Negative (LT)—Set when the result is negative. |
| CR0[1] | 1 | 00001 | Positive (GT)—Set when the result is positive (and not zero). |
| CR0[2] | 2 | 00010 | Zero (EQ)—Set when the result is zero. |
| CR0[3] | 3 | 00011 | Summary overflow (SO). Copy of XER[SO] at the instruction's completion. |
| CR1[0] | 4 | 00100 | Copy of FPSCR[FX] at the instruction's completion. |
| CR1[1] | 5 | 00101 | Copy of FPSCR[FEX] at the instruction's completion. |
| CR1[2] | 6 | 00110 | Copy of FPSCR[VX] at the instruction's completion. |
| CR1[3] | 7 | 00111 | Copy of FPSCR[OX] at the instruction's completion. |
| CR*n*[0] | 8<br>12<br>16<br>20<br>24<br>28 | 01000<br>01100<br>10000<br>10100<br>11000<br>11100 | Less than or floating-point less than (LT, FL).<br>For integer compare instructions:<br>**r**A < SIMM or **r**B (signed comparison) or **r**A < UIMM or **r**B (unsigned comparison).<br>For floating-point compare instructions:**fr**A < **fr**B. |
| CR*n*[1] | 9<br>13<br>17<br>21<br>25<br>29 | 01001<br>01101<br>10001<br>10101<br>11001<br>11101 | Greater than or floating-point greater than (GT, FG).<br>For integer compare instructions:<br>**r**A > SIMM or **r**B (signed comparison) or **r**A > UIMM or **r**B (unsigned comparison).<br>For floating-point compare instructions:**fr**A > **fr**B. |
| CR*n*[2] | 10<br>14<br>18<br>22<br>26<br>30 | 01010<br>01110<br>10010<br>10110<br>11010<br>11110 | Equal or floating-point equal (EQ, FE).<br>For integer compare instructions: **r**A = SIMM, UIMM, or **r**B.<br>For floating-point compare instructions: **fr**A = **fr**B. |
| CR*n*[3] | 11<br>15<br>19<br>23<br>27<br>31 | 01011<br>01111<br>10011<br>10111<br>11011<br>11111 | Summary overflow or floating-point unordered (SO, FU).<br>For integer compare instructions, this is a copy of XER[SO] at the completion of the instruction.<br>For floating-point compare instructions, one or both of **fr**A and **fr**B is a NaN. |

Table 8-7 shows BO encodings. See also Section 4.2.4.2, “Conditional Branch Control.”

**Table 8-9. BO Operand Encodings**

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

*z* bits are ignored and should be cleared, as they may be assigned a meaning in a future version of the architecture.
*y* bits provides a hint about whether a conditional branch is likely to be taken and may be used by some implementations to improve performance.

The branch target address is CTR || 0b00.

If LK = 1, the EA of the instruction following the branch instruction is placed into the LR.

If “decrement and test CTR” is specified (BO[2] = 0), the instruction form is invalid.

Other registers altered:

| | |
|---|---|
| Affected: Link Register (LR) | (if LK = 1) |

Simplified mnemonics:

| | | |
|---|---|---|
| **bltctr** | equivalent to | **bcctr 12,0** |
| **bnectr cr2** | equivalent to | **bcctr 4,10** |

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | XL |

## bclr*x* bclr*x*

Branch Conditional to Link Register

| | | |
|---|---|---|
| **bclr** | BO,BI | (LK = 0) |
| **bclrl** | BO,BI | (LK = 1) |

[POWER mnemonics: **bcr**, **bcrl**]

▨ Reserved

| 19 | BO | BI | ▨ 0 0 0 0 0 | 16 | LK |
|---|---|---|---|---|---|
| 0 5 | 6 10 | 11 15 | 16 20 | 21 30 | 31 |

```
if ¬ BO[2] then CTR ← CTR – 1
ctr_ok ← BO[2] | ((CTR ≠ 0) ⊕ BO[3])
cond_ok ← BO[0] | (CR[BI] ≡ BO[1])
if ctr_ok & cond_ok then
 NIA $\leftarrow_{iea}$ LR || 0b00
 if LK then LR $\leftarrow_{iea}$ CIA + 4
```

The BI field specifies the CR bit used as the condition of the branch, as shown in Table 8-10.

**Table 8-10. BI Operand Settings for CR Fields**

| CR*n* Bits | CR Bits | BI | Description |
|---|---|---|---|
| CR0[0] | 0 | 00000 | Negative (LT)—Set when the result is negative. |
| CR0[1] | 1 | 00001 | Positive (GT)—Set when the result is positive (and not zero). |
| CR0[2] | 2 | 00010 | Zero (EQ)—Set when the result is zero. |
| CR0[3] | 3 | 00011 | Summary overflow (SO). Copy of XER[SO] at the instruction’s completion. |
| CR1[0] | 4 | 00100 | Copy of FPSCR[FX] at the instruction’s completion. |
| CR1[1] | 5 | 00101 | Copy of FPSCR[FEX] at the instruction’s completion. |
| CR1[2] | 6 | 00110 | Copy of FPSCR[VX] at the instruction’s completion. |
| CR1[3] | 7 | 00111 | Copy of FPSCR[OX] at the instruction’s completion. |
| CR*n*[0] | 8<br>12<br>16<br>20<br>24<br>28 | 01000<br>01100<br>10000<br>10100<br>11000<br>11100 | Less than or floating-point less than (LT, FL).<br>For integer compare instructions:<br>**r**A < SIMM or **r**B (signed comparison) or **r**A < UIMM or **r**B (unsigned comparison).<br>For floating-point compare instructions:**fr**A < **fr**B. |
| CR*n*[1] | 9<br>13<br>17<br>21<br>25<br>29 | 01001<br>01101<br>10001<br>10101<br>11001<br>11101 | Greater than or floating-point greater than (GT, FG).<br>For integer compare instructions:<br>**r**A > SIMM or **r**B (signed comparison) or **r**A > UIMM or **r**B (unsigned comparison).<br>For floating-point compare instructions:**fr**A > **fr**B. |
| CR*n*[2] | 10<br>14<br>18<br>22<br>26<br>30 | 01010<br>01110<br>10010<br>10110<br>11010<br>11110 | Equal or floating-point equal (EQ, FE).<br>For integer compare instructions: **r**A = SIMM, UIMM, or **r**B.<br>For floating-point compare instructions: **fr**A = **fr**B. |
| CR*n*[3] | 11<br>15<br>19<br>23<br>27<br>31 | 01011<br>01111<br>10011<br>10111<br>11011<br>11111 | Summary overflow or floating-point unordered (SO, FU).<br>For integer compare instructions, this is a copy of XER[SO] at the completion of the instruction.<br>For floating-point compare instructions, one or both of **fr**A and **fr**B is a NaN. |

Table 8-7 shows BO encodings. See also Section 4.2.4.2, “Conditional Branch Control.”

**Table 8-11. BO Operand Encodings**

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

*z* bits are ignored and should be cleared, as they may be assigned a meaning in a future version of the architecture.
*y* bits provides a hint about whether a conditional branch is likely to be taken and may be used by some implementations to improve performance.

The branch target address is LR || 0b00.

If LK = 1, the EA of the instruction following the branch instruction is placed into the LR.

Other registers altered:

| | |
|---|---|
| Affected: Count Register (CTR) | (if BO[2] = 0) |
| Affected: Link Register (LR) | (if LK = 1) |

Simplified mnemonics:

| | | | |
|---|---|---|---|
| **bltlr** | equivalent to | **bclr** | **12,0** |
| **bnelr cr2** | equivalent to | **bclr** | **4,10** |
| **bdnzlr** | equivalent to | **bclr** | **16,0** |

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | XL |

## cmp cmp

Compare

**cmp** **crf**D,L,**r**A,**r**B

▨ Reserved

|  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|
| 31 | crfD | 0 ▨ | L | A | B | 0 0 0 0 0 0 0 0 0 0 | 0 ▨ |
| 0 5 | 6 8 | 9 | 10 | 11 15 | 16 20 | 21 30 | 31 |

```
if L = 0 then a ← EXTS(rA)
        b ← EXTS(rB)
else    a ← (rA)
        b ← (rB)
if  a < b then c ← 0b100
else if a > b then c ← 0b010
else     c ← 0b001
CR[(4 * crfD) through (4 * crfD + 3)] ← c || XER[SO]
```

The contents of **r**A are compared with the contents of **r**B, treating the operands as signed integers. The result of the comparison is placed into CR field **crf**D. The L bit has no effect on 32-bit operations.

Other registers altered:

- Condition Register (CR field specified by operand **crf**D):

  Affected: LT, GT, EQ, SO

Simplified mnemonics:

**cmpw crf**D**,r**A**,r**B equivalent to **cmp** **crf**D**,0,r**A**,r**B

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA |  |  | X |

## cmpi cmpi

Compare Immediate

**cmpi** **crf**D,L,**r**A,SIMM

▨ Reserved

| 0–5 | 6–8 | 9 | 10 | 11–15 | 16–31 |
|---|---|---|---|---|---|
| 11 | crfD | 0 ▨ | L | A | SIMM |

```
a ← (rA)
if   a < EXTS(SIMM) then c ← 0b100
else if a > EXTS(SIMM) then c ← 0b010
else     c ← 0b001
CR[(4 * crfD) through (4 * crfD + 3)] ← c || XER[SO]
```

The contents of **r**A 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.

In 32-bit implementations, if L = 1 the instruction form is invalid.

Other registers altered:

- Condition Register (CR field specified by operand **crf**D):
  Affected: LT, GT, EQ, SO

Simplified mnemonics:

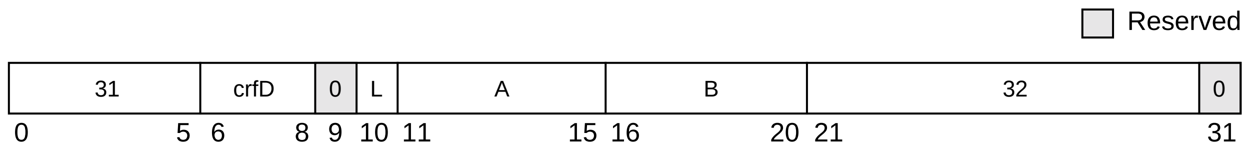
| | | |
|---|---|---|
| **cmpwi crf**D**,r**A**,**value | equivalent to | **cmpi** **crf**D**,0,r**A**,**value |

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | D |

## cmpl cmpl

Compare Logical

**cmpl** **crf**D,L,**r**A,**r**B



```
a ← (rA)
b ← (rB)
if   a <U b then c ← 0b100
else if a >U b then c ← 0b010
else     c ← 0b001
CR[4 * crfD–4 * crfD + 3] ← c || XER[SO]
```

The contents of **r**A are compared with the contents of **r**B, treating the operands as unsigned integers. The result of the comparison is placed into CR field **crf**D.

In 32-bit implementations, if L = 1 the instruction form is invalid.

Other registers altered:

- Condition Register (CR field specified by operand **crf**D):
  Affected: LT, GT, EQ, SO

Simplified mnemonics:

**cmplw crf**D**,r**A**,r**B equivalent to **cmpl** **crf**D**,0,r**A**,r**B

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | X |

## cmpli cmpli

Compare Logical Immediate

**cmpli** **crf**D,L,**r**A,UIMM

▨ Reserved

| 10 | crfD | 0 | L | A | UIMM |
|---|---|---|---|---|---|
| 0 5 | 6 8 | 9 | 10 | 11 15 | 16 31 |

```
a ← (rA)
if   a <U ((16)0 || UIMM) then c ← 0b100
else if a >U ((16)0 || UIMM) then c ← 0b010
else    c ← 0b001
CR[4 * crfD–4 * crfD + 3] ← c || XER[SO]
```

The contents of **r**A are compared with 0x0000 || UIMM, treating the operands as unsigned integers. The result of the comparison is placed into CR field **crf**D.

In 32-bit implementations, if L = 1 the instruction form is invalid.

Other registers altered:

- Condition Register (CR field specified by operand **crf**D):
  Affected: LT, GT, EQ, SO

Simplified mnemonics:

| | | |
|---|---|---|
| **cmplwi crf**D,**r**A,value | equivalent to | **cmpli crf**D,**0**,**r**A,value |

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | D |

## cntlzw*x* cntlzw*x*

Count Leading Zeros Word

| | | |
|---|---|---|
| **cntlzw** | **r**A,**r**S | (Rc = 0) |
| **cntlzw.** | **r**A,**r**S | (Rc = 1) |

[POWER mnemonics: **cntlz**, **cntlz.**]

▨ Reserved

| 31 | S | A | 0 0 0 0 0 | 26 | Rc |
|---|---|---|---|---|---|
| 0 5 | 6 10 | 11 15 | 16 20 | 21 30 | 31 |

```
n ← 0
do while n < 32
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 **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.

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | X |

## crand crand

Condition Register AND

**crand** **crb**D,**crb**A,**crb**B

▨ Reserved

| 19 | crbD | crbA | crbB | 257 | 0 |
|---|---|---|---|---|---|
| 0 5 | 6 10 | 11 15 | 16 20 | 21 30 | 31 |

```
CR[crbD] ← CR[crbA] & CR[crbB]
```

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.

Other registers altered:

- Condition Register:
  Affected: Bit specified by operand **crb**D

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | XL |

## crandc crandc

Condition Register AND with Complement

**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 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 condition register bit specified by **crb**D.

Other registers altered:

- Condition Register:

  Affected: Bit specified by operand **crb**D

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | XL |

## creqv                                                    creqv

Condition Register Equivalent

**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 CR bit specified by **crb**A is XORed with the CR bit 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:

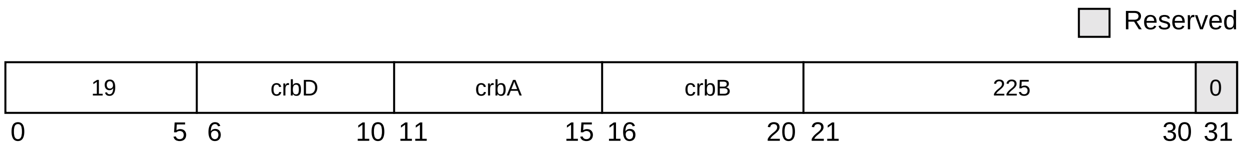
| | | |
|---|---|---|
| **crset** **crb**D | equivalent to | **creqv** **crb**D,**crb**D,**crb**D |

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | XL |

## crnand crnand

Condition Register NAND

**crnand** **crb**D,**crb**A,**crb**B



```
CR[crbD] ← ¬ (CR[crbA] & CR[crbB])
```

The CR bit specified by **crb**A is ANDed with the CR bit 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

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | XL |

## crnor crnor

Condition Register NOR

**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 CR bit specified by **crb**A is ORed with the CR bit 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 |

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | XL |

## cror cror

Condition Register OR

**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 CR bit specified by **crb**A is ORed with the CR bit 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 |

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | XL |

## crorc crorc

Condition Register OR with Complement

**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 CR bit 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

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | XL |

## crxor crxor

Condition Register XOR

**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 CR bit specified by **crb**A is XORed with the CR bit specified by **crb**B and the result is placed into the CR bit 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 |

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | XL |

## dcba dcba

Data Cache Block Allocate

**dcba** **r**A,**r**B

▨ Reserved

| | | | | | |
|---|---|---|---|---|---|
| 31 | ▨ 0 0 0 0 0 | A | B | 758 | 0 |
| 0 5 | 6 10 | 11 15 | 16 20 | 21 30 | 31 |

EA is the sum (**r**A|0) + (**r**B).

**dcba** allocates the data cache block addressed by EA by marking it valid without reading the contents of the block from memory; data in the cache block is considered undefined after **dcba** completes. **dcba** is a hint that the program will probably soon store into a portion of the block, but the contents of the rest of the block are not meaningful to the program (eliminating the need to read the entire block from main memory), and can provide for improved performance in these code sequences. **dcba** executes as follows:

- If the cache block containing the byte addressed by EA is in the data cache, the contents of all bytes are made undefined but the cache block is still considered valid. Note that programming errors can occur if the data in this cache block is subsequently read or used inadvertently.
- If the cache block containing the byte addressed by EA is not in the data cache and the corresponding space is caching-allowed, the cache block is allocated and made valid without fetching the block from main memory. All byte values are undefined.
- If the addressed byte corresponds to a caching-inhibited page or block (the I bit is set), **dcba** is treated as a no-op.
- If the cache block containing the byte addressed by EA is coherency-required and it exists in the any other processor's cache, those processors perform the appropriate bus transactions to enforce coherency.

**dcba** acts as a store to the addressed byte with respect to translation, referenced and changed recording, memory protection, and ordering enforced by **eieio** or by a combination of caching-inhibited and guarded attributes. However, a DSI exception is not invoked for a translation or protection violation and the referenced and changed bits need not be updated when the page or block is cache-inhibited (causing **dcba** to be treated as a no-op).

Other registers altered: None

The OEA defines **dcba** to clear all bytes of a new cache block it was not in the cache. Additionally, as **dcba** may establish a data cache block without verifying that the associated physical address is valid, a delayed machine check exception may occur.

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| VEA | | √ | X |

## dcbf dcbf

Data Cache Block Flush

**dcbf** **r**A,**r**B

▨ Reserved

| 31 | 0 0 0 0 0 | A | B | 86 | 0 |
|---|---|---|---|---|---|
| 0 5 | 6 10 | 11 15 | 16 20 | 21 30 | 31 |

EA is the sum (**r**A|0) + (**r**B).

**dcbf** invalidates the block in the data cache addressed by EA, copying the block to memory first if it contains modified data. If the block is marked coherency-required in a multiprocessor system, if necessary the processor sends an address-only broadcast to other processors. If the block is modified in another processor, broadcasting **dcbf** causes the processor to copy the block to memory and invalidate the block. The action taken depends on the memory mode associated with the block containing the byte addressed by EA and on the state of that block. The list below describes the action taken for the various states of the memory coherency attribute (M bit).

- Coherency required
  - Unmodified block—Invalidates copies of the data cache block in all processors.
  - Modified block—Copies the block to memory. Invalidates copies of the block in the data caches of all processors.
  - Absent block—If modified copies of the block are in the data caches of other processors, causes them to be copied to memory and invalidated in those data caches. If unmodified copies are in the data caches of other processors, causes those copies to be invalidated in those data caches.
- Coherency not 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 (target block not in cache)—No action is taken.

The function of **dcbf** 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 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

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| VEA | | | X |

## dcbi dcbi

Data Cache Block Invalidate

**dcbi** **r**A,**r**B

▨ Reserved

|  |  |  |  |  |  |
|---|---|---|---|---|---|
| 31 | 0 0 0 0 0 ▨ | A | B | 470 | 0 ▨ |
| 0 5 | 6 10 | 11 15 | 16 20 | 21 30 | 31 |

EA is the sum (**r**A|0) + (**r**B).

The action taken is dependent on the memory mode associated with the block containing the byte addressed by EA and on the state of that block. The list below describes the action taken if the block containing the byte addressed by EA is or is not in the cache.

- Coherency required
  - Unmodified block—Invalidates copies of the block in the data caches of all processors.
  - Modified block—Invalidates copies of the block in the data caches of all processors. (Discards the modified contents.)
  - Absent block—If copies of the block are in the data caches of any other processor, causes the copies to be invalidated in those data caches. (Discards any modified contents.)
- Coherency not required
  - Unmodified block—Invalidates the block in the processor’s data cache.
  - Modified block—Invalidates the block in the processor’s data cache. (Discards the modified contents.)
  - Absent block (target block not in cache)—No action is taken.

When data address translation is enabled, MSR[DR] = 1, and the virtual address has no translation, a DSI exception occurs.

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. This instruction operates as a store to the addressed byte with respect to address translation and protection. The referenced and changed bits are modified appropriately.

This is a supervisor-level instruction.

Other registers altered:

- None

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| OEA | √ |  | X |

## dcbst dcbst

Data Cache Block Store

**dcbst** **r**A,**r**B

▨ Reserved

| 0–5 | 6–10 | 11–15 | 16–20 | 21–30 | 31 |
|---|---|---|---|---|---|
| 31 | ▨ 0 0 0 0 0 | A | B | 54 | ▨ 0 |

EA is the sum (**r**A|0) + (**r**B).

The **dcbst** instruction executes as follows:

- 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 data cache of any processor and has been modified, the writing of it to main memory is initiated.
- 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 data cache of this processor and has been modified, the writing of it to main memory is initiated.

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. 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

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| VEA | | | X |

## dcbt dcbt

Data Cache Block Touch

**dcbt** **r**A,**r**B

▨ Reserved

|  |  |  |  |  |  |
|---|---|---|---|---|---|
| 31 | 0 0 0 0 0 | A | B | 278 | 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 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.

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 that this instruction is defined architecturally to perform the same functions as the **dcbtst** instruction. Both are defined to allow implementations to differentiate the bus actions when fetching into the cache for the case of a load and for a store.

Other registers altered:

- None

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| VEA |  |  | X |

## dcbtst dcbtst

Data Cache Block Touch for Store

**dcbtst** **r**A,**r**B

▨ Reserved

| | | | | | |
|---|---|---|---|---|---|
| 31 | 0 0 0 0 0 | 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, 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 that this instruction is defined architecturally to perform the same functions as the **dcbt** instruction. Both are defined to allow implementations to differentiate the bus actions when fetching into the cache for the case of a load and for a store.

Other registers altered:

- None

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| VEA | | | X |

## dcbz dcbz

Data Cache Block Clear to Zero

**dcbz** **r**A,**r**B

[POWER mnemonic: **dclz**]

▨ Reserved

| 31 | 0 0 0 0 0 | A | B | 1014 | 0 |
|---|---|---|---|---|---|
| 0 5 | 6 10 | 11 15 | 16 20 | 21 30 | 31 |

EA is the sum (**r**A|0) + (**r**B).

The **dcbz** instruction executes as follows:

- If the cache block containing the byte addressed by EA is in the data cache, all bytes are cleared.
- If the cache block containing the byte addressed by EA is not in the data cache and the corresponding memory page or block is caching-allowed, the cache block is allocated (and made valid) in the data cache without fetching the block from main memory, and all bytes are cleared.
- 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.
- If the cache block containing the byte addressed by EA is in coherency-required mode, and the cache block exists in the data cache(s) of any other processor(s), it is kept coherent in those caches (i.e. the processor performs the appropriate bus transactions to enforce this).

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).

Other registers altered:

- None

The OEA describes how **dcbz** may establish a data cache block without verifying that the associated physical address is valid. This can cause a delayed machine check exception; see Chapter 6, “Exceptions.”

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| VEA | | | X |

## **divw*x*** **divw*x***

Divide Word

| | | |
|---|---|---|
| **divw** | **r**D,**r**A,**r**B | (OE = 0 Rc = 0) |
| **divw.** | **r**D,**r**A,**r**B | (OE = 0 Rc = 1) |
| **divwo** | **r**D,**r**A,**r**B | (OE = 1 Rc = 0) |
| **divwo.** | **r**D,**r**A,**r**B | (OE = 1 Rc = 1) |

| 31 | D | A | B | OE | 491 | Rc |
|---|---|---|---|---|---|---|
| 0 5 | 6 10 | 11 15 | 16 20 | 21 | 22 30 | 31 |

```
dividend ← (rA)
divisor ← (rB)
rD ← dividend ÷ divisor
```

The dividend is the contents of **r**A. The divisor is the contents of **r**B. The 32-bit quotient is formed and placed in **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 $0 \le r < |\text{divisor}|$ (if the dividend is non-negative), and $-|\text{divisor}| < r \le 0$ (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 **r**A by the contents of **r**B can be computed as follows, except in the case that the contents of **r**A = $-2^{31}$ and the contents 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)

- XER:

  Affected: SO, OV (if OE = 1)

  **Note:** The setting of the affected XER bits is mode-independent and reflects overflow of the 32-bit result.

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | XO |

## divwu*x* divwu*x*

Divide Word Unsigned

| | | |
|---|---|---|
| **divwu** | **r**D,**r**A,**r**B | (OE = 0 Rc = 0) |
| **divwu.** | **r**D,**r**A,**r**B | (OE = 0 Rc = 1) |
| **divwuo** | **r**D,**r**A,**r**B | (OE = 1 Rc = 0) |
| **divwuo.** | **r**D,**r**A,**r**B | (OE = 1 Rc = 1) |

| 31 | D | A | B | OE | 459 | Rc |
|---|---|---|---|---|---|---|
| 0 5 | 6 10 | 11 15 | 16 20 | 21 | 22 30 | 31 |

```
dividend ← (rA)
divisor ← (rB)
rD ← dividend ÷ divisor
```

The dividend is the contents of **r**A. The divisor is the contents of **r**B. A 32-bit quotient is formed. The 32-bit quotient is placed into **r**D. 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 **r**A by the contents 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)

- 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 32-bit result.

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | XO |

## eciwx eciwx

External Control In Word Indexed

**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 ← 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 **r**D.

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.

The **eciwx** instruction is supported for EAs that reference memory segments in which SR[T] = 1 and for EAs mapped by DBAT registers. 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 architecture.

Other registers altered:

- None

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| VEA | | √ | X |

## ecowx ecowx

External Control Out Word Indexed

**ecowx** **r**S,**r**A,**r**B

▨ Reserved

| 0–5 | 6–10 | 11–15 | 16–20 | 21–30 | 31 |
|---|---|---|---|---|---|
| 31 | S | A | B | 438 | 0 |

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 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 **r**S are sent to the device identified by EAR[RID], bypassing the cache.

If EAR[E] = 0, a DSI exception is generated. If EA is not a multiple of four, 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.

The **ecowx** instruction is supported for EAs that reference memory segments in which SR[T] = 0), and for EAs mapped by DBATs. If this instruction is executed when MSR[DR] = 0 (real addressing mode), results are boundedly undefined.

**ecowx** is treated as a store from the addressed byte with respect to address translation, memory protection, referenced and changed bit recording, and the ordering performed by **eieio**. Software synchronization is required 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 architecture.

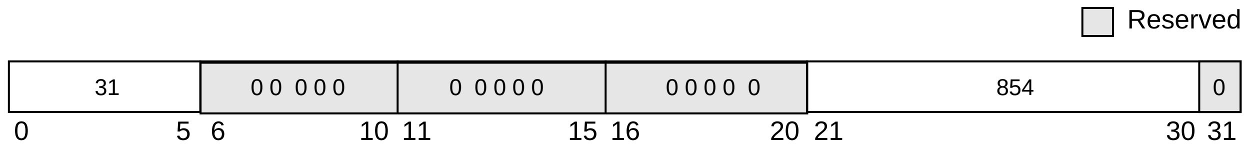
Other registers altered:

- None

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| VEA | | √ | X |

## eieio eieio

Enforce In-Order Execution of I/O



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 a **dcba** instruction are ordered like a store. 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.

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.

With the exception of **dcbz** and **dcba**, **eieio** does not affect the order of cache operations (whether caused explicitly by execution of a cache management instruction, or implicitly by the cache coherency mechanism). See Chapter 5, “Cache Model and Memory Coherency.” The **eieio** instruction does not affect the order of accesses in one set with respect to accesses in the other set.

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 5.2.1, “Memory Access Ordering”), **eieio** is needed for such memory only when loads must be ordered with respect to stores or with respect to other loads.

Note that **eieio** 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, **eieio** broadcasts signals to that buffer that previous loads/stores (marked caching-inhibited, guarded, or write-through required) must complete before any following loads/stores (marked caching-inhibited, guarded, or write-through required).

Other registers altered:

- None

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| VEA | | | X |

## **eqv***x* **eqv***x*

Equivalent

| | | |
|---|---|---|
| **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 |

```
rA ← (rS) ≡ (rB)
```

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)

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | X |

## extsb*x* extsb*x*

Extend Sign Byte

| | | |
|---|---|---|
| **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[24]
rA[24-31] ← rS[24-31]
rA[0–23] ← (24)S
```

The contents of **r**S[24–31] are placed into **r**A[24–31]. Bit 24 of **r**S is placed into **r**A[0–23].

Other registers altered:

- Condition Register (CR0 field):
  Affected: LT, GT, EQ, SO (if Rc = 1)

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | X |

## extsh*x* extsh*x*

Extend Sign Half Word

| | | |
|---|---|---|
| **extsh** | **r**A,**r**S | (Rc = 0) |
| **extsh.** | **r**A,**r**S | (Rc = 1) |

[POWER mnemonics: **exts**, **exts.**]

▨ Reserved

| 31 | S | A | 0 0 0 0 0 | 922 | Rc |
|---|---|---|---|---|---|
| 0 5 | 6 10 | 11 15 | 16 20 | 21 30 | 31 |

```
S ← rS[16]
rA[16-31] ← rS[16-31]
rA[0-15] ← (16)S
```

The contents of **r**S[16–31] are placed into **r**A[16–31]. Bit 16 of **r**S is placed into **r**A[0–15].

Other registers altered:

- Condition Register (CR0 field):
  Affected: LT, GT, EQ, SO (if Rc = 1)

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | X |

## fabs*x* fabs*x*

Floating Absolute Value

| | | |
|---|---|---|
| **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 that 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)

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | X |

## **fadd***x* **fadd***x*

Floating Add (Double-Precision)

| | | |
|---|---|---|
| **fadd** | **fr**D**,fr**A**,fr**B | (Rc = 0) |
| **fadd.** | **fr**D**,fr**A**,fr**B | (Rc = 1) |

[POWER mnemonics: **fa**, **fa.**]

▨ 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

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | A |

## fadds*x* fadds*x*

Floating Add Single

| | | |
|---|---|---|
| **fadds** | **fr**D,**fr**A,**fr**B | (Rc = 0) |
| **fadds.** | **fr**D,**fr**A,**fr**B | (Rc = 1) |

▨ Reserved

| 59 | 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 msb 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

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | A |

## fcmpo fcmpo

Floating Compare Ordered

**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 ← 0b0001
else if (frA)< (frB) thenc ← 0b1000
else if (frA)> (frB) thenc ← 0b0100
else          c ← 0b0010

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

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | X |

## fcmpu fcmpu

Floating Compare Unordered

**fcmpu** **crf**D,**fr**A,**fr**B

▨ Reserved

| 63 | crfD | 0 0 | A | B | 0 0 0 0 0 0 0 0 0 0 | 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 ← 0b0001
else if (frA) < (frB) thenc ← 0b1000
else if (frA) > (frB) thenc ← 0b0100
else        c ← 0b0010

FPCC ← c
CR[4 * crfD–4 * 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

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | X |

## fctiw*x* fctiw*x*

Floating Convert to Integer Word

| | | |
|---|---|---|
| **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 Section D.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.

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

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | X |

## fctiwz*x* fctiwz*x*

Floating Convert to Integer Word with Round toward Zero

| | | |
|---|---|---|
| **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 Section D.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.

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

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | X |

## fdiv*x* fdiv*x*

Floating Divide (Double-Precision)

| | | |
|---|---|---|
| **fdiv** | **fr**D**,fr**A**,fr**B | (Rc = 0) |
| **fdiv.** | **fr**D**,fr**A**,fr**B | (Rc = 1) |

[POWER mnemonics: **fd**, **fd.**]

▨ 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 msb 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

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | A |

## fdivs*x* fdivs*x*

Floating Divide Single

| | | |
|---|---|---|
| **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 msb 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

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | A |

## fmadd*x* fmadd*x*

Floating Multiply-Add (Double-Precision)

| | | |
|---|---|---|
| **fmadd** | **fr**D**,fr**A**,fr**C**,fr**B | (Rc = 0) |
| **fmadd.** | **fr**D**,fr**A**,fr**C**,fr**B | (Rc = 1) |

[POWER mnemonics: **fma**, **fma.**]

| | | | | | | |
|---|---|---|---|---|---|---|
| 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 msb 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

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | A |

## fmadds*x* fmadds*x*

Floating Multiply-Add Single

| | | |
|---|---|---|
| **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 msb 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

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | A |

## fmr*x* fmr*x*

Floating Move Register

| | | |
|---|---|---|
| **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 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)

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | X |

## fmsub*x* fmsub*x*

Floating Multiply-Subtract (Double-Precision)

| | | |
|---|---|---|
| **fmsub** | **fr**D,**fr**A,**fr**C,**fr**B | (Rc = 0) |
| **fmsub.** | **fr**D,**fr**A,**fr**C,**fr**B | (Rc = 1) |

[POWER mnemonics: **fms**, **fms.**]

| 0–5 | 6–10 | 11–15 | 16–20 | 21–25 | 26–30 | 31 |
|---|---|---|---|---|---|---|
| 63 | D | A | B | C | 28 | Rc |

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 msb 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

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | A |

## fmsubs*x* fmsubs*x*

Floating Multiply-Subtract Single

| | | |
|---|---|---|
| **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 msb 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

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | A |

## **fmul***x* **fmul***x*

Floating Multiply (Double-Precision)

| | | |
|---|---|---|
| **fmul** | **fr**D**,fr**A**,fr**C | (Rc = 0) |
| **fmul.** | **fr**D**,fr**A**,fr**C | (Rc = 1) |

[POWER mnemonics: **fm**, **fm.**]

▨ 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 floating-point operand in register **fr**A is multiplied by the floating-point operand in register **fr**C.

If the msb 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

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | A |

## fmuls*x* fmuls*x*

Floating Multiply Single

| | | |
|---|---|---|
| **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 floating-point operand in register **fr**A is multiplied by the floating-point operand in register **fr**C.

If the msb 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

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | A |

## fnabs*x* fnabs*x*

Floating Negative Absolute Value

| | | |
|---|---|---|
| **fnabs** | **fr**D,**fr**B | (Rc = 0) |
| **fnabs.** | **fr**D,**fr**B | (Rc = 1) |

▨ Reserved

| | | | | | |
|---|---|---|---|---|---|
| 63 | D | 0 0 0 0 0 | B | 136 | Rc |
| 0 5 | 6 10 | 11 15 | 16 20 | 21 25 | 26 30 | 31 |

The contents of register **fr**B with bit 0 set are placed into **fr**D.

Note that 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)

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | X |

## **fneg***x* **fneg***x*

Floating Negate

| | | |
|---|---|---|
| **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 contents of register **fr**B with bit 0 inverted are placed into **fr**D.

Note that 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)

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | X |

## fnmadd*x* fnmadd*x*

Floating Negative Multiply-Add (Double-Precision)

| | | |
|---|---|---|
| **fnmadd** | **fr**D,**fr**A,**fr**C,**fr**B | (Rc = 0) |
| **fnmadd.** | **fr**D,**fr**A,**fr**C,**fr**B | (Rc = 1) |

[POWER mnemonics: **fnma**, **fnma.**]

| 0–5 | 6–10 | 11–15 | 16–20 | 21–25 | 26–30 | 31 |
|---|---|---|---|---|---|---|
| 63 | D | A | B | C | 31 | Rc |

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 msb 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

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | A |

## fnmadds*x* fnmadds*x*

Floating Negative Multiply-Add Single

| | | |
|---|---|---|
| **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 msb 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

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | A |

## fnmsub*x* fnmsub*x*

Floating Negative Multiply-Subtract (Double-Precision)

| | | |
|---|---|---|
| **fnmsub** | **fr**D,**fr**A,**fr**C,**fr**B | (Rc = 0) |
| **fnmsub.** | **fr**D,**fr**A,**fr**C,**fr**B | (Rc = 1) |

[POWER mnemonics: **fnms**, **fnms.**]

| 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 msb 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

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | A |

## fnmsubs*x* fnmsubs*x*

Floating Negative Multiply-Subtract Single

| | | |
|---|---|---|
| **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 msb 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

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | A |

## **fres***x* **fres***x*

Floating Reciprocal Estimate Single

| | | |
|---|---|---|
| **fres** | **fr**D,**fr**B | (Rc = 0) |
| **fres.** | **fr**D,**fr**B | (Rc = 1) |

▨ Reserved

| 59 | D | 0 0000 ▨ | B | 000 00 ▨ | 24 | Rc |
|---|---|---|---|---|---|---|
| 0 5 | 6 10 | 11 15 | 16 20 | 21 25 | 26 30 | 31 |

A single-precision estimate of the reciprocal of the **fr**B value is placed in **fr**D. The estimate placed into **fr**D is correct to a precision of 1 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:

| Operand | Result | Exception |
|---|---|---|
| –∞ | –0 | None |
| –0 | –∞* | ZX |
| +0 | +∞* | ZX |
| +∞ | +0 | None |
| SNaN | QNaN** | VXSNAN |
| QNaN | QNaN | None |

**Notes**: * No result if FPSCR[ZE] = 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 and zero divide exceptions when FPSCR[ZE] = 1.

