PowerPC Operating Environment Architecture

Book III

Version 2.02

January 28, 2005

Manager:

Joe Wetzel/Poughkeepsie/IBM

Technical Content:

Ed Silha/Austin/IBM Cathy May/Watson/IBM Brad Frey/Austin/IBM

Junichi Furukawa/Austin/IBM Giles Frazier/Austin/IBM

The following paragraph does not apply to the United Kingdom or any country or state where such provisions are inconsistent with local law.

The specifications in this manual are subject to change without notice. This manual is provided “AS IS”. International Business Machines Corp. makes no warranty of any kind, either expressed or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose.

International Business Machines Corp. does not warrant that the contents of this publication or the accompanying source code examples, whether individually or as one or more groups, will meet your requirements or that the publication or the accompanying source code examples are error-free.

This publication could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication.

Address comments to IBM Corporation, Internal Zip 9630, 11400 Burnett Road, Austin, Texas 78758-3493. IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

The following terms are trademarks of the International Business Machines Corporation in the United States and/or other countries:

IBM PowerPC RISC/System 6000 POWER
POWER2 POWER4 POWER4+ IBM System/370

Notice to U.S. Government Users&mdash.Documentation Related to Restricted Rights&mdash.Use, duplication or disclosure is subject to restrictions set fourth in GSA ADP Schedule Contract with IBM Corporation.

Preface

This document defines the additional instructions and facilities, beyond those of the PowerPC User Instruction Set Architecture and PowerPC Virtual Environment Architecture, that are provided by the PowerPC Operating Environment Architecture. It covers instructions and facilities not available to the application programmer, affecting storage control, interrupts, and timing facilities.

Other related documents define the PowerPCUser Instruction Set Architecture, the PowerPC Virtual Environment Architecture, and PowerPC Implementation Features. Book I, *PowerPC User Instruction Set Architecture* defines the base instruction set and related facilities available to the application programmer. Book II, *PowerPC Virtual Environment Architecture* defines the storage model and related instructions and facilities available to the application programmer, and the Time Base as seen by the application programmer. Book IV, *PowerPC Implementation Features* defines the implementation-dependent aspects of a particular implementation.

As used in this document, the term “PowerPC Architecture” refers to the instructions and facilities described in Books I, II, and III. The description of the instantiation of the PowerPC Architecture in a given implementation includes also the material in Book IV for that implementation.

Note: Change bars indicate changes relative to Version 2.01.

Table of Contents

Figures

Chapter 1. Introduction

1.1 Overview

Chapter 1 of Book I, *PowerPC User Instruction Set Architecture* describes computation modes, compatibility with the POWER Architecture, document conventions, a general systems overview, instruction formats, and storage addressing. This chapter augments that description as necessary for the PowerPC Operating Environment Architecture.

1.2 Compatibility with the POWER Architecture

The PowerPC Architecture provides binary compatibility for POWER application programs, except as described in the appendix entitled “Incompatibilities with the POWER Architecture” in Book I, *PowerPC User Instruction Set Architecture*. Binary compatibility is not necessarily provided for privileged POWER instructions.

1.3 Document Conventions

The notation and terminology used in Book I apply to this Book also, with the following substitutions.

- For “system alignment error handler” substitute “Alignment interrupt”.
- For “system data storage error handler” substitute “Data Storage interrupt”, “Data Segment interrupt”, or “Data Storage or Data Segment interrupt”, as appropriate.
- For “system error handler” substitute “interrupt”.
- For “system floating-point enabled exception error handler” substitute “Floating-Point Enabled Exception type Program interrupt”.
- For “system illegal instruction error handler” substitute “Illegal Instruction type Program Interrupt”.
- For “system instruction storage error handler” substitute “Instruction Storage interrupt”, “Instruction Segment interrupt”, or “Instruction Storage or Instruction Segment interrupt”, as appropriate.
- For “system privileged instruction error handler” substitute “Privileged Instruction type Program interrupt”.
- For “system service program” substitute “System Call interrupt”, as appropriate.
- For “system trap handler” substitute “Trap type Program interrupt”.

1.3.1 Definitions and Notation

The definitions and notation given in Book I, *PowerPC User Instruction Set Architecture* are augmented by the following.

- A real page is a 4 KB unit of real storage that is aligned at a 4 KB boundary.

- The context of a program is the environment (e.g., privilege and relocation) in which the program executes. That context is controlled by the contents of certain System Registers, such as the MSR and SDR1, of certain lookaside buffers, such as the SLB and TLB, and of the Page Table.
- An exception is an error, unusual condition, or external signal, that may set a status bit and may or may not cause an interrupt, depending upon whether the corresponding interrupt is enabled.
- An interrupt is the act of changing the machine state in response to an exception, as described in Chapter 5. “Interrupts” on page 61.
- A trap interrupt is an interrupt that results from execution of a *Trap* instruction.
- Additional exceptions to the rule that the processor obeys the sequential execution model, beyond those described in the section entitled “Instruction Fetching” in Book I, are the following.
  - A System Reset or Machine Check interrupt may occur. The determination of whether an instruction is required by the sequential execution model is not affected by the potential occurrence of a System Reset or Machine Check interrupt. (The determination is affected by the potential occurrence of any other kind of interrupt.)
  - A context-altering instruction is executed (Chapter 7. “Synchronization Requirements for Context Alterations” on page 85). The context alteration need not take effect until the required subsequent synchronizing operation has occurred.
  - A Reference and Change bit is updated by the processor. The update need not be performed with respect to that processor until the required subsequent synchronizing operation has occurred.
- Hardware means any combination of hard-wired implementation, emulation assist, or interrupt for software assistance. In the last case, the interrupt may be to an architected location or to an implementation-dependent location. Any use of emulation assists or interrupts to implement the architecture is described in Book IV, *PowerPC Implementation Features*.
- /, //, ///, ... denotes a field that is reserved in an instruction, in a register, or in an architected storage table.

1.3.2 Reserved Fields

Some fields of certain architected storage tables may be written to automatically by the processor, e.g., Reference and Change bits in the Page Table. When the processor writes to such a table, the following rules are obeyed.

- Unless otherwise stated, no defined field other than the one(s) the processor is specifically updating are modified.
- Contents of reserved fields are either preserved by the processor or written as zero.

The handling of reserved bits in System Registers described in Book I applies here as well. The reader should be aware that reading and writing of some of these registers (e.g., the MSR) can occur as a side effect of processing an interrupt and of returning from an interrupt, as well as when requested explicitly by the appropriate instruction (e.g., ***mtmsrd***).

Programming Note

Software should set reserved fields in architected storage tables (e.g., the Page Table) to zero, because these fields may be assigned a meaning in some future version of the architecture.

1.4 General Systems Overview

The processor or processor unit contains the sequencing and processing controls for instruction fetch, instruction execution, and interrupt action. Instructions that the processing unit can execute fall into three classes:

- instructions executed in the Branch Processor
- instructions executed in the Fixed-Point Processor
- instructions executed in the Floating-Point Processor

Almost all instructions executed in the Branch Processor, Fixed-Point Processor, and Floating-Point Processor are nonprivileged and are described in Book I, *PowerPC User Instruction Set Architecture*. Book II, *PowerPC Virtual Environment Architecture* may describe additional nonprivileged instructions (e.g., Book II describes some nonprivileged instructions for cache management). Instructions related to the privileged state of the processor, control of processor resources, control of the storage hierarchy, and all other privileged instructions are described here or in Book IV, *PowerPC Implementation Features*.

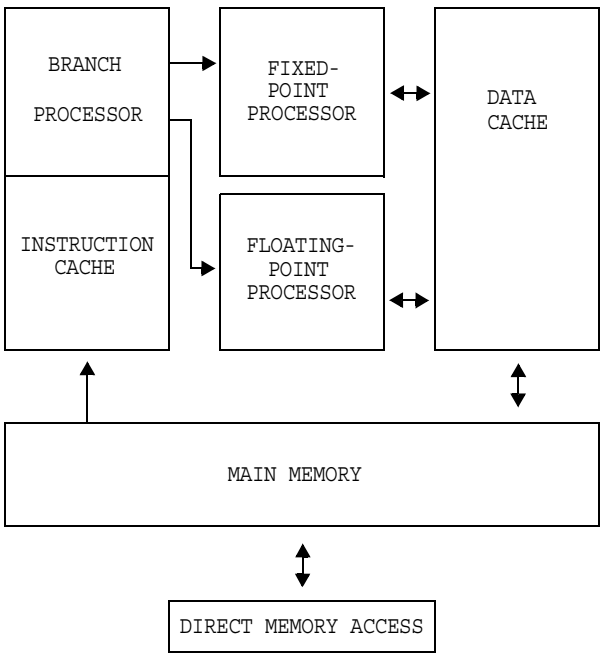


Figure 1. Logical view of the PowerPC processor architecture

1.5 Exceptions

The following augments the list, given in Book I, of exceptions that can be caused directly by the execution of an instruction:

- the execution of a floating-point instruction when MSR_{FP}=0 (Floating-Point Unavailable interrupt)
- an attempt to modify a hypervisor resource when the processor is in privileged but non-hypervisor state (see Section 1.7), or an attempt to execute a hypervisor-only instruction (e.g., ***tlbie***) when the processor is in privileged but non-hypervisor state.
- the execution of a traced instruction (Trace interrupt)

1.6 Synchronization

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

1.6.1 Context Synchronization

An instruction or event is *context synchronizing* if it satisfies the requirements listed below. Such instructions and events are collectively called *context synchronizing operations*. Examples of context synchronizing operations include the ***isync***, ***sc***, and ***rfid*** instructions, the ***mtmsr***[***d***] instruction if L=0, and most interrupts.

1. The operation causes instruction dispatching (the issuance of instructions by the instruction fetch mechanism to any instruction execution mechanism) to be halted.
2. The operation is not initiated or, in the case of ***isync***, does not complete, until all instructions already in execution have completed to a point at which they have reported all exceptions they will cause.
3. The operation ensures that the instructions that precede the operation will complete execution in the context (privilege, relocation, storage protection, etc.) in which they were initiated, except that the operation has no effect on the context in which the associated Reference and Change bit updates are performed.
4. If the operation directly causes an interrupt (e.g., ***sc*** directly causes a System Call interrupt) or is an interrupt, the operation is not initiated until no exception exists having higher priority than the exception associated with the interrupt (see Section 5.8, “Interrupt Priorities” on page 76).
5. The operation ensures that the instructions that follow the operation will be fetched and executed in the context established by the operation. (This requirement dictates that any prefetched instructions be discarded and that any effects and side effects of executing them out-of-order also be discarded, except as described inSection 4.2.4, “Performing Operations Out-of-Order” on page 27.)

Programming Note

A context synchronizing operation is necessarily execution synchronizing; see Section 1.6.2.

Unlike the Synchronize instruction, a context synchronizing operation does not affect the order in which storage accesses are performed.

1.6.2 Execution Synchronization

An instruction is *execution synchronizing* if it satisfies items 2 and 3 of the definition of context synchronization (see Section 1.6.1). ***sync*** and ***ptesync*** are treated like ***isync*** with respect to item 2 (i.e., the conditions described in item 2 apply to the completion of ***sync*** and ***ptesync***). Examples of execution synchronizing instructions are ***sync***, ***ptesync***, and ***mtmsrd***.

Programming Note

All context synchronizing instructions are execution synchronizing.

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

1.7 Logical Partitioning (LPAR)

The Logical Partitioning (LPAR) facility permits processors and portions of real storage to be assigned to logical collections called *partitions*, such that a program executing on a processor in one partition cannot interfere with any program executing on a processor in a different partition. This isolation can be provided for both problem state and privileged state programs, by using a layer of trusted software, called a *hypervisor* program (or simply a "hypervisor"), and the resources provided by this facility to manage system resources. (A hypervisor is a program that runs in hypervisor state; see below.)

The number of partitions supported is implementation-dependent.

A processor is assigned to one partition at any given time. A processor can be assigned to any given partition without consideration of the physical configuration of the system (e.g., shared registers, caches, organization of the storage hierarchy), except that processors that share certain hypervisor resources may need to be assigned to the same partition; see Section 1.7.5. The registers and facilities used to control Logical Partitioning are listed below and described in the following subsections.

Except in the following subsections, references to the "operating system" in Books I - III include the hypervisor unless otherwise stated or obvious from context.

1.7.1 Logical Partitioning Control Register (LPCR)

The layout of the Logical Partitioning Control Register (LPCR) is shown in Figure 2 below.

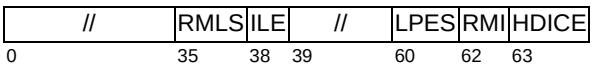


Figure 2. Logical Partitioning Control Register

The contents of the LPCR control a number of aspects of the operation of the processor with respect to a logical partition. Below are shown the bit definitions for the LPCR.

Bit **Description**

0:34 Reserved

35:37 ***Real Mode Offset Selector*** (RMLS)

The RMLS field specifies the largest effective address that can be used by partition software when address translation is disabled. The valid RMLS values and their meaning are implementation-dependent powers of 2, 2^j, where $12 \leq j \leq m$.

38 ***Interrupt Little-Endian*** (ILE)

This bit is part of the optional Little-Endian facility; see the section entitled "Little-Endian" in Book I.

If the Little-Endian facility is implemented, the content of ILE is copied into MSR_{LE} on some interrupts (only those that set MSR_{HV} to 0; see Section 5.5 on page 65) to establish the Endian environment for the interrupt handling

If the Little-Endian facility is not implemented, this bit is treated as reserved.

39:59 Reserved

60:61 ***Logical Partitioning Environment Selector*** (LPES)

Three of the four LPES values are supported. The 0b10 value is reserved.

60 $LPES_0$

Controls whether External interrupts set MSR_{HV} to 1 or leave it unchanged.

61 $LPES_1$

Controls how storage is accessed when address translation is disabled, and whether a subset of interrupts set MSR_{HV} to 1.

Programming Note

$LPES_1$=0 provides an environment in which only the hypervisor can run with address translation disabled and in which all interrupts invoke the hypervisor. This value (along with MSR_{HV}=1) can also be used in a system that is not partitioned, to permit the operating system to access all system resources.

Setting $LPES_{0:1}$ to 0b00 can be used to configure LPAR environments similar to the environment selected by the LPES bit on the POWER4 processor.

62 ***Real Mode Caching Inhibited Bit*** (RMI)

The RMI bit affects the manner in which storage accesses are performed in hypervisor mode when address translation is disabled (see Section 4.2.6.2 on page 30).

Programming Note

Because in real addressing mode all storage is not Caching Inhibited (unless the Real Mode Caching Inhibited bit is 1), software should not map a Caching Inhibited virtual page to storage that is treated as non-Guarded in real addressing mode. Doing so could permit storage locations in the virtual page to be copied into the cache, which could lead to violations of the requirement given in Section 4.8.2 for changing the value of the I bit. See also Section 8.2 on page 90.

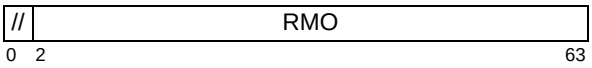
63 ***Hypervisor Decrementer Interrupt Conditionally Enable*** (HDICE)

- **0** Hypervisor Decrementer interrupts are disabled.
- **1** Hypervisor Decrementer interrupts are enabled if permitted by MSR_{EE}, MSR_{HV}, and MSR_{PR}; see Section 5.5.12 on page 72.

See Section 4.2.6 on page 29 (including subsections) and Section 4.10 on page 45 for a description of how storage accesses are affected by the setting of $LPES_1$, RMLS, and RMI. See Section 5.5 on page 65 for a description of how the setting of $LPES_{0:1}$ affects the processing of interrupts.

1.7.2 Real Mode Offset Register (RMOR)

The layout of the Real Mode Offset Register (RMOR) is shown in Figure 3 below.



Bits	Name	Description
2:63	RMO	Real Mode Offset

All other fields are reserved.

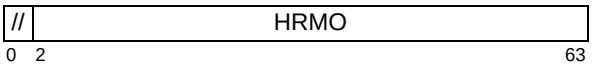
The supported RMO values are the non-negative multiples of 2^S, where 2^S is the smallest implementation-dependent limit value representable by the contents of the Real Mode Limit Selector field of the LPCR.

Figure 3. Real Mode Offset Register

The contents of the RMOR affect how some storage accesses are performed as described in Section 4.2.6 on page 29 and Section 4.2.7 on page 31.

1.7.3 Hypervisor Real Mode Offset Register (HRMOR)

The layout of the Hypervisor Real Mode Offset Register (HRMOR) is shown in Figure 4 below.



Bits	Name	Description
2:63	HRMO	Hypervisor Real Mode Offset

All other fields are reserved.

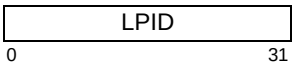
The supported HRMO values are the non-negative multiples of 2^r, where r is an implementation-dependent value and $12 \leq r \leq 26$.

Figure 4. Hypervisor Real Mode Offset Register

The contents of the HRMOR affect how some storage accesses are performed as described in Section 4.2.6 on page 29 and Section 4.2.7 on page 31.

1.7.4 Logical Partition Identification Register (LPIDR)

The layout of the Logical Partition Identification Register (LPIDR) is shown in Figure 5 below.



Bits	Name	Description
0:31	LPID	Logical Partition Identifier

Figure 5. Logical Partition Identification Register

The contents of the LPIDR identify the partition to which the processor is assigned, affecting operations necessary to manage the coherency of some translation lookaside buffers (see Section 4.12.1 on page 57 and Chapter 7 on page 85).

The supported LPID values consist of all non-negative values that are less than an implementation-dependent power of 2, 2^q, where $2^q \geq$ (the maximum number of processors in a system) $\times$ 2.

Programming Note

On some implementations, software must prevent the execution of a ***tlbie*** instruction on any processor for which the contents of the LPIDR is the same as on the processor on which the LPIDR is being modified or is the same as the new value being written to the LPIDR. This restriction can be met with less effort if one partition identity is used only on processors on which no ***tlbie*** instruction is ever executed. This partition can be thought of as the transfer partition used exclusively to move a processor from one partition to another.

1.7.5 Other Hypervisor Resources

In addition to the resources described above, the following resources are hypervisor resources, accessible to software only when the processor is in hypervisor mode.

- All implementation-specific resources, including implementation-specific registers (e.g., “HID” registers), that control hardware functions or affect the results of instruction execution. Examples include resources that disable caches, disable hardware error detection, set breakpoints, control power management, or significantly affect performance.
- ME bit of the MSR
- SDR1, EAR (if implemented), Time Base, PIR, Hypervisor Decrementer, DABR, DABRX, and PURR. (Note: Although the Time Base and the PURR can be altered only by a hypervisor program, the Time Base can be read by all programs and the PURR can be read when the processor is in privileged state.)

The contents of a hypervisor resource can be modified by the execution of an instruction (e.g., ***mtspr***) only in hypervisor state ($MSR_{HV\ PR}$ = 0b10). Whether an attempt to modify the contents of a given hypervisor resource, other than MSR_{ME}, in privileged but non-hypervisor state ($MSR_{HV\ PR}$ = 0b00) is ignored (i.e., treated as a no-op) or causes a Privileged Instruction type Program interrupt is implementation-dependent. An attempt to modify MSR_{ME} in privileged but non-hypervisor state is ignored (i.e., the bit is not changed).

The ***tlbie***, ***tlbiel***, ***tlbia***, and ***tlbsync*** instructions can be executed only in hypervisor state; see the descriptions of these instructions on pages 53 and 56.

In general, if software violates a rule that is stated in the Books using the word “must” (e.g., “this field must be set to 0”) the results are boundedly undefined. The only exception is that if hypervisor software violates such a rule that pertains to the contents of a hypervisor resource, to accessing storage in real addressing mode, or to using the ***tlbie*** and ***tlbsync*** instructions, the results are undefined, and may include altering resources belonging to other partitions, causing the system to “hang”, etc.

Programming Note

Because the SPRs listed above are privileged for writing, an attempt to modify the contents of any of these SPRs in problem state (MSR_{PR}=1) using ***mtspr*** causes a Privileged Instruction type Program exception, and similarly for MSR_{ME}.

1.7.6 Sharing Hypervisor Resources

Some hypervisor resources may be shared among processors. Programs that modify these resources must be aware of this sharing, and must allow for the fact that changes to these resources may affect more than one processor. The following resources may be shared among processors.

- RMOR (see Section 1.7.2.)
- HRMOR (see Section 1.7.3.)
- LPIDR (see Section 1.7.4.)
- PIR (see Section 3.3.6.)
- SDR1 (see Section 4.5.2.)
- Time Base (see Section 6.2.)
- Hypervisor Decrementer (see Section 6.4.)
- certain implementation-specific registers

See Book IV for information about the set of resources that are shared for a given implementation.

Processors that share any of the resources listed above, with the exception of the PIR and the HRMOR, must be in the same partition.

For each field of the LPCR except the RMI field and the HDICE field, software must ensure that the contents of the field are identical among all processors that are in the same partition and are in a state such that the contents of the field could have side effects. (E.g., software must ensure that the contents of $LPCR_{LPES}$ are identical among all processors that are in the same partition and are not in hypervisor state.) For the HDICE field, software must ensure that the contents of the field are identical among all processors that share the Hypervisor Decrementer and are in a state such that the contents of the field could have side effects. (There are no identity requirements for the RMI field.)

Chapter 2. Branch Processor

2.1 Branch Processor Overview

This chapter describes the details concerning the registers and the privileged instructions implemented in the Branch Processor that are not covered in Book I, *PowerPC User Instruction Set Architecture*.

2.2 Branch Processor Registers

2.2.1 Machine Status Save/ Restore Registers

When an interrupt occurs, the state of the machine is saved in the Machine Status Save/Restore registers (SRR0 and SRR1) unless the interrupt is an interrupt that sets HSRR0 and HSRR1. The contents of these registers is used to restore machine state when an ***rfid*** instruction is executed.

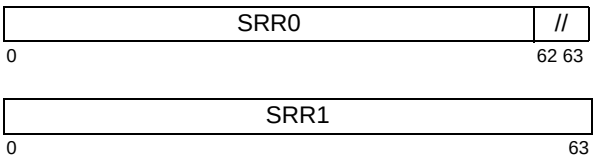


Figure 6. Save/Restore Registers

2.2.2 Hypervisor Machine Status Save/Restore Registers

When a Hypervisor Decrementer interrupt occurs, the state of the machine is saved in the Hypervisor Machine Status Save/Restore registers (HSRR0 and HSRR1). The contents of these registers is used to restore machine state when an ***hrfid*** instruction is executed.

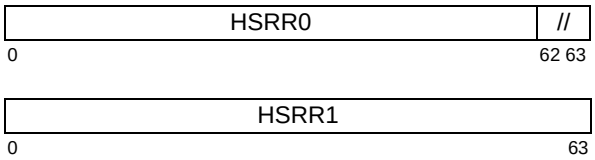


Figure 7. Hypervisor Save/Restore Registers

2.2.3 Machine State Register

The Machine State Register (MSR) is a 64-bit register. This register defines the state of the processor. On interrupt, the MSR bits are altered in accordance with Figure 35 on page 65. The MSR can also be modified by the ***mtmsr***[***d***], ***rfid***, and ***hrfid*** instructions. It can be read by the ***mfmsr*** instruction.

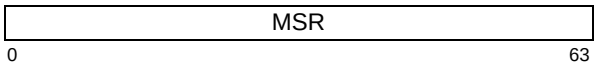


Figure 8. Machine State Register

Below are shown the bit definitions for the Machine State Register.

Bit **Description**

0 ***Sixty-Four-Bit Mode*** (SF)

0 The processor is in 32-bit mode.
1 The processor is in 64-bit mode.

1:2 Reserved

3 ***Hypervisor State*** (HV)

0 The processor is not in hypervisor state.
1 If MSR_{PR}=0 the processor is in hypervisor state; otherwise the processor is not in hypervisor state.

Programming Note

The privilege state of the processor is determined by MSR_{HV} and MSR_{PR}, as follows.

HV	PR	
0	0	privileged
0	1	problem
1	0	privileged and hypervisor
1	1	problem

MSR_{HV} can be set to 1 only by the *System Call* instruction and some interrupts. It can be set to 0 only by the ***rfid*** and ***hrfid*** instructions.

4:46 Reserved

47 Reserved

48 ***External Interrupt Enable*** (EE)

0 External and Decrementer interrupts are disabled.
1 External and Decrementer interrupts are enabled.

This bit also affects whether Hypervisor Decrementer interrupts are enabled; Section 5.5.12 on page 72.

49 ***Problem State*** (PR)

0 The processor is in privileged state.
1 The processor is in problem state.

Programming Note

Any instruction that sets MSR_{PR} to 1 also sets MSR_{EE}, MSR_{IR}, and MSR_{DR} to 1.

50 ***Floating-Point Available*** (FP)

0 The processor cannot execute any floating-point instructions, including floating-point loads, stores, and moves.
1 The processor can execute floating-point instructions.

51 ***Machine Check Interrupt Enable*** (ME)

0 Machine Check interrupts are disabled.
1 Machine Check interrupts are enabled.

This bit is a hypervisor resource; see Section 1.7, “Logical Partitioning (LPAR)” on page 5.

Programming Note

The only instructions that can alter MSR_{ME} are the ***rfid*** and ***hrfid*** instructions.

52 Floating-Point Exception Mode 0 (FE0)

See below.

53 ***Single-Step Trace Enable*** (SE)

0 The processor executes instructions normally.
1 The processor generates a Single-Step type Trace interrupt after successfully completing the execution of the next instruction, unless that instruction is ***hrfid*** or ***rfid***, which are never traced. Successful completion means that the instruction caused no other interrupt.

54 ***Branch Trace Enable*** (BE)

0 The processor executes branch instructions normally.
1 The processor generates a Branch type Trace interrupt after completing the execution of a branch instruction, whether or not the branch is taken. See Book IV, *PowerPC Implementation Features*.

Branch tracing may not be present on all implementations. If the function is not implemented, this bit is treated as reserved.

55 ***Floating-Point Exception Mode 1*** (FE1)

See below.

56:57 Reserved

58 ***Instruction Relocate*** (IR)

0 Instruction address translation is disabled.

1 Instruction address translation is enabled.

Programming Note

See the Programming Note in the definition of MSR_{PR}.

59 ***Data Relocate*** (DR)

0 Data address translation is disabled. Effective Address Overflow (EAO) (see Book I) does not occur.
1 Data address translation is enabled. EAO causes a Data Storage interrupt.

Programming Note

See the Programming Note in the definition of MSR_{PR}.

60 Reserved

61 ***Performance Monitor Mark*** (PMM)

This bit is part of the optional Performance Monitor facility; see Appendix E. If the Performance Monitor facility is not implemented or does not use this bit, this bit is treated as reserved.

62 ***Recoverable Interrupt*** (RI)

0 Interrupt is not recoverable.
1 Interrupt is recoverable.

Additional information about the use of this bit is given in Sections 5.4, “Interrupt Processing” on page 62, 5.5.1, “System Reset Interrupt” on page 66, and 5.5.2, “Machine Check Interrupt” on page 66.

63 ***Little-Endian Mode*** (LE)

This bit is part of the optional Little-Endian facility; see the section entitled “Little-Endian” in Book I.

If the Little-Endian facility is implemented, this bit has the following meaning.

0 The processor is in Big-Endian mode.
1 The processor is in Little-Endian mode.

If the Little-Endian facility is not implemented, this bit is treated as reserved.

The Floating-Point Exception Mode bits FE0 and FE1 are interpreted as shown below. For further details see Book I, *PowerPC User Instruction Set Architecture*.

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

2.3 Branch Processor Instructions

2.3.1 System Linkage Instructions

These instructions provide the means by which a program can call upon the system to perform a service, and by which the system can return from performing a service or from processing an interrupt.

The *System Call* instruction is described in Book I, *PowerPC User Instruction Set Architecture*, but only at the level required by an application programmer. A complete description of this instruction appears below.

System Call SC-form

sc LEV

[POWER mnemonic: svca]

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

```
SRR0 ←iea CIA + 4
SRR1 33:36 42:47 ← 0
SRR1 0:32 37:41 48:63 ← MSR 0:32 37:41 48:63
MSR ← new_value (see below)
NIA ← 0x0000_0000_0000_0C00
```

The effective address of the instruction following the *System Call* instruction is placed into SRR0. Bits 0:32, 37:41, and 48:63 of the MSR are placed into the corresponding bits of SRR1, and bits 33:36 and 42:47 of SRR1 are set to zero.

Then a System Call interrupt is generated. The interrupt causes the MSR to be set as described in Section 5.5, “Interrupt Definitions” on page 65. The setting of the MSR is affected by the contents of the LEV field. LEV values greater than 1 are reserved. Bits 0:5 of the LEV field (instruction bits 20:25) are treated as a reserved field.

The interrupt causes the next instruction to be fetched from effective address 0x0000_0000_0000_0C00.

This instruction is context synchronizing.

Special Registers Altered:

SRR0 SRR1 MSR

Programming Note

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

Programming Note

If LEV=1 the hypervisor is invoked.

If $LPES_1$=1, executing this instruction with LEV=1 is the only way that executing an instruction can cause hypervisor state to be entered.

Because this instruction is not privileged, it is possible for application software to invoke the hypervisor. However, such invocation should be considered a programming error.

Return From Interrupt Doubleword XL-form

rfid

19	///	///	///	18	/
0	6	11	16	21	31

$MSR_0 \leftarrow SRR1_0 \mid SRR1_1$
$MSR_{51} \leftarrow (MSR_3 \ \&\ SRR1_{51}) \mid ((\neg MSR_3) \ \&\ MSR_{51})$
$MSR_3 \leftarrow MSR_3 \ \&\ SRR1_3$

$MSR_{48} \leftarrow SRR1_{48} \mid SRR1_{49}$
$MSR_{58} \leftarrow SRR1_{58} \mid SRR1_{49}$
$MSR_{59} \leftarrow SRR1_{59} \mid SRR1_{49}$
$MSR_{1:2\ 4:32\ 37:41\ 49:50\ 52:57\ 60:63} \leftarrow SRR1_{1:2\ 4:32\ 37:41\ 49:50\ 52:57\ 60:63}$
$NIA \leftarrow_{iea} SRR0_{0:61} \mid\mid 0b00$

The result of ORing bits 0 and 1 of SRR1 is placed into MSR_0. If MSR_3=1 then bits 3 and 51 of SRR1 are placed into the corresponding bits of the MSR. The result of ORing bits 48 and 49 of SRR1 is placed into MSR_{48}. The result of ORing bits 58 and 49 of SRR1 is placed into MSR_{58}. The result of ORing bits 59 and 49 of SRR1 is placed into MSR_{59}. Bits 1:2, 4:32, 37:41, 49:50, 52:57, and 60:63 of SRR1 are placed in the corresponding bits of the MSR.

If the new MSR value does not enable any pending exceptions, then the next instruction is fetched, under control of the new MSR value, from the address $SRR0_{0:61}$ || 0b00 (when SF=1 in the new MSR value) or $^{32}0$ || $SRR0_{32:61}$ || 0b00 (when SF=0 in the new MSR value). If the new MSR value enables one or more pending exceptions, the interrupt associated with the highest priority pending exception is generated; in this case the value placed into SRR0 or HSRR0 by the interrupt processing mechanism (see Section 5.4, “Interrupt Processing” on page 62) is the address of the instruction that would have been executed next had the interrupt not occurred.

This instruction is privileged and context synchronizing.

Special Registers Altered:

MSR

Programming Note

If this instruction sets MSR_{PR} to 1, it also sets MSR_{EE}, MSR_{IR}, and MSR_{DR} to 1.

On processors prior to POWER4+, setting MSR_{PR} to 1 did not cause MSR_{EE} to be set to 1.

Hypervisor Return From Interrupt Doubleword XL-form

hrfid

19	///	///	///	274	/
0	6	11	16	21	31

$MSR_0 \leftarrow HSRR1_0 \mid HSRR1_1$
$MSR_{48} \leftarrow HSRR1_{48} \mid HSRR1_{49}$
$MSR_{58} \leftarrow HSRR1_{58} \mid HSRR1_{49}$
$MSR_{59} \leftarrow HSRR1_{59} \mid HSRR1_{49}$
$MSR_{1:32\ 37:41\ 49:57\ 60:63} \leftarrow HSRR1_{1:32\ 37:41\ 49:57\ 60:63}$
$NIA \leftarrow HSRR0_{0:61} \mid\mid 0b00$

The result of ORing bits 0 and 1 of HSRR1 is placed into MSR_0. The result of ORing bits 48 and 49 of HSRR1 is placed into MSR_{48}. The result of ORing bits 58 and 49 of HSRR1 is placed into MSR_{58}. The result of ORing bits 59 and 49 of HSRR1 is placed into MSR_{59}. Bits 1:32, 37:41, 49:57, and 60:63 of HSRR1 are placed into the corresponding bits of the MSR.

If the new MSR value does not enable any pending exceptions, then the next instruction is fetched, under control of the new MSR value, from the address $HSRR0_{0:61}$ || 0b00 (when SF=1 in the new MSR value) or $^{32}0$ || $HSRR0_{32:61}$ || 0b00 (when SF=0 in the new MSR value). If the new MSR value enables one or more pending exceptions, the interrupt associated with the highest priority pending exception is generated; in this case the value placed into SRR0 or HSRR0 by the interrupt processing mechanism (see Section 5.4, “Interrupt Processing” on page 62) is the address of the instruction that would have been executed next had the interrupt not occurred.

This instruction is privileged and context synchronizing, and can be executed only in hypervisor state. If it is executed in privileged but non-hypervisor state either a Privileged Instruction type Program interrupt occurs or the results are boundedly undefined.

Special Registers Altered:

MSR

Programming Note

If this instruction sets MSR_{PR} to 1, it also sets MSR_{EE}, MSR_{IR}, and MSR_{DR} to 1.

