

# **The IBM PowerPC Embedded Environment**

**Architectural Specifications  
for IBM PowerPC Embedded Controllers**

**Preliminary**

73G7334 000001

## **Preliminary Edition (May 1997)**

This edition of the IBM PowerPC Embedded Environment provides architectural specifications for all implementations of the IBM PowerPC 400 series of Embedded Controllers unless otherwise indicated in new versions or technical newsletters. Note that some of the specifications in this document differ from the PowerPC Architecture as defined by the document "The PowerPC Architecture: A Specification for a New Family of RISC Processors." This document specifically does NOT define the PowerPC Architecture, and is instead a set of alternative specifications which are unique to IBM's PowerPC Embedded Controllers.

**The following paragraph does not apply to the United Kingdom or any country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS MANUAL "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions; therefore, this statement may not apply to you.**

IBM does not warrant that the products in this publication, whether individually or as one or more groups, will meet your requirements or that the publication or the accompanying product descriptions are error-free.

This publication could contain technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or program(s) described in this publication at any time.

It is possible that this publication may contain references to, or information about, IBM products (machines and programs), programming, or services that are not announced in your country. Such references or information must not be construed to mean that IBM intends to announce such IBM products, programming, or services in your country. Any reference to an IBM licensed program in this publication is not intended to state or imply that you can use only IBM's licensed program. You can use any functionally equivalent program instead.

No part of this publication may be reproduced or distributed in any form or by any means, or stored in a data base or retrieval system, without the written permission of IBM.

Requests for copies of this publication and for technical information about IBM products should be made to your IBM Authorized Dealer or your IBM Marketing Representative.

Address comments about this publication to:

IBM Corporation  
Department H83A  
P.O. Box 12195  
Research Triangle Park, NC 27709

IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

©Copyright International Business Machines Corporation 1993, 1994. All rights reserved.

Printed in the United States of America.

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

## **Patents and Trademarks**

IBM may have patents or pending patent applications covering the subject matter in this publication. The furnishing of this publication does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Licensing, IBM Corporation, 208 Harbor Drive, Stamford, CT 06904, United States of America.

The following terms are trademarks of IBM Corporation:

IBM

PowerPC

PowerPC Architecture

PowerPC Embedded Controllers

Other terms which are trademarks are the property of their respective owners.



# Contents

<b>About This Book .....</b>	<b>xv</b>
------------------------------	-----------

## **Part One: Background and Introduction**

<b>Overview .....</b>	<b>1-1</b>
-----------------------	------------

PowerPC Architecture Organization .....	1-2
Compatibility with the User Instruction Set Architecture .....	1-2
The IBM PowerPC Embedded Environment .....	1-3
The IBM PowerPC Embedded Virtual Environment .....	1-3
The IBM PowerPC Embedded Operating Environment .....	1-4

## **Part Two: The IBM PowerPC Embedded Virtual Environment**

<b>Storage Model .....</b>	<b>2-1</b>
----------------------------	------------

Storage Model .....	2-1
Virtual Storage .....	2-2
Single Copy Atomicity .....	2-3
Memory Coherence .....	2-4
Storage Control Attributes .....	2-5
Write Through .....	2-6
Cacheability .....	2-6
Memory Coherence .....	2-7
Guarded Storage .....	2-7
Endianness .....	2-8
Cache Models .....	2-8
Split or Dual Caches .....	2-9
Instruction Cache Block Invalidate (icbi) .....	2-9
Data Cache Block Touch (dcbt, dcbtst) .....	2-10
Data Cache Block Set to Zero (dcbz) .....	2-11
Data Cache Block Store (dcbst) .....	2-11
Data Cache Block Flush (dcbf) .....	2-12
Combined Cache .....	2-12
Write Through Data Cache .....	2-12
Write Through to Main Storage .....	2-12
Write Through to Multilevel Cache .....	2-13
Shared Storage .....	2-13
Storage Access Ordering .....	2-13
Enforce In-Order Execution of I/O (eieio) .....	2-14
Synchronize (sync) .....	2-14
Atomic Update Primitives .....	2-15
Reservations .....	2-16
Forward Progress .....	2-17

Reservation Loss Due to Granularity .....	2-18
Operand Placement .....	2-18
Instruction Restart .....	2-19
Access Atomicity .....	2-19
Access Order .....	2-19
Storage Control Instructions .....	2-20
Parameters Useful to Application Programs .....	2-20
Instruction Cache Management Instructions .....	2-21
Instruction Cache Block Invalidate (icbi) .....	2-22
Instruction Synchronize (isync) .....	2-23
Data Cache Management Instructions .....	2-24
Data Cache Block Flush (dcbf) .....	2-24
Data Cache Block Store (dcbst) .....	2-25
Data Cache Block Touch (dcbt) .....	2-26
Data Cache Block Touch for Store (dcbtst) .....	2-27
Data Cache Block set to Zero (dcbz) .....	2-28
Enforce In-order Execution of I/O (eieio) .....	2-29
<b>Time Base .....</b>	<b>3-1</b>
Time Base .....	3-1
Time Base Instructions .....	3-4
Move From Time Base (mftb) .....	3-4
Extended Mnemonics .....	3-5
Reading the Time Base .....	3-6
Computing Time of Day from the Time Base .....	3-6
Non-constant update frequency .....	3-8
<b>Instruction Set Summary.....</b>	<b>4-1</b>
Summary of Instructions .....	4-1
 <b>Part Three: The IBM PowerPC Embedded Operating Environment</b>	
<b>Integer Unit .....</b>	<b>5-1</b>
Special Purpose Registers .....	5-1
Privileged SPR Specification .....	5-3
Special Purpose Register General (SPRG0-SPRG3) .....	5-4
Processor Version Register (PVR) .....	5-5
Special Purpose Register Instructions .....	5-6
Move From Special Purpose Register (mfspr) .....	5-6
Move To Special Purpose Register (mtspr) .....	5-7
Device Control Registers .....	5-8
Device Control Register Instructions .....	5-8
Move From Device Control Register (mfdcr) .....	5-8
Move To Device Control Register (mtdcr) .....	5-9

<b>Storage Control.....</b>	<b>6-1</b>
Storage Addressing .....	6-1
Storage Model .....	6-1
Instruction Fetch .....	6-2
Implicit Branch .....	6-2
Data Storage Access .....	6-3
Performing Operations Out-of-Order .....	6-3
Guarded Storage .....	6-4
Out-of-Order Accesses to Guarded Storage .....	6-4
Real Addressing Mode .....	6-5
Invalid Real Address .....	6-5
Storage Attributes .....	6-6
W, I, M, G and E bits .....	6-6
Supported Storage Modes .....	6-7
Mismatched WIMGE Bits .....	6-7
Storage Attribute Control Registers .....	6-8
Address Translation and Storage Protection .....	6-10
Virtual to Real Address Translation .....	6-10
Translation Lookaside Buffer (TLB) .....	6-11
TLB Fields .....	6-13
Page Identification Fields .....	6-13
Translation Field .....	6-14
Access Control Fields .....	6-14
Storage Attribute Fields .....	6-15
Storage Protection .....	6-15
TID TLB Field and Process ID (PID) Register .....	6-16
EX TLB Field .....	6-16
WR TLB Field .....	6-17
ZSEL TLB Field and Zone Protection Register (ZPR) .....	6-17
Protection Applied to Cache Management Instructions .....	6-19
Protection Applied to String Instructions .....	6-20
Page Reference and Change Recording .....	6-20
TLB Management .....	6-21
Storage Control Instructions .....	6-22
Cache Management Instructions .....	6-22
Data Cache Block Invalidate (dcbi) .....	6-22
Instruction Cache Block Touch (icbt) .....	6-23
TLB Management Instructions .....	6-25
TLB Invalidate All (tlbia) .....	6-25
TLB Read Entry (tlbre) .....	6-26
TLB Search Indexed (tlbsx) .....	6-27
TLB Synchronize (tlbsync) .....	6-28
TLB Write Entry (tlbwe) .....	6-28
<b>Interrupts and Exceptions .....</b>	<b>7-1</b>

Overview .....	7-1
Interrupt Classes .....	7-2
Synchronous/Asynchronous Interrupts .....	7-2
Precise/Imprecise Interrupts .....	7-2
Precise Interrupts .....	7-2
Imprecise Interrupts .....	7-3
Critical/Non-Critical Interrupts .....	7-4
Machine Check Interrupts .....	7-4
Interrupt Processing .....	7-4
Interrupt and Exception Types .....	7-6
Partially Executed Instructions .....	7-10
Interrupt Ordering and Masking .....	7-11
Guidelines for System Software .....	7-12
Interrupt Order .....	7-14
Exception Priorities .....	7-15
Integer Loads and Stores, Cache Management Instructions .....	7-16
Floating Point Loads and Stores .....	7-16
Floating Point (other) .....	7-16
Auxiliary Processor Loads and Stores .....	7-17
Auxiliary Processor (other) .....	7-17
Illegal Instructions .....	7-17
Privileged Instructions .....	7-18
Trap Instructions .....	7-18
System Call Instruction .....	7-18
Branch Instructions .....	7-18
Exception Handling Registers .....	7-19
Machine State Register (MSR) .....	7-19
Save Restore Registers .....	7-21
Save/Restore Registers 0 and 1 (SRR0 - SRR1) .....	7-21
Save/Restore Registers 2 and 3 (SRR2 - SRR3) .....	7-23
Data Exception Address Register (DEAR) .....	7-24
Exception Syndrome Register (ESR) .....	7-24
Exception Vector Prefix Register (EVPR) .....	7-26
Interrupt Definitions .....	7-27
Critical Input Interrupt .....	7-27
Machine Check Interrupt .....	7-28
Data Storage Interrupt .....	7-28
Instruction Storage Interrupt .....	7-30
External Input Interrupt .....	7-31
Alignment Interrupt .....	7-32
Program Interrupt .....	7-33
Illegal Instruction Exception .....	7-34
Privileged Instruction Exception .....	7-35
Trap Exception .....	7-35
Floating-Point Enabled Exception .....	7-35