Note that the architecture makes no provision for a double-precision version of **fres***x*.

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

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | √ | A |

## **frsp***x* **frsp***x*

Floating Round to Single

| | | |
|---|---|---|
| **frsp** | **fr**D,**fr**B | (Rc = 0) |
| **frsp.** | **fr**D,**fr**B | (Rc = 1) |

▨ Reserved

| 0–5 | 6–10 | 11–15 | 16–20 | 21–30 | 31 |
|---|---|---|---|---|---|
| 63 | D | 0 0 0 0 0 | B | 12 | Rc |

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 Section D.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

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | X |

## frsqrte*x* frsqrte*x*

Floating Reciprocal Square Root Estimate

| | | |
|---|---|---|
| **frsqrte** | **fr**D,**fr**B | (Rc = 0) |
| **frsqrte.** | **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 | 26 | Rc |

A double-precision estimate of the reciprocal of the square root of the **fr**B value is placed into **fr**D. The estimate is correct to a precision of 1 part in 32. 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 **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:

| Operand | Result | Exception |
|---|---|---|
| –∞ | QNaN** | VXSQRT |
| <0 | QNaN** | VXSQRT |
| –0 | –∞* | ZX |
| +0 | +∞* | ZX |
| +∞ | +0 | None |
| SNaN | QNaN** | VXSNAN |
| QNaN | QNaN | None |

**Notes**: * No result if FPSCR[ZE] = 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 and zero divide exceptions when FPSCR[ZE] = 1. No single-precision version of the **frsqrte** instruction is provided; however, both **fr**B and **fr**D are representable in single-precision format.

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

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | √ | A |

## **fsel***x* **fsel***x*

Floating Select

| | | |
|---|---|---|
| **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 Section D.3, “Floating-Point Conversions,” and Section D.5, “Floating-Point Selection.”

This instruction is optional in the architecture.

Other registers altered:

- Condition Register (CR1 field):
  Affected: FX, FEX, VX, OX (if Rc = 1)

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | √ | A |

## fsqrt*x* fsqrt*x*

Floating Square Root (Double-Precision)

| | | |
|---|---|---|
| **fsqrt** | **fr**D,**fr**B | (Rc = 0) |
| **fsqrt.** | **fr**D,**fr**B | (Rc = 1) |

▨ Reserved

| 63 | D | 0 0000 | B | 000 00 | 22 | Rc |
|---|---|---|---|---|---|---|
| 0 5 | 6 10 | 11 15 | 16 20 | 21 25 | 26 30 | 31 |

The square root of the floating-point operand in register **fr**B is placed into register **fr**D.

If the msb 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 below:

| Operand | Result | Exception |
|---|---|---|
| –∞ | QNaN* | VXSQRT |
| <0 | QNaN* | VXSQRT |
| –0 | –0 | None |
| +∞ | +∞ | None |
| SNaN | QNaN* | VXSNAN |
| QNaN | QNaN | None |

**Notes**: * 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.

This instruction is optional in the 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

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | √ | A |

## fsqrts*x* fsqrts*x*

Floating Square Root Single

| | | |
|---|---|---|
| **fsqrts** | **fr**D,**fr**B | (Rc = 0) |
| **fsqrts.** | **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 | 22 | Rc |

The square root of the floating-point operand in register **fr**B is placed into register **fr**D.

If the msb 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 below.

| Operand | Result | Exception |
|---|---|---|
| –∞ | QNaN* | VXSQRT |
| <0 | QNaN* | VXSQRT |
| –0 | –0 | None |
| +∞ | +∞ | None |
| SNaN | QNaN* | VXSNAN |
| QNaN | QNaN | None |

**Notes**: * 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.

This instruction is optional in the 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

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | √ | A |

## fsub*x* fsub*x*

Floating Subtract (Double-Precision)

| | | |
|---|---|---|
| **fsub** | **fr**D**,fr**A**,fr**B | (Rc = 0) |
| **fsub.** | **fr**D**,fr**A**,fr**B | (Rc = 1) |

[POWER mnemonics: **fs**, **fs.**]

▨ 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 floating-point operand in register **fr**B is subtracted from the floating-point operand in register **fr**A. If the msb 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

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | A |

## fsubs*x* fsubs*x*

Floating Subtract Single

| | | |
|---|---|---|
| **fsubs** | **fr**D,**fr**A,**fr**B | (Rc = 0) |
| **fsubs.** | **fr**D,**fr**A,**fr**B | (Rc = 1) |

▨ Reserved

| 59 | D | A | B | 0 0 0 0 0 | 20 | Rc |
|---|---|---|---|---|---|---|
| 0 5 | 6 10 | 11 15 | 16 20 | 21 25 | 26 30 | 31 |

The floating-point operand in register **fr**B is subtracted from the floating-point operand in register **fr**A. If the msb 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

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | A |

## icbi icbi

Instruction Cache Block Invalidate

**icbi** **r**A,**r**B

▨ Reserved

| 31 | ▨ 0 0 0 0 0 | A | B | 982 | 0 |
|---|---|---|---|---|---|
| 0 5 | 6 10 | 11 15 | 16 20 | 21 30 | 31 |

EA 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

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| VEA | | | X |

## isync isync

Instruction Synchronize

**isync**

[POWER mnemonic: **ics**]

▨ Reserved

| 19 | 0 0 0 0 0 | 0 0 0 0 0 | 0 0 0 0 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 that all exceptions and the **rfi** instruction are also context synchronizing.

Other registers altered:

- None

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| VEA | | | XL |

## lbz lbz

Load Byte and Zero

**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 ← (24)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

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | D |

## lbzu lbzu

Load Byte and Zero with Update

**lbzu** **r**D,d(**r**A)

|  |  |  |  |
|---|---|---|---|
| 35 | D | A | d |
| 0 5 | 6 10 | 11 15 | 16 31 |

```
EA ← (rA) + EXTS(d)
rD ← (24)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

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA |  |  | D |

## lbzux lbzux

Load Byte and Zero with Update Indexed

**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 ← (24)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

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | X |

## lbzx lbzx

Load Byte and Zero Indexed

**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 ← (24)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

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | X |

## lfd lfd

Load Floating-Point Double

**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 double word in memory addressed by EA is placed into **fr**D.

Other registers altered:

- None

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | D |

## lfdu                                                                 lfdu

Load Floating-Point Double with Update

**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 double word 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

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | D |

## lfdux lfdux

Load Floating-Point Double with Update Indexed

**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 double word 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

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | X |

# lfdx lfdx

Load Floating-Point Double Indexed

**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 double word in memory addressed by EA is placed into **fr**D.

Other registers altered:

- None

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | X |

## lfs lfs

Load Floating-Point Single

**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 Section D.6, “Floating-Point Load Instructions”) and placed into **fr**D.

Other registers altered:

- None

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | D |

## lfsu                                                                                                    lfsu

Load Floating-Point Single with Update

**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 Section D.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

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | D |

## lfsux lfsux

Load Floating-Point Single with Update Indexed

**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 Section D.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

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | X |

# lfsx lfsx

Load Floating-Point Single Indexed

**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 Section D.6, “Floating-Point Load Instructions”) and placed into **fr**D.

Other registers altered:

- None

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | X |

## lha lha

Load Half Word Algebraic

**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 half word 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 msb of the loaded half word.

Other registers altered:

- None

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | D |

## lhau lhau

Load Half Word Algebraic with Update

**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 half word 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 msb of the loaded half 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

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | D |

## lhaux lhaux

Load Half Word Algebraic with Update Indexed

**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 half word 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 msb of the loaded half 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

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | X |

## lhax lhax

Load Half Word Algebraic Indexed

**lhax** **r**D,**r**A,**r**B

▨ Reserved

| 0–5 | 6–10 | 11–15 | 16–20 | 21–30 | 31 |
|---|---|---|---|---|---|
| 31 | D | A | B | 343 | ▨ 0 |

```
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 half word 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 msb of the loaded half word.

Other registers altered:

- None

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | X |

# lhbrx lhbrx

Load Half Word Byte-Reverse Indexed

**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 ← (16)0 || MEM(EA + 1, 1) || MEM(EA, 1)
```

EA is the sum (**r**A|0) + (**r**B). Bits 0–7 of the half word in memory addressed by EA are loaded into the low-order eight bits of **r**D. Bits 8–15 of the half word 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 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

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | X |

## lhz lhz

Load Half Word and Zero

**lhz** **r**D,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 ← (16)0 || MEM(EA, 2)
```

EA is the sum (**r**A|0) + d. The half word 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

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | D |

## lhzu lhzu

Load Half Word and Zero with Update

**lhzu** **r**D,d(**r**A)

| 41 | D | A | d |
|---|---|---|---|
| 0 5 | 6 10 | 11 15 | 16 31 |

```
EA ← rA + EXTS(d)
rD ← (16)0 || MEM(EA, 2)
rA ← EA
```

EA is the sum (**r**A) + d. The half word 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

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | D |

## lhzux lhzux

Load Half Word and Zero with Update Indexed

**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 ← (16)0 || MEM(EA, 2)
rA ← EA
```

EA is the sum (**r**A) + (**r**B). The half word 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

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | X |

## lhzx lhzx

Load Half Word and Zero Indexed

**lhzx** **r**D**,r**A**,r**B

▨ Reserved

| 0–5 | 6–10 | 11–15 | 16–20 | 21–30 | 31 |
|---|---|---|---|---|---|
| 31 | D | A | B | 279 | 0 ▨ |

```
if rA = 0 then b ← 0
else    b ← (rA)
EA ← b + (rB)
rD ← (16)0 || MEM(EA, 2)
```

EA is the sum (**r**A|0) + (**r**B). The half word 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

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | X |

## lmw lmw

Load Multiple Word

**lmw** **r**D,d(**r**A)

[POWER mnemonic: **lm**]

| 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) ← 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 GPRs **r**D through **r31**.

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 that, 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

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | D |

## lswi lswi

Load String Word Immediate

**lswi** **r**D,**r**A,NB

[POWER mnemonic: **lsi**]

▨ 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
else n ← NB
r ← rD – 1
i ← 0
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 = 32 then i ← 0
   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 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.

Bytes are loaded left to right in each register. The sequence of registers wraps around to **r0** if required. If the 4 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. See Section 6.4.3, “DSI Exception (0x00300).”

In some implementations this instruction is likely to have greater latency than a sequence of individual load or store instructions that produce the same results.

Other registers altered:

- None

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | X |

## lswx lswx

Load String Word Indexed

**lswx** **r**D,**r**A,**r**B

[POWER mnemonic: **lsx**]

▢ 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[25–31]
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 = 32 then i ← 0
  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 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.

Bytes are loaded left to right in each register. The register sequence wraps around through **r0** if required. If the 4 bytes of **r**D + *nr* – 1 are only partially filled, the unfilled low-order bytes 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 **r**A = 0, either the system illegal instruction error handler is invoked or 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 some conditions (for example, segment boundary crossing) the data alignment exception handler may be invoked. See Section 6.4.3, "DSI Exception (0x00300)."

In some implementations, **lswx** is likely to take much longer to execute than a sequence of individual load or store instructions that produce the same results.

Other registers altered: None

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | X |

## lwarx lwarx

Load Word and Reserve Indexed

**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 ← MEM(EA,4)
```

EA is the sum (**r**A|0) + (**r**B).

The word in memory addressed by EA is loaded into **r**D.

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 that the atomic sequence is not guaranteed, but its failure can be detected if CR0[EQ] = 0 after the **stwcx.** instruction.

Other registers altered:

- None

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | X |

## lwbrx lwbrx

Load Word Byte-Reverse Indexed

**lwbrx** **r**D,**r**A,**r**B

[POWER mnemonic: **lbrx**]

▨ Reserved

| 31 | D | A | B | 534 | ▨ 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 + 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 eight 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 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

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | X |

## lwz lwz

Load Word and Zero

**lwz** **r**D,d(**r**A)

[POWER mnemonic: **l**]

| 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 ← MEM(EA, 4)
```

EA is the sum (**r**A|0) + d. The word in memory addressed by EA is loaded into **r**D.

Other registers altered:

- None

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | D |

## lwzu　　　　　　　　　　　　　　　　　　　　　lwzu

Load Word and Zero with Update

**lwzu**　　　　　　　　　**r**D,d(**r**A)

[POWER mnemonic: **lu**]

|  |  |  |  |
|---|---|---|---|
| 33 | D | A | d |
| 0　　　5 | 6　　　10 | 11　　　15 | 16　　　31 |

```
EA ← rA + EXTS(d)
rD ← MEM(EA, 4)
rA ← EA
```

EA is the sum (**r**A) + d. The word 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

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA |  |  | D |

## lwzux lwzux

Load Word and Zero with Update Indexed

**lwzux** **r**D,**r**A,**r**B

[POWER mnemonic: **lux**]

▨ Reserved

|  |  |  |  |  |  |
|---|---|---|---|---|---|
| 31 | D | A | B | 55 | ▨ 0 |
| 0 5 | 6 10 | 11 15 | 16 20 | 21 30 | 31 |

```
EA ← (rA) + (rB)
rD ← MEM(EA, 4)
rA ← EA
```

EA is the sum (**r**A) + (**r**B). The word 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

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA |  |  | X |

## lwzx  lwzx

Load Word and Zero Indexed

**lwzx** **r**D**,r**A**,r**B

[POWER mnemonic: **lx**]

▨ 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 ← MEM(EA, 4)
```

EA is the sum (**r**A|0) + (**r**B). The word in memory addressed by EA is loaded into **r**D.

Other registers altered:

- None

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA |  |  | X |

## mcrf                                                                 mcrf

Move Condition Register Field

**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 * crfD) through (4 * crfD + 3)] ← CR[(4 * crfS) through (4 * crfS + 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

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | XL |

## mcrfs mcrfs

Move to Condition Register from FPSCR

**mcrfs** **crf**D,**crf**S

▨ Reserved

| 63 | crfD | 0 0 | crfS | 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)

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | X |

## mcrxr mcrxr

Move to Condition Register from XER

**mcrxr** **crf**D

▨ Reserved

| 31 | crfD | 0 0 | 0 0000 | 0000 0 | 512 | 0 |
|---|---|---|---|---|---|---|
| 0 5 | 6 8 | 9 10 | 11 15 | 16 20 | 21 30 | 31 |

```
CR[4 * crfD–4 * crfD + 3] ← XER[0–3]
XER[0–3] ← 0b0000
```

The contents of XER[0–3] are copied into the condition register field designated by **crf**D. All other fields of the condition register remain unchanged. XER[0–3] is cleared.

Other registers altered:

- Condition Register (CR field specified by operand **crf**D):
  Affected: LT, GT, EQ, SO
- XER[0–3]

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | X |

## mfcr mfcr

Move from Condition Register

**mfcr** **r**D

▨ Reserved

|  |  |  |  |  |  |
|---|---|---|---|---|---|
| 31 | D | 0 0000 | 0000 0 | 19 | 0 |
| 0 5 | 6 10 | 11 15 | 16 20 | 21 30 | 31 |

```
rD ← CR
```

The contents of the condition register (CR) are placed into **r**D.

Other registers altered:

- None

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA |  |  | X |

## mffs*x* mffs*x*

Move from FPSCR

| | | |
|---|---|---|
| **mffs** | **fr**D | (Rc = 0) |
| **mffs.** | **fr**D | (Rc = 1) |

▨ Reserved

| | | | | | |
|---|---|---|---|---|---|
| 63 | D | ▨ 0 0000 | ▨ 0000 0 | 583 | Rc |
| 0 5 | 6 10 | 11 15 | 16 20 | 21 30 | 31 |

```
frD[32-63] ← FPSCR
```

The contents of the FPSCR are placed into the low-order 32-bits of register **fr**D. The high-order 32-bits of register **fr**D are undefined.

Other registers altered:

- Condition Register (CR1 field):
  Affected: FX, FEX, VX, OX (if Rc = 1)

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | X |

## mfmsr mfmsr

Move from Machine State Register

**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

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| OEA | √ | | X |

## mfspr mfspr

Move from Special-Purpose Register

**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]
  rD ← SPR(n)
```

In the UISA, the SPR field denotes a special-purpose register, encoded as shown in Table 8-12. The contents of the designated special-purpose register are placed into **r**D.

**Table 8-12. UISA SPR Encodings for mfspr**

| SPR** | | | Register Name |
|---|---|---|---|
| **Decimal** | **spr[5–9]** | **spr[0–4]** | |
| 1 | 00000 | 00001 | XER |
| 8 | 00000 | 01000 | LR |
| 9 | 00000 | 01001 | CTR |

** Note that 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 8-12 (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 OEA, the SPR field denotes a special-purpose register, encoded as shown in Table 8-13. The contents of the designated SPR are placed into **r**D.

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 8-13 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 8-13, either an illegal instruction type program exception occurs or the results are boundedly undefined.

Other registers altered:

- None

**Table 8-13. 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 |
| 282 | 01000 | 11010 | EAR | Supervisor |
| 287 | 01000 | 11111 | PVR | Supervisor |
| 528 | 10000 | 10000 | IBAT0U | Supervisor |
| 529 | 10000 | 10001 | IBAT0L | Supervisor |
| 530 | 10000 | 10010 | IBAT1U | Supervisor |

**Table 8-13. OEA SPR Encodings for mfspr (continued)**

| SPR[1] | | | Register Name | Access |
|---|---|---|---|---|
| Decimal | spr[5–9] | spr[0–4] | | |
| 531 | 10000 | 10011 | IBAT1L | Supervisor |
| 532 | 10000 | 10100 | IBAT2U | Supervisor |
| 533 | 10000 | 10101 | IBAT2L | Supervisor |
| 534 | 10000 | 10110 | IBAT3U | Supervisor |
| 535 | 10000 | 10111 | IBAT3L | Supervisor |
| 536 | 10000 | 11000 | DBAT0U | Supervisor |
| 537 | 10000 | 11001 | DBAT0L | Supervisor |
| 538 | 10000 | 11010 | DBAT1U | Supervisor |
| 539 | 10000 | 11011 | DBAT1L | Supervisor |
| 540 | 10000 | 11100 | DBAT2U | Supervisor |
| 541 | 10000 | 11101 | DBAT2L | Supervisor |
| 542 | 10000 | 11110 | DBAT3U | Supervisor |
| 543 | 10000 | 11111 | DBAT3L | Supervisor |
| 1013 | 11111 | 10101 | DABR | Supervisor |

[1]Note that the order of the two 5-bit halves of the SPR number is reversed compared with actual instruction coding.

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.

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA/OEA | √* | | XFX |

* Note that **mfspr** is supervisor-level only if SPR[0] = 1.

## mfsr mfsr

Move from Segment Register

**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 |

```
rD ← SEGREG(SR)
```

The contents of segment register SR are placed into **r**D.

This is a supervisor-level instruction.

This instruction is defined only for 32-bit implementations; using it on a 64-bit implementation causes an illegal instruction type program exception.

Other registers altered:

- None

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| OEA | √ | | X |

## mfsrin mfsrin

Move from Segment Register Indirect

**mfsrin** **r**D,**r**B

▨ Reserved

| 31 | D | 0 0000 | B | 659 | 0 |
|---|---|---|---|---|---|
| 0–5 | 6–10 | 11–15 | 16–20 | 21–30 | 31 |

```
rD ← SEGREG(rB[0–3])
```

The contents of the segment register selected by bits 0–3 of **r**B are copied into **r**D.

This is a supervisor-level instruction.

This instruction is defined only for 32-bit implementations. Using it on a 64-bit implementation causes an illegal instruction type program exception.

Note that the **r**A field is not defined for the **mfsrin** instruction in the architecture. However, **mfsrin** performs the same function in the architecture as does the **mfsri** instruction in the POWER architecture (if **r**A = 0).

Other registers altered:

- None

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| OEA | √ | | X |

## mftb mftb

Move from Time Base

**mftb** **r**D,TBR

▨ Reserved

| 0–5 | 6–10 | 11–20 | 21–30 | 31 |
|---|---|---|---|---|
| 31 | D | tbr* | 371 | ▨ 0 |

*__Note:__ This is a split field.

```
n ← tbr[5–9] || tbr[0–4]
if n = 268 then
        rD ← TBL
else if n = 269 then
        rD ← TBU
```

The contents of TBL or TBU are copied into rD, as designated by the value in TBR, encoded as shown in Table 8-14.

**Table 8-14. TBR Encodings for mftb**

| TBR* | | | Register Name | Access |
|---|---|---|---|---|
| Decimal | tbr[5–9] | tbr[0–4] | | |
| 268 | 01000 | 01100 | TBL | User |
| 269 | 01000 | 01101 | TBU | User |

*Note that the order of the two 5-bit halves of the TBR number is reversed.

If the TBR contains a value not shown in Table 8-14, 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.

Some implementations may implement **mftb** and **mfspr** identically. See Section 2.2, “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** |

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| VEA | | | XFX |

## mtcrf mtcrf

Move to Condition Register Fields

**mtcrf** CRM,**r**S

▨ Reserved

|  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|
| 31 | S | ▨ 0 | CRM | ▨ 0 | 144 | ▨ 0 |
| 0 5 | 6 10 | 11 | 12 19 | 20 | 21 30 | 31 |

```
mask ← (4)(CRM[0]) || (4)(CRM[1]) ||... (4)(CRM[7])
CR ← (rS & mask) | (CR & ¬ mask)
```

The contents 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 **r**S.

Note that updating a subset of the eight fields of the condition register may have 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 |

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | XFX |

## mtfsb0*x* mtfsb0*x*

Move to FPSCR Bit 0

| | | |
|---|---|---|
| **mtfsb0** | **crb**D | (Rc = 0) |
| **mtfsb0.** | **crb**D | (Rc = 1) |

▨ Reserved

| 0–5 | 6–10 | 11–15 | 16–20 | 21–30 | 31 |
|---|---|---|---|---|---|
| 63 | crbD | 0 0000 | 0000 0 | 70 | Rc |

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.

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | X |

## mtfsb1*x* mtfsb1*x*

Move to FPSCR Bit 1

| | | |
|---|---|---|
| **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 0 0 0 0 | 0 0 0 0 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.

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | X |

## mtfsf*x* mtfsf*x*

Move to FPSCR Fields

| | | |
|---|---|---|
| **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.

Updating fewer than all eight fields of the FPSCR may have substantially poorer performance on some implementations than updating all the fields.

When FPSCR[0–3] is specified, bits 0 (FX) and 3 (OX) are set to the values of **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

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | XFL |

## mtfsfi*x* mtfsfi*x*

Move to FPSCR Field Immediate

| | | |
|---|---|---|
| **mtfsfi** | **crf**D,IMM | (Rc = 0) |
| **mtfsfi.** | **crf**D,IMM | (Rc = 1) |

▨ Reserved

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 63 | crfD | ▨ 0 0 | ▨ 0 0 0 0 0 | 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

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | X |

## mtmsr mtmsr

Move to Machine State Register

**mtmsr** **r**S

▨ Reserved

| | | | | | |
|---|---|---|---|---|---|
| 31 | S | 0 0000 ▨ | 0000 0 ▨ | 146 | 0 ▨ |
| 0 5 | 6 10 | 11 15 | 16 20 | 21 30 | 31 |

```
MSR ← (rS)
```

The contents of **r**S are placed into the MSR.

This is a supervisor-level instruction. It is also an execution synchronizing instruction except with respect to alterations to the POW and LE bits. Refer to Section 2.3.17, “Synchronization Requirements for Special Registers and for Lookaside Buffers,” for more information.

In addition, alterations to the MSR[EE] and MSR[RI] bits are effective as soon as the instruction completes. Thus if MSR[EE] = 0 and an external or decrementer exception is pending, executing an **mtmsr** instruction that sets MSR[EE] = 1 will cause the external or decrementer exception to be taken before the next instruction is executed, if no higher priority exception exists.

This instruction is defined only for 32-bit implementations. Using it on a 64-bit implementation causes an illegal instruction type program exception.

Other registers altered:

- MSR

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| OEA | √ | | X |

## mtspr mtspr

Move to Special-Purpose Register

**mtspr** SPR**,rS**

▨ 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]
  SPR(n) ← (rS)
```

In the UISA, 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. -

**Table 8-15. UISA SPR Encodings for mtspr**

| SPR** | | | Register Name |
|---|---|---|---|
| Decimal | spr[5–9] | spr[0–4] | |
| 1 | 00000 | 00001 | XER |
| 8 | 00000 | 01000 | LR |
| 9 | 00000 | 01001 | CTR |

** Note that 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 8-15, 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 8-15.

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 OEA, the SPR field denotes a special-purpose register, encoded as shown in Table 8-16. The contents of **r**S are placed into the designated special-purpose register. -

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 8-16 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 8-16, either an illegal instruction type program exception occurs or the results are boundedly undefined.

Other registers altered:

- See Table 8-16.

**Table 8-16. 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 |
| 282 | 01000 | 11010 | EAR | Supervisor |
| 284 | 01000 | 11100 | TBL | Supervisor |
| 285 | 01000 | 11101 | TBU | Supervisor |

## Table 8-16. OEA SPR Encodings for mtspr (continued)

| SPR[1] | | | Register Name | Access |
|---|---|---|---|---|
| Decimal | spr[5–9] | spr[0–4] | | |
| 528 | 10000 | 10000 | IBAT0U | Supervisor |
| 529 | 10000 | 10001 | IBAT0L | Supervisor |
| 530 | 10000 | 10010 | IBAT1U | Supervisor |
| 531 | 10000 | 10011 | IBAT1L | Supervisor |
| 532 | 10000 | 10100 | IBAT2U | Supervisor |
| 533 | 10000 | 10101 | IBAT2L | Supervisor |
| 534 | 10000 | 10110 | IBAT3U | Supervisor |
| 535 | 10000 | 10111 | IBAT3L | Supervisor |
| 536 | 10000 | 11000 | DBAT0U | Supervisor |
| 537 | 10000 | 11001 | DBAT0L | Supervisor |
| 538 | 10000 | 11010 | DBAT1U | Supervisor |
| 539 | 10000 | 11011 | DBAT1L | Supervisor |
| 540 | 10000 | 11100 | DBAT2U | Supervisor |
| 541 | 10000 | 11101 | DBAT2L | Supervisor |
| 542 | 10000 | 11110 | DBAT3U | Supervisor |
| 543 | 10000 | 11111 | DBAT3L | Supervisor |
| 1013 | 11111 | 10101 | DABR | Supervisor |

[1]Note that 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.

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA/OEA | √* | | XFX |

* Note that **mtspr** is supervisor-level only if SPR[0] = 1.

## mtsr

Move to Segment Register

**mtsr** SR,**r**S

▨ Reserved

| | | | | | | |
|---|---|---|---|---|---|---|
| 31 | S | 0 | SR | 0 0 0 0 0 | 210 | 0 |
| 0 5 | 6 10 | 11 | 12 15 | 16 20 | 21 30 | 31 |

```
SEGREG(SR) ← (rS)
```

The contents of **r**S are placed into SR.

This is a supervisor-level instruction.

This instruction is defined only for 32-bit implementations. Using it on a 64-bit implementation causes an illegal instruction type program exception.

Other registers altered:

- None

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| OEA | √ | | X |

## mtsrin mtsrin

Move to Segment Register Indirect

**mtsrin** **r**S,**r**B

[POWER mnemonic: **mtsri**]

▨ Reserved

| 31 | S | 0 0000 ▨ | B | 242 | 0 ▨ |
|---|---|---|---|---|---|
| 0 5 | 6 10 | 11 15 | 16 20 | 21 30 | 31 |

```
    SEGREG(rB[0–3]) ← (rS)
```

The contents of **r**S are copied to the segment register selected by bits 0–3 of **r**B.

This is a supervisor-level instruction.

This instruction is defined only for 32-bit implementations. Using it on a 64-bit implementation causes an illegal instruction type program exception.

Note that the architecture does not define the **r**A field for the **mtsrin** instruction. However, **mtsrin** performs the same function in the architecture as does the **mtsri** instruction in the POWER architecture (if **r**A = 0).

Other registers altered:

- None

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| OEA | √ | | X |

## mulhw*x* mulhw*x*

Multiply High Word

| | | |
|---|---|---|
| **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 * rB
rD ← prod
```

The 32-bit product is formed from the contents of **r**A and **r**B. The high-order 32 bits of the 64-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 32-bit result.

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | XO |

## mulhwu*x* mulhwu*x*

Multiply High Word Unsigned

| | | |
|---|---|---|
| **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 * rB
rD ← prod[0–31]
```

The 32-bit operands are the contents of **r**A and **r**B. The high-order 32 bits of the 64-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 32-bit result.

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | XO |

## mulli mulli

Multiply Low Immediate

**mulli** **r**D,**r**A,SIMM

[POWER mnemonic: **muli**]

| 07 | D | A | SIMM |
|---|---|---|---|
| 0 5 | 6 10 | 11 15 | 16 31 |

```
prod[0–48]← (rA) * SIMM
rD ← prod[16-48]
```

The 32-bit first operand is (**r**A). The 16-bit second operand is the value of the SIMM field. The low-order 32-bits of the 48-bit product of the operands are placed into **r**D.

Both the operands and the product are interpreted as signed integers. The low-order 32 bits of the product are calculated independently of whether the operands are treated as signed or unsigned 32-bit integers.

This instruction can be used with **mulhd***x* or **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 implementations.

Other registers altered:

- None

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | D |

## mullw*x* mullw*x*

Multiply Low Word

| | | |
|---|---|---|
| **mullw** | **r**D,**r**A,**r**B | (OE = 0 Rc = 0) |
| **mullw.** | **r**D,**r**A,**r**B | (OE = 0 Rc = 1) |
| **mullwo** | **r**D,**r**A,**r**B | (OE = 1 Rc = 0) |
| **mullwo.** | **r**D,**r**A,**r**B | (OE = 1 Rc = 1) |

[POWER mnemonics: **muls**, **muls.**, **mulso**, **mulso.**]

| 31 | D | A | B | OE | 235 | Rc |
|---|---|---|---|---|---|---|
| 0 5 | 6 10 | 11 15 | 16 20 | 21 | 22 30 | 31 |

```
rD ← rA * rB
```

The 32-bit operands are the contents 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.

The low-order 32 bits of the product are the correct 32-bit product for 32-bit implementations. The low-order 32-bits of the product are independent of whether the operands are regarded as signed or unsigned 32-bit integers.

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.

Note that 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 32-bit result.

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | XO |

## **nand***x* **nand***x*

NAND

| | | |
|---|---|---|
| **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.

**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)

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | X |

## **neg***x* **neg***x*

Negate

| | | |
|---|---|---|
| **neg** | **r**D,**r**A | (OE = 0 Rc = 0) |
| **neg.** | **r**D,**r**A | (OE = 0 Rc = 1) |
| **nego** | **r**D,**r**A | (OE = 1 Rc = 0) |
| **nego.** | **r**D,**r**A | (OE = 1 Rc = 1) |

▨ 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 **r**A contains the most negative 32-bit number (0x8000_0000), the result is the most negative 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)

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | XO |

## **nor***x* **nor***x*

NOR

| | | |
|---|---|---|
| **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 |

```
rA ← ¬ ((rS) | (rB))
```

The contents of **r**S are ORed with the contents of **r**B and the complemented result is placed into **r**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 |

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | X |

## or*x* or*x*

OR

| | | |
|---|---|---|
| **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 |

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | X |

## **orc***x* **orc***x*

OR with Complement

| | | |
|---|---|---|
| **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)

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | X |

## ori ori

OR Immediate

**ori** **r**A,**r**S,UIMM

[POWER mnemonic: **oril**]

| 24 | S | A | UIMM |
|---|---|---|---|
| 0 5 | 6 10 | 11 15 | 16 31 |

```
rA ← (rS) | ((16)0 || UIMM)
```

The contents of **r**S are ORed with 0x0000 || 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** |

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | D |

## oris oris

OR Immediate Shifted

**oris** **r**A,**r**S,UIMM

[POWER mnemonic: **oriu**]

| 25 | S | A | UIMM |
|---|---|---|---|
| 0 5 | 6 10 | 11 15 | 16 31 |

```
rA ← (rS) | (UIMM || (16)0)
```

The contents of **r**S are ORed with UIMM || 0x0000 and the result is placed into **r**A.

Other registers altered:

- None

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | D |

## rfi rfi

Return from Interrupt

▨ Reserved

| 19 | 00 000 | 0 0000 | 0000 0 | 50 | 0 |
|---|---|---|---|---|---|
| 0 5 | 6 10 | 11 15 | 16 20 | 21 30 | 31 |

```
MSR[16–23, 25–27, 30–31] ← SRR1[16–23, 25–27, 30–31]
NIA ←iea SRR0[0–29] || 0b00
```

Bits SRR1[16–23, 25–27, 30–31] 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[0–29] || 0b00. 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 that 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 **rfi**.

This is a supervisor-level, context synchronizing instruction. This instruction is defined only for 32-bit implementations. Using it on a 64-bit implementation causes an illegal instruction type program exception.

Other registers altered:

- MSR

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| OEA | √ | | XL |

# **rlwimi***x* **rlwimi***x*

Rotate Left Word Immediate then Mask Insert

| | | |
|---|---|---|
| **rlwimi** | **r**A**,r**S,SH**,**MB**,**ME | (Rc = 0) |
| **rlwimi.** | **r**A**,r**S,SH**,**MB**,**ME | (Rc = 1) |

[POWER mnemonics: **rlimi, rlimi**.]

| 20 | S | A | SH | MB | ME | Rc |
|---|---|---|---|---|---|---|
| 0 5 | 6 10 | 11 15 | 16 20 | 21 25 | 26 30 | 31 |

```
n ← SH
r ← ROTL(rS, n)
m ← MASK(MB, ME)
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 ME and 0 bits elsewhere. The rotated data is inserted into **r**A under control of the generated mask.

Note that **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 **r**S, into **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 **r**S, into **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 |

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | M |

## rlwinm*x* rlwinm*x*

Rotate Left Word Immediate then AND with Mask

| | | |
|---|---|---|
| **rlwinm** | **r**A**,r**S,SH**,**MB**,**ME | (Rc = 0) |
| **rlwinm.** | **r**A**,r**S,SH**,**MB**,**ME | (Rc = 1) |

[POWER mnemonics: **rlinm, rlinm.**]

| 21 | S | A | SH | MB | ME | Rc |
|---|---|---|---|---|---|---|
| 0 5 | 6 10 | 11 15 | 16 20 | 21 25 | 26 30 | 31 |

```
n ← SH
r ← ROTL(rS, n)
m ← MASK(MB, ME)
rA ← r & m
```

The **r**S[0–31] contents are rotated left the number of bits specified by SH. A mask formed with 1 bits from MB through ME and 0 bits elsewhere is ANDed with the rotated data and the result is placed into **r**A. **rlwinm** can extract, rotate, shift, and clear bit fields as follows:

- To extract an *n*-bit field that starts at bit position *b* in **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 **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 register contents left or right by *n* bits, set SH = *n* (32 – *n*), MB = 0, and ME = 31.
- To shift register contents right by *n* bits, set SH = 32 – *n*, MB = *n*, and ME = 31. To clear the high-order *b* bits of a register and shift the result left by *n* bits, set SH = *n*, MB = *b* – *n* and ME = 31 – *n*.
- To clear a register’s low-order *n* bits , set SH = 0, MB = 0, and ME = 31 – *n*.

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* |

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | M |

## **rlwnm***x* **rlwnm***x*

Rotate Left Word then AND with Mask

| | | |
|---|---|---|
| **rlwnm** | **r**A**,r**S**,r**B**,**MB**,**ME | (Rc = 0) |
| **rlwnm.** | **r**A**,r**S**,r**B**,**MB**,**ME | (Rc = 1) |

[POWER mnemonics: **rlnm, rlnm**.]

| 23 | S | A | B | MB | ME | Rc |
|---|---|---|---|---|---|---|
| 0 5 | 6 10 | 11 15 | 16 20 | 21 25 | 26 30 | 31 |

```
n ← rB[27-31]
r ← ROTL(rS, n)
m ← MASK(MB, ME)
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 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 **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 **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 **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.