Chapter 3. Fixed-Point Processor

3.1 Fixed-Point Processor Overview

This chapter describes the details concerning the registers and the privileged instructions implemented in the Fixed-Point Processor that are not covered in Book I, *PowerPC User Instruction Set Architecture*.

3.2 Special Purpose Registers

Special Purpose Registers (SPRs) are read and written using the ***mfspr*** (page 21) and ***mtspr*** (page 19) instructions. Most SPRs are defined in other chapters of this book; see the index to locate those definitions.

3.3 Fixed-Point Processor Registers

3.3.1 Data Address Register

The Data Address Register (DAR) is a 64-bit register that is set by the Machine Check, Data Storage, Data Segment, and Alignment interrupts; see Sections 5.5.2, 5.5.3, 5.5.4, and 5.5.8. In general, when one of these interrupts occurs the DAR is set to an effective address associated with the storage access that caused the interrupt, with the high-order 32 bits of the DAR set to 0 if the interrupt occurs in 32-bit mode.

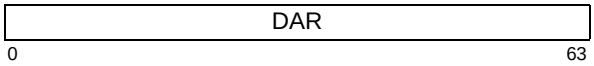


Figure 9. Data Address Register

3.3.2 Data Storage Interrupt Status Register

The Data Storage Interrupt Status Register (DSISR) is a 32-bit register that is set by the Machine Check, Data Storage, Data Segment, and Alignment interrupts; see Sections 5.5.2, 5.5.3, 5.5.4, and 5.5.8. In general, when one of these interrupts occurs the DSISR is set to indicate the cause of the interrupt.

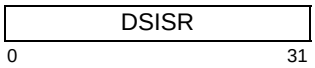


Figure 10. Data Storage Interrupt Status Register

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

3.3.3 Software-use SPRs

SPRG0 through SPRG3 are 64-bit registers provided for use by privileged software.

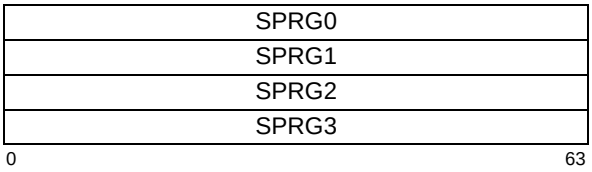


Figure 11. SPRs for use by privileged non-hypervisor programs

SPRG0, SPRG1, and SPRG2 are privileged registers. SPRG3 is a privileged register except that the contents may be copied to a GPR in Problem state when accessed using the ***mfspr*** instruction.

Programming Note

Neither the contents of the SPRGs, nor accessing them using ***mtspr*** or ***mfspr***, has a side effect on the operation of the processor. One or more of the registers is likely to be needed by non-hypervisor interrupt handler programs (e.g., as scratch registers and/or pointers to per processor save areas).

Operating systems must ensure that no sensitive data are left in SPRG3 when a problem state program is dispatched, and operating systems for secure systems must ensure that SPRG3 cannot be used to implement a “covert channel” between problem state programs. These requirements can be satisfied by clearing SPRG3 before passing control to a program that will run in problem state.

In implementations prior to POWER5 that provided LPAR facilities, SPRG0 was treated as a privileged SPR except that the execution of a ***mtspr*** instruction would alter the contents only if $MSR_{HV} = 1$.

HSPRG0 and HSPRG1 are 64-bit registers provided for use by hypervisor programs.

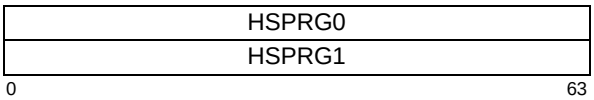


Figure 12. SPRs for use by hypervisor programs

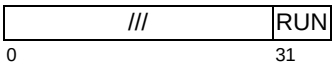
Programming Note

Neither the contents of the HSPRGs, nor accessing them using ***mtspr*** or ***mfspr***, has a side effect on the operation of the processor. One or more of the registers is likely to be needed by hypervisor interrupt handler programs (e.g., as scratch registers and/or pointers to per processor save areas).

3.3.4 Control Register

The Control Register (CTRL) is a 32-bit register that controls an external I/O pin. This signal may be used for the following:

- driving the RUN Light on a system operator panel
- External interrupt routing
- Performance Monitor Counter incrementing (see Appendix E, “Example Performance Monitor (Optional)” on page 107)



Bit	Name	Description
31	RUN	Run state bit

All other fields are implementation-dependent.

Figure 13. Control Register

The CTRL RUN can be used by the operating system to indicate when the processor is doing useful work.

The contents of the CTRL can be written by the ***mtspr*** instruction and read by the ***mfspr*** instruction. Write access to the CTRL is privileged. Reads can be performed in privileged or problem state.

3.3.5 Processor Version Register

The Processor Version Register (PVR) is a 32-bit read-only register that contains a value identifying the version and revision level of the processor. The contents of the PVR can be copied to a GPR by the ***mfspr*** instruction. Read access to the PVR is privileged; write access is not provided.

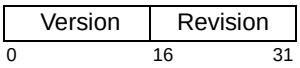


Figure 14. Processor Version Register

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

Version A 16-bit number that identifies the version of the processor. Different version numbers indicate major differences between processors, such as which optional facilities and instructions are supported.

Revision A 16-bit number that distinguishes between implementations of the version. Different revision numbers indicate minor differences between processors having the same version number, such as clock rate and Engineering Change level.

Version numbers are assigned by the PowerPC Architecture process. Revision numbers are assigned by an implementation-defined process.

3.3.6 Processor Identification Register

The Processor Identification Register (PIR) is a 32-bit register that contains a value that can be used to distinguish the processor from other processors in the system. The contents of the PIR can be copied to a GPR by the ***mfspr*** instruction. Read access to the PIR is privileged; write access, if provided, is described in the Book IV, *PowerPC Implementation Features* document for the implementation.

Bits	Name	Description
0:31	PROCID	Processor ID

Figure 15. Processor Identification Register

The means by which the PIR is initialized are implementation-dependent (see Book IV).

The PIR is a hypervisor resource; see Section 1.7, “Logical Partitioning (LPAR)” on page 5.

3.4 Fixed-Point Processor Instructions

3.4.1 OR Instruction

Some forms of the *OR* instruction in which RS = RA = RB provide a hint to the processor regarding the priority level of the program. The supported encodings are shown in Figure 16. No hint is provided if the privilege state of the processor executing the instruction is lower than the privilege indicated in the figure.

Rx	Priority	Privileged
31	very low	yes
1	low	no
6	medium low	no
2	medium (normal)	no
5	medium high	yes
3	high	yes
7	very high	hypv

Figure 16. Priority hint levels for *or* Rx,Rx,Rx

3.4.2 Move To/From System Register Instructions

The *Move To Special Purpose Register* and *Move From Special Purpose Register* instructions are described in Book I, *PowerPC User Instruction Set Architecture*, but only at the level available to an application programmer. For example, no mention is made there of registers that can be accessed only in privileged state. The descriptions of these instructions given below extend the descriptions given in Book I, but do not list Special Purpose Registers that are defined in Book IV, *PowerPC Implementation Features*. In the descriptions of these instructions given below, the "defined" SPR numbers are the SPR numbers shown in the figure for the instruction and the SPR numbers defined in Book IV for the instruction, and similarly for "defined" registers.

Extended mnemonics

Extended mnemonics are provided for the ***mtspr*** and ***mfspr*** instructions so that they can be coded with the SPR name as part of the mnemonic rather than as a numeric operand. See Appendix A, "Assembler Extended Mnemonics" on page 97.

Move To Special Purpose Register XFX-form

mtspr SPR,RS

31	RS	spr	467	/
0	6	11	21	31

```
n ← spr5:9 || spr0:4
if length(SPREG(n)) = 64 then
  SPREG(n) ← (RS)
else
  SPREG(n) ← (RS)32:63
```

The SPR field denotes a Special Purpose Register, encoded as shown in Figure 17. The contents of register RS are placed into the designated Special Purpose Register. For Special Purpose Registers that are 32 bits long, the low-order 32 bits of RS are placed into the SPR.

For this instruction, SPRs TBL and TBU are treated as separate 32-bit registers; setting one leaves the other unaltered.

spr_0=1 if and only if writing the register is privileged. Execution of this instruction specifying a defined and privileged register when MSR_{PR}=1 causes a Privileged Instruction type Program interrupt. Execution of this instruction specifying a hypervisor resource when $MSR_{HV\ PR}$ = 0b00 either has no effect or causes a Privileged Instruction type Program interrupt (see Section 1.7 on page 5).

Execution of this instruction specifying an SPR number that is not defined for the implementation causes either an Illegal Instruction type Program interrupt or one of the following.

- if spr_0=0: boundedly undefined results
- if spr_0=1:
  - if MSR_{PR}=1: Privileged Instruction type Program interrupt
  - if MSR_{PR}=0 and MSR_{HV}=0: boundedly undefined results
  - if MSR_{PR}=0 and MSR_{HV}=1: undefined results

If the SPR field contains a value that is shown in Figure 17but corresponds to an optional Special Purpose Register that is not provided by the implementation, the effect of executing this instruction is the same as if the SPR number were not shown in the figure.

Special Registers Altered:

See Figure 17.

decimal	SPR[1] $spr_{5:9}$ $spr_{0:4}$	Register Name	Privileged
1	00000 00001	XER	no
8	00000 01000	LR	no
9	00000 01001	CTR	no
18	00000 10010	DSISR	yes
19	00000 10011	DAR	yes
22	00000 10110	DEC	yes
25	00000 11001	SDR1 [5]	hypv
26	00000 11010	SRR0	yes
27	00000 11011	SRR1	yes
29	00000 11101	ACCR	yes
152	00100 11000	CTRL	yes
272	01000 10000	SPRG0	yes
273	01000 10001	SPRG1	yes
274	01000 10010	SPRG2	yes
275	01000 10011	SPRG3	yes
282	01000 11010	EAR [2,5]	hypv
284	01000 11100	TBL [5]	hypv
285	01000 11101	TBU [5]	hypv
304	01001 10000	HSPRG0 [5]	hypv
305	01001 10001	HSPRG1 [5]	hypv
309	01001 10101	PURR [5]	hypv
310	01001 10110	HDEC [5]	hypv
312	01001 11000	RMOR [5]	hypv
313	01001 11001	HRMOR [5]	hypv
314	01001 11010	HSRR0 [5]	hypv
315	01001 11011	HSRR1 [5]	hypv
318	01001 11110	LPCR [5]	hypv
319	01001 11111	LPIDR [5]	hypv
784-799	11000 1xxxx	perf_mon[3]	yes
1012	11111 10101	DABR [4,5]	hypv
1015	11111 10111	DABRX [4,5]	hypv

[1] Note that the order of the two 5-bit halves of the SPR number is reversed.

[2] Part of the optional External Control facility (see Section 8.1).

[3] Part of the optional Performance Monitor facility (see Appendix E).

[4] Part of the Data Address Breakpoint mechanism (see Section 4.7).

[5] This register is a hypervisor resource, and can be modified by this instruction only in hypervisor state (see Section 1.7).

All SPR numbers not shown above, or in Figure 18, or in Book IV are reserved.

Figure 17. SPR encodings for mtspr

Compiler and Assembler Note

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

Programming Note

On processors prior to POWER4+ the following differences exist.

- SPR alias number 259 (SPRG3 - not privileged) is not implemented.
- HDEC and PURR are not implemented.
- SPRG0 is treated as a hypervisor register by this instruction.

Programming Note

For a discussion of software synchronization requirements when altering certain Special Purpose Registers, see Chapter 7. “Synchronization Requirements for Context Alterations” on page 85.

Compatibility Note

For a discussion of POWER compatibility with respect to SPR numbers not shown in the instruction descriptions for ***mtspr*** and ***mfspr***, see the appendix entitled “Incompatibilities with the POWER Architecture” in Book I, *PowerPC User Instruction Set Architecture.*

Move From Special Purpose Register XFX-form

mfspr RT,SPR

31	RT	spr	339	/
0	6	11	21	31

```
n ← spr5:9 || spr0:4
if length(SPREG(n)) = 64 then
  RT ← SPREG(n)
else
  RT ← 32 0 || SPREG(n)
```

The SPR field denotes a Special Purpose Register, encoded as shown in Figure 18. The contents of the designated Special Purpose Register are placed into register RT. For Special Purpose Registers that are 32 bits long, the low-order 32 bits of RT receive the contents of the Special Purpose Register and the high-order 32 bits of RT are set to zero.

spr_0=1 if and only if reading the register is privileged. Execution of this instruction specifying a defined and privileged register when MSR_{PR}=1 causes a Privileged Instruction type Program interrupt.

Execution of this instruction specifying an SPR number that is not defined for the implementation causes either an Illegal Instruction type Program interrupt or one of the following.

- if spr_0=0: boundedly undefined results
- if spr_0=1:
  - if MSR_{PR}=1: Privileged Instruction type Program interrupt
  - if MSR_{PR}=0: boundedly undefined results

If the SPR field contains a value that is shown in Figure 18 but corresponds to an optional Special Purpose Register that is not provided by the implementation, the effect of executing this instruction is the same as if the SPR number were not shown in the figure.

Special Registers Altered:

None

Note

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

Programming Note

On processors prior to POWER4+ the following differences exist.

- SPR alias number 259 (SPRG3) is not implemented.
- HDEC and PURR are not implemented.

decimal	SPR[1] $spr_{5:9}$	SPR[1] $spr_{0:4}$	Register Name	Privileged
1	00000	00001	XER	no
8	00000	01000	LR	no
9	00000	01001	CTR	no
18	00000	10010	DSISR	yes
19	00000	10011	DAR	yes
22	00000	10110	DEC	yes
25	00000	11001	SDR1	yes
26	00000	11010	SRR0	yes
27	00000	11011	SRR1	yes
29	00000	11101	ACCR	yes
136	00100	01000	CTRL	no
272	01000	10000	SPRG0	yes
273	01000	10001	SPRG1	yes
274	01000	10010	SPRG2	yes
259,275	01000	n0011	SPRG3 [5]	no,yes
282	01000	11010	EAR [2]	yes
287	01000	11111	PVR	yes
304	01001	10000	HSPRG0	hypv
305	01001	10001	HSPRG1	hypv
309	01001	10101	PURR [5]	yes
310	01001	10110	HDEC	yes
312	01001	11000	RMOR	hypv
313	01001	11001	HRMOR	hypv
314	01001	11010	HSRR0	hypv
315	01001	11011	HSRR1	hypv
318	01001	11110	LPCR	hypv
319	01001	11111	LPIDR	hypv
768-799	11000	nxxxx	perf_mon [3,5]	no,yes
1013	11111	10101	DABR [4]	yes
1015	11111	10111	DABRX [4]	yes
1023	11111	11111	PIR	yes

[1] Note that the order of the two 5-bit halves of the SPR number is reversed.

[2] Part of the optional External Control facility (see Section 8.1).

[3] Part of the optional Performance Monitor facility (see Appendix E).

[4] Part of the Data Address Breakpoint mechanism (see Section 4.7).

[5] Reading the SPR is privileged if and only if n=1.

Moving from the Time Base (TB and TBU) is accomplished with the ***mftb*** instruction, described in Book II.

All SPR numbers not shown above, or in Figure 17, or in Book IV are reserved.

Figure 18. SPR encodings for mfspr

Move To Machine State Register Doubleword X-form

mtmsrd RS,L

31	RS	///	L	///	178	/
0	6	11	15	16	21	31

```
if L = 0 then
   MSR0 ← (RS)0 | (RS)1

   MSR48 ← (RS)48 | (RS)49
   MSR58 ← (RS)58 | (RS)49
   MSR59 ← (RS)59 | (RS)49
   MSR1:2 4:47 49:50 52:57 60:63 ← (RS)1:2 4:47 49:50 52:57 60:63
else
   MSR48 62 ← (RS)48 62
```

The MSR is set based on the contents of register RS and of the L field.

L=0:

The result of ORing bits 0 and 1 of register RS is placed into MSR_0. The result of ORing bits 48 and 49 of register RS is placed into MSR_{48}. The result of ORing bits 58 and 49 of register RS is placed into MSR_{58}. The result of ORing bits 59 and 49 of register RS is placed into MSR_{59}. Bits 1:2, 4:47, 49:50, 52:57, and 60:63 of register RS are placed into the corresponding bits of the MSR.

L=1:

Bits 48 and 62 of register RS are placed into the corresponding bits of the MSR. The remaining bits of the MSR are unchanged.

This instruction is privileged.

If L=0 this instruction is context synchronizing except with respect to alterations to the LE bit; see Chapter 7. “Synchronization Requirements for Context Alterations” on page 85. If L=1 this instruction is execution synchronizing; in addition, the alterations of the EE and RI bits take effect as soon as the instruction completes.

Special Registers Altered:

MSR

Except in the ***mtmsrd*** instruction description in this section, references to “***mtmsrd***” in Books I - III imply either L value unless otherwise stated or obvious from context (e.g., a reference to an ***mtmsrd*** instruction that modifies an MSR bit other than the EE or RI bit implies L=0).

Programming Note

Warning: Processors that comply with versions of the architecture that precede Version 2.01 ignore the L field. These processors set the MSR as if L were 0, and perform synchronization as if L were 1. Therefore software that uses ***mtmsrd*** and will run on such processors must obey the following rules.

1. If L=1, the contents of bits of register RS other than bits 48 and 62 must be such that if L were 0 the instruction would not alter the contents of the corresponding MSR bits.
2. If L=0 and the instruction alters the contents of any of the MSR bits listed below, the instruction must be followed by a context synchronizing instruction or event in order to ensure that the context alteration caused by the ***mtmsrd*** instruction has taken effect on such processors; Chapter 7.

   SF, PR, FP, FE0, FE1, SE, BE, US, IR, DR

   To obtain the best performance on processors that comply with Version 2.01 of the architecture and with subsequent versions, if the context synchronizing instruction is ***isync*** the ***isync*** should immediately follow the ***mtmsrd***. (Some such processors treat an ***isync*** instruction that immediately follows an ***mtmsrd*** instruction having L=0 as a no-op, thereby avoiding the performance penalty of a second context synchronization.)

Programming Note

If this instruction sets MSR_{PR} to 1, it also sets MSR_{EE}, MSR_{IR}, and MSR_{DR} to 1. On processors prior to POWER4+, the setting of MSR_{PR} does not affect the setting of MSR_{EE}.

This instruction does not alter MSR_{HV} or MSR_{ME}.

If the only MSR bits to be altered are $MSR_{EE\ RI}$, to obtain the best performance L=1 should be used.

Programming Note

If MSR_{EE}=0 and an External or Decrementer exception is pending, executing an ***mtmsrd*** instruction that sets MSR_{EE} to 1 will cause the External or Decrementer interrupt to occur before the next instruction is executed, if no higher priority exception exists (see Section 5.8, “Interrupt Priorities” on page 76). Similarly, if a Hypervisor Decrementer interrupt is pending, execution of the instruction by the hypervisor causes a Hypervisor Decrementer interrupt to occur if HDICE=1.

For a discussion of software synchronization requirements when altering certain MSR bits, see Chapter 7.

Programming Note

mtmsrd serves as both a basic and an extended mnemonic. The Assembler will recognize an ***mtmsrd*** mnemonic with two operands as the basic form, and an ***mtmsrd*** mnemonic with one operand as the extended form. In the extended form the L operand is omitted and assumed to be 0.

Move From Machine State Register X-form

mfmsr RT

31	RT	///	///	83	/
0	6	11	16	21	31

RT ← MSR

The contents of the MSR are placed into register RT.

This instruction is privileged.

Special Registers Altered:

None

Chapter 4. Storage Control

4.1 Storage Addressing

A program references storage using the effective address computed by the processor when it executes a *Load*, *Store*, *Branch*, or *Cache Management* instruction, or when it fetches the next sequential instruction. The effective address is translated to a real address according to procedures described in Section 4.2.6 on page 29, in Section 4.3 on page 32 and in the following sections. The real address is what is presented to the storage subsystem.

For a complete discussion of storage addressing and effective address calculation, see the section entitled "Storage Addressing" in Book I, *PowerPC User Instruction Set Architecture*.

Storage Control Overview

- Real address space size is 2^m bytes, $m \leq 62$; see Note 1.
- Real page size is 2^{12} bytes (4 KB).
- Effective address space size is 2^{64} bytes.

- An effective address is translated to a virtual address via the Segment Lookaside Buffer (SLB).
  - Virtual address space size is 2^n bytes, $65 \le n \le 80$; see Note 2.
  - Segment size is 2^{28} bytes (256 MB).
  - Number of virtual segments is 2^{n-28}; see Note 2.
  - Virtual page size is 2^p bytes, $12 \le p \le 28$; two sizes are supported simultaneously, 4 KB (p=12) and a larger size; see Note 3.
- A virtual address is translated to a real address via the Page Table.

Notes:

1. The value of m is implementation-dependent (subject to the maximum given above). When used to address storage, the high-order 62-m bits of the "62-bit" real address must be zeros.
2. The value of n is implementation-dependent (subject to the range given above). In references to 80-bit virtual addresses elsewhere in this Book, the high-order 80-n bits of the "80-bit" virtual address are assumed to be zeros.
3. The value of p for the larger virtual page size is implementation-dependent (subject to the range given above).

4.2 Storage Model

The storage model provides the following features.

1. The architecture allows the storage implementations to take advantage of the performance benefits of weak ordering of storage accesses between processors or between processors and I/O devices.
2. In general, storage accesses appear to be performed in program order with respect to the processor performing them but may be performed in different orders with respect to other processors and mechanisms. Exceptions to this rule are stated in the appropriate sections.
3. The architecture provides instructions that allow the programmer to ensure a consistent and ordered storage state.

   - ***dcbf*[*l*]**
   - ***dcbst***
   - ***eieio***
   - ***icbi***
   - ***isync***
   - ***ldarx***
   - ***lwarx***
   - ***stdcx.***
   - ***stwcx.***
   - *Synchronize*
   - ***tlbsync***

4. Storage consistency between processors, and between a processor and an I/O device, is controlled by software using the "WIM" storage control bits (see Section 4.8). These bits allow software to control whether a given storage location has any of the following attributes.

   - Write Through Required (W)
   - Caching Inhibited (I)
   - Memory Coherence Required (M)

4.2.1 Storage Exceptions

A *storage exception* is an exception that causes an Instruction Storage interrupt, an Instruction Segment interrupt, a Data Storage interrupt, a Data Segment interrupt, or an Alignment interrupt. Attempting to fetch or execute an instruction causes a storage exception if certain conditions apply. Such conditions include the following.

- The appropriate relocate bit in the MSR is set to 1 and the effective address cannot be translated to a real address.
- The access is not permitted by the storage protection mechanism.
- The access causes a Data Address Compare match or a Data Address Breakpoint match.

In certain cases a storage exception may result in the "restart" of (re-execution of at least part of) a Load or Store instruction. See the section entitled "Instruction Restart" in Book II, *PowerPC Virtual Environment*

Architecture, and Section 5.6, “Partially Executed Instructions” on page 73 in this Book.

4.2.2 Instruction Fetch

Instructions are fetched under control of MSR_{IR}.

MSR_{IR}=0

The effective address of the instruction is interpreted as described in Section 4.2.6, “Real Addressing Mode” on page 29.

MSR_{IR}=1

The effective address of the instruction is translated by the Address Translation mechanism. (If it cannot be translated, a storage exception occurs.)

4.2.2.1 Implicit Branch

Explicitly altering certain MSR bits (using ***mtmsr***[***d***]), or explicitly altering SLB entries, Page Table entries, or certain System Registers (including the HRMOR, and possibly other implementation-dependent registers), may have the side effect of changing the addresses, effective or real, from which the current instruction stream is being fetched. This side effect is called an *implicit branch*. For example, an ***mtmsrd*** instruction that changes the value of MSR_{SF} may change the effective addresses from which the current instruction stream is being fetched. The MSR bits and System Registers (excluding implementation-dependent registers) for which alteration can cause an implicit branch are indicated as such in Chapter 7. “Synchronization Requirements for Context Alterations” on page 85. Implicit branches are not supported by the PowerPC Architecture. If an implicit branch occurs, the results are boundedly undefined.

4.2.2.2 Address Wrapping Combined with Changing MSR Bit SF

If the current instruction is at effective address $2^{32} - 4$ and is an ***mtmsrd*** instruction that changes the contents of MSR_{SF}, the effective address of the next sequential instruction is undefined.

Programming Note

In the case described in the preceding paragraph, if an interrupt occurs before the next sequential instruction is executed, the contents of SRR0 or HSRR0, as appropriate to the interrupt, are undefined.

4.2.3 Data Access

Data accesses are controlled by MSR_{DR}.

MSR_{DR}=0

The effective address of the data is interpreted as described in Section 4.2.6, “Real Addressing Mode”.

MSR_{DR}=1

The effective address of the data is translated by the Address Translation mechanism. (If it cannot be translated, a storage exception occurs.)

4.2.4 Performing Operations Out-of-Order

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

Operations are performed out-of-order by the processor on the expectation that the results will be needed by an instruction that will be required by the sequential execution model. Whether the results are really needed is contingent on everything that might divert the control flow away from the instruction, such as *Branch*, *Trap*, *System Call*, ***rfid***, and ***hrfid*** instructions, and interrupts, and on everything that might change the context in which the instruction is executed.

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

In the remainder of this section, including its subsections, “*Load* instruction” includes the *Cache Management* and other instructions that are stated in the instruction descriptions to be “treated as a *Load*”, and similarly for “*Store* instruction”.

A data access that is performed out-of-order may correspond to an arbitrary *Load* or ***Store*** instruction (e.g., a *Load* or *Store* instruction that is not in the instruction stream being executed). Similarly, an instruction fetch that is performed out-of-order may be for an arbitrary instruction (e.g., the aligned word at an arbitrary location in instruction storage).

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

- Stores

  Stores are not performed out-of-order (even if the Store instructions that caused them were executed out-of-order). Moreover, address translations associated with instructions preceding the corresponding Store instruction are not performed again after the store has been performed.

  Programming Note

  The fact that address translations associated with preceding instructions are not performed again after the store has been performed permits Page Table Entries to be updated without a preceding context synchronizing operation; Section 4.12, “Page Table Update Synchronization Requirements” on page 57. (These address translations must have been performed before the store was determined to be required by the sequential execution model, because they might have caused an exception.)

- Accessing Guarded Storage

  The restrictions for this case are given in Section 4.2.4.2.

The only permitted side effects of performing an operation out-of-order are the following.

- A Machine Check or Checkstop that could be caused by in-order execution may occur out-of-order, except as described in Section 8.2 if the optional Real Mode Storage Control facility is implemented.
- Reference and Change bits may be set as described in Section 4.9, “Reference and Change Recording” on page 43.
- Non-Guarded storage locations that could be fetched into a cache by in-order fetching or execution of an arbitrary instruction may be fetched out-of-order into that cache.

4.2.4.1 Guarded Storage

Storage is said to be “well-behaved” if the corresponding real storage exists and is not defective, and if the effects of a single access to it are indistinguishable from the effects of multiple identical accesses to it. Data and instructions can be fetched out-of-order from well-behaved storage without causing undesired side effects.

Storage is said to be Guarded if either of the following conditions is satisfied.

- MSR bit IR or DR is 1 for instruction fetches or data accesses respectively, and the G bit is 1 in the relevant Page Table Entry.
- MSR bit IR or DR is 0 for instruction fetches or data accesses respectively, MSR_{HV}=1, and the optional Real Mode Storage Control facility (see Section 8.2) is not implemented. In this case all of storage is Guarded for the corresponding accesses.

In general, storage that is not well-behaved should be Guarded. Because such storage may represent a control register on an I/O device or may include locations that do not exist, an out-of-order access to such storage may cause an I/O device to perform unintended operations or may result in a Machine Check.

The following rules apply to in-order execution of *Load* and *Store* instructions for which the first byte of the storage operand is in storage that is both Caching Inhibited and Guarded.

- *Load* or *Store* instruction that causes an atomic access

  If any portion of the storage operand has been accessed and an External, Decrementer, Hypervisor Decrementer, or Imprecise mode Floating-Point Enabled exception is pending, the instruction completes before the interrupt occurs.

- *Load* or *Store* instruction that causes an Alignment exception, or that causes a Data Storage exception for reasons other than Data Address Compare match or Data Address Breakpoint match

  The portion of the storage operand that is in Caching Inhibited and Guarded storage is not accessed.

  (The corresponding rules for instructions that cause a Data Address Compare match or Data Address Breakpoint match are given in Sections 4.6 and 4.7 respectively.)

4.2.4.2 Out-of-Order Accesses to Guarded Storage

In general, Guarded storage is not accessed out-of-order. The only exceptions to this rule are the following.

Load Instruction

If a copy of any byte of the storage operand is in a cache then that byte may be accessed in the cache or in main storage.

Instruction Fetch

If MSR_{IR}=0 then an instruction may be fetched if any of the following conditions are met.

1. The instruction is in a cache. In this case it may be fetched from the cache or from main storage.
2. The instruction is in a real page from which an instruction has previously been fetched, except that if that previous fetch was based on condition 1 then the previously fetched instruction must have been in the instruction cache.
3. The instruction is in the same real page as an instruction that is required by the sequential execution model, or is in the real page immediately following such a page.

Programming Note

Software should ensure that only well-behaved storage is copied into a cache, either by accessing as Caching Inhibited (and Guarded) all storage that may not be well-behaved, or by accessing such storage as not Caching Inhibited (but Guarded) and referring only to cache blocks that are well-behaved.

If a real page contains instructions that will be executed when MSR_{IR}=0 and MSR_{HV}=1, software should ensure that this real page and the next real page contain only well-behaved storage (or, if the optional Real Mode Storage Control facility is implemented, that this real page is not Guarded).

4.2.5 32-Bit Mode

The computation of the 64-bit effective address is independent of whether the processor is in 32-bit mode or 64-bit mode. In 32-bit mode (MSR_{SF}=0), the high-order 32 bits of the 64-bit effective address are treated as zeros for the purpose of addressing storage. This applies to both data accesses and instruction fetches. It applies independent of whether address translation is enabled or disabled. This truncation of the effective address is the only respect in which storage accesses in 32-bit mode differ from those in 64-bit mode.

Programming Note

Treating the high-order 32 bits of the effective address as zeros effectively truncates the 64-bit effective address to a 32-bit effective address such as would have been generated on a 32-bit implementation of the PowerPC Architecture. Thus, for example, the ESID in 32-bit mode is the high-order four bits of this truncated effective address; the ESID thus lies in the range 0-15. When address translation is enabled, these four bits would select a Segment Register on a 32-bit implementation of the PowerPC Architecture. The SLB entries that translate these 16 ESIDs can be used to emulate these Segment Registers.

4.2.6 Real Addressing Mode

A storage access is said to be performed in "real addressing mode" if the access is an instruction fetch and instruction address translation is disabled, or if the access is a data access and data address translation is disabled. Storage accesses in real addressing mode are performed in a manner that depends on the contents of MSR_{HV}, LPES, HRMOR, RMLR, and RMOR (see Section 1.7, "Logical Partitioning (LPAR)" on page 5), and bit 0 of the effective address (EA_0) as described below. Bit 1 of the effective address is ignored.

MSR_{HV}=1

- If EA_0=0, the Hypervisor Offset Real Mode Address mechanism, described in Section 4.2.6.1, controls the access.
- If EA_0=1, bits 2:63 of the effective address are used as the real address for the access.

MSR_{HV}=0

- If $LPES_1$=0, the access causes a storage exception as described in Section 4.10.2, "Storage Protection, Address Translation Disabled" on page 46.
- If $LPES_1$=1, the Offset Real Mode Address mechanism, described in Section 4.2.6.2, controls the access.

4.2.6.1 Hypervisor Offset Real Mode Address

If MSR_{HV} = 1 and EA_0 = 0, the access is controlled by the contents of the Hypervisor Real Mode Offset Register, as follows.

Hypervisor Real Mode Offset Register (HRMOR)

Bits 2:63 of the effective address for the access are ORed with the 62-bit offset represented by the contents of the HRMOR, and the 62-bit result is used as the real address for the access. The supported offset values are all values of the form $i \times 2^r$, where $0 \leq i < 2^j$, and j and r are implementation-dependent values having the properties that $12 \leq r \leq 26$ (i.e., the minimum offset granularity is 4 KB and the maximum offset granularity is 64 MB) and $j+r \leq m \leq 62$, where the real address size supported by the implementation is m bits.

Programming Note

$EA_{2:63-r}$ should equal $^{62-r}0$. If this condition is satisfied, ORing the effective address with the offset produces a result that is equivalent to adding the effective address and the offset.

If m<62, $EA_{2:63-m}$ and $HRMOR_{0:61-m}$ must be zeros.

Software must ensure that altering the HRMOR does not cause an implicit branch.

4.2.6.2 Offset Real Mode Address

If MSR_{HV}=0 and $LPES_1$=1, the access is controlled by the contents of the Real Mode Limit Register and Real Mode Offset Register, as follows.

Real Mode Limit Register (RMLR)

If bits 2:63 of effective address for the access are greater than or equal to the value (limit) represented by the contents of the RMLR, the access causes a storage exception (see Section 4.10.2). In this comparison, if m<62, bits 2:63-m of the effective address may be ignored (i.e., treated as if they were zeros), where the real address size supported by the implementation is m bits. The supported limit values are of the form 2^j, where $12 \leq j \leq 62$. Subject to the preceding sentence, the number and values of the limits supported are implementation-dependent.

Real Mode Offset Register (RMOR)

If the access is permitted by the RMLR, bits 2:63 of the effective address for the access are ORed with the 62-bit offset represented by the contents of the RMOR, and the low-order m bits of the 62-bit result are used as the real address for the access. The supported offset values are all values of the form $i \times 2^s$, where $0 \leq i < 2^k$, and k and s are implementation-dependent values having the properties that 2^s is the minimum limit value supported by the implementation (i.e., the minimum value representable by the contents of the RMLR) and $m \leq k+s \leq 62$.