Floating Point Enabled but Unimplemented Exception .....	7-36
Auxiliary Processor Unavailable Exception .....	7-36
Auxiliary Processor Enabled Exception .....	7-36
Floating-Point Unavailable Interrupt .....	7-37
System Call Interrupt .....	7-38
Programmable Interval Timer Interrupt .....	7-38
Fixed Interval Timer Interrupt .....	7-39
Watchdog Timer Interrupt .....	7-40
Data TLB Miss Interrupt .....	7-41
Instruction TLB Miss Interrupt .....	7-42
Debug Interrupt .....	7-42
Interrupt Control Instructions .....	7-44
Move From Machine State Register (mfmsr) .....	7-44
Move to Machine State Register (mtmsr) .....	7-45
Return From Critical Interrupt (rfci) .....	7-45
Return From Interrupt (rfi) .....	7-46
System Call (sc) .....	7-47
Write External Enable (wrtee) .....	7-48
Write External Enable Immediate (wrteei) .....	7-48
<b>Reset and Initialization .....</b>	<b>8-1</b>
Reset and Initialization .....	8-1
Reset Mechanisms .....	8-1
Debug Control Register (DBCR) .....	8-2
Timer Control Register (TCR) .....	8-2
Processor State After Reset .....	8-2
Software Initialization Requirements .....	8-4
Initialization Code Example .....	8-4
<b>Debug Facilities .....</b>	<b>9-1</b>
Background .....	9-1
Internal Debug Mode .....	9-1
Debug Events .....	9-2
Instruction Address Compare (IAC) .....	9-3
Data Address Compare (DAC) .....	9-3
Trap (TR) .....	9-5
Branch Taken (BT) .....	9-5
Instruction Complete (ICMP) .....	9-6
Exception (EXC) .....	9-7
Unconditional Debug Event (UDE) .....	9-8
Debug Registers .....	9-9
Debug Control Register (DBCR) .....	9-9
Debug Status Register (DBSR) .....	9-11
Data Address Compare Register (DACR) .....	9-12
Instruction Address Compare Registers (IACR) .....	9-13

<b>Timer Facilities .....</b>	<b>10-1</b>
Background and Information .....	10-1
Time Base .....	10-3
Writing the Time Base .....	10-4
Programmable Interval Timer (PIT) .....	10-5
Fixed Interval Timer (FIT) .....	10-7
Watch Dog Timer (WDT) .....	10-8
Timer Control Register (TCR) .....	10-11
Timer Status Register (TSR) .....	10-12
Freezing the Timer Facilities .....	10-13
<b>Synchronization Requirements .....</b>	<b>11-1</b>
Context Synchronization .....	11-1
Execution Synchronization .....	11-2
Synchronization Requirements .....	11-2
<b>Instruction Set Summary.....</b>	<b>12-1</b>
Summary of Instructions .....	12-1
<b>Byte Ordering .....</b>	<b>A-1</b>
Structure Mapping Examples .....	A-2
Big-Endian Mapping .....	A-2
Little Endian Mapping .....	A-3
PowerPC Byte Ordering .....	A-3
PowerPC Endian Mode .....	A-3
Byte Ordering in PowerPC Little Endian Mode .....	A-4
Control of PowerPC Endian Mode .....	A-7
Addressing in PowerPC Little Endian Mode .....	A-7
Little Endian Mode Alignment Requirements .....	A-8
Switching Endian Modes .....	A-9
Direct Memory Access in PowerPC Little Endian Mode .....	A-9
Endian Storage Attribute .....	A-9
Fetching Instructions from Little Endian Storage Regions .....	A-10
Accessing Data in Little Endian Storage Regions .....	A-11
Control of the Endian Storage Attribute .....	A-12
PowerPC Byte-Reverse Instructions .....	A-12
<b>Index.....</b>	<b>X-1</b>

# Figures

Figure 3-1.	Time Base (TB) .....	3-1
Figure 3-2.	Time Base Lower (TBL) .....	3-2
Figure 3-3.	Time Base Upper (TBU) .....	3-2
Figure 5-1.	Special Purpose Registers .....	5-1
Figure 5-2.	Special Purpose Register General (SPRG0-SPRG3) .....	5-4
Figure 5-3.	Processor Version Register (PVR) .....	5-5
Figure 6-1.	Storage Attribute Control Registers .....	6-9
Figure 6-2.	Effective to Real Address Translation Flow .....	6-11
Figure 6-3.	TLB Entries .....	6-12
Figure 6-4.	Process ID (PID) .....	6-16
Figure 6-5.	Zone Protection Register (ZPR) .....	6-18
Figure 7-1.	Machine State Register (MSR) .....	7-19
Figure 7-2.	Save/Restore Register 0 (SRR0) .....	7-22
Figure 7-3.	Save/Restore Register 1 (SRR1) .....	7-22
Figure 7-4.	Save/Restore Register 2 (SRR2) .....	7-23
Figure 7-5.	Save/Restore Register 3 (SRR3) .....	7-23
Figure 7-6.	Data Exception Address Register (DEAR) .....	7-24
Figure 7-7.	Exception Syndrome Register (ESR) .....	7-25
Figure 7-8.	Exception Vector Prefix Register (EVPR) .....	7-26
Figure 9-1.	Debug Control Register (DBCR) .....	9-9
Figure 9-2.	Debug Status Register (DBSR) .....	9-11
Figure 9-3.	Data Address Compare Register (DACR) .....	9-13
Figure 9-4.	Instruction Address Compare Registers (!ACR) .....	9-13
Figure 10-1.	Relationship of Timer Facilities to Time Base .....	10-2
Figure 10-2.	Time Base (TB) .....	10-3
Figure 10-3.	Programmable Interval Timer (PIT) .....	10-6
Figure 10-4.	Watchdog State Machine .....	10-9
Figure 10-5.	Timer Control Register (TCR) .....	10-11
Figure 10-6.	Timer Status Register (TSR) .....	10-12
Figure A-1.	Normal Word Load or Store (Big Endian Storage Region) .....	A-13
Figure A-2.	Byte-reverse Word Load or Store (Little Endian Storage Region) .....	A-13
Figure A-3.	Byte-reverse Word Load or Store (Big Endian Storage Region) .....	A-14
Figure A-4.	Normal Word Load or Store (Little Endian Storage Region) .....	A-14



# Tables

Table 3-1.	TBRN and TBRF Values .....	3-4
Table 3-2.	Extended Mnemonics for mftb .....	3-5
Table 4-1.	IBM PowerPC Embedded Virtual Environment Instruction Set .....	4-1
Table 6-1.	TLB Fields Related to Page Size .....	6-13
Table 6-2.	Protection Applied to Cache Instructions .....	6-20
Table 7-1.	Interrupt and Exception Types .....	7-6
Table 7-2.	Register Settings during Critical Input Interrupts .....	7-27
Table 7-3.	Register Settings during Machine Check Interrupts .....	7-28
Table 7-4.	Register Settings during Data Storage Interrupts .....	7-30
Table 7-5.	Register Settings during Instruction Storage Interrupts .....	7-31
Table 7-6.	Register Settings during External Input Interrupts .....	7-32
Table 7-7.	Register Settings during Alignment Interrupts .....	7-33
Table 7-8.	Register Settings during Program Interrupts .....	7-34
Table 7-9.	Register Settings during Floating-Point Unavailable Interrupts .....	7-37
Table 7-10.	Register Settings during System Call Interrupts .....	7-38
Table 7-11.	Register Settings during Programmable Interval Timer Interrupts .....	7-39
Table 7-12.	Register Settings during Fixed Interval Timer Interrupts .....	7-40
Table 7-13.	Register Settings during Watchdog Interrupts .....	7-41
Table 7-14.	Register Settings during Data TLB Miss Interrupts .....	7-41
Table 7-15.	Register Settings during Instruction TLB Miss Interrupts .....	7-42
Table 7-16.	Register Settings during Debug Interrupt .....	7-43
Table 8-1.	Contents of Processor Resources After Reset .....	8-2
Table 10-1.	Fixed Interval Timer .....	10-7
Table 10-2.	Watch Dog Timer .....	10-8
Table 10-3.	Watchdog Timer Controls .....	10-10
Table 11-1.	Data Access .....	11-4
Table 11-2.	Instruction Fetch And/Or Execution .....	11-5
Table 12-1.	IBM PowerPC Embedded Operating Environment Instruction Set .....	12-1



# About This Book

---

The IBM PowerPC Embedded Environment provides architectural specifications for IBM's PowerPC 400 series of embedded controllers. These specifications differ somewhat from, and are an alternative to, certain features defined by the PowerPC Architecture in the document, "The PowerPC Architecture: A Specification for a New Family of RISC Processors." While many of the architectural attributes defined here are consistent with the corresponding attributes in the PowerPC Architecture, certain others are specifically optimized for embedded applications and are unique to IBM's PowerPC 400 series of embedded controllers. This book complies with PowerPC Architecture Version 1.06.

The IBM PowerPC Embedded Environment features:

- Memory management unit optimized for embedded software environment
- Cache management instructions to optimize performance and memory in complex applications which are graphically and numerically intensive.
- Storage attributes for controlling storage subsystem behavior.
- Special purpose registers that control the use of the debug facilities, timer facilities, interrupts, real-mode storage attributes, memory management unit and other architected processor resources.
- Device control register address space for the control of on-chip peripherals, such as memory controllers.
- Dual level interrupt structure and interrupt control instructions.
- Multiple timer facilities.
- Debug facilities that enable hardware and software debug functions such as instruction and data breakpoints and program single stepping.

## Who Should Use This Book

This book is for hardware, software, and application developers who need to understand the IBM PowerPC Embedded Environment. The audience should understand embedded system design, operating systems, and the principles of computer organization.

## Related Publications

- *The PowerPC Architecture: A Specification for a new family of RISC processors*

## How This Book is Organized

This book is organized in three parts:

### Part One - Background and Introduction

Part One provides a brief history and overview of the IBM PowerPC Embedded Environment, defines various architectural levels, explains compliance with the PowerPC User Instruction Set Architecture, and briefly discusses architectural features of the embedded virtual and operating environments.

### Part Two - The IBM PowerPC Embedded Virtual Environment

Part Two provides a detailed description of user-level instructions, registers and other facilities that are intended to be accessed by system software via library routines, but that are available to the application programmer. The chapters that comprise Part Two are:

- Storage Model
- Time Base
- Instruction Set Summary

### Part Three - The IBM PowerPC Embedded Operating Environment

Part Three provides a detailed description of instructions, registers and other facilities whose access is privileged, and which are available to the operating system programmer. The chapters that comprise Part Three are:

- Integer Unit
- Storage Control
- Interrupts and Exceptions
- Reset and Initialization
- Debug Facilities
- Timer Facilities
- Synchronization Requirements
- Instruction Set Summary



## Document Conventions

The following is a list of notational conventions frequently used in this book.