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 | **rlwnmr**A**,r**S**,r**B**,0,31** |

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | M |

## sc sc

System Call

[POWER mnemonic: **svca**]

▨ Reserved

| 17 | 00 000 | 0 0000 | 0000 0000 0000 00 | 1 | 0 |
|---|---|---|---|---|---|
| 0 5 | 6 10 | 11 15 | 16 29 | 30 | 31 |

In the 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 OEA, the **sc** instruction does the following:

```
SRR0 ←iea CIA + 4
SRR1[1-4, 10-15] ← 0
SRR1[16–23, 25–27, 30–31] ← MSR[16–23, 25–27, 30–31]
MSR ← new_value (see below)
NIA ←iea base_ea + 0xC00 (see below)
```

The EA of the instruction following the **sc** instruction is placed into SRR0. Bits 16–23, 25–27, and 30–31 of the MSR are placed into the corresponding bits of SRR1, and bits 1-4 and 10-15 of SRR1 are set to undefined values. Note that 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 **rfi**.

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 0xC00 from the physical base address determined by the new setting of MSR[IP].

Other registers altered:

- SRR0
- SRR1
- MSR

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA/OEA | | | SC |

## slw*x* slw*x*

Shift Left Word

| | | |
|---|---|---|
| **slw** | **r**A,**r**S,**r**B | (Rc = 0) |
| **slw.** | **r**A,**r**S,**r**B | (Rc = 1) |

[POWER mnemonics: **sl, sl.**]

| 31 | S | A | B | 24 | Rc |
|---|---|---|---|---|---|
| 0 5 | 6 10 | 11 15 | 16 20 | 21 30 | 31 |

```
n ← rB[27-31]
r ← ROTL(rS, n)
```

If bit 26 of **r**B = 0, the contents of **r**S are shifted left the number of bits specified by **r**B[27–31]. Bits shifted out of position 0 are lost. Zeros are supplied to the vacated positions on the right. The 32-bit result is placed into **r**A. If bit 26 of **r**B = 1, 32 zeros are placed into **r**A.

Other registers altered:

- Condition Register (CR0 field):
  Affected: LT, GT, EQ, SO (if Rc = 1)

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | X |

## **sraw***x* **sraw***x*

Shift Right Algebraic Word

| | | |
|---|---|---|
| **sraw** | **r**A,**r**S,**r**B | (Rc = 0) |
| **sraw.** | **r**A,**r**S,**r**B | (Rc = 1) |

[POWER mnemonics: **sra, sra.**]

| 31 | S | A | B | 792 | Rc |
|---|---|---|---|---|---|
| 0 5 | 6 10 | 11 15 | 16 20 | 21 30 | 31 |

```
n ← rB[27-31]
r ← ROTL(rS, n)
if rB[26] = 0 then
m ← MASK(n )
else m ← (32)0
S ← rS
rA ← r & m | S & ¬ m
XER[CA] ← S & (r & ¬ m ≠ 0
```

If **r**B[26] = 0,then the contents of **r**S are shifted right the number of bits specified by **r**B[27–31]. Bits shifted out of position 31 are lost. The result is padded on the left with sign bits before being placed into **r**A. If **r**B[26] = 1, then **r**A is filled with 32 sign bits (bit 0) from **r**S. CR0 is set based on the value written into **r**A. XER[CA] is set if **r**S contains a negative number and any 1 bits are shifted out of position 31; otherwise XER[CA] is cleared. A shift amount of zero causes XER[CA] to be cleared.

Note that the **sraw** instruction, followed by **addze**, can by used to divide quickly by $2^n$. The setting of XER[CA] by **sraw** is independent of 32/64-bit mode.

Other registers altered:

- Condition Register (CR0 field):
  Affected: LT, GT, EQ, SO (if Rc = 1)
- XER:
  Affected: CA

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | X |

## **srawi***x* **srawi***x*

Shift Right Algebraic Word Immediate

| | | |
|---|---|---|
| **srawi** | **r**A,**r**S,SH | (Rc = 0) |
| **srawi.** | **r**A,**r**S,SH | (Rc = 1) |

[POWER mnemonics: **srai, srai.**]

| 31 | S | A | SH | 824 | Rc |
|---|---|---|---|---|---|
| 0 5 | 6 10 | 11 15 | 16 20 | 21 30 | 31 |

```
n ← SH
r ← ROTL(rS, 32 – n)
m← MASK(n )
S ← rS
rA ← r & m | S & ¬ m
XER[CA] ← S & ((r & ¬ m) ≠ 0)
```

The contents of **r**S are shifted right the number of bits specified by operand SH. Bits shifted out of position 31 are lost. The shifted value is sign-extended before being placed in **r**A. The 32-bit 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 31; otherwise XER[CA] is cleared. A shift amount of zero causes XER[CA] to be cleared.

Note that 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 32/64-bit mode.

Other registers altered:

- Condition Register (CR0 field):
  Affected: LT, GT, EQ, SO (if Rc = 1)
- XER:
  Affected: CA

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | X |

## srw*x* srw*x*

Shift Right Word

| | | |
|---|---|---|
| **srw** | **r**A,**r**S,**r**B | (Rc = 0) |
| **srw.** | **r**A,**r**S,**r**B | (Rc = 1) |

[POWER mnemonics: **sr, sr.**]

| 31 | S | A | B | 536 | Rc |
|---|---|---|---|---|---|
| 0 5 | 6 10 | 11 15 | 16 20 | 21 30 | 31 |

```
n ← rB[27-31]
r ← ROTL(rS, 32 – n)
```

The contents 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 31 are lost. Zeros are supplied to the vacated positions on the left. The result is placed into **r**A.

Other registers altered:

- Condition Register (CR0 field):
  Affected: LT, GT, EQ, SO (if Rc = 1)

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | X |

## stb stb

Store Byte

**stb** **r**S,d(**r**A)

| 38 | S | A | d |
|---|---|---|---|
| 0 5 | 6 10 | 11 15 | 16 31 |

```
if rA = 0 then b ← 0
else    b ← (rA)
EA ← b + EXTS(d)
MEM(EA, 1) ← rS[24-31]
```

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 EA.

Other registers altered:

- None

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | D |

## stbu stbu

Store Byte with Update

**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[24-31]
rA ← EA
```

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 EA.

EA is placed into **r**A.

If **r**A = 0, the instruction form is invalid.

Other registers altered:

- None

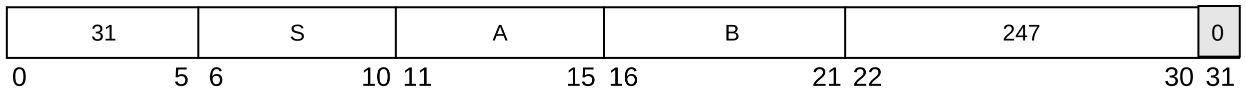
| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | D |

## stbux stbux

Store Byte with Update Indexed

**stbux** **r**S,**r**A,**r**B

▨ Reserved



```
EA ← (rA) + (rB)
MEM(EA, 1) ← rS[24-31]
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

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | X |

## stbx stbx

Store Byte Indexed

**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[24-31]
```

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

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | X |

## stfd stfd

Store Floating-Point Double

**stfd** **fr**S,d(**r**A)

|  |  |  |  |
|---|---|---|---|
| 54 | S | A | d |
| 0 5 | 6 10 | 11 15 | 16 30 31 |

```
if rA = 0 then b ← 0
else    b ← (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 double word in memory addressed by EA.

Other registers altered:

- None

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA |  |  | D |

## stfdu stfdu

Store Floating-Point Double with Update

**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 double 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

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | D |

## stfdux stfdux

Store Floating-Point Double with Update Indexed

**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 double 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

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | X |

## stfdx stfdx

Store Floating-Point Double Indexed

**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
else    b ← (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 double word in memory addressed by EA.

Other registers altered:

- None

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | X |

## stfiwx stfiwx

Store Floating-Point as Integer Word Indexed

**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
```

EA is the sum (**r**A|0) + (**r**B).

The low-order 32 bits of **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.

This instruction is defined as optional by the architecture to ensure backwards compatibility with earlier processors; however, it will likely be required for subsequent processors.

Other registers altered:

- None

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | √ | X |

## stfs stfs

Store Floating-Point Single

**stfs** **fr**S,d(**r**A)

| 52 | S | A | d |
|---|---|---|---|
| 0 5 | 6 10 | 11 15 | 16 31 |

```
if rA = 0 then b ← 0
else    b ← (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 Section D.7, “Floating-Point Store Instructions.”

Other registers altered:

- None

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | D |

## stfsu stfsu

Store Floating-Point Single with Update

**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 Section D.7, “Floating-Point Store Instructions.”

EA is placed into **r**A.

If **r**A = 0, the instruction form is invalid.

Other registers altered:

- None

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | D |

## stfsux stfsux

Store Floating-Point Single with Update Indexed

**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 Section D.7, “Floating-Point Store Instructions.”

EA is placed into **r**A.

If **r**A = 0, the instruction form is invalid.

Other registers altered:

- None

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | X |

## stfsx stfsx

Store Floating-Point Single Indexed

**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 Section D.7, “Floating-Point Store Instructions.”

Other registers altered:

- None

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | X |

## sth sth

Store Half Word

**sth** **r**S,d**(r**A**)**

| 44 | S | A | d |
|---|---|---|---|
| 0 5 | 6 10 | 11 15 | 16 31 |

```
if rA = 0 then b ← 0
else    b ← (rA)
EA ← b + EXTS(d)
MEM(EA, 2) ← rS[16-31]
```

EA is the sum (**r**A|0) + d. The contents of the low-order 16 bits of **r**S are stored into the half word in memory addressed by EA.

Other registers altered:

- None

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | D |

## sthbrx sthbrx

Store Half Word Byte-Reverse Indexed

**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[24-31] || rS[16-23]
```

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 half word in memory addressed by EA.

Other registers altered:

- None

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA |  |  | X |

## sthu sthu

Store Half Word with Update

**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[16-31]
rA ← EA
```

EA is the sum (**r**A) + d. The contents of the low-order 16 bits of **r**S are stored into the half 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

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | D |

## sthux sthux

Store Half Word with Update Indexed

**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[16-31]
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 half 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

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | X |

# sthx sthx

Store Half Word Indexed

**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
else    b ← (rA)
EA ← b + (rB)
MEM(EA, 2) ← rS[16-31]
```

EA is the sum (**r**A|0) + (**r**B). The contents of the low-order 16 bits of **r**S are stored into the half word in memory addressed by EA.

Other registers altered:

- None

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | X |

## stmw stmw

Store Multiple Word

**stmw** **r**S,d(**r**A)

[POWER mnemonic: **stm**]

| 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)
  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 GPRs **r**S through **r31**. 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 that, 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

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | D |

## stswi stswi

Store String Word Immediate

**stswi** **r**S,**r**A,NB

[POWER mnemonic: **stsi**]

▨ Reserved

| 31 | S | A | NB | 725 | 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
else    n ← NB
r ← rS – 1
i ← 32
do while n > 0
  if i = 32 then r ← r + 1 (mod 32)
  MEM(EA, 1) ← GPR(r)[i–i + 7]
  i ← i + 8
  if i = 64 then i ← 32
  EA ← EA + 1
  n ← n – 1
```

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. Bytes are stored left to right from each register. The sequence of registers wraps around through **r0** 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 that, 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

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | X |

## stswx stswx

Store String Word Indexed

**stswx** **r**S,**r**A,**r**B

[POWER mnemonic: **stsx**]

▨ Reserved

| 31 | S | A | B | 661 | 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[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. Bytes are stored left to right from each register. The sequence of registers wraps around through **r0** 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 that, 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

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | X |

## stw stw

Store Word

**stw** **r**S,d**(r**A**)**

[POWER mnemonic: **st**]

| 36 | S | A | d |
|---|---|---|---|
| 0 5 | 6 10 | 11 15 | 16 31 |

```
if rA = 0 then b ← 0
else    b ← (rA)
EA ← b + EXTS(d)
MEM(EA, 4) ← rS
```

EA is the sum (**r**A|0) + d. The contents of **r**S are stored into the word in memory addressed by EA.

Other registers altered:

- None

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | D |

## stwbrx stwbrx

Store Word Byte-Reverse Indexed

**stwbrx** **r**S,**r**A,**r**B

[POWER mnemonic: **stbrx**]

▨ Reserved

| | | | | | |
|---|---|---|---|---|---|
| 31 | S | A | B | 662 | ▨ 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) ← rS[24-31] || rS[16-23] || rS[8-15] || rS[0-7]
```

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

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | X |

## stwcx. stwcx.

Store Word Conditional Indexed

**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
    CR0 ← 0b00 || 0b1 || XER[SO]
  else
    u ← undefined 1-bit value
    if u then MEM(EA, 4) ← rS
    CR0 ← 0b00 || u || XER[SO]
  RESERVE ← 0
else
  CR0 ← 0b00 || 0b0 || XER[SO]
```

EA is the sum (**r**A|0) + (**r**B). If the reserved bit is set, **stwcx.** 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, **stwcx.** 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 and is cleared by any **stwcx.** to any address and 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.** 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 word in memory addressed by EA and the reservation is cleared.

If a reservation exists, but the address specified by the **stwcx.** is not the same as that specified by the instruction that established the reservation, the reservation is cleared and it is undefined whether the **r**S contents are stored into the word in memory addressed by EA.

If no reservation exists, the instruction completes without altering memory.

CR0 is set to reflect whether the store operation was performed as follows.

```
CR0[LT GT EQ S0] = 0b00 || store_performed || XER[SO]
```

If EA is not a multiple of four, either the system alignment exception handler is invoked or the results are boundedly undefined. 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

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | X |

## stwu stwu

Store Word with Update

**stwu** **r**S,d**(r**A**)**

[POWER mnemonic: **stu**]

| 37 | S | A | d |
|---|---|---|---|
| 0 5 | 6 10 | 11 15 | 16 31 |

```
EA ← (rA) + EXTS(d)
MEM(EA, 4) ← rS
rA ← EA
```

EA is the sum (**r**A) + d. The contents 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

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | D |

## stwux stwux

Store Word with Update Indexed

**stwux** **r**S,**r**A,**r**B

[POWER mnemonic: **stux**]

▨ 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
rA ← EA
```

EA is the sum (**r**A) + (**r**B). The contents 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

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | X |

# stwx stwx

Store Word Indexed

**stwx** **r**S,**r**A,**r**B

[POWER mnemonic: **stx**]

▨ Reserved

| 31 | S | A | B | 151 | 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) ← rS
```

EA is the sum (**r**A|0) + (**r**B). The contents of **r**S are is stored into the word in memory addressed by EA.

Other registers altered:

- None

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | X |

## **subf***x* **subf***x*

Subtract From

| | | |
|---|---|---|
| **subf** | **r**D,**r**A,**r**B | (OE = 0 Rc = 0) |
| **subf.** | **r**D,**r**A,**r**B | (OE = 0 Rc = 1) |
| **subfo** | **r**D,**r**A,**r**B | (OE = 1 Rc = 0) |
| **subfo.** | **r**D,**r**A,**r**B | (OE = 1 Rc = 1) |

| 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

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | XO |

# **subfc*x*** **subfc*x***

Subtract from Carrying

|  |  |  |
|---|---|---|
| **subfc** | **r**D,**r**A,**r**B | (OE = 0 Rc = 0) |
| **subfc.** | **r**D,**r**A,**r**B | (OE = 0 Rc = 1) |
| **subfco** | **r**D,**r**A,**r**B | (OE = 1 Rc = 0) |
| **subfco.** | **r**D,**r**A,**r**B | (OE = 1 Rc = 1) |

[POWER mnemonics: **sf, sf., sfo, sfo.**]

| 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 (see XER below).
- XER:

  Affected: CA

  Affected: SO, OV (if OE = 1)

Simplified mnemonics:

**subc** **r**D,**r**A,**r**B equivalent to **subfc** **r**D,**r**B,**r**A

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA |  |  | XO |

## **subfe***x* **subfe***x*

Subtract from Extended

| | | |
|---|---|---|
| **subfe** | **r**D,**r**A,**r**B | (OE = 0 Rc = 0) |
| **subfe.** | **r**D,**r**A,**r**B | (OE = 0 Rc = 1) |
| **subfeo** | **r**D,**r**A,**r**B | (OE = 1 Rc = 0) |
| **subfeo.** | **r**D,**r**A,**r**B | (OE = 1 Rc = 1) |

[POWER mnemonics: **sfe, sfe., sfeo, sfeo.**]

| 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 (see XER below).
- XER:

  Affected: CA

  Affected: SO, OV (if OE = 1)

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | XO |

## subfic subfic

Subtract from Immediate Carrying

**subfic** **r**D**,r**A**,**SIMM

[POWER mnemonic: **sfi**]

| | | | |
|---|---|---|---|
| 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

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | D |

## subfme*x* subfme*x*

Subtract from Minus One Extended

| | | |
|---|---|---|
| **subfme** | **r**D,**r**A | (OE = 0 Rc = 0) |
| **subfme.** | **r**D,**r**A | (OE = 0 Rc = 1) |
| **subfmeo** | **r**D,**r**A | (OE = 1 Rc = 0) |
| **subfmeo.** | **r**D,**r**A | (OE = 1 Rc = 1) |

[POWER mnemonics: **sfme, sfme., sfmeo, sfmeo.**]

▨ 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] + (32)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 (see XER below).
- XER:

  Affected: CA

  Affected: SO, OV (if OE = 1)

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | XO |

# subfze*x* subfze*x*

Subtract from Zero Extended

| | | |
|---|---|---|
| **subfze** | **r**D,**r**A | (OE = 0 Rc = 0) |
| **subfze.** | **r**D,**r**A | (OE = 0 Rc = 1) |
| **subfzeo** | **r**D,**r**A | (OE = 1 Rc = 0) |
| **subfzeo.** | **r**D,**r**A | (OE = 1 Rc = 1) |

[POWER mnemonics: **sfze, sfze., sfzeo, sfzeo.**]

▨ Reserved

| 31 | D | A | ▨ 0 0 0 0 0 | OE | 200 | 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)

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | XO |

## sync sync

Synchronize

[POWER mnemonic: **dcs**]

▨ Reserved

| | | | | | |
|---|---|---|---|---|---|
| 31 | 0 0 0 0 0 | 0 0 0 0 0 | 0 0 0 0 0 | 598 | 0 |
| 0 5 | 6 10 | 11 15 | 16 20 | 21 30 | 31 |

The **sync** instruction provides an ordering function for the effects of all instructions executed by a given processor. 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 external accesses caused by instructions preceding the **sync** instruction will have been performed with respect to all other mechanisms that access memory. For more information on how the **sync** instruction affects the VEA, refer to Chapter 5, “Cache Model and Memory Coherency.”

Multiprocessor implementations also send a **sync** address-only broadcast that is useful in some designs. For example, if a design has an external buffer that re-orders loads and stores for better bus efficiency, the **sync** broadcast signals to that buffer that previous loads/stores must be completed before any following loads/stores.

The **sync** instruction can be used to ensure that the results of all stores into a data structure, caused by store instructions executed in a “critical section” of a program, are seen by other processors before the data structure is seen as unlocked.

The functions performed by the **sync** instruction will normally take a significant amount of time to complete, so indiscriminate use of this instruction may adversely affect performance. In addition, the time required to execute **sync** may vary from one execution to another.

The **eieio** instruction may be more appropriate than **sync** for many cases.

This instruction is execution synchronizing. For more information on execution synchronization, see Section 4.1.5, “Synchronizing Instructions.”

Other registers altered:

- None

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | X |

## tlbia tlbia

Translation Lookaside Buffer Invalidate All

▨ 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 architecture.

Other registers altered:

- None

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| OEA | √ | √ | X |

## tlbie tlbie

Translation Lookaside Buffer Invalidate Entry

**tlbie** **r**B

[POWER mnemonic: **tlbi**]

▨ Reserved

| | | | | | |
|---|---|---|---|---|---|
| 31 | 0 0 0 0 0 | 0 0 0 0 0 | B | 306 | 0 |
| 0 5 | 6 10 | 11 15 | 16 20 | 21 30 | 31 |

```
VPS ← rB[4-19]
Identify TLB entries corresponding to VPS
Each such TLB entry ← invalid
```

VPS is the page index. EA is the contents of **r**B. If the translation lookaside buffer (TLB) contains an entry corresponding to EA, that entry is made invalid (that is, removed from the TLB).

Multiprocessing implementations send a **tlbie** address-only broadcast over the address bus to tell other processors to invalidate the same TLB entry in their TLBs.

The TLB search is done regardless of the settings of MSR[IR] and MSR[DR]. The search is done based on a portion of the logical page number within a segment, without reference to the segment registers. All entries matching the search criteria are invalidated.

Block address translation for EA, if any, is ignored. Refer to Section 7.6.3.4, “Synchronization of Memory Accesses and Referenced and Changed Bit Updates,” and Section 7.7.3, “Page Table Updates,” for other requirements associated with the use of this instruction.

This is a supervisor-level instruction and optional in the architecture.

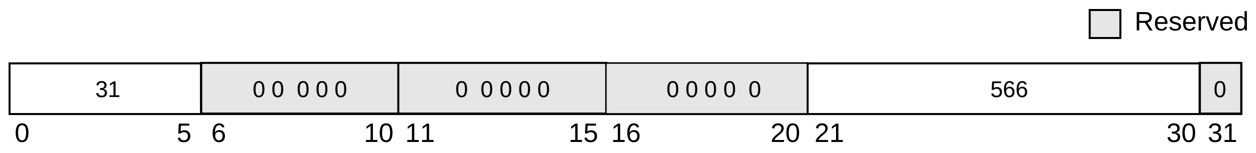
Other registers altered:

- None

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| OEA | √ | √ | X |

## tlbsync tlbsync

TLB Synchronize



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 7.6.3.4, “Synchronization of Memory Accesses and Referenced and Changed Bit Updates,” and Section 7.7.3, “Page Table Updates,” for other requirements associated with the use of this instruction.

This instruction is supervisor-level and optional in the architecture.

Other registers altered:

- None

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| OEA | √ | √ | X |

## tw

Trap Word

**tw** TO,**r**A,**r**B

[POWER mnemonic: **t**]

▨ Reserved

| 31 | TO | A | B | 4 | ▨ 0 |
|---|---|---|---|---|---|
| 0 5 | 6 10 | 11 15 | 16 20 | 21 30 | 31 |

```
a ← EXTS(rA)
b ← EXTS(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 TO bit 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** |

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | X |

## twi                                                                 twi

Trap Word Immediate

**twi** TO,**r**A,SIMM

[POWER mnemonic: **ti**]

| 03 | TO | A | SIMM |
|---|---|---|---|
| 0 5 | 6 10 | 11 15 | 16 31 |

```
a ← EXTS(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:

| | | | | |
|---|---|---|---|---|
| **twgti** | **r**A,value | equivalent to | **twi** | **8,r**A**,**value |
| **twllei** | **r**A,value | equivalent to | **twi** | **6,r**A**,**value |

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | D |

## **xor***x* **xor***x*

XOR

| | | |
|---|---|---|
| **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 |

```
xrA ← (rS) ⊕ (rB)
```

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)

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | X |

## xori xori

XOR Immediate

**xori** **r**A,**r**S,UIMM

[POWER mnemonic: **xoril**]

| 26 | S | A | UIMM |
|---|---|---|---|
| 0 5 | 6 10 | 11 15 | 16 31 |

```
rA ← (rS) ⊕ ((16)0 || UIMM)
```

The contents of **r**S are XORed with 0x0000 || UIMM and the result is placed into **r**A.

Other registers altered:

- None

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | D |

## xoris xoris

XOR Immediate Shifted

**xoris** **r**A,**r**S,UIMM

[POWER mnemonic: **xoriu**]

| 27 | S | A | UIMM |
|---|---|---|---|
| 0 5 | 6 10 | 11 15 | 16 31 |

```
rA ← (rS) ⊕ (UIMM || (16)0)
```

The contents of **r**S are XORed with UIMM || 0x0000 and the result is placed into **r**A.

Other registers altered:

- None

| Architecture Level | Supervisor Level | Optional | Form |
|---|---|---|---|
| UISA | | | D |

# Appendix A
Instruction Set Listings

This appendix lists the instructions by mnemonic, opcode, function, and form and includes a quick reference table with general information, such as the architecture level, privilege level, form, and whether the instruction is optional. The tables in the chapter are organized as follows:

- Section A.1, “Instructions Sorted by Mnemonic (Decimal and Hexadecimal)”
- Section A.2, “Instructions Sorted by Primary and Secondary Opcodes (Decimal and Hexadecimal)”
- Section A.3, “Instructions Sorted by Mnemonic (Binary)”
- Section A.4, “Instructions Sorted by Opcode (Binary)”
- Section A.5, “Instructions Grouped by Functional Categories”
- Section A.6, “Instructions Sorted by Form”
- Section A.7, “Instruction Set Legend”

Note that split fields, which represent the concatenation of sequences from left to right, are shown in lowercase. Reserved fields are shaded, as follows:

▨ Reserved bits

## A.1 Instructions Sorted by Mnemonic (Decimal and Hexadecimal)

Table A-1 lists instructions in alphabetical order by mnemonic, showing decimal and hexadecimal values of the primary opcode (0–5) and secondary opcode (21–31).

**Table A-1. Instructions by Mnemonic (Dec, Hex)**

| Mnemonic | 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 (0x1F) | D | A | B | OE | 0266 (0x10A) | Rc |
| **addc***x* | 31 (0x1F) | D | A | B | OE | 0010 (0x00A) | Rc |
| **adde***x* | 31 (0x1F) | D | A | B | OE | 0138 (0x08A) | Rc |
| **addi** | 14 (0xE) | D | A | SIMM | | | |
| **addic** | 12 (0xC) | D | A | SIMM | | | |

## Table A-1. Instructions by Mnemonic (Dec, Hex) (continued)

| Mnemonic | 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 |
|---|---|---|---|---|---|---|---|---|---|---|
| **addic.** | 13 (0xD) | D | | | A | SIMM | | | | |
| **addis** | 15 (0xF) | D | | | A | SIMM | | | | |
| **addme*x*** | 31 (0x1F) | D | | | A | 0_0000 | OE | 0234 (0x0EA) | | Rc |
| **addze*x*** | 31 (0x1F) | D | | | A | 0_0000 | OE | 0202 (0x0CA) | | Rc |
| **and*x*** | 31 (0x1F) | S | | | A | B | 0028 (0x01C) | | | Rc |
| **andc*x*** | 31 (0x1F) | S | | | A | B | 0060 (0x03C) | | | Rc |
| **andi.** | 28 (0x1C) | S | | | A | UIMM | | | | |
| **andis.** | 29 (0x1D) | S | | | A | UIMM | | | | |
| **b*x*** | 18 (0x12) | LI | | | | | | | AA | LK |
| **bc*x*** | 16 (0x10) | BO | | | BI | BD | | | AA | LK |
| **bcctr*x*** | 19 (0x13) | BO | | | BI | 0_0000 | 00528 (0x210) | | | LK |
| **bclr*x*** | 19 (0x13) | BO | | | BI | 0_0000 | 0016 (0x010) | | | LK |
| **cmp** | 31 (0x1F) | crfD | 0 | L | A | B | 0000 (0x000) | | | 0 |
| **cmpi** | 11 (0x0B) | crfD | 0 | L | A | SIMM | | | | |
| **cmpl** | 31 (0x1F) | crfD | 0 | L | A | B | 0032 (0x020) | | | 0 |
| **cmpli** | 10 (0x0A) | crfD | 0 | L | A | UIMM | | | | |
| **cntlzw*x*** | 31 (0x1F) | S | | | A | 0_0000 | 0026 (0x01A) | | | Rc |
| **crand** | 19 (0x13) | crbD | | | crbA | crbB | 0257 (0x101) | | | 0 |
| **crandc** | 19 (0x13) | crbD | | | crbA | crbB | 0129 (0x081) | | | 0 |
| **creqv** | 19 (0x13) | crbD | | | crbA | crbB | 0289 (0x121) | | | 0 |
| **crnand** | 19 (0x13) | crbD | | | crbA | crbB | 0225 (0x0E1) | | | 0 |
| **crnor** | 19 (0x13) | crbD | | | crbA | crbB | 0033 (0x21) | | | 0 |
| **cror** | 19 (0x13) | crbD | | | crbA | crbB | 0449 (0x1C1) | | | 0 |
| **crorc** | 19 (0x13) | crbD | | | crbA | crbB | 0417 (0x1A1) | | | 0 |
| **crxor** | 19 (0x13) | crbD | | | crbA | crbB | 0193 (0C1) | | | 0 |
| **dcba** [1] | 31 (0x1F) | 000_00 | | | A | B | 0758 (0x2F6) | | | 0 |
| **dcbf** | 31 (0x1F) | 000_00 | | | A | B | 0086 (0x056) | | | 0 |
| **dcbi** [2] | 31 (0x1F) | 000_00 | | | A | B | 0470 (0x1D6) | | | 0 |
| **dcbst** | 31 (0x1F) | 000_00 | | | A | B | 0054 (0x036) | | | 0 |
| **dcbt** | 31 (0x1F) | 000_00 | | | A | B | 0278 (0x116) | | | 0 |
| **dcbtst** | 31 (0x1F) | 000_00 | | | A | B | 0246 (0x0F6) | | | 0 |
| **dcbz** | 31 (0x1F) | 000_00 | | | A | B | 1014 (0x3F6) | | | 0 |
| **divw*x*** | 31 (0x1F) | D | | | A | B | OE | 0491 (0x1EB) | | Rc |
| **divwu*x*** | 31 (0x1F) | D | | | A | B | OE | 0459 (0x1CB) | | Rc |

**Table A-1. Instructions by Mnemonic (Dec, Hex) (continued)**

| Mnemonic | 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** [1] | 31 (0x1F) | D | | A | B | 0310 (0x136) | | 0 |
| **ecowx** [1] | 31 (0x1F) | S | | A | B | 0438 (0x1B6) | | 0 |
| **eieio** | 31 (0x1F) | 000_00 | | 00_000 | 0_0000 | 0854 (0x356) | | 0 |
| **eqv***x* | 31 (0x1F) | S | | A | B | 0284 (0x11C) | | Rc |
| **extsb***x* | 31 (0x1F) | S | | A | 0_0000 | 0954 (0x3BA) | | Rc |
| **extsh***x* | 31 (0x1F) | S | | A | 0_0000 | 0922 (0x39A) | | Rc |
| **fabs***x* | 63 (0x3F) | D | | 00_000 | B | 0264 (0x108) | | Rc |
| **fadd***x* | 63 (0x3F) | D | | A | B | 0000_0 | 0021 (0x015) | Rc |
| **fadds***x* | 59 (0x3B) | D | | A | B | 0000_0 | 0021 (0x015) | Rc |
| **fcmpo** | 63 (0x3F) | crfD | 00 | A | B | 0032 (0x020) | | 0 |
| **fcmpu** | 63 (0x3F) | crfD | 00 | A | B | 0000 (0x000) | | 0 |
| **fctiw***x* | 63 (0x3F) | D | | 00_000 | B | 0014 (0x00E) | | Rc |
| **fctiwz***x* | 63 (0x3F) | D | | 00_000 | B | 0015 (0x00F) | | Rc |
| **fdiv***x* | 63 (0x3F) | D | | A | B | 0000_0 | 0018 (0x012) | Rc |
| **fdivs***x* | 59 (0x3B) | D | | A | B | 0000_0 | 0018 (0x012) | Rc |
| **fmadd***x* | 63 (0x3F) | D | | A | B | C | 0029 (0x01D) | Rc |
| **fmadds***x* | 59 (0x3B) | D | | A | B | C | 0029 (0x01D) | Rc |
| **fmr***x* | 63 (0x3F) | D | | 00_000 | B | 0072 (0x48) | | Rc |
| **fmsub***x* | 63 (0x3F) | D | | A | B | C | 0028 (0x01C) | Rc |
| **fmsubs***x* | 59 (0x3B) | D | | A | B | C | 0028 (0x01C) | Rc |
| **fmul***x* | 63 (0x3F) | D | | A | 0_0000 | C | 0025 (0x019) | Rc |
| **fmuls***x* | 59 (0x3B) | D | | A | 0_0000 | C | 0025 (0x019) | Rc |
| **fnabs***x* | 63 (0x3F) | D | | 00_000 | B | 0136 (0x88) | | Rc |
| **fneg***x* | 63 (0x3F) | D | | 00_000 | B | 0040 (0x28) | | Rc |
| **fnmadd***x* | 63 (0x3F) | D | | A | B | C | 0031 (0x01F) | Rc |
| **fnmadds***x* | 59 (0x3B) | D | | A | B | C | 0031 (0x01F) | Rc |
| **fnmsub***x* | 63 (0x3F) | D | | A | B | C | 0030 (0x01E) | Rc |
| **fnmsubs***x* | 59 (0x3B) | D | | A | B | C | 0030 (0x01E) | Rc |
| **fres***x* [1] | 59 (0x3B) | D | | 00_000 | B | 0000_0 | 0024 (0x018) | Rc |
| **frsp***x* | 63 (0x3F) | D | | 00_000 | B | 0012 (0xC) | | Rc |
| **frsqrte***x* [1] | 63 (0x3F) | D | | 00_000 | B | 0000_0 | 0026 (0x01A) | Rc |
| **fsel***x* [1] | 63 (0x3F) | D | | A | B | C | 0023 (0x017) | Rc |
| **fsqrt***x* [1] | 63 (0x3F) | D | | 00_000 | B | 0000_0 | 0022 (0x016) | Rc |
| **fsqrts***x* [1] | 59 (0x3B) | D | | 00_000 | B | 0000_0 | 0022 (0x016) | Rc |