Programming Note

The offset specified by the RMOR should be a non-zero multiple of the limit specified by the RMLR. If these registers are set thus, ORing the effective address with the offset produces a result that is equivalent to adding the effective address and the offset. (The offset must not be zero, because real page 0 contains the fixed interrupt vectors and real pages 1 and 2 may be used for implementation-specific purposes; see Section 4.2.7, “Address Ranges Having Defined Uses” on page 31.)

4.2.6.3 Storage Control Attributes for Real Addressing Mode and for Implicit Storage Accesses

Data accesses and instruction fetches in real addressing mode when the processor is in hypervisor state are performed as though all of storage had the following storage control attributes, except as modified by the optional Real Mode Storage Control facility (see Section 8.2) if that facility is implemented. (The storage control attributes are defined in Book II, *PowerPC Virtual Environment Architecture*.)

- not Write Through Required
- not Caching Inhibited, for instruction fetches
- not Caching Inhibited, for data accesses if the Real Mode Caching Inhibited bit is set to 0; Caching Inhibited, for data accesses if the Real Mode Caching Inhibited bit is set to 1
- Memory Coherence Required, for data accesses
- Guarded

Storage accesses in real addressing mode when the processor is not in hypervisor state are performed as though all of storage had the following storage control attributes. (Such accesses use the Offset Real Mode Address mechanism.)

- not Write Through Required
- not Caching Inhibited
- Memory Coherence Required, for data accesses
- not Guarded

Implicit accesses to the Page Table by the processor in performing address translation and in recording reference and change information are performed as though the storage occupied by the Page Table had the following storage control attributes.

- not Write Through Required
- not Caching Inhibited
- Memory Coherence Required
- not Guarded

The definition of “performed” given in Book II applies also to these implicit accesses; accesses for performing address translation are considered to be loads in this respect, and accesses for recording reference and change information are considered to be stores. These implicit accesses are ordered by the ***ptesync*** instruction as described in Section 4.11.2 on page 47.

Software must ensure that any data storage location that is accessed with the Real Mode Caching Inhibited bit set to 1 is not in the caches.

Software must ensure that the Real Mode Caching Inhibited bit contains 0 whenever data address translation is enabled and whenever the processor is not in hypervisor state.

Programming Note

Because storage accesses in real addressing mode do not use the SLB or the Page Table, accesses in this mode bypass all checking and recording of information contained therein (e.g., storage protection checks that use information contained therein are not performed, and reference and change information is not recorded).

The Real Mode Caching Inhibited bit can be used to permit a control register on an I/O device to be accessed without permitting the corresponding storage location to be copied into the caches. The bit should normally contain 0. Software would set the bit to 1 just before accessing the control register, access the control register as needed, and then set the bit back to 0.

4.2.7 Address Ranges Having Defined Uses

The address ranges described below have uses that are defined by the architecture.

- Fixed interrupt vectors

  Except for the first 256 bytes, which are reserved for software use, the real page beginning at real address 0x0000_0000_0000_0000 is either used for interrupt vectors or reserved for future interrupt vectors.

- Implementation-specific use

  The two contiguous real pages beginning at real address 0x0000_0000_0000_1000 are reserved for implementation-specific purposes.

- Offset Real Mode interrupt vectors

  The real pages beginning at the real address specified by the HRMOR and RMOR are used similarly to the page for the fixed interrupt vectors.

- Page Table

  A contiguous sequence of real pages beginning at the real address specified by SDR1 contains the Page Table.

4.2.8 Invalid Real Address

A storage access (including an access that is performed out-of-order; see Section 4.2.4) may cause a Machine Check if the accessed storage location contains an uncorrectable error or does not exist. In the latter case the Checkstop state may be entered. See Section 5.5.2, “Machine Check Interrupt” on page 66.

Programming Note

Hypervisor software must ensure that a storage access by a program in one partition will not cause a Checkstop or other system-wide event that could affect the integrity of other partitions (see Section 1.7, “Logical Partitioning (LPAR)” on page 5). For example, such an event could occur if a real address placed in a Page Table Entry or made accessible to a partition using the Offset Real Mode Address mechanism (see Section 4.2.6.3) does not exist.

4.3 Address Translation Overview

The effective address (EA) is the address generated by the processor for an instruction fetch or for a data access. If address translation is enabled, this address is passed to the Address Translation mechanism, which attempts to convert the address to a real address which is then used to access storage.

The first step in address translation is to convert the effective address to a virtual address (VA), as described in Section 4.4. The second step, conversion of the virtual address to a real address (RA), is described in Section 4.5.

If the effective address cannot be translated, a storage exception (see Section 4.2.1) occurs.

Figure 19 gives an overview of the address translation process.

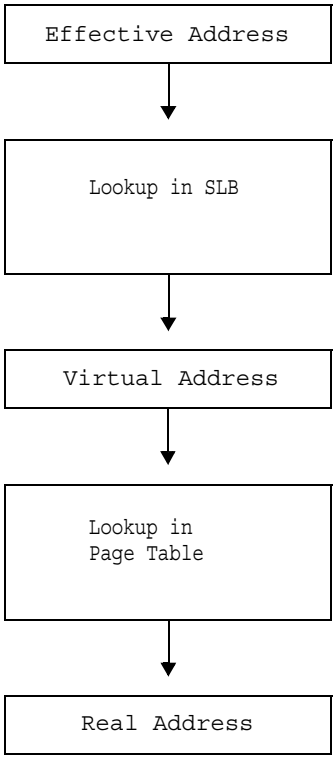


Figure 19. Address translation overview

4.4 Virtual Address Generation

Conversion of a 64-bit effective address to a virtual address is done by searching the Segment Lookaside Buffer (SLB) as shown in Figure 20.

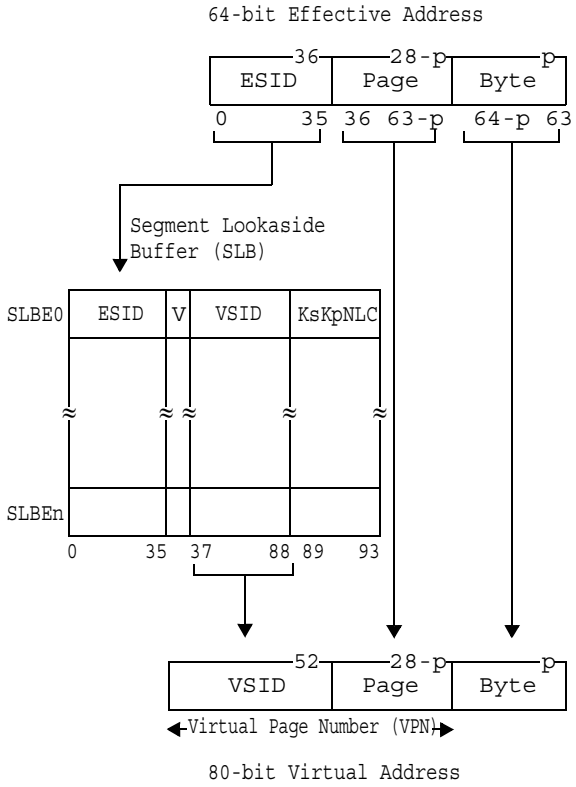


Figure 20. Translation of 64-bit effective address to 80-bit virtual address

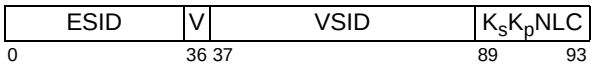
4.4.1 Segment Lookaside Buffer (SLB)

The Segment Lookaside Buffer (SLB) specifies the mapping between Effective Segment IDs (ESIDs) and Virtual Segment IDs (VSIDs). The number of SLB entries is implementation-dependent, except that all implementations provide at least 32 entries.

The contents of the SLB are managed by software, using the instructions described in Section 4.11.3.1, “SLB Management Instructions” on page 49. See Chapter 7. “Synchronization Requirements for Context Alterations” on page 85 for the rules that software must follow when updating the SLB.

SLB Entry

Each SLB entry (SLBE) maps one ESID to one VSID. Figure 21 shows the layout of an SLB entry.



Bit(s)	Name	Description
0:35	ESID	Effective Segment ID
36	V	Entry valid (V=1) or invalid (V=0)
37:88	VSID	Virtual Segment ID
89	K_s	Supervisor (privileged) state storage key
90	K_p	Problem state storage key
91	N	No-execute segment if N=1
92	L	Virtual pages are large (L=1) or 4 KB (L=0)
93	C	Class

Figure 21. SLB Entry

On implementations that support a virtual address size of only n bits, n<80, bits 0:79-n of the VSID field are treated as reserved bits, and software must set them to zeros.

A No-execute segment (N=1) contains data that should not be executed.

The L bit selects between two virtual page sizes, 4 KB (p=12) and “large”. The large page size is an implementation-dependent value that is a power of 2 and is in the range 8 KB : 256 MB ($13 \le p \le 28$). Some implementations may provide a means by which software can select the large page size from a set of several implementation-dependent sizes during system initialization.

If “large page” is used in reference to real storage, it means the sequence of contiguous real (4 KB) pages to which a large virtual page is mapped.

The Class field is used in conjunction with the ***slbie*** instruction (see Section 4.11.3.1).

Software must ensure that the SLB contains at most one entry that translates a given effective address (i.e., that a given ESID is contained in no more than one SLB entry).

4.4.2 SLB Search

When the hardware searches the SLB, all entries are tested for a match with the EA. For a match to exist, the following conditions must be satisfied.

- $SLBE_V = 1$
- $SLBE_{ESID} = EA_{0:35}$

If the SLB search succeeds, the virtual address (VA) is formed by concatenating the VSID from the matching SLB entry with bits 36:63 of the EA.

The Virtual Page Number (VPN) is bits 0:79-p of the virtual address.

If the SLB search fails, a *segment fault* occurs. This is an Instruction Segment exception or a Data Segment exception, depending on whether the effective address is for an instruction fetch or for a data access.

4.5 Virtual to Real Translation

Conversion of an 80-bit virtual address to a real address is done by searching the Page Table as shown in Figure 22.

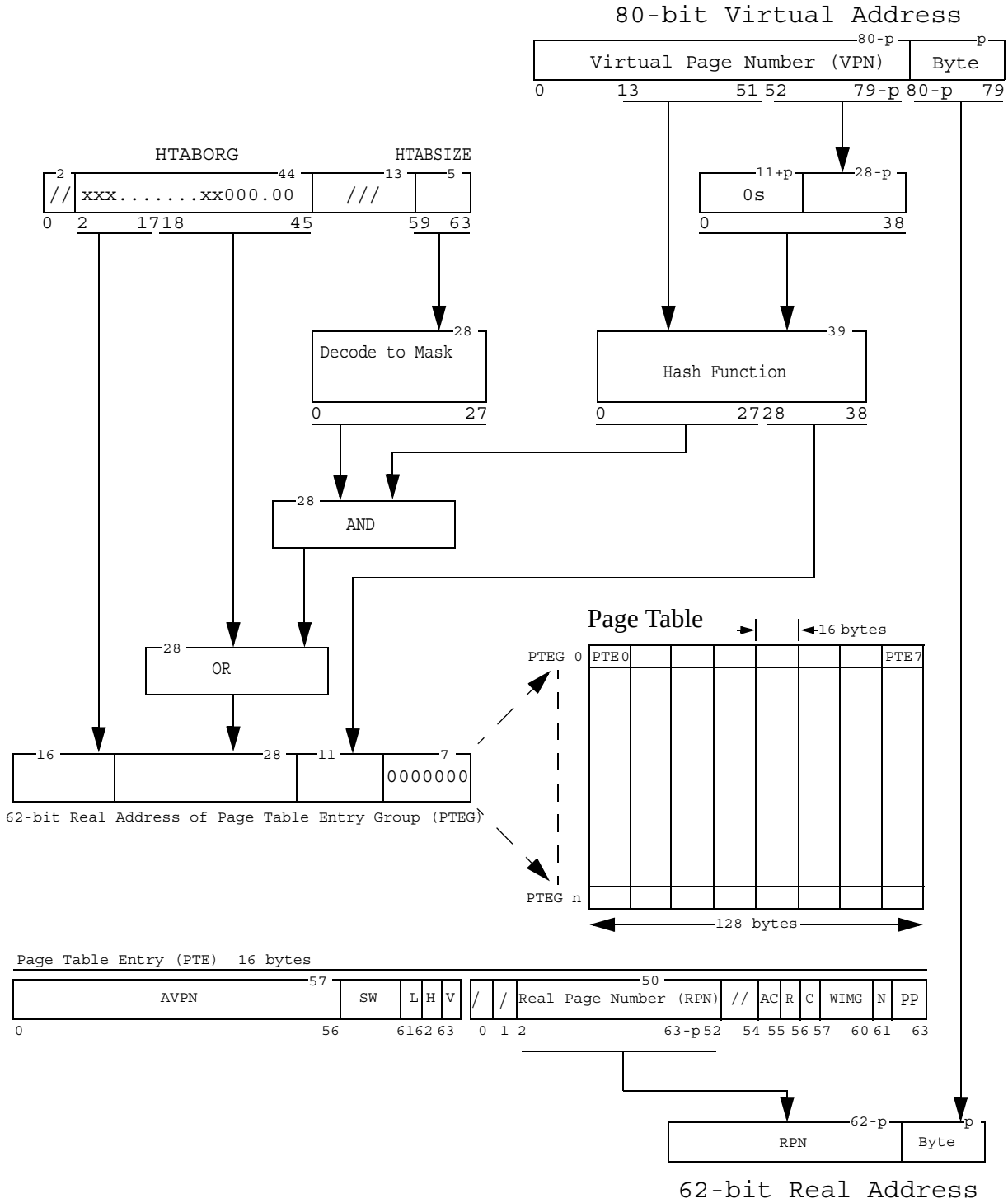


Figure 22. Translation of 80-bit virtual address to 62-bit real address

4.5.1 Page Table

The Hashed Page Table (HTAB) is a variable-sized data structure that specifies the mapping between Virtual Page Numbers and Real Page Numbers. The HTAB's size must be a multiple of 4 KB, its starting address must be a multiple of its size, and it must be located in storage having the storage control attributes that are used for implicit accesses to it (see Section 4.2.6.3).

The HTAB contains Page Table Entry Groups (PTEGs). A PTEG contains 8 Page Table Entries (PTEs) of 16 bytes each; each PTEG is thus 128 bytes long. PTEGs are entry points for searches of the Page Table.

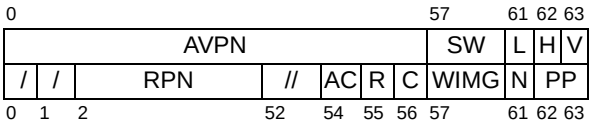
See see Section 4.12, "Page Table Update Synchronization Requirements" on page 57 for the rules that software must follow when updating the Page Table.

Programming Note

The Page Table must be treated as a hypervisor resource (see Section 1.7, "Logical Partitioning (LPAR)" on page 5), and therefore must be placed in real storage to which only the hypervisor has write access. Moreover, the contents of the Page Table must be such that non-hypervisor software cannot modify storage that contains hypervisor programs or data.

Page Table Entry

Each Page Table Entry (PTE) maps one VPN to one RPN. Figure 23 shows the layout of a PTE.



Dword	Bit(s)	Name	Description
0	0:56	AVPN	Abbreviated Virtual Page Number
	57:60	SW	Available for software use
	61	L	Virtual page is large (L=1) or 4 KB (L=0)
	62	H	Hash function identifier
	63	V	Entry valid (V=1)or invalid (V=0)
1	2:51	RPN	Real Page Number
	54	AC	Address Compare bit
	55	R	Reference bit
	56	C	Change bit
	57:60	WIMG	Storage control bits
	61	N	No-execute page if N=1
	62:63	PP	Page protection bits

All other fields are reserved.

Figure 23. Page Table Entry

If p≤23, the Abbreviated Virtual Page Number (AVPN) field contains bits 0:56 of the VPN. Otherwise bits 0:79-p of the AVPN field contain bits 0:79-p of the VPN, and bits 80-p:56 of the AVPN field must be zeros.

Programming Note

If p≤23, the AVPN field omits the low-order 23-p bits of the VPN. These bits are not needed in the PTE, because the low-order 11 bits of the VPN are always used in selecting the PTEGs to be searched (see Section 4.5.3).

On implementations that support a virtual address size of only n bits, n<80, bits 0:79-n of the AVPN field must be zeros.

The RPN field contains the page number of the real page that contains the first byte of the block of real storage to which the virtual page is mapped. If p>12, the low-order p-12 bits of the RPN field (bits 64-p:51 of doubleword 1 of the PTE) must be 0. On implementations that support a real address size of only m bits, m<62, bits 0:61-m of the RPN field must be zeros.

Programming Note

For a large virtual page, the high-order 62-p bits of the RPN field (bits 0:61-p) comprise the large real page number.

A No-execute page (N=1) contains data that should not be executed.

Page Table Size

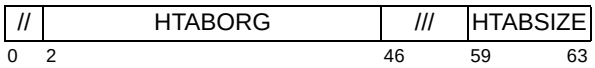
The number of entries in the Page Table directly affects performance because it influences the hit ratio in the Page Table and thus the rate of page faults. If the table is too small, it is possible that not all the virtual pages that actually have real pages assigned can be mapped via the Page Table. This can happen if too many hash collisions occur and there are more than 16 entries for the same primary/secondary pair of PTEGs. While this situation cannot be guaranteed not to occur for any size Page Table, making the Page Table larger than the minimum size (see Section 4.5.2) will reduce the frequency of occurrence of such collisions.

Programming Note

If large pages are not used, it is recommended that the number of PTEGs in the Page Table be at least half the number of real pages to be accessed. For example, if the amount of real storage to be accessed is 2^{31} bytes (2 GB), then we have $2^{31-12}=2^{19}$ real pages. The minimum recommended Page Table size would be 2^{18} PTEGs, or 2^{25} bytes (32 MB).

4.5.2 Storage Description Register 1

The SDR1 register is shown in Figure 24.



Bits	Name	Description
2:45	HTABORG	Real address of Page Table
59:63	HTABSIZE	Encoded size of Page Table

All other fields are reserved.

Figure 24. SDR1

SDR1 is a hypervisor resource; see Section 1.7, “Logical Partitioning (LPAR)” on page 5.

The HTABORG field in SDR1 contains the high-order 44 bits of the 62-bit real address of the Page Table. The Page Table is thus constrained to lie on a 2^{18} byte (256 KB) boundary at a minimum. At least 11 bits from the hash function (see Figure 22) are used to index into the Page Table. The minimum size Page Table is 256 KB (2^{11} PTEGs of 128 bytes each).

The Page Table can be any size 2^n bytes where $18 \le n \le 46$. As the table size is increased, more bits are used from the hash to index into the table and the value in HTABORG must have more of its low-order bits equal to 0.

The HTABSIZE field in SDR1 contains an integer giving the number of bits (in addition to the minimum of 11 bits) from the hash that are used in the Page Table index. This number must not exceed 28. HTABSIZE is used to generate a mask of the form 0b00...011...1, which is a string of 28 - HTABSIZE 0-bits followed by a string of HTABSIZE 1-bits. The 1-bits determine which additional bits (beyond the minimum of 11) from the hash are used in the index (see Figure 22). The number of low-order 0 bits in HTABORG must be greater than or equal to the value in HTABSIZE.

On implementations that support a real address size of only m bits, m<62, bits 0:61-m of the HTABORG field are treated as reserved bits, and software must set them to zeros.

Programming Note

Let n equal the virtual address size (in bits) supported by the implementation. If n<67, software should set the HTABSIZE field to a value that does not exceed n-39. Because the high-order 80-n bits of the VSID are assumed to be zeros, the hash value used in the Page Table search will have the high-order 67-n bits either all 0s (primary hash; see Section 4.5.3) or all 1s (secondary hash). If HTABSIZE > n-39, some of these hash value bits will be used to index into the Page Table, with the result that certain PTEGs will not be searched.

Example:

Suppose that the Page Table is 16,384 (2^{14}) 128-byte PTEGs, for a total size of 2^{21} bytes (2 MB). A 14-bit index is required. Eleven bits are provided from the hash to start with, so 3 additional bits from the hash must be selected. Thus the value in HTABSIZE must be 3 and the value in HTABORG must have its low-order 3 bits (bits 43:45 of SDR1) equal to 0. This means that the Page Table must begin on a $2^{3+11+7} = 2^{21}$ = 2 MB boundary.

4.5.3 Page Table Search

When the hardware searches the Page Table, the accesses are performed as described in Section 4.2.6.3, “Storage Control Attributes for Real Addressing Mode and for Implicit Storage Accesses” on page 30.

An outline of the HTAB search process is shown in Figure 22. The detailed algorithm is as follows.

1. **Primary Hash:**
   A 39-bit hash value is computed by Exclusive ORing bits 13:51 of the VPN with a 39-bit value formed by concatenating 11+p 0-bits with the low-order 28-p bits of the VPN. The 62-bit real address of a PTEG is formed by concatenating the following values:
   - Bits 2:17 of SDR1 (the high-order 16 bits of HTABORG).
   - Bits 0:27 of the 39-bit hash value ANDed with the mask generated from bits 59:63 of SDR1 (HTABSIZE) and then ORed with bits 18:45 of SDR1 (the low-order 28 bits of HTABORG).
   - Bits 28:38 of the 39-bit hash value.
   - Seven 0-bits.

   This operation identifies a particular PTEG, called the “primary PTEG”, whose eight PTEs will be tested.
2. **Secondary Hash:**
   A 39-bit hash value is computed by taking the one’s complement of the Exclusive OR of bits 13:51 of the VPN with a 39-bit value formed by concatenating 11+p 0-bits with the low-order 28-p bits of the VPN. The 62-bit real address of a PTEG is formed by concatenating the following values:
   - Bits 2:17 of SDR1 (the high-order 16 bits of HTABORG).
   - Bits 0:27 of the 39-bit hash value ANDed with the mask generated from bits 59:63 of SDR1 (HTABSIZE) and then ORed with bits 18:45 of SDR1 (the low-order 28 bits of HTABORG).
   - Bits 28:38 of the 39-bit hash value.
   - Seven 0-bits.

   This operation identifies the “secondary PTEG”.
3. As many as 16 PTEs in the two identified PTEGs are tested to determine if any translate the given virtual address. Let q = minimum(5, 28-p). For a match to exist, the following conditions must be satisfied.
   - PTE_H=0 for the primary PTEG, 1 for the secondary PTEG
   - PTE_V=1
   - $PTE_{AVPN[0:51]}$=$VA_{0:51}$
   - if p<28, $PTE_{AVPN[52:51+q]}$=$VA_{52:51+q}$
   - if EA is an SLS address
     then PTE_L = 0
     else PTE_L = $SLBE_L$

   If one or more matches are found, the search is successful; otherwise it fails. If more than one match is found, the matching entries must be identical in all defined fields with the exception of SW, H, AC, R, and C. If they are, one of the matching entries is used, for the translation, Data Address Compare, and the setting of the R and C bits. If they are not, the translation and Data Address Compare are undefined, as is the setting of the R and C bits in the matching entries, and the remainder of this section does not apply.

If the Page Table search succeeds, the real address (RA) is formed by concatenating bits 0:61-p of the RPN from the matching PTE with bits 64-p:63 of the effective address (the byte offset).

$$RA=RPN_{0:61-p} \,||\, EA_{64-p:63}$$

The N (No-execute) value used for the storage access is the result of ORing the N bit from the matching PTE with the N bit from the SLB entry that was used to translate the effective address.

If the Page Table search fails, a *page fault* occurs. This is an Instruction Storage exception or a Data Storage exception, depending on whether the effective address is for an instruction fetch or for a data access.

Programming Note

To obtain the best performance, Page Table Entries should be allocated beginning with the first empty entry in the primary PTEG, or with the first empty entry in the secondary PTEG if the primary PTEG is full.

Translation Lookaside Buffer

Conceptually, the Page Table is searched by the address relocation hardware to translate every reference. For performance reasons, the hardware usually keeps a Translation Lookaside Buffer (TLB) that holds PTEs that have recently been used. The TLB is searched prior to searching the Page Table. As a consequence, when software makes changes to the Page Table it must perform the appropriate TLB invalidate operations to maintain the consistency of the TLB with the Page Table (see Section 4.12).

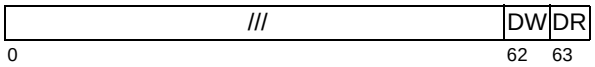
Programming Notes

1. Page Table entries may or may not be cached in a TLB.
2. It is possible that the hardware implements more than one TLB, such as one for data and one for instructions. In this case the size and shape of the TLBs may differ, as may the values contained therein.
3. Use the ***tlbie*** or ***tlbia*** instruction to ensure that the TLB no longer contains a mapping for a particular virtual page.

4.6 Data Address Compare

The Data Address Compare mechanism provides a means of detecting load and store accesses to a virtual page.

The Data Address Compare mechanism is controlled by the Address Compare Control Register (ACCR), and by a bit in each Page Table Entry (PTE_{AC}).



Bit	Name	Description
62	DW	Data Write
63	DR	Data Read

All other fields are reserved.

Figure 25. Address Compare Control Register

A Data Address Compare match occurs for a Load or Store instruction if for any byte accessed, the following conditions are satisfied.

- PTE_{AC}=1 for the PTE that translates the virtual address, and
- the instruction is a Store and $ACCR_{DW}$=1, or the instruction is a Load and $ACCR_{DR}$=1.

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

If the match 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 *Cache Management* instructions other than ***dcbz*** never cause a match.

A Data Address Compare match causes a Data Storage exception (see Section 5.5.3, “Data Storage Interrupt” on page 66). If a match occurs, some or all of the bytes of the storage operand may have been accessed; however, if a *Store*, ***dcbz***, or ***ecowx*** instruction causes the match, the bytes of the storage operand that are in a virtual page for which PTE_{AC}=1 are not modified.

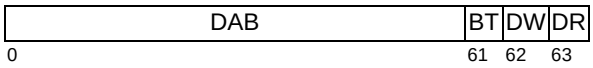
Programming Note

The Data Address Compare mechanism does not apply to instruction fetches, or to data accesses in real addressing mode (MSR_{DR}=0).

4.7 Data Address Breakpoint

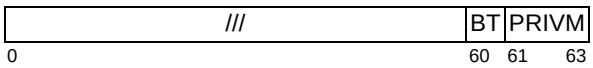
The Data Address Breakpoint mechanism provides a means of detecting load and store accesses to a designated doubleword. The address comparison is done on an effective address (EA).

The Data Address Breakpoint mechanism is controlled by the Data Address Breakpoint Register (DABR), shown in Figure 26, and the Data Address Breakpoint Register Extension (DABRX), shown in Figure 27.



Bit(s)	Name	Description
0:60	DAB	Data Address Breakpoint
61	BT	Breakpoint Translation
62	DW	Data Write
63	DR	Data Read

Figure 26. Data Address Breakpoint Register



Bit(s)	Name	Description
60	BTI	Breakpoint Translation Ignore
61:63	PRIVM	Privilege Mask
61	HYP	Hypervisor state
62	PNH	Privileged but Non-Hypervisor state
63	PRO	Problem state

All other fields are reserved.

Figure 27. Data Address Breakpoint Register Extension

The DABR and DABRX are hypervisor resources; see Section 1.7, “Logical Partitioning (LPAR)” on page 5.

The supported PRIVM values are 0b000, 0b001, 0b010, 0b011, 0b100, and 0b111. If the PRIVM field does not contain one of the supported values, then whether a match occurs for a given storage access is undefined. Elsewhere in this section it is assumed that the PRIVM field contains one of the supported values.

Programming Note

PRIVM value 0b000 causes matches not to occur regardless of the contents of other DABR and DABRX fields. PRIVM values 0b101 and 0b110 are not supported because a storage location that is shared between the hypervisor and non-hypervisor software is unlikely to be accessed using the same EA by both the hypervisor and the non-hypervisor software. (PRIVM value 0b111 is supported primarily for reasons of software compatibility, as described in a subsequent Programming Note.)

A Data Address Breakpoint match occurs for a Load or Store instruction if, for any byte accessed, all of the following conditions are satisfied.

- $EA_{0:60} = DABR_{DAB}$
- $(MSR_{DR} = DABR_{BT}) \mid DABRX_{BTI}$
- if the processor is in
  - hypervisor state and $DABRX_{HYP} = 1$ or
  - privileged but non-hypervisor state and $DABRX_{PNH} = 1$ or
  - problem state and $DABRX_{PR} = 1$
- the instruction is a *Store* and $DABR_{DW} = 1$, or the instruction is a *Load* and $DABR_{DR} = 1$.

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

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

If the above conditions are satisfied, it is undefined whether a match occurs in the following cases.

- The instruction is *Store Conditional* but the store is not performed.
- The instruction is a *Load/Store String* of zero length.
- The instruction is ***dcbz***. (For the purpose of determining whether a match occurs, ***dcbz*** is treated as a *Store*.)

The *Cache Management* instructions other than ***dcbz*** never cause a match.

A Data Address Breakpoint match causes a Data Storage exception (see Section 5.5.3, “Data Storage Interrupt” on page 66). If a match occurs, some or all of the bytes of the storage operand may have been accessed; however, if a *Store* or ***ecowx*** instruction causes the match, the storage operand is not modified if the instruction is one of the following:

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

Programming Note

The Data Address Breakpoint mechanism does not apply to instruction fetches.

Programming Note

Before setting a breakpoint requested by the operating system, the hypervisor must verify that the requested contents of the DABR and DABRX cannot cause the hypervisor to receive a Data Storage interrupt that it is not prepared to handle, or that it intrinsically cannot handle (e.g., the EA is in the range of EAs at which the hypervisor's Data Storage interrupt handler saves registers, $DABR_{BT}$ || $DABRX_{BTI} \neq 0b10$, $DABR_{DW} = 1$, and $DABRX_{HYP} = 1$).

Programming Note

Processors that comply with versions of the architecture that precede Version 2.02 do not provide the DABRX. Forward compatibility for software that was written for such processors (and uses the Data Address Breakpoint facility) can be obtained by setting $DABRX_{60:63}$ to 0b0111.

4.8 Storage Control Bits

When address translation is enabled, each storage access is performed under the control of the Page Table Entry used to translate the effective address. Each Page Table Entry contains storage control bits that specify the presence or absence of the corresponding storage control attribute (see the section entitled “Storage Control Attributes” in Book II, *PowerPC Virtual Environment Architecture*) for all accesses translated by the entry, as shown in Figure 28. The bits are called W, I, M, and G.

Bit	Storage Control Attribute
W[1]	0 - not Write Through Required 1 - Write Through Required
I	0 - not Caching Inhibited 1 - Caching Inhibited
M[2]	0 - not Memory Coherence Required 1 - Memory Coherence Required
G	0 - not Guarded 1 - Guarded

1. Support for the 1 value of the W bit is optional. Implementations that do not support the 1 value treat the bit as reserved and assume its value to be 0.
2. Support for the 0 value of the M bit is optional. Implementations that do not support the 0 value assume the value of the bit to be 1, and may either preserve the value of the bit or write it as 1.

Figure 28. Storage control bits

When address translation is enabled, instructions are not fetched from storage for which the G bit in the Page Table Entry is set to 1 (see Section 4.10, “Storage Protection” on page 45)

When address translation is disabled, the storage control attributes are implicit; see Section 4.2.6.3, “Storage Control Attributes for Real Addressing Mode and for Implicit Storage Accesses” on page 30.

In Section 4.8.1 and 4.8.2, "access" includes accesses that are performed out-of-order, and references to W, I, M, and G bits include the values of those bits that are implied when address translation is disabled.

Programming Note

In a uniprocessor system in which only the processor has caches, correct coherent execution does not require the processor to access storage as Memory Coherence Required, and accessing storage as not Memory Coherence Required may give better performance.

4.8.1 Storage Control Bit Restrictions

All combinations of W, I, M, and G values are supported except those for which both W and I are 1.

Programming Note

If an application program requests both the Write Through Required and the Caching Inhibited attributes for a given storage location, the operating system should set the I bit to 1 and the W bit to 0.

At any given time, the value of the I bit must be the same for all accesses to a given real page.

At any given time, the value of the W bit must be the same for all accesses to a given real page.

4.8.2 Altering the Storage Control Bits

When changing the value of the I bit for a given real page from 0 to 1, software must set the I bit to 1 and then flush all copies of locations in the page from the caches using ***dcbf***[***l***] and ***icbi*** before permitting any other accesses to the page.

When changing the value of the W bit for a given real page from 0 to 1, software must ensure that no processor modifies any location in the page until after all copies of locations in the page that are considered to be modified in the data caches have been copied to main storage using ***dcbst*** or ***dcbf***[***l***].

Programming Note

It is recommended that ***dcbf*** be used, rather than ***dcbfl***, when changing the value of the I or W bit from 0 to 1. (***dcbfl*** would have to be executed on all processors for which the contents of the data cache may be inconsistent with the new value of the bit, whereas, if the M bit for the page is 1, ***dcbf*** need be executed on only one processor in the system.)

When changing the value of the M bit for a given real page, software must ensure that all data caches are consistent with main storage. The actions required to do this to are system-dependent.

Programming Note

For example, when changing the M bit in some directory-based systems, software may be required to execute ***dcbf***[***l***] instructions on each processor to flush all storage locations accessed with the old M value before permitting the locations to be accessed with the new M value.

Additional requirements for changing the storage control bits in the Page Table are given in see Section 4.12, “Page Table Update Synchronization Requirements” on page 57.

4.9 Reference and Change Recording

If address translation is enabled, Reference (R) and Change (C) bits are maintained in the Page Table Entry that is used to translate the virtual address. If the storage operand of a *Load* or *Store* instruction crosses a virtual page boundary, the accesses to the components of the operand in each page are treated as separate and independent accesses to each of the pages for the purpose of setting the Reference and Change bits.

Reference and Change bits are set by the processor as described below. Setting the bits need not be atomic with respect to performing the access that caused the bits to be updated. An attempt to access storage may cause one or more of the bits to be set (as described below) even if the access is not performed. The bits are updated in the Page Table Entry if the new value would otherwise be different from the old, as determined by examining either the Page Table Entry or any corresponding lookaside information maintained by the processor (e.g., in a TLB).