$\overline{\text{Active\_Low}}$	An overbar indicates an active-low signal.
0x1f	Hexadecimal numbers
0b1001	Binary numbers
CR <sub>FLD</sub>	The field in the condition register pointed to by a field of an instruction.
$24_s$	The sign bit is replicated (sign-extended) 24 times.
xx	Bit positions which are don't-cares.
$\leftarrow$	Assignment
$\wedge$	AND logical operator
$\neg$	NOT logical operator
$\vee$	OR logical operator
$\oplus$	Exclusive-OR (XOR) logical operator
+	Twos complement addition
-	Twos complement subtraction, unary minus
$\times$	Multiplication
$\div$	Division yielding a quotient
%	Remainder of an integer division; $(33 \% 32) = 1$ .
	Concatenation
=, $\neq$	Equal, not equal relations
<, >	Signed comparison relations
$\leq$ , $\geq$	Unsigned comparison relations
if...then...else...	Conditional execution; if <i>condition</i> then <i>a</i> else <i>b</i> , where <i>a</i> and <i>b</i> represent one or more pseudocode statements. Indenting indicates the ranges of <i>a</i> and <i>b</i> . If <i>b</i> is null, the else does not appear.
do	Do loop. "to" and "by" clauses specify incrementing an iteration variable; "while" and "until" clauses specify terminating conditions. Indenting indicates the range of the loop.
leave	Leave innermost do loop or do loop specified in a leave statement.
n	A decimal number
x'n'	A hexadecimal number

b'n'	A binary number
FLD	An instruction field
FLD <sub>b</sub>	A bit in an instruction field
FLD <sub>b:b</sub>	A range of bits in an instruction field
FLD <sub>b,b,...</sub>	A list of bits, by number or name, in a named field
REG <sub>b</sub>	A bit in a named register
REG <sub>b:b</sub>	A range of bits in a named register
REG <sub>b,b,...</sub>	A list of bits, by number or name, in a named register
REG[FLD]	A field in a named register
REG[FLD, FLD ...]	A list of fields in a named register
GPR(r)	General Purpose Register r, where $0 \leq r \leq 31$ .
(GPR(r))	The contents of General Purpose Register r, where $0 \leq r \leq 31$ .
DCR(DCRN)	A DCR specified by the DCRF field in a <b>mfdcr</b> or <b>mtdcr</b> instruction
SPR(SPRN)	An SPR specified by the SPRF field in a <b>mfspr</b> or <b>mtspr</b> instruction
RA, RB, ...	GPRs
(Rx)	The contents of a GPR, where x is A, B, S, or T
(RA 0)	The contents of the register RA or 0, if the RA field is 0.
C <sub>0:3</sub>	A four-bit object used to store condition results in compare instructions.
<sup>n</sup> b	The bit or bit value <i>b</i> is replicated <i>n</i> times.
xx	Bit positions which are don't-cares.
CEIL(x)	Least integer $\geq x$ .
EXTS(x)	The result of extending x on the left with sign bits.
PC	Program counter.
RESERVE	Reservation bit; indicates whether a process has reserved a block of storage.
CIA	Current instruction address; the 32-bit address of the instruction being described by a sequence of pseudocode. This address may be used to set the next instruction address (NIA). Does not correspond to any architected register.
NIA	Next instruction address; the 32-bit address of the next instruction to be executed. In pseudocode, a successful branch is indicated by

	assigning a value to NIA. For instructions that do not branch, the NIA is CIA +4.
MS(addr, n)	The number of bytes represented by <i>n</i> at the location in main storage represented by addr.
EA	Effective address; the 32-bit address, derived by applying indexing or indirect addressing rules to the specified operand, that specifies a location in main storage.
ROTL((RS),n)	Rotate left; the contents of RS are shifted left the number of bits specified by <i>n</i> .
MASK(MB,ME)	Mask having 1's in positions MB through ME (wrapping if MB > ME) and 0's elsewhere.
instruction(EA)	An instruction operating on a data or instruction cache block associated with an effective address.

## Instruction Formats

Instructions are four bytes long. Bits of instructions are numbered 0:31, with 0 being the most significant bit (MSB). Instruction addresses are always word-aligned.

Instruction bits 0 through 5 always contain the primary opcode. Many instructions have an extended opcode in another field. The remaining instruction bits contain additional fields. All instruction fields belong to one of the following categories:

- Defined

These instruction fields contain values, such as opcodes, that cannot be altered. The instruction format diagrams specify the values of defined fields.

- Variable

These fields contain operands, such as general purpose register selectors and immediate values, that may vary from execution to execution. The instruction format diagrams specify the operands in variable fields.

- Reserved

Bits in a reserved field should be set to 0. In the instruction format diagrams, reserved fields are shaded.

If any bit in a defined field does not contain the expected value, the instruction is illegal and an illegal instruction exception occurs. If any bit in a reserved field does not contain 0, the instruction form is invalid and its result is architecturally undefined.

## Instruction Fields

AA (30)	Absolute address bit.  0 The immediate field represents an address relative to the current instruction address (CIA). The effective address (EA) of the branch target is either the sum of the LI field sign-extended to 32 bits and the branch instruction address, or the sum of the BD field sign-extended to 32 bits and the branch instruction address.  1 The immediate field represents an absolute address. The EA of the branch target is either the LI field or the BD field, sign-extended to 32 bits.
BA (11:15)	Specifies a bit in the condition register (CR) used as a source of a CR-Logical instruction.
BB (16:20)	Specifies a bit in the CR used as a source of a CR-Logical instruction.
BD (16:29)	An immediate field specifying a 14-bit signed twos complement branch displacement. This field is concatenated on the right with 0b00 and sign-extended to 32 bits.
BF (6:8)	Specifies a field in the CR used as a target in a compare or <b>mcrf</b> instruction.
BFA (11:13)	Specifies a field in the CR used as a source in a <b>mcrf</b> instruction.
BI (11:15)	Specifies a bit in the CR used as a source for the condition of a conditional branch instruction.
BO (6:10)	Specifies options for conditional branch instructions.
BT (6:10)	Specifies a bit in the CR used as a target as the result of a CR-Logical instruction.
D (16:31)	Specifies a 16-bit signed two's-complement integer displacement for load/store instructions.
DCRF (11:20)	Specifies a device control register (DCR).
FXM (12:19)	Field mask used to identify CR fields to be updated by the <b>mtcrf</b> instruction.
IM (16:31)	An immediate field used to specify a 16-bit value (either signed integer or unsigned).
LI (6:29)	An immediate field specifying a 24-bit signed twos complement branch displacement; this field is concatenated on the right with b'00' and sign-extended to 32 bits.
LK (31)	Link bit.  0 Do not update the link register (LR).

	1 Update the LR with the address of the next instruction.
MB (21:25)	Mask begin.  Used in rotate-and-mask instructions to specify the beginning bit of a mask.
ME (26:30)	Mask end.  Used in rotate-and-mask instructions to specify the ending bit of a mask.
NB (16:20)	Specifies the number of bytes to move in an immediate string load or store.
OPCD (0:5)	Primary opcode. Primary opcodes, in decimal, appear in the instruction format diagrams presented with individual instructions. The OPCODE field name does not appear in instruction descriptions.
OE (21)	Enables setting the OV and SO fields in the fixed-point exception register (XER) for extended arithmetic.
RA (11:15)	A GPR used as a source or target.
RB (16:20)	A GPR used as a source.
Rc (31)	Record bit.  0 Do not set the CR. 1 Set the CR to reflect the result of an operation.
	RS (6:10) A GPR used as a source.
RT (6:10)	A GPR used as a target.
SH (16:20)	An immediate field specifying a shift amount.
SPRF (11:20)	Specifies a special purpose register (SPR).
TO (6:10)	Specifies the conditions on which to trap, as specified in the descriptions of the <b>tw</b> and <b>twi</b> instructions.
XO (21:30)	Extended opcode for instructions without an OE field. Extended opcodes, in decimal, appear in the instruction format diagrams presented with individual instructions. The XO field name does not appear in instruction descriptions.
XO (22:30)	Extended opcode for instructions with an OE field. Extended opcodes, in decimal, appear in the instruction format diagrams presented with individual instructions. The XO field name does not appear in instruction descriptions.



## **Part One: Background and Introduction**

Part One provides a brief history and overview of the IBM PowerPC Embedded Environment, defines various architectural levels, explains the book's compliance with the PowerPC User Instruction Set Architecture, and discusses embedded architectural features.





In a constantly changing market environment where computers are becoming more and more humanly interactive and applications for embedded systems keep mushrooming, greater demands are being placed for *microprocessors that deliver*. To meet this demand, IBM has defined the IBM PowerPC Embedded Environment to provide an intelligent template of specifications for controllers designed for these embedded systems.

The value of the IBM PowerPC Embedded Environment is demonstrated in the IBM's PowerPC 400 Series of embedded controllers which are the first implementations of this embedded architecture. As an alternative set of specifications to the Virtual and Operating Environment portions of the PowerPC Architecture, it provides for high performance and functional integration with specific features optimized for embedded applications. At the same time, compatibility with the PowerPC User Instruction Set Architecture is maintained. A business benefit of this is that beginning with this family of general purpose embedded controllers, custom systems based on it can preserve the coherence of application development efforts. The scalable design provides a major benefit in terms of a structured road map for future enhancements with the same underlying architecture.

IBM PowerPC Embedded Environment is an extension of the PowerPC Architecture which was designed to enable a broad assortment of implementations. It carries with it the same flexibility and expandability which enables processors to be designed to meet the needs of a variety of markets without compromising efficiency or functionality. It is this embedded architectural flexibility, expandability, and scalability that make the IBM PowerPC 400 Series appealing to the consumer electronics and embedded control market.

## 1.1 PowerPC Architecture Organization

The PowerPC Architecture is organized at three levels:

The PowerPC User Instruction Set Architecture defines the instructions, registers and other facilities and mechanisms that are visible to user-level or application programs. These include the user-level instruction set; general purpose registers; link, count, condition, and integer exception registers; floating-point status and control register; addressing modes; instruction encoding and instruction formats; and the storage and execution models.