## Table A-1. Instructions by Mnemonic (Dec, Hex) (continued)

| Mnemonic | 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 |
|---|---|---|---|---|---|---|---|
| **fsub***x* | 63 (0x3F) | D | A | B | 0000_0 | 0020 (0x014) | Rc |
| **fsubs***x* | 59 (0x3B) | D | A | B | 0000_0 | 0020 (0x014) | Rc |
| **icbi** | 31 (0x1F) | 000_00 | A | B | 0982 (0x3D6) | | 0 |
| **isync** | 19 (0x13) | 000_00 | 00_000 | 0_0000 | 0150 (0x096) | | 0 |
| **lbz** | 34 (0x22) | D | A | d | | | |
| **lbzu** | 35 (0x23) | D | A | d | | | |
| **lbzux** | 31 (0x1F) | D | A | B | 0119 (0x077) | | 0 |
| **lbzx** | 31 (0x1F) | D | A | B | 087 (0x057) | | 0 |
| **lfd** | 50 (0x32) | D | A | d | | | |
| **lfdu** | 51 (0x33) | D | A | d | | | |
| **lfdux** | 31 (0x1F) | D | A | B | 0631 (0x277) | | 0 |
| **lfdx** | 31 (0x1F) | D | A | B | 0599 (0x257) | | 0 |
| **lfs** | 48 (0x30) | D | A | d | | | |
| **lfsu** | 49 (0x31) | D | A | d | | | |
| **lfsux** | 31 (0x1F) | D | A | B | 0567 (0x237) | | 0 |
| **lfsx** | 31 (0x1F) | D | A | B | 0535 (0x217) | | 0 |
| **lha** | 42 (0x2A) | D | A | d | | | |
| **lhau** | 43 (0x2B) | D | A | d | | | |
| **lhaux** | 31 (0x1F) | D | A | B | 0375 (0x177) | | 0 |
| **lhax** | 31 (0x1F) | D | A | B | 0343 (0x157) | | 0 |
| **lhbrx** | 31 (0x1F) | D | A | B | 0790 (0x316) | | 0 |
| **lhz** | 40 (0x28) | D | A | d | | | |
| **lhzu** | 41 (0x29) | D | A | d | | | |
| **lhzux** | 31 (0x1F) | D | A | B | 0311 (0x137) | | 0 |
| **lhzx** | 31 (0x1F) | D | A | B | 0279 (0x117) | | 0 |
| **lmw**[3] | 46 (0x2E) | D | A | d | | | |
| **lswi**[3] | 31 (0x1F) | D | A | NB | 0597 (0x255) | | 0 |
| **lswx**[3] | 31 (0x1F) | D | A | B | 0533 (0x215) | | 0 |
| **lwarx** | 31 (0x1F) | D | A | B | 0020 (0x014) | | 0 |
| **lwbrx** | 31 (0x1F) | D | A | B | 0534 (0x216) | | 0 |
| **lwz** | 32 (0x20) | D | A | d | | | |
| **lwzu** | 33 (0x21) | D | A | d | | | |
| **lwzux** | 31 (0x1F) | D | A | B | 0055 (0x037) | | 0 |
| **lwzx** | 31 (0x1F) | D | A | B | 0023 (0x017) | | 0 |

## Table A-1. Instructions by Mnemonic (Dec, Hex) (continued)

| Mnemonic | 0–5 | 6 | 7–8 | 9–10 | 11 | 12–13 | 14 | 15 | 16–19 | 20 | 21 | 22–25 | 26–30 | 31 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **mcrf** | 19 (0x13) | crfD | | 00 | crfS | | 00 | | 0_0000 | | 0000 (0x000) | | | 0 |
| **mcrfs** | 63 (0x3F) | crfD | | 00 | crfS | | 00 | | 0_0000 | | 0064 (0x040) | | | 0 |
| **mcrxr** | 31 (0x1F) | crfD | | 00 | 00_000 | | | | 0_0000 | | 0512 (0x200) | | | 0 |
| **mfcr** | 31 (0x1F) | D | | | 00_000 | | | | 0_0000 | | 0019 (0x013) | | | 0 |
| **mffs**_x_ | 63 (0x3F) | D | | | 00_000 | | | | 0_0000 | | 0583 (0x247) | | | Rc |
| **mfmsr** [2] | 31 (0x1F) | D | | | 00_000 | | | | 0_0000 | | 0083 (0x053) | | | 0 |
| **mfspr** [4] | 31 (0x1F) | D | | | spr | | | | | | 0339 (0x153) | | | 0 |
| **mfsr** [2] | 31 (0x1F) | D | | | 0 | SR | | | 0_0000 | | 0595 (0x099) | | | 0 |
| **mfsrin** [2] | 31 (0x1F) | D | | | 00_000 | | | | B | | 0659 (0x293) | | | 0 |
| **mftb** | 31 (0x1F) | D | | | tbr | | | | | | 0371 (0x173) | | | 0 |
| **mtcrf** | 31 (0x1F) | S | | | 0 | CRM | | | | 0 | 0144 (0x090) | | | 0 |
| **mtfsb0**_x_ | 63 (0x3F) | crbD | | | 00_000 | | | | 0_0000 | | 0070 (0x046) | | | Rc |
| **mtfsb1**_x_ | 63 (0x3F) | crbD | | | 00_000 | | | | 0_0000 | | 0038 (0x026) | | | Rc |
| **mtfsf**_x_ | 63 (0x3F) | 0 | FM | | | | | 0 | B | | 0711 (0x2C7) | | | Rc |
| **mtfsfi**_x_ | 63 (0x3F) | crfD | | 00 | 00_000 | | | | IMM | 0 | 0134 (0x086) | | | Rc |
| **mtmsr** [2] | 31 (0x1F) | S | | | 00_000 | | | | 0_0000 | | 0146 (0x092) | | | 0 |
| **mtspr** [4] | 31 (0x1F) | S | | | spr | | | | | | 0467 (0x1D3) | | | 0 |
| **mtsr** [2] | 31 (0x1F) | S | | | 0 | SR | | | 0_0000 | | 0210 (0x001) | | | 0 |
| **mtsrin** [2] | 31 (0x1F) | S | | | 00_000 | | | | B | | 0242 (0x0F2) | | | 0 |
| **mulhw**_x_ | 31(0x1F) | D | | | A | | | | B | | 0 | 0075 (0x04B) | | Rc |
| **mulhwu**_x_ | 31 (0x1F) | D | | | A | | | | B | | 0 | 0011 (0x00B) | | Rc |
| **mulli** | 07 (0x07) | D | | | A | | | | SIMM | | | | | |
| **mullw**_x_ | 31 (0x1F) | D | | | A | | | | B | | OE | 0235 (0x0EB) | | Rc |
| **nand**_x_ | 31 (0x1F) | S | | | A | | | | B | | 0476 (0x1DC) | | | Rc |
| **neg**_x_ | 31 (0x1F) | D | | | A | | | | 0_0000 | | OE | 0104 (0x068) | | Rc |
| **nor**_x_ | 31 (0x1F) | S | | | A | | | | B | | 0124 (0x07C) | | | Rc |
| **or**_x_ | 31 (0x1F) | S | | | A | | | | B | | 0444 (0x1BC) | | | Rc |
| **orc**_x_ | 31 (0x1F) | S | | | A | | | | B | | 0412 (0x19C) | | | Rc |
| **ori**[2] | 24 (0x18) | S | | | A | | | | UIMM | | | | | |
| **oris** | 25 (0x19) | S | | | A | | | | UIMM | | | | | |
| **rfi** [2] | 19 (0x13) | 000_00 | | | 00_000 | | | | 0_0000 | | 0050 (0x032) | | | 0 |
| **rlwimi**_x_ | 20 (0x14) | S | | | A | | | | SH | | MB | | ME | Rc |
| **rlwinm**_x_ | 21 (0x15) | S | | | A | | | | SH | | MB | | ME | Rc |
| **rlwnm**_x_ | 23 (0x17) | S | | | A | | | | B | | MB | | ME | Rc |

**Table A-1. Instructions by Mnemonic (Dec, Hex) (continued)**

| Mnemonic | 0–5 | 6–10 | 11–15 | 16–20 | 21 | 22–29 | 30 | 31 |
|---|---|---|---|---|---|---|---|---|
| **sc** | 17 (0x11) | 000_0000_0000_0000_0000_0000_00 | | | | | 1 | 0 |
| **slw***x* | 31 (0x1F) | S | A | B | 0024 (0x018) | | | Rc |
| **sraw***x* | 31 (0x1F) | S | A | B | 0792 (0x318) | | | Rc |
| **srawi***x* | 31 (0x1F) | S | A | SH | 0824 (0x338) | | | Rc |
| **srw***x* | 31 (0x1F) | S | A | B | 0536 (0x218) | | | Rc |
| **stb** | 38 (0x26) | S | A | d | | | | |
| **stbu** | 39 (0x27) | S | A | d | | | | |
| **stbux** | 31 (0x1F) | S | A | B | 0247 (0x0F7) | | | 0 |
| **stbx** | 31 (0x1F) | S | A | B | 0215 (0x0D7) | | | 0 |
| **stfd** | 54 (0x36) | S | A | d | | | | |
| **stfdu** | 55 (0x37) | S | A | d | | | | |
| **stfdux** | 31 (0x1F) | S | A | B | 0759 (0x2F7) | | | 0 |
| **stfdx** | 31 (0x1F) | S | A | B | 0727 (0x2D7) | | | 0 |
| **stfiwx** [1] | 31 (0x1F) | S | A | B | 0983 (0x3D7) | | | 0 |
| **stfs** | 52 (0x34) | S | A | d | | | | |
| **stfsu** | 53 (0x35) | S | A | d | | | | |
| **stfsux** | 31 (0x1F) | S | A | B | 0695 (0x2B7) | | | 0 |
| **stfsx** | 31 (0x1F) | S | A | B | 0663 (0x297) | | | 0 |
| **sth** | 44 (0x2C) | S | A | d | | | | |
| **sthbrx** | 31 (0x1F) | S | A | B | 0918 (0x396) | | | 0 |
| **sthu** | 45 (0x2D) | S | A | d | | | | |
| **sthux** | 31 (0x1F) | S | A | B | 0439 (0x1B7) | | | 0 |
| **sthx** | 31 (0x1F) | S | A | B | 0407 (0x197) | | | 0 |
| **stmw** [3] | 47 (0x2F) | S | A | d | | | | |
| **stswi** [3] | 31 (0x1F) | S | A | NB | 0725 (0x2D5) | | | 0 |
| **stswx** [3] | 31 (0x1F) | S | A | B | 0661 (0x295) | | | 0 |
| **stw** | 36 (0x24) | S | A | d | | | | |
| **stwbrx** | 31 (0x1F) | S | A | B | 0662 (0x296) | | | 0 |
| **stwcx.** | 31 (0x1F) | S | A | B | 0150 (0x096) | | | 1 |
| **stwu** | 37 (0x25) | S | A | d | | | | |
| **stwux** | 31 (0x1F) | S | A | B | 0183 (0x0B7) | | | 0 |
| **stwx** | 31 (0x1F) | S | A | B | 0151 (0x097) | | | 0 |
| **subf***x* | 31 (0x1F) | D | A | B | OE | 0040 (0x028) | | Rc |
| **subfc***x* | 31 (0x1F) | D | A | B | OE | 0008 (0x008) | | Rc |

**Table A-1. Instructions by Mnemonic (Dec, Hex) (continued)**

| Mnemonic | 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 |
|---|---|---|---|---|---|---|---|
| **subfe***x* | 31 (0x1F) | D | A | B | OE | 0136 (0x088) | Rc |
| **subfic** | 08 (0x08) | D | A | SIMM | | | |
| **subfme***x* | 31 (0x1F) | D | A | 0_0000 | OE | 0232 (0x0E8) | Rc |
| **subfze***x* | 31 (0x1F) | D | A | 0_0000 | OE | 0200 (0x0C8) | Rc |
| **sync** | 31 (0x1F) | 000_00 | 00_000 | 0_0000 | 0598 (0x256) | | 0 |
| **tlbia** [1] | 31 (0x1F) | 000_00 | 00_000 | 0_0000 | 0370 (0x172) | | 0 |
| **tlbie** [1, 2] | 31 (0x1F) | 000_00 | 00_000 | B | 0306 (0x132) | | 0 |
| **tlbld** [1, 2] | 31 (0x1F) | 000_00 | 00_000 | B | 0978 (0x3D2) | | 0 |
| **tlbli** [1, 2] | 31 (0x1F) | 000_00 | 00_000 | B | 1010 (0x3F2) | | 0 |
| **tlbsync** [1, 2] | 31 (0x1F) | 000_00 | 00_000 | 0_0000 | 0566 (0x236) | | 0 |
| **tw** | 31 (0x1F) | TO | A | B | 0004 (0x004) | | 0 |
| **twi** | 03 (0x03) | TO | A | SIMM | | | |
| **xor***x* | 31 (0x1F) | S | A | B | 0316 (0x13C) | | Rc |
| **xori** | 26 (0x1A) | S | A | UIMM | | | |
| **xoris** | 27 (0x1B) | S | A | UIMM | | | |

[1] Optional to the PowerPC architecture
[2] Supervisor-level instruction
[3] Load/store string/multiple instruction
[4] Supervisor- and user-level instruction

## A.2 Instructions Sorted by Primary and Secondary Opcodes (Decimal and Hexadecimal)

Table A-2 lists instructions by their primary (0–5) and secondary (21–31) opcodes in decimal and hexadecimal format.

**Table A-2. Instructions by Primary and Secondary Opcodes (Dec, Hex)**

| Mnemonic | 0 5 | 6 7 8 | 9 | 10 | 11 12 13 14 15 | 16 … 31 |
|---|---|---|---|---|---|---|
| **twi** | 03 (0x03) | TO | | | A | SIMM |
| **mulli** | 07 (0x07) | D | | | A | SIMM |
| **subfic** | 08 (0x08) | D | | | A | SIMM |
| **cmpli** | 10 (0x0A) | crfD | 0 | L | A | UIMM |
| **cmpi** | 11 (0x0B) | crfD | 0 | L | A | SIMM |
| **addic** | 12 (0xC) | D | | | A | SIMM |
| **addic.** | 13 (0xD) | D | | | A | SIMM |
| **addi** | 14 (0xE) | D | | | A | SIMM |
| **addis** | 15 (0xF) | D | | | A | SIMM |

## Table A-2. Instructions by Primary and Secondary Opcodes (Dec, Hex) (continued)

| Mnemonic | 0–5 | 6–8 | 9 | 10 | 11–13 | 14–15 | 16–20 | 21 | 22–25 | 26–29 | 30 | 31 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **bc***x* | 16 (0x10) | BO | | | BI | | BD | | | | AA | LK |
| **sc** | 17 (0x11) | 000_0000_0000_0000_0000_0000_00 | | | | | | | | | 1 | 0 |
| **b***x* | 18 (0x12) | LI | | | | | | | | | AA | LK |
| **mcrf** | 19 (0x13) | crfD | 00 | | crfS | 00 | 0_0000 | 0000 (0x000) | | | | 0 |
| **bclr***x* | 19 (0x13) | BO | | | BI | | 0_0000 | 0016 (0x010) | | | | LK |
| **crnor** | 19 (0x13) | crbD | | | crbA | | crbB | 0033 (0x21) | | | | 0 |
| **rfi** [1] | 19 (0x13) | 000_00 | | | 00_000 | | 0_0000 | 0050 (0x032) | | | | 0 |
| **crandc** | 19 (0x13) | crbD | | | crbA | | crbB | 0129 (0x081) | | | | 0 |
| **isync** | 19 (0x13) | 000_00 | | | 00_000 | | 0_0000 | 0150 (0x096) | | | | 0 |
| **crxor** | 19 (0x13) | crbD | | | crbA | | crbB | 0193 (0C1) | | | | 0 |
| **crnand** | 19 (0x13) | crbD | | | crbA | | crbB | 0225 (0x0E1) | | | | 0 |
| **crand** | 19 (0x13) | crbD | | | crbA | | crbB | 0257 (0x101) | | | | 0 |
| **creqv** | 19 (0x13) | crbD | | | crbA | | crbB | 0289 (0x121) | | | | 0 |
| **crorc** | 19 (0x13) | crbD | | | crbA | | crbB | 0417 (0x1A1) | | | | 0 |
| **cror** | 19 (0x13) | crbD | | | crbA | | crbB | 0449 (0x1C1) | | | | 0 |
| **bcctr***x* | 19 (0x13) | BO | | | BI | | 0_0000 | 0528 (0x210) | | | | LK |
| **rlwimi***x* | 20 (0x14) | S | | | A | | SH | MB | | ME | | Rc |
| **rlwinm***x* | 21 (0x15) | S | | | A | | SH | MB | | ME | | Rc |
| **rlwnm***x* | 23 (0x17) | S | | | A | | B | MB | | ME | | Rc |
| **ori** | 24 (0x18) | S | | | A | | UIMM | | | | | |
| **oris** | 25 (0x19) | S | | | A | | UIMM | | | | | |
| **xori** | 26 (0x1A) | S | | | A | | UIMM | | | | | |
| **xoris** | 27 (0x1B) | S | | | A | | UIMM | | | | | |
| **andi.** | 28 (0x1C) | S | | | A | | UIMM | | | | | |
| **andis.** | 29 (0x1D) | S | | | A | | UIMM | | | | | |
| **cmp** | 31 (0x1F) | crfD | 0 | L | A | | B | 0000 (0x000) | | | | 0 |
| **tw** | 31 (0x1F) | TO | | | A | | B | 0004 (0x004) | | | | 0 |
| **lvsl** [1] | 31 (0x1F) | vD | | | A | | B | 0006 (0x006) | | | | 0 |
| **lvebx** [1] | 31 (0x1F) | vD | | | A | | B | 0007 (0x007) | | | | 0 |
| **subfc***x* | 31 (0x1F) | D | | | A | | B | OE | 0008 (0x008) | | | Rc |
| **addc***x* | 31 (0x1F) | D | | | A | | B | OE | 0010 (0x00A) | | | Rc |
| **mulhwu***x* | 31 (0x1F) | D | | | A | | B | 0 | 0011 (0x00B) | | | Rc |
| **mfcr** | 31 (0x1F) | D | | | 00_000 | | 0_0000 | 0019 (0x013) | | | | 0 |
| **lwarx** | 31 (0x1F) | D | | | A | | B | 0020 (0x014) | | | | 0 |

## Table A-2. Instructions by Primary and Secondary Opcodes (Dec, Hex) (continued)

| Mnemonic | 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 |
|---|---|---|---|---|---|---|
| **lwzx** | 31 (0x1F) | D | A | B | 0023 (0x017) | 0 |
| **slw***x* | 31 (0x1F) | S | A | B | 0024 (0x018) | Rc |
| **cntlzw***x* | 31 (0x1F) | S | A | 0_0000 | 0026 (0x01A) | Rc |
| **and***x* | 31 (0x1F) | S | A | B | 0028 (0x01C) | Rc |
| **cmpl** | 31 (0x1F) | crfD 0 L | A | B | 0032 (0x020) | 0 |
| **lvsr**[1] | 31 (0x1F) | **v**D | A | B | 0038 (0x026) | 0 |
| **lvehx**[1] | 31 (0x1F) | **v**D | A | B | 0039 (0x027) | 0 |
| **subf***x* | 31 (0x1F) | D | A | B | OE 0040 (0x028) | Rc |
| **dcbst** | 31 (0x1F) | 000_00 | A | B | 0054 (0x036) | 0 |
| **lwzux** | 31 (0x1F) | D | A | B | 0055 (0x037) | 0 |
| **andc***x* | 31 (0x1F) | S | A | B | 0060 (0x03C) | Rc |
| **lvewx**[1] | 31 (0x1F) | **v**D | A | B | 0071 (0x047) | 0 |
| **mulhw***x* | 31(0x1F) | D | A | B | 0 0075 (0x04B) | Rc |
| **mfmsr**[1] | 31 (0x1F) | D | 00_000 | 0_0000 | 0083 (0x053) | 0 |
| **dcbf** | 31 (0x1F) | 000_00 | A | B | 0086 (0x056) | 0 |
| **lbzx** | 31 (0x1F) | D | A | B | 0087 (0x057) | 0 |
| **neg***x* | 31 (0x1F) | D | A | 0_0000 | OE 0104 (0x068) | Rc |
| **lbzux** | 31 (0x1F) | D | A | B | 0119 (0x077) | 0 |
| **nor***x* | 31 (0x1F) | S | A | B | 0124 (0x07C) | Rc |
| **subfe***x* | 31 (0x1F) | D | A | B | OE 0136 (0x088) | Rc |
| **adde***x* | 31 (0x1F) | D | A | B | OE 0138 (0x08A) | Rc |
| **mtcrf** | 31 (0x1F) | S | 0 CRM | 0 | 0144 (0x090) | 0 |
| **mtmsr**[1] | 31 (0x1F) | S | 00_000 | 0_0000 | 0146 (0x092) | 0 |
| **stwcx.** | 31 (0x1F) | S | A | B | 0150 (0x096) | 1 |
| **stwx** | 31 (0x1F) | S | A | B | 0151 (0x097) | 0 |
| **stwux** | 31 (0x1F) | S | A | B | 0183 (0x0B7) | 0 |
| **subfze***x* | 31 (0x1F) | D | A | 0_0000 | OE 0200 (0x0C8) | Rc |
| **addze***x* | 31 (0x1F) | D | A | 0_0000 | OE 0202 (0x0CA) | Rc |
| **mtsr**[1] | 31 (0x1F) | S | 0 SR | 0_0000 | 0210 (0x001) | 0 |
| **stbx** | 31 (0x1F) | S | A | B | 0215 (0x0D7) | 0 |
| **subfme***x* | 31 (0x1F) | D | A | 0_0000 | OE 0232 (0x0E8) | Rc |
| **addme***x* | 31 (0x1F) | D | A | 0_0000 | OE 0234 (0x0EA) | Rc |
| **mullw***x* | 31 (0x1F) | D | A | B | OE 0235 (0x0EB) | Rc |
| **mtsrin**[1] | 31 (0x1F) | S | 00_000 | B | 0242 (0x0F2) | 0 |

## Table A-2. Instructions by Primary and Secondary Opcodes (Dec, Hex) (continued)

| Mnemonic | 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 |
|---|---|---|---|---|---|---|
| **dcbtst** | 31 (0x1F) | 000_00 | A | B | 0246 (0x0F6) | 0 |
| **stbux** | 31 (0x1F) | S | A | B | 0247 (0x0F7) | 0 |
| **add***x* | 31 (0x1F) | D | A | B | OE 0266 (0x10A) | Rc |
| **dcbt** | 31 (0x1F) | 000_00 | A | B | 0278 (0x116) | 0 |
| **lhzx** | 31 (0x1F) | D | A | B | 0279 (0x117) | 0 |
| **eqv***x* | 31 (0x1F) | S | A | B | 0284 (0x11C) | Rc |
| **tlbie** [1, 2] | 31 (0x1F) | 000_00 | 00_000 | B | 0306 (0x132) | 0 |
| **eciwx** [2] | 31 (0x1F) | D | A | B | 0310 (0x136) | 0 |
| **lhzux** | 31 (0x1F) | D | A | B | 0311 (0x137) | 0 |
| **xor***x* | 31 (0x1F) | S | A | B | 0316 (0x13C) | Rc |
| **mfspr** [3] | 31 (0x1F) | D | spr | | 0339 (0x153) | 0 |
| **dst** [1] | 31 (0x1F) | 0 00 STRM | A | B | 0342 (0x156) | 0 |
| **dstt** [1] | 31 (0x1F) | 1 00 STRM | A | B | 0342 (0x156) | 0 |
| **lhax** | 31 (0x1F) | D | A | B | 0343 (0x157) | 0 |
| **lvxl** [1] | 31 (0x1F) | **v**D | A | B | 0359 (0x167) | 0 |
| **tlbia** [2] | 31 (0x1F) | 000_00 | 00_000 | 0_0000 | 0370 (0x172) | 0 |
| **mftb** | 31 (0x1F) | D | tbr | | 0371 (0x173) | 0 |
| **dstst** [1] | 31 (0x1F) | 0 00 STRM | A | B | 0374 (0x176) | 0 |
| **dststt** [1] | 31 (0x1F) | 1 00 STRM | A | B | 0374 (0x176) | 0 |
| **lhaux** | 31 (0x1F) | D | A | B | 0375 (0x177) | 0 |
| **sthx** | 31 (0x1F) | S | A | B | 0407 (0x197) | 0 |
| **orc***x* | 31 (0x1F) | S | A | B | 0412 (0x19C) | Rc |
| **ecowx** [2] | 31 (0x1F) | S | A | B | 0438 (0x1B6) | 0 |
| **sthux** | 31 (0x1F) | S | A | B | 0439 (0x1B7) | 0 |
| **or***x* | 31 (0x1F) | S | A | B | 0444 (0x1BC) | Rc |
| **divwu***x* | 31 (0x1F) | D | A | B | OE 0459 (0x1CB) | Rc |
| **mtspr** [3] | 31 (0x1F) | S | spr | | 0467 (0x1D3) | 0 |
| **dcbi** [1] | 31 (0x1F) | 000_00 | A | B | 0470 (0x1D6) | 0 |
| **nand***x* | 31 (0x1F) | S | A | B | 0476 (0x1DC) | Rc |
| **divw***x* | 31 (0x1F) | D | A | B | OE 0491 (0x1EB) | Rc |
| **mcrxr** | 31 (0x1F) | crfD 00 | 00_000 | 0_0000 | 0512 (0x200) | 0 |
| **lswx** [4] | 31 (0x1F) | D | A | B | 0533 (0x215) | 0 |
| **lwbrx** | 31 (0x1F) | D | A | B | 0534 (0x216) | 0 |
| **lfsx** | 31 (0x1F) | D | A | B | 0535 (0x217) | 0 |

## Table A-2. Instructions by Primary and Secondary Opcodes (Dec, Hex) (continued)

| Mnemonic | 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 |
|---|---|---|---|---|---|---|
| **srw***x* | 31 (0x1F) | S | A | B | 0536 (0x218) | Rc |
| **tlbsync**[1, 2] | 31 (0x1F) | 000_00 | 00_000 | 0_0000 | 0566 (0x236) | 0 |
| **lfsux** | 31 (0x1F) | D | A | B | 0567 (0x237) | 0 |
| **mfsr**[1] | 31 (0x1F) | D | 0 SR | 0_0000 | 0595 (0x099) | 0 |
| **lswi**[4] | 31 (0x1F) | D | A | NB | 0597 (0x255) | 0 |
| **sync** | 31 (0x1F) | 000_00 | 00_000 | 0_0000 | 0598 (0x256) | 0 |
| **lfdx** | 31 (0x1F) | D | A | B | 0599 (0x257) | 0 |
| **lfdux** | 31 (0x1F) | D | A | B | 0631 (0x277) | 0 |
| **mfsrin**[1] | 31 (0x1F) | D | 00_000 | B | 0659 (0x293) | 0 |
| **stswx**[4] | 31 (0x1F) | S | A | B | 0661 (0x295) | 0 |
| **stwbrx** | 31 (0x1F) | S | A | B | 0662 (0x296) | 0 |
| **stfsx** | 31 (0x1F) | S | A | B | 0663 (0x297) | 0 |
| **stfsux** | 31 (0x1F) | S | A | B | 0695 (0x2B7) | 0 |
| **stswi**[4] | 31 (0x1F) | S | A | NB | 0725 (0x2D5) | 0 |
| **stfdx** | 31 (0x1F) | S | A | B | 0727 (0x2D7) | 0 |
| **dcba**[2] | 31 (0x1F) | 000_00 | A | B | 0758 (0x2F6) | 0 |
| **stfdux** | 31 (0x1F) | S | A | B | 0759 (0x2F7) | 0 |
| **lhbrx** | 31 (0x1F) | D | A | B | 0790 (0x316) | 0 |
| **sraw***x* | 31 (0x1F) | S | A | B | 0792 (0x318) | Rc |
| **dss**[1] | 31 (0x1F) | 0 00 STRM | 00_000 | 0_0000 | 0822 (0x336) | 0 |
| **dssall**[1] | 31 (0x1F) | 1 00 STRM | 00_000 | 0_0000 | 0822 (0x336) | 0 |
| **srawi***x* | 31 (0x1F) | S | A | SH | 0824 (0x338) | Rc |
| **eieio** | 31 (0x1F) | 000_00 | 00_000 | 0_0000 | 0854 (0x356) | 0 |
| **sthbrx** | 31 (0x1F) | S | A | B | 0918 (0x396) | 0 |
| **extsh***x* | 31 (0x1F) | S | A | 0_0000 | 0922 (0x39A) | Rc |
| **extsb***x* | 31 (0x1F) | S | A | 0_0000 | 0954 (0x3BA) | Rc |
| **tlbld**[1, 2] | 31 (0x1F) | 000_00 | 00_000 | B | 0978 (0x3D2) | 0 |
| **icbi** | 31 (0x1F) | 000_00 | A | B | 0982 (0x3D6) | 0 |
| **stfiwx**[2] | 31 (0x1F) | S | A | B | 0983 (0x3D7) | 0 |
| **tlbli**[1,2] | 31 (0x1F) | 000_00 | 00_000 | B | 1010 (0x3F2) | 0 |
| **dcbz** | 31 (0x1F) | 000_00 | A | B | 1014 (0x3F6) | 0 |
| **lwz** | 32 (0x20) | D | A | d | | |
| **lwzu** | 33 (0x21) | D | A | d | | |
| **lbz** | 34 (0x22) | D | A | d | | |

## Table A-2. Instructions by Primary and Secondary Opcodes (Dec, Hex) (continued)

| Mnemonic | 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 |
|---|---|---|---|---|---|---|---|
| **lbzu** | 35 (0x23) | D | A | d | | | |
| **stw** | 36 (0x24) | S | A | d | | | |
| **stwu** | 37 (0x25) | S | A | d | | | |
| **stb** | 38 (0x26) | S | A | d | | | |
| **stbu** | 39 (0x27) | S | A | d | | | |
| **lhz** | 40 (0x28) | D | A | d | | | |
| **lhzu** | 41 (0x29) | D | A | d | | | |
| **lha** | 42 (0x2A) | D | A | d | | | |
| **lhau** | 43 (0x2B) | D | A | d | | | |
| **sth** | 44 (0x2C) | S | A | d | | | |
| **sthu** | 45 (0x2D) | S | A | d | | | |
| **lmw** [4] | 46 (0x2E) | D | A | d | | | |
| **stmw** [4] | 47 (0x2F) | S | A | d | | | |
| **lfs** | 48 (0x30) | D | A | d | | | |
| **lfsu** | 49 (0x31) | D | A | d | | | |
| **lfd** | 50 (0x32) | D | A | d | | | |
| **lfdu** | 51 (0x33) | D | A | d | | | |
| **stfs** | 52 (0x34) | S | A | d | | | |
| **stfsu** | 53 (0x35) | S | A | d | | | |
| **stfd** | 54 (0x36) | S | A | d | | | |
| **stfdu** | 55 (0x37) | S | A | d | | | |
| **fdivs***x* | 59 (0x3B) | D | A | B | 0000_0 | 0018 (0x012) | Rc |
| **fsubs***x* | 59 (0x3B) | D | A | B | 0000_0 | 0020 (0x014) | Rc |
| **fadds***x* | 59 (0x3B) | D | A | B | 0000_0 | 0021 (0x015) | Rc |
| **fsqrts***x* [2] | 59 (0x3B) | D | 00_000 | B | 0000_0 | 0022 (0x016) | Rc |
| **fres***x* [2] | 59 (0x3B) | D | 00_000 | B | 0000_0 | 0024 (0x018) | Rc |
| **fmuls***x* | 59 (0x3B) | D | A | 0_0000 | C | 0025 (0x019) | Rc |
| **fmsubs***x* | 59 (0x3B) | D | A | B | C | 0028 (0x01C) | Rc |
| **fmadds***x* | 59 (0x3B) | D | A | B | C | 0029 (0x01D) | Rc |
| **fnmsubs***x* | 59 (0x3B) | D | A | B | C | 0030 (0x01E) | Rc |
| **fnmadds***x* | 59 (0x3B) | D | A | B | C | 0031 (0x01F) | Rc |
| **fcmpu** | 63 (0x3F) | crfD 00 | A | B | 0000 (0x000) | | 0 |
| **frsp***x* | 63 (0x3F) | D | 00_000 | B | 0012 (0xC) | | Rc |
| **fctiw***x* | 63 (0x3F) | D | 00_000 | B | 0014 (0x00E) | | Rc |

**Table A-2. Instructions by Primary and Secondary Opcodes (Dec, Hex) (continued)**

| Mnemonic | 0–5 | 6–10 | 11–15 | 16–20 | 21–25 | 26–30 | 31 |
|---|---|---|---|---|---|---|---|
| **fctiwz***x* | 63 (0x3F) | D | 00_000 | B | 0015 (0x00F) | | Rc |
| **fdiv***x* | 63 (0x3F) | D | A | B | 0000_0 | 0018 (0x012) | Rc |
| **fsub***x* | 63 (0x3F) | D | A | B | 0000_0 | 0020 (0x014) | Rc |
| **fadd***x* | 63 (0x3F) | D | A | B | 0000_0 | 0021 (0x015) | Rc |
| **fsqrt***x* [2] | 63 (0x3F) | D | 00_000 | B | 0000_0 | 0022 (0x016) | Rc |
| **fsel***x* [2] | 63 (0x3F) | D | A | B | C | 0023 (0x017) | Rc |
| **fmul***x* | 63 (0x3F) | D | A | 0_0000 | C | 0025 (0x019) | Rc |
| **frsqrte***x* [2] | 63 (0x3F) | D | 00_000 | B | 0000_0 | 0026 (0x01A) | Rc |
| **fmsub***x* | 63 (0x3F) | D | A | B | C | 0028 (0x01C) | Rc |
| **fmadd***x* | 63 (0x3F) | D | A | B | C | 0029 (0x01D) | Rc |
| **fnmsub***x* | 63 (0x3F) | D | A | B | C | 0030 (0x01E) | Rc |
| **fnmadd***x* | 63 (0x3F) | D | A | B | C | 0031 (0x01F) | Rc |
| **fcmpo** | 63 (0x3F) | crfD 00 | A | B | 0032 (0x020) | | 0 |
| **mtfsb1***x* | 63 (0x3F) | crbD | 00_000 | 0_0000 | 0038 (0x026) | | Rc |
| **fneg***x* | 63 (0x3F) | D | 00_000 | B | 0040 (0x28) | | Rc |
| **mcrfs** | 63 (0x3F) | crfD 00 | crfS 00 | 0_0000 | 0064 (0x040) | | 0 |
| **mtfsb0***x* | 63 (0x3F) | crbD | 00_000 | 0_0000 | 0070 (0x046) | | Rc |
| **fmr***x* | 63 (0x3F) | D | 00_000 | B | 0072 (0x48) | | Rc |
| **mtfsfi***x* | 63 (0x3F) | crfD 00 | 00_000 | IMM 0 | 0134 (0x086) | | Rc |
| **fnabs***x* | 63 (0x3F) | D | 00_000 | B | 0136 (0x88) | | Rc |
| **fabs***x* | 63 (0x3F) | D | 00_000 | B | 0264 (0x108) | | Rc |
| **mffs***x* | 63 (0x3F) | D | 00_000 | 0_0000 | 0583 (0x247) | | Rc |
| **mtfsf***x* | 63 (0x3F) | 0 FM 0 | | B | 0711 (0x2C7) | | Rc |

[1] Supervisor-level instruction
[2] Optional to the PowerPC architecture
[3] Supervisor- and user-level instructions
[4] Load/store string/multiple instruction

## A.3 Instructions Sorted by Mnemonic (Binary)

Table A-3 lists instructions in alphabetical order by mnemonic with binary values.