Reference Bit

The Reference bit is set to 1 if the corresponding access (load, store, or instruction fetch) is required by the sequential execution model and is performed. Otherwise the Reference bit may be set to 1 if the corresponding access is attempted, either in-order or out-of-order, even if the attempt causes an exception.

Change Bit

The Change bit is set to 1 if a *Store* instruction is executed and the store is performed. Otherwise the Change bit may be set to 1 if a *Store* instruction is executed and the store is permitted by the storage protection mechanism and, if the *Store* instruction is executed out-of-order, the instruction would be required by the sequential execution model in the absence of the following kinds of interrupts:

- system-caused interrupts (see Section 5.3)
- Floating-Point Enabled Exception type Program interrupts when the processor is in an Imprecise mode

Programming Note

Even though the execution of a *Store* instruction causes the Change bit to be set to 1, the store might not be performed or might be only partially performed in cases such as the following.

- A *Store Conditional* instruction (***stwcx.*** or ***stdcx.***) is executed, but no store is performed.
- A *Store String Word Indexed* instruction (***stswx***) is executed, but the length is zero.
- The *Store* instruction causes a Data Storage exception (for which setting the Change bit is not prohibited).
- The *Store* instruction causes an Alignment exception.
- The Page Table Entry that translates the virtual address of the storage operand is altered such that the new contents of the Page Table Entry preclude performing the store (e.g., the PTE is made invalid, or the PP bits are changed).

  For example, when executing a *Store* instruction, the processor may search the Page Table for the purpose of setting the Change bit and then reexecute the instruction. When reexecuting the instruction, the processor may search the Page Table a second time. If the Page Table Entry has meanwhile been altered, by a program executing on another processor, the second search may obtain the new contents, which may preclude the store.
- A system-caused interrupt occurs before the store has been performed.

Figure 29 on page 44 summarizes the rules for setting the Reference and Change bits. The table applies to each atomic storage reference. It should be read from the top down; the first line matching a given situation applies. For example, if ***stwcx.*** fails due to both a storage protection violation and the lack of a reservation, the Change bit is not altered.

In the figure, the "*Load*-type" instructions are the *Load* instructions described in Books I and II, ***eciwx***, and the *Cache Management* instructions that are treated as *Loads*. The "*Store*-type" instructions are the *Store* instructions described in Books I and II, ***ecowx***, and the *Cache Management* instructions that are treated as *Stores*. The "ordinary" *Load* and *Store* instructions are those described in Books I and II. "set" means "set to 1".

When the processor updates the Reference and Change bits in the Page Table Entry, the accesses are performed as described in Section 4.2.6.3, "Storage Control Attributes for Real Addressing Mode and for Implicit Storage Accesses" on page 30. The accesses

may be performed using operations equivalent to a store to a byte, halfword, word, or doubleword, and are not necessarily performed as an atomic read/modify/write of the affected bytes.

These Reference and Change bit updates are not necessarily immediately visible to software. Executing a ***sync*** instruction ensures that all Reference and Change bit updates associated with address translations that were performed, by the processor executing the ***sync*** instruction, before the ***sync*** instruction is executed will be performed with respect to that processor before the ***sync*** instruction's memory barrier is created. There are additional requirements for synchronizing Reference and Change bit updates in multiprocessor systems; see Section 4.12, "Page Table Update Synchronization Requirements" on page 57.

Programming Note

Because the ***sync*** instruction is execution synchronizing, the set of Reference and Change bit updates that are performed with respect to the processor executing the ***sync*** instruction before the memory barrier is created includes all Reference and Change bit updates associated with instructions preceding the ***sync*** instruction.

If software refers to a Page Table Entry when MSR_{DR}=1, the Reference and Change bits in the associated Page Table Entry are set as for ordinary loads and stores. See Section 4.12 for the rules software must follow when updating Reference and Change bits.

Status of Access	R	C
Storage protection violation	Acc[1]	No
Out-of-order I-fetch or *Load*-type insn	Acc	No
Out-of-order *Store*-type insn		
Would be required by the sequential execution model in the absence of system-caused or imprecise interrupts[3]	Acc	Acc[1 2]
All other cases	Acc	No
In-order *Load*-type or *Store*-type insn, access not performed		
Load-type insn	Acc	No
Store-type insn	Acc	Acc[2]
Other in-order access		
I-fetch	Yes	No
Ordinary *Load*, ***eciwx***	Yes	No
Other ordinary *Store*, ***ecowx***, ***dcbz***	Yes	Yes
icbi, ***dcbt***, ***dcbtst***, ***dcbst***, ***dcbf***[***l***]	Acc	No

"Acc" means that it is acceptable to set the bit.
1 It is preferable not to set the bit.
2 If C is set, R is also set unless it is already set.
3 For Floating-Point Enabled Exception type Program interrupts, "imprecise" refers to the exception mode controlled by $MSR_{FE0\ FE1}$.

Figure 29. Setting the Reference and Change bits

4.10 Storage Protection

The storage protection mechanism provides a means for selectively granting instruction fetch access, granting read access, granting read/write access, and prohibiting access to areas of storage based on a number of control criteria.

The operation of the protection mechanism depends on whether address translation is enabled or disabled .

If an instruction fetch is not permitted by the protection mechanism, an Instruction Storage exception is generated. If a data access is not permitted by the protection mechanism, a Data Storage exception is generated. (See Section 4.2.1, “Storage Exceptions” on page 26)

When address translation is enabled, a *protection domain* is a range of unmapped effective addresses, a virtual page, or a segment. When address translation is disabled and $LPES_1$=1 there are two protection domains: the set of effective addresses that are less than the value specified by the RMLR, and all other effective addresses. When address translation is disabled and $LPES_1$=0 the entire effective address space comprises a single protection domain. A *protection boundary* is a boundary between protection domains.

4.10.1 Storage Protection, Address Translation Enabled

When address translation is enabled , the protection mechanism is controlled by the following.

- MSR_{PR}, which distinguishes between supervisor (privileged) state and problem state
- K_s and K_p, the supervisor (privileged) state and problem state storage key bits in the SLB entry used to translate the effective address
- PP, page protection bits in the Page Table Entry used to translate the effective address
- For instruction fetches only:
  - the N (No-execute) value used for the access (see Section 4.5.3)
  - PTE_G, the G (Guarded) bit in the Page Table Entry used to translate the effective address

Using the above values, the following rules are applied.

1. For an instruction fetch, the access is not permitted if the N value is 1 or if PTE_G=1.
2. For any access except an instruction fetch that is not permitted by rule 1, a “Key” value is computed using the following formula:

   $$Key \leftarrow (K_p \ \&\ MSR_{PR}) \mid (K_s \ \&\ \neg MSR_{PR})$$

   Using the computed Key, Figure 30 is applied. An instruction fetch is permitted for any entry in the figure except “no access”. A load is permitted for any entry except “no access”. A store is permitted only for entries with “read/write”.

Key	PP	Access Authority
0	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

Figure 30. PP bit protection states, address translation enabled

4.10.2 Storage Protection, Address Translation Disabled

When address translation is disabled, the protection mechanism is controlled by the following (see Section 1.7, “Logical Partitioning (LPAR)” on page 5 and Section 4.2.6, “Real Addressing Mode” on page 29).

- $LPES_1$, which distinguishes between the two modes of accessing storage using the LPAR facility
- MSR_{HV}, which distinguishes between hypervisor state and other privilege states
- RMLR, which specifies the real mode limit value

Using the above values, Figure 31 is applied. The access is permitted for any entry in the figure except “no access”.

$LPES_1$	HV	Access Authority
0	0	no access
0	1	read/write
1	0	read/write or no access[1]
1	1	read/write

1. If the effective address for the access is less than the value specified by the RMLR the access authority is read/write; otherwise the access is not permitted.

Figure 31. Protection states, address translation disabled

Programming Note

The comparison described in note 1 in Figure 31 ignores bits 0:1 of the effective address and may ignore bits 2:63-m; see Section 4.2.6.

4.11 Storage Control Instructions

4.11.1 Cache Management Instructions

This section describes aspects of cache management that are relevant only to privileged software programmers.

For a ***dcbz*** instruction that causes the target block to be newly established in the data cache without being fetched from main storage, the processor need not verify that the associated real address is valid. The existence of a data cache block that is associated with an invalid real address (see Section 4.2.8) can cause a delayed Machine Check interrupt or a delayed Checkstop.

Each implementation provides an efficient means by which software can ensure that all blocks that are considered to be modified in the data cache have been copied to main storage before the processor enters any power conserving mode in which data cache contents are not maintained. The means are described in the Book IV, *PowerPC Implementation Features* document for the implementation.

4.11.2 Synchronize Instruction

The *Synchronize* instruction is described in Book II, *PowerPC Virtual Environment Architecture*, but only at the level required by an application programmer (***sync*** with L=0 or L=1). This section describes properties of the instruction that are relevant only to operating system and hypervisor software programmers. This variant of the *Synchronize* instruction is designated the page table entry ***sync*** and is specified by the extended mnemonic ***ptesync*** (equivalent to ***sync*** with L=2).

The ***ptesync*** instruction has all of the properties of ***sync*** with L=0 and also the following additional properties.

- The memory barrier created by the ***ptesync*** instruction provides an ordering function for the storage accesses associated with all instructions that are executed by the processor executing the ***ptesync*** instruction and, as elements of set A, for all Reference and Change bit updates associated with additional address translations that were performed, by the processor executing the ***ptesync*** instruction, before the ***ptesync*** instruction is executed. The applicable pairs are all pairs a_i,b_j in which b_j is a data access and a_i is not an instruction fetch.
- The ***ptesync*** instruction causes all Reference and Change bit updates associated with address translations that were performed, by the processor executing the ***ptesync*** instruction, before the ***ptesync*** instruction is executed, to be performed with respect to that processor before the ***ptesync*** instruction's memory barrier is created.
- The ***ptesync*** instruction provides an ordering function for all stores to the Page Table caused by Store instructions preceding the ***ptesync*** instruction with respect to searches of the Page Table that are performed, by the processor executing the ***ptesync*** instruction, after the ***ptesync*** instruction completes. Executing a ***ptesync*** instruction ensures that all such stores will be performed, with respect to the processor executing the ***ptesync*** instruction, before any implicit accesses to the affected Page Table Entries, by such Page Table searches, are performed with respect to that processor.
- In conjunction with the ***tlbie*** and ***tlbsync*** instructions, the ***ptesync*** instruction provides an ordering function for TLB invalidations and related storage accesses on other processors as described in the ***tlbsync*** instruction description on page 56.

Programming Note

For instructions following a ***ptesync*** instruction, the memory barrier need not order implicit storage accesses for purposes of address translation and reference and change recording.

The functions performed by the ***ptesync*** instruction may take a significant amount of time to complete, so this form of the instruction should be used only if the functions listed above are needed. Otherwise ***sync*** with L=0 should be used (or ***sync*** with L=1 or ***eieio***, if appropriate).

Section 4.12, "Page Table Update Synchronization Requirements" on page 57 gives examples of uses of ***ptesync***.

4.11.3 Lookaside Buffer Management

All implementations have a Segment Lookaside Buffer (SLB), and provide the SLB Management instructions described in Section 4.11.3.1.

For performance reasons, most implementations have a Translation Lookaside Buffer (TLB), which is a cache of recently used Page Table Entries (PTEs). The TLB is not necessarily kept consistent with the Page Table

in main storage. When software alters the contents of a PTE, it must also invalidate all corresponding TLB entries.

Each implementation that has a TLB provides a means by which software can do the following.

- Invalidate the TLB entry that translates a given effective address
- Invalidate all TLB entries

An implementation may provide one or more of the *TLB Management* instructions described in Section 4.11.3.2 in order to satisfy requirements in the preceding list. Alternatively, an algorithm may be given 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. Such algorithms or instructions are described in Book IV, *PowerPC Implementation Features*. Because most implementations have a TLB and also provide instructions similar or identical to the *TLB Management* instructions described in Section 4.11.3.2, other sections of the Books assume that the TLB exists and that the instructions described in Section 4.11.3.2 are provided.

An implementation that does not have a TLB treats the corresponding instructions (***tlbie***, ***tlbiel***, ***tlbia***, and ***tlbsync***) either as no-ops or as illegal instructions.

Programming Note

Because the presence, absence, and exact semantics of the *TLB Management* instructions are implementation-dependent, it is recommended that system software “encapsulate” uses of these instructions into subroutines to minimize the impact of moving from one implementation to another.

Programming Note

The function of all the instructions described in Sections 4.11.3.1 and 4.11.3.2 is independent of whether address translation is enabled or disabled.

For a discussion of software synchronization requirements when invalidating SLB and TLB entries, see Chapter 7. “Synchronization Requirements for Context Alterations” on page 85.

For performance reasons, some implementations have implementation-specific lookaside information used in address translation, such as a set of recently-used translations of effective addresses to real addresses. Because this information may include "translations" that apply in real addressing mode, and such "translations" are affected by the contents of the LPCR, RMOR, and HRMOR, when software modifies the contents of these registers it must also invalidate the corresponding implementation-specific lookaside information.

Each implementation that has such implementation-specific lookaside information provides a means by which software can do the following.

- Invalidate all implementation-specific lookaside information used in converting effective addresses to real addresses.

4.11.3.1 SLB Management Instructions

Programming Note

Accesses to a given SLB entry caused by the instructions described in this section obey the sequential execution model with respect to the contents of the entry and with respect to data dependencies on those contents. That is, if an instruction sequence contains two or more of these instructions, when the sequence has completed, the final state of the SLB entry and of General Purpose Registers is as if the instructions had been executed in program order.

However, software synchronization is required in order to ensure that any alterations of the entry take effect correctly with respect to address translation; see Chapter 7.

SLB Invalidate Entry X-form

slbie RB

31	///	///	RB	434	/
0	6	11	16	21	31

```
esid ← (RB)0:35
class ← (RB)36
if class = SLBEC for SLB entry that translates
  or most recently translated esid
 then for SLB entry (if any) that translates esid
  SLBEV ← 0
  all other fields of SLBE ← undefined
 else translation of esid ← undefined
```

Let the Effective Segment ID (ESID) be $(RB)_{0:35}$. Let the class be $(RB)_{36}$. The class value must be the same as the Class value in the SLB entry that translates the ESID, or the Class value that was in the SLB entry that most recently translated the ESID if the translation is no longer in the SLB; if the class value is not the same, the results of translating effective addresses for which $EA_{0:35}$=ESID are undefined, and the next paragraph need not apply.

If the SLB contains an entry that translates the specified ESID, the V bit in that entry is set to 0, making the entry invalid, and the remaining fields of the entry are set to undefined values.

$(RB)_{37:63}$ must be zeroes.

If this instruction is executed in 32-bit mode, $(RB)_{0:31}$ must be zeros (i.e., the ESID must be in the range 0-15).

This instruction is privileged.

Special Registers Altered:

None

Programming Note

The only SLB entry that is invalidated is the entry (if any) that translates the specified ESID.

slbie does not affect SLBs on other processors.

Programming Note

The reason the class value specified by ***slbie*** must be the same as the Class value that is or was in the relevant SLB entry is that the processor may use these values to optimize invalidation of implementation-specific lookaside information used in address translation. If the value specified by ***slbie*** differs from the value that is or was in the relevant SLB entry, these optimizations may produce incorrect results. (An example of implementation-specific address translation lookaside information is the set of recently used translations of effective addresses to real addresses that some processors maintain in an Effective to Real Address Translation (ERAT) lookaside buffer.)

The recommended use of the Class field is to classify SLB entries according to the expected longevity of the translations they contain, or a similar property such as whether the translations are used by all programs or only by a single program. If this is done and the processor invalidates certain implementation-specific lookaside information based only on the specified class value, an ***slbie*** instruction that invalidates a short-lived translation will preserve such lookaside information for long-lived translations.

If the optional “Bridge” facility is implemented (see Section 9.1), the *Move To Segment Register* instructions create SLB entries in which the Class value is 0.

SLB Move To Entry X-form

slbmte RS,RB

31	RS	///	RB	402	/
0	6	11	16	21	31

The SLB entry specified by bits 52:63 of register RB is loaded from register RS and from the remainder of register RB. The contents of these registers are interpreted as shown in Figure 32.

RS

VSID	K_sK_pNLC	0s
0	52	57 63

RB

ESID	V	0s	index
0	36 37	52	63

$RS_{0:51}$ VSID
RS_{52} K_s
RS_{53} K_p
RS_{54} N
RS_{55} L
RS_{56} C
$RS_{57:63}$ must be 0b000_0000

$RB_{0:35}$ ESID
RB_{36} V
$RB_{37:51}$ must be 0b000 || 0x000
$RB_{52:63}$ index, which selects the SLB entry

Figure 32. GPR contents for slbmte

On implementations that support a virtual address size of only n bits, n<80, $(RS)_{0:79-n}$ must be zeros.

High-order bits of $(RB)_{52:63}$ that correspond to SLB entries beyond the size of the SLB provided by the implementation must be zeros.

If this instruction is executed in 32-bit mode, $(RB)_{0:31}$ must be zeros (i.e., the ESID must be in the range 0-15).

This instruction cannot be used to invalidate an SLB entry.

This instruction is privileged.

Special Registers Altered:
None

Programming Note

The reason ***slbmte*** cannot be used to invalidate an SLB entry is that it does not necessarily affect implementation-specific address translation lookaside information. ***slbie*** (or ***slbia***) must be used for this purpose.

SLB Move From Entry VSID X-form

slbmfev RT,RB

31	RT	///	RB	851	/
0	6	11	16	21	31

If the SLB entry specified by bits 52:63 of register RB is valid (V=1), the contents of the VSID, K_s, K_p, N, L, and C fields of the entry are placed into register RT. The contents of these registers are interpreted as shown in Figure 33.

RT

VSID	K_sK_pNLC	0s
0	52	57 63

RB

0s	index
0	52 63

$RT_{0:51}$ VSID
RT_{52} K_s
RT_{53} K_p
RT_{54} N
RT_{55} L
RT_{56} C
$RT_{57:63}$ set to 0b000_0000

$RB_{0:51}$ must be 0x0_0000_0000_0000
$RB_{52:63}$ index, which selects the SLB entry

Figure 33. GPR contents for slbmfev

On implementations that support a virtual address size of only n bits, n<80, $RT_{0:79-n}$ are set to zeros.

If the SLB entry specified by bits 52:63 of register RB is invalid (V=0), the contents of register RT are undefined.

High-order bits of $(RB)_{52:63}$ that correspond to SLB entries beyond the size of the SLB provided by the implementation must be zeros.

This instruction is privileged.

Special Registers Altered:
None

SLB Move From Entry ESID X-form

slbmfee RT,RB

31	RT	///	RB	915	/
0	6	11	16	21	31

If the SLB entry specified by bits 52:63 of register RB is valid (V=1), the contents of the ESID and V fields of the entry are placed into register RT. The contents of these registers are interpreted as shown in Figure 34.

RT

ESID	V	0s
0	36 37	63

RB

0s	index
0	52 63

$RT_{0:35}$ ESID
RT_{36} V
$RT_{37:63}$ set to 0b000 || 0x00_0000

$RB_{0:51}$ must be 0x0_0000_0000_0000
$RB_{52:63}$ index, which selects the SLB entry

Figure 34. GPR contents for slbmfee

If the SLB entry specified by bits 52:63 of register RB is invalid (V=0), RT_{36} is set to 0 and the contents of $RT_{0:35}$ and $RT_{37:63}$ are undefined.

High-order bits of $(RB)_{52:63}$ that correspond to SLB entries beyond the size of the SLB provided by the implementation must be zeros.

This instruction is privileged.

Special Registers Altered:
None

4.11.3.2 TLB Management Instructions (Optional)

TLB Invalidate Entry X-form

tlbie RB,L
[POWER mnemonic: tlbi]

31	///	L	///	RB	306	/
0	6	10	11	16	21	31

```
if L = 0
  then pg_size ← 4 KB
  else pg_size ← large page size
p ← log_base_2(pg_size)
for each processor in the partition
  for each TLB entry
    if (entry_VPN32:79-p = (RB)16:63-p) &
       (entry_pg_size = pg_size)     then TLB entry
← invalid
```

The contents of $(RB)_{0:15}$ must be 0x0000. If the L field of the instruction is 1 let the page size be large; otherwise let the page size be 4 KB.

All TLB entries that have all of the following properties are made invalid on all processors that are in the same partition as the processor executing the ***tlbie*** instruction.

- The entry translates a virtual address for which $VPN_{32:79-p}$ is equal to $(RB)_{16:63-p}$.
- The page size of the entry matches the page size specified by the L field of the instruction.

Additional TLB entries may also be made invalid on any processor that is in the same partition as the processor executing the ***tlbie*** instruction.

MSR_{SF} must be 1 when this instruction is executed; otherwise the results are undefined.

The operation performed by this instruction is ordered by the ***eieio*** (or ***sync*** or ***ptesync***) instruction with respect to a subsequent ***tlbsync*** instruction executed by the processor executing the ***tlbie*** instruction. The operations caused by ***tlbie*** and ***tlbsync*** are ordered by ***eieio*** as a fourth set of operations, which is independent of the other three sets that ***eieio*** orders.

This instruction is privileged, and can be executed only in hypervisor state. If it is executed in privileged but non-hypervisor state either a Privileged Instruction type Program interrupt occurs or the results are boundedly undefined.

This instruction is optional.

See Section 4.12, “Page Table Update Synchronization Requirements” for a description of other requirements associated with the use of this instruction.

Special Registers Altered:

None

TLB Invalidate Entry Local X-form

tlbiel RB,L

31	///	L	///	RB	274	/
0	6	10	11	16	21	31

```
if L = 0
  then pg_size ← 4 KB
  else pg_size ← large page size
p ← log_base_2(pg_size)
for each TLB entry
  if (entry_VPN32:79-p = (RB)16:63-p) &
    (entry_pg_size = pg_size)  then TLB entry ←
invalid
```

The contents of $(RB)_{0:15}$ must be 0x0000. If the L field of the instruction is 1 let the page size be large; otherwise let the page size be 4 KB.

All TLB entries that have all of the following properties are made invalid on the processor executing the ***tlbiel*** instruction.

- The entry translates a virtual address for which $VPN_{32:79-p}$ is equal to $(RB)_{16:63-p}$.
- The page size of the entry matches the page size specified by the L field of the instruction.

Only TLB entries on the processor executing the ***tlbiel*** instruction are affected.

$(RB)_{52:63}$ must be zero.

MSR_{SF} must be 1 when this instruction is executed; otherwise the results are undefined.

This instruction is privileged, and can be executed only in hypervisor state. If it is executed in privileged but non-hypervisor state either a Privileged Instruction type Program interrupt occurs or the results are boundedly undefined.

This instruction is optional.

See Section 4.12, “Page Table Update Synchronization Requirements” on page 57 for a description of other requirements associated with the use of this instruction.

Special Registers Altered:

None

Programming Note

The primary use of this instruction by hypervisor state code is to invalidate TLB entries prior to reassigning a processor to a new logical partition.

tlbiel may be executed on a given processor even if the sequence of ***tlbie*** - ***sync*** - ***tlbsync*** - ***ptesync*** is being concurrently executed on a different processor. In other words, no programmatic synchronization is required relative to the execution of ***tlbie*** or ***tlbiel***.

To synchronize the completion of this processor local form of ***tlbie***, only a ***ptesync*** is required (***tlbsync*** should not be used).

TLB Invalidate All X-form

tlbia

31	///	///	///	370	/
0	6	11	16	21	31

```
all TLB entries ← invalid
```

All TLB entries are made invalid on the processor executing the ***tlbia*** instruction.

This instruction is privileged, and can be executed only in hypervisor state. If it is executed in privileged but non-hypervisor state either a Privileged instruction type Program interrupt occurs or the results are boundedly undefined.

This instruction is optional.

Special Registers Altered:

None

Programming Note

tlbia does not affect TLBs on other processors.

TLB Synchronize X-form

tlbsync

31	///	///	///	566	/
0	6	11	16	21	31

The ***tlbsync*** instruction provides an ordering function for the effects of all ***tlbie*** instructions executed by the processor executing the ***tlbsync*** instruction, with respect to the memory barrier created by a subsequent ***ptesync*** instruction executed by the same processor. Executing a ***tlbsync*** instruction ensures that all of the following will occur.

- All TLB invalidations caused by ***tlbie*** instructions preceding the ***tlbsync*** instruction will have completed on any other processor before any data accesses caused by instructions following the ***ptesync*** instruction are performed with respect to that processor.
- All storage accesses by other processors for which the address was translated using the translations being invalidated, and all Reference and Change bit updates associated with address translations that were performed by other processors using the translations being invalidated, will have been performed with respect to the processor executing the ***ptesync*** instruction, to the extent required by the associated Memory Coherence Required attributes, before the ***ptesync*** instruction's memory barrier is created.

The operation performed by this instruction is ordered by the ***eieio*** (or ***sync*** or ***ptesync***) instruction with respect to preceding ***tlbie*** instructions executed by the processor executing the ***tlbsync*** instruction. The operations caused by ***tlbie*** and ***tlbsync*** are ordered by ***eieio*** as a fourth set of operations, which is independent of the other three sets that ***eieio*** orders.

The ***tlbsync*** instruction may complete before operations caused by ***tlbie*** instructions preceding the ***tlbsync*** instruction have been performed.

This instruction is privileged, and can be executed only in hypervisor state. If it is executed in privileged but non-hypervisor state either a Privileged Instruction type Program interrupt occurs or the results are boundedly undefined.

This instruction is optional.

See Section 4.12, "Page Table Update Synchronization Requirements" on page 57 for a description of other requirements associated with the use of this instruction.

Special Registers Altered:

None

Programming Note

tlbsync should not be used to synchronize the completion of ***tlbiel***.

4.12 Page Table Update Synchronization Requirements

This section describes rules that software should follow when updating the Page Table, and includes suggested sequences of operations for some representative cases.

In the sequences of operations shown in the following subsections, any alteration of a Page Table Entry (PTE) that corresponds to a single line in the sequence is assumed to be done using a *Store* instruction for which the access is atomic. Appropriate modifications must be made to these sequences if this assumption is not satisfied (e.g., if a store doubleword operation is done using two *Store Word* instructions).

All of the sequences require a context synchronizing operation after the sequence if the new contents of the PTE are to be used for address translations associated with subsequent instructions.

As noted in the description of the *Synchronize* instruction in Book II, address translation associated with instructions which occur in program order subsequent to the *Synchronize* (and this includes the ***ptesync*** variant) may actually be performed prior to the completion of the *Synchronize*. To ensure that these instructions and data which may have been speculatively fetched are discarded, a context synchronizing operation is required.

Programming Note

The context synchronizing operation after the sequence ensures that any address translations associated with instructions following the context synchronizing operation that were performed using the old contents of the PTE will be discarded, with the result that these address translations will be performed again using the values stored by the sequence (or values stored subsequently). In many cases this context synchronization will occur naturally; for example, if the sequence is executed within an interrupt handler the ***rfid*** or ***hrfid*** instruction that returns from the interrupt handler may provide the required context synchronization.

No context synchronizing operation is needed before any of the sequences, because (a) each sequence begins with a store to the PTE, (b) no context synchronizing operation is needed before the corresponding *Store* instruction (see Note 8 of Chapter 7 on page 87), and (c) each sequence (except the sequence for resetting the Reference bit) explicitly orders subsequent operations with respect to the store. These properties ensure that all address translations associated with instructions preceding the sequence will be performed using the old contents of the PTE.

Page Table Entries must not be changed in a manner that causes an implicit branch.

4.12.1 Page Table Updates

TLBs are non-coherent caches of the HTAB. TLB entries must be invalidated explicitly with one of the *TLB Invalidate* instructions.

Unsynchronized lookups in the HTAB continue even while it is being modified. Any processor, including a processor on which software is modifying the HTAB, may look in the HTAB at any time in an attempt to translate a virtual address. When modifying a PTE, software must ensure that the PTE's Valid bit is 0 if the PTE is inconsistent (e.g., if the RPN field is not correct for the current AVPN field).

Updates of Reference and Change bits by the processor are not synchronized with the accesses that cause the updates. When modifying 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.

Before permitting one or more ***tlbie*** instructions to be executed on a given processor in a given partition software must ensure that no other processor will execute a "conflicting instruction" until after the following sequence of instructions has been executed on the given processor.

the ***tlbie*** instruction(s)
eieio
tlbsync
ptesync

The "conflicting instructions" in this case are the following.

- a ***tlbie*** or ***tlbsync*** instruction, if executed on another processor in the given partition
- an ***mtspr*** instruction that modifies the LPIDR, if the modification has either of the following properties.
  - The old LPID value (i.e., the contents of the LPIDR just before the ***mtspr*** instruction is executed) is the value that identifies the given partition
  - The new LPID value (i.e., the value specified by the ***mtspr*** instruction) is the value that identifies the given partition

Other instructions (excluding ***mtspr*** instructions that modify the LPIDR as described above, and excluding ***tlbie*** instructions except as shown) may be interleaved with the instruction sequence shown above, but the instructions in the sequence must appear in the order

shown. On uniprocessor systems, the ***eieio*** and ***tlbsync*** instructions can be omitted. Other instructions may be interleaved with this sequence of instructions, but these instructions must appear in the order shown.

Programming Note

The ***eieio*** instruction prevents the reordering of ***tlbie*** instructions previously executed by the processor with respect to the subsequent ***tlbsync*** instruction. The ***tlbsync*** instruction and the subsequent ***ptesync*** instruction together ensure that all storage accesses for which the address was translated using the translations being invalidated, and all Reference and Change bit updates associated with address translations that were performed using the translations being invalidated, will be performed with respect to any processor or mechanism, to the extent required by the associated Memory Coherence Required attributes, before any data accesses caused by instructions following the ***ptesync*** instruction are performed with respect to that processor or mechanism.

The requirements specified above for ***tlbie*** instructions apply also to ***tlbsync*** instructions, except that the "sequence of instructions" consists solely of the ***tlbsync*** instruction(s) followed by a ***ptesync*** instruction.

Before permitting an ***mtspr*** instruction that modifies the LPIDR to be executed on a given processor, software must ensure that no other processor will execute a "conflicting instruction" until after the ***mtspr*** instruction followed by a context synchronizing instruction have been executed on the given processor (a context synchronizing event can be used instead of the context synchronizing instruction; see Chapter 7).

The "conflicting instructions" in this case are the following.

- a ***tlbie*** or ***tlbsync*** instruction, if executed on a processor in either of the following partitions
  - the partition identified by the old LPID value
  - the partition identified by the new LPID value

Programming Note

The restrictions specified above regarding modifying the LPIDR apply even on uniprocessor systems, and even if the new LPID value is equal to the old LPID value.

Similarly, when a ***tlbsync*** instruction has been executed by a processor in a given partition, a ***ptesync*** instruction must be executed by that processor before a ***tlbie*** or ***tlbsync*** instruction is executed by another processor in that partition.

The sequences of operations shown in the following subsections assume a multiprocessor environment. In a uniprocessor environment the ***tlbsync*** can be omitted, as can the ***eieio*** that separates the ***tlbie*** from the ***tlbsync***. In a multiprocessor environment, when ***tlbiel*** is used instead of ***tlbie*** in a Page Table update, the synchronization requirements are the same as when ***tlbie*** is used in a uniprocessor environment.

Programming Note

For all of the sequences shown in the following subsections, if it is necessary to communicate completion of the sequence to software running on another processor, the ***ptesync*** instruction at the end of the sequence should be followed by a *Store* instruction that stores a chosen value to some chosen storage location X. The memory barrier created by the ***ptesync*** instruction ensures that if a *Load* instruction executed by another processor returns the chosen value from location X, the sequence's stores to the Page Table have been performed with respect to that other processor. The *Load* instruction that returns the chosen value should be followed by a context synchronizing instruction in order to ensure that all instructions following the context synchronizing instruction will be fetched and executed using the values stored by the sequence (or values stored subsequently). (These instructions may have been fetched or executed out-of-order using the old contents of the PTE.)

This Note assumes that the Page Table and location X are in storage that is Memory Coherence Required.