The PowerPC Virtual Environment Architecture describes features of the architecture that permit application programs to create or modify code, to manage the coherency of data and instructions in storage, and to optimize the performance of storage accesses. It defines the cache and memory models, the timekeeping facility from a user perspective, and resources that are accessible in user mode but that are primarily accessed from within system library routines.

The PowerPC Operating Environment Architecture describes features of the architecture that permit operating systems to allocate and manage storage, to handle errors encountered by application programs, to support I/O devices, and to provide operating system services. It specifies the resources and mechanisms to which access is privileged, including the memory protection and address translation mechanisms, the exception handling model, and privileged timer facilities.

Following the style of the PowerPC Architecture, the IBM PowerPC Embedded Environment maintains compatibility with the PowerPC User Instruction Set Architecture for application programs, while at the same time simplifying, enhancing, and extending the Virtual and Operating Environments. This allows IBM's PowerPC Embedded Controller family to provide the embedded system designer with the power, function, and flexibility needed to build an optimal embedded software environment.

## 1.2 Compatibility with the User Instruction Set Architecture

The IBM PowerPC Embedded Environment helps to maximize cross-platform portability of applications developed for PowerPC processors, by guaranteeing application code compatibility across all PowerPC 32-bit implementations. Implementations that conform to the IBM PowerPC Embedded Environment also adhere to the PowerPC User Instruction Set Architecture. Implementations of the IBM PowerPC Embedded Environment are 32-bit only implementations, and thus do not include the special 64-bit extensions to the PowerPC User Instruction Set Architecture. Also, it is optional as to whether these implementations provide floating point support in hardware or via software emulation.

## **1.3 The IBM PowerPC Embedded Environment**

The features that distinguish the IBM PowerPC Embedded Environment are briefly discussed in the following sections.

### **1.3.1 The IBM PowerPC Embedded Virtual Environment**

The IBM PowerPC Embedded Virtual Environment describes features of the architecture that permit application programs to create or modify code, to manage the coherency of data and instructions in storage, and to optimize the performance of storage accesses. It defines the cache and memory models, the timekeeping facility from a user perspective, and resources that are accessible in user mode but that are primarily accessed from within system library routines. The following features are part of the IBM PowerPC Embedded Virtual Environment:

#### **Storage Model**

- Storage control instructions as defined in the PowerPC Virtual Environment Architecture, for managing instruction and data caches, and for synchronizing and ordering instruction execution.
- Storage attributes for controlling storage subsystem behavior, which include:
  - Write-Through
  - Cacheability
  - Memory Coherence (optional)
  - Guarded
  - Endian
- Operand placement requirements and effects on performance

#### **Time Base**

- Time Base function as defined in the PowerPC Virtual Environment Architecture, for user-mode read access to 64-bit Time Base.

### **1.3.2 The IBM PowerPC Embedded Operating Environment**

The IBM PowerPC Embedded Operating Environment describes features of the architecture that permit operating systems to allocate and manage storage, to handle errors encountered by application programs, to support I/O devices, and to provide operating system services. It specifies the resources and mechanisms to which access is privileged, including the memory protection and address translation mechanisms, the exception handling model, and privileged timer facilities. The following features are part of the IBM PowerPC Embedded Operating Environment:

#### **Integer Unit**

- Privileged special purpose registers and instructions
- Device control registers and instructions

#### **Storage Control**

- Privileged cache management instructions
- Storage attribute controls
- Address translation and memory protection
- Privileged TLB management instructions

#### **Interrupts and Exceptions**

- Dual level interrupt structure
- Various interrupt and exception types
- Various privileged SPR's
- Partially executed instructions
- Interrupt ordering and masking
- Exception priorities
- Exception handling registers
- Interrupt control instructions

#### **Reset and Initialization**

- Internal reset mechanisms consisting of:
  - Debug Control Register (DBCR)
  - Timer Control Register (TCR)
- Contents of processor resources after reset
- Software initialization requirements
- Initialization code example

## **Debug Facilities**

- Privileged SPR's for controlling Debug modes and events
- Seven types of Debug events
- Debug reset
- Debug control of timers
- Debug registers

## **Timer Facilities**

- 64-bit Time Base
- 32-bit Programmable Interval Timer (PIT)
- Fixed Interval Timer (FIT)
- Watchdog Timer (WDT)
- Privileged SPR's for controlling timer facilities
- Freezing the timer facilities

## **Synchronization Requirements**

- Synchronization requirements for special registers and Translation Lookaside Buffers
- Synchronization requirements for Data Access
- Synchronization Requirements for Instruction Fetch and/or Execution
- Context Synchronization
- Execution Synchronization



## **Part Two: The IBM PowerPC Embedded Virtual Environment**

The IBM PowerPC Embedded Virtual Environment describes features of the architecture that permit application programs to create or modify code, to manage the coherency of data and instructions in storage, and to optimize the performance of storage accesses. It defines the cache and memory models, the time-keeping facility from a user perspective, and resources that are accessible in user mode but that are primarily accessed from within system library routines.

The chapters that comprise Part Two are:

- Chapter Two: Storage Model
- Chapter Three: Time Base
- Chapter Four: Instruction Set Summary





# 2

## Storage Model

---

This chapter discusses in detail the following topics:

- Storage Model
- Virtual Storage
- Single Copy Atomicity
- Memory Coherence
- Storage Control Attributes
- Cache Models
- Shared Storage
- Operand Placement
- Storage Control Instructions

### 2.1 Storage Model

The PowerPC User Instruction Set Architecture defines storage as a linear array of bytes indexed from 0 to a maximum of  $2^{32} - 1$ . Each byte is identified by its index, called its address, and each byte contains a value. This information is sufficient to allow the programming of applications that require no special features of any particular system environment. The IBM PowerPC Embedded Virtual Environment expands this simple storage model to include caches, virtual storage, and storage attributes. The IBM PowerPC Embedded Virtual Environment, in conjunction with services based on the IBM PowerPC Embedded Operating Environment and provided by the operating system, permits explicit control of this expanded storage model. A simple model for sequential execution allows at most one storage access to be performed at a time and requires that all storage accesses appear to be performed in program order. In contrast to this simple model, the PowerPC Architecture specifies a relaxed model of memory consistency. This chapter describes features of the PowerPC Architecture that enable programmers to write correct programs for this memory model.

## 2.2 Virtual Storage

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

Each program can access  $2^{32}$  bytes of effective address (EA) space, subject to limitations imposed by the operating system. In a typical PowerPC system, each program's effective address space is a subset of a larger virtual address (VA) space managed by the operating system.

Each effective address is translated to a real address (that is, to an address in real storage) before being used to access storage. The hardware accomplishes this using the address translation mechanism described in the IBM PowerPC Embedded Operating Environment (see Section 6.4, "Address Translation and Storage Protection," on page 6-10). The operating system manages the real (physical) storage resources of the system by setting up the tables and other information used by the hardware address translation mechanism.

The IBM PowerPC Embedded Virtual Environment deals with effective addresses that are in virtual pages which are mapped to real pages before data in the virtual pages are accessed.

In general, main storage may not be large enough to map all the virtual pages used by the currently active applications. With support provided by hardware, the operating system can attempt to use the available real pages to map a sufficient set of virtual pages of the applications. If a sufficient set is maintained, paging activity is minimized. If not, performance degradation is likely to occur.

The operating system can support restricted access to virtual pages (including read-write, read-only, execute-only, and no access) as described in the IBM PowerPC Embedded Operating Environment (see Section 6.4, "Address Translation and Storage Protection," on page 6-10), based on system standards (e.g., program code might be execute-only) and application requests.

## 2.3 Single Copy Atomicity

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

In PowerPC the following single-register accesses are always atomic:

- byte accesses
- halfword accesses aligned on halfword boundaries
- word accesses aligned on word boundaries

No other accesses are guaranteed to be atomic. In particular, multiple-register loads and stores are not atomic, nor are floating-point doubleword accesses.

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

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

## 2.4 Memory Coherence

Coherence refers to the ordering of writes to a single location. Atomic stores to a given location are coherent if they are serialized in some order, and no processor is able to observe any subset of those stores as occurring in a conflicting order. This serialization order is an abstract sequence of values; the physical memory location need not assume each of the values written to it. For example, if a processor has a write back cache, it may update a location several times before the value is written to the physical memory. The result of a store operation is not available to every processor at the same instant, and it may be that a processor observes only some of the values that are written to a location. However, when a location is accessed atomically and coherently by all processors, then, for any processor, the sequence of values it loads from the location during any interval of time forms a subsequence of the sequence of values that the location logically held during that interval. That is, a processor can never load a newer value first and then, later, load an older value.

As discussed in Section 2.5, “Storage Control Attributes,” on page 2-5, the coherence of storage locations may be managed by hardware or software depending on the setting of the memory coherence attribute. Memory coherence is managed in blocks called coherence blocks. Their size is implementation-dependent (see the User’s Manual for the implementation), but is usually larger than a word and often the size of a cache block.

### Coherence Required

When a storage location is in Memory Coherence Required mode, each store to that location is serialized with all stores to that location by all other processors that also access the location coherently. (This can be implemented, for example, by an ownership protocol that allows at most one processor at a time to store to the location.) Moreover, the current copy of a cache block that is in this mode may be copied to main storage any number of times, for example, by successive **dcbst** instructions.

Coherence does not ensure that the result of a store by one processor will be immediately visible to all other processors and mechanisms in the system. Only after a program has executed the **sync** instruction are the previous storage accesses it executed guaranteed to have been performed with respect to all other processors and mechanisms.

### Coherence Not Required

When a storage location is in Memory Coherence Not Required mode, storage coherence need not be enforced. This coherence mode may be selected by software to improve performance when it is known that the particular area of storage the processor is accessing will not be accessed by another processor or mechanism. For a storage area that is in this mode, software must ensure that the data cache is consistent with main storage before changing the mode or allowing access to the storage area by another processor or mechanism. Executing a **dcbst** or **dcbf** instruction specifying a cache block that is in this mode causes the block to be copied to main storage if and only if the processor modified the contents of a location in the block and the modified contents have not been written to main storage.

## **Programming Note**

In a single-cache system, correct coherent execution does not require use of memory coherence required mode, and memory coherence not required mode may give better performance.

## **2.5 Storage Control Attributes**

Some operating systems may provide means to allow programs to specify storage control attributes not described in the IBM PowerPC Embedded Virtual Environment. The definition of these attributes can be found in the IBM PowerPC Embedded Operating Environment (see Section 6.3, “Storage Attributes,” on page 6-6). The following describes what an operating system that supports these functions is expected to provide. The details may vary between operating systems, so the details of the specific system being used must be known before these functions can be used.