**Table A-3. Instructions by Mnemonic (Binary)**

| Mnemonic | 0–5 | 6–10 | 11–15 | 16–20 | 21 | 22–30 | 31 |
|---|---|---|---|---|---|---|---|
| **add***x* | 011111 | D | A | B | OE | 100001 010 | Rc |
| **addc***x* | 011111 | D | A | B | OE | 000001010 | Rc |

## Table A-3. Instructions by Mnemonic (Binary) (continued)

| Mnemonic | 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 |
|---|---|---|---|---|---|---|---|---|---|---|
| **adde*x*** | 011111 | D | | | A | B | OE | 010001010 | | Rc |
| **addi** | 001110 | D | | | A | SIMM | | | | |
| **addic** | 001100 | D | | | A | SIMM | | | | |
| **addic.** | 001101 | D | | | A | SIMM | | | | |
| **addis** | 001111 | D | | | A | SIMM | | | | |
| **addme*x*** | 011111 | D | | | A | 0_0000 | OE | 11101010 | | Rc |
| **addze*x*** | 011111 | D | | | A | 0_0000 | OE | 11001010 | | Rc |
| **and*x*** | 011111 | S | | | A | B | 000011100 | | | Rc |
| **andc*x*** | 011111 | S | | | A | B | 000111100 | | | Rc |
| **andi.** | 011100 | S | | | A | UIMM | | | | |
| **andis.** | 011101 | S | | | A | UIMM | | | | |
| **b*x*** | 010010 | LI | | | | | | | AA | LK |
| **bc*x*** | 010000 | BO | | | BI | BD | | | AA | LK |
| **bcctr*x*** | 010011 | BO | | | BI | 0_0000 | 1000010000 | | | LK |
| **bclr*x*** | 010011 | BO | | | BI | 0_0000 | 0000010000 | | | LK |
| **cmp** | 011111 | crfD | 0 | L | A | B | 0000000000 | | | 0 |
| **cmpi** | 001011 | crfD | 0 | L | A | SIMM | | | | |
| **cmpl** | 011111 | crfD | 0 | L | A | B | 0000100000 | | | 0 |
| **cmpli** | 001010 | crfD | 0 | L | A | UIMM | | | | |
| **cntlzw*x*** | 011111 | S | | | A | 0_0000 | 0000011010 | | | Rc |
| **crand** | 010011 | crbD | | | crbA | crbB | 0100000001 | | | 0 |
| **crandc** | 010011 | crbD | | | crbA | crbB | 0010000001 | | | 0 |
| **creqv** | 010011 | crbD | | | crbA | crbB | 0100100001 | | | 0 |
| **crnand** | 010011 | crbD | | | crbA | crbB | 0011100001 | | | 0 |
| **crnor** | 010011 | crbD | | | crbA | crbB | 0000100001 | | | 0 |
| **cror** | 010011 | crbD | | | crbA | crbB | 0111000001 | | | 0 |
| **crorc** | 010011 | crbD | | | crbA | crbB | 0110100001 | | | 0 |
| **crxor** | 010011 | crbD | | | crbA | crbB | 0011000001 | | | 0 |
| **dcba** [1] | 011111 | 000_00 | | | A | B | 1011110110 | | | 0 |
| **dcbf** | 011111 | 000_00 | | | A | B | 0001010110 | | | 0 |
| **dcbi** [2] | 011111 | 000_00 | | | A | B | 0111010110 | | | 0 |
| **dcbst** | 011111 | 000_00 | | | A | B | 0000110110 | | | 0 |
| **dcbt** | 011111 | 000_00 | | | A | B | 0100010110 | | | 0 |
| **dcbtst** | 011111 | 000_00 | | | A | B | 0011110110 | | | 0 |

## Table A-3. Instructions by Mnemonic (Binary) (continued)

| Mnemonic | 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 |
|---|---|---|---|---|---|---|---|
| **dcbz** | 011111 | 000_00 | A | B | 1111110110 | | 0 |
| **divw***x* | 011111 | D | A | B | OE | 11110 1011 | Rc |
| **divw***x* | 011111 | D | A | B | OE | 11100 1011 | Rc |
| **eciwx** 1 | 011111 | D | A | B | 0100110110 | | 0 |
| **ecowx** 1 | 011111 | S | A | B | 0110110110 | | 0 |
| **eieio** | 011111 | 000_00 | 00_000 | 0_0000 | 1101010110 | | 0 |
| **eqv***x* | 011111 | S | A | B | 0100011100 | | Rc |
| **extsb***x* | 011111 | S | A | 0_0000 | 1110111010 | | Rc |
| **extsh***x* | 011111 | S | A | 0_0000 | 1110011010 | | Rc |
| **fabs***x* | 111111 | D | 00_000 | B | 0100001000 | | Rc |
| **fadd***x* | 111111 | D | A | B | 0000_0 | 1 0101 | Rc |
| **fadds***x* | 111011 | D | A | B | 0000_0 | 1 0101 | Rc |
| **fcmpo** | 111111 | crfD 00 | A | B | 0000100000 | | 0 |
| **fcmpu** | 111111 | crfD 00 | A | B | 0000000000 | | 0 |
| **fctiw***x* | 111111 | D | 00_000 | B | 0000001110 | | Rc |
| **fctiwz***x* | 111111 | D | 00_000 | B | 0000001111 | | Rc |
| **fdiv***x* | 111111 | D | A | B | 0000_0 | 1 0010 | Rc |
| **fdivs***x* | 111011 | D | A | B | 0000_0 | 1 0010 | Rc |
| **fmadd***x* | 111111 | D | A | B | C | 1 1101 | Rc |
| **fmadds***x* | 111011 | D | A | B | C | 1 1101 | Rc |
| **fmr***x* | 111111 | D | 00_000 | B | 0001001000 | | Rc |
| **fmsub***x* | 111111 | D | A | B | C | 1 1100 | Rc |
| **fmsubs***x* | 111011 | D | A | B | C | 1 1100 | Rc |
| **fmul***x* | 111111 | D | A | 0_0000 | C | 1 1001 | Rc |
| **fmuls***x* | 111011 | D | A | 0_0000 | C | 1 1001 | Rc |
| **fnabs***x* | 111111 | D | 00_000 | B | 0010001000 | | Rc |
| **fneg***x* | 111111 | D | 00_000 | B | 0000101000 | | Rc |
| **fnmadd***x* | 111111 | D | A | B | C | 1 1111 | Rc |
| **fnmadds***x* | 111011 | D | A | B | C | 1 1111 | Rc |
| **fnmsub***x* | 111111 | D | A | B | C | 1 1110 | Rc |
| **fnmsubs***x* | 111011 | D | A | B | C | 1 1110 | Rc |
| **fres***x* 1 | 111011 | D | 00_000 | B | 0000_0 | 1 1000 | Rc |
| **frsp***x* | 111111 | D | 00_000 | B | 0000001100 | | Rc |
| **frsqrte***x* 1 | 111111 | D | 00_000 | B | 0000_0 | 1 1010 | Rc |

**Table A-3. Instructions by Mnemonic (Binary) (continued)**

| Mnemonic | 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 |
|---|---|---|---|---|---|---|---|
| **fsel***x* [1] | 111111 | D | A | B | C | 1 0111 | Rc |
| **fsqrt***x* [1] | 111111 | D | 00_000 | B | 0000_0 | 1 0110 | Rc |
| **fsqrts***x* [1] | 111011 | D | 00_000 | B | 0000_0 | 1 0110 | Rc |
| **fsub***x* | 111111 | D | A | B | 0000_0 | 1 0100 | Rc |
| **fsubs**x | 111011 | D | A | B | 0000_0 | 1 0100 | Rc |
| **icbi** | 011111 | 000_00 | A | B | 1111010110 | | 0 |
| **isync** | 010011 | 000_00 | 00_000 | 0_0000 | 0010010110 | | 0 |
| **lbz** | 100010 | D | A | d | | | |
| **lbzu** | 100011 | D | A | d | | | |
| **lbzux** | 011111 | D | A | B | 0001110111 | | 0 |
| **lbzx** | 011111 | D | A | B | 0001010111 | | 0 |
| **lfd** | 110010 | D | A | d | | | |
| **lfdu** | 110011 | D | A | d | | | |
| **lfdux** | 011111 | D | A | B | 1001110111 | | 0 |
| **lfdx** | 011111 | D | A | B | 1001010111 | | 0 |
| **lfs** | 110000 | D | A | d | | | |
| **lfsu** | 110001 | D | A | d | | | |
| **lfsux** | 011111 | D | A | B | 1000110111 | | 0 |
| **lfsx** | 011111 | D | A | B | 1000010111 | | 0 |
| **lha** | 101010 | D | A | d | | | |
| **lhau** | 101011 | D | A | d | | | |
| **lhaux** | 011111 | D | A | B | 0101110111 | | 0 |
| **lhax** | 011111 | D | A | B | 0101010111 | | 0 |
| **lhbrx** | 011111 | D | A | B | 1100010110 | | 0 |
| **lhz** | 101000 | D | A | d | | | |
| **lhzu** | 101001 | D | A | d | | | |
| **lhzux** | 011111 | D | A | B | 0100110111 | | 0 |
| **lhzx** | 011111 | D | A | B | 0100010111 | | 0 |
| **lmw** [3] | 101110 | D | A | d | | | |
| **lswi** [3] | 011111 | D | A | NB | 1001010101 | | 0 |
| **lswx** [3] | 011111 | D | A | B | 1000010101 | | 0 |
| **lwarx** | 011111 | D | A | B | 0000010100 | | 0 |
| **lwbrx** | 011111 | D | A | B | 1000010110 | | 0 |
| **lwz** | 100000 | D | A | d | | | |

## Table A-3. Instructions by Mnemonic (Binary) (continued)

| Mnemonic | 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 |
|---|---|---|---|---|---|---|
| **lwzu** | 100001 | D | A | d | | |
| **lwzux** | 011111 | D | A | B | 0000110111 | 0 |
| **lwzx** | 011111 | D | A | B | 0000010111 | 0 |
| **mcrf** | 010011 | crfD 00 | crfS 00 | 0_0000 | 000000000 | 0 |
| **mcrfs** | 111111 | crfD 00 | crfS 00 | 0_0000 | 001000000 | 0 |
| **mcrxr** | 011111 | crfD 00 | 00_000 | 0_0000 | 1000000000 | 0 |
| **mfcr** | 011111 | D | 00_000 | 0_0000 | 0000010011 | 0 |
| **mffs***x* | 111111 | D | 00_000 | 0_0000 | 1001000111 | Rc |
| **mfmsr** [2] | 011111 | D | 00_000 | 0_0000 | 0001010011 | 0 |
| **mfspr** [4] | 011111 | D | spr | | 0101010011 | 0 |
| **mfsr**[2] | 011111 | D | 0 SR | 0_0000 | 1001010011 | 0 |
| **mfsrin**[2] | 011111 | D | 00_000 | B | 1010010011 | 0 |
| **mftb** | 011111 | D | tbr | | 0101110011 | 0 |
| **mfvscr**[3] | 000100 | **v**D | 00_000 | 0_0000 | 11000000100 | 0 |
| **mtcrf** | 011111 | S | 0 CRM | 0 | 0010010000 | 0 |
| **mtfsb0**x | 111111 | crbD | 00_000 | 0_0000 | 0001000110 | Rc |
| **mtfsb1**x | 111111 | crbD | 00_000 | 0_0000 | 0000100110 | Rc |
| **mtfsf**x | 111111 | 0 FM | 0 | B | 1011000111 | Rc |
| **mtfsfi**x | 111111 | crfD 00 | 00_000 | IMM 0 | 0010000110 | Rc |
| **mtmsr** [2] | 011111 | S | 00_000 | 0_0000 | 0010010010 | 0 |
| **mtspr** [4] | 011111 | S | spr | | 0111010011 | 0 |
| **mtsr** [2] | 011111 | S | 0 SR | 0_0000 | 0011010010 | 0 |
| **mtsrin** [2] | 011111 | S | 00_000 | B | 0011110010 | 0 |
| **mulhw***x* | 011111 | D | A | B | 0 001001011 | Rc |
| **mulhwu***x* | 011111 | D | A | B | 0 000001011 | Rc |
| **mulli** | 000111 | D | A | SIMM | | |
| **mullw***x* | 011111 | D | A | B | OE 011101011 | Rc |
| **nand***x* | 011111 | S | A | B | 0111011100 | Rc |
| **neg***x* | 011111 | D | A | 0_0000 | OE 001101000 | Rc |
| **nor***x* | 011111 | S | A | B | 0001111100 | Rc |
| **or***x* | 011111 | S | A | B | 0110111100 | Rc |
| **orc***x* | 011111 | S | A | B | 0110011100 | Rc |
| **ori** | 011000 | S | A | UIMM | | |
| **oris** | 011001 | S | A | UIMM | | |

**Table A-3. Instructions by Mnemonic (Binary) (continued)**

| Mnemonic | 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 |
|---|---|---|---|---|---|---|---|
| rfi[2] | 010011 | 000_00 | 00_000 | 0_0000 | 0000110010 | | 0 |
| rlwimi*x* | 010100 | S | A | SH | MB | ME | Rc |
| rlwinm*x* | 010101 | S | A | SH | MB | ME | Rc |
| rlwnm*x* | 010111 | S | A | B | MB | ME | Rc |
| sc | 010001 | 000_0000_0000_0000_0000_0000_00 | | | | 1 | 0 |
| slw*x* | 011111 | S | A | B | 0000011000 | | Rc |
| sraw*x* | 011111 | S | A | B | 1100011000 | | Rc |
| srawi*x* | 011111 | S | A | SH | 1100011000 | | Rc |
| srw*x* | 011111 | S | A | B | 1000011000 | | Rc |
| stb | 100110 | S | A | d | | | |
| stbu | 100111 | S | A | d | | | |
| stbux | 011111 | S | A | B | 0011110111 | | 0 |
| stbx | 011111 | S | A | B | 0011010111 | | 0 |
| stfd | 110110 | S | A | d | | | |
| stfdu | 110111 | S | A | d | | | |
| stfdux | 011111 | S | A | B | 1011110111 | | 0 |
| stfdx | 011111 | S | A | B | 1011010111 | | 0 |
| stfiwx[1] | 011111 | S | A | B | 1111010111 | | 0 |
| stfs | 110100 | S | A | d | | | |
| stfsu | 110101 | S | A | d | | | |
| stfsux | 011111 | S | A | B | 1010110111 | | 0 |
| stfsx | 011111 | S | A | B | 1010010111 | | 0 |
| sth | 101100 | S | A | d | | | |
| sthbrx | 011111 | S | A | B | 1110010110 | | 0 |
| sthu | 101101 | S | A | d | | | |
| sthux | 011111 | S | A | B | 110110111 | | 0 |
| sthx | 011111 | S | A | B | 110010111 | | 0 |
| stmw[5] | 101111 | S | A | d | | | |
| stswi[3] | 011111 | S | A | NB | 1011010101 | | 0 |
| stswx[3] | 011111 | S | A | B | 1010010101 | | 0 |
| stw | 100100 | S | A | d | | | |
| stwbrx | 011111 | S | A | B | 1010010110 | | 0 |
| stwcx. | 011111 | S | A | B | 10010110 | | 1 |
| stwu | 100101 | S | A | d | | | |

**Table A-3. Instructions by Mnemonic (Binary) (continued)**

| Mnemonic | 0–5 | 6–10 | 11–15 | 16–20 | 21 | 22–30 | 31 |
|---|---|---|---|---|---|---|---|
| **stwux** | 011111 | S | A | B | 10110111 | | 0 |
| **stwx** | 011111 | S | A | B | 10010111 | | 0 |
| **subf*x*** | 011111 | D | A | B | OE | 000101000 | Rc |
| **subfc*x*** | 011111 | D | A | B | OE | 000001000 | Rc |
| **subfe*x*** | 011111 | D | A | B | OE | 010001000 | Rc |
| **subfic** | 001000 | D | A | SIMM | | | |
| **subfme*x*** | 011111 | D | A | 0_0000 | OE | 011101000 | Rc |
| **subfze*x*** | 011111 | D | A | 0_0000 | OE | 011001000 | Rc |
| **sync** | 011111 | 000_00 | 00_000 | 0_0000 | 1001010110 | | 0 |
| **tlbia** [1] | 011111 | 000_00 | 00_000 | 0_0000 | 0101110010 | | 0 |
| **tlbie** [1,2] | 011111 | 000_00 | 00_000 | B | 0100110010 | | 0 |
| **tlbld** [1,2] | 011111 | 000_00 | 00_000 | B | 1111010010 | | 0 |
| **tlbli** [1,2] | 011111 | 000_00 | 00_000 | B | 1111110010 | | 0 |
| **tlbsync** [1,2] | 011111 | 000_00 | 00_000 | 0_0000 | 1000110110 | | 0 |
| **tw** | 011111 | TO | A | B | 0000000100 | | 0 |
| **twi** | 000011 | TO | A | SIMM | | | |
| **xor*x*** | 011111 | S | A | B | 0100111100 | | Rc |
| **xori** | 011010 | S | A | UIMM | | | |
| **xoris** | 011011 | S | A | UIMM | | | |

[1]Optional to the PowerPC architecture
[2]Supervisor-level instruction
[3]Load/store multiple/string instruction
[4]Supervisor- and user-level instruction

## A.4 Instructions Sorted by Opcode (Binary)

Table A-4 lists instructions by opcode, shown in binary.

**Table A-4. Instructions by Primary and Secondary Opcode (Bin)**

| Mnemonic | 0–5 | 6–8 | 9 | 10 | 11–15 | 16–31 |
|---|---|---|---|---|---|---|
| **twi** | 000011 | TO | | | A | SIMM |
| **mulli** | 000111 | D | | | A | SIMM |
| **subfic** | 001000 | D | | | A | SIMM |
| **cmpli** | 001010 | crfD | 0 | L | A | UIMM |
| **cmpi** | 001011 | crfD | 0 | L | A | SIMM |
| **addic** | 001100 | D | | | A | SIMM |

## Table A-4. Instructions by Primary and Secondary Opcode (Bin) (continued)

| Mnemonic | 0–5 | 6–10 | 11–15 | 16–20 | 21–25 | 26–30 | 31 |
|---|---|---|---|---|---|---|---|
| **addic.** | 001101 | D | A | SIMM | | | |
| **addi** | 001110 | D | A | SIMM | | | |
| **addis** | 001111 | D | A | SIMM | | | |
| **bc***x* | 010000 | BO | BI | BD | | AA | LK |
| **sc** | 010001 | 000_0000_0000_0000_0000_0000_00 | | | | 1 | 0 |
| **b***x* | 010010 | LI | | | | AA | LK |
| **mcrf** | 010011 | crfD 00 | crfS 00 | 0_0000 | 000000000 | | 0 |
| **bclr***x* | 010011 | BO | BI | 0_0000 | 0000010000 | | LK |
| **crnor** | 010011 | crbD | crbA | crbB | 0000100001 | | 0 |
| **rfi** [1] | 010011 | 000_00 | 00_000 | 0_0000 | 0000110010 | | 0 |
| **crandc** | 010011 | crbD | crbA | crbB | 0010000001 | | 0 |
| **isync** | 010011 | 000_00 | 00_000 | 0_0000 | 0010010110 | | 0 |
| **crxor** | 010011 | crbD | crbA | crbB | 0011000001 | | 0 |
| **crnand** | 010011 | crbD | crbA | crbB | 0011100001 | | 0 |
| **crand** | 010011 | crbD | crbA | crbB | 0100000001 | | 0 |
| **creqv** | 010011 | crbD | crbA | crbB | 0100100001 | | 0 |
| **crorc** | 010011 | crbD | crbA | crbB | 0110100001 | | 0 |
| **cror** | 010011 | crbD | crbA | crbB | 0111000001 | | 0 |
| **bcctr***x* | 010011 | BO | BI | 0_0000 | 1000010000 | | LK |
| **rlwimi***x* | 010100 | S | A | SH | MB | ME | Rc |
| **rlwinm***x* | 010101 | S | A | SH | MB | ME | Rc |
| **rlwnm***x* | 010111 | S | A | B | MB | ME | Rc |
| **ori** | 011000 | S | A | UIMM | | | |
| **oris** | 011001 | S | A | UIMM | | | |
| **xori** | 011010 | S | A | UIMM | | | |
| **xoris** | 011011 | S | A | UIMM | | | |
| **andi.** | 011100 | S | A | UIMM | | | |
| **andis.** | 011101 | S | A | UIMM | | | |
| **cmp** | 011111 | crfD 0 L | A | B | 0000000000 | | 0 |
| **tw** | 011111 | TO | A | B | 0000000100 | | 0 |
| **subfc***x* | 011111 | D | A | B | OE | 000001000 | Rc |
| **addc***x* | 011111 | D | A | B | OE | 000001010 | Rc |
| **mulhwu***x* | 011111 | D | A | B | 0 | 000001011 | Rc |
| **mfcr** | 011111 | D | 00_000 | 0_0000 | 0000010011 | | 0 |

## Table A-4. Instructions by Primary and Secondary Opcode (Bin) (continued)

| Mnemonic | 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 |
|---|---|---|---|---|---|---|
| **lwarx** | 011111 | D | A | B | 0000010100 | 0 |
| **lwzx** | 011111 | D | A | B | 0000010111 | 0 |
| **slw***x* | 011111 | S | A | B | 0000011000 | Rc |
| **cntlzw***x* | 011111 | S | A | 0_0000 | 0000011010 | Rc |
| **and***x* | 011111 | S | A | B | 000011100 | Rc |
| **cmpl** | 011111 | crfD 0 L | A | B | 0000100000 | 0 |
| **lvsr** [1] | 011111 | **v**D | A | B | 0000100110 | 0 |
| **lvehx** [1] | 011111 | **v**D | A | B | 0000100111 | 0 |
| **subf***x* | 011111 | D | A | B | OE 000101000 | Rc |
| **dcbst** | 011111 | 000_00 | A | B | 0000110110 | 0 |
| **lwzux** | 011111 | D | A | B | 0000110111 | 0 |
| **andc***x* | 011111 | S | A | B | 000111100 | Rc |
| **lvewx** [1] | 011111 | **v**D | A | B | 0001000111 | 0 |
| **mulhw***x* | 011111 | D | A | B | 0 001001011 | Rc |
| **mfmsr** [1] | 011111 | D | 00_000 | 0_0000 | 0001010011 | 0 |
| **dcbf** | 011111 | 000_00 | A | B | 0001010110 | 0 |
| **lbzx** | 011111 | D | A | B | 0001010111 | 0 |
| **neg***x* | 011111 | D | A | 0_0000 | OE 001101000 | Rc |
| **lbzux** | 011111 | D | A | B | 0001110111 | 0 |
| **nor***x* | 011111 | S | A | B | 0001111100 | Rc |
| **subfe***x* | 011111 | D | A | B | OE 010001000 | Rc |
| **adde***x* | 011111 | D | A | B | OE 010001010 | Rc |
| **mtcrf** | 011111 | S | 0 CRM 0 | | 0010010000 | 0 |
| **mtmsr** [1] | 011111 | S | 00_000 | 0_0000 | 0010010000 | 0 |
| **stwcx.** | 011111 | S | A | B | 10010110 | 1 |
| **stwx** | 011111 | S | A | B | 10010111 | 0 |
| **stwux** | 011111 | S | A | B | 10110111 | 0 |
| **subfze***x* | 011111 | D | A | 0_0000 | OE 011001000 | Rc |
| **addze***x* | 011111 | D | A | 0_0000 | OE 11001010 | Rc |
| **stbx** | 011111 | S | A | B | 0011010111 | 0 |
| **subfme***x* | 011111 | D | A | 0_0000 | OE 011101000 | Rc |
| **addme***x* | 011111 | D | A | 0_0000 | OE 11101010 | Rc |
| **mullw***x* | 011111 | D | A | B | OE 011101011 | Rc |
| **mtsrin** [1] | 011111 | S | 00_000 | B | 0011110010 | 0 |

## Table A-4. Instructions by Primary and Secondary Opcode (Bin) (continued)

| Mnemonic | 0–5 | 6–10 | 11–15 | 16–20 | 21–30 | 31 |
|---|---|---|---|---|---|---|
| **dcbtst** | 011111 | 000_00 | A | B | 0011110110 | 0 |
| **stbux** | 011111 | S | A | B | 0011110111 | 0 |
| **add***x* | 011111 | D | A | B | OE 100 001 010 | Rc |
| **dcbt** | 011111 | 000_00 | A | B | 0100010110 | 0 |
| **lhzx** | 011111 | D | A | B | 0100010111 | 0 |
| **eqv***x* | 011111 | S | A | B | 0100011100 | Rc |
| **tlbie** [1, 2] | 011111 | 000_00 | 00_000 | B | 0100110010 | 0 |
| **eciwx** [2] | 011111 | D | A | B | 0100110110 | 0 |
| **lhzux** | 011111 | D | A | B | 0100110111 | 0 |
| **xor***x* | 011111 | S | A | B | 0100111100 | Rc |
| **mfspr** [3] | 011111 | D | spr | | 0101010011 | 0 |
| **lhax** | 011111 | D | A | B | 0101010111 | 0 |
| **tlbia** [2] | 011111 | 000_00 | 00_000 | 0_0000 | 0101110010 | 0 |
| **mftb** | 011111 | D | tbr | | 0101110011 | 0 |
| **lhaux** | 011111 | D | A | B | 0101110111 | 0 |
| **sthx** | 011111 | S | A | B | 110010111 | 0 |
| **orc***x* | 011111 | S | A | B | 0110011100 | Rc |
| **ecowx** [2] | 011111 | S | A | B | 0110110110 | 0 |
| **sthux** | 011111 | S | A | B | 110110111 | 0 |
| **or***x* | 011111 | S | A | B | 0110111100 | Rc |
| **divwu***x* | 011111 | D | A | B | OE 1 1100 1011 | Rc |
| **mtspr** [3] | 011111 | S | spr | | 0111010011 | 0 |
| **dcbi** [1] | 011111 | 000_00 | A | B | 0111010110 | 0 |
| **nand***x* | 011111 | S | A | B | 0111011100 | Rc |
| **divw***x* | 011111 | D | A | B | OE 1 1110 1011 | Rc |
| **mcrxr** | 011111 | crfD 00 | 00_000 | 0_0000 | 1000000000 | 0 |
| **lswx** [4] | 011111 | D | A | B | 1000010101 | 0 |
| **lwbrx** | 011111 | D | A | B | 1000010110 | 0 |
| **lfsx** | 011111 | D | A | B | 1000010111 | 0 |
| **srw***x* | 011111 | S | A | B | 1000011000 | Rc |
| **tlbsync** [1, 2] | 011111 | 000_00 | 00_000 | 0_0000 | 1000110110 | 0 |
| **lfsux** | 011111 | D | A | B | 1000110111 | 0 |
| **mfsr** [1] | 011111 | D | 0 SR | 0_0000 | 1001010011 | 0 |
| **lswi** [4] | 011111 | D | A | NB | 1001010101 | 0 |

## Table A-4. Instructions by Primary and Secondary Opcode (Bin) (continued)

| Mnemonic | 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 |
|---|---|---|---|---|---|---|---|---|
| **sync** | 011111 | 000_00 | | | 00_000 | 0_0000 | 1001010110 | 0 |
| **lfdx** | 011111 | D | | | A | B | 1001010111 | 0 |
| **lfdux** | 011111 | D | | | A | B | 1001110111 | 0 |
| **mfsrin**[1] | 011111 | D | | | 00_000 | B | 1010010011 | 0 |
| **stswx**[4] | 011111 | S | | | A | B | 1010010101 | 0 |
| **stwbrx** | 011111 | S | | | A | B | 1010010110 | 0 |
| **stfsx** | 011111 | S | | | A | B | 1010010111 | 0 |
| **stfsux** | 011111 | S | | | A | B | 1010110111 | 0 |
| **stswi**[4] | 011111 | S | | | A | NB | 1011010101 | 0 |
| **stfdx** | 011111 | S | | | A | B | 1011010111 | 0 |
| **dcba**[2] | 011111 | 000_00 | | | A | B | 1011110110 | 0 |
| **stfdux** | 011111 | S | | | A | B | 1011110111 | 0 |
| **lhbrx** | 011111 | D | | | A | B | 1100010110 | 0 |
| **sraw***x* | 011111 | S | | | A | B | 1100011000 | Rc |
| **dss**[1] | 011111 | A | 00 | STRM | 00_000 | 0_0000 | 1100110110 | 0 |
| **dssall**[1] | 011111 | A | 00 | STRM | 00_000 | 0_0000 | 1100110110 | 0 |
| **srawi***x* | 011111 | S | | | A | SH | 1100011000 | Rc |
| **eieio** | 011111 | 000_00 | | | 00_000 | 0_0000 | 1101010110 | 0 |
| **sthbrx** | 011111 | S | | | A | B | 1110010110 | 0 |
| **extsh***x* | 011111 | S | | | A | 0_0000 | 1110011010 | Rc |
| **extsb***x* | 011111 | S | | | A | 0_0000 | 1110111010 | Rc |
| **tlbld**[1, 2] | 011111 | 000_00 | | | 00_000 | B | 1111010010 | 0 |
| **icbi** | 011111 | 000_00 | | | A | B | 1111010110 | 0 |
| **stfiwx**[2] | 011111 | S | | | A | B | 1111010111 | 0 |
| **tlbli**[1, 2] | 011111 | 000_00 | | | 00_000 | B | 1111110010 | 0 |
| **dcbz** | 011111 | 000_00 | | | A | B | 1111110110 | 0 |
| **lwz** | 100000 | D | | | A | d | | |
| **lwzu** | 100001 | D | | | A | d | | |
| **lbz** | 100010 | D | | | A | d | | |
| **lbzu** | 100011 | D | | | A | d | | |
| **stw** | 100100 | S | | | A | d | | |
| **stwu** | 100101 | S | | | A | d | | |
| **stb** | 100110 | S | | | A | d | | |
| **stbu** | 100111 | S | | | A | d | | |

## Table A-4. Instructions by Primary and Secondary Opcode (Bin) (continued)

| Mnemonic | 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 |
|---|---|---|---|---|---|---|---|
| **lhz** | 101000 | D | A | d | | | |
| **lhzu** | 101001 | D | A | d | | | |
| **lha** | 101010 | D | A | d | | | |
| **lhau** | 101011 | D | A | d | | | |
| **sth** | 101100 | S | A | d | | | |
| **sthu** | 101101 | S | A | d | | | |
| **lmw** [4] | 101110 | D | A | d | | | |
| **stmw** [4] | 101111 | S | A | d | | | |
| **lfs** | 110000 | D | A | d | | | |
| **lfsu** | 110001 | D | A | d | | | |
| **lfd** | 110010 | D | A | d | | | |
| **lfdu** | 110011 | D | A | d | | | |
| **stfs** | 110100 | S | A | d | | | |
| **stfsu** | 110101 | S | A | d | | | |
| **stfd** | 110110 | S | A | d | | | |
| **stfdu** | 110111 | S | A | d | | | |
| **fdivs***x* | 111011 | D | A | B | 0000_0 | 1 0010 | Rc |
| **fsubs***x* | 111011 | D | A | B | 0000_0 | 1 0100 | Rc |
| **fadds***x* | 111011 | D | A | B | 0000_0 | 1 0101 | Rc |
| **fsqrts***x* [2] | 111011 | D | 00_000 | B | 0000_0 | 1 0110 | Rc |
| **fres***x* [2] | 111011 | D | 00_000 | B | 0000_0 | 1 1000 | Rc |
| **fmuls***x* | 111011 | D | A | 0_0000 | C | 1 1001 | Rc |
| **fmsubs***x* | 111011 | D | A | B | C | 1 1100 | Rc |
| **fmadds***x* | 111011 | D | A | B | C | 1 1101 | Rc |
| **fnmsubs***x* | 111011 | D | A | B | C | 1 1110 | Rc |
| **fnmadds***x* | 111011 | D | A | B | C | 1 1111 | Rc |
| **fcmpu** | 111111 | crfD 00 | A | B | 0000000000 | | 0 |
| **frsp***x* | 111111 | D | 00_000 | B | 0000001100 | | Rc |
| **fctiw***x* | 111111 | D | 00_000 | B | 0000001110 | | Rc |
| **fctiwz***x* | 111111 | D | 00_000 | B | 0000001111 | | Rc |
| **fdiv***x* | 111111 | D | A | B | 0000_0 | 1 0010 | Rc |
| **fsub***x* | 111111 | D | A | B | 0000_0 | 1 0100 | Rc |
| **fadd***x* | 111111 | D | A | B | 0000_0 | 1 0101 | Rc |
| **fsqrt***x* [2] | 111111 | D | 00_000 | B | 0000_0 | 1 0110 | Rc |

**Table A-4. Instructions by Primary and Secondary Opcode (Bin) (continued)**

| Mnemonic | 0–5 | 6–10 | 11–15 | 16–20 | 21–25 | 26–30 | 31 |
|---|---|---|---|---|---|---|---|
| **fsel***x* [2] | 111111 | D | A | B | C | 1 0111 | Rc |
| **fmul***x* | 111111 | D | A | 0_0000 | C | 1 1001 | Rc |
| **frsqrte***x* [2] | 111111 | D | 00_000 | B | 0000_0 | 1 1010 | Rc |
| **fmsub***x* | 111111 | D | A | B | C | 1 1100 | Rc |
| **fmadd***x* | 111111 | D | A | B | C | 1 1101 | Rc |
| **fnmsub***x* | 111111 | D | A | B | C | 1 1110 | Rc |
| **fnmadd***x* | 111111 | D | A | B | C | 1 1111 | Rc |
| **fcmpo** | 111111 | crfD 00 | A | B | 0000100000 | | 0 |
| **mtfsb1***x* | 111111 | crbD | 00_000 | 0_0000 | 0000100110 | | Rc |
| **fneg***x* | 111111 | D | 00_000 | B | 0000101000 | | Rc |
| **mcrfs** | 111111 | crfD 00 | crfS 00 | 0_0000 | 001000000 | | 0 |
| **mtfsb0***x* | 111111 | crbD | 00_000 | 0_0000 | 0001000110 | | Rc |
| **fmr***x* | 111111 | D | 00_000 | B | 0001001000 | | Rc |
| **mtfsfi***x* | 111111 | crfD 00 | 00_000 | IMM 0 | 0010000110 | | Rc |
| **fnabs***x* | 111111 | D | 00_000 | B | 0010001000 | | Rc |
| **fabs***x* | 111111 | D | 00_000 | B | 0100001000 | | Rc |
| **mffs***x* | 111111 | D | 00_000 | 0_0000 | 1001000111 | | Rc |
| **mtfsf***x* | 111111 | 0 FM | 0 | B | 1011000111 | | Rc |

[1]Supervisor-level instruction
[2]Optional to the PowerPC architecture
[3]Supervisor- and user-level instruction
[4]Load/store string/multiple instruction

## A.5 Instructions Grouped by Functional Categories

Table A-5 through Table A-45 list instructions by function.