4.12.1.1 Adding a Page Table Entry

This is the simplest Page Table case. The Valid bit of the old entry is assumed to be 0. The following sequence can be used to create a PTE, maintain a consistent state, and ensure that a subsequent reference to the virtual address translated by the new entry will use the correct real address and associated attributes.

```
PTE_RPN,AC,R,C,WIMG,N,PP ← new values
eieio     /* order 1st update before 2nd  */
PTE_AVPN,SW,L,H,V ← new values (V=1)
ptesync   /* order updates before next
             Page Table search and before
             next data access.            */
```

4.12.1.2 Modifying a Page Table Entry

General Case

If a valid entry is to be modified and the translation instantiated by the entry being modified is to be invalidated, the following sequence can be used to modify

the PTE, maintain a consistent state, ensure that the translation instantiated by the old entry is no longer available, and ensure that a subsequent reference to the virtual address translated by the new entry will use the correct real address and associated attributes. (The sequence is equivalent to deleting the PTE and then adding a new one; see Sections 4.12.1.3 and 4.12.1.1.)

```
PTE_V ← 0  /* (other fields don't matter)*/
ptesync    /* order update before tlbie and
              before next Page Table search  */
tlbie(old_VPN_32:79-p,old_L) /*invalidate old
      translation                    */
eieio      /* order tlbie before tlbsync     */
tlbsync    /* order tlbie before ptesync     */
ptesync    /* order tlbie, tlbsync and 1st
              update before 2nd update       */
PTE_RPN,AC,R,C,WIMG,N,PP ← new values
eieio      /* order 2nd update before 3rd */
PTE_AVPN,SW,L,H,V ← new values (V=1)
ptesync    /* order 2nd and 3rd updates before
              next Page Table search and
              before next data access       */
```

Resetting the Reference Bit

If the only change being made to a valid entry is to set the Reference bit to 0, a simpler sequence suffices because the Reference bit need not be maintained exactly.

```
oldR ← PTE_R  /* get old R                        */
if oldR = 1 then
  PTE_R ← 0       /* store byte (R=0, other bits
                     unchanged)                 */
  tlbie(VPN_32:79-p,L) /* invalidate entry */
  eieio    /* order tlbie before tlbsync          */
  tlbsync  /* order tlbie before ptesync          */
  ptesync  /* order tlbie, tlbsync, and update
              before next Page Table search
              and before next data access         */
```

Modifying the Virtual Address

If the virtual address translated by a valid PTE is to be modified and the new virtual address hashes to the same two PTEGs as does the old virtual address, the following sequence can be used to modify the PTE, maintain a consistent state, ensure that the translation instantiated by the old entry is no longer available, and ensure that a subsequent reference to the virtual address translated by the new entry will use the correct real address and associated attributes.

```
PTE_AVPN,SW,L,H,V ← new values (V=1)
ptesync    /* order update before tlbie and
              before next Page Table search       */
tlbie(old_VPN_32:79-p,old_L) /* invalidate old
     translation                 */
eieio      /* order tlbie before tlbsync          */
tlbsync    /* order tlbie before ptesync          */
ptesync    /* order tlbie, tlbsync, and update
              before next data access             */
```

To modify the AC, N, or PP bits without overwriting a Reference or Change bit update being performed by the processor or by some other processor, a sequence similar to that shown above can be used except that the first line would be replaced by a ***ptesync*** instruction followed by a loop containing a ***ldarx***/***stdcx.*** pair that emulates an atomic "Compare and Swap" of the low-order doubleword of the PTE. (See the section entitled "Atomic Update Primitives" in Book II, *PowerPC Virtual Environment Architecture* for a description of "Compare and Swap".)

4.12.1.3 Deleting a Page Table Entry

The following sequence can be used to ensure that the translation instantiated by an existing entry is no longer available.

```
PTE_V ← 0  /* (other fields don't matter)         */
ptesync    /* order update before tlbie and
              before next Page Table search  */
tlbie(old_VPN_32:79-p,old_L) /* invalidate old
      translation                 */
eieio      /* order tlbie before tlbsync          */
tlbsync    /* order tlbie before ptesync          */
ptesync    /* order tlbie, tlbsync, and update
              before next data access             */
```

Chapter 5. Interrupts

5.1 Overview

The PowerPC Architecture provides an interrupt mechanism to allow the processor to change state as a result of external signals, errors, or unusual conditions arising in the execution of instructions.

System Reset and Machine Check interrupts are not ordered. All other interrupts are ordered such that only one interrupt is reported, and when it is processed (taken) no program state is lost. Since Save/Restore Registers SRR0 and SRR1 are serially reusable resources used by most interrupts, program state may be lost when an unordered interrupt is taken.

5.2 Interrupt Synchronization

When an interrupt occurs, SRR0 or HSRR0 is set to point to an instruction such that all preceding instructions have completed execution, no subsequent instruction has begun execution, and the instruction addressed by SRR0 or HSRR0 may or may not have completed execution, depending on the interrupt type.

With the exception of System Reset and Machine Check interrupts, all interrupts are context synchronizing as defined in Section 1.6.1, “Context Synchronization” on page 4. System Reset and Machine Check interrupts are context synchronizing if they are recoverable (i.e., if bit 62 of SRR1 is set to 1 by the interrupt). If a System Reset or Machine Check interrupt is not recoverable (i.e., if bit 62 of SRR1 is set to 0 by the interrupt), it acts like a context synchronizing operation with respect to subsequent instructions. That is, a nonrecoverable System Reset or Machine Check interrupt need not satisfy items 1 through 3 of Section 1.6.1, but does satisfy items 4 and 5.

5.3 Interrupt Classes

Interrupts are classified by whether they are directly caused by the execution of an instruction or are caused by some other system exception. Those that are "system-caused" are:

- System Reset
- Machine Check
- External
- Decrementer
- Hypervisor Decrementer

External, Decrementer, and Hypervisor Decrementer interrupts are maskable interrupts. Therefore, software may delay the generation of these interrupts. System Reset and Machine Check interrupts are not maskable.

"Instruction-caused" interrupts are further divided into two classes, *precise* and *imprecise*.

5.3.1 Precise Interrupt

Except for the Imprecise Mode Floating-Point Enabled Exception type Program interrupt, all instruction-caused interrupts are precise.

When the fetching or execution of an instruction causes a precise interrupt, the following conditions exist at the interrupt point.

1. SRR0 addresses either the instruction causing the exception or the immediately following instruction. Which instruction is addressed can be determined from the interrupt type and status bits.
2. An interrupt is generated such that all instructions preceding the instruction causing the exception appear to have completed with respect to the executing processor.
3. The instruction causing the exception may appear not to have begun execution (except for causing the exception), may have been partially executed, or may have completed, depending on the interrupt type.
4. Architecturally, no subsequent instruction has begun execution.

5.3.2 Imprecise Interrupt

This architecture defines one imprecise interrupt, the Imprecise Mode Floating-Point Enabled Exception type Program interrupt.

When an Imprecise Mode Floating-Point Enabled Exception type Program interrupt occurs, the following conditions exist at the interrupt point.

1. SRR0 addresses either the instruction causing the exception or some instruction following that instruction; see Section 5.5.9, "Program Interrupt" on page 70.
2. An interrupt is generated such that all instructions preceding the instruction addressed by SRR0 appear to have completed with respect to the executing processor.
3. The instruction addressed by SRR0 may appear not to have begun execution (except, in some cases, for causing the interrupt to occur), may have been partially executed, or may have completed; see Section 5.5.9.
4. No instruction following the instruction addressed by SRR0 appears to have begun execution.

All Floating-Point Enabled Exception type Program interrupts are maskable using the MSR bits FE0 and FE1. Although these interrupts are maskable, they differ significantly from the other maskable interrupts in that the masking of these interrupts is usually controlled by the application program, whereas the masking of all other maskable interrupts is controlled by either the operating system or the hypervisor.

5.4 Interrupt Processing

Associated with each kind of interrupt is an *interrupt vector*, which contains the initial sequence of instructions that is executed when the corresponding interrupt occurs.

Interrupt processing consists of saving a small part of the processor's state in certain registers, identifying the cause of the interrupt in other registers, and continuing execution at the corresponding interrupt vector location. When an exception exists that will cause an interrupt to be generated and it has been determined that the interrupt will occur, the following actions are performed. The handling of Machine Check interrupts (see Section 5.5.2) differs from the description given below in several respects.

1. SRR0 or HSRR0 is loaded with an instruction address that depends on the type of interrupt; see the specific interrupt description for details.
2. Bits 33:36 and 42:47 of SRR1 or HSRR1 are loaded with information specific to the interrupt type.
3. Bits 0:32, 37:41, and 48:63 of SRR1 or HSRR1 are loaded with a copy of the corresponding bits of the MSR.
4. The MSR is set as shown in Figure 35 on page 65. In particular, MSR bits IR and DR are set to 0, disabling relocation, and MSR bit SF is set to 1, selecting 64-bit mode. The new values take effect

beginning with the first instruction executed following the interrupt.

5. Instruction fetch and execution resumes, using the new MSR value, at the effective address specific to the interrupt type. These effective addresses are shown in Figure 36 on page 65.

Interrupts do not clear reservations obtained with ***lwarx*** or ***ldarx***.

Programming Note

In general, when an interrupt occurs, the following instructions should be executed by the operating system before dispatching a "new" program.

- ***stwcx.*** or ***stdcx.***, to clear the reservation if one is outstanding, to ensure that a ***lwarx*** or ***ldarx*** in the interrupted program is not paired with a ***stwcx.*** or ***stdcx.*** in the "new" program.
- ***sync***, to ensure that all storage accesses caused by the interrupted program will be performed with respect to another processor before the program is resumed on that other processor.
- ***isync*** or ***rfid***, to ensure that the instructions in the "new" program execute in the "new" context.

Programming Note

If a program modifies an instruction that it or another program will subsequently execute and the execution of the instruction causes an interrupt, the state of storage and the content of some processor registers may appear to be inconsistent to the interrupt handler program. For example, this could be the result of one program executing an instruction that causes an Illegal Instruction type Program interrupt just before another instance of the same program stores an *Add Immediate* instruction in that storage location. To the interrupt handler code, it would appear that a processor generated the Program interrupt as the result of executing a valid instruction.

Programming Note

In order to handle Machine Check and System Reset interrupts correctly, the operating system should manage MSR_{RI} as follows.

- In the Machine Check and System Reset interrupt handlers, interpret SRR1 bit 62 (where MSR_{RI} is placed) as:
  - 0: interrupt is not recoverable
  - 1: interrupt is recoverable
- In each interrupt handler, when enough state has been saved that a Machine Check or System Reset interrupt can be recovered from, set MSR_{RI} to 1.
- In each interrupt handler, do the following (in order) just before returning.
  1. Set MSR_{RI} to 0.
  2. Set SRR0 and SRR1 to the values to be used by ***rfid***. The new value of SRR1 should have bit 62 set to 1 (which will happen naturally if SRR1 is restored to the value saved there by the interrupt, because the interrupt handler will not be executing this sequence unless the interrupt is recoverable).
  3. Execute ***rfid***.

For interrupts that set the SRRs other than Machine Check or System Reset, MSR_{RI} can be managed similarly when these interrupts occur within interrupt handlers for other interrupts that set the SRRs.

This Note does not apply to interrupts that set the HSRRs because these interrupts put the processor into hypervisor state, and either do not occur or can be prevented from occurring within interrupt handlers for other interrupts that set the HSRRs.

5.4.1 Hypervisor Interrupts

The execution of some of the more complex instructions may alter the content of HSRR0 and HSRR1 as a side effect of executing the instruction. For hypervisor interrupts, interrupts that use HSRR0 and HSRR1 to save and restore the state of the interrupted program, the following list identifies the set of instruction that when executed will not alter the content of HSRR0 and HSRR1. Any omission of instruction suffixes is significant; e.g., ***add*** is listed but ***add.*** is excluded.

Programming Note

Instructions not listed below may be used after the contents of HSSR0 and HSRR1 have been saved on entry to the interrupt and before they are restored on return to the interrupted program.

1. Branch Instructions

   b[***l***][***a***], ***bc***[***l***][***a***], ***bclr***[***l***], ***bcctr***[***l***]

2. Fixed-Point Load and Store Instructions

   lbz, ***lbzx***, ***lhz***, ***lhzx***, ***lwz***, ***lwzx***, ***ld***, ***ldx***, ***stb***, ***stbx***, ***sth***, ***sthx***, ***stw***, ***stwx***, ***std***, ***stdx***

   Storage operands must be aligned and $MSR_{IR\ DR}$ must be 0b00 (translation disabled). Accessing an unaligned storage operand or translating a virtual address may have the side effect of altering HSRR0 and HSRR1.

3. Arithmetic Instructions

   addi, ***addis***, ***add***, ***subf***, ***neg***

4. Compare Instructions

   cmpi, ***cmp***, ***cmpli***, ***cmpl***

5. Logical and Extend Sign Instructions

   ori, ***oris***, ***xori***, ***xoris***, ***and***, ***or***, ***xor***, ***nand***, ***nor***, ***eqv***, ***andc***, ***orc, extsb***, ***extsh***, ***extsw***

6. Rotate and Shift Instructions

   rldicl, ***rldicr***, ***rldic***, ***rlwinm***, ***rldcl***, ***rldcr***, ***rlwnm***, ***rldimi***, ***rlwimi***, ***sld***, ***slw***, ***srd***, ***srw***

7. Other instructions

   isync

   rfid, ***hrfid***

   mtspr, ***mfspr***, ***mtmsrd***, ***mfmsr***

5.5 Interrupt Definitions

Figure 35 shows all the types of interrupts and the values assigned to the MSR for each. Figure 36 shows the effective address of the interrupt vector for each interrupt type. (Section 4.2.7 on page 31 summarizes all architecturally defined uses of effective addresses, including those implied by Figure 36.)

Interrupt Type	MSR Bit							
	IR	DR	FE0	FE1	EE	RI	ME	HV
System Reset	0	0	0	0	0	0	-	1
Machine Check	0	0	0	0	0	0	0	1
Data Storage	0	0	0	0	0	0	-	m
Data Segment	0	0	0	0	0	0	-	m
Instruction Storage	0	0	0	0	0	0	-	m
Instruction Segment	0	0	0	0	0	0	-	m
External	0	0	0	0	0	0	-	e
Alignment	0	0	0	0	0	0	-	m
Program	0	0	0	0	0	0	-	m
FP Unavailable	0	0	0	0	0	0	-	m
Decrementer	0	0	0	0	0	0	-	m
Hypervisor Decrem'er	0	0	0	0	0	-	-	1
System Call	0	0	0	0	0	0	-	s
Trace	0	0	0	0	0	0	-	m
Performance Monitor	0	0	0	0	0	0	-	m

0 bit is set to 0
1 bit is set to 1
- bit is not altered
m if $LPES_1$=0, set to 1; otherwise not altered
e if $LPES_0$=0, set to 1; otherwise not altered
s if LEV=1 or $LPES/LPES_1$=0, set to 1; otherwise not altered

Settings for Other Bits

Bits BE, FP, PMM, PR, and SE are set to 0.

If the optional Little-Endian facility is implemented (see the section entitled "Little-Endian" in Book I), the bits associated with the facility are set as follows. If the interrupt sets HV to 1, LE is set to 0; otherwise the LE bit is copied from the $LPCR_{ILE}$ bit

Bit SF is set to 1.

Reserved bits are set as if written as 0.

Figure 35. MSR setting due to interrupt

Effective Address[1]	Interrupt Type
00..0000_0100	System Reset
00..0000_0200	Machine Check
00..0000_0300	Data Storage
00..0000_0380	Data Segment
00..0000_0400	Instruction Storage
00..0000_0480	Instruction Segment
00..0000_0500	External
00..0000_0600	Alignment
00..0000_0700	Program
00..0000_0800	Floating-Point Unavailable
00..0000_0900	Decrementer
00..0000_0980	Hypervisor Decrementer
00..0000_0A00	Reserved
00..0000_0B00	Reserved
00..0000_0C00	System Call
00..0000_0D00	Trace
00..0000_0E00	Reserved
00..0000_0E10	Reserved
. . .	. . .
00..0000_0EFF	Reserved
00..0000_0F00	Performance Monitor
00..0000_0F10	Reserved
. . .	. . .
00..0000_0FFF	Reserved

[1] The values in the Effective Address column are interpreted as follows.
- 00...0000_nnnn *means* 0x0000_0000_0000_nnnn

[2] Effective addresses 0x0000_0000_0000_0000 through 0x0000_0000_0000_00FF are used by software and will not be assigned as interrupt vectors.

Figure 36. Effective address of interrupt vector by interrupt type

Programming Note

When address translation is disabled, use of any of the effective addresses that are shown as reserved in Figure 36 risks incompatibility with future implementations.

5.5.1 System Reset Interrupt

If a System Reset exception causes an interrupt that is not context synchronizing or causes the loss of a Machine Check exception or an External exception, or if the state of the processor has been corrupted, the interrupt is not recoverable.

The following registers are set:

SRR0 Set to the effective address of the instruction that the processor would have attempted to execute next if no interrupt conditions were present.

SRR1

- **33:36** Set to 0.
- **42:44** See the Book IV, *PowerPC Implementation Features* document for the implementation.
- **45:47** Set to 0.
- **62** Loaded from bit 62 of the MSR if the processor is in a recoverable state; otherwise set to 0.
- **Others** Loaded from the MSR.

MSR See Figure 35 on page 65.

Execution resumes at effective address 0x0000_0000_0000_0100.

Each implementation provides a means for software to distinguish power-on Reset from other types of System Reset, and describes it in the Book IV.

5.5.2 Machine Check Interrupt

The causes of Machine Check interrupts are implementation-dependent. For example, a Machine Check interrupt may be caused by a reference to a storage location that contains an uncorrectable error or does not exist (see Section 4.2.8, “Invalid Real Address” on page 31), or by an error in the storage subsystem.

Machine Check interrupts are enabled when MSR_{ME}=1. If MSR_{ME}=0 and a Machine Check occurs, the processor enters the Checkstop state. The Checkstop state may also be entered if an access is attempted to a storage location that does not exist (see Section 4.2.8).

Disabled Machine Check (Checkstop State)

When a processor is in Checkstop state, instruction processing is suspended and generally cannot be restarted without resetting the processor. Some implementations may preserve some or all of the internal state of the processor when entering Checkstop state, so that the state can be analyzed as an aid in problem determination.

Enabled Machine Check

If a Machine Check exception causes an interrupt that is not context synchronizing or causes the loss of an External exception, or if the state of the processor has been corrupted, the interrupt is not recoverable.

In some systems, the operating system may attempt to identify and log the cause of the Machine Check.

The following registers are set:

SRR0 Set on a “best effort” basis to the effective address of some instruction that was executing or was about to be executed when the Machine Check exception occurred. For further details see the Book IV, *PowerPC Implementation Features* document for the implementation.

SRR1

- **62** Loaded from bit 62 of the MSR if the processor is in a recoverable state; otherwise set to 0.
- **Others** See Book IV.

MSR See Figure 35.

DSISR See Book IV.

DAR See Book IV.

Execution resumes at effective address 0x0000_0000_0000_0200.

Programming Note

If a Machine Check interrupt is caused by an error in the storage subsystem, the storage subsystem may return incorrect data, which may be placed into registers. This corruption of register contents may occur even if the interrupt is recoverable.

5.5.3 Data Storage Interrupt

A Data Storage interrupt occurs when no higher priority exception exists and a data access cannot be performed for any of the following reasons.

- Data address translation is enabled (MSR_{DR}=1) and the virtual address of any byte of the storage location specified by a *Load*, *Store*, ***icbi***, ***dcbz***, ***dcbst***, ***dcbf***[***l***], ***eciwx***, or ***ecowx*** instruction cannot be translated to a real address.
- The effective address specified by a ***lwarx***, ***ldarx***, ***stwcx.***, or ***stdcx.*** instruction refers to storage that is Write Through Required or Caching Inhibited.
- The access violates storage protection.
- A Data Address Compare match or a Data Address Breakpoint match occurs.
- Execution of an ***eciwx*** or ***ecowx*** instruction is disallowed because EA_{RE}=0.

If a ***stwcx.*** or ***stdcx.*** would not perform its store in the absence of a Data Storage interrupt, and either (a) the specified effective address refers to storage that is Write Through Required or Caching Inhibited, or (b) a non-conditional *Store* to the specified effective address would cause a Data Storage interrupt, it is implementation-dependent whether a Data Storage interrupt occurs.

If the contents of the XER specifies a length of zero bytes for a *Move Assist* instruction, a *Data Storage* interrupt does not occur for reasons of address translation, or storage protection. If such an instruction causes a Data Storage interrupt for other reasons, the setting of the DSISR and DAR reflects only these other reasons listed in the preceding sentence. (E.g., if such an instruction causes a storage protection violation and a Data Address Breakpoint match, the DSISR and DAR are set as if the storage protection violation did not occur.)

The following registers are set:

SRR0 Set to the effective address of the instruction that caused the interrupt.

SRR1

33:36 Set to 0.

42:47 Set to 0.

Others Loaded from the MSR.

MSR See Figure 35.

DSISR

0 Set to 0.

1 Set to 1 if MSR_{DR}=1 and the translation for an attempted access is not found in the primary PTEG or in the secondary PTEG; otherwise set to 0.

2:3 Set to 0.

4 Set to 1 if the access is not permitted by the storage protection mechanism; otherwise set to 0.

Programming Note

The only cases in which $DSISR_4$ can be set to 1 for an access that occurs when MSR_{DR}=0 are those described in Figure 31. These cases can be distinguished from other causes of data storage protection violations by examining $SRR1_{59}$ (the bit in which MSR_{DR} was saved by the interrupt).

5 Set to 1 if the access is due to a ***lwarx***, ***ldarx***, ***stwcx.***, or ***stdcx.*** instruction that addresses storage that is Write Through Required or Caching Inhibited; otherwise set to 0.

6 Set to 1 for a *Store*, ***dcbz***, or ***ecowx*** instruction; otherwise set to 0.

7:8 Set to 0.

9 Set to 1 if a Data Address Compare match or a Data Address Breakpoint match occurs; otherwise set to 0.

10 Set to 0.

11 Set to 1 if execution of an ***eciwx*** or ***ecowx*** instruction is attempted when EAR_E=0; otherwise set to 0.

12:14 Set to 0.

15 Set to an undefined value.

Programming Note

Warning: This setting of $DSISR_{15}$ is being phased out of the architecture. Future versions of the architecture will specify that the Data Storage interrupt sets $DSISR_{15}$ to 0.

16:31 Set to 0.

DAR Set to the effective address of a storage element as described in the following list. The list should be read from the top down; the DAR is set as described by the first item that corresponds to an exception that is reported in the DSISR. For example, if a *Load* instruction causes a storage protection violation and a Data Address Breakpoint match (and both are reported in the DSISR), the DAR is set to the effective address of a byte in the first aligned doubleword for which access was attempted in the page that caused the exception.

- a Data Storage exception occurs for reasons other than a Data Address Breakpoint match or, for ***eciwx*** and ***ecowx***, EAR_E=0
  - a byte in the block that caused the exception, for a *Cache Management* instruction
  - a byte in the first aligned doubleword for which access was attempted in the page that caused the exception, for a *Load*, *Store*, ***eciwx***, or ***ecowx*** instruction ("first" refers to address order; see Section 5.7)
- undefined, for a Data Address Breakpoint match, or if ***eciwx*** or ***ecowx*** is executed when EAR_E=0

For the cases in which the DAR is specified above to be set to a defined value, if the interrupt occurs in 32-bit mode the high-order 32 bits of the DAR are set to 0.

If multiple Data Storage exceptions occur for a given effective address, any one or more of the bits corresponding to these exceptions may be set to 1 in the DSISR.

Programming Note

More than one bit may be set to 1 in the DSISR in the following combinations.

1, {s+}
1, 15, {s+}
4, {s+}
4, 5, {s}
5, {s}
{s+}

In this list, "{s}" represents any combination of the set of bits {6, 9} and "{s+}" adds bit 11 to this set.

Execution resumes at effective address 0x0000_0000_0000_0300.

5.5.4 Data Segment Interrupt

A Data Segment interrupt occurs when no higher priority exception exists and a data access cannot be performed because data address translation is enabled and the effective address of any byte of the storage location specified by a *Load*, *Store*, ***icbi***, ***dcbz***, ***dcbst***, ***dcbf***[***l***] ***eciwx***, or ***ecowx*** instruction cannot be translated to a virtual address.

If a ***stwcx.*** or ***stdcx.*** would not perform its store in the absence of a Data Segment interrupt, and a non-conditional *Store* to the specified effective address would cause a Data Segment interrupt, it is implementation-dependent whether a Data Segment interrupt occurs.

If a *Move Assist* instruction has a length of zero (in the XER), a Data Segment interrupt does not occur, regardless of the effective address.

The following registers are set:

SRR0 Set to the effective address of the instruction that caused the interrupt.

SRR1
- **33:36** Set to 0.
- **42:47** Set to 0.
- **Others** Loaded from the MSR.

MSR See Figure 35.

DSISR Set to an undefined value.

DAR Set to the effective address of a storage element as described in the following list.
- a byte in the block that caused the Data Segment interrupt, for a *Cache Management* instruction
- a byte in the first aligned doubleword for which access was attempted in the segment that caused the Data Segment interrupt, for a *Load*, *Store*, ***eciwx***, or ***ecowx*** instruction ("first" refers to address order; see Section 5.7)

If the interrupt occurs in 32-bit mode, the high-order 32 bits of the DAR are set to 0.

Execution resumes at effective address 0x0000_0000_0000_0380.

Programming Note

A Data Segment interrupt occurs if MSR_{DR}=1 and the translation of the effective address of any byte of the specified storage location is not found in the SLB.

5.5.5 Instruction Storage Interrupt

An Instruction Storage interrupt occurs when no higher priority exception exists and the next instruction to be executed cannot be fetched for any of the following reasons.

- Instruction address translation is enabled and the virtual address cannot be translated to a real address.
- The fetch access violates storage protection.

The following registers are set:

SRR0 Set to the effective address of the instruction that the processor would have attempted to execute next if no interrupt conditions were present (if the interrupt occurs on attempting to fetch a branch target, SRR0 is set to the branch target address).

SRR1
- **33** Set to 1 if MSR_{IR}=1 and the translation for an attempted access is not found in the primary PTEG or in the secondary PTEG; otherwise set to 0.
- **34** Set to 0.
- **35** Set to 1 if the access occurs when MSR_{IR}=1 and is to No-execute storage or to Guarded storage; otherwise set to 0.
- **36** Set to 1 if the access is not permitted by Figure 30 or 31, as appropriate; otherwise set to 0.

Programming Note

The only cases in which $SRR1_{36}$ can be set to 1 for an access that occurs when MSR_{IR}=0 are those described in Figure 31. These cases can be distinguished from other causes of instruction storage protection violations that set $SRR1_{36}$ to 1 by examining $SRR1_{58}$ (the bit in which MSR_{IR} was saved by the interrupt).

42:46 Set to 0.
47 Set to an undefined value.

Programming Note

Warning: This setting of $SRR1_{47}$ is being phased out of the architecture. Future versions of the architecture will specify that the Instruction Storage interrupt sets $SRR1_{47}$ to 0. New software should not depend on the setting described above, and any such dependency in existing software should be removed. (In distinction from the corresponding case for $DSISR_{15}$ as set by the Data Storage interrupt, implementations of future versions of the architecture cannot treat $SRR1_{47}$ as reserved, because $SRR1_{47}$ is set to a meaningful value by other interrupts, e.g., the Program interrupt.)

Others Loaded from the MSR.

MSR See Figure 35.

If multiple Instruction Storage exceptions occur due to attempting to fetch a single instruction, any one or more of the bits corresponding to these exceptions may be set to 1 in SRR1.

Programming Note

More than one bit may be set to 1 in SRR1 in the following combinations.

33, 35
33, 47
33, 35, 47
35, 36

Execution resumes at effective address 0x0000_0000_0000_0400.

5.5.6 Instruction Segment Interrupt

An Instruction Segment interrupt occurs when no higher priority exception exists and the next instruction to be executed cannot be fetched because instruction address translation is enabled and the effective address cannot be translated to a virtual address.

The following registers are set:

SRR0 Set to the effective address of the instruction that the processor would have attempted to execute next if no interrupt conditions were present (if the interrupt occurs on attempting to fetch a branch target, SRR0 is set to the branch target address).

SRR1
33:36 Set to 0.
42:47 Set to 0.
Others Loaded from the MSR.

MSR See Figure 35 on page 65.

Execution resumes at effective address 0x0000_0000_0000_0480.

Programming Note

An Instruction Segment interrupt occurs if MSR_{IR}=1 and the translation of the effective address of the next instruction to be executed is not found in the SLB.

5.5.7 External Interrupt

An External interrupt occurs when no higher priority exception exists, an External exception exists, and MSR_{EE}=1. The occurrence of the interrupt does not cause the exception to cease to exist.

The following registers are set:

SRR0 Set to the effective address of the instruction that the processor would have attempted to execute next if no interrupt conditions were present.

SRR1
33:36 Set to 0.
42:47 Set to 0.
Others Loaded from the MSR.

MSR See Figure 35.

Execution resumes at effective address 0x0000_0000_0000_0500.

5.5.8 Alignment Interrupt

An Alignment interrupt occurs when no higher priority exception exists and a data access cannot be performed for any of the following reasons.

- The operand of a floating-point *Load* or *Store* is not word-aligned, or crosses a virtual page boundary.
- The operand of ***lmw***, ***stmw***, ***lwarx***, ***ldarx***, ***stwcx.***, ***stdcx.***, ***eciwx***, or ***ecowx*** is not aligned.
- The operand of a single-register *Load* or *Store* is not aligned and the processor is in Little-Endian mode.
- The instruction is ***lmw***, ***stmw***, ***lswi***, ***lswx***, ***stswi***, or ***stswx***, and the operand is in storage that is Write

Through Required or Caching Inhibited, or the processor is in Little-Endian mode.

- The operand of a *Load* or *Store* crosses a segment boundary, or crosses a boundary between virtual pages that have different storage control attributes.
- The operand of a *Load* or *Store* is not aligned and is in storage that is Write Through Required or Caching Inhibited.
- The operand of ***dcbz***, ***lwarx***, ***ldarx***, ***stwcx.***, or ***stdcx.*** is in storage that is Write Through Required or Caching Inhibited.

If a ***stwcx.*** or ***stdcx.*** would not perform its store in the absence of an Alignment interrupt and the specified effective address refers to storage that is Write Through Required or Caching Inhibited, it is implementation-dependent whether an Alignment interrupt occurs.

Setting the DSISR and DAR as described below is optional for implementations on which Alignment interrupts occur rarely, if ever, for cases that the Alignment interrupt handler emulates. For such implementations, if the DSISR and DAR are not set as described below they are set to undefined values.

The following registers are set:

SRR0 Set to the effective address of the instruction that caused the interrupt.

SRR1

33:36 Set to 0.
42:47 Set to 0.
Others Loaded from the MSR.

MSR See Figure 35.

DSISR

0:11 Set to 0.
12:13 Set to bits 30:31 of the instruction if DS-form. Set to 0b00 if D-, or X-form.
14 Set to 0.
15:16 Set to bits 29:30 of the instruction if X-form. Set to 0b00 if D- or DS-form.
17 Set to bit 25 of the instruction if X-form. Set to bit 5 of the instruction if D- or DS-form.
18:21 Set to bits 21:24 of the instruction if X-form. Set to bits 1:4 of the instruction if D- or DS-form.
22:26 Set to bits 6:10 of the instruction (RT/RS/FRT/FRS), except undefined for ***dcbz***.
27:31 Set to bits 11:15 of the instruction (RA) for update form instructions; set to either bits 11:15 of the instruction or to any register number not in the range of registers to be loaded for a valid form ***lmw***, a valid form ***lswi***, or a valid form ***lswx*** for which neither RA nor RB is in the range of registers to be loaded; otherwise undefined.

DAR Set to the effective address computed by the instruction, except that if the interrupt occurs in 32-bit mode the high-order 32 bits of the DAR are set to 0.

For an X-form *Load* or *Store*, it is acceptable for the processor to set the DSISR to the same value that would have resulted if the corresponding D- or DS-form instruction had caused the interrupt. Similarly, for a D- or DS-form *Load* or *Store*, it is acceptable for the processor to set the DSISR to the value that would have resulted for the corresponding X-form instruction. For example, an unaligned ***lwax*** (that crosses a protection boundary) would normally, following the description above, cause the DSISR to be set to binary:

```
000000000000 00 0 01 0 0101 ttttt ?????
```

where “ttttt” denotes the RT field, and “?????” denotes an undefined 5-bit value. However, it is acceptable if it causes the DSISR to be set as for ***lwa***, which is

```
000000000000 10 0 00 0 1101 ttttt ?????
```

If there is no corresponding alternative form instruction (e.g., for ***lwaux***), the value described above is set in the DSISR.

The instruction pairs that may use the same DSISR value are.

lhz/lhzx	lhzu/lhzux	lha/lhax	lhau/lhaux
lwz/lwzx	lwzu/lwzux	lwa/lwax	
ld/ldx	ldu/ldux		
lsth/sthx	sthu/sthux	stw/stwx	stwu/stwux
std/stdx	stdu/stdux		
lfs/lfsx	lfsu/lfsux	lfd/lfdx	lfdu/lfdux
stfs/stfsx	stfsu/stfsux	stfd/stfdx	stfdu/stfdux

Execution resumes at effective address 0x0000_0000_0000_0600.

Programming Note

The architecture does not support the use of an unaligned effective address by ***lwarx***, ***ldarx***, ***stwcx.***, ***stdcx.***, ***eciwx***, and ***ecowx***. If an Alignment interrupt occurs because one of these instructions specifies an unaligned effective address, the Alignment interrupt handler must not attempt to simulate the instruction, but instead should treat the instruction as a programming error.

5.5.9 Program Interrupt

A Program interrupt occurs when no higher priority exception exists and one of the following exceptions arises during execution of an instruction:

Floating-Point Enabled Exception

A Floating-Point Enabled Exception type Program interrupt is generated when the value of the expression

$$(MSR_{FE0} \mid MSR_{FE1}) \ \& \ FPSCR_{FEX}$$

is 1. $FPSCR_{FEX}$ is set to 1 by the execution of a floating-point instruction that causes an enabled exception, including the case of a *Move To FPSCR* instruction that causes an exception bit and the corresponding enable bit both to be 1.

Illegal Instruction

An Illegal Instruction type Program interrupt is generated when execution is attempted of an illegal instruction, or of a reserved or optional instruction that is not provided by the implementation.

An Illegal Instruction type Program interrupt may be generated when execution is attempted of any of the following kinds of instruction.

- an instruction that is in invalid form
- an ***lswx*** instruction for which RA or RB is in the range of registers to be loaded
- an ***mtspr*** or ***mfspr*** instruction with an SPR field that does not contain one of the defined values, or an ***mftb*** instruction with a TBR field that does not contain one of the defined values

Privileged Instruction

The following applies if the instruction is executed when MSR_{PR} = 1.

A Privileged Instruction type Program interrupt is generated when execution is attempted of a privileged instruction, or of an ***mtspr*** or ***mfspr*** instruction with an SPR field that contains one of the defined values having spr_0=1. It may be generated when execution is attempted of an ***mtspr*** or ***mfspr*** instruction with an SPR field that does not contain one of the defined values but has spr_0=1, or when execution is attempted of an ***mftb*** instruction with a TBR field that does not contain one of the defined values but has tbr_0=1.

The following applies if the instruction is executed when $MSR_{HV\ PR}$ = 0b00.