Generally, the program may use one of each of the following pairs of storage attributes:

- Write Through Required or Not Required
- Caching Inhibited or Allowed
- Memory Coherence Required or Not Required (optional in hardware)
- Guarded or Not Guarded
- Big Endian or Little Endian

These attributes have meaning only when an effective address is generated in the processor performing a storage access. Moreover, not all combinations of these attributes are supported. The supported combinations are described in the IBM PowerPC Embedded Operating Environment (see Section 6.3, “Storage Attributes,” on page 6-6).

A program can specify, through an operating system service, the attributes for each virtual page to which it has access. Each access will be performed as shown in the following sections, depending on the setting of the storage control attributes for the page containing the addressed storage location.

### 2.5.1 Write Through

This attribute is meaningful only for caching allowed storage, and applies only to the data cache. It is ignored for instruction fetching. It provides the program control over whether:

1. the processor is required to update the copy of the storage location in main storage, or
2. the processor is allowed to update the copy of the storage location in the cache and to defer the update of main storage.

#### Required

Loads use the copy in the cache if it is there. Stores update the copy of the storage location in main storage.

#### Not Required

Loads and stores use the copy in the cache if it is there. The block containing the target storage location may be copied to the cache. The storage location in main storage need not contain the value most recently stored to that location.

### 2.5.2 Cacheability

This attribute controls whether data and instructions being read from or written to main storage can be read from or placed into a cache.

#### Inhibited

When caching is inhibited, the write through attribute has no meaning. The access is performed in the following manner:

1. The operation is performed to main storage. Neither the target location nor any of the block(s) containing it is copied to the cache.
2. The operation causes an access of appropriate length (that is, byte, halfword, word, etc.) to the target location in main storage.

It is considered a programming error if a copy of the target location of a load, store, or **dcbz** to Caching Inhibited storage is in the cache, or if an instruction fetched from Caching Inhibited storage is in the cache. Software must ensure that the location has not previously been brought into the cache or, if it has, that it has been flushed from the cache. If the programming error occurs, the result of the access is boundedly undefined. It is not considered a programming error if the target location of any other cache management instruction to Caching Inhibited storage is in the cache.

#### Allowed

When caching is allowed, the access is performed in the following manner:

1. If the block containing the target storage location is in the cache, it is used.

2. If the block containing the target location is not in the cache, the block(s) of storage containing the target location may be copied to the cache and, if the access is a store, the target location is updated in the data cache if it is in that cache.

Software must ensure that all locations in a page have been purged from the cache prior to changing the storage mode for the page from caching allowed to caching inhibited.

### **2.5.3 Memory Coherence**

This attribute provides the program control over whether the processor maintains storage coherence. The provision of hardware support for this attribute is optional. If the attribute is not provided, the processor operates as if the attribute specified coherence not required. In such implementations, software is required to manage the coherence of memory and caches.

#### **Required**

Stores by all processors to the same location are serialized into some order and no processor is able to observe any subset of those stores as occurring in a conflicting order. When coherence is required, its serialization function is effective for all supported combinations of the write through and caching modes.

#### **Not Required**

The order in which one processor observes the stores performed by one or more other processors is undefined. When coherence is not required, the programmer must manage the coherence of storage through use of sync and cache management instructions, and facilities provided by the operating system.

### **2.5.4 Guarded Storage**

This attribute provides the program control over the conditions under which data and instructions can be accessed out-of-order.

#### **Guarded**

Data cannot be accessed out-of-order, and instructions cannot be fetched out-of-order, except under the conditions described in the IBM PowerPC Embedded Operating Environment (see Section 6.2.4.1, “Guarded Storage,” on page 6-4).

#### **Not Guarded**

Data can be accessed out-of-order, and instructions can be fetched out-of-order.

## 2.5.5 Endianness

Objects may be loaded from or stored to memory in byte, halfword, word, or doubleword units. For a particular data length, the loading and storing operations are symmetric; a store followed by a load of the same data object will yield an unchanged value. There is no information in the process about the order in which the bytes which comprise the multi-byte data object are stored in memory.

The Endian Storage Attribute is different from, and independent of, the Endian Mode Control.

### Big Endian

If a stored multi-byte object is probed by reading its component bytes one at a time using load-byte instructions, then the storage order may be perceived. If such probing shows that the lowest memory address contains the highest-order byte of the multi-byte scalar, the next higher sequential address the next least significant byte, and so on, then the multi-byte object is stored in Big Endian form.

### Little Endian

Alternatively, if the probing shows that the lowest memory address contains the lowest-order byte of the multi-byte scalar, the next higher sequential address the next most significant byte, and so on, then the multi-byte object is stored in Little Endian form.

## 2.6 Cache Models

The PowerPC Architecture does not require any particular cache organization and allows many different implementations. However, for a program to execute correctly on all implementations, the programmer should assume that separate instruction and data caches exist, and should program to the separate cache model. The functions of these caches are affected by the storage control attributes associated with each storage access as described in Section 2.5, “Storage Control Attributes,” on page 2-5. Cache management instructions are provided so programs can manage the caches when needed. Depending on the storage control attributes specified by the program and the function being performed, the program may need to use these instructions to guarantee that the function is performed correctly. The instructions are also useful to optimize the use of memory bandwidth in such applications as graphics and numerically intensive computing.

The processor is not required to maintain copies of storage locations in the instruction cache consistent with changes to storage resulting from the execution of store instructions. Program management of the cache is required when the program generates or modifies code that will be executed (that is, when the program modifies data in storage and then attempts to execute the modified data as instructions).

The instructions provided allow the program to:

- Invalidate the copy of storage in an instruction cache block (**icbi**)



- Perform context synchronization, as described in the IBM PowerPC Embedded Operating Environment (**isync**) (see Section 11.1, “Context Synchronization,” on page 11-1).
- Give a hint that a block of storage should be copied to the data cache, so that the copy of the block may be in the cache when subsequent accesses to the block occur, thereby reducing delays (**dcbt**, **dcbtst**).
- Establish an address in the data cache and set the contents of the block to zeros (**dcbz**).
- Copy the contents of a data cache block to main storage (**dcbst**).
- Copy the contents of a data cache block to main storage and make the copy of the block in the data cache invalid (**dcbf**).

The function of the cache management instructions depends on the implementation of the caches and on the storage control attributes associated with the cache block that is the target of the cache management instruction.

There are many variations of cache implementations and the following sections do not attempt to describe them exhaustively. However, the variations that affect the function of the cache management instructions are discussed here.

### **Programming Note**

Implementations will vary as to what instructions need to be executed to perform a function such as code modification. Operating systems are encouraged to provide a service, which is implementation dependent, to perform the function in an efficient manner.

## **2.6.1 Split or Dual Caches**

A cache model in which there are separate caches for instructions and data is called a “Harvard style” cache. This style is the standard PowerPC cache model; that is, it is the model assumed by this architecture, and the function of the cache management instructions depends on this model as well as on the storage control attributes of the target storage block. A copy of a target block in the cache is said to be marked invalid if it will not be used for subsequent accesses. The following sections describe the functions performed by each of the cache management instructions in this model.

### **2.6.1.1 Instruction Cache Block Invalidate (icbi)**

This instruction permits the program to invalidate the target storage block in the instruction cache, causing any subsequent fetch request for an instruction in the block to be sent to main storage (because the block is not found in the instruction cache). The instruction performs the following operations:

1. If the target block is not accessible to the program for loads, the system data storage error handler may be invoked.
2. Memory coherence:

- Required

If the target block is in any of the instruction caches in the system, it is marked invalid in those caches.

- Not required

If the target block is in the instruction cache of the executing processor, it is marked invalid in that cache.

3. This access need not be recorded, but if it is it is considered a load and not a store.

### 2.6.1.2 Data Cache Block Touch (dcbt, dcbtst)

The two *touch* instructions (one for reading, the other for writing) permit the program to attempt to have the target storage block in the cache prior to its first use, and thereby to avoid some of the delays due to accessing storage. These instructions are performance hints, and perform the following operations:

1. If the target block is not accessible to the program for loads, no other operation is performed.

2. Caching

- Inhibited

The target block is not copied to the data cache of the executing processor and no other operations are performed.

- Allowed

- Memory Coherence Required

If the target block is not in the data cache of the executing processor, the most recent version of the block may be copied to that data cache.

- Memory Coherence Not Required

If the target block is not in the data cache of the executing processor, the block may be copied to that data cache from main storage without regard for the location of the most recently modified version.

3. This access need not be recorded, but if it is, it is considered a load and not a store.

If the instruction is *touch for store* and the block is copied to the cache, it is copied in a manner such that a subsequent store to the block will execute efficiently.

The execution of either of these instructions *never* causes the system data storage error handler to be invoked.

### 2.6.1.3 Data Cache Block Set to Zero (dcbz)

This instruction permits the program to set large areas of storage to zeros in an efficient manner. The instruction performs the following operations:

1. If the target block is not accessible to the program for stores, the system data storage error handler is invoked.

2. Write Through Required

Either each byte of the target block in main storage is set to 0x00, or the system alignment error handler is invoked.

3. Caching Inhibited

Either each byte of the target block in main storage is set to 0x00, or the system alignment error handler is invoked.

4. Memory Coherence

- Required

If the target block is in the data cache of the executing processor, each byte in the block is set to 0x00 and all copies of the block in all data caches in the system are made consistent. If the target block is not in the data cache of the executing processor, the block is established in that data cache without fetching it from main storage and each byte in the block is set to 0x00. All copies of the block in all data caches in the system are made consistent.

- Not Required

If the target block is in the data cache of the executing processor, each byte in the block is set to 0x00. If the target block is not in the data cache of the executing processor, the block is established in that data cache without fetching it from main storage and each byte in the block is set to 0x00.

5. This access must be recorded. It is considered a store.

### 2.6.1.4 Data Cache Block Store (dcbst)

This instruction permits the program to ensure that the latest version of the target storage block is in main storage. This instruction permits the program to ensure that the latest version of the target storage block is in main storage. The instruction performs the following operations:

1. If the target block is not accessible to the program for loads, the system data storage error handler may be invoked.

2. Memory Coherence

- Required

If the target block is in any of the data caches in the system and has been modified, it is copied to main storage.