**Table A-5. Integer Arithmetic Instructions**

| Mnemonic | 0–5 | 6–10 | 11–15 | 16–20 | 21–30 | 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 | | |

## Table A-5. Integer Arithmetic Instructions

| Mnemonic | 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 |
|---|---|---|---|---|---|---|---|
| **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 |
| **divw***x* | 31 | D | A | B | OE | 491 | Rc |
| **divwu**x | 31 | D | A | B | OE | 459 | Rc |
| **mulhw***x* | 31 | D | A | B | 0 | 75 | Rc |
| **mulhwu***x* | 31 | D | A | B | 0 | 11 | 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** | 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 |

## Table A-6. Integer Compare Instructions

| Mnemonic | 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-7. Integer Logical Instructions

| Mnemonic | 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 | | |
| **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 |
| **nand***x* | 31 | S | A | B | 476 | Rc |

## Table A-7. Integer Logical Instructions (continued)

| Mnemonic | 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 |
|---|---|---|---|---|---|---|---|
| **nor***x* | 31 | S | A | B | 124 | | Rc |
| **or***x* | 31 | S | A | B | 444 | | Rc |
| **orc***x* | 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 | | | |

## Table A-8. Integer Rotate Instructions

| Mnemonic | 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* | 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 |

## Table A-9. Integer Shift Instruction

| Mnemonic | 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 |
|---|---|---|---|---|---|---|---|
| **slw***x* | 31 | S | A | B | 24 | | Rc |
| **sraw***x* | 31 | S | A | B | 792 | | Rc |
| **srawi***x* | 31 | S | A | SH | 824 | | Rc |
| **srw***x* | 31 | S | A | B | 536 | | Rc |

## Table A-10. Floating-Point Arithmetic Instructions

| Mnemonic | 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* | 63 | D | A | B | C | 23 | Rc |

**Table A-10. Floating-Point Arithmetic Instructions (continued)**

| Mnemonic | 0–5 | 6–10 | 11–15 | 16–20 | 21–25 | 26–30 | 31 |
|---|---|---|---|---|---|---|---|
| **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 |

[1]Optional to the PowerPC architecture

**Table A-11. Floating-Point Multiply-Add Instructions**

| Mnemonic | 0–5 | 6–10 | 11–15 | 16–20 | 21–25 | 26–30 | 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-12. Floating-Point Rounding and Conversion Instructions**

| Mnemonic | 0–5 | 6–10 | 11–15 | 16–20 | 21–30 | 31 |
|---|---|---|---|---|---|---|
| **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 |

**Table A-13. Floating-Point Compare Instructions**

| Mnemonic | 0–5 | 6–8 | 9–10 | 11–15 | 16–20 | 21–30 | 31 |
|---|---|---|---|---|---|---|---|
| fcmpo | 63 | crfD | 0 0 | A | B | 32 | 0 |
| fcmpu | 63 | crfD | 0 0 | A | B | 0 | 0 |

**Table A-14. Floating-Point Status and Control Register Instructions**

| Mnemonic | 0–5 | 6 | 7–8 | 9–10 | 11–13 | 14 | 15 | 16–19 | 20 | 21–30 | 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-15. Integer Load Instructions

| Mnemonic | 0–5 | 6–10 | 11–15 | 16–20 | 21–30 | 31 |
|---|---|---|---|---|---|---|
| lbz | 34 | D | A | d | | |
| lbzu | 35 | D | A | d | | |
| lbzux | 31 | D | A | B | 119 | 0 |
| lbzx | 31 | D | A | B | 87 | 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 |
| lwz | 32 | D | A | d | | |
| lwzu | 33 | D | A | d | | |
| lwzux | 31 | D | A | B | 55 | 0 |
| lwzx | 31 | D | A | B | 23 | 0 |

## Table A-16. Integer Store Instructions

| Mnemonic | 0–5 | 6–10 | 11–15 | 16–20 | 21–30 | 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 |
| 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 |

**Table A-17. Integer Load and Store with Byte Reverse Instructions**

| Mnemonic | 0–5 | 6–10 | 11–15 | 16–20 | 21–30 | 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-18. Integer Load and Store Multiple Instructions**

| Mnemonic | 0–5 | 6–10 | 11–15 | 16–31 |
|---|---|---|---|---|
| **lmw** | 46 | D | A | d |
| **stmw** | 47 | S | A | d |

**Table A-19. Integer Load and Store String Instructions**

| Mnemonic | 0–5 | 6–10 | 11–15 | 16–20 | 21–30 | 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 |

[1]Load/store string/multiple instruction

**Table A-20. Memory Synchronization Instructions**

| Mnemonic | 0–5 | 6–10 | 11–15 | 16–20 | 21–30 | 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 |
| **lwarx** | 31 | D | A | B | 20 | 0 |
| **stwcx.** | 31 | S | A | B | 150 | 1 |
| **sync** | 31 | 0 0 0 0 0 | 0 0 0 0 0 | 0 0 0 0 0 | 598 | 0 |

**Table A-21. Floating-Point Load Instructions**

| Mnemonic | 0–5 | 6–10 | 11–15 | 16–20 | 21–30 | 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 | | |

## Table A-21. Floating-Point Load Instructions

| Mnemonic | 0 … 5 | 6 … 10 | 11 … 15 | 16 … 20 | 21 … 30 | 31 |
|---|---|---|---|---|---|---|
| **lfsux** | 31 | D | A | B | 567 | 0 |
| **lfsx** | 31 | D | A | B | 535 | 0 |

## Table A-22. Floating-Point Store Instructions

| Mnemonic | 0 … 5 | 6 … 10 | 11 … 15 | 16 … 20 | 21 … 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 |

[1]Optional to the PowerPC architecture

## Table A-23. Floating-Point Move Instructions

| Mnemonic | 0 … 5 | 6 … 10 | 11 … 15 | 16 … 20 | 21 … 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-24. Branch Instructions

| Mnemonic | 0 … 5 | 6 … 10 | 11 … 15 | 16 … 20 | 21 … 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 0 0 | 528 | | LK |
| **bclr***x* | 19 | BO | BI | 0 0 0 0 0 | 16 | | LK |

## Table A-25. Condition Register Logical Instructions

| Mnemonic | 0 … 5 | 6 … 10 | 11 … 15 | 16 … 20 | 21 … 30 | 31 |
|---|---|---|---|---|---|---|
| **crand** | 19 | crbD | crbA | crbB | 257 | 0 |
| **crandc** | 19 | crbD | crbA | crbB | 129 | 0 |
| **creqv** | 19 | crbD | crbA | crbB | 289 | 0 |

## Table A-25. Condition Register Logical Instructions

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **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-26. System Linkage Instructions

| **Mnemonic** | 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 |
|---|---|---|---|---|---|---|---|
| **rfi** [1] | 19 | 0 0 0 0 0 | 0 0 0 0 0 | 0 0 0 0 0 | 50 | | 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 0 | | 1 | 0 |

[1]Supervisor-level instruction

## Table A-27. Trap Instructions

| **Mnemonic** | 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 |
|---|---|---|---|---|---|---|
| **tw** | 31 | TO | A | B | 4 | 0 |
| **twi** | 03 | TO | A | SIMM | | |

## Table A-28. Processor Control Instructions

| **Mnemonic** | 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 |
|---|---|---|---|---|---|---|---|---|---|
| **mcrxr** | 31 | crfS | 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 |
| **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 |
| **mtmsr** [1] | 31 | S | | 0 0 0 0 0 | | 0 0 0 0 0 | | 146 | 0 |
| **mtspr** [2] | 31 | D | | spr | | | | 467 | 0 |

[1]Supervisor-level instruction

[2]Supervisor- and user-level instruction

## Table A-29. Cache Management Instructions

| **Mnemonic** | 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 |
|---|---|---|---|---|---|---|
| **dcba** [1] | 31 | 0 0 0 0 0 | A | B | 758 | 0 |
| **dcbf** | 31 | 0 0 0 0 0 | A | B | 86 | 0 |
| **dcbi** [2] | 31 | 0 0 0 0 0 | A | B | 470 | 0 |

**Table A-29. Cache Management Instructions**

| | | | | | | |
|---|---|---|---|---|---|---|
| **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 |

1Optional to the PowerPC architecture

2Supervisor-level instruction

**Table A-30. Segment Register Manipulation Instructions**

| **Mnemonic** | 0–5 | 6–10 | 11 | 12–15 | 16–20 | 21–30 | 31 |
|---|---|---|---|---|---|---|---|
| **mfsr** 1 | 31 | D | 0 | SR | 0 0 0 0 0 | 595 | 0 |
| **mfsrin** 1 | 31 | D | 0 0 0 0 0 | | B | 659 | 0 |
| **mtsr**1 | 31 | S | 0 | SR | 0 0 0 0 0 | 210 | 0 |
| **mtsrin** 1 | 31 | S | 0 0 0 0 0 | | B | 242 | 0 |

1Supervisor-level instruction

**Table A-31. Lookaside Buffer Management Instructions**

| **Mnemonic** | 0–5 | 6–10 | 11–15 | 16–20 | 21–30 | 31 |
|---|---|---|---|---|---|---|
| **tlbia** 1 | 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 0 | 0 0 0 0 0 | B | 306 | 0 |
| **tlbld** 1,2 | 31 | 0 0 0 0 0 | 0 0 0 0 0 | B | 978 | 0 |
| **tlbli** 1,2 | 31 | 0 0 0 0 0 | 0 0 0 0 0 | B | 1010 | 0 |
| **tlbsync** 1,2 | 31 | 0 0 0 0 0 | 0 0 0 0 0 | 0 0 0 0 0 | 566 | 0 |

1Optional to the PowerPC architecture

2Supervisor-level instruction

**Table A-32. External Control Instructions**

| **Mnemonic** | 0–5 | 6–10 | 11–15 | 16–20 | 21–30 | 31 |
|---|---|---|---|---|---|---|
| **eciwx** 1 | 31 | D | A | B | 310 | 0 |
| **ecowx** 1 | 31 | S | A | B | 438 | 0 |

1Optional to the PowerPC architecture

## A.6 Instructions Sorted by Form

Table A-33 through Table A-43 list instructions grouped by form.

| | | | |
|---|---|---|---|
| OPCD | LI | AA | LK |

Figure A-1. I Form

Table A-33. I-Form Instruction Set

| Mnemonic | 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 |

| | | | | | |
|---|---|---|---|---|---|
| OPCD | BO | BI | BD | AA | LK |

Figure A-2. B Form

Table A-34. B-Form Instructions

| Mnemonic | 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 |

| | | | | | |
|---|---|---|---|---|---|
| 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 | 1 | 0 |

Figure A-3. SC Form

Table A-35. SC-Form Instruction

| Mnemonic | 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 | 1 | 0 |

| | | | | | |
|---|---|---|---|---|---|
| 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 |

Figure A-4. D Form

Table A-36. D-Form Instructions

| Mnemonic | 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 |

## Table A-36. D-Form Instructions (continued)

| Mnemonic | 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 |
|---|---|---|---|---|---|---|
| **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 |
| **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 |
| **twi** | 03 | TO | | | A | SIMM |

## Table A-36. D-Form Instructions (continued)

| Mnemonic | 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 |
|---|---|---|---|---|
| **xori** | 26 | S | A | UIMM |
| **xoris** | 27 | S | A | UIMM |

[1] Load/store string/multiple instruction

| | | | | | |
|---|---|---|---|---|---|
| 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 |
| OPCD | **v**D | **v**A | **v**B | XO | 0 |

**Figure A-5. X Form**

| 0–5 | 6–10 | 11–15 | 16–20 | 21–30 | 31 |
|---|---|---|---|---|---|
| OPCD | **v**S | **v**A | **v**B | XO | 0 |
| OPCD | T \| 0 0 \| STRM | A | B | XO | 0 |

**Figure A-5. X Form**

**Table A-37. X-Form Instructions**

| Mnemonic | 0 … 5 | 6 … 10 | 11 … 15 | 16 … 20 | 21 … 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 |
| **cntlzw***x* | 31 | S | A | 0 0 0 0 0 | 26 | Rc |
| **dcba** [1] | 31 | 0 0 0 0 0 | A | B | 758 | 0 |
| **dcbf** | 31 | 0 0 0 0 0 | A | B | 86 | 0 |
| **dcbi** [2] | 31 | 0 0 0 0 0 | A | B | 470 | 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 |
| **dst** | 31 | T \| 0 0 \| STRM | A | B | 342 | 0 |
| **dstt** [3] | 31 | 1 \| 0 0 \| STRM | A | B | 342 | 0 |
| **dstst** [3] | 31 | T \| 0 0 \| STRM | A | B | 374 | 0 |
| **dststt** [3] | 31 | 1 \| 0 0 \| STRM | A | B | 374 | 0 |
| **dss** [3] | 31 | A \| 0 0 \| STRM | 0 0 0 0 0 | 0 0 0 0 0 | 822 | 0 |
| **dssall** [3] | 31 | A \| 0 0 \| STRM | 0 0 0 0 0 | 0 0 0 0 0 | 822 | 0 |
| **eciwx** [1] | 31 | D | A | B | 310 | 0 |
| **ecowx** [1] | 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 |
| **fabs***x* | 63 | D | 0 0 0 0 0 | B | 264 | Rc |
| **fcmpo** | 63 | crfD \| 0 0 | A | B | 32 | 0 |
| **fcmpu** | 63 | crfD \| 0 0 | A | B | 0 | 0 |
| **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 |

**Table A-37. X-Form Instructions (continued)**

| Mnemonic | 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 |
|---|---|---|---|---|---|---|
| **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 |
| **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 |
| **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 |
| **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 |
| **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** [2] | 31 | D | 0 0 0 0 0 | 0 0 0 0 0 | 83 | 0 |
| **mfsr** [2] | 31 | D | 0 \| SR | 0 0 0 0 0 | 595 | 0 |
| **mfsrin** [2] | 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** [2] | 31 | S | 0 0 0 0 0 | 0 0 0 0 0 | 146 | 0 |
| **mtsr** [2] | 31 | S | 0 \| SR | 0 0 0 0 0 | 210 | 0 |
| **nand***x* | 31 | S | A | B | 476 | Rc |

## Table A-37. X-Form Instructions (continued)

| Mnemonic | 0–5 | 6–10 | 11–15 | 16–20 | 21–30 | 31 |
|---|---|---|---|---|---|---|
| **nor***x* | 31 | S | A | B | 124 | Rc |
| **or***x* | 31 | S | A | B | 444 | Rc |
| **orc***x* | 31 | S | A | B | 412 | Rc |
| **slw***x* | 31 | S | A | B | 24 | Rc |
| **sraw***x* | 31 | S | A | B | 792 | Rc |
| **srawi***x* | 31 | S | A | SH | 824 | Rc |
| **srw***x* | 31 | S | A | B | 536 | Rc |
| **stbux** | 31 | S | A | B | 247 | 0 |
| **stbx** | 31 | S | A | B | 215 | 0 |
| **stfdux** | 31 | S | A | B | 759 | 0 |
| **stfdx** | 31 | S | A | B | 727 | 0 |
| **stfiwx** [1] | 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 |
| **stvebx** [3] | 31 | **v**S | A | B | 135 | 0 |
| **stvehx** [3] | 31 | **v**S | A | B | 167 | 0 |
| **stvewx** [3] | 31 | **v**S | A | B | 199 | 0 |
| **stvx** [3] | 31 | **v**S | A | B | 231 | 0 |
| **stvxl** [3] | 31 | **v**S | A | B | 487 | 0 |
| **stwbrx** [3] | 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 0 0 | 0 0 0 0 0 | 0 0 0 0 0 | 598 | 0 |
| **tlbia** [1] | 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 0 | 0 0 0 0 0 | B | 306 | 0 |
| **tlbld** [1,2] | 31 | 0 0 0 0 0 | 0 0 0 0 0 | B | 978 | 0 |
| **tlbli** [1,2] | 31 | 0 0 0 0 0 | 0 0 0 0 0 | B | 1010 | 0 |
| **tlbsync** [2] | 31 | 0 0 0 0 0 | 0 0 0 0 0 | 0 0 0 0 0 | 566 | 0 |

## Table A-37. X-Form Instructions (continued)

| Mnemonic | 0–5 | 6–10 | 11–15 | 16–20 | 21–30 | 31 |
|---|---|---|---|---|---|---|
| **tw** | 31 | TO | A | B | 4 | 0 |
| **xor***x* | 31 | S | A | B | 316 | Rc |

[1]Optional to the PowerPC architecture
[2]Supervisor-level instruction
[3]Altivec technology-specific instruction
[4]Load/store string/multiple instruction

| 0–5 | 6–10 | | 11–15 | | 16–20 | 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 |

Figure A-6. XL Form

## Table A-38. XL Form Instructions

| Mnemonic | 0–5 | 6–10 | | 11–15 | | 16–20 | 21–30 | 31 |
|---|---|---|---|---|---|---|---|---|
| **bcctr***x* | 19 | BO | | BI | | 0 0 0 0 0 | 528 | LK |
| **bclr***x* | 19 | BO | | BI | | 0 0 0 0 0 | 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 |
| **rfi** [1] | 19 | 0 0 0 0 0 | | 0 0 0 0 0 | | 0 0 0 0 0 | 50 | 0 |

[1]Supervisor-level instruction

| 0–5 | 6–10 | 11 | 12–19 | 20 | 21–30 | 31 |
|---|---|---|---|---|---|---|
| OPCD | D | spr | | | XO | 0 |
| OPCD | D | 0 | CRM | 0 | XO | 0 |
| OPCD | S | spr | | | XO | 0 |
| OPCD | D | tbr | | | XO | 0 |

Figure A-7. XFX Form

## Table A-39. XFX-Form Instructions

| Mnemonic | 0–5 | 6–10 | 11 | 12–19 | 20 | 21–30 | 31 |
|---|---|---|---|---|---|---|---|
| **mfspr** [1] | 31 | D | spr | | | 339 | 0 |
| **mftb** | 31 | D | tbr | | | 371 | 0 |
| **mtcrf** | 31 | S | 0 | CRM | 0 | 144 | 0 |
| **mtspr** [1] | 31 | D | spr | | | 467 | 0 |

[1]Supervisor- and user-level instruction

| | | | | | | |
|---|---|---|---|---|---|---|
| OPCD | 0 | FM | 0 | B | XO | Rc |

**Figure A-8. XFL Form**

## Table A-40. XFL-Form Instructions

| Mnemonic | 0–5 | 6 | 7–14 | 15 | 16–20 | 21–30 | 31 |
|---|---|---|---|---|---|---|---|
| **mtfsf***x* | 63 | 0 | FM | 0 | B | 711 | Rc |

| | | | | | | |
|---|---|---|---|---|---|---|
| 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 |

**Figure A-9. XO Form**

## Table A-41. XO-Form Instructions

| Mnemonic | 0–5 | 6–10 | 11–15 | 16–20 | 21 | 22–30 | 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 |
| **divw***x* | 31 | D | A | B | OE | 491 | Rc |
| **divwu***x* | 31 | D | A | B | OE | 459 | Rc |
| **mulhw***x* | 31 | D | A | B | 0 | 75 | Rc |
| **mulhwu***x* | 31 | D | A | B | 0 | 11 | 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 |

## Table A-41. XO-Form Instructions (continued)

| Mnemonic | 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 |
|---|---|---|---|---|---|---|---|
| **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 |

| | | | | | | |
|---|---|---|---|---|---|---|
| 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 |

**Figure A-10. A Form**

## Table A-42. A-Form Instructions

| Mnemonic | 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 |
| **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 |
| **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 |

[1]Optional to the PowerPC architecture

| | | | | | | |
|---|---|---|---|---|---|---|
| OPCD | S | A | SH | MB | ME | Rc |
| OPCD | S | A | B | MB | ME | Rc |

**Figure A-11. M Form**

**Table A-43. M-Form Instructions**

| Mnemonic | 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 |

## A.7 Instruction Set Legend

Table A-45 provides general information on the instruction set (such as architectural level, privilege level, and form).

**Table A-44. PowerPC Instruction Set Legend**

| | UISA | VEA | OEA | Supervisor Level | Optional | Form |
|---|---|---|---|---|---|---|
| **add***x* | √ | | | | | XO |
| **addc***x* | √ | | | | | XO |
| **adde***x* | √ | | | | | XO |
| **addi** | √ | | | | | D |
| **addic** | √ | | | | | D |
| **addic.** | √ | | | | | D |
| **addis** | √ | | | | | D |
| **addme***x* | √ | | | | | XO |
| **addze***x* | √ | | | | | XO |
| **and***x* | √ | | | | | X |
| **andc***x* | √ | | | | | X |
| **andi.** | √ | | | | | D |
| **andis.** | √ | | | | | D |
| **b***x* | √ | | | | | I |
| **bc***x* | √ | | | | | B |
| **bcctr***x* | √ | | | | | XL |
| **bclr***x* | √ | | | | | XL |
| **cmp** | √ | | | | | X |
| **cmpi** | √ | | | | | D |
| **cmpl** | √ | | | | | X |

**Table A-44. PowerPC Instruction Set Legend (continued)**

| | UISA | VEA | OEA | Supervisor Level | Optional | Form |
|---|---|---|---|---|---|---|
| **cmpli** | √ | | | | | D |
| **cntlzw***x* | √ | | | | | X |
| **crand** | √ | | | | | XL |
| **crandc** | √ | | | | | XL |
| **creqv** | √ | | | | | XL |
| **crnand** | √ | | | | | XL |
| **crnor** | √ | | | | | XL |
| **cror** | √ | | | | | XL |
| **crorc** | √ | | | | | XL |
| **crxor** | √ | | | | | XL |
| **dcba** | | √ | | | √ | X |
| **dcbf** | | √ | | | | X |
| **dcbi** | | | √ | √ | | X |
| **dcbst** | | √ | | | | X |
| **dcbt** | | √ | | | | X |
| **dcbts**t | | √ | | | | X |
| **dcbz** | | √ | | | | X |
| **divw***x* | √ | | | | | XO |
| **divwu***x* | √ | | | | | XO |
| **eciwx** | | √ | | | √ | X |
| **ecowx** | | √ | | | √ | X |
| **eieio** | | √ | | | | X |
| **eqv***x* | √ | | | | | X |
| **extsb***x* | √ | | | | | X |
| **extsh***x* | √ | | | | | X |
| **fabs***x* | √ | | | | | X |
| **fadd***x* | √ | | | | | A |
| **fadds***x* | √ | | | | | A |
| **fcmpo** | √ | | | | | X |
| **fcmpu** | √ | | | | | X |
| **fctiw***x* | √ | | | | | X |
| **fctiwz***x* | √ | | | | | X |
| **fdiv***x* | √ | | | | | A |
| **fdivs***x* | √ | | | | | A |

**Table A-44. PowerPC Instruction Set Legend (continued)**

| | UISA | VEA | OEA | Supervisor Level | Optional | Form |
|---|---|---|---|---|---|---|
| **fmadd***x* | √ | | | | | A |
| **fmadds***x* | √ | | | | | A |
| **fmr***x* | √ | | | | | X |
| **fmsub***x* | √ | | | | | A |
| **fmsubs***x* | √ | | | | | A |
| **fmul***x* | √ | | | | | A |
| **fmuls***x* | √ | | | | | A |
| **fnabs***x* | √ | | | | | X |
| **fneg***x* | √ | | | | | X |
| **fnmadd***x* | √ | | | | | A |
| **fnmadds***x* | √ | | | | | A |
| **fnmsub***x* | √ | | | | | A |
| **fnmsubs***x* | √ | | | | | A |
| **fres***x* | √ | | | | √ | A |
| **frsp***x* | √ | | | | | X |
| **frsqrte***x* | √ | | | | √ | A |
| **fsel***x* | √ | | | | √ | A |
| **fsqrt***x* | √ | | | | √ | A |
| **fsqrts***x* | √ | | | | √ | A |
| **fsub***x* | √ | | | | | A |
| **fsubs***x* | √ | | | | | A |
| **icbi** | | √ | | | | X |
| **isync** | | √ | | | | XL |
| **lbz** | √ | | | | | D |
| **lbzu** | √ | | | | | D |
| **lbzux** | √ | | | | | X |
| **lbzx** | √ | | | | | X |
| **lfd** | √ | | | | | D |
| **lfdu** | √ | | | | | D |
| **lfdux** | √ | | | | | X |
| **lfdx** | √ | | | | | X |
| **lfs** | √ | | | | | D |
| **lfsu** | √ | | | | | D |
| **lfsux** | √ | | | | | X |

**Table A-44. PowerPC Instruction Set Legend (continued)**

| | UISA | VEA | OEA | Supervisor Level | Optional | Form |
|---|---|---|---|---|---|---|
| **lfsx** | √ | | | | | X |
| **lha** | √ | | | | | D |
| **lhau** | √ | | | | | D |
| **lhaux** | √ | | | | | X |
| **lhax** | √ | | | | | X |
| **lhbrx** | √ | | | | | X |
| **lhz** | √ | | | | | D |
| **lhzu** | √ | | | | | D |
| **lhzux** | √ | | | | | X |
| **lhzx** | √ | | | | | X |
| **lmw** [1] | √ | | | | | D |
| **lswi** [1] | √ | | | | | X |
| **lswx** [1] | √ | | | | | X |
| **lwarx** | √ | | | | | X |
| **lwbrx** | √ | | | | | X |
| **lwz** | √ | | | | | D |
| **lwzu** | √ | | | | | D |
| **lwzux** | √ | | | | | X |
| **lwzx** | √ | | | | | X |
| **mcrf** | √ | | | | | XL |
| **mcrfs** | √ | | | | | X |
| **mcrx**r | √ | | | | | X |
| **mfcr** | √ | | | | | X |
| **mffs** | √ | | | | | X |
| **mfmsr** | | | √ | √ | | X |
| **mfspr**[1] | √ | | √ | √ | | XFX |
| **mfsr** | | | √ | √ | | X |
| **mfsrin** | | | √ | √ | | X |
| **mftb** | | √ | | | | XFX |
| **mtcrf** | √ | | | | | XFX |
| **mtfsb0***x* | √ | | | | | X |
| **mtfsb1***x* | √ | | | | | X |
| **mtfs**f*x* | √ | | | | | XFL |
| **mtfsfi***x* | √ | | | | | X |

**Table A-44. PowerPC Instruction Set Legend (continued)**

| | UISA | VEA | OEA | Supervisor Level | Optional | Form |
|---|---|---|---|---|---|---|
| **mtmsr** | | | √ | √ | | X |
| **mtspr** [2] | √ | | √ | √ | | XFX |
| **mtsr** | | | √ | √ | | X |
| **mtsrin** | | | √ | √ | | X |
| **mulhw***x* | √ | | | | | XO |
| **mulhwu***x* | √ | | | | | XO |
| **mulli** | √ | | | | | D |
| **mullw***x* | √ | | | | | XO |
| **nand***x* | √ | | | | | X |
| **neg***x* | √ | | | | | XO |
| **nor***x* | √ | | | | | X |
| **or***x* | √ | | | | | X |
| **orc***x* | √ | | | | | X |
| **ori** | √ | | | | | D |
| **oris** | √ | | | | | D |
| **rfi** | | | √ | √ | | XL |
| **rlwimi***x* | √ | | | | | M |
| **rlwinm***x* | √ | | | | | M |
| **rlwnm***x* | √ | | | | | M |
| **sc** | √ | | √ | | | SC |
| **slw***x* | √ | | | | | X |
| **sraw***x* | √ | | | | | X |
| **srawi***x* | √ | | | | | X |
| **srw***x* | √ | | | | | X |
| **stb** | √ | | | | | D |
| **stbu** | √ | | | | | D |
| **stbux** | √ | | | | | X |
| **stbx** | √ | | | | | X |
| **stfd** | √ | | | | | D |
| **stfdu** | √ | | | | | D |
| **stfdux** | √ | | | | | X |
| **stfdx** | √ | | | | | X |
| **stfiwx** | √ | | | | | X |
| **stfs** | √ | | | | | D |

**Table A-44. PowerPC Instruction Set Legend (continued)**

| | UISA | VEA | OEA | Supervisor Level | Optional | Form |
|---|---|---|---|---|---|---|
| **stfsu** | √ | | | | | D |
| **stfsux** | √ | | | | | X |
| **stfsx** | √ | | | | | X |
| **sth** | √ | | | | | D |
| **sthbrx** | √ | | | | | X |
| **sthu** | √ | | | | | D |
| **sthux** | √ | | | | | X |
| **sthx** | √ | | | | | X |
| **stmw** 1 | √ | | | | | D |
| **stswi** 1 | √ | | | | | X |
| **stswx** 1 | √ | | | | | X |
| **stw** | √ | | | | | D |
| **stwbrx** | √ | | | | | X |
| **stwcx.** | √ | | | | | X |
| **stwu** | √ | | | | | D |
| **stwux** | √ | | | | | X |
| **stwx** | √ | | | | | X |
| **subf***x* | √ | | | | | XO |
| **subfc***x* | √ | | | | | XO |
| **subfe***x* | √ | | | | | XO |
| **subfic** | √ | | | | | D |
| **subfme***x* | √ | | | | | XO |
| **subfze***x* | √ | | | | | XO |
| **sync** | √ | | | | | X |
| **tlbia***x* | | | √ | √ | √ | X |
| **tlbie***x* | | | √ | √ | √ | X |
| **tlbsync** | | | √ | √ | √ | X |
| **tw** | √ | | | | | X |
| **tw**i | √ | | | | | D |
| **xor***x* | √ | | | | | X |

**Table A-44. PowerPC Instruction Set Legend (continued)**

| | UISA | VEA | OEA | Supervisor Level | Optional | Form |
|---|---|---|---|---|---|---|
| **xori** | √ | | | | | D |
| **xoris** | √ | | | | | D |

1 Load/store string or multiple instruction
2 Supervisor- and user-level instruction

**Table A-45. MPC7451 Instruction Set Legend**

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **dss** [1] | | √ | | | | | | √ | VX |
| **dssall** [1] | | √ | | | | | | √ | VX |
| **dst** [1] | √ | | | | | | | √ | VX |
| **dstst** [1] | | √ | | | | | | √ | VX |
| **dststt** [1] | | √ | | | | | | √ | VX |
| **dstt** [1] | | √ | | | | | | √ | VX |
| **lvebx** [1] | √ | | | | | | | √ | X |
| **lvehx** [1] | √ | | | | | | | √ | X |
| **lvewx** [1] | √ | | | | | | | √ | X |
| **lvsl** [1] | √ | | | | | | | √ | X |
| **lvsr** [1] | √ | | | | | | | √ | X |
| **lvx** [1] | √ | | | | | | | √ | X |
| **lvxl** [1] | √ | | | | | | | √ | X |
| **mfvscr** [1] | √ | | | | | | | √ | VX |
| **mtvscr** [1] | √ | | | | | | | √ | VX |
| **stvebx** [1] | √ | | | | | | | √ | X |
| **stvehx** [1] | √ | | | | | | | √ | X |
| **stvewx** [1] | √ | | | | | | | √ | X |
| **stvx** [1] | √ | | | | | | | √ | X |
| **stvxl** [1] | √ | | | | | | | √ | X |
| **vaddcuw** [1] | √ | | | | | | | √ | VX |
| **vaddfp** [1] | √ | | | | | | | √ | VX |
| **vaddsbs** [1] | √ | | | | | | | √ | VX |
| **vaddshs** [1] | √ | | | | | | | √ | VX |
| **vaddsws** [1] | √ | | | | | | | √ | VX |
| **vaddubm** [1] | √ | | | | | | | √ | VX |
| **vaddubs** [1] | √ | | | | | | | √ | VX |
| **vadduhm** [1] | √ | | | | | | | √ | VX |

## Table A-45. MPC7451 Instruction Set Legend (continued)

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **vadduhs** [1] | √ | | | | | | | √ | VX |
| **vadduwm** [1] | √ | | | | | | | √ | VX |
| **vadduws** [1] | √ | | | | | | | √ | VX |
| **vand** [1] | √ | | | | | | | √ | VX |
| **vandc** [1] | √ | | | | | | | √ | VX |
| **vavgsb** [1] | √ | | | | | | | √ | VX |
| **vavgsh** [1] | √ | | | | | | | √ | VX |
| **vavgsw** [1] | √ | | | | | | | √ | VX |
| **vavgub** [1] | √ | | | | | | | √ | VX |
| **vavguh** [1] | √ | | | | | | | √ | VX |
| **vavguw** [1] | √ | | | | | | | √ | VX |
| **vcfsx** [1] | √ | | | | | | | √ | VX |
| **vcfux** [1] | √ | | | | | | | √ | VX |
| **vcmpbfp***x* [1] | √ | | | | | | | √ | VXR |
| **vcmpeqfp***x* [1] | √ | | | | | | | √ | VXR |
| **vcmpequb***x* [1] | √ | | | | | | | √ | VXR |
| **vcmpequh***x* [1] | √ | | | | | | | √ | VXR |
| **vcmpequw***x* [1] | √ | | | | | | | √ | VXR |
| **vcmpgefp***x* [1] | √ | | | | | | | √ | VXR |
| **vcmpgtfp***x* [1] | √ | | | | | | | √ | VXR |
| **vcmpgtsb***x* [1] | √ | | | | | | | √ | VXR |
| **vcmpgtsh***x* [1] | √ | | | | | | | √ | VXR |
| **vcmpgtsw***x* [1] | √ | | | | | | | √ | VXR |
| **vcmpgtub***x* [1] | √ | | | | | | | √ | VXR |
| **vcmpgtuh***x* [1] | √ | | | | | | | √ | VXR |
| **vcmpgtuw***x* [1] | √ | | | | | | | √ | VXR |
| **vctsxs** [1] | √ | | | | | | | √ | VX |
| **vctuxs** [1] | √ | | | | | | | √ | VX |
| **vexptefp** [1] | √ | | | | | | | √ | VX |
| **vlogefp** [1] | √ | | | | | | | √ | VX |
| **vmaddfp** [1] | √ | | | | | | | √ | VA |
| **vmaxfp** [1] | √ | | | | | | | √ | VX |
| **vmaxsb** [1] | √ | | | | | | | √ | VX |
| **vmaxsh** [1] | √ | | | | | | | √ | VX |
| **vmaxsw** [1] | √ | | | | | | | √ | VX |

**Table A-45. MPC7451 Instruction Set Legend (continued)**

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **vmaxub** [1] | √ | | | | | | | √ | VX |
| **vmaxuh** [1] | √ | | | | | | | √ | VX |
| **vmaxuw** [1] | √ | | | | | | | √ | VX |
| **vmhaddshs** [1] | √ | | | | | | | √ | VA |
| **vmhraddshs** [1] | √ | | | | | | | √ | VA |
| **vminfp** [1] | √ | | | | | | | √ | VX |
| **vminsb** [1] | √ | | | | | | | √ | VX |
| **vminsh** [1] | √ | | | | | | | √ | VX |
| **vminsw** [1] | √ | | | | | | | √ | VX |
| **vminub** [1] | √ | | | | | | | √ | VX |
| **vminuh** [1] | √ | | | | | | | √ | VX |
| **vminuw** [1] | √ | | | | | | | √ | VX |
| **vmladduhm** [1] | √ | | | | | | | √ | VA |
| **vmrghb** [1] | √ | | | | | | | √ | VX |
| **vmrghh** [1] | √ | | | | | | | √ | VX |
| **vmrghw** [1] | √ | | | | | | | √ | VX |
| **vmrglb** [1] | √ | | | | | | | √ | VX |
| **vmrglh** [1] | √ | | | | | | | √ | VX |
| **vmrglw** [1] | √ | | | | | | | √ | VX |
| **vmsummbm** [1] | √ | | | | | | | √ | VA |
| **vmsumshm** [1] | √ | | | | | | | √ | VA |
| **vmsumshs** [1] | √ | | | | | | | √ | VA |
| **vmsumubm** [1] | √ | | | | | | | √ | VA |
| **vmsumuhm** [1] | √ | | | | | | | √ | VA |
| **vmsumuhs** [1] | √ | | | | | | | √ | VA |
| **vmulesb** [1] | √ | | | | | | | √ | VX |
| **vmulesh** [1] | √ | | | | | | | √ | VX |
| **vmuleub** [1] | √ | | | | | | | √ | VX |
| **vmuleuh** [1] | √ | | | | | | | √ | VX |
| **vmulosb** [1] | √ | | | | | | | √ | VX |
| **vmulosh** [1] | √ | | | | | | | √ | VX |
| **vmuloub** [1] | √ | | | | | | | √ | VX |
| **vmulouh** [1] | √ | | | | | | | √ | VX |
| **vnmsubfp** [1] | √ | | | | | | | √ | VA |
| **vnor** [1] | √ | | | | | | | √ | VX |

Table A-45. MPC7451 Instruction Set Legend (continued)

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **vor** [1] | √ | | | | | | | √ | VX |
| **vperm** [1] | √ | | | | | | | √ | VA |
| **vpkpx** [1] | √ | | | | | | | √ | VX |
| **vpkshss** [1] | √ | | | | | | | √ | VX |
| **vpkshus** [1] | √ | | | | | | | √ | VX |
| **vpkswss** [1] | √ | | | | | | | √ | VX |
| **vpkswus** [1] | √ | | | | | | | √ | VX |
| **vpkuhum** [1] | √ | | | | | | | √ | VX |
| **vpkuhus** [1] | √ | | | | | | | √ | VX |
| **vpkuwum** [1] | √ | | | | | | | √ | VX |
| **vpkuwus** [1] | √ | | | | | | | √ | VX |
| **vrefp** [1] | √ | | | | | | | √ | VX |
| **vrfim** [1] | √ | | | | | | | √ | VX |
| **vrfin** [1] | √ | | | | | | | √ | VX |
| **vrfip** [1] | √ | | | | | | | √ | VX |
| **vrfiz** [1] | √ | | | | | | | √ | VX |
| **vrlb** [1] | √ | | | | | | | √ | VX |
| **vrlh** [1] | √ | | | | | | | √ | VX |
| **vrlw** [1] | √ | | | | | | | √ | VX |
| **vrsqrtefp** [1] | √ | | | | | | | √ | VX |
| **vsel** [1] | √ | | | | | | | √ | VA |
| **vsl** [1] | √ | | | | | | | √ | VX |
| **vslb** [1] | √ | | | | | | | √ | VX |
| **vsldoi** [1] | √ | | | | | | | √ | VA |
| **vslh** [1] | √ | | | | | | | √ | VX |
| **vslo** [1] | √ | | | | | | | √ | VX |
| **vslw** [1] | √ | | | | | | | √ | VX |
| **vspltb** [1] | √ | | | | | | | √ | VX |
| **vsplth** [1] | √ | | | | | | | √ | VX |
| **vspltisb** [1] | √ | | | | | | | √ | VX |
| **vspltish** [1] | √ | | | | | | | √ | VX |
| **vspltisw** [1] | √ | | | | | | | √ | VX |
| **vspltw** [1] | √ | | | | | | | √ | VX |
| **vsr** [1] | √ | | | | | | | √ | VX |
| **vsrab**[1] | √ | | | | | | | √ | VX |

## Table A-45. MPC7451 Instruction Set Legend (continued)

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **vsrah** [1] | √ | | | | | | | √ | VX |
| **vsraw** [1] | √ | | | | | | | √ | VX |
| **vsrb** [1] | √ | | | | | | | √ | VX |
| **vsrh** [1] | √ | | | | | | | √ | VX |
| **vsro** [1] | √ | | | | | | | √ | VX |
| **vsrw** [1] | √ | | | | | | | √ | VX |
| **vsubcuw** [1] | √ | | | | | | | √ | VX |
| **vsubfp** [1] | √ | | | | | | | √ | VX |
| **vsubsbs** [1] | √ | | | | | | | √ | VX |
| **vsubshs** [1] | √ | | | | | | | √ | VX |
| **vsubsws** [1] | √ | | | | | | | √ | VX |
| **vsububm** [1] | √ | | | | | | | √ | VX |
| **vsubuhm** [1] | √ | | | | | | | √ | VX |
| **vsububs** [1] | √ | | | | | | | √ | VX |
| **vsubuhs** [1] | √ | | | | | | | √ | VX |
| **vsubuwm** [1] | √ | | | | | | | √ | VX |
| **vsubuws** [1] | √ | | | | | | | √ | VX |
| **vsumsws** [1] | √ | | | | | | | √ | VX |
| **vsum2sws** [1] | √ | | | | | | | √ | VX |
| **vsum4sbs** [1] | √ | | | | | | | √ | VX |
| **vsum4shs** [1] | √ | | | | | | | √ | VX |
| **vsum4ubs** [1] | √ | | | | | | | √ | VX |
| **vupkhp**x [1] | √ | | | | | | | √ | VX |
| **vupkhsb** [1] | √ | | | | | | | √ | VX |
| **vupkhsh** [1] | | | | | | | | √ | VX |
| **vupklpx** [1] | | | | | | | | √ | VX |
| **vupklsb** [1] | | | | | | | | √ | VX |
| **vupklsh** [1] | √ | | | | | | | √ | VX |
| **vxor** [1] | √ | | | | | | | √ | VX |

[1] AltiVec technology-specific instruction

# Appendix B
# POWER Architecture Cross-Reference

This appendix identifies the incompatibilities that must be managed in migration from the POWER architecture to PowerPC architecture. Some of the incompatibilities can, at least in principle, be detected by the processor, which traps and lets software simulate the POWER operation. Others cannot be detected by the processor.