A Privileged Instruction type Program interrupt may be generated when execution is attempted of an ***mtspr*** instruction with an SPR field that designates a hypervisor resource, or when execution of a ***tlbie*** or ***tlbsync*** instruction is attempted.

Programming Note

These are the only cases in which a Privileged Instruction type Program interrupt can be generated when MSR_{PR}=0. They can be distinguished from other causes of Privileged Instruction type Program interrupts by examining $SRR1_{49}$ (the bit in which MSR_{PR} was saved by the interrupt).

Trap

A Trap type Program interrupt is generated when any of the conditions specified in a *Trap* instruction is met.

The following registers are set:

SRR0 For all Program interrupts except a Floating-Point Enabled Exception type Program interrupt, set to the effective address of the instruction that caused the corresponding exception.

For a Floating-Point Enabled Exception type Program interrupt, set as described in the following list.

- If $MSR_{FE0\ FE1}$ = 0b00, $FPSCR_{FEX}$ = 1, and an instruction is executed that changes $MSR_{FE0\ FE1}$ to a nonzero value, set to the effective address of the instruction that the processor would have attempted to execute next if no interrupt conditions were present.

Programming Note

Recall that all instructions that can alter $MSR_{FE0\ FE1}$ are context synchronizing, and therefore are not initiated until all preceding instructions have reported all exceptions they will cause.

- If $MSR_{FE0\ FE}$ = 0b11, set to the effective address of the instruction that caused the Floating-Point Enabled Exception.
- If $MSR_{FE0\ FE}$ = 0b01 or 0b10, set to the effective address of the first instruction that caused a Floating-Point Enabled Exception since the most recent time

$FPSCR_{FEX}$ was changed from 1 to 0 or of some subsequent instruction.

Programming Note

If SRR0 is set to the effective address of a subsequent instruction, that instruction will not be beyond the first such instruction at which synchronization of floating-point instructions occurs. (Recall that such synchronization is caused by *Floating-Point Status and Control Register* instructions, as well as by execution synchronizing instructions and events.)

SRR1

33:36 Set to 0.

42 Set to 0.

43 Set to 1 for a Floating-Point Enabled Exception type Program interrupt; otherwise set to 0.

44 Set to 1 for an Illegal Instruction type Program interrupt; otherwise set to 0.

45 Set to 1 for a Privileged Instruction type Program interrupt; otherwise set to 0.

46 Set to 1 for a Trap type Program interrupt; otherwise set to 0.

47 Set to 0 if SRR0 contains the address of the instruction causing the exception and there is only one such instruction; otherwise set to 1.

Programming Note

$SRR1_{47}$ can be set to 1 only if the exception is a Floating-Point Enabled Exception and either $MSR_{FE0\ FE1}$ = 0b01 or 0b10 or $MSR_{FE0\ FE1}$ has just been changed from 0b00 to a nonzero value. ($SRR1_{47}$ is always set to 1 in the last case.)

Others Loaded from the MSR.

Only one of bits 43:46 can be set to 1.

MSR See Figure 35 on page 65.

Execution resumes at effective address 0x0000_0000_0000_0700.

5.5.10 Floating-Point Unavailable Interrupt

A Floating-Point Unavailable interrupt occurs when no higher priority exception exists, an attempt is made to execute a floating-point instruction (including floating-point loads, stores, and moves), and MSR_{FP}=0.

The following registers are set:

SRR0 Set to the effective address of the instruction that caused the interrupt.

SRR1

33:36 Set to 0.

42:47 Set to 0.

Others Loaded from the MSR.

MSR See Figure 35 on page 65.

Execution resumes at effective address 0x0000_0000_0000_0800.

5.5.11 Decrementer Interrupt

A Decrementer interrupt occurs when no higher priority exception exists, a Decrementer exception exists, and MSR_{EE}=1.

The following registers are set:

SRR0 Set to the effective address of the instruction that the processor would have attempted to execute next if no interrupt conditions were present.

SRR1

33:36 Set to 0.

42:47 Set to 0.

Others Loaded from the MSR.

MSR See Figure 35 on page 65.

Execution resumes at effective address 0x0000_0000_0000_0900.

Programming Note

On processors prior to POWER4+, the occurrence of the interrupt caused the exception to cease to exist.

5.5.12 Hypervisor Decrementer Interrupt

A Hypervisor Decrementer interrupt occurs when no higher priority exception exists, a Hypervisor Decrementer exception exists, and the value of the following expression is 1.

$(MSR_{EE} \mid \neg(MSR_{HV}) \mid MSR_{PR})$ & HDICE

The following registers are set:

HSRR0 Set to the effective address of the instruction that the processor would have attempted to execute next if no interrupt conditions were present.

HSRR1
- **33:36** Set to 0.
- **42:47** Set to 0.
- **Others** Loaded from the MSR.

MSR See Figure 35 on page 65.

Execution resumes at effective address 0x0000_0000_0000_0980.

Programming Note

Because the value of MSR_{EE} is always 1 when the processor is in problem state, the simpler expression

$$(MSR_{EE} \mid \neg(MSR_{HV})) \ \& \ HDICE$$

is equivalent to the expression given above.

Programming Note

On processors prior to POWER4+, the Hypervisor Decrementer was not implemented.

5.5.13 System Call Interrupt

A System Call interrupt occurs when a *System Call* instruction is executed.

The following registers are set:

SRR0 Set to the effective address of the instruction following the System Call instruction.

SRR1
- **33:36** Set to 0.
- **42:47** Set to 0.
- **Others** Loaded from the MSR.

MSR See Figure 35 on page 65.

Execution resumes at effective address 0x0000_0000_0000_0C00.

Programming Note

An attempt to execute an ***sc*** instruction with LEV=1 in problem state should be treated as a programming error.

5.5.14 Trace Interrupt

A Trace interrupt occurs when no higher priority exception exists and either MSR_{SE}=1 and any instruction except ***rfid*** or ***hrifd,*** is successfully completed, or MSR_{BE}=1 and a *Branch* instruction is completed. Successful completion means that the instruction caused no other interrupt. Thus a Trace interrupt never occurs for a *System Call* instruction, or for a *Trap* instruction that traps. The instruction that causes a Trace interrupt is called the “traced instruction”.

When a Trace interrupt occurs, the following registers are set:

SRR0 Set to the effective address of the instruction that the processor would have attempted to execute next if no interrupt conditions were present.

SRR1
- **33:36 and 42:47** See the Book IV, *PowerPC Implementation Features* document for the implementation.
- **Others** Loaded from the MSR.

MSR See Figure 35 on page 65.

Execution resumes at effective address 0x0000_0000_0000_0D00.

Extensions to the Trace facility are described in Appendix F, “Example Trace Extensions (Optional)” on page 117.

Programming Note

The following instructions are not traced.

- ***rfid***
- ***hrfid***
- ***sc***, and *Trap* instructions that trap
- other instructions that cause interrupts (other than Trace interrupts)
- the first instructions of any interrupt handler
- instructions that are emulated by software

In general, interrupt handlers can achieve the effect of tracing these instructions.

5.5.15 Performance Monitor Interrupt (Optional)

The Performance Monitor interrupt is part of the optional Performance Monitor facility; see Appendix E. If the Performance Monitor facility is not implemented or does not use this interrupt, the corresponding interrupt vector (see Figure 36 on page 65) is treated as reserved.

5.6 Partially Executed Instructions

If a Data Storage, Data Segment, Alignment, system-caused, or imprecise exception occurs while a *Load* or *Store* instruction is executing, the instruction may be aborted. In such cases the instruction is not completed,

but may have been partially executed in the following respects.

- Some of the bytes of the storage operand may have been accessed, except that if access to a given byte of the storage operand would violate storage protection, that byte is neither copied to a register by a *Load* instruction nor modified by a *Store* instruction. Also, the rules for storage accesses given in Section 4.2.4.1, "Guarded Storage" on page 28 and in the section entitled "Instruction Restart" in Book II are obeyed.
- Some registers may have been altered as described in the Book II section cited above.
- Reference and Change bits may have been updated as described in Section 4.9.
- For a ***stwcx.*** or ***stdcx.*** instruction that is executed in-order, CR0 may have been set to an undefined value and the reservation may have been cleared.

The architecture does not support continuation of an aborted instruction but intends that the aborted instruction be re-executed if appropriate.

Programming Note

An exception may result in the partial execution of a *Load* or *Store* instruction. For example, if the Page Table Entry that translates the address of the storage operand is altered, by a program running on another processor, such that the new contents of the Page Table Entry preclude performing the access, the alteration could cause the *Load* or *Store* instruction to be aborted after having been partially executed.

As stated in the Book II section cited above, if an instruction is partially executed the contents of registers are preserved to the extent that the instruction can be re-executed correctly. The consequent preservation is described in the following list. For any given instruction, zero or one item in the list applies.

- For a fixed-point *Load* instruction that is not a multiple or string form, or for an ***eciwx*** instruction, if RT = RA or RT = RB then the contents of register RT are not altered.
- For an update form *Load* or *Store* instruction, the contents of register RA are not altered.

5.7 Exception Ordering

Since multiple exceptions can exist at the same time and the architecture does not provide for reporting more than one interrupt at a time, the generation of more than one interrupt is prohibited. Some exceptions, such as the External exception, persist and can be deferred. However, other exceptions would be lost if they were not recognized and handled when they occur. For example, if an External interrupt was generated when a Data Storage exception existed, the Data Storage exception would be lost. If the Data Storage exception was caused by a *Store Multiple* instruction for which the storage operand crosses a virtual page boundary and the exception was a result of attempting to access the second virtual page, the store could have modified locations in the first virtual page even though it appeared that the *Store Multiple* instruction was never executed.

For the above reasons, all exceptions are prioritized with respect to other exceptions that may exist at the same instant to prevent the loss of any exception that is not persistent. Some exceptions cannot exist at the same instant as some others.

Data Storage, Data Segment, and Alignment exceptions occur as if the storage operand were accessed one byte at a time in order of increasing effective address (with the obvious caveat if the operand includes both the maximum effective address and effective address 0).

5.7.1 Unordered Exceptions

The exceptions listed here are unordered, meaning that they may occur at any time regardless of the state of the interrupt processing mechanism. These exceptions are recognized and processed when presented.

1. System Reset
2. Machine Check

5.7.2 Ordered Exceptions

The exceptions listed here are ordered with respect to the state of the interrupt processing mechanism.

System-Caused or Imprecise

1. Program
   - Imprecise Mode Floating-Point Enabled Exception
2. External, Decrementer, and Hypervisor Decrementer

Instruction-Caused and Precise

1. Instruction Segment
2. Instruction Storage
3. Program
   - Illegal Instruction
   - Privileged Instruction
4. Function-Dependent

   4.a Fixed-Point and Branch

   1a Program
   - Trap

1b System Call
1c Data Storage, Data Segment, or Alignment
2 Trace

4.b Floating-Point
1 FP Unavailable
2a Program
- Precise Mode Floating-Pt Enabled Excep'n
2b Data Storage, Data Segment, or Alignment
3 Trace

For implementations that execute multiple instructions in parallel using pipeline or superscalar techniques, or combinations of these, it can be difficult to understand the ordering of exceptions. To understand this ordering it is useful to consider a model in which each instruction is fetched, then decoded, then executed, all before the next instruction is fetched. In this model, the exceptions a single instruction would generate are in the order shown in the list of instruction-caused exceptions. Exceptions with different numbers have different ordering. Exceptions with the same numbering but different lettering are mutually exclusive and cannot be caused by the same instruction. The External, Decrementer, and Hypervisor Decrementer interrupts have equal ordering. Similarly, where Data Storage, Data Segment, and Alignment exceptions are listed in the same item they have equal ordering.

Even on processors that are capable of executing several instructions simultaneously, or out of order, instruction-caused interrupts (precise and imprecise) occur in program order.

5.8 Interrupt Priorities

This section describes the relationship of nonmaskable, maskable, precise, and imprecise interrupts. In the following descriptions, the interrupt mechanism waiting for all possible exceptions to be reported includes only exceptions caused by previously initiated instructions (e.g., it does not include waiting for the Decrementer to step through zero). The exceptions are listed in order of highest to lowest priority.

1. System Reset

   System Reset exception has the highest priority of all exceptions. If this exception exists, the interrupt mechanism ignores all other exceptions and generates a System Reset interrupt.

   Once the System Reset interrupt is generated, no nonmaskable interrupts are generated due to exceptions caused by instructions issued prior to the generation of this interrupt.

2. Machine Check

   Machine Check exception is the second highest priority exception. If this exception exists and a System Reset exception does not exist, the interrupt mechanism ignores all other exceptions and generates a Machine Check interrupt.

   Once the Machine Check interrupt is generated, no nonmaskable interrupts are generated due to exceptions caused by instructions issued prior to the generation of this interrupt.

3. Instruction-Dependent

   This exception is the third highest priority exception. When this exception is created, the interrupt mechanism waits for all possible Imprecise exceptions to be reported. It then generates the appropriate ordered interrupt if no higher priority exception exists when the interrupt is to be generated. Within this category a particular instruction may present more than a single exception. When this occurs, those exceptions are ordered in priority as indicated in the following lists. Where Data Storage, Data Segment, and Alignment exceptions are listed in the same item they have equal priority (i.e., the processor may generate any one of the three interrupts for which an exception exists).

   A. Fixed-Point Loads and Stores

      a. Program - Illegal Instruction
      b. Data Storage, Data Segment, or Alignment
      c. Trace

   B. Floating-Point Loads and Stores

      a. Program - Illegal Instruction
      b. Floating-Point Unavailable
      c. Data Storage, Data Segment, or Alignment
      d. Trace

   C. Other Floating-Point Instructions

      a. Floating-Point Unavailable
      b. Program - Precise Mode Floating-Point Enabled Exception
      c. Trace

   D. ***rfid, hrfid*** and ***mtmsr***[***d***]

      a. Program - Privileged Instruction
      b. Program - Floating-Point Enabled Exception
      c. Trace, for ***mtmsr***[***d***] only

   E. Other Instructions

      a. These exceptions are mutually exclusive and have the same priority:
         - Program - Trap
         - System Call
         - Program - Privileged Instruction
         - Program - Illegal Instruction

      b. Trace

   F. Instruction Storage and Instruction Segment

      These exceptions have the lowest priority in this category. They are recognized only when all instructions prior to the instruction causing one of these exceptions appear to have completed and that instruction is the next instruction to be executed. The two exceptions are mutually exclusive.

      The priority of these exceptions is specified for completeness and to ensure that they are not given more favorable treatment. It is acceptable for an implementation to treat these exceptions as though they had a lower priority.

4. Program - Imprecise Mode Floating-Point Enabled Exception

   This exception is the fourth highest priority exception. When this exception is created, the interrupt mechanism waits for all other possible exceptions to be reported. It then generates this interrupt if no higher priority exception exists when the interrupt is to be generated.

5. External, Decrementer, and Hypervisor Decrementer

   These exceptions are the lowest priority exceptions. All have equal priority (i.e., the processor may generate any one of these interrupts for which an exception exists). When one of these exceptions is created, the interrupt processing mechanism waits for all other possible exceptions to be reported. It then generates the corresponding interrupt if no higher priority exception exists when the interrupt is to be generated.

If a Hypervisor Decrementer exception exists and the Hypervisor Decrementer interrupt is enabled, and each attempt to execute an instruction causes an exception (see the Programming Note below), the Hypervisor Decrementer interrupt is not delayed indefinitely.

Programming Note

An incorrect or malicious operating system could corrupt the first instruction in the interrupt vector location for an instruction-caused interrupt such that the attempt to execute the instruction causes the same exception that caused the interrupt (a looping interrupt; e.g., illegal instruction and Program interrupt). Similarly, the first instruction of the interrupt vector for one instruction-caused interrupt could cause a different instruction-caused interrupt, and the first instruction of the interrupt vector for the second instruction-caused interrupt could cause the first instruction-caused interrupt (e.g., Program interrupt and Floating-Point Unavailable interrupt). The looping caused by these and similar cases is terminated by the occurrence of a System Reset or Hypervisor Decrementer interrupt.

Chapter 6. Timer Facilities

6.1 Overview

The Time Base, Decrementer, Hypervisor Decrementer, and the Processor Utilization of Resources registers provide timing functions for the system. All are volatile resources and must be initialized during startup. The ***mftb*** instruction is used to read the Time Base; the ***mtspr*** and ***mfspr*** instructions are used to write the Time Base and Decrementer(s) and to read the Decrementer(s).

Time Base (TB)

The Time Base provides a long-period counter driven by an implementation-dependent frequency.

Decrementer (DEC)

The Decrementer, a counter that is updated at the same rate as the Time Base, provides a means of signaling an interrupt after a specified amount of time has elapsed unless

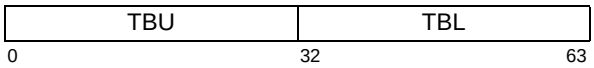
- the Decrementer is altered by software in the interim, or
- the Time Base update frequency changes.

Hypervisor Decrementer (HDEC)

The Hypervisor Decrementer provides a means for the hypervisor to manage timing functions independently of the Decrementer, which is managed by virtual partitions. Similar to the Decrementer, the HDEC is a counter that is updated at the same rate as the Time Base, and it provides a means of signaling an interrupt after a specified amount of time has elapsed. Software must have hypervisor privilege to update the HDEC.

6.2 Time Base

The Time Base (TB) is a 64-bit register (see Figure 37) containing a 64-bit unsigned integer that is incremented periodically. Each increment adds 1 to the low-order bit (bit 63). The frequency at which the integer is updated is implementation-dependent.



Field	Description
TBU	Upper 32 bits of Time Base
TBL	Lower 32 bits of Time Base

Figure 37. Time Base

The Time Base is a hypervisor resource; see Section 1.7, “Logical Partitioning (LPAR)” on page 5.

The Time Base increments until its value becomes 0xFFFF_FFFF_FFFF_FFFF (2^{64} - 1). At the next increment, its value becomes 0x0000_0000_0000_0000. There is no interrupt or other indication when this occurs.

The period of the Time Base depends on the driving frequency. As an order of magnitude example, suppose that the CPU clock is 1 GHz and that the Time Base is driven by this frequency divided by 32. Then the period of the Time Base would be

$$T_{TB} = \frac{2^{64} \times 32}{1\ \text{GHz}} = 5.90 \times 10^{11}\ \text{seconds}$$

which is approximately 18,700 years.

The Time Base is implemented such that:

1. Loading a GPR from the Time Base has no effect on the accuracy of the Time Base.
2. Copying the contents of a GPR to the Time Base replaces the contents of the Time Base with the contents of the GPR.

The PowerPC Architecture does not specify a relationship between the frequency at which the Time Base is updated and other frequencies, such as the CPU clock or bus clock in a PowerPC system. The Time Base update frequency is not required to be constant. What *is* required, so that system software can keep time of day and operate interval timers, is one of the following.

- The system provides an (implementation-dependent) interrupt to software whenever the update frequency of the Time Base changes, and a means to determine what the current update frequency is.
- The update frequency of the Time Base is under the control of the system software.

Implementations must provide a means for either preventing the Time Base from incrementing or preventing it from being read in problem state (MSR_{PR}=1). If the means is under software control, it must be accessible only in hypervisor state ($MSR_{HV\ PR}$ = 0b10). There must be a method for getting all processors' Time Bases to start incrementing with values that are identical or almost identical in all processors.

Programming Note

If the hypervisor initializes the Time Base on power-on to some reasonable value and the update frequency of the Time Base is constant, the Time Base can be used as a source of values that increase at a constant rate, such as for time stamps in trace entries.

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

Successive readings of the Time Base may return identical values.

See the description of the Time Base in Book II, *PowerPC Virtual Environment Architecture* for ways to compute time of day in POSIX format from the Time Base.

6.2.1 Writing the Time Base

Writing the Time Base is privileged, and can be done only in hypervisor state. Reading the Time Base is not privileged; it is discussed in Book II, *PowerPC Virtual Environment Architecture*.

It is not possible to write the entire 64-bit Time Base using a single instruction. The ***mttbl*** and ***mttbu*** extended mnemonics write the lower and upper halves of the Time Base (TBL and TBU), respectively, preserving the other half. These are extended mnemonics for the ***mtspr*** instruction; see page 98.

The Time Base can be written by a sequence such as:

```
lwz    Rx,upper # load 64-bit value for
lwz    Ry,lower # TB into Rx and Ry
li     Rz,0
mttbl  Rz       # set TBL to 0
mttbu  Rx       # set TBU
mttbl  Ry       # set TBL
```

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

Programming Note

The instructions for writing the Time Base are mode-independent. Thus code written to set the Time Base will work correctly in either 64-bit or 32-bit mode.

6.3 Decrementer

The Decrementer (DEC) is a 32-bit decrementing counter that provides a mechanism for causing a Decrementer interrupt after a programmable delay. The contents of the Decrementer are treated as a signed integer.

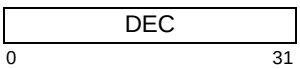


Figure 38. Decrementer

The Decrementer is driven by the same frequency as the Time Base. The period of the Decrementer will depend on the driving frequency, but if the same values are used as given above for the Time Base (see Section 6.2), and if the Time Base update frequency is constant, the period would be

$$T_{DEC} = \frac{2^{32} \times 32}{1\ \text{GHz}} = 137 \text{ seconds.}$$

The Decrementer counts down.

The exception effects of the Decrementer are said to be consistent with the contents of the Decrementer if one of the following statements is true.

- DEC_0=0 and a Decrementer exception does not exist.
- DEC_0=1 and a Decrementer exception exists.

If DEC_0=0, a context synchronizing instruction or event ensures that the exception effects of the Decrementer are consistent with the contents of the Decrementer. Otherwise, when the contents of DEC_0 change, the exception effects of the Decrementer become consistent with the new contents of the Decrementer reasonably soon after the change.

The preceding paragraph applies regardless of whether the change in the contents of DEC_0 is the result of decrementation of the Decrementer by the processor or of modification of the Decrementer caused by execution of an ***mtspr*** instruction.

The operation of the Decrementer satisfies the following constraints.

1. The operation of the Time Base and the Decrementer is coherent, i.e., the counters are driven by the same fundamental time base.
2. Loading a GPR from the Decrementer has no effect on the accuracy of the Time Base.
3. Copying the contents of a GPR to the Decrementer replaces the contents of the Decrementer with the contents of the GPR.

Programming Note

On processors prior to POWER4+, the Decrementer exception was hidden state that was cleared when the Decrementer interrupt was taken.

Programming Note

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

6.3.1 Writing and Reading the Decrementer

The contents of the Decrementer can be read or written using the ***mfspr*** and ***mtspr*** instructions, both of which are privileged when they refer to the Decrementer.

Using an extended mnemonic (see page 98), the Decrementer can be written from GPR Rx using:

```
mtdec  Rx
```

Programming Note

On processors prior to POWER4+, if the execution of the ***mtdec*** instruction causes bit 0 of the Decrementer to change from 0 to 1, an interrupt request is signaled.

The Decrementer can be read into GPR Rx using:

```
mfdec  Rx
```

Copying the Decrementer to a GPR has no effect on the Decrementer contents or on the interrupt mechanism.

6.4 Hypervisor Decrementer

The Hypervisor Decrementer (HDEC) is a 32-bit decrementing counter that provides a mechanism for causing a Hypervisor Decrementer interrupt after a programmable delay. The contents of the Decrementer are treated as a signed integer.

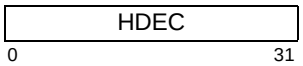


Figure 39. Hypervisor Decrementer

The Hypervisor Decrementer is a hypervisor resource; see Section 1.7, “Logical Partitioning (LPAR)” on page 5.

The Hypervisor Decrementer is driven by the same frequency as the Time Base. The period of the Hypervisor Decrementer will depend on the driving frequency, but if the same values are used as given above for the Time Base (see Section 6.2), and if the Time Base update frequency is constant, the period would be

$$T_{DEC} = \frac{2^{32} \times 32}{1\ \text{GHz}} = 137 \text{ seconds.}$$

The exception effects of the Hypervisor Decrementer are said to be consistent with the contents of the Hypervisor Decrementer if one of the following statements is true.

- $HDEC_0$=0 and a Hypervisor Decrementer exception does not exist.
- $HDEC_0$=1 and a Hypervisor Decrementer exception exists.

If $HDEC_0$=0, a context synchronizing instruction or event ensures that the exception effects of the Hypervi-

sor Decrementer are consistent with the contents of the Hypervisor Decrementer. Otherwise, when the contents of $HDEC_0$ change, the exception effects of the Hypervisor Decrementer become consistent with the new contents of the Hypervisor Decrementer reasonably soon after the change.

The preceding paragraph applies regardless of whether the change in the contents of $HDEC_0$ is the result of decrementation of the Hypervisor Decrementer by the processor or of modification of the Hypervisor Decrementer caused by execution of an mtspr instruction.

The operation of the Hypervisor Decrementer satisfies the following constraints.

1. The operation of the Time Base and the Hypervisor Decrementer is coherent, i.e., the counters are driven by the same fundamental time base.
2. Loading a GPR from the Hypervisor Decrementer has no effect on the accuracy of the Hypervisor Decrementer.
3. Copying the contents of a GPR to the Hypervisor Decrementer replaces the contents of the Hypervisor Decrementer with the contents of the GPR.

Programming Note

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

6.5 Processor Utilization of Resources Register (PURR)

The Processor Utilization of Resources Register (PURR) is a 64-bit counter, the contents of which provide an estimate of the resources used by the processor. The contents of the PURR are treated as a 64-bit unsigned integer.

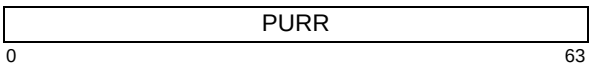


Figure 40. Processor Utilization of Resources Register

The PURR is a hypervisor resource; see Section 1.7, Logical Partitioning (LPAR) on p. 4.

The contents of the PURR increase monotonically, unless altered by software, until the sum of the contents plus the amount by which it is to be increased exceed 0xFFFF_FFFF_FFFF_FFFF (2^{64} - 1) at which point the contents are replaced by that sum modulo 2^{64}. There is no interrupt or other indication when this occurs.

The rate at which the value represented by the contents of the PURR increases is an estimate of the portion of resources used by the processor with respect to other processors that share those resources monitored by the PURR.

Let the difference between the value represented by the contents of the Time Base at times T_a and T_b be T_{ab}. Let the difference between the value represented by the contents of the PURR at time T_a and T_b be the value P_{ab}. The ratio of P_{ab}/T_{ab} is an estimate of the percentage of shared resources used by the processor during the interval T_{ab}. For the set {S} of processors that share the resources monitored by the PURR, the sum of the usage estimates for all the processors in the set is 1.0.

The definition of the set of processors S, the shared resources corresponding to the set S, and specifics of the algorithm for incrementing the PURR are implementation-specific. See Book IV, *PowerPC Implementation Features* for details on each implementation.

The PURR is implemented such that:

1. Loading a GPR from the PURR has no effect on the accuracy of the PURR.
2. Copying the contents of a GPR to the PURR replaces the contents of the PURR with the contents of the GPR.

Programming Note

Estimates computed as described above may be useful for purposes of resource use accounting, program dispatching, etc.

Because the rate at which the PURR accumulates resource usage estimates is dependent on the frequency at which the Time Base is incremented, the interpretation of the contents of the PURR must be adjusted if the frequency at which the Time Base is incremented is altered.

Chapter 7. Synchronization Requirements for Context Alterations

Changing the contents of certain System Registers and of SLB entries and Page Table Entries, and invalidating SLB and TLB entries, can have the side effect of altering the context in which data addresses and instruction addresses are interpreted, and in which instructions are executed and data accesses are performed. For example, changing MSR_{IR} from 0 to 1 has the side effect of enabling translation of instruction addresses. These side effects need not occur in program order, and therefore may require explicit synchronization by software. (Program order is defined in Book II, *PowerPC Virtual Environment Architecture*.)

An instruction that alters the context in which data addresses or instruction addresses are interpreted, or in which instructions are executed or data accesses are performed, is called a *context-altering instruction*. This chapter covers all the context-altering instructions. The software synchronization required for them is shown in Table 1 (for data access) and Table 2 (for instruction fetch and execution).

The notation "CSI" in the tables means any context synchronizing instruction (e.g., ***sc***, ***isync***, or ***rfid***). A context synchronizing interrupt (i.e., any interrupt except non-recoverable System Reset or non-recoverable Machine Check) can be used instead of a context synchronizing instruction. If it is, phrases like "the synchronizing instruction", below, should be interpreted as meaning the instruction at which the interrupt occurs. If no software synchronization is required before (after) a context-altering instruction, "the synchronizing instruction before (after) the context-altering instruction" should be interpreted as meaning the context-altering instruction itself.

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

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

Programming Note

Sometimes advantage can be taken of the fact that certain events, such as interrupts, and certain instructions that occur naturally in the program, such as the ***rfid*** that returns from an interrupt handler, provide the required synchronization.

No software synchronization is required before or after a context-altering instruction that is also context synchronizing (e.g., ***rfid***, ***mtmsr***[***d***] with L=0), except perhaps when altering the LE bit (see the tables). No software synchronization is required before most of the other alterations shown in Table 2, because all instructions preceding the context-altering instruction are fetched and decoded before the context-altering instruction is executed (the processor must determine whether any of these preceding instructions are context synchronizing).

Unless otherwise stated, the material in this chapter assumes a uniprocessor environment.

Instruction or Event	Required Before	Required After	Notes
interrupt	none	none	
rfid	none	none	
hrfid	none	none	
sc	none	none	
Trap	none	none	
mtmsrd (SF)	none	none	3
mtmsr[***d***] (PR)	none	none	3
mtmsr[***d***] (DR)	none	none	3
mtmsr[***d***] (LE)	--	--	1, 3
mtsr[***in***]	CSI	CSI	
mtspr (ACCR)	CSI	CSI	
mtspr (SDR1)	***ptesync***	CSI	5, 6
mtspr (DABR)	--	--	4
mtspr (DABRX)	--	--	4
mtspr (EAR)	CSI	CSI	
mtspr (RMOR)	CSI	CSI	15
mtspr (HRMOR)	CSI	CSI	15
mtspr (LPCR)	CSI	CSI	15
slbie	CSI	CSI	
slbia	CSI	CSI	
slbmte	CSI	CSI	13
tlbie	CSI	CSI	7, 9
tlbiel	CSI	***ptesync***	7
tlbia	CSI	CSI	7
Store(PTE)	none	{***ptesync***, CSI}	8, 9

Table 1: Synchronization requirements for data access

Instruction or Event	Required Before	Required After	Notes
interrupt	none	none	
rfid	none	none	
hrfid	none	none	
sc	none	none	
Trap	none	none	
mtmsrd (SF)	none	none	3, 10
mtmsr[***d***] (EE)	none	none	2, 3
mtmsr[***d***] (PR)	none	none	3, 11
mtmsr[***d***] (FP)	none	none	3
mtmsr[***d***](FE0,FE1)	none	none	3
mtmsr[***d***] (SE, BE)	none	none	3
mtmsr[***d***] (IR)	none	none	3, 11
mtmsr[***d***] (RI)	none	none	3
mtmsr[***d***] (LE)	--	--	1, 3
mtsr[***in***]	none	CSI	11
mtspr (SDR1)	***ptesync***	CSI	5, 6
mtspr (DEC)	none	none	12
mtspr (HDEC)	none	none	12
mtspr (LPIDR)	CSI	CSI	9, 14
mtspr (CTRL)	none	none	
mtspr (RMOR)	none	CSI	15
mtspr (HRMOR)	none	CSI	11,15
mtspr (LPCR)	none	CSI	15
slbie	none	CSI	
slbia	none	CSI	
slbmte	none	CSI	11, 13
tlbie	none	CSI	7, 9
tlbiel	none	CSI	7
tlbia	none	CSI	7
Store(PTE)	none	{***ptesync***, CSI}	8, 9

Table 2: Synchronization requirements for instruction fetch and/or execution

Notes:

1. Synchronization requirements for changing from one Endian mode to the other using the ***mtmsr***[***d***] instruction are implementation-dependent, and are specified in the Book IV, *PowerPC Implementation Features* document for the implementation.
2. The effect of changing the EE bit is immediate, even if the ***mtmsr***[***d***] instruction is not context synchronizing (i.e., even if L=1).
   - If an ***mtmsr***[***d***] instruction sets the EE bit to 0, neither an External interrupt nor a Decrementer interrupt occurs after the ***mtmsr***[***d***] is executed.
   - If an ***mtmsr***[***d***] instruction changes the EE bit from 0 to 1 when an External, Decrementer, or higher priority exception exists, the corresponding interrupt occurs immediately after the ***mtmsr***[***d***] is executed, and before the next instruction is executed in the program that set EE to 1.
   - If a hypervisor executes the ***mtmsr***[***d***] instruction that sets the EE bit to 0, a Hypervisor Decrementer interrupt does not occur after ***mtmsr***[***d***] is executed as long as the processor remains in hypervisor state.
   - If the hypervisor executes an ***mtmsr***[***d***] instruction that changes the EE bit from 0 to 1 when a Hypervisor Decrementer or higher priority exception exists, the corresponding interrupt occurs immediately after the ***mtmsr***[***d***] instruction is executed, and before the next instruction is executed, provided HDICE is 1.
3. For software that will run on processors that comply with versions of the architecture that precede Version 2.01, a context synchronizing instruction is

required after the ***mtmsr***[***d***] instruction; see the first Programming Note in the descriptions of these instructions on pages 22 and 91.

4. Synchronization requirements for changing the DABR and DABRX are implementation-dependent, and are specified in the Book IV, *PowerPC Implementation Features* document for the implementation.