- Not Required

If the target block is in the data cache of the executing processor and has been modified, it is copied to main storage.

3. This access need not be recorded, but if it is, it is considered a load and not a store.

### 2.6.1.5 Data Cache Block Flush (dcbf)

This instruction permits the program to ensure that the latest version of the target storage block is in main storage and no longer in the data cache. The instruction performs the same operations as does **dcbst**. In addition to those operations, the following is done.

1. Memory Coherence

- Required

If the target block is in any of the data caches in the system, it is marked invalid in those data caches.

- Not Required

If the target block is in the data cache of the executing processor, it is marked invalid in that data cache.

2. This access need not be recorded, but if it is, it is considered a load and not a store.

## 2.6.2 Combined Cache

A combined cache implementation provides a single cache for instructions and data. For this implementation, the **icbi** instruction need not perform the same operations as it would for an implementation with separate caches. The instruction is treated as a no-op, except that it is acceptable to invalidate the target block in the instruction caches of other processors if the block is in memory coherence required mode.

## 2.6.3 Write Through Data Cache

The cache management instructions affected by the write through implementation of the data cache are listed in this section. These instructions must perform all the operations specified for a Harvard style cache except as specified in this section. Some of the differences depend on whether the write through implementation is a write through to main storage or just a write through to a second level of cache.

### 2.6.3.1 Write Through to Main Storage

For Data Cache Block Set to Zero (**dcbz**) the processor may invoke the system alignment error handler regardless of the setting of the storage control attributes.

For Data Cache Block Store (**dcbst**) and Data Cache Block Flush (**dcbf**) the processor is not required to copy the target block to main storage. By definition, the cache cannot contain a modified block.

### 2.6.3.2 Write Through to Multilevel Cache

For Data Cache Block Set to Zero (**dcbz**), the processor may invoke the system alignment error handler regardless of the setting of the storage control attributes.

If a cache is the interface to main storage for all processors and other mechanisms that access storage, that cache can be considered main storage with respect to the cache management instructions. Otherwise, the cache instructions that cause the content of a cache block to be copied back to main storage or to be marked invalid must be performed against all levels of the cache.

## 2.7 Shared Storage

This architecture supports the sharing of storage between programs, between different instances of the same program on systems with one or more processors, and between processors and other mechanisms. It also supports access to a storage location by one or more programs using different effective addresses. All these cases are considered storage sharing. Storage is shared in blocks that are an integral number of pages.

When the same storage location has different effective addresses, the addresses are said to be aliases. Each application can be granted separate access privileges to aliased pages.

### 2.7.1 Storage Access Ordering

The PowerPC Architecture specifies a weakly consistent storage model for shared storage multiprocessor systems. This model provides an opportunity for significantly improved performance over the strongly consistent model, but places the responsibility on the program to ensure that ordering or synchronization instructions are properly placed when necessary for the correct execution of the program.

In this architecture, the order in which the processor performs storage accesses, the order in which those accesses complete in main storage, and the order in which those accesses are viewed as occurring by another processor may all be different. This property is referred to as storage access ordering. A means of enforcing an ordering of storage accesses is provided to allow programs or instances of programs to share storage. Similar means are needed to allow programs executing on a processor to share storage with some other mechanism, such as an I/O device, that can also access storage.

The purpose of specifying a weakly consistent storage model is to allow the processor to run very fast for most storage accesses. Two instructions, Enforce In-Order Execution of I/O (**eieio**) and Synchronize (**sync**), are provided to enable the program to control the order in which storage accesses are performed by separate instructions. No ordering should be

assumed for the storage accesses done by a multiple-register load or store instruction, and no means are provided for controlling that order.

### 2.7.1.1 Enforce In-Order Execution of I/O (eieio)

The **eieio** instruction permits the program to control the order in which loads and stores are performed when the accessed storage has certain attributes (see Section 2.9.4, “Enforce In-order Execution of I/O (eieio),” on page 2-29). For example, **eieio** can be used to ensure that a sequence of stores to the control registers of an I/O device update those control registers in the desired order. It can also be used to ensure that all stores to a shared data structure are visible to other processors before the store that releases the lock is visible to them. However, it cannot be used to order a caching allowed store with respect to a caching inhibited store.

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

### 2.7.1.2 Synchronize (sync)

When a portion of storage that is accessed coherently (that is, with memory coherence required) must be forced to a known state, it is necessary to synchronize storage with respect to other processors and mechanisms. This synchronization is accomplished by requiring programs to indicate explicitly in the instruction stream, by inserting a **sync** instruction, that synchronization is required. Only when **sync** completes are the effects of all coherent storage accesses previously executed by the program guaranteed to have been performed with respect to all other processors and mechanisms that access those locations coherently.

The **sync** instruction permits the program to ensure that all the coherent storage accesses it has initiated have been performed with respect to all other processors and mechanisms that access the target locations coherently, before its next instruction is executed. A program can use this instruction to ensure that all updates to a shared data structure that it accesses coherently are visible to all other processors that access the data structure coherently, before executing a store that will release a lock on that data structure. Execution of this instruction does the following:

- Performs the functions described for the **sync** instruction in the PowerPC User Instruction Set Architecture.
- Ensures that consistency operations and the effects of **dcbf**, **dcbi**, **dcbst**, **dcbz**, **icbi** and **icbt** instructions (see the IBM PowerPC Embedded Operating Environment Section 6.5, “Storage Control Instructions,” on page 6-22) previously executed by the processor executing the **sync** have completed on such other processors as the storage control attributes of the target locations require.
- Ensures that TLB invalidates previously executed by the processor executing the **sync** have completed on that processor. **sync** does not wait for such invalidates to complete on other processors (see the IBM PowerPC Embedded Operating Environment Section 6.5.2, “TLB Management Instructions,” on page 6-25).

The **sync** instruction is execution synchronizing (see Section 11.2, “Execution Synchronization,” on page 11-2). It is not context synchronizing (see Section 11.1, “Context Synchronization,” on page 11-1), and therefore need not discard prefetched instructions.

For storage that is accessed noncoherently (that is, with Memory Coherence Not Required), the **sync** instruction operates as described above except that its only effect on storage operations is to ensure that all previous storage operations have completed, with respect to the processor executing the **sync** instruction, to the level of storage specified by the caching and write through storage control attributes.

## 2.7.2 Atomic Update Primitives

The *load and reserve* and *store conditional* instructions together permit atomic update of a storage location. These instructions function in caching inhibited, as well as in caching allowed, storage. The addressed page must, however, have the memory coherence required attribute for every processor, other than the one doing the atomic update, that might execute a store to the location being atomically updated. The remainder of this section assumes that if the system is a multiprocessor, then all processors have the addressed page in memory coherence required mode.

Furthermore, unless otherwise noted, the remainder of this section assumes that the processor executing the **lwarx/stwcx**. pair includes hardware support for memory coherence. If this is not the case, the **lwarx** and **stwcx**. will in general still operate as described, except that accesses by other processors or mechanisms to the storage location for which this processor has a reservation are not required to cancel the reservation, and thus a subsequent **stwcx**. by this processor may succeed.

If the addressed storage is in Write Through Required Mode, it is implementation-dependent whether these instructions function correctly or cause the system data storage error handler to be invoked.

The **lwarx** instruction is a load from a word-aligned location that has two side effects.

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

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

### Programming Note

To ensure that a store or **stwcx**. to a given location has been performed with respect to all other processors and mechanisms, it must be followed by a **sync**. A subsequent load or **lwarx** from the given location by another processor will

then return a value at least as recent as the value stored. This is often more synchronization than is actually needed to ensure program correctness.

### 2.7.2.1 Reservations

The ability to emulate an atomic operation using **lwarx** and **stwcx** is based on the conditional behavior of **stwcx**, the reservation set by **lwarx**, and the clearing of that reservation if the target location is modified by another processor or mechanism before the **stwcx** performs its store.

A reservation is held on an aligned unit of real storage called a reservation granule. The size of the reservation granule is implementation-dependent, but is a multiple of 4 bytes. The reservation granule associated with effective address (EA) contains the real address to which EA maps. (real\_addr(EA) in the RTL for the **lwarx** and **stwcx** instructions in the PowerPC User Instruction Set Architecture stands for real address to which EA maps.) When one processor holds a reservation and another processor performs a store, the first processor's reservation is cleared if the store affects any bytes in the reservation granule.

A processor has at most one reservation at any time. A reservation is established by executing a **lwarx** instruction, and is lost (or may be lost, in the case of the 4th bullet) if any of the following occur. Only the first 2 bullets apply to a processor that does not include the optional support for hardware-enforced memory-coherence:

1. The processor holding the reservation executes another **lwarx**; this clears the first reservation and establishes a new one.
2. The processor holding the reservation executes any **stwcx**, whether or not its address matches that of the **lwarx**.
3. Some other processor executes a store or **dcbz** to the same reservation granule.
4. Some other processor executes a **dcbf**, **dcbi** (see the IBM PowerPC Embedded Operating Environment), **dcbst** or **dcbtst**, to the same reservation granule; whether the reservation is lost is undefined.
5. Some other mechanism modifies a storage location in the same reservation granule.

Interrupts (see Section 7.3, "Interrupt Processing," on page 7-4) do not clear reservations (however, system software invoked by interrupts may clear reservations).

#### Programming Note

One use of **lwarx** and **stwcx** is to emulate a Compare and Swap primitive like that provided by the IBM System/370 compare and swap instruction: see the programming examples appendix of the PowerPC User Instruction Set Architecture. A System/370-style compare and swap checks only that the old and current values of the word being tested are equal, with the result that programs that use such a compare and swap to control a shared resource can err if the word has been modified and the old value subsequently restored.



The combination of **lwarx** and **stwcx**. improves on such a compare and swap, because the reservation reliably binds the **lwarx** and **stwcx**. together. The reservation is always lost if the word is modified by another processor or mechanism between the **lwarx** and **stwcx**., so the **stwcx**. never succeeds unless the word has not been stored into (by another processor or mechanism) since the **lwarx**. In general, programming conventions must ensure that **lwarx** and **stwcx**. addresses match; a **stwcx**. should be paired with a specific **lwarx** to the same storage location. Situations in which a **stwcx**. may erroneously be issued after some **lwarx** other than that with which it is intended to be paired must be scrupulously avoided.