In general, the incompatibilities identified here are those that affect a POWER application program. Incompatibilities for instructions that can be used only by POWER system programs are not discussed. Note that this appendix describes incompatibilities with respect to the PowerPC architecture in general.

## B.1 New Instructions, Formerly Supervisor-Level Instructions

Instructions new to PowerPC typically use opcode values (including extended opcode) that are illegal in the POWER architecture. A few instructions that are supervisor-level in the POWER architecture (for example, **dclz**, called **dcbz** in the PowerPC architecture) have been made user-level in the PowerPC architecture. Any POWER program that executes one of these now-valid, or now-user-level, instructions expecting to cause the system illegal instruction error handler (program exception) or the system supervisor-level instruction error handler to be invoked, not execute correctly on processors that implement the PowerPC architecture. (Note that, in the architecture specification, user- and supervisor-level are referred to as problem and privileged state, respectively, and exceptions are referred to as interrupts.)

## B.2 New Supervisor-Level Instructions

The following instructions are user-level in the POWER architecture but are supervisor-level in processors that implement the PowerPC architecture.

- **mfmsr**
- **mfsr**

## B.3 Reserved Bits in Instructions

These are shown as zeros and the bit field is shaded in the instruction opcode definitions. In the POWER architecture such bits are ignored by the processor. In the PowerPC architecture, they must be zero or the instruction form is invalid. In several cases, the PowerPC architecture assumes that such bits in POWER instructions are indeed zero. The cases include the following:

- **cmpi**, **cmp**, **cmpli**, and **cmpl** assume that bit 10 in the POWER instructions is 0.
- **mtspr** and **mfspr** assume that bits 16–20 in the POWER instructions are 0.

## B.4 Reserved Bits in Registers

The POWER architecture defines these bits to be zero when read, and either zero or one when written to. In the PowerPC architecture it is implementation-dependent for each register, whether these bits are zero when read, and ignored when written to, or are copied from source to destination when read or written to.

## B.5 Alignment Check

The AL bit in the POWER machine state register, MSR[24], is not supported in the PowerPC architecture. The bit is reserved in the PowerPC architecture. The low-order bits of the EA are always used. Notice that value zero—the normal value for a reserved SPR bit—means ignore the low-order EA bits in the POWER architecture, and value one means use the low-order EA bits. However, MSR[24] is not assigned new meaning in the PowerPC architecture.

## B.6 Condition Register

The following instructions specify a field in the condition register (CR) explicitly (via the **crf**D field) and also have the record bit (Rc) option. In the PowerPC architecture, if Rc = 1 for these instructions the instruction form is invalid. In the POWER architecture, if Rc = 1 the instructions execute normally except as shown in Table B-1.

**Table B-1. Condition Register Settings**

| Instruction | Setting |
|---|---|
| **cmp** | CR0 is undefined if Rc = 1 and **crf**D ≠ 0 |
| **cmpl** | CR0 is undefined if Rc = 1 and **crf**D ≠ 0 |
| **mcrxr** | CR0 is undefined if Rc = 1 and **crf**D ≠ 0 |
| **fcmpu** | CR1 is undefined if Rc = 1 |
| **fcmpo** | CR1 is undefined if Rc = 1 |
| **mcrfs** | CR1 is undefined if Rc = 1 and **crf**D ≠ 1 |

## B.7 Inappropriate Use of LK and Rc bits

For the instructions listed below, if LK = 1 or Rc = 1, POWER processors execute the instruction normally with the exception of setting the link register (if LK = 1) or the CR0 or CR1 fields (if Rc = 1) to an undefined value. In the PowerPC architecture, such instruction forms are invalid.

The PowerPC instruction form is invalid if LK = 1:

- **sc** (**svc***x* in the POWER architecture)
- Condition register logical instructions (that is, **crand**, **crandc**, **creqv**, **crnand**, **crnor**, **cror**, **crorc**, and **crxor**)
- **mcrf**
- **isync** (**ics** in the POWER architecture)

The PowerPC instruction form is invalid if Rc = 1:

- Integer X-form load and store instructions:
  - X-form load instructions—**lbzux**, **lbzx**, **lhaux**, **lhax**, **lhbrx**, **lhzux**, **lhzx**, **lswi**, **lswx**, **lwarx**, **lwbrx**, **lwzux**, **lwzx**
  - X-form store instructions—**stbux**, **stbx**, **sthbrx**, **sthux**, **sthx**, **stswi**, **stswx**, **stwbrx**, **stwcx.**, **stwux**, **stwx**
- Integer X-form compare instructions (that is, **cmp**, **cmpl**)
- X-form trap instruction (that is, **td**)
- **mtspr**, **mfspr**, **mtcrf**, **mcrxr**, **mfcr**
- Floating-point X-form load and store instructions and floating-point compare instructions
  - Floating-point X-form load instructions— **lfdux**, **lfdx**, **lfsux**, **lfsx**
  - Floating-point X-form store instructions—**stfdux**, **stfdx**, **stfiwx**, **stfsux**, **stfsx**
  - Floating-point X-form compare instruction—**fcmpo**, **fcmpu**
- **mcrfs**
- **dcbz** (**dclz** in the POWER architecture)

## B.8 BO Field

The POWER architecture shows certain bits in the BO field—used by branch conditional instructions—as *x* without indicating how these bits are to be interpreted. These bits are ignored by POWER processors.

The PowerPC architecture shows these bits as either *z* or *y*. The z bits are ignored, as in POWER. However, the *y* bit need not be ignored, but rather can be used to give a hint about whether the branch is likely to be taken. If a POWER program has the incorrect value for this bit, the program will run correctly but performance may suffer.

## B.9 Branch Conditional to Count Register

For the case in which the count register is decremented and tested (that is, the case in which BO[2] = 0), the POWER architecture specifies only that the branch target address is undefined, implying that the count register, and the link register (if LK = 1), are updated in the normal way. The PowerPC architecture considers this instruction form invalid.

## B.10 System Call/Supervisor Call

The System Call (**sc**) instruction in the PowerPC architecture is called Supervisor Call (**svc***x*) in the POWER architecture. Differences in implementations are as follows:

- The POWER architecture provides a version of the **svc***x* instruction (bit 30 = 0) that allows instruction fetching to continue at any one of 128 locations. It is used for “fast Supervisor Calls.” The PowerPC architecture provides no such version. If bit 30 of the instruction is zero the instruction form is invalid.
- The POWER architecture provides a version of the **svc***x* instruction (bits 30–31 = 0b11) that resumes instruction fetching at one location and sets the link register (LR) to the address of the next instruction. The PowerPC architecture provides no such version; if Rc = 1, the instruction form is invalid.
- For the POWER architecture, information from the MSR is saved in the count register (CTR). For the PowerPC architecture, this information is saved in the machine status save/restore register 1 (SRR1).
- The POWER architecture permits bits 16–29 of the instruction to be nonzero, while in the PowerPC architecture, such an instruction form is invalid.
- The POWER architecture saves the low-order 16 bits of the **svc***x* instruction in the CTR; the PowerPC architecture does not save them.
- The settings of the MSR bits by the system call exception differ between the POWER architecture and the PowerPC architecture.

## B.11 XER Register

Bits 16–23 of the XER are reserved in the PowerPC architecture, whereas in the POWER architecture they are defined to contain the comparison byte for the **lscbx** instruction, which is not included in the PowerPC architecture.

## B.12 Update Forms of Memory Access

The PowerPC architecture requires that **r**A not be equal to either **r**D (integer load only) or zero. If the restriction is violated, the instruction form is invalid. See Section 4.1.3, “Classes of Instructions,” for information about invalid instructions. The POWER architecture permits these cases and simply avoids saving the EA.

## B.13 Multiple Register Loads

When executing instructions that load multiple registers, the PowerPC architecture requires that **r**A, and **r**B if present in the instruction format, not be in the range of registers to be loaded, while the POWER architecture permits this and does not alter **r**A or **r**B in this case. (The PowerPC architecture restriction applies even if **r**A = 0, although there is no obvious benefit to the restriction in this case since **r**A is not used to compute the effective address if **r**A = 0.) If the PowerPC architecture restriction is violated, either the system illegal instruction error handler is invoked or the results are boundedly undefined.

The instructions affected are listed as follows:

- **lmw** (**lm** in the POWER architecture)
- **lswi** (**lsi** in the POWER architecture)
- **lswx** (**lsx** in the POWER architecture)

For example, an **lmw** instruction that loads all 32 registers is valid in the POWER architecture but is an invalid form in the PowerPC architecture.

## B.14 Alignment for Load/Store Multiple

When executing load/store multiple instructions, the PowerPC architecture requires the EA to be word-aligned and yields an alignment exception or boundedly-undefined results if it is not. The POWER architecture specifies that an alignment exception occurs (if AL = 1).

## B.15 Load and Store String Instructions

In the PowerPC architecture, an **lswx** instruction with zero length leaves the content of **r**D undefined (if **r**D ≠ **r**A and **r**D ≠ **r**B) or is an invalid instruction form (if **r**D = **r**A or **r**D = **r**B), while in the POWER architecture the corresponding instruction (**lsx**) is a no-op in these cases.

Note also that, in the PowerPC architecture, an **lswx** instruction with zero length may alter the referenced bit, and an **stswx** instruction with zero length may alter the referenced and changed bits, while in the POWER architecture the corresponding instructions (**lsx** and **stsx**) do not alter the referenced and changed bits.

## B.16 Synchronization

The **sync** instruction (called **dcs** in the POWER architecture) and the **isync** instruction (called the **ics** in the POWER architecture) cause a much more pervasive synchronization in the PowerPC architecture than in the POWER architecture. For more information, refer to Chapter 8, “Instruction Set.”

## B.17 Move to/from SPR

Differences in how the Move to/from Special Purpose Register (**mtspr** and **mfspr**) instructions function are as follows:

- The SPR field is 10 bits long in the PowerPC architecture, but only 5 bits in POWER architecture.
- The **mfspr** instruction can be used to read the decrementer (DEC) register in problem state (user mode) in the POWER architecture, but only in supervisor state in the PowerPC architecture.
- If the SPR value specified in the instruction is not one of the defined values, the POWER architecture behaves as follows:
  - If the instruction is executed in user-level privilege state and SPR[0] = 1, a supervisor-level instruction type program exception occurs. No architected registers are altered except those set by the exception.
  - If the instruction is executed in supervisor-level privilege state and SPR[0] = 0, no architected registers are altered.

  In this same case, the PowerPC architecture behaves as follows:
  - If the instruction is executed in user-level privilege state and SPR[0] = 1, either an illegal instruction type program exception or a supervisor-level instruction type program exception occurs. No architected registers are altered except those set by the exception.
  - Otherwise, (the instruction is executed in supervisor-level privilege state or SPR[0] = 0), either an illegal instruction type program exception occurs (in which case no architected registers are altered except those set by the exception) or the results are boundedly undefined.

## B.18 Effects of Exceptions on FPSCR Bits FR and FI

For the following cases, the POWER architecture does not specify how the FR and FI bits are set, while the PowerPC architecture preserves them for illegal operation exceptions caused by compare instructions and clears them otherwise.

- Invalid operation exception (enabled or disabled)
- Zero divide exception (enabled or disabled)

- Disabled overflow exception

## B.19 Floating-Point Store Single Instructions

There are several respects in which the PowerPC architecture is incompatible with the POWER architecture when executing store floating-point single instructions.

The POWER architecture uses FPSCR[UE] to help determine whether denormalization should be done, while the PowerPC architecture does not. Note that in the PowerPC architecture, if FPSCR[UE] = 1 and a denormalized single-precision number is copied from one memory location to another by means of an **lfs** instruction followed by an **stfs** instruction, the two copies may not be the same. Refer to Section 3.3.6.2.2, “Underflow Exception Condition,” for more information about underflow exceptions.

For an operand having an exponent that is less than 874 (an unbiased exponent less than -149), the POWER architecture specifies storage of a zero (if FPSCR[UE] = 0), while the PowerPC architecture specifies the storage of an undefined value.

## B.20 Move from FPSCR

The POWER architecture defines the high-order 32 bits of the result of **mffs** to be 0xFFFF_FFFF. In the PowerPC architecture they are undefined.

## B.21 Clearing Bytes in the Data Cache

The **dclz** instruction of the POWER architecture and the **dcbz** instruction of the PowerPC architecture have the same opcode. However, the functions differ in the following respects.

- The **dclz** instruction clears a line; **dcbz** clears a block.
- The **dclz** instruction saves the EA in **r**A (if **r**A ≠ 0); **dcbz** does not.
- The **dclz** instruction is supervisor-level; **dcbz** is not.

## B.22 Segment Register Instructions

The definitions of the four segment register instructions (**mtsr**, **mtsrin**, **mfsr**, and **mfsrin**) differ in two respects between the POWER architecture and the PowerPC architecture. Instructions similar to **mtsrin** and **mfsrin** are called **mtsri** and **mfsri** in the POWER architecture. The definitions follow:

- Privilege—**mfsr** and **mfsri** are problem state instructions in the POWER architecture, while **mfsr** and **mfsrin** are supervisor-level in the PowerPC architecture.
- Function—the indirect instructions (**mtsri** and **mfsri**) in the POWER architecture use an **r**A register in computing the segment register number, and the computed EA

is stored into **r**A (if **r**A ≠ 0 and **r**A ≠ **r**D); in the PowerPC architecture **mtsrin** and **mfsrin** have no **r**A field and EA is not stored.

The **mtsr**, **mtsrin** (**mtsri**), and **mfsr** instructions have the same opcodes in the PowerPC architecture as in the POWER architecture. The **mfsri** instruction in the POWER architecture and the **mfsrin** instruction in PowerPC architecture have different opcodes.

## B.23 TLB Entry Invalidation

The **tlbi** instruction in the POWER architecture and the **tlbie** instruction in the PowerPC architecture have the same opcode. However, the functions differ in the following respects.

- The **tlbi** instruction computes the EA as (**r**A|0) + **r**B, while **tlbie** lacks an **r**A field and computes the EA as **r**B.
- The **tlbi** instruction saves the EA in **r**A (if **r**A ≠ 0); **tlbie** lacks an **r**A field and does not save the EA.

## B.24 Floating-Point Exceptions

Both the PowerPC and the POWER architectures use bit 20 of the MSR to control the generation of exceptions for floating-point enabled exceptions. However, in the PowerPC architecture this bit is part of a 2-bit value which controls the occurrence, precision, and recoverability of the exception, whereas, in the POWER architecture this bit is used independently to control the occurrence of the exception (in the POWER architecture all floating-point exceptions are precise).

## B.25 Timing Facilities

This section describes differences between the POWER architecture and the PowerPC architecture timer facilities.

### B.25.1 Real-Time Clock

The POWER real-time clock (RTC) is not supported in the PowerPC architecture. Instead, the PowerPC architecture provides a time base register (TB). Both the RTC and the TB are 64-bit special-purpose registers, but they differ in the following respects:

- The RTC counts seconds and nanoseconds, while the TB counts ticks. The frequency of the TB is implementation-dependent.
- The RTC increments discontinuously—1 is added to RTCU when the value in RTCL passes 999_999_999. The TB increments continuously—1 is added to TBU when the value in TBL passes 0xFFFF_FFFF.

- The RTC is written and read by the **mtspr** and **mfspr** instructions, using SPR numbers that denote the RTCU and RTCD. The TB is written by the **mtspr** instruction (using new SPR numbers) and read by the new **mftb** instruction.
- The SPR numbers that denote POWER architectures's RTCL and RTCU are invalid in the PowerPC architecture.
- The RTC is guaranteed to increment at least once in the time required to execute ten Add Immediate (**addi**) instructions. No analogous guarantee is made for the TB.
- Not all bits of RTCL need be implemented, while all bits of the TB must be implemented.

### B.25.2 Decrementer

The decrementer (DEC) register differs, in the PowerPC and POWER architectures, in the following respects:

- The PowerPC architecture DEC register decrements at the same rate that the TB increments, while the POWER decrementer decrements every nanosecond (which is the same rate that the RTC increments).
- Not all bits of the POWER DEC need be implemented, while all bits of the PowerPC DEC must be implemented.
- The exception caused by the DEC has its own exception vector location in the PowerPC architecture, but is considered an external exception in the POWER architecture.

## B.26 Deleted Instructions

The following instructions, shown in Table B-2, are part of the POWER architecture but have been dropped from the PowerPC architecture.

**Table B-2. Deleted POWER Instructions**

| Mnemonic | Instruction | Primary Opcode | Extended Opcode |
|---|---|---|---|
| **abs** | Absolute | 31 | 360 |
| **clcs** | Cache Line Compute Size | 31 | 531 |
| **clf** | Cache Line Flush | 31 | 118 |
| **cli** | Cache Line Invalidate | 31 | 502 |
| **dclst** | Data Cache Line Store | 31 | 630 |
| **div** | Divide | 31 | 331 |
| **divs** | Divide Short | 31 | 363 |
| **doz** | Difference or Zero | 31 | 264 |
| **dozi** | Difference or Zero Immediate | 09 | — |

**Table B-2. Deleted POWER Instructions (continued)**

| Mnemonic | Instruction | Primary Opcode | Extended Opcode |
|---|---|---|---|
| **lscbx** | Load String and Compare Byte Indexed | 31 | 277 |
| **maskg** | Mask Generate | 31 | 29 |
| **maskir** | Mask Insert from Register | 31 | 541 |
| **mfsrin** | Move from Segment Register Indirect | 31 | 627 |
| **mul** | Multiply | 31 | 107 |
| **nabs** | Negative Absolute | 31 | 488 |
| **rac** | Real Address Compute | 31 | 818 |
| **rlmi** | Rotate Left then Mask Insert | 22 | — |
| **rrib** | Rotate Right and Insert Bit | 31 | 537 |
| **sle** | Shift Left Extended | 31 | 153 |
| **sleq** | Shift Left Extended with MQ | 31 | 217 |
| **sliq** | Shift Left Immediate with MQ | 31 | 184 |
| **slliq** | Shift Left Long Immediate with MQ | 31 | 248 |
| **sllq** | Shift Left Long with MQ | 31 | 216 |
| **slq** | Shift Left with MQ | 31 | 152 |
| **sraiq** | Shift Right Algebraic Immediate with MQ | 31 | 952 |
| **sraq** | Shift Right Algebraic with MQ | 31 | 920 |
| **sre** | Shift Right Extended | 31 | 665 |
| **srea** | Shift Right Extended Algebraic | 31 | 921 |
| **sreq** | Shift Right Extended with MQ | 31 | 729 |
| **sriq** | Shift Right Immediate with MQ | 31 | 696 |
| **srliq** | Shift Right Long Immediate with MQ | 31 | 760 |
| **srlq** | Shift Right Long with MQ | 31 | 728 |
| **srq** | Shift Right with MQ | 31 | 664 |
| **Note**: Many of these instructions use the MQ register. The MQ is not defined in the PowerPC architecture. | | | |

## B.27 POWER Instructions Supported by the PowerPC Architecture

Table B-3 lists the POWER instructions implemented in the PowerPC architecture.

**Table B-3. POWER Instructions Implemented in PowerPC Architecture**

| POWER Architecture | | PowerPC Architecture | |
|---|---|---|---|
| **Mnemonic** | **Instruction** | **Mnemonic** | **Instruction** |
| **a***x* | Add | **addc***x* | Add Carrying |
| **ae***x* | Add Extended | **adde***x* | Add Extended |
| **ai** | Add Immediate | **addic** | Add Immediate Carrying |
| **ai.** | Add Immediate and Record | **addic.** | Add Immediate Carrying and Record |
| **ame***x* | Add to Minus One Extended | **addme***x* | Add to Minus One Extended |
| **andil.** | AND Immediate Lower | **andi.** | AND Immediate |
| **andiu.** | AND Immediate Upper | **andis.** | AND Immediate Shifted |
| **aze***x* | Add to Zero Extended | **addze***x* | Add to Zero Extended |
| **bcc***x* | Branch Conditional to Count Register | **bcctr***x* | Branch Conditional to Count Register |
| **bcr***x* | Branch Conditional to Link Register | **bclr***x* | Branch Conditional to Link Register |
| **cal** | Compute Address Lower | **addi** | Add Immediate |
| **cau** | Compute Address Upper | **addis** | Add Immediate Shifted |
| **cax***x* | Compute Address | **add***x* | Add |
| **cntlz***x* | Count Leading Zeros | **cntlzw***x* | Count Leading Zeros Word |
| dclz | Data Cache Line Set to Zero | **dcbz** | Data Cache Block Set to Zero |
| **dcs** | Data Cache Synchronize | **sync** | Synchronize |
| **exts***x* | Extend Sign | **extsh***x* | Extend Sign Half Word |
| **fa***x* | Floating Add | **fadd***x* | Floating Add |
| **fd***x* | Floating Divide | **fdiv***x* | Floating Divide |
| **fm***x* | Floating Multiply | **fmul***x* | Floating Multiply |
| **fma***x* | Floating Multiply-Add | **fmadd***x* | Floating Multiply-Add |
| **fms***x* | Floating Multiply-Subtract | **fmsub***x* | Floating Multiply-Subtract |
| **fnma***x* | Floating Negative Multiply-Add | **fnmadd***x* | Floating Negative Multiply-Add |
| **fnms***x* | Floating Negative Multiply-Subtract | **fnmsub***x* | Floating Negative Multiply-Subtract |
| **fs***x* | Floating Subtract | **fsub***x* | Floating Subtract |
| ics | Instruction Cache Synchronize | **isync** | Instruction Synchronize |
| **l** | Load | **lwz** | Load Word and Zero |
| **lbrx** | Load Byte-Reverse Indexed | **lwbrx** | Load Word Byte-Reverse Indexed |
| **lm** | Load Multiple | **lmw** | Load Multiple Word |

## Table B-3. POWER Instructions Implemented in PowerPC Architecture (continued)

| POWER Architecture | | PowerPC Architecture | |
|---|---|---|---|
| **Mnemonic** | **Instruction** | **Mnemonic** | **Instruction** |
| **lsi** | Load String Immediate | **lswi** | Load String Word Immediate |
| **lsx** | Load String Indexed | **lswx** | Load String Word Indexed |
| **lu** | Load with Update | **lwzu** | Load Word and Zero with Update |
| **lux** | Load with Update Indexed | **lwzux** | Load Word and Zero with Update Indexed |
| **lx** | Load Indexed | **lwzx** | Load Word and Zero Indexed |
| **mtsri** | Move to Segment Register Indirect | **mtsrin** | Move to Segment Register Indirect * |
| **muli** | Multiply Immediate | **mulli** | Multiply Low Immediate |
| **muls***x* | Multiply Short | **mullw***x* | Multiply Low |
| **oril** | OR Immediate Lower | **ori** | OR Immediate |
| **oriu** | OR Immediate Upper | **oris** | OR Immediate Shifted |
| **rlimi***x* | Rotate Left Immediate then Mask Insert | **rlwimi***x* | Rotate Left Word Immediate then Mask Insert |
| **rlinm***x* | Rotate Left Immediate then AND With Mask | **rlwinm***x* | Rotate Left Word Immediate then AND with Mask |
| **rlnm***x* | Rotate Left then AND with Mask | **rlwnm***x* | Rotate Left Word then AND with Mask |
| **sf***x* | Subtract from | **subfc***x* | Subtract from Carrying |
| **sfe***x* | Subtract from Extended | **subfe***x* | Subtract from Extended |
| **sfi** | Subtract from Immediate | **subfic** | Subtract from Immediate Carrying |
| **sfme***x* | Subtract from Minus One Extended | **subfme***x* | Subtract from Minus One Extended |
| **sfze***x* | Subtract from Zero Extended | **subfze***x* | Subtract from Zero Extended |
| **sl***x* | Shift Left | **slw***x* | Shift Left Word |
| **sr***x* | Shift Right | **srw***x* | Shift Right Word |
| **sra***x* | Shift Right Algebraic | **sraw***x* | Shift Right Algebraic Word |
| **srai***x* | Shift Right Algebraic Immediate | **srawi***x* | Shift Right Algebraic Word Immediate |
| **st** | Store | **stw** | Store Word |
| **stbrx** | Store Byte-Reverse Indexed | **stwbrx** | Store Word Byte-Reverse Indexed |
| **stm** | Store Multiple | **stmw** | Store Multiple Word |
| **stsi** | Store String Immediate | **stswi** | Store String Word Immediate |
| **stsx** | Store String Indexed | **stswx** | Store String Word Indexed |
| **stu** | Store with Update | **stwu** | Store Word with Update |
| **stux** | Store with Update Indexed | **stwux** | Store Word with Update Indexed |
| **stx** | Store Indexed | **stwx** | Store Word Indexed |
| **svca** | Supervisor Call | **sc** | System Call |

Table B-3. POWER Instructions Implemented in PowerPC Architecture (continued)

| POWER Architecture | | PowerPC Architecture | |
|---|---|---|---|
| **Mnemonic** | **Instruction** | **Mnemonic** | **Instruction** |
| **t** | Trap | **tw** | Trap Word |
| **ti** | Trap Immediate | **twi** | Trap Word Immediate * |
| **tlbi** | TLB Invalidate Entry | **tlbie** | Translation Lookaside Buffer Invalidate Entry |
| **xoril** | XOR Immediate Lower | **xori** | XOR Immediate |
| **xoriu** | XOR Immediate Upper | **xoris** | XOR Immediate Shifted |

* Supervisor-level instruction

# Appendix C
# Multiple-Precision Shifts

This appendix gives examples of how multiple precision shifts can be programmed.

## C.1 Overview

A multiple-precision shift is initially defined to be a shift of an *n*-word quantity, where *n* > 1. The quantity to be shifted is contained in *n* registers. The shift amount is specified either by an immediate value in the instruction or by bits 27-31 of a register.

The examples shown below distinguish between the cases *n* = 2 and *n* > 2. However if *n* > 2, the shift amount must be in the range 0–31 for the examples to yield the desired result. The specific instance shown for *n* > 2 is *n* = 3: extending those 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. For non-immediate shifts, the shift amount is assumed to be in bits 27-31 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 2*n* – 1 (immediate shifts) or 3*n* – 1 (non-immediate shifts).

The following sections provide examples of multiple-precision shifts.

## C.2 Multiple-Precision Shifts in 32-Bit Implementations

**Shift Left Immediate, *n* = 3 (Shift Amount < 32)**

```
rlwinm  r2,r2,sh,0,31 – sh
rlwimi  r2,r3,sh,32 – sh,31
rlwinm  r3,r3,sh,0,31 – sh
rlwimi  r3,r4,sh,32 – sh,31
rlwinm  r4,r4,sh,0,31 – sh
```

**Shift Left, *n* = 2 (Shift Amount < 64)**

```
subfic  r31,r6,32
slw     r2,r2,r6
srw     r0,r3,r31
or      r2,r2,r0
addi    r31,r6,–32
slw     r0,r3,r31
or      r2,r2,r0
slw     r3,r3,r6
```

## Shift Left, *n* = 3 (Shift Amount < 32)

```
subfic  r31,r6,32
slw     r2,r2,r6
srw     r0,r3,r31
or      r2,r2,r0
slw     r3,r3,r6
srw     r0,r4,r31
or      r3,r3,r0
slw     r4,r4,r6
```

## Shift Right Immediate, *n* = 3 (Shift Amount < 32)

```
rlwinm  r4,r4,32 – sh,sh,31
rlwimi  r4,r3,32 – sh,0,sh – 1
rlwinm  r3,r3,32 – sh,sh,31
rlwimi  r3,r2,32 – sh,0,sh – 1
rlwinm  r2,r2,32 – sh,sh,31
```

## Shift Right, *n* = 2 (Shift Amount < 64)

```
subfic  r31,r6,32
srw     r3,r3,r6
slw     r0,r2,r31
or      r3,r3,r0
addi    r31,r6, –32
srw     r0,r2,r31
or      r3,r3,r0
srw     r2,r2,r6
```

## Shift Right, *n* = 3 (Shift Amount < 32)

```
subfic  r31,r6,–32
srw     r4,r4,r6
slw     r0,r3,r31
or      r4,r4,r0
srw     r3,r3,r6
slw     r0,r2,r31
or      r3,r3,r0
srw     r2,r2,r6
```

## Shift Right Algebraic Immediate, *n* = 3 (Shift Amount < 32)

```
rlwinm  r4,r4,32 – sh,sh,31
rlwimi  r4,r3,32 – sh,0,sh – 1
rlwinm  r3,r3,32 – sh,sh,31
```

```
rlwimi  r3,r2,32 – sh,0,sh – 1
srawi   r2,r2,sh
```

## Shift Right Algebraic, *n* = 2 (Shift Amount < 64)

```
subfic  r31,r6,32
srw     r3,r3,r6
slw     r0,r2,r31
or      r3,r3,r0
addic.  r31,r6,–32
sraw    r0,r2,r31
ble     $+8
ori     r3,r0,0
sraw    r2,r2,r6
```

## Shift Right Algebraic, *n* = 3 (Shift Amount < 32)

```
subfic  r31,r6,32
srw     r4,r4,r6
slw     r0,r3,r31
or      r4,r4,r0
srw     r3,r3,r6
slw     r0,r2,r31
or      r3,r3,r0
sraw    r2,r2,r6
```

# Appendix D
# 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.

## D.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 D-1, comprise the significand of the intermediate result.

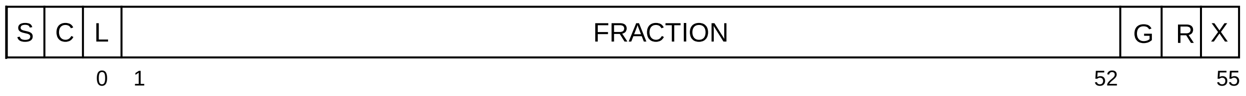


**Figure D-1. IEEE 64-Bit Execution Model**

The bits and fields for the IEEE double-precision execution model are defined in Table D-1.

**Table D-1. IEEE 64-Bit Execution Model Field Descriptions**

| Bits | Description |
|---|---|
| S | Sign bit. |
| C | Carry bit that captures the carry out of the significand. |
| L | Leading unit bit of the significand that receives the implicit bit from the operands. |
| FRACTION | 52-bit field that accepts the fraction of the operands. |
| G<br>R<br>X | Guard, round, and sticky. These bits are extensions to the low-order accumulator bits. G and R are required for postnormalization of the result. G, R, and X are required during rounding to determine if the intermediate result is equally near the two nearest representable values. This is shown in Table D-2. X 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. G and R participate in the left shifts with zeros being shifted into the R bit. |

Table D-2 shows the significance of G, R, and 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 D-2. Interpretation of G, R, and X Bits**

| G | R | X | Interpretation |
|---|---|---|---|
| 0 | 0 | 0 | IR is exact |
| 0 | 0 | 1 | |
| 0 | 1 | 0 | IR closer to NL |
| 0 | 1 | 1 | |
| 1 | 0 | 0 | IR midway between NL & NH |
| 1 | 0 | 1 | |
| 1 | 1 | 0 | IR closer to NH |
| 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 D-3 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 D-3. 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 occurs:
  - 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 a 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.

## D.2 Multiply-Add Type Instruction Execution Model

The architecture uses of a special instruction form that performs up to three operations in one instruction (a multiply, an add, and a negate). With this comes an 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 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.

As Figure D-2 shows, 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.