5. SDR1 must not be altered when MSR_{DR}=1 or MSR_{IR}=1; if it is, the results are undefined.

6. A ***ptesync*** instruction is required before the ***mtspr*** instruction because (a) SDR1 identifies the Page Table and thereby the location of Reference and Change bits, and (b) on some implementations, use of SDR1 to update Reference and Change bits may be independent of translating the virtual address. (For example, an implementation might identify the PTE in which to update the Reference and Change bits in terms of its offset in the Page Table, instead of its real address, and then add the Page Table address from SDR1 to the offset to determine the real address at which to update the bits.) To ensure that Reference and Change bits are updated in the correct Page Table, SDR1 must not be altered until all Reference and Change bits are updated in the correct Page Table, SDR1 must not be altered until all Reference and Change bit updates associated with address translations that were performed, by the processor executing the ***mtspr*** instruction, before the ***mtspr*** instruction is executed have been performed with respect to that processor. A ***ptesync*** instruction guarantees this synchronization of Reference and Change bit updates, while neither a context synchronizing operation nor the instruction fetching mechanism does so.

7. For data accesses, the context synchronizing instruction before the ***tlbie***, ***tlbiel***, or ***tlbia*** instruction ensures that all preceding instructions that access data storage have completed to a point at which they have reported all exceptions they will cause.

   The context synchronizing instruction after the ***tlbie***, ***tlbiel***, or ***tlbia*** instruction ensures that storage accesses associated with instructions following the context synchronizing instruction will not use the TLB entry(s) being invalidated.

   (If it is necessary to order storage accesses associated with preceding instructions, or Reference and Change bit updates associated with preceding address translations, with respect to subsequent data accesses, a ***ptesync*** instruction must also be used, either before or after the ***tlbie***, ***tlbiel***, or ***tlbia*** instruction. These effects of the ***ptesync*** instruction are described in the last paragraph of Note 8.)

8. The notation "{***ptesync***,CSI}" denotes an instruction sequence. Other instructions may be interleaved with this sequence, but these instructions must appear in the order shown.

   No software synchronization is required before the *Store* instruction because (a) stores are not performed out-of-order and (b) address translations associated with instructions preceding the *Store* instruction are not performed again after the store has been performed (see Section 4.2.4). These properties ensure that all address translations associated with instructions preceding the *Store* instruction will be performed using the old contents of the PTE.

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

   The ***ptesync*** instruction also ensures that all storage accesses associated with instructions preceding the ***ptesync*** instruction, and all Reference and Change bit updates associated with additional address translations that were performed, by the processor executing the ***ptesync*** instruction, before the ***ptesync*** instruction is executed, will be performed with respect to any processor or mechanism, to the extent required by the associated Memory Coherence Required attributes, before any data accesses caused by instructions following the ***ptesync*** instruction are performed with respect to that processor or mechanism.

9. There are additional software synchronization requirements for the ***tlbie*** instruction in multiprocessor environments; see Section 4.12, "Page Table Update Synchronization Requirements" on page 57.

   Section 4.12 also gives examples of using ***tlbie***, *Store*, and related instructions to maintain the Page Table, in both multiprocessor and uniprocessor environments.

Programming Note

In a multiprocessor system, if software locking is used to help ensure that the requirements described in Section 4.12 are satisfied, the ***isync*** instruction near the end of the lock acquisition sequence (see the section entitled "Acquire Lock and Import Shared Storage" in Book II, *PowerPC Virtual Environment Architecture*) may naturally provide the context synchronization that is required before the alteration.

10. The alteration must not cause an implicit branch in effective address space. Thus, when changing MSR_{SF} from 1 to 0, the ***mtmsrd*** instruction must have an effective address that is less than 2^{32} - 4. Furthermore, when changing MSR_{SF} from 0 to 1, the ***mtmsrd*** instruction must not be at effective address 2^{32} - 4 (see Section 4.2.2.2 on page 27).

11. The alteration must not cause an implicit branch in real address space. Thus the real address of the context-altering instruction and of each subsequent instruction, up to and including the next context synchronizing instruction, must be independent of whether the alteration has taken effect.

12. The elapsed time between the contents of the Decrementer or Hypervisor Decrementer becoming negative and the signaling of the corresponding exception is not defined.

13. If an ***slbmte*** instruction alters the mapping, or associated attributes, of a currently mapped ESID, the ***slbmte*** must be preceded by an ***slbie*** (or ***slbia***) instruction that invalidates the existing translation. This applies even if the corresponding entry is no longer in the SLB (the translation may still be in implementation-specific address translation lookaside information). No software synchronization is needed between the ***slbie*** and the ***slbmte***, regardless of whether the index of the SLB entry (if any) containing the current translation is the same as the SLB index specified by the ***slbmte***.

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

14. The context synchronizing instruction before the ***mtspr*** instruction ensures that the LPIDR is not altered out-of-order. (Out-of-order alteration of the LPIDR could permit the requirements described in Section 4.12.1 to be violated. For the same reason, such a context synchronizing instruction may be needed even if the new LPID value is equal to the old LPID value.)

    See also Section 1.7, "Logical Partitioning (LPAR)" on page 5 regarding moving a processor from one partition to another.

15. When the RMOR or HRMOR is modified, or the RMLS, $LPES_1$, or RMI fields of the LPCR are modified, software must invalidate all implementation-specific lookaside information used in address translation that depends on values stored in these registers. All implementations provide a means by which software can do this.

Chapter 8. Optional Facilities and Instructions

The facilities and instructions described in this chapter are optional. An implementation may provide all, some, or none of them.

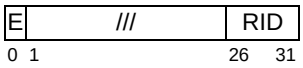
8.1 External Control

The External Control facility permits a program to communicate with a special-purpose device. The facility consists of a Special Purpose Register, called EAR, and two instructions, called *External Control In Word Indexed* (***eciwx***) and *External Control Out Word Indexed* (***ecowx***).

This facility must provide a means of synchronizing the devices with the processor to prevent the use of an address by the device when the translation that produced that address is being invalidated.

8.1.1 External Access Register

This 32-bit Special Purpose Register controls access to the External Control facility and, for external control operations that are permitted, identifies the target device.



Bit(s)	Name	Description
0	E	Enable bit
26:31	RID	Resource ID

All other fields are reserved.

Figure 41. External Access Register

The EAR is a hypervisor resource; see Section 1.7, “Logical Partitioning (LPAR)” on page 5.

The high-order bits of the RID field that correspond to bits of the Resource ID beyond the width of the Resource ID supported by the implementation are treated as reserved bits.

Programming Note

The hypervisor can use the EAR to control which programs are allowed to execute *External Access* instructions, when they are allowed to do so, and which devices they are allowed to communicate with using these instructions.

8.1.2 External Access Instructions

The *External Access* instructions, *External Control In Word Indexed* (***eciwx***) and *External Control Out Word Indexed* (***ecowx***), are described in Book II, *PowerPC Virtual Environment Architecture*. Additional information about them is given below.

If attempt is made to execute either of these instructions when EAR_E=0, a Data Storage interrupt occurs with bit 11 of the DSISR set to 1.

The instructions are supported whenever MSR_{DR}=1. If either instruction is executed when MSR_{DR}=0 (real addressing mode), the results are boundedly undefined.

8.2 Real Mode Storage Control

The Real Mode Storage Control facility provides a means of specifying portions of real storage that are treated as non-Guarded in hypervisor real addressing mode ($MSR_{HV\ PR}$=0b10, and MSR_{IR}=0 or MSR_{DR}=0, as appropriate for the type of access). The remaining portions are treated as Guarded in hypervisor real addressing mode (as is all of storage on implementations that do not provide this means). The means is a hypervisor resource (see Section 1.7, “Logical Partitioning (LPAR)” on page 5), and may also be system-specific.

If the Real Mode Caching Inhibited (RMI) bit is set to 1, it is undefined whether a given data access to a storage location that is treated as non-Guarded in hypervisor real addressing mode is treated as Caching Inhibited or as not Caching Inhibited. If the access is treated as Caching Inhibited and is performed out-of-order, the access cannot cause a Machine Check or Checkstop to occur out-of-order due to violation of the requirements given in Section 4.8.2, “Altering the Storage Control Bits” on page 42 for changing the value of the effective I bit. (Recall that software must ensure that RMI = 0 when the processor is not in hypervisor real addressing mode; see Section 4.2.6.3, “Storage Control Attributes for Real Addressing Mode and for Implicit Storage Accesses” on page 30.)

The facility does not apply to implicit accesses to the Page Table by the processor in performing address translation or in recording reference and change information. These accesses are performed as described in Section 4.2.6.3 on page 30.

Programming Note

The preceding capability can be used to improve the performance of hypervisor software that runs in hypervisor real addressing mode, by causing accesses to instructions and data that occupy well-behaved storage to be treated as non-Guarded. See also the second paragraph of the Programming Note in Section 4.2.6.3.

If RMI=1, the statement in Section 4.2.4, “Performing Operations Out-of-Order” on page 27, that non-Guarded storage locations may be fetched out-of-order into a cache only if they could be fetched into that cache by in-order execution does not preclude the out-of-order fetching into the data cache of storage locations that are treated as non-Guarded in hypervisor real addressing mode, because the effective RMI value that could be used for an in-order data access to such a storage location is undefined and hence could be 0.

8.3 Move to Machine State Register Instruction

Move To Machine State Register X-form

mtmsr RS,L

31	RS	///	L	///	146	/
0	6	11	15	16	21	31

```
if L = 0 then
   MSR48 ← (RS)48 | (RS)49
   MSR58 ← (RS)58 | (RS)49
   MSR59 ← (RS)59 | (RS)49
   MSR32:47 49:50 52:57 60:63 ←(RS)32:47 49:50 52:57 60:63
else
   MSR48 62 ← (RS)48 62
```

The MSR is set based on the contents of register RS and of the L field.

L=0:

The result of ORing bits 48 and 49 of register RS is placed into MSR_{48}. The result of ORing bits 58 and 49 of register RS is placed into MSR_{58}. The result of ORing bits 59 and 49 of register RS is placed into MSR_{59}. Bits 32:47, 49:50, 52:57, and 60:63 of register RS are placed into the corresponding bits of the MSR.

L=1:

Bits 48 and 62 of register RS are placed into the corresponding bits of the MSR. The remaining bits of the MSR are unchanged.

This instruction is privileged.

If L=0 this instruction is context synchronizing except with respect to alterations to the LE bit; see Chapter 7. “Synchronization Requirements for Context Alterations” on page 85. If L=1 this instruction is execution synchronizing; in addition, the alterations of the EE and RI bits take effect as soon as the instruction completes.

Special Registers Altered:

MSR

Except in the ***mtmsr*** instruction description in this section, references to “***mtmsr***” in Books I - III imply either L value unless otherwise stated or obvious from context (e.g., a reference to an ***mtmsr*** instruction that modifies an MSR bit other than the EE or RI bit implies L=0).

Programming Note

Warning: The first Programming Note in the ***mtmsrd*** instruction description applies to ***mtmsr*** as well as to ***mtmsrd***. Therefore software that uses ***mtmsr*** and will run on such processors must obey the rules given in that Programming Note.

Programming Note

If this instruction sets MSR_{PR} to 1, it also sets MSR_{EE}, MSR_{IR}, and MSR_{DR} to 1. On processors prior to POWER4+, the setting of MSR_{PR} does not affect the setting of MSR_{EE}.

This instruction does not alter MSR_{ME}. (This instruction does not alter MSR_{HV} because it does not alter any of the high-order 32 bits of the MSR.)

If the only MSR bits to be altered are $MSR_{EE\ RI}$, to obtain the best performance L=1 should be used.

Programming Note

If MSR_{EE}=0 and an External or Decrementer exception is pending, executing an ***mtmsr*** instruction that sets MSR_{EE} to 1 will cause the External or Decrementer interrupt to occur before the next instruction is executed, if no higher priority exception exists (see Section 5.8, “Interrupt Priorities” on page 76). Similarly, if a Hypervisor Decrementer interrupt is pending, execution of the instruction by the hypervisor causes a Hypervisor Decrementer interrupt to occur if HDICE=1.

For a discussion of software synchronization requirements when altering certain MSR bits, see Chapter 7.

Programming Note

mtmsr serves as both a basic and an extended mnemonic. The Assembler will recognize an ***mtmsr*** mnemonic with two operands as the basic form, and an ***mtmsr*** mnemonic with one operand as the extended form. In the extended form the L operand is omitted and assumed to be 0.

Programming Note

There is no need for an analogous version of the ***mfmsr*** instruction, because the existing instruction copies the entire contents of the MSR to the selected GPR.

8.4 Fixed-Point Storage Access Instructions

WARNING: These instructions are optional. They are not implemented in all processors. These instructions should be used only in code contained in a Programming Abstraction Layer (PAL) (i.e., code that is used only on an implementation-dependent basis).

Load Quadword DQ-form

lq RT,DQ(RA)

56	RT	RA	DQ	//
0	6	11	16	28 31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + EXTS(DQ || 0b0000)
RT ←  MEM(EA, 8)
GPR(RT+1) ← MEM(EA+8, 8)
```

Let the effective address (EA) be the sum (RA|0)+ (DQ||0b0000). The quadword in storage addressed by EA is loaded into registers RT and RT+1, in increasing order of storage address and register number.

EA must be a multiple of 16. If it is not, the system alignment error handler is invoked.

If RT is odd or RT=RA, the instruction form is invalid. If RT=RA, an attempt to execute this instruction will invoke the system illegal instruction error handler. (The RT=RA case includes the case of RT=RA=0.)

This instruction is privileged.

This instruction is optional.

Special Registers Altered:
None

Store Quadword DS-form

stq RS,DS(RA)

62	RS	RA	DS	2
0	6	11	16	30 31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + EXTS(DS || 0b00)
MEM(EA, 8) ← RS
MEM(EA+8, 8) ← GPR(RS+1)
```

Let the effective address (EA) be the sum (RA|0)+ (DS||0b00). (RS) and (RS+1) are stored into the quadword in storage addressed by EA, in increasing order of storage address and register number.

EA must be a multiple of 16. If it is not, the system alignment error handler is invoked.

If RS is odd, the instruction form is invalid.

This instruction is privileged.

This instruction is optional.

Special Registers Altered:
None

Additional Notes

The following descriptions are additions to the architecture associated with the ***lq*** and ***stq*** instructions and not replacements for existing descriptions.

Data Storage Interrupt

The effective address specified by a ***lq*** or ***stq*** instruction refers to storage that is Write Through Required or Caching Inhibited. $DSISR_5$: Set to 1 if the access was caused by the execution of a ***lq*** or ***stq*** instruction.

Alignment Interrupt

The operand of a ***lq*** or ***stq*** instruction is not aligned. The instruction is a ***lq*** or ***stq*** and the operand is in storage that is Write Through Required or Caching Inhibited.

Note: ***lq*** and ***stq*** instructions may cause either a Data Storage or Alignment interrupt if the operand is in storage that is Write Through Required or Caching Inhibited.

Chapter 9. Optional Facilities and Instructions that are being Phased Out

The facilities and instructions described in this chapter are optional. An implementation may provide all, some, or none of them.

Warning: These facilities and instructions are being phased out of the architecture.

The facilities and instructions described in this chapter are generally not mentioned elsewhere in Books I - III. Any conflict between this chapter and other parts of the Books is deemed to be resolved in favor of this chapter.

9.1 Bridge to SLB Architecture

The facility described in this section can be used to ease the transition to the current PowerPC software-managed Segment Lookaside Buffer (SLB) architecture, from the Segment Register architecture provided by 32-bit PowerPC implementations.

The facility permits the operating system to continue to use the 32-bit PowerPC implementation's *Segment Register Manipulation* instructions.

Programming Note

Warning: This facility is being phased out of the architecture. It is likely not to be supported on future implementations. New programs should not use it.

9.1.1 Segment Register Manipulation Instructions

The instructions described in this section -- ***mtsr***, ***mtsrin***, ***mfsr***, and ***mfsrin*** -- allow software to associate effective segments 0 through 15 with any of virtual segments 0 through 2^{27}-1. SLB entries 0:15 serve as virtual Segment Registers, with SLB entry i used to emulate Segment Register i. The ***mtsr*** and ***mtsrin*** instructions move 32 bits from a selected GPR to a selected SLB entry. The ***mfsr*** and ***mfsrin*** instructions move 32 bits from a selected SLB entry to a selected GPR.

The contents of the GPRs used by the instructions described in this section are shown in Figure 42. Fields shown as zeros must be zero for the *Move To Segment Register* instructions. Fields shown as hyphens are ignored. Fields shown as periods are ignored by the *Move To Segment Register* instructions and set to zero by the *Move From Segment Register* instructions. Fields shown as colons are ignored by the *Move To Segment Register* instructions and set to undefined values by the *Move From Segment Register* instructions.

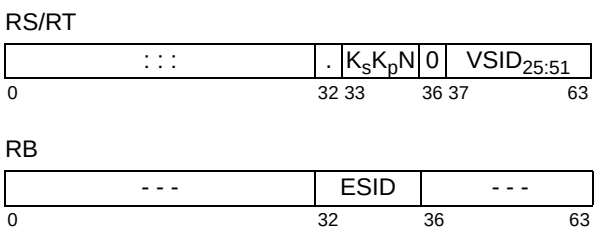


Figure 42. GPR contents for *mtsr*, *mtsrin*, *mfsr*, and *mfsrin*

Programming Note

The "Segment Register" format used by the instructions described in this section corresponds to the low-order 32 bits of RS and RT shown in the figure. This format is essentially the same as that for the Segment Registers of 32-bit PowerPC implementations. The only differences are the following.

- Bit 36 corresponds to a reserved bit in Segment Registers. Software must supply 0 for the bit because it corresponds to the L bit in SLB entries, and large pages are not supported for SLB entries created by the *Move To Segment Register* instructions.
- VSID bits 25:27 correspond to reserved bits in Segment Registers. Software can use these extra VSID bits to create VSIDs that are larger than those supported by the *Segment Register Manipulation* instructions of 32-bit PowerPC implementations.

Bit 32 of RS and RT corresponds to the T (direct-store) bit of early 32-bit PowerPC implementations. No corresponding bit exists in SLB entries.

Programming Note

The Programming Note in the introduction to Section 4.11.3.1, "SLB Management Instructions" on page 49 applies also to the *Segment Register Manipulation* instructions described in this section, and to any combination of the instructions described in the two sections, except as specified below for ***mfsr*** and ***mfsrin***.

The requirement that the SLB contain at most one entry that translates a given effective address (see Section 4.4.1, "Segment Lookaside Buffer (SLB)" on page 33) applies to SLB entries created by ***mtsr*** and ***mtsrin***. This requirement is satisfied naturally if only ***mtsr*** and ***mtsrin*** are used to create SLB entries for a given ESID, because for these instructions the association between SLB entries and ESID values is fixed (SLB entry i is used for ESID i). However, care must be taken if ***slbmte*** is also used to create SLB entries for the ESID, because for ***slbmte*** the association between SLB entries and ESID values is specified by software.

Move To Segment Register X-form

mtsr SR,RS

31	RS	/	SR	///	210	/
0	6	11	12	16	21	31

The SLB entry specified by SR is loaded from register RS, as follows.

SLBE Bit(s)	Set to	SLB Field(s)
0:31	0x0000_0000	$ESID_{0:31}$
32:35	SR	$ESID_{32:35}$
36	0b1	V
37:61	0x00_0000||0b0	$VSID_{0:24}$
62:88	$(RS)_{37:63}$	$VSID_{25:51}$
89:91	$(RS)_{33:35}$	K_sK_pN
92	$(RS)_{36}$	L ($(RS)_{36}$ must be 0b0)
93	0b0	C

MSR_{SF} must be 0 when this instruction is executed; otherwise the results are boundedly undefined.

This instruction is privileged.

Special Registers Altered:

None

Move To Segment Register Indirect X-form

mtsrin RS,RB
[POWER mnemonic: mtsri]

31	RS	///	RB	242	/
0	6	11	16	21	31

The SLB entry specified by $(RB)_{32:35}$ is loaded from register RS, as follows.

SLBE Bit(s)	Set to	SLB Field(s)
0:31	0x0000_0000	$ESID_{0:31}$
32:35	$(RB)_{32:35}$	$ESID_{32:35}$
36	0b1	V
37:61	0x00_0000||0b0	$VSID_{0:24}$
62:88	$(RS)_{37:63}$	$VSID_{25:51}$
89:91	$(RS)_{33:35}$	K_sK_pN
92	$(RS)_{36}$	L ($(RS)_{36}$ must be 0b0)
93	0b0	C

MSR_{SF} must be 0 when this instruction is executed; otherwise the results are boundedly undefined.

This instruction is privileged.

Special Registers Altered:

None

Move From Segment Register X-form

mfsr RT,SR

31	RT	/	SR	///	595	/
0	6	11	12	16	21	31

The contents of the low-order 27 bits of the VSID field, and the contents of the K_s, K_p, N, and L fields, of the SLB entry specified by SR are placed into register RT, as follows.

SLBE Bit(s)	Copied to	SLB Field(s)
62:88	$RT_{37:63}$	$VSID_{25:51}$
89:91	$RT_{33:35}$	K_sK_pN
92	RT_{36}	L ($SLBE_L$ must be 0b0)

RT_{32} is set to 0. The contents of $RT_{0:31}$ are undefined.

MSR_{SF} must be 0 when this instruction is executed; otherwise the results are boundedly undefined.

This instruction must be used only to read an SLB entry that was, or could have been, created by ***mtsr*** or ***mtsrin*** and has not subsequently been invalidated (i.e., an SLB entry in which ESID<16, V=1, VSID<2^{27}, L=0, and C=0). Otherwise the contents of register RT are undefined.

This instruction is privileged.

Special Registers Altered:

None

Move From Segment Register Indirect X-form

mfsrin RT,RB

31	RT	///	RB	659	/
0	6	11	16	21	31

The contents of the low-order 27 bits of the VSID field, and the contents of the K_s, K_p, N, and L fields, of the SLB entry specified by $(RB)_{32:35}$ are placed into register RT, as follows.

SLBE Bit(s)	Copied to	SLB Field(s)
62:88	$RT_{37:63}$	$VSID_{25:51}$
89:91	$RT_{33:35}$	K_sK_pN
92	RT_{36}	L ($SLBE_L$ must be 0b0)

RT_{32} is set to 0. The contents of $RT_{0:31}$ are undefined.

MSR_{SF} must be 0 when this instruction is executed; otherwise the results are boundedly undefined.

This instruction must be used only to read an SLB entry that was, or could have been, created by ***mtsr*** or ***mtsrin*** and has not subsequently been invalidated (i.e., an SLB entry in which ESID<16, V=1, VSID<2^{27}, L=0, and C=0). Otherwise the contents of register RT are undefined.

This instruction is privileged.

Special Registers Altered:

None

Appendix A. Assembler Extended Mnemonics

In order to make assembler language programs simpler to write and easier to understand, a set of extended mnemonics and symbols is provided for certain instructions. This appendix defines extended mnemonics and symbols related to instructions defined in Book III.

Assemblers should provide the extended mnemonics and symbols listed here, and may provide others.

A.1 Move To/From Special Purpose Register Mnemonics

This section defines extended mnemonics for the ***mtspr*** and ***mfspr*** instructions, including the Special Purpose Registers (SPRs) defined in Book I and certain privileged SPRs, and for the *Move From Time Base* instruction defined in Book II.

The ***mtspr*** and ***mfspr*** instructions specify an SPR as a numeric operand; extended mnemonics are provided that represent the SPR in the mnemonic rather than requiring it to be coded as an operand. Similar extended mnemonics are provided for the *Move From Time Base* instruction, which specifies the portion of the Time Base as a numeric operand.

Note: ***mftb*** serves as both a basic and an extended mnemonic. The Assembler will recognize an ***mftb*** mnemonic with two operands as the basic form, and an ***mftb*** mnemonic with one operand as the extended form. In the extended form the TBR operand is omitted and assumed to be 268 (the value that corresponds to TB).

Programming Note

The extended mnemonics in Table 3 for SPRs associated with the Performance Monitor facility are based on the definitions in Appendix E.

Other versions of Performance Monitor facilities used different sets of SPR numbers (all 32-bit PowerPC processors used a different set, and some early PowerPC processors used yet a different set).

Table 3: Extended mnemonics for moving to/from an SPR

Special Purpose Register	Move To SPR		Move From SPR[1]	
	Extended	Equivalent to	Extended	Equivalent to
Fixed-Point Exception Register	mtxer Rx	mtspr 1,Rx	mfxer Rx	mfspr Rx,1
Link Register	mtlr Rx	mtspr 8,Rx	mflr Rx	mfspr Rx,8
Count Register	mtctr Rx	mtspr 9,Rx	mfctr Rx	mfspr Rx,9
Data Storage Interrupt Status Register	mtdsisr Rx	mtspr 18,Rx	mfdsisr Rx	mfspr Rx,18
Data Address Register	mtdar Rx	mtspr 19,Rx	mfdar Rx	mfspr Rx,19
Decrementer	mtdec Rx	mtspr 22,Rx	mfdec Rx	mfspr Rx,22
Storage Description Register 1	mtsdr1 Rx	mtspr 25,Rx	mfsdr1 Rx	mfspr Rx,25
Save/Restore Register 0	mtsrr0 Rx	mtspr 26,Rx	mfsrr0 Rx	mfspr Rx,26
Save/Restore Register 1	mtsrr1 Rx	mtspr 27,Rx	mfsrr1 Rx	mfspr Rx,27
ACCR	mtaccr Rx	mtspr 29,Rx	mfaccr Rx	mfspr Rx,29
CTRL	mtctrl Rx	mtspr 152,Rx	mfctrl Rx	mfspr Rx,136
Special Purpose Registers G0 through G3	mtsprg n,Rx	mtspr 272+n,Rx	mfsprg Rx,n	mfspr Rx,272+n
Time Base [Lower]	mttbl Rx	mtspr 284,Rx	mftb Rx	mftb Rx,268
Time Base Upper	mttbu Rx	mtspr 285,Rx	mftbu Rx	mftb Rx,269
Processor Version Register	-	-	mfpvr Rx	mfspr Rx,287
MMCRA	mtmmcra Rx	mtspr 786,Rx	mfmmcra Rx	mfspr Rx,770
PMC1	mtpmc1 Rx	mtspr 787,Rx	mfpmc1 Rx	mfspr Rx,771
PMC2	mtpmc2 Rx	mtspr 788,Rx	mfpmc2 Rx	mfspr Rx,772
PMC3	mtpmc3 Rx	mtspr 789,Rx	mfpmc3 Rx	mfspr Rx,773
PMC4	mtpmc4 Rx	mtspr 790,Rx	mfpmc4 Rx	mfspr Rx,774
PMC5	mtpmc5 Rx	mtspr 791,Rx	mfpmc5 Rx	mfspr Rx,775
PMC6	mtpmc6 Rx	mtspr 792,Rx	mfpmc6 Rx	mfspr Rx,776
MMCR0	mtmmcr0 Rx	mtspr 795,Rx	mfmmcr0 Rx	mfspr Rx,779
MMCR1	mtmmcr1 Rx	mtspr 798,Rx	mfmmcr1 Rx	mfspr Rx,782
Processor Identification Register	-	-	mfpir Rx	mfspr Rx,1023

[1] Except for ***mftb*** and ***mftbu***.

Appendix B. Cross-Reference for Changed POWER Mnemonics

The following table lists the POWER instruction mnemonics that have been changed in the PowerPC Operating Environment Architecture, sorted by POWER mnemonic.

To determine the PowerPC mnemonic for one of these POWER mnemonics, find the POWER mnemonic in the second column of the table: the remainder of the line gives the PowerPC mnemonic and the page on which the instruction is described, as well as the instruction names.

POWER mnemonics that have not changed are not listed. POWER instruction names that are the same in PowerPC are not repeated: i.e., for these, the last column of the table is blank.

Page	POWER		PowerPC	
	Mnemonic	**Instruction**	**Mnemonic**	**Instruction**
95	mtsri	Move To Segment Register Indirect	mtsrin	
12	svca	Supervisor Call	sc	System Call
53	tlbi	TLB Invalidate Entry	tlbie	

Appendix C. New and Newly Optional Instructions

The following instructions in the PowerPC Operating Environment Architecture are new; they are not in the POWER Architecture. The ***tlbia***, ***tlbsync***, and ***mtmsr*** instructions are optional. In addition, the ***mfsr***, ***mfsrin***, ***mtsr***, and ***mtsrin*** instructions may optionally be provided as part of a "bridge" facility as described in Section 9.1, "Bridge to SLB Architecture" on page 93.

hrfid	Hypervisor Return From Interrupt Doubleword
mfsrin	Move From Segment Register Indirect
mtmsrd	Move To Machine State Register Doubleword
rfid	Return From Interrupt Doubleword
slbia	SLB Invalidate All
slbie	SLB Invalidate Entry
slbmfee	SLB Move From Entry ESID
slbmfev	SLB Move From Entry VSID
slbmte	SLB Move To Entry
tlbia	TLB Invalidate All
tlbsync	TLB Synchronize

Appendix D. Interpretation of the DSISR as Set by an Alignment Interrupt

For most causes of Alignment interrupt, the interrupt handler will emulate the interrupting instruction. To do this, it needs the following characteristics of the interrupting instruction:

Load or store
Length (halfword, word, doubleword)
String, multiple, or elementary
Fixed-point or floating-point
Update or non-update
Byte reverse or not
Is it ***dcbz***?

The PowerPC Architecture optionally provides this information by setting bits in the DSISR that identify the interrupting instruction type. It is not necessary for the interrupt handler to load the interrupting instruction from storage. The mapping is unique except for a few exceptions that are discussed below. The near-uniqueness depends on the fact that many instructions, such as the fixed- and floating-point arithmetic instructions and the one-byte loads and stores, cannot cause an Alignment interrupt.

See Section 5.5.8, “Alignment Interrupt” on page 69 for a description of how the opcode and extended opcode are mapped to a DSISR value for an X-, D-, or DS-form instruction that causes an Alignment interrupt.

The table on the next page shows the inverse mapping: how the DSISR bits identify the interrupting instruction. The following notes are cited in the table.

(1) The instructions ***lwz*** and ***lwarx*** give the same DSISR bits (all zero). But if ***lwarx*** causes an Alignment interrupt, it should not be emulated. It is adequate for the Alignment interrupt handler simply to treat the instruction as if it were ***lwz***. The emulator must use the address in the DAR, rather than compute it from RA/RB/D, because ***lwz*** and ***lwarx*** have different instruction formats.

If opcode 0 (“Illegal or Reserved”) can cause an Alignment interrupt, it will be indistinguishable to the interrupt handler from ***lwarx*** and ***lwz***.

(2) These are distinguished by DSISR bits 12:13, which are not shown in the table.

The interrupt handler has no need to distinguish between an X-form instruction and the corresponding D- or DS-form instruction if one exists, and vice versa. Therefore two such instructions may yield the same DSISR value (all 32 bits). For example, ***stw*** and ***stwx*** may both yield either the DSISR value shown in the following table for ***stw***, or that shown for ***stwx***.

If DSISR 15:21 is:	then it is either X-form opcode:	or D/DS-form opcode:	so the instruction is:
00 0 0000	00000xxx00	x00000	lwarx, lwz, reserved (1)
00 0 0001	00010xxx00	x00010	ldarx
00 0 0010	00100xxx00	x00100	stw
00 0 0011	00110xxx00	x00110	-
00 0 0100	01000xxx00	x01000	lhz
00 0 0101	01010xxx00	x01010	lha
00 0 0110	01100xxx00	x01100	sth
00 0 0111	01110xxx00	x01110	lmw
00 0 1000	10000xxx00	x10000	lfs
00 0 1001	10010xxx00	x10010	lfd
00 0 1010	10100xxx00	x10100	stfs
00 0 1011	10110xxx00	x10110	stfd
00 0 1100	11000xxx00	x11000	-
00 0 1101	11010xxx00	x11010	ld, ldu, lwa (2)
00 0 1110	11100xxx00	x11100	-
00 0 1111	11110xxx00	x11110	std, stdu (2)
00 1 0000	00001xxx00	x00001	lwzu
00 1 0001	00011xxx00	x00011	-
00 1 0010	00101xxx00	x00101	stwu
00 1 0011	00111xxx00	x00111	-
00 1 0100	01001xxx00	x01001	lhzu
00 1 0101	01011xxx00	x01011	lhau
00 1 0110	01101xxx00	x01101	sthu
00 1 0111	01111xxx00	x01111	stmw
00 1 1000	10001xxx00	x10001	lfsu
00 1 1001	10011xxx00	x10011	lfdu
00 1 1010	10101xxx00	x10101	stfsu
00 1 1011	10111xxx00	x10111	stfdu
00 1 1100	11001xxx00	x11001	-
00 1 1101	11011xxx00	x11011	-
00 1 1110	11101xxx00	x11101	-
00 1 1111	11111xxx00	x11111	-
01 0 0000	00000xxx01		ldx
01 0 0001	00010xxx01		-
01 0 0010	00100xxx01		stdx
01 0 0011	00110xxx01		-
01 0 0100	01000xxx01		-
01 0 0101	01010xxx01		lwax
01 0 0110	01100xxx01		-
01 0 0111	01110xxx01		-
01 0 1000	10000xxx01		lswx
01 0 1001	10010xxx01		lswi
01 0 1010	10100xxx01		stswx
01 0 1011	10110xxx01		stswi
01 0 1100	11000xxx01		-
01 0 1101	11010xxx01		-
01 0 1110	11100xxx01		-
01 0 1111	11110xxx01		-
01 1 0000	00001xxx01		ldux
01 1 0001	00011xxx01		-
01 1 0010	00101xxx01		stdux
01 1 0011	00111xxx01		-
01 1 0100	01001xxx01		-
01 1 0101	01011xxx01		lwaux
01 1 0110	01101xxx01		-
01 1 0111	01111xxx01		-
01 1 1000	10001xxx01		-
01 1 1001	10011xxx01		-
01 1 1010	10101xxx01		-
01 1 1011	10111xxx01		-
01 1 1100	11001xxx01		-
01 1 1101	11011xxx01		-
01 1 1110	11101xxx01		-
01 1 1111	11111xxx01		-

If DSISR 15:21 is:	then it is either X-form opcode:	or D/DS-form opcode:	so the instruction is:
10 0 0000	00000xxx10		-
10 0 0001	00010xxx10		-
10 0 0010	00100xxx10		stwcx.
10 0 0011	00110xxx10		stdcx.
10 0 0100	01000xxx10		-
10 0 0101	01010xxx10		-
10 0 0110	01100xxx10		-
10 0 0111	01110xxx10		-
10 0 1000	10000xxx10		lwbrx
10 0 1001	10010xxx10		-
10 0 1010	10100xxx10		stwbrx
10 0 1011	10110xxx10		-
10 0 1100	11000xxx10		lhbrx
10 0 1101	11010xxx10		-
10 0 1110	11100xxx10		sthbrx
10 0 1111	11110xxx10		-
10 1 0000	00001xxx10		-
10 1 0001	00011xxx10		-
10 1 0010	00101xxx10		-
10 1 0011	00111xxx10		-
10 1 0100	01001xxx10		eciwx
10 1 0101	01011xxx10		-
10 1 0110	01101xxx10		ecowx
10 1 0111	01111xxx10		-
10 1 1000	10001xxx10		-
10 1 1001	10011xxx10		-
10 1 1010	10101xxx10		-
10 1 1011	10111xxx10		-
10 1 1100	11001xxx10		-
10 1 1101	11011xxx10		-
10 1 1110	11101xxx10		-
10 1 1111	11111xxx10		dcbz
11 0 0000	00000xxx11		lwzx
11 0 0001	00010xxx11		-
11 0 0010	00100xxx11		stwx
11 0 0011	00110xxx11		-
11 0 0100	01000xxx11		lhzx
11 0 0101	01010xxx11		lhax
11 0 0110	01100xxx11		sthx
11 0 0111	01110xxx11		-
11 0 1000	10000xxx11		lfsx
11 0 1001	10010xxx11		lfdx
11 0 1010	10100xxx11		stfsx
11 0 1011	10110xxx11		stfdx
11 0 1100	11000xxx11		-
11 0 1101	11010xxx11		-
11 0 1110	11100xxx11		-
11 0 1111	11110xxx11		stfiwx
11 1 0000	00001xxx11		lwzux
11 1 0001	00011xxx11		-
11 1 0010	00101xxx11		stwux
11 1 0011	00111xxx11		-
11 1 0100	01001xxx11		lhzux
11 1 0101	01011xxx11		lhaux
11 1 0110	01101xxx11		sthux
11 1 0111	01111xxx11		-
11 1 1000	10001xxx11		lfsux
11 1 1001	10011xxx11		lfdux
11 1 1010	10101xxx11		stfsux
11 1 1011	10111xxx11		stfdux
11 1 1100	11001xxx11		-
11 1 1101	11011xxx11		-
11 1 1110	11101xxx11		-
11 1 1111	11111xxx11		-

Appendix E. Example Performance Monitor (Optional)

A Performance Monitor facility provides a means of collecting information about program and system performance.