For example, there must not be a context switch in which the processor holds a reservation in behalf of the old context, and the new context resumes after a **lwarx** and before the paired **stwcx**. The **stwcx**. in the new context would complete successfully, which is not what was intended by the programmer. Such a situation must be prevented by issuing a **stwcx**. to a dummy writable word-aligned location as part of the context switch, thereby clearing any reservation established by the old context. Executing **stwcx**. to a word-aligned location suffices to clear the reservation.

### 2.7.2.2 Forward Progress

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

The architecture guarantees that when a processor executes a **lwarx** to obtain a reservation for location X and then a **stwcx**. to store a value to location X, either

1. The store succeeds and the value is written to location X, or
2. The store fails because some other processor or mechanism modified location X, or
3. The store fails because the processor's reservation was lost for some other reason.

In cases 1 and 2, the system as a whole makes progress in the sense that some processor successfully modifies location X. Case 3 covers reservation loss required for correct operation of the rest of the system. This includes cancellation caused by some other processor writing elsewhere in the reservation granule for X, as well as cancellation caused by the operating system in managing certain limited resources such as real memory. It may also include implementation-dependent causes of reservation loss.

An implementation may make a forward progress guarantee, defining the conditions under which the system as a whole makes progress. Such a guarantee must specify the possible causes of reservation loss in case 3. While the architecture alone cannot provide such a guarantee, the characteristics listed in cases 1 and 2 are necessary conditions for any forward progress guarantee. An implementation and operating system can build on them to provide such a guarantee.

## Architectural Note

The architecture does not include a fairness guarantee. In competing for a reservation, two processors can indefinitely lock out a third.

### 2.7.2.3 Reservation Loss Due to Granularity

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

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

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

## 2.8 Operand Placement

The placement (location and alignment) of operands in storage affects relative performance of storage accesses, and may affect it significantly. The best performance is guaranteed if storage operands are aligned. In order to obtain the best performance across the widest range of implementations, the programmer should align all operands. Performance of accesses varies depending on the following:

1. Operand size
2. Operand alignment
3. Endian Mode (Big-Endian or Little-Endian, but not the Endian Storage Attribute)
4. Crossing no boundary
5. Crossing a cache block boundary
6. Crossing a protection boundary (see Section 6.4, “Address Translation and Storage Protection,” on page 6-10).

The *load* and *store multiple* instructions are defined to operate only on aligned, 4-byte operands. The *move assist* instructions have no alignment requirements, but will perform optimally if they begin or end on a cache block boundary. All scalar *load* and *store* instructions have no alignment requirements, but will perform optimally if aligned on an operand size boundary. Cache management instructions have no alignment requirements, and will perform optimally regardless of alignment (low order bits of address are ignored).

Finally, it is permissible for hardware to invoke the system alignment error handler for software emulation of any scalar *load* or *store* which is unaligned, or for any multiple or string *load* or *store*, regardless of alignment.

### 2.8.1 Instruction Restart

Any unaligned scalar load or store instruction, or any string or multiple instruction may be aborted after part of the access has been performed. This may occur, for example, when an external interrupt occurs after the instruction has begun execution and before it has completed. When this occurs, the implementation or the operating system may restart the instruction. If the instruction is restarted, some bytes of the location may be loaded from or stored to the target location a second time.

The following rules apply to storage accesses with regard to restarting the instruction.

#### 1. Aligned accesses

A single-register instruction that accesses an aligned operand is never restarted.

#### 2. Unaligned accesses and load and store multiple, move assist

A single-register instruction that accesses an unaligned operand or a load or store multiple or move assist instruction, may be restarted if the access crosses a protection boundary, or if an asynchronous interrupt occurs after the instruction has been partially executed, or if a Data Address Compare (DAC) Debug interrupt occurs on an address referenced by the instruction. Furthermore, these instructions may in fact never be started, and the system alignment error handler invoked (see the IBM PowerPC Embedded Operating Environment Section 7.5, “Partially Executed Instructions,” on page 7-10 for more information).

### 2.8.2 Access Atomicity

With the exception of double-precision floating-point operands, all aligned accesses are atomic. No other access is required to be atomic. Instructions causing multiple accesses, such as *load* and *store multiple* and *move assist* are not atomic.

### 2.8.3 Access Order

Since the ordering of storage accesses is not guaranteed unless the programmer inserts the appropriate ordering instructions, the order of accesses generated by a single instruction is not guaranteed. Unaligned accesses, load and store multiple instructions, and move assist instructions have no implicit ordering characteristics. For example, processor A may store a word operand on an odd halfword boundary. It may appear to processor A that the store completed atomically. Processor or other mechanism B, executing a load from the same location, may get a result that is a combination of the value of the first halfword that existed prior to the store by processor A and the value of the second halfword stored by processor A.

## 2.9 Storage Control Instructions

The instructions in this section are not privileged. For most of them, if the applicable cache is not present the operation is a no-op and has no effect on any register or on storage. The only exception is the **dcbz** instruction. When the data cache does not exist, **dcbz** either zeros a certain number of bytes of main storage (which has an effect similar to zeroing bytes in a cache block which are later written to main storage) or invokes the system alignment error handler (so that its function can be simulated).

As with other storage instructions, the effect of the cache management instructions on storage is weakly consistent. If the programmer needs to ensure that cache management or other instructions have been performed with respect to all other processors and mechanisms, a **sync** instruction must be placed in the program following those instructions.

### 2.9.1 Parameters Useful to Application Programs

It is suggested that the operating system provide a service that allows an application program to obtain the following information.

1. Page size
2. Coherence block size
3. Granule size for reservations
4. An indicator of whether the processor has a combined cache or no caches, or some other cache configuration (split caches or one cache only; if instruction cache fetches pass through the data cache, the cache is considered to be a split cache)
5. Instruction cache size
6. Data cache size
7. Instruction cache line size (see the User's Manual for the implementation)
8. Data cache line size (see the User's Manual for the implementation)
9. Block size for **icbi** (if no instruction cache, number of bytes zeroed by **dcbz**)
10. Block size for **dcbt** and **dcbtst** (if no data cache, number of bytes zeroed by **dcbz**)
11. Block size for **dcbz**, **dcbst**, **dcbf**, and **dcbi** (see Section 6.5.1.1, "Data Cache Block Invalidate (dcbi)," on page 6-22 in the IBM PowerPC Embedded Operating Environment for a description of **dcbi**) (if no data cache, number of bytes zeroed by **dcbz**)
12. Instruction cache associativity
13. Data cache associativity
14. Factors for converting the Time Base to seconds

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

## Architectural Note

All processors in a symmetric multiprocessor must be identical with respect to the cache model, the coherence block size, and the reservation granule sizes.

### 2.9.2 Instruction Cache Management Instructions

Instruction caches, if they exist, are not required to be consistent with data caches, storage, or I/O data transfers. Software must use the appropriate cache management instructions to ensure that instruction caches are kept consistent when instructions are modified by the processor or by input data transfer. When a processor alters a storage location that may be contained in an instruction cache, software must ensure that updates to storage are visible to the instruction-fetching mechanism. Although the instructions to accomplish this vary among implementations and hence many operating systems will provide a system service for this function, the following sequence is typical.

The following code example illustrates the necessary steps for self-modifying code. This example assumes that `addr1` is both data and instruction cacheable.

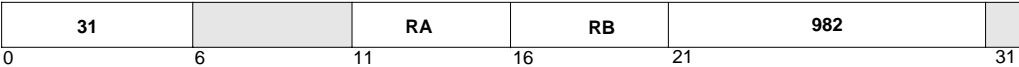
```
stw      regN, addr1    # the data in regN is to become an instruction at addr1
dcbst    addr1          # forces data from the data cache to memory
sync                                # wait until the data actually reaches the memory
icbi     addr1          # the previous value at addr1 might already be in
                        # the instruction cache; invalidate in the cache
isync                                # the previous value at addr1 might already have been
                        # pre-fetched into the queue; invalidate the queue
                        # so that the instruction must be re-fetched
```

These operations are necessary because the storage may be in Write Through Not Required mode. Since instruction fetching may bypass the data cache, changes made to items in the data cache may not be reflected in storage until after the instruction fetch completes.

### 2.9.2.1 Instruction Cache Block Invalidate (icbi)

**icbi**

RA,RB



$EA \leftarrow (RA|0) + (RB)$   
ICBI(EA)

An effective address is formed by adding an index to a base address. The index is the contents of register RB. The base address is 0 if the RA field is 0 and is the contents of register RA otherwise.

If the block containing the byte addressed by EA is in Coherence Required mode, and a block containing the byte addressed by EA is in the instruction cache of any processor, the block is made invalid in all such processors, so that subsequent references cause the block to be refetched.

If the block containing the byte addressed by EA is in Coherence Not Required mode, and a block containing the byte addressed by EA is in the instruction cache of this processor, the block is made invalid in this processor, so that subsequent references cause the block to be refetched.

The function of this instruction is independent of the Write Through Required/Not Required and Caching Inhibited/Allowed modes of the block containing the byte addressed by EA.

Implementations with a combined data and instruction cache treat the **icbi** instruction as a no-op, except that they may invalidate the target block in the instruction caches of other processors if the block is in Memory Coherence Required mode.

#### Registers Altered

- None

#### Invalid Instruction Forms

- Reserved fields

#### Exceptions

Instruction storage exceptions and instruction-side TLB Miss exceptions are associated with the fetching of instructions, not with the execution of instructions. Exceptions that occur during the execution of instruction cache ops cause data-side exceptions (Data Storage exceptions and Data TLB Miss exceptions).

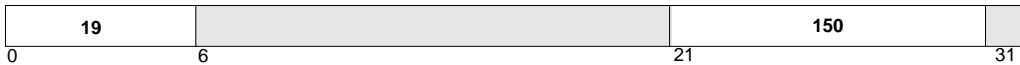
This instruction is considered a *load* with respect to address translation and storage protection, and with respect to Data Address Compare (DAC) Debug exceptions (see the IBM PowerPC Embedded Operating Environment).

### Programming Note

Instruction cache ops use MSR[DR], not MSR[IR], to determine translation of their operands (since they are treated as loads with respect to translation). When data translation is disabled, cacheability for the effective address of the operand of instruction cache ops is determined by the ICCR, not the DCCR (see the IBM PowerPC Embedded Operating Environment).

## 2.9.2.2 Instruction Synchronize (isync)

### isync



The **isync** instruction provides an ordering function for the effects of all instructions executed by a processor. Executing an **isync** instruction ensures that all instructions preceding the **isync** instruction have completed before the **isync** instruction completes, except that storage accesses caused by those instructions need not have been performed with respect to other processors and mechanisms.