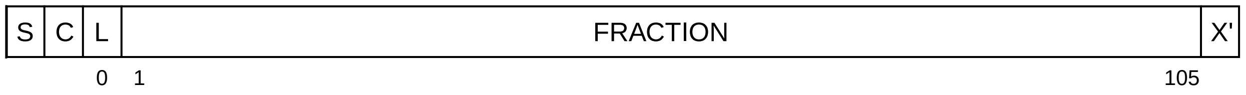


**Figure D-2. Multiply-Add 64-Bit Execution Model**

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. Figure D-4 shows the positions of these bits for double-precision and single-precision floating-point numbers in the multiply-add execution model.

**Table D-4. Location of G, R, and X Bits—Multiply-Add Execution Model**

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

The rules for rounding the intermediate result are the same as those given in Section D.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, 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.

## D.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.

### D.3.1 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)
```

### D.3.2 Conversion from Floating-Point Number to Unsigned Fixed-Point Integer Word

In a 32-bit implementation, the full convert to unsigned fixed-point integer word function can be implemented with the sequence below, assuming that 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 value $2^{31}$ is in FPR4, the result is returned in GPR3, and a double word at displacement (disp) from the address in GPR1 can be used as scratch space.

```
fsel    f2,f1,f1,f0     #use 0 if < 0
fsub    f5,f3,f1        #use max if > max
fsel    f2,f5,f2,f3
fsub    f5,f2,f4        #subtract 2**31
fcmpu   cr2,f2,f4       #use diff if ≥ 2**31
fsel    f2,f5,f5,f2
fctiw[z]f2,f2           #convert to fx int
stfd    f2,disp(r1)     #store float
lwz     r3,disp + 4(r1) #load word
blt     cr2,$+8         #add 2**31 if input
xoris   r3,r3,0x8000    #was ≥ 2**31
```

## D.4 Floating-Point Models

This section describes models for floating-point instructions.

### D.4.1 Floating-Point Round to Single-Precision Model

The following describes the operation of the **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
        **fr**D[0] ← sign
        **fr**D[1–11] ← exp + 1023
        **fr**D[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 **fr**B[0] = 1 then **fr**D ← 0xFFF0_0000_0000_0000
        If **fr**B[0] = 0 then FPSCR[FPRF] ← “+normal number”
        If **fr**B[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
```

## D.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 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 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
```

## D.4.3 Floating-Point Convert from Integer Model

The following describes the operation of 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 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
```

## D.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, floating-point variables *a*, *b*, *x*, *y*, and *z* are assumed to be in FPRs *fa*, *fb*, *fx*, *fy*, and *fz*. FPR *fs* is assumed to be available for scratch space.

Additional examples can be found in Section D.3, “Floating-Point Conversions.”

Note that care must be taken in using **fse**l if IEEE compatibility is required, or if the values being tested can be NaNs or infinities; see Section D.5.4, “Notes.”

### D.5.1 Comparison to Zero

This section provides examples in a program fragment code sequence for the comparison to zero case.

**High-level language:** **PowerPC:**

```
if a ≥ 0.0 then x ← y     fsel  fx, fa, fy, fz (see Section D.5.4, “Notes” number 1)
           else x ← z

if a > 0.0 then x ← y     fneg  fs, fa
           else x ← z     fsel  fx, fs, fz, fy (see Section D.5.4, “Notes” numbers 1 and 2)

if a = 0.0 then x ← y     fsel  fx, fa, fy, fz
           else x ← z     fneg  fs, fa
                          fsel  fx, fs, fx, fz (see Section D.5.4, “Notes” number 1)
```

### D.5.2 Minimum and Maximum

This section provides program fragment code examples for minimum and maximum cases.

**High-level language:** **PowerPC:**

```
x ← min(a, b)   fsub  fs, fa, fb (see Section D.5.4, “Notes” numbers 3, 4, and 5)
                fsel  fx, fs, fb, fa

x ← max(a, b)   fsub  fs, fa, fb (see Section D.5.4, “Notes” numbers 3, 4, and 5)
                fsel  fx, fs, fa, fb
```

## D.5.3 Simple If-Then-Else Constructions

This section provides examples in a program fragment code sequence for simple if-then-else statements.

**High-level language:** **PowerPC:**

```
if a ≥ b then x ← yfsubfs, fa, fb
      else x ← zfsel         fx, fs, fy, fz (see Section D.5.4, “Notes” numbers 4 and 5)

if a >b then x ← yfsub       fs, fb, fa
      else x ← zfsel         fx, fs, fz, fy (see Section D.5.4, “Notes” numbers 3, 4, and 5)

if a = b then x ← yfsub      fs, fa, fb
      else x ← zfsel         fx, fs, fy, fz
        fneg                 fs, fs
        fsel                 fx, fs, fx, fz (see Section D.5.4, “Notes” numbers 4 and 5)
```

## D.5.4 Notes

The following notes apply to the examples found in Section D.5.1, “Comparison to Zero,” Section D.5.2, “Minimum and Maximum,” and Section D.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 program shown and the unoptimized program (not shown) is the corresponding program that uses **fcmpu** and branch conditional instructions instead of **fsel**.

1. The unoptimized program affects FPSCR[VXSNAN] and so may cause the system error handler to be invoked if the corresponding exception is enabled; the optimized program does not affect this bit. This property of the optimized program is incompatible with the IEEE standard.
2. The optimized program gives the incorrect result if ‘a’ is a NaN.
3. The optimized program gives the incorrect result if ‘a’ and/or ‘b’ is a NaN (except that it may give the correct result in some cases for the minimum and maximum functions, depending on how those functions are defined to operate on NaNs).
4. The optimized program gives the incorrect result if ‘a’ and ‘b’ are infinities of the same sign. (Here it is assumed that invalid operation exceptions are disabled, in which case the result of the subtraction is a NaN. The analysis is more complicated if invalid operation exceptions are enabled, because in that case the target register of the subtraction is unchanged.)
5. The optimized program affects FPSCR[OX,UX,XX,VXISI], and therefore may cause the system error handler to be invoked if the corresponding exceptions are enabled, while the unoptimized program does not affect these bits. This property of the optimized program is incompatible with the IEEE standard.

## D.6 Floating-Point Load Instructions

There are two basic forms of load instruction—single-precision and double-precision. Because FPRs support only double-precision format, single-precision load floating-point instructions convert single-precision data to double-precision format. 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 **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 FPR specified by operand **fr**D; if operand **r**A = 0, the instruction form is invalid.

## D.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 optional **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
```

Note 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 does not compare equal to the contents of the original source register).

Note that 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.

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.

# Appendix E
# 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.

## E.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 Section E.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 Section E.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 Section E.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 Section E.5, "List Insertion," can cause livelock if two list elements have next element pointers in the same reservation granule.

## E.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.

### E.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).

### E.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
```

### E.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
```

### E.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.).

### E.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
```

## E.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:**

- The semantics in this example are based on the IBM System/370™ compare and swap instruction. Other architectures may define this instruction differently.
- 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.
- 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.

## E.4 Lock Acquisition and Release

This example provides an algorithm for locking that demonstrates the use of synchronization with an atomic read/modify/write operation. GPR3 provides a shared memory location, the address of which is an argument of the lock and unlock procedures. This argument is used as a lock to control access to some shared resource such as a data structure. The lock is open when its value is zero and locked when it is one. Before accessing the shared resource, a processor sets the lock by having the lock procedure call TEST_AND_SET, which executes the code sequence in Section E.2.5, “Test and Set.” This atomically sets the old value of the lock, and writes the new value (1) given to it in GPR4, returning the old value in GPR5 (not used in the following example) and setting the EQ bit in CR0 according to whether the value loaded is zero. The lock procedure repeats the test and set procedure until it successfully changes the value in the lock from zero to one.

The processor must not access the shared resource until it sets the lock. After the **bne-** instruction that checks for the successful test and set operation, the processor executes the **isync** instruction. This delays all subsequent instructions until all previous instructions have completed to the extent required by context synchronization. The **sync** instruction could be used but performance would be degraded because the **sync** instruction waits for all outstanding memory accesses to complete with respect to other processors. This is not necessary here.

```
lock:   li      r4,1            #obtain lock
loop:   bl      test_and_set    #test and set
        bne-    loop            #retry until old = 0
                                #delay subsequent instructions until
                                #previous ones complete
        isync
        blr                     #return
```

The unlock procedure writes a zero to the lock location. If the access to the shared resource includes write operations, most applications that use locking require the processor to execute a **sync** instruction to make its modification visible to all processors before releasing the lock. For this reason, the unlock procedure in the following example begins with a **sync**.

```
unlock: sync                    #delay until prior stores finish
        li      r1,0
        stw     r1,0(r3)        #store zero to lock location
        blr                     #return
```

## E.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
        sync            #let store settle (can omit if not MP)
        stwcx.  r4,0,r3 #add new element to list
        bne-    loop    #loop if stwcx. failed
```

In the preceding example, if two list elements have next element pointers in the same reservation granule in a multiprocessor system, livelock can occur.

**List Insertion**

If it is not possible to allocate list elements such that each element's next element pointer is in a different reservation granule, livelock can be avoided by using the following sequence:

```
        lwz     r2,0(r3)#get next pointer
loopl:  mr      r5,r2   #keep a copy
        stw     r2,0(r4)#store in new element
        sync            #let store settle
loop2:  lwarx   r2,0,r3 #get it again
        cmpw    r2,r5   #loop if changed (someone
        bne-    loopl   #else progressed)
        stwcx.  r4,0,r3 #add new element to list
        bne-    loop2   #loop if failed
```

# Appendix F
# Simplified Mnemonics

This appendix describes simplfied mnemonics which are provided for easier coding of assembly language programs. Simplified mnemonics are defined for the most frequently used forms of branch conditional, compare, trap, rotate and shift, and certain other instructions. (Note that the architecture specification refers to simplified mnemonics as extended mnemonics.)

## F.1 Symbols

The symbols in Table F-1 are defined for use in instructions (basic or simplified mnemonics) that specify a condition register (CR) field or bit.

**Table F-1. Condition Register Bit and Identification Symbol Descriptions**

| Symbol | Value | Bit Field Range [1] | 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 |

[1] 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 that the simplified mnemonics in Section F.5.2, “Basic Branch Mnemonics,” and Section F.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 Section F.5.3, "Branch Mnemonics Incorporating Conditions," and Section F.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 Section F.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.)

## F.2 Simplified Mnemonics for Subtract Instructions

This section discusses simplified mnemonics for the subtract instructions.

### F.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.

**Table 8-17. Subtract Immediate Simplified Mnemonics**

| Simplified Mnemonic | Standard Mnemonic |
|---|---|
| **subi r**D,**r**A,value | **addi r**D,**r**A,–value |
| **subis r**D,**r**A,value | **addis r**D,**r**A,–value |
| **subic r**D,**r**A,value | **addic r**D,**r**A,–value |
| **subic. r**D,**r**A,value | **addic. r**D,**r**A,–value |

### F.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.

**Table 8-18. Subtract Immediate Simplified Mnemonics**

| Simplified Mnemonic | Standard Mnemonic |
|---|---|
| **sub r**D,**r**A,**r**B | **subf r**D,**r**B,**r**A |
| **subc r**D,**r**A,**r**B | **subfcr**D,**r**B,**r**A |

## F.3 Simplified Mnemonics for Compare Instructions

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 Section F.1, “Symbols,” can be used for this operand.

Note that the basic compare mnemonics are the same as those of POWER, but the POWER instructions have three operands while PowerPC instructions have four. The assembler recognizes a basic compare mnemonic with the three operands as the POWER form, and generates the instruction with L = 0. The **crf**D field can normally be omitted when the CR0 field is the target.

### F.3.1 Word Comparisons

The instructions listed in Table F-2 are simplified mnemonics that should be supported by assemblers for all implementations.

**Table F-2. 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 with immediate value 100 as signed 32-bit integers and place result in CR0.
   **cmpwi r**A**,100** equivalent to **cmpi 0,0,r**A**,100**
2. Same as (1), but place results in CR4.
   **cmpwi cr4,r**A**,100** equivalent to **cmpi 4,0,r**A**,100**
3. Compare **r**A and **r**Bas unsigned 32-bit integers and place result in CR0.
   **cmplw r**A**,r**B equivalent to **cmpl 0,0,r**A**,r**B

## F.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.

## F.4.1 Operations on Words

The operations shown in Table F-3 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.

**Table F-3. 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 0) of **r**S and place the result right-justified into **r**A.
   **extrwi r**A**,r**S**,1,0** equivalent to **rlwinm r**A**,r**S**,1,31,31**
2. Insert the bit extracted in (1) into the sign bit (bit 0) of **r**B.
   **insrwi r**B**,r**A**,1,0** equivalent to **rlwimi r**B**,r**A**,31,0,0**

3. Shift the contents of **r**A left 8 bits.

   | | | |
   |---|---|---|
   | **slwi r**A**,r**A**,8** | equivalent to | **rlwinm r**A**,r**A**,8,0,23** |

4. Clear the high-order 16 bits of **r**S and place the result into **r**A.

   | | | |
   |---|---|---|
   | **clrlwi r**A**,r**S**,16** | equivalent to | **rlwinm r**A**,r**S**,0,16,31** |

## F.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.

Mnemonics discussed in this section are variations of the branch conditional instructions.

### F.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 Section F.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.

### F.5.2 Basic Branch Mnemonics

The mnemonics in Table F-1Table F-4 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 F-4 provides the abbreviated set of simplified mnemonics for the most commonly performed conditional branches.

Table F-4. 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 F-4 that test a condition require a corresponding CR bit as the first operand of the instruction. The symbols defined in Section F.1, "Symbols," can be used in the operand in place of a numeric value.

The simplified mnemonics found in Table F-4 are used in the following examples:

1. Decrement CTR and branch if it is still nonzero (closure of a loop controlled by a count loaded into CTR).
   **bdnz** target  equivalent to  **bc 16,0,**target
2. Same as (1) but branch only if CTR is non-zero and equal condition in CR0 .
   **bdnzt eq,**target  equivalent to  **bc 8,2,**target
3. Same as (2), but equal condition is in CR5.
   **bdnzt 4 * cr5 + eq,**target  equivalent to  **bc 8,22,**target
4. Branch if bit 27 of CR is false.
   **bf 27,**target  equivalent to  **bc 4,27,**target
5. Same as (4), but set the link register. This is a form of conditional call.
   **bfl 27,**target  equivalent to  **bcl 4,27,**target

Table F-5 provides the simplified mnemonics for the **bc** and **bca** instructions without link register updating, and the syntax associated with these instructions. Note that the default CR specified by the simplified mnemonics in the table is CR0.

**Table F-5. Simplified Mnemonics for bc and bca without LR Update**

| Branch Semantics | 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 F-6 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 CR specified by the simplified mnemonics in the table is CR0.

**Table F-6. Simplified Mnemonics for bclr and bcclr without LR Update**

| Branch Semantics | 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 F-7 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 CR specified by the simplified mnemonics in the table is CR0.

**Table F-7. Simplified Mnemonics for bcl and bcla \with LR Update**

| Branch Semantics | bcl Relative | Simplified Mnemonic | bcla Absolute | Simplified Mnemonic |
|---|---|---|---|---|
| Branch unconditionally | — | — | — | — |
| Branch if condition true | **bcl1** 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 F-8 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 CR specified by the simplified mnemonics in the table is CR0.

**Table F-8. Simplified Mnemonics for bclrl and bcctrl with LR 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 | — | — |

## F.5.3 Branch Mnemonics Incorporating Conditions

The mnemonics defined in Table F-4 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 F-9) has been adopted for the most common combinations of branch conditions.

**Table F-9. 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 F-10 shows the simplified branch mnemonics incorporating conditions.

**Table F-10. Simplified 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 F-10. Simplified 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 F-10 specify the CR 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 Section F.1, “Symbols,” can be used for this operand.

The simplified mnemonics found in Table F-10 are used in the following examples:

1. Branch if CR0 reflects not equal condition .
   **bne** target equivalent to **bc 4,2,**target
2. Same as (1) but condition is in CR3.
   **bne cr3,**target equivalent to **bc 4,14,**target
3. Branch to an absolute target if CR4 specifies greater than condition, setting the LR. This is a form of conditional call.
   **bgtla cr4,**target equivalent to **bcla 12,17,**target
4. Same as (3), but target address is in the CTR.
   **bgtctrl cr4** equivalent to **bcctrl 12,17**

Table F-11 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 CR specified by the simplified mnemonics in the table is CR0.

**Table F-11. Simplified Mnemonics for bc and bca without Comparison Conditions and LR Updating**

| Branch Semantics | 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 |

Table F-11. Simplified Mnemonics for bc and bca without Comparison Conditions and LR Updating

| Branch Semantics | bc Relative | Simplified Mnemonic | bca Absolute | Simplified Mnemonic |
|---|---|---|---|---|
| Branch if summary overflow | **bc** 12,3,target | **bso** target | **bca** 12,3,target | **bsoa** target |
| 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 F-12 shows the simplified branch mnemonics for the **bclr** and **bcctr** instructions without link register updating, and the syntax associated with these instructions. Note that the default CR specified by the simplified mnemonics in the table is CR0.

Table F-12. Simplified Mnemonics for bclr and bcctr without Comparison Conditions and LR Updating

| Branch Semantics | 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 F-13 shows the simplified branch mnemonics for the **bcl** and **bcla** instructions with link register updating, and the syntax associated with these instructions. Note that the default CR specified by the simplified mnemonics in the table is CR0.

Table F-13. Simplified Mnemonics for bcl and bcla with Comparison Conditions and LR Update

| Branch Semantics | 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 |

**Table F-13. Simplified Mnemonics for bcl and bcla with Comparison Conditions and LR Update**

| Branch Semantics | bcl Relative | Simplified Mnemonic | bcla Absolute | Simplified Mnemonic |
|---|---|---|---|---|
| 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 F-14 shows the simplified branch mnemonics for the **bclrl** and **bcctl** instructions with link register updating, and the syntax associated with these instructions. Note that the default CR specified by the simplified mnemonics in the table is CR0.

**Table F-14. Simplified Mnemonics for bclrl and bcctl with Comparison Conditions and LR Update**

| Branch Semantics | 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 |

## F.5.4 Branch Prediction

In branch conditional instructions that are not always taken, the low-order bit (*y* bit) of the BO field provides a hint about whether the branch is likely to be taken. See Section 4.2.4.2, “Conditional Branch Control,” for more information on the *y* bit.

Assemblers should clear this bit unless otherwise directed. This default action indicates the following:

- A branch conditional with a negative displacement field is predicted to be taken.
- A branch conditional with a non-negative displacement field is predicted not to be taken (fall through).
- A branch conditional to an address in the LR or CTR is predicted not to be taken (fall through).

If the likely outcome (branch or fall through) of a given branch conditional instruction is known, a suffix can be added to the mnemonic that tells the assembler how to set the *y* bit. That is, ‘+’ indicates that the branch is to be taken and ‘–’ indicates that the branch is not to be taken. Such a suffix can be added to any branch conditional mnemonic, either basic or simplified.

For relative and absolute branches (**bc**[**l**][**a**]), the setting of the *y* bit depends on whether the displacement field is negative or non-negative. For negative displacement fields, coding the suffix ‘+’ causes the bit to be cleared, and coding the suffix ‘–’ causes the bit to be set. For non-negative displacement fields, coding the suffix ‘+’ causes the bit to be set, and coding the suffix ‘–’ causes the bit to be cleared.

For branches to an address in the LR or CTR (**bcclr**[**l**] or **bcctr**[**l**]), coding the suffix ‘+’ causes the *y* bit to be set, and coding the suffix ‘–’ causes the bit to be cleared.

Examples of branch prediction follow:

1. Branch if CR0 reflects less than condition, specifying that the branch should be predicted to be taken.
   **blt+** target
2. Same as (1), but target address is in the LR and the branch should be predicted not to be taken.
   **bltlr–**

## F.6 Simplified Mnemonics for Condition Register Logical Instructions

The CR logical instructions, shown in Table F-15, can be used to set, clear, copy, or invert a given CR bit. Simplified mnemonics allow these operations to be coded easily. Note that the symbols defined in Section F.1, “Symbols,” can be used to identify the CR bit.

**Table F-15. 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** |

Table F-15. Condition Register Logical Mnemonics (continued)

| Operation | Simplified Mnemonic | Equivalent to |
|---|---|---|
| Condition register move | **crmove bx,by** | **cror bx,by,by** |
| Condition register not | **crnot bx,by** | **crnor bx,by,by** |

Examples using the CR logical mnemonics follow:

1. Set CR[25].
   **crset 25** equivalent to **creqv 25,25,25**
2. Clear CR0[SO].
   **crclr so** equivalent to **crxor 3,3,3**
3. Same as (2), but clear CR3[SO].
   **crclr 4 * cr3 + so** equivalent to **crxor 15,15,15**
4. Invert the EQ bit.
   **crnot eq,eq** equivalent to **crnor 2,2,2**
5. Same as (4), but CR4[EQ] is inverted and the result is placed into CR5[EQ].
   **crnot 4 * cr5 + eq, 4 * cr4 + eq** equivalent to **crnor 22,18,18**

## F.7 Simplified Mnemonics for Trap Instructions

A standard set of codes, shown in Table F-16, has been adopted for the most common combinations of trap conditions.

Table F-16. Standard Codes for Trap Instructions

| Code | Description | TO Encoding | < | > | = | <U [1] | >U [2] |
|---|---|---|---|---|---|---|---|
| lt | Less than | 16 | 1 | 0 | 0 | 0 | 0 |
| le | Less than or equal | 20 | 1 | 0 | 1 | 0 | 0 |
| eq | Equal | 4 | 0 | 0 | 1 | 0 | 0 |
| ge | Greater than or equal | 12 | 0 | 1 | 1 | 0 | 0 |
| gt | Greater than | 8 | 0 | 1 | 0 | 0 | 0 |
| nl | Not less than | 12 | 0 | 1 | 1 | 0 | 0 |
| ne | Not equal | 24 | 1 | 1 | 0 | 0 | 0 |
| ng | Not greater than | 20 | 1 | 0 | 1 | 0 | 0 |
| llt | Logically less than | 2 | 0 | 0 | 0 | 1 | 0 |
| lle | Logically less than or equal | 6 | 0 | 0 | 1 | 1 | 0 |
| lge | Logically greater than or equal | 5 | 0 | 0 | 1 | 0 | 1 |
| lgt | Logically greater than | 1 | 0 | 0 | 0 | 0 | 1 |
| lnl | Logically not less than | 5 | 0 | 0 | 1 | 0 | 1 |

**Table F-16. Standard Codes for Trap Instructions (continued)**

| Code | Description | TO Encoding | < | > | = | <U [1] | >U [2] |
|---|---|---|---|---|---|---|---|
| lng | Logically not greater than | 6 | 0 | 0 | 1 | 1 | 0 |
| — | Unconditional | 31 | 1 | 1 | 1 | 1 | 1 |

[1] The symbol '<U' indicates an unsigned less than evaluation will be performed.

[2] The symbol '>U' indicates an unsigned greater than evaluation will be performed.

The mnemonics defined in Table F-18 are variations of trap instructions, with the most useful values of TO represented in the mnemonic rather than specified as a numeric operand.

**Table F-18. Trap Mnemonics**

| Trap Semantics | 32-Bit Comparison | |
|---|---|---|
| | twi Immediate | tw Register |
| Trap unconditionally | — | trap |
| Trap if less than | **twlti** | **twlt** |
| Trap if less than or equal | **twlei** | **twle** |
| Trap if equal | **tweqi** | **tweq** |
| Trap if greater than or equal | **twgei** | **twge** |
| Trap if greater than | **twgti** | **twgt** |
| Trap if not less than | **twnli** | **twnl** |
| Trap if not equal | **twnei** | **twne** |
| Trap if not greater than | **twngi** | **twng** |
| Trap if logically less than | **twllti** | **twllt** |
| Trap if logically less than or equal | **twllei** | **twlle** |
| Trap if logically greater than or equal | **twlgei** | **twlge** |
| Trap if logically greater than | **twlgti** | **twlgt** |
| Trap if logically not less than | **twlnli** | **twlnl** |
| Trap if logically not greater than | **twlngi** | **twlng** |

Examples of the uses of trap mnemonics, shown in , Table F-18 follow:

1. Trap if register **r**A is not zero.
   **twnei** **r**A,**0** equivalent to **twi 24,r**A,**0**
2. Trap if register **r**A is not equal to **r**B.
   **twne** **r**A, **r**B equivalent to **tw 24,r**A,**r**B
3. Trap if **r**A is logically greater than 0x7FF.
   **twlgti r**A, 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.

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 F-19 for these conditions.

**Table F-19. 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 |

## F.8 Simplified Mnemonics for SPRs

The **mtspr** and **mfspr** instructions specify a special-purpose register (SPR) as a numeric operand. Simplified mnemonics are provided that represent the SPR in the mnemonic rather than requiring it to be coded as a numeric operand. Table F-20 provides a list of the simplified mnemonics that should be provided by assemblers for SPR operations.

**Table F-20. Simplified Mnemonics for SPRs**

| SPR | 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** |

**Table F-20. Simplified Mnemonics for SPRs (continued)**

| SPR | Move to SPR | | Move from SPR | |
|---|---|---|---|---|
| | Simplified Mnemonic | Equivalent to | Simplified Mnemonic | Equivalent to |
| 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** |
| IBAT register, upper | **mtibatu** *n,* **r**S | **mtspr** 528 + (2 * *n*),**r**S | **mfibatu r**D*, n* | **mfspr r**D,528 + (2 * *n*) |
| IBAT register, lower | **mtibatl** *n,* **r**S | **mtspr** 529 + (2 * *n*),**r**S | **mfibatl r**D*, n* | **mfspr r**D,529 + (2 * *n*) |
| DBAT register, upper | **mtdbatu** *n,* **r**S | **mtspr** 536 + (2 **n*),**r**S | **mfdbatu r**D*, n* | **mfspr r**D,536 + (2 **n*) |
| DBAT register, lower | **mtdbatl** *n,* **r**S | **mtspr** 537 + (2 * *n*),**r**S | **mfdbatl r**D*, n* | **mfspr r**D,537 + (2 * *n*) |

Following are examples using the SPR simplified mnemonics found in Table F-20:

1. Copy the contents of **r**S to the XER.
   **mtxer r**S equivalent to **mtspr 1,r**S
2. Copy the contents of the LR to **r**S.
   **mflr r**S equivalent to **mfspr r**S**,8**
3. Copy the contents of **r**S to the CTR.
   **mtctr r**S equivalent to **mtspr 9,r**S

## F.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).

### F.9.1 No-Op (nop)

Many 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**

## F.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 r**D**,**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 r**D**,**value | equivalent to | **addis r**D**,0,**value |

## F.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 r**D**,**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 r**D**,***v* | equivalent to | **addi r**D**,r***v***,**d*v* |

## F.9.4 Move Register (mr)

Several instructions can be coded to copy the contents of one register to another. A simplified mnemonic is provided that signifies that no computation is being performed, but merely that data is being moved from one register to another.

The following instruction copies the contents of **r**S into **r**A. This mnemonic can be coded with a dot (**.**) suffix to cause the Rc bit to be set in the underlying instruction.

| | | |
|---|---|---|
| **mr r**A**,r**S | equivalent to | **or r**A**,r**S**,r**S |

## F.9.5 Complement Register (not)

Several instructions can be coded in a way that they complement the contents of one register and place the result into another register. 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 r**A**,r**S | equivalent to | **nor r**A**,r**S**,r**S |

### F.9.6 Move to Condition Register (mtcr)

This mnemonic permits copying the contents of a GPR to the CR, using the same syntax as the **mfcr** instruction.

| | | |
|---|---|---|
| **mtcr r**S | equivalent to | **mtcrf** 0xFF**,r**S |

# Appendix G
# *Programming Environments Manual (32-Bit)* Revision History

This errata describes corrections to the previous revision of the *Programming Environments Manual (32-Bit)*. For convenience, the section and page numbers of the change in the previous edition of the user's manual are provided.

Major changes since the previous revision of this document are as follows.

| **Section, Page** | **Change** |
|---|---|
| 2.1.1, 2-4 | Figure 2-2 should show the GPR bit numbering as 0–31 (instead of 0–63). |
| 2.1.6, 2-12 | Figure 2-2 should show the link register bit numbering as 0–31 (instead of 0–63). |
| 2.1.7, 2-12 | Figure 2-2 should show the count register bit numbering as 0–31 (instead of 0–63). |
| 3.1.3.1, 3-4 | The first sentence in the last paragraph has a faulty cross reference. It should read as follows:<br>The structure mapping introduces padding (skipped bytes indicated by (x) in Figure 3-2)... |
| 3.1.6, 5-12 | The section "Shared Memory," has been added to clarify matters concerning memory access ordering. |
| 7.4.6, 7-32 | In Figure 7-11, PA0–PA63 should be PA00–PA31 for 32-bit implementations. |
| 7.6.1, 7-49 | In Figure 7-18, the size of a PTE should be 8 bytes not 16. |
| 7.5.5, 7-47 | Figure 7-16 should read "PA0–PA31<- RPN\|\|A20–A31" instead of "PA0–PA31<- RPN\|\|A30–A31." |
| 7.6.1.1, 7-51 | Replace the paragraph under the heading "Example" with the following:<br>For example, suppose that the page table is 16,384 ($2^{14}$) 64-byte PTEGs, for a total size of $2^{20}$ bytes (1 Mbyte). A 14-bit index is required. Ten bits are provided from the hash to start with, so 4 additional bits from the hash must be selected. Thus the value in |

| | |
|---|---|
| | HTABMASK must be 15 and the value in HTABORG must have its low-order 4 bits (SDR1[12–15]) equal to 0. This means that the page table must begin on a $2^{<4 + 10 + 6>} = 2^{20}$ = 1-Mbyte boundary. |
| 7.6.1.3, 7-53 | Figure 7-20 should show that the primary hash is formed by XORing with page index bits 24–39 from the virtual address or EA[4–19] (instead of EA[24–39]. |
| 7.6.1.5, 7-56 | The first sentence in the second paragraph should read as follows: “The physical address of the PTE(s) to be checked is derived as shown in Figure 7-21 and Figure 7-22, and the generated address is the address of a group of eight PTEs (a PTEG).” |
| 7.6.3, 7-65 | Delete the third from the last paragraph in this section (beginning “Explicitly altering certain MSR bits...”) |
| 8.2, 8-30 | The eighth line of the code sequences should read “CR[(4 * **crf**D) through (4 * **crf**D + 3)] ← c \|\| XER[SO]” instead of “CR[4 * **crf**D–4 * **crf**D + 3] ← c \|\| XER[SO].” |
| 8.2, 8-30 | The fifth line of the code sequences should read “CR[(4 * **crf**D) through (4 * **crf**D + 3)] ← c \|\| XER[SO]” instead of “CR[4 * **crf**D–4 * **crf**D + 3] ← c \|\| XER[SO].” |
| 8.2, 8-30 | Add the following sentence at the end of the first paragraph: “The L bit has no effect on 32-bit operations.” |
| 8.2, 8-33 | The second to last sentence of the **cmpli** PowerPC instruction should read as follows: **cmpldi r**A**,** value equivalent to **cmpli 0,1,r**A**,**value |
| 8.2, 8-132 | The code sequence should read “CR[(4 * **crf**D) through (4 * **crf**D + 3)] ← CR[(4 * **crf**S) through (4 * **crf**S + 3)]” instead of “CR[4 * **crf**D–4 * **crf**D + 3] ← CR[4 * **crf**S–4 * **crf**S + 3].” |
| 8.2, 8-136 | Replace the “low-order bits” with “low-order 32-bits” and “high-order bits” with “high-order 32-bits.” |
| 8.2, 8-169 | The first sentence should read “The contents of **r**S[0–31] are rotated left...” rather than “The contents of **r**S[32-63] are rotated left ...” |
| 8.2, 8-173 | The third line of the instruction description (if **r**B[58] = 0 then) should be removed. |
| 8.2, 8-174 | The second sentence in the second paragraph should read “The setting.. , is independent of 32/64-bit mode” rather than “The setting.. , is independent of mode.” |
| 8.2, 8-175 | The second sentence in the second paragraph should read “ The setting.. , is independent of 32/64-bit mode” rather than “The setting.. , is independent of mode.” |

| Section, Page | Change |
|---|---|
| 8.2, 8-213 | The bit layout of **tlbie**[21–30] should read “306” rather than “30k6.” |
| A.3, A-14 | In Table A-3, replace “**subficx**” with“**subfic**.” |
| A.5, A-41 | In Table A-42, replace instruction “**tlbiax**” with “**tlbia**” and “**tlbiex**” with “**tlbie**.” |
| A.5, A-41 | In the column labelled “Optional” in Table A-42, instruction “**tlbsync**” should be flagged. |

# 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 document, 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.** instruction pair.

## B

**BAT (block address translation) mechanism**. A software-controlled array that stores the available block address translations on-chip.

**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.

**Block**. An area of memory that ranges from 128 Kbyte to 256 Mbyte, whose size, translation, and protection attributes are controlled by the *BAT mechanism.*

**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 **rfi**) 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.

**Direct-store**. Interface available on PowerPC processors only to support direct-store devices from the POWER architecture. When the T bit of a *segment descriptor* is set, the descriptor defines the region of memory that is to be used as a direct-store segment. Note that this facility is being phased out of the architecture and will not likely be supported in future devices. Therefore, software should not depend on it and new software should not use it.

## 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 speculative (out-of-order) execution. When a page is designated as guarded, instructions and data cannot be accessed speculatively.

## 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.** *See* nonspeculative.

**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 segment register and each BAT register. The key bits determine whether supervisor or user programs can access a *page* within that *segment* or *block*.

**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.

## 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 or block address translation mechanisms provided by the MMU and that occur externally with the bus protocol defined for memory.

**Memory coherency**. An aspect of caching in which it is ensured that an accurate view of memory is provided to all devices that share system memory.

**Memory consistency**. Refers to agreement of levels of memory with respect to a single processor and system memory (for example, on-chip cache, secondary cache, and system memory).

**Memory management unit (MMU)**. The functional unit that is capable of translating an *effective* (logical) *address* to a physical address, providing protection mechanisms, and defining caching methods.

**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 double words. Note that munging affects only the effective address and not the byte order. Note also that 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).

**Nonspeculative**. An aspect of an operation that adheres to a sequential model. An operation is said to be nonspeculative if, at the time that it is performed, it is known to be required by the sequential execution model. *See* speculative.

**No-op**. No-operation. A single-cycle operation that does not affect registers or generate bus activity.

**Normalization**. A process by which a floating-point value is manipulated such that it can be represented in the format for the appropriate precision (single- or double-precision). For a floating-point value to be representable in the single- or double-precision format, the leading implied bit must be a 1.

---

## O

**OEA (operating environment architecture)**. The level of the architecture that describes PowerPC memory management model, supervisor-level registers, synchronization requirements, and the 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 memory access.** A memory access performed ahead of one that may have preceded it in the sequential model, such as is allowed by a weakly-ordered memory model. *See* speculative.

**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.

**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 changed bit is set when the page is stored into. *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)**. Data structures containing information used to translate *effective address* to physical address on a 4-Kbyte page basis. A PTE consists of 8 bytes of information in a 32-bit processor and 16 bytes of information in a 64-bit processor.

**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, a BAT area, 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** 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

**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 in 16 on-chip segment registers for 32-bit implementations. For 64-bit implementations, the segment descriptors reside as *segment table entries* in a hashed segment table in memory.

**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.

**Speculative memory access**. An access to memory that occurs before it is known to be required by the sequential execution model.

**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.

# INDEX

# INDEX

# INDEX

# INDEX

# INDEX

# INDEX

# INDEX

# INDEX

# INDEX

# INDEX

# INDEX

# INDEX