The resources (e.g., SPR numbers) that a Performance Monitor facility may use are identified elsewhere in this Book. All other aspects of any Performance Monitor facility are implementation-dependent, and are described in the Book IV, *PowerPC Implementation Features* document for the implementation.

This appendix provides an example of a Performance Monitor facility. It is only an example; implementations may provide all, some, or none of the features described here, or may provide features that are similar to those described here but differ in detail.

Programming Note

Because the features provided by a Performance Monitor facility are implementation-dependent, operating systems should provide services that support the useful performance monitoring functions in a generic fashion. Application programs should use these services, and should not depend on the features provided by a particular implementation.

The example Performance Monitor facility consists of the following features (described in detail in subsequent sections).

- one MSR bit
  - PMM (Performance Monitor Mark), which can be used to select one or more programs for monitoring
- SPRs
  - PMC1 - PMC6 (Performance Monitor Counter registers 1 - 6), which count events
  - MMCR0, MMCR1, and MMCRA (Monitor Mode Control Registers 0, 1, and A), which control the Performance Monitor facility
  - SIAR and SDAR (Sampled Instruction Address Register and Sampled Data Address Register), which contain the address of the "sampled instruction" and of the "sampled data"
- the Performance Monitor interrupt, which can be caused by monitored conditions and events

The minimal subset of the features that makes the resulting Performance Monitor useful to software consists of MSR_{PMM}, PMC1, PMC2, PMC3, PMC4, MMCR0, MMCR1, and MMCRA and certain bits and fields of these three Monitor Mode Control Registers, and the Performance Monitor Interrupt. These features are identified as the "basic" features below. The remaining features (the remaining SPRs, and the remaining bits and fields in the three Monitor Mode Control Registers) are considered "extensions".

The events that can be counted in the PMCs are implementation-dependent. The Book IV, *PowerPC Implementation Features* document for the implementation describes the events that are available for each PMC, and also the code that identifies each event. The events and codes may vary between PMCs, as well as between implementations. For the programmable PMCs, the event to be counted is selected by specifying the appropriate code in the MMCR "Selector" field for the PMC. As described in Book IV, some events may include operations that are performed out-of-order.

Many aspects of the operation of the Performance Monitor are summarized by the following hierarchy, which is described starting at the lowest level.

- A "counter negative condition" exists when the value in a PMC is negative (i.e., when bit 0 of the PMC is 1). A "Time Base transition event" occurs when a selected bit of the Time Base changes from 0 to 1 (the bit is selected by an MMCR field). The term "condition or event" is used as an abbreviation for "counter negative condition or Time Base transition event". A condition or event can be caused implicitly by the processor (e.g., incrementing a PMC) or explicitly by software (***mtspr***).
- A condition or event is enabled if the corresponding "Enable" bit in an MMCR is 1. The occurrence of an enabled condition or event can have side effects within the Performance Monitor, such as causing the PMCs to cease counting.
- An enabled condition or event causes a Performance Monitor alert if Performance Monitor events are enabled by the corresponding "Enable" bit in an MMCR. A single Performance Monitor alert

may reflect multiple enabled conditions and events.

- A Performance Monitor alert causes a Performance Monitor exception.

  The exception effects of the Performance Monitor are said to be consistent with the contents of $MMCR0_{PMAO}$ if one of the following statements is true. ($MMCR0_{PMAO}$ reflects the occurrence of Performance Monitor events; see the definition of that bit in Section E.2.1.1.)
  - $MMCR0_{PMAO}$=0 and a Performance Monitor exception does not exist.
  - $MMCR0_{PMAO}$=1 and a Performance Monitor exception exists.

  A context synchronizing instruction or event that occurs when $MMCR0_{PMAO}$=0 ensures that the exception effects of the Performance Monitor are consistent with the contents of $MMCR0_{PMAO}$.

  Even without software synchronization, when the contents of $MMCR0_{PMAO}$ change, the exception effects of the Performance Monitor become consistent with the new contents of $MMCR0_{PMAO}$ sufficiently soon that the Performance Monitor facility is useful to software for its intended purposes.

- A Performance Monitor exception causes a Performance Monitor interrupt when MSR_{EE}=1.

Programming Note

The Performance Monitor can be effectively disabled (i.e., put into a state in which Performance Monitor SPRs are not altered and Performance Monitor interrupts do not occur) by setting MMCR0 to 0x0000_0000_8000_0000.

E.1 PMM Bit of the Machine State Register

The Performance Monitor uses MSR bit PMM, which is defined as follows.

Bit	Description
61	***Performance Monitor Mark*** (PMM) This bit is a basic feature. This bit contains the Performance Monitor "mark" (0 or 1).

Programming Note

Software can use this bit as a process-specific marker which, in conjunction with $MMCR0_{FCM0\ FCM1}$ (see Section E.2.1.1), permits events to be counted on a process-specific basis. (The bit is saved by interrupts and restored by ***rfid***.)

Common uses of the PMM bit include the following.

- Count events for a few selected processes. This use requires the following bit settings.
  - MSR_{PMM}=1 for the selected processes, MSR_{PMM}=0 for all other processes
  - $MMCR0_{FCM0}$=1
  - $MMCR0_{FCM1}$=0
- Count events for all but a few selected processes. This use requires the following bit settings.
  - MSR_{PMM}=1 for the selected processes, MSR_{PMM}=0 for all other processes
  - $MMCR0_{FCM0}$=0
  - $MMCR0_{FCM1}$=1

Notice that for both of these uses a mark value of 1 identifies the "few" processes and a mark value of 0 identifies the remaining "many" processes. Because the PMM bit is set to 0 when an interrupt occurs (see Figure 35 on page 65), interrupt handlers are treated as one of the "many". If it is desired to treat interrupt handlers as one of the "few", the mark value convention just described would be reversed.

E.2 Special Purpose Registers

The Performance Monitor SPRs count events, control the operation of the Performance Monitor, and provide associated information.

The Performance Monitor SPRs can be read and written using the ***mfspr*** and ***mtspr*** instructions (see Section 3.4.2, "Move To/From System Register Instructions" on page 18). The Performance Monitor SPR numbers are shown in Figure 43. Writing any of the Performance Monitor SPRs is privileged. Reading any of the Performance Monitor SPRs is *not* privileged (however, the privileged SPR numbers used to write the SPRs can also be used to read them; see the figure).

The elapsed time between the execution of an instruction and the time at which events due to that instruction have been reflected in Performance Monitor SPRs is not defined. No means are provided by which software can ensure that all events due to preceding instructions have been reflected in Performance Monitor SPRs. Similarly, if the events being monitored may be caused by operations that are performed out-of-order, no means are provided by which software can prevent such events due to subsequent instructions from being reflected in Performance Monitor SPRs. Thus the contents obtained by reading a Performance Monitor SPR may not be precise: it may fail to reflect some events due to instructions that precede the ***mfspr*** and may reflect some events due to instructions that follow the ***mfspr***. This lack of precision applies regardless of whether the state of the processor is such that the SPR is subject to change by the processor at the time the ***mfspr*** is executed. Similarly, if an ***mtspr*** instruction is executed that changes the contents of the Time Base, the change is not guaranteed to have taken effect with respect to causing Time Base transition events until after a subsequent context synchronizing instruction has been executed.

If an ***mtspr*** instruction is executed that changes the value of a Performance Monitor SPR other than SIAR or SDAR, the change is not guaranteed to have taken effect until after a subsequent context synchronizing instruction has been executed (see Chapter 7. "Synchronization Requirements for Context Alterations" on page 85).

Programming Note

Depending on the events being monitored, the contents of Performance Monitor SPRs may be affected by aspects of the runtime environment (e.g., cache contents) that are not directly attributable to the programs being monitored.

decimal	SPR[1,2] $spr_{5:9}$ $spr_{0:4}$	Register Name	Privileged
770,786	11000 n0010	MMCRA	no,yes
771,787	11000 n0011	PMC1	no,yes
772,788	11000 n0100	PMC2	no,yes
773,789	11000 n0101	PMC3	no,yes
774,790	11000 n0110	PMC4	no,yes
775,791	11000 n0111	PMC5	no,yes
776,792	11000 n1000	PMC6	no,yes
779,795	11000 n1011	MMCR0	no,yes
780,796	11000 n1100	SIAR	no,yes
781,797	11000 n1101	SDAR	no,yes
782,798	11000 n1110	MMCR1	no,yes

1 Note that the order of the two 5-bit halves of the SPR number is reversed.
2 For ***mtspr***, n must be 1. For ***mfspr***, reading the SPR is privileged if and only if n=1.

Figure 43. Performance Monitor SPR encodings for mtspr and mfspr

E.2.1 Performance Monitor Counter Registers

The six Performance Monitor Counter registers, PMC1 through PMC6, are 32-bit registers that count events.

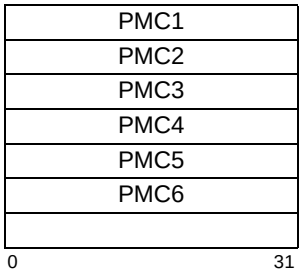


Figure 44. Performance Monitor Counter registers

PMC1, PMC2, PMC3, and PMC4 are basic features. PMC5 and PMC6 are not programmable. PMC5 counts instructions completed and PMC6 counts cycles, see Book IV for more detail.

Normally each PMC is incremented each processor cycle by the number of times the corresponding event occurred in that cycle. Other modes of incrementing may also be provided (e.g., see the description of MMCR1 bits PMC1HIST and PMCjHIST).

"PMCj" is used as an abbreviation for "PMCi, i > 1".

Programming Note

PMC5 and PMC6 are defined to facilitate calculating basic performance metrics such as cycles per instruction (CPI).

Programming Note

Software can use a PMC to “pace” the collection of Performance Monitor data. For example, if it is desired to collect event counts every n cycles, software can specify that a particular PMC count cycles and set that PMC to 0x8000_0000 - n. The events of interest would be counted in other PMCs. The counter negative condition that will occur after n cycles can, with the appropriate setting of MMCR bits, cause counter values to become frozen, cause a Performance Monitor interrupt to occur, etc.

E.2.1.1 Monitor Mode Control Register 0

Monitor Mode Control Register 0 (MMCR0) is a 64-bit register. This register, along with MMCR1 and MMCRA, controls the operation of the Performance Monitor.

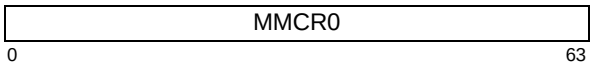


Figure 45. Monitor Mode Control Register 0

MMCR0 is a basic feature. Within MMCR0, some of the bits and fields are basic features and some are extensions. The basic bits and fields are identified as such, below.

Some bits of MMCR0 are altered by the processor when various events occur, as described below.

The bit definitions of MMCR0 are as follows. MMCR0 bits that are not implemented are treated as reserved.

Bit(s) Description

0:31 Reserved

32 ***Freeze Counters*** (FC)

This bit is a basic feature.

0 The PMCs are incremented (if permitted by other MMCR bits).
1 The PMCs are not incremented.

The processor sets this bit to 1 when an enabled condition or event occurs and $MMCR0_{FCECE}$=1.

33 ***Freeze Counters in Supervisor State*** (FCS)

This bit is a basic feature.

0 The PMCs are incremented (if permitted by other MMCR bits).
1 The PMCs are not incremented if $MSR_{HV\ PR}$=0b00.

34 ***Freeze Counters in Problem State*** (FCP)

This bit is a basic feature.

0 The PMCs are incremented (if permitted by other MMCR bits).
1 The PMCs are not incremented if MSR_{PR}=1.

35 ***Freeze Counters while Mark = 1*** (FCM1)

This bit is a basic feature.

0 The PMCs are incremented (if permitted by other MMCR bits).
1 The PMCs are not incremented if MSR_{PMM}=1.

36 ***Freeze Counters while Mark = 0*** (FCM0)

This bit is a basic feature.

0 The PMCs are incremented (if permitted by other MMCR bits).
1 The PMCs are not incremented if MSR_{PMM}=0.

37 ***Performance Monitor Alert Enable*** (PMAE)

This bit is a basic feature.

0 Performance Monitor events are disabled.
1 Performance Monitor events are enabled until a Performance Monitor event occurs, at which time:
- $MMCR0_{PMAE}$ is set to 0
- $MMCR0_{PMAO}$ is set to 1

Programming Note

Software can set this bit and $MMCR0_{PMAO}$ to 0 to prevent Performance Monitor interrupts.

Software can set this bit to 1 and then poll the bit to determine whether an enabled condition or event has occurred. This is especially useful for software that runs with MSR_{EE}=0.

38 ***Freeze Counters on Enabled Condition or Event*** (FCECE)

0 The PMCs are incremented (if permitted by other MMCR bits).
1 The PMCs are incremented (if permitted by other MMCR bits) until an enabled condition or event occurs when $MMCR0_{TRIGGER}$=0, at which time:
- $MMCR0_{FC}$ is set to 1

If the enabled condition or event occurs when $MMCR0_{TRIGGER}$=1, the FCECE bit is treated as if it were 0.

39:40 ***Time Base Selector*** (TBSEL)

This field selects the Time Base bit that can cause a Time Base transition event (the event occurs when the selected bit changes from 0 to 1).

00 Time Base bit 63 is selected.
01 Time Base bit 55 is selected.
10 Time Base bit 51 is selected.
11 Time Base bit 47 is selected.

Programming Note

Time Base transition events can be used to collect information about processor activity, as revealed by event counts in PMCs and by addresses in SIAR and SDAR, at periodic intervals.

In multiprocessor systems in which the Time Base registers are synchronized among the processors, Time Base transition events can be used to correlate the Performance Monitor data obtained by the several processors. For this use, software must specify the same TBSEL value for all the processors in the system.

Because the frequency of the Time Base is implementation-dependent, software should invoke a system service program to obtain the frequency before choosing a value for TBSEL.

41 ***Time Base Event Enable*** (TBEE)

0 Time Base transition events are disabled.
1 Time Base transition events are enabled.

42:47 Reserved

48 ***PMC1 Condition Enable*** (PMC1CE)

This bit controls whether counter negative conditions due to a negative value in PMC1 are enabled.

0 Counter negative conditions for PMC1 are disabled.
1 Counter negative conditions for PMC1 are enabled.

49 ***PMCj Condition Enable*** (PMCjCE)

This bit controls whether counter negative conditions due to a negative value in any PMCj (i.e., in any PMC except PMC1) are enabled.

0 Counter negative conditions for all PMCjs are disabled.
1 Counter negative conditions for all PMCjs are enabled.

50 ***Trigger*** (TRIGGER)

0 The PMCs are incremented (if permitted by other MMCR bits).
1 PMC1 is incremented (if permitted by other MMCR bits). The PMCjs are not incremented until PMC1 is negative or an enabled condition or event occurs, at which time:
- the PMCjs resume incrementing (if permitted by other MMCR bits)
- $MMCR0_{TRIGGER}$ is set to 0

See the description of the FCECE bit, above, regarding the interaction between TRIGGER and FCECE.

Programming Note

Uses of TRIGGER include the following.

- Resume counting in the PMCjs when PMC1 becomes negative, without causing a Performance Monitor interrupt. Then freeze all PMCs (and optionally cause a Performance Monitor interrupt) when a PMCj becomes negative. The PMCjs then reflect the events that occurred between the time PMC1 became negative and the time a PMCj becomes negative. This use requires the following MMCR0 bit settings.
  - TRIGGER=1
  - PMC1CE=0
  - PMCjCE=1
  - TBEE=0
  - FCECE=1
  - PMAE=1 (if a Performance Monitor interrupt is desired)
- Resume counting in the PMCjs when PMC1 becomes negative, and cause a Performance Monitor interrupt without freezing any PMCs. The PMCjs then reflect the events that occurred between the time PMC1 became negative and the time the interrupt handler reads them. This use requires the following MMCR0 bit settings.
  - TRIGGER=1
  - PMC1CE=1
  - TBEE=0
  - FCECE=0
  - PMAE=1

51:52 Setting is implementation-dependent; see Book IV.

53:55 Reserved

56 ***Performance Monitor Alert Occurred*** (PMAO)

This bit is a basic feature.

0 A Performance Monitor event has not occurred since the last time software set this bit to 0.
1 A Performance Monitor event has occurred since the last time software set this bit to 0.

This bit is set to 1 by the processor when a Performance Monitor event occurs. This bit can be set to 0 only by the ***mtspr*** instruction.

Programming Note

Software can set this bit to 1 to simulate the occurrence of a Performance Monitor event.

Software should set this bit to 0 after handling the Performance Monitor event.

This bit was first implemented in the POWER4+ processor.

57 Setting is implementation-dependent; see Book IV.

58 ***Freeze Counters 1-4*** (FC1-4)

0 PMC1 - PMC4 are incremented (if permitted by other MMCR bits).
1 PMC1 - PMC4 are not incremented.

59 ***Freeze Counters 5-6*** (FC5-6)

0 PMC5 - PMC6 are incremented (if permitted by other MMCR bits).
1 PMC5 - PMC6 are not incremented.

60:61 Reserved

62 ***Freeze Counters in Wait State*** (FCWAIT)

This bit is a basic feature.

0 The PMCs are incremented (if permitted by other MMCR bits).
1 The PMCs are not incremented if $CTRL_{31}$=0. Software is expected to set $CTRL_{31}$=0 when it is in a “wait state”, i.e, when there is no process ready to run.

Only Branch Unit type of events do not increment if $CTRL_{31}$=0. Other units continue to count.

63 ***Freeze Counters in Hypervisor State*** (FCH)

This bit is a basic feature.

0 The PMCs are incremented (if permitted by other MMCR bits).
1 The PMCs are not incremented if $MSR_{HV\ PR}$=0b10.

E.2.2 Monitor Mode Control Register 1

Monitor Mode Control Register 1 (MMCR1) is a 64-bit register. This register, along with MMCR0 and MMCRA, controls the operation of the Performance Monitor.

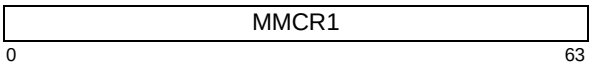


Figure 46. Monitor Mode Control Register 1

MMCR1 is a basic feature. Within MMCR1, some of the bits and fields are basic features and some are extensions. The basic bits and fields are identified as such, below.

Some bits of MMCR1 are altered by the processor when various events occur, as described below.

The bit definitions of MMCR1 are as follows. MMCR1 bits that are not implemented are treated as reserved.

Bit(s) Description

0:31 Implementation-Dependent Use

These bits have implementation-dependent uses (e.g., extended event selection).

32:39 ***PMC1 Selector*** (PMC3SEL)
40:47 ***PMC2 Selector*** (PMC4SEL)
48:55 ***PMC3 Selector*** (PMC5SEL)
56:63 ***PMC4 Selector*** (PMC6SEL)

Each of these fields contains a code that identifies the event to be counted by PMCs 1 through 4 respectively; see Book IV.

PMC Selectors are basic features.

Compatibility Note

In versions of the architecture that precede Version 2.02 the PMC Selector Fields were six bits long, and were split between MMCR0 and MMCR1. PMC1-8 were all programmable.

If more programmable PMCs are implemented in the future, additional MMCRs may be defined to cover the additional selectors.

E.2.3 Monitor Mode Control Register A

Monitor Mode Control Register A (MMCRA) is a 64-bit register. This register, along with MMCR0 and MMCR1, controls the operation of the Performance Monitor.

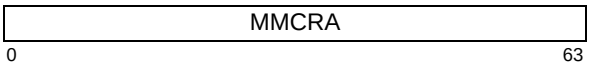


Figure 47. Monitor Mode Control Register A

MMCRA is a basic feature. Within MMCRA, some of the bits and fields are basic features and some are extensions. The basic bits and fields are identified as such, below.

Some bits of MMCRA are altered by the processor when various events occur, as described below.

The bit definitions of MMCRA are as follows. MMCRA bits that are not implemented are treated as reserved.

Bit(s) Description

0:31 Reserved

32 ***Contents of SIAR and SDAR Are Related*** (CSSR)

Set to 1 by the processor if the contents of SIAR and SDAR are associated with the same instruction; otherwise set to 0.

33:34 Setting is implementation-dependent; see Book IV.

35 ***Sampled MSR_{HV}*** (SAMPHV)

Value of MSR_{HV} when the Performance Monitor Alert occurred.

36 ***Sampled MSR_{PR}*** (SAMPPR)

Value of MSR_{PR} when the Performance Monitor Alert occurred.

37:47 Setting is implementation-dependent; see Book IV.

48:53 ***Threshold*** (THRESHOLD)

This field contains a “threshold value”, which is a value such that only events that exceed the value are counted. The events to which a threshold value can apply are implementation-dependent, as are the dimension of the threshold (e.g., duration in cycles) and the granularity with which the threshold value is interpreted. See the Book IV, *PowerPC Implementation Features* document for the implementation.

Programming Note

By varying the threshold value, software can obtain a profile of the characteristics of the events subject to the threshold. For example, if PMC1 counts the number of cache misses for which the duration exceeds the threshold value, then software can obtain the distribution of cache miss durations for a given program by monitoring the program repeatedly using a different threshold value each time.

54:59 Reserved for implementation-specific use.

60:62 Reserved

63 Setting is implementation-dependent; see Book IV.

E.2.4 Sampled Instruction Address Register

The Sampled Instruction Address Register (SIAR) is a 64-bit register. It contains the address of the “sampled instruction” when a Performance Monitor alert occurs.

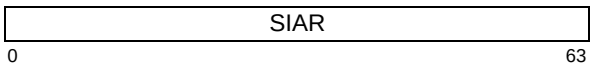


Figure 48. Sampled Instruction Address Register

When a Performance Monitor alert occurs, SIAR is set to the effective address of an instruction that was being executed, possibly out-of-order, at or around the time that the Performance Monitor alert occurred. This instruction is called the “sampled instruction”.

The contents of SIAR may be altered by the processor if and only if $MMCR0_{PMAE}$=1. Thus after the Performance Monitor alert occurs, the contents of SIAR are not altered by the processor until software sets $MMCR0_{PMAE}$ to 1. After software sets $MMCR0_{PMAE}$ to 1, the contents of SIAR are undefined until the next Performance Monitor alert occurs.

See Section E.4 regarding the effects of the Trace facility on SIAR.

Programming Note

If the Performance Monitor alert causes a Performance Monitor interrupt, the value of $MSR_{HV\ PR}$ that was in effect when the sampled instruction was being executed is reported in MMCRA.

E.2.5 Sampled Data Address Register

The Sampled Data Address Register (SDAR) is a 64-bit register. It contains the address of the "sampled data" when a Performance Monitor alert occurs.

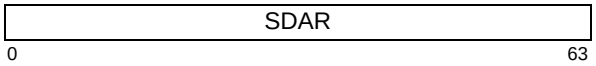


Figure 49. Sampled Data Address Register

When a Performance Monitor alert occurs, SDAR is set to the effective address of the storage operand of an instruction that was being executed, possibly out-of-order, at or around the time that the Performance Monitor alert occurred. This storage operand is called the "sampled data". The sampled data may be, but need not be, the storage operand (if any) of the sampled instruction (see Section E.2.4).

The contents of SDAR may be altered by the processor if and only if $MMCR0_{PMAE}$=1. Thus after the Performance Monitor alert occurs, the contents of SDAR are not altered by the processor until software sets $MMCR0_{PMAE}$ to 1. After software sets $MMCR0_{PMAE}$ to 1, the contents of SDAR are undefined until the next Performance Monitor alert occurs.

See Section E.4 regarding the effects of the Trace facility on SDAR.

Programming Note

If the Performance Monitor alert causes a Performance Monitor interrupt, MMCRA indicates whether the sampled data is the storage operand of the sampled instruction.

E.3 Performance Monitor Interrupt

The Performance Monitor interrupt is a system-caused interrupt (see Section 5.3, "Interrupt Classes" on page 62). It is masked by MSR_{EE} in the same manner that External and Decrementer interrupts are.

The Performance Monitor interrupt is a basic feature.

A Performance Monitor interrupt occurs when no higher priority exception exists, a Performance Monitor exception exists, and MSR_{EE}=1.

If multiple Performance Monitor exceptions occur before the first causes a Performance Monitor interrupt, the interrupt reflects the most recent Performance Monitor exception and the preceding Performance Monitor exceptions are lost.

The following registers are set:

SRR0 Set to the effective address of the instruction that the processor would have attempted to execute next if no interrupt conditions were present.

SRR1

33:36 and 42:47 See the Book IV, PowerPC Implementation Features document for the implementation.

Others Loaded from the MSR.

MSR See Figure 35 on page 65.

SIAR Set to the effective address of the "sampled instruction" (see Section E.2.4).

SDAR Set to the effective address of the "sampled data" (see Section E.2.5).

Execution resumes at effective address 0x0000_0000_0000_0F00.

In general, statements about External and Decrementer interrupts elsewhere in this Book apply also to the Performance Monitor interrupt; for example, if a Performance Monitor exception exists when an ***mtmsrd***[***d***] instruction is executed that changes MSR_{EE} from 0 to 1, the Performance Monitor interrupt will occur before the next instruction is executed (if no higher priority exception exists).

The priority of the Performance Monitor exception is equal to that of the External, Decrementer, and Hypervisor Decrementer exceptions (i.e., the processor may generate any one of the four interrupts for which an exception exists) (see Section 5.7.2, "Ordered Exceptions" on page 74 and Section 5.8, "Interrupt Priorities" on page 76).

E.4 Interaction with the Trace Facility

If the Trace facility includes setting SIAR and SDAR (see Appendix F, "Example Trace Extensions (Optional)" on page 117), and tracing is active (MSR_{SE}=1 or MSR_{BE}=1), the contents of SIAR and SDAR as used by the Performance Monitor facility are undefined and may change even when $MMCR0_{PMAE}$=0.

Programming Note

A potential combined use of the Trace and Performance Monitor facilities is to trace the control flow of a program and simultaneously count events for that program.

Appendix F. Example Trace Extensions (Optional)

This appendix provides an example of extensions that may be added to the Trace facility described in Section 5.5.14, “Trace Interrupt” on page 73. It is only an example; implementations may provide all, some, or none of the features described here, or may provide features that are similar to those described here but differ in detail. See the Book IV, *PowerPC Implementation Features* document for the implementation.

The extensions consist of the following features (described in detail below).

- use of $MSR_{SE\ BE}$=0b11 to specify new causes of Trace interrupts
- specification of how certain SRR1 bits are set when a Trace interrupt occurs
- setting of SIAR and SDAR (see Appendix E) when a Trace interrupt occurs

$MSR_{SE\ BE}$ = 0b11

If $MSR_{SE\ BE}$=0b11, the processor generates a Trace exception under the conditions described in Section 5.5.14 for $MSR_{SE\ BE}$=0b01, and also after successfully completing the execution of any instruction that would cause at least one of SRR1 bits 33:36, 42, and 44:46 to be set to 1 (see below) if the instruction were executed when $MSR_{SE\ BE}$=0b10.

This overrides the implicit statement in Section 5.5.14 that the effects of $MSR_{SE\ BE}$=0b11 are the same as those of $MSR_{SE\ BE}$=0b10.

SRR1

When a Trace interrupt occurs, the SRR1 bits that are not loaded from the MSR are set as follows instead of as described in Section 5.5.14.

33 Set to 1 if the traced instruction is ***icbi***; otherwise set to 0.

34 Set to 1 if the traced instruction is ***dcbt***, ***dcbtst***, ***dcbz***, ***dcbst***, ***dcbf***[***l***]; otherwise set to 0.

35 Set to 1 if the traced instruction is a Load instruction or ***eciwx***; may be set to 1 if the traced instruction is ***icbi***, ***dcbt***, ***dcbtst***, ***dcbst***, ***dcbf***[***l***]; otherwise set to 0.

36 Set to 1 if the traced instruction is a *Store* instruction, ***dcbz***, or ***ecowx***; otherwise set to 0.

42 Set to 1 if the traced instruction is ***lswx*** or ***stswx***; otherwise set to 0.

43 See the Book IV, *PowerPC Implementation Features* document for the implementation.

44 Set to 1 if the traced instruction is a *Branch* instruction and the branch is taken; otherwise set to 0.

45 Set to 1 if the traced instruction is ***eciwx*** or ***ecowx***; otherwise set to 0.

46 Set to 1 if the traced instruction is ***lwarx***, ***ldarx***, ***stwcx.***, or ***stdcx.***; otherwise set to 0.

47 See the Book IV, *PowerPC Implementation Features* document for the implementation.

SIAR and SDAR

If the optional Performance Monitor facility is implemented and includes SIAR and SDAR (see Appendix E. “Example Performance Monitor (Optional)” on page 107), the following additional registers are set when a Trace interrupt occurs:

SIAR Set to the effective address of the traced instruction.

SDAR Set to the effective address of the storage operand (if any) of the traced instruction; otherwise undefined.

If the state of the Performance Monitor is such that the Performance Monitor may be altering these registers (i.e., if $MMCR0_{PMAE}$=1), the contents of SIAR and SDAR as used by the Trace facility are undefined and may change even when no Trace interrupt occurs.

Appendix G. PowerPC Operating Environment Instruction Set

Form	Opcode		Mode Dep.[1]	Priv.[2]	Page	Mne-monic	Instruction
	Pri-mary	Extend					
XL	19	274		H	13	hrfid	Hypervisor Return From Interrupt Doubleword
X	31	83		P	23	mfmsr	Move From Machine State Register
XFX	31	339		O	21	mfspr	Move From Special Purpose Register
X	31	595	32	P	96	mfsr	Move From Segment Register
X	31	659	32	P	96	mfsrin	Move From Segment Register Indirect
X	31	146		P	91	mtmsr	Move To Machine State Register
X	31	178		P	22	mtmsrd	Move To Machine State Register Doubleword
XFX	31	467		O	19	mtspr	Move To Special Purpose Register
X	31	210	32	P	95	mtsr	Move To Segment Register
X	31	242	32	P	95	mtsrin	Move To Segment Register Indirect
XL	19	18		P	13	rfid	Return From Interrupt Doubleword
SC	17				12	sc	System Call
X	31	498		P	50	slbia	SLB Invalidate All
X	31	434		P	49	slbie	SLB Invalidate Entry
X	31	915		P	52	slbmfee	SLB Move From Entry ESID
X	31	851		P	52	slbmfev	SLB Move From Entry VSID
X	31	402		P	51	slbmte	SLB Move To Entry
X	31	370		P	55	tlbia	TLB Invalidate All
X	31	306	64	H	53	tlbie	TLB Invalidate Entry
X	31	566		H	56	tlbsync	TLB Synchronize

[1]Key to Mode Dependency Column

Except as described below and in the section entitled "Effective Address Calculation" in Book I, all instructions in the PowerPC Operating Environment Architecture are independent of whether the processor is in 32-bit or 64-bit mode.

32 The instruction must be executed only in 32-bit mode.

64 The instruction must be executed only in 64-bit mode.

[2]Key to Privilege Column

P denotes a privileged instruction.

O denotes an instruction that is treated as privileged or nonprivileged (or hypervisor, for ***mtspr***), depending on the SPR number.

H denotes an instruction that can be executed only in hypervisor state.

Index

F

H

I

O

P

R

S

T

V

W

Numerics

Last Page - End of Document