The **isync** instruction also ensures that no subsequent instructions are initiated by the processor until after the **isync** instruction completes. Finally, it causes the processor to discard any prefetched instructions, with the effect that subsequent instructions will be fetched and executed in the context established by the instructions preceding the **isync** instruction. The **isync** instruction has no effect on other processors or on their caches.

### Registers Altered

- None

### Invalid Instruction Forms

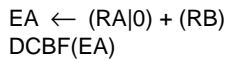
- Reserved fields

### Programming Note

This instruction is context synchronizing (see Section 11.1, “Context Synchronization,” on page 11-1).

Data caches and combined caches, if they exist, are required to be consistent with other data caches, combined caches, storage, and I/O data transfers, as specified by the Coherence Required Storage Attribute (see the IBM PowerPC Embedded Operating Environment Section 2.7, “Shared Storage,” on page 2-13).

<b>dcbf</b>	RA,RB
-------------	-------



The action taken depends on the storage mode associated with the block containing the byte addressed by EA and on the state of that block. The list below describes the action taken for the various cases.

- Unmodified Block

- Modified Block

- Absent Block

- Unmodified Block

- Modified Block



Copy the block to storage. Invalidate the block in the processor's data cache.

- Absent Block

Do nothing.

The function of this instruction is independent of the Write Through Required/Not Required and Caching Inhibited/Allowed modes of the block containing the byte addressed by EA.

### Registers Altered

- None

### Invalid Instruction Forms

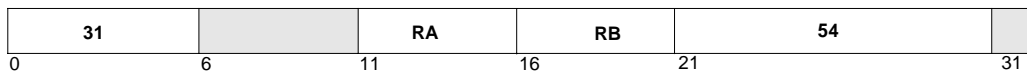
- Reserved fields

### Exceptions

This instruction is considered a load with respect to address translation and storage protection. It is considered a *store* with respect to Data Address Compare (DAC) debug exceptions. See IBM PowerPC Embedded Operating Environment Section 9.3.2, "Data Address Compare (DAC)," on page 9-3.

#### 2.9.3.2 Data Cache Block Store (dcbst)

**dcbst**                      RA,RB



$EA \leftarrow (RA[0] + (RB))$   
DCBST(EA)

An effective address is formed by adding an index to a base address. The index is the contents of register RB. The base address is 0 if the RA field is 0, and is the contents of register RA otherwise.

If the block containing the byte addressed by EA is in Coherence Required mode, and a block containing the byte addressed by EA is in the data cache of any processor and has been modified, the writing of it to main storage is initiated.

If the block containing the byte addressed by EA is in Coherence Not Required mode, and a block containing the byte addressed by EA is in the data cache of this processor and has been modified, the writing of it to main storage is initiated.

The function of this instruction is independent of the Write Through Required/Not Required and Caching Inhibited/Allowed modes of the block containing the byte addressed by EA.



### Invalid Instruction Forms

- Reserved fields

### Exceptions

This instruction is considered a *load* with respect to address translation and storage protection. It is considered a *load* with respect to Data Address Compare (DAC) Debug exceptions. However, this instruction is not allowed to cause a Data Storage interrupt or a Data TLB Miss interrupt. If execution of the instruction causes an exception which would generate such an interrupt, then no operation is performed, and no interrupt occurs. The instruction *is* permitted, however, to cause Debug interrupts (see the IBM PowerPC Embedded Operating Environment).

### Programming Note

The purpose of this instruction is to allow the program to request a cache block fetch before it is actually needed by the program. The program can later execute *load* instructions to put data into registers. However, the processor is not obliged to load the addressed block into the data cache.

### 2.9.3.4 Data Cache Block Touch for Store (dcbtst)

**dcbtst**            RA,RB



$EA \leftarrow (RA|0) + (RB)$   
DCBTST(EA)

An effective address is formed by adding an index to a base address. The index is the contents of register RB. The base address is 0 if the RA field is 0 and is the contents of register RA otherwise.

This instruction is a hint that performance will probably be improved if the block containing the byte addressed by EA is fetched into the data cache, because the program will probably soon store into the addressed byte. The hint is ignored if the block is Caching Inhibited. Executing **dcbtst** will not cause the system error handler to be invoked.

It is permissible in all circumstances for an implementation to treat this instruction as a no-op. If the instruction is not treated as a no-op, it should be handled in the following way:

- If the data block at the effective address is not in the data cache and the effective address is marked as Caching Allowed, the block is read from main storage into the data cache.
- If the data block at the effective address is in the data cache, or if the effective address is marked as Caching Inhibited, no operation is performed.

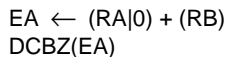
- None

- Reserved fields

This instruction is considered a *load* with respect to address translation and storage protection. It is considered a *load* with respect to Data Address Compare (DAC) Debug exceptions. This instruction is not allowed to cause a Data Storage interrupt or a Data TLB Miss interrupt. If execution of the instruction causes an exception which would generate such an interrupt, then no operation is performed, and no interrupt occurs. This instruction *is* permitted, however, to cause Debug interrupts (see the IBM PowerPC Embedded Operating Environment). Since **dcbtst** does not modify storage, it must not be recorded as a *store*.

The purpose of this instruction is to allow the program to request a cache block fetch before it is actually needed by the program. The program can later execute *store* instructions to put data into storage. However, the processor is not obliged to load the addressed block into the data cache.

**dcbz** RA,RB



An effective address is formed by adding an index to a base address. The index is the contents of register RB. The base address is 0 if the RA field is 0 and is the contents of register RA otherwise.

If the block containing the byte addressed by EA is in the data cache, all bytes of the block are set to zero.

If the block containing the byte addressed by EA is not in the data cache and the corresponding page is Caching Allowed, the block is established in the data cache without fetching the block from main storage, and all bytes of the block are set to zero. Note that nothing is read from main storage, as described in the following programming note.

If the page containing the byte addressed by EA is Caching Inhibited or Write Through Required, then either (a) all bytes of the area of main storage that corresponds to the addressed block are set to zero, or (b) the system alignment error handler is invoked.

If the block containing the byte addressed by EA is in Coherence Required mode, and the block exists in the data cache(s) of any other processor(s), it is kept coherent in those caches.

**Registers Altered**

- None

**Invalid Instruction Forms**

- Reserved fields

**Exceptions**

This instruction is considered a *store* with respect to address translation, storage protection, the ordering done by **eieio**, and with respect to Data Address Compare (DAC) Debug exceptions (see the IBM PowerPC Embedded Operating Environment).

**Programming Notes**

If the page containing the byte addressed by EA is caching inhibited or write through required, the system alignment error handler should set to zero all bytes of the area of main storage that correspond to the addressed block.

Because the **dcbz** instruction can establish an address in the data cache without copying the contents of that address from main storage, the address established may be invalid with respect to the storage subsystem. A subsequent operation may cause the address to be copied back to main storage to make room for a new cache block. A Machine Check exception could result (see the IBM PowerPC Embedded Operating Environment, Section 7.9.2, “Machine Check Interrupt,” on page 7-28 for discussion of a possible delayed Machine Check interrupt that can be caused by **dcbz** if an invalid address is used).

**2.9.4 Enforce In-order Execution of I/O (eieio)**

**eieio**



The **eieio** instruction provides an ordering function for the effects of loads and stores executed by a processor. These loads and stores are divided into two sets, which are ordered separately. The storage access caused by a **dcbz** instruction is ordered like a store.

1. Loads and stores to storage that is both caching inhibited and guarded, and stores to storage that is write through required.

**eieio** controls the order in which the accesses are performed in main storage. It ensures that all applicable storage accesses caused by instructions preceding the **eieio** instruction have completed with respect to main storage before any applicable storage accesses caused by instructions following the **eieio** instruction access main storage. It acts like a barrier that flows through the storage queues and to main storage, preventing the reordering of storage accesses across the barrier. No ordering is done for **dcbz** if the instruction causes the system alignment error handler to be invoked.

All accesses in this set are ordered as a single set, i.e., there is not one order for loads and stores to caching inhibited and guarded storage and another order for stores to write through required storage.

2. Stores to storage that has all of the following attributes: caching allowed, write through not required, and memory coherence required.

**eieio** controls the order in which the accesses are performed with respect to coherent storage. It ensures that all applicable stores caused by instructions preceding the **eieio** instruction have completed with respect to coherent storage before any applicable stores caused by instructions following the **eieio** instruction complete with respect to coherent storage.

**eieio** does not affect the order of other data accesses. With the exception of **dcbz**, **eieio** does not affect the order of cache operations (whether caused explicitly by execution of a cache management instruction or implicitly by the cache coherence mechanism). **eieio** does not affect the order of accesses in one set with respect to accesses in the other set.

The **eieio** instruction may complete before storage accesses caused by instructions preceding the **eieio** instruction have been performed with respect to main storage or coherent storage as appropriate.

## Registers Altered

- None

## Invalid Instruction Forms

- Reserved fields

### Programming Note

The **eieio** instruction is intended for use in managing shared data structures, in doing memory-mapped I/O, and in preventing load/store combining operations in main storage. For the first use, the shared data structure and the lock that protects it must be altered only by stores that are in the same set (1 or 2, see above). For the second use, **eieio** can be thought of as placing a barrier into the stream of storage accesses issued by a processor, such that any given storage access appears to be on the same side of the barrier to both the processor and the I/O device.

# 3

## Time Base

---

This chapter discusses in detail the following topics:

- Time Base
- Time Base Instructions
- Reading the Time Base
- Computing Time of Day from the Time Base

### 3.1 Time Base

The Time Base is a 64-bit register 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.

The Time Base is accessed using the 32-bit registers TBL and TBU. Software access to the time base is through the **mftb** and **mtspr** instructions.

Figure 3-1 illustrates the 64-bit Time Base (TB) register.

**Figure 3-1. Time Base (TB)**

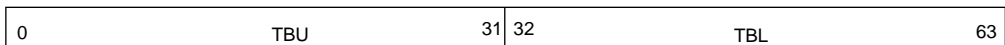
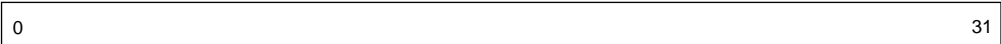


Figure 3-2 illustrates Time Base Lower(TBL) and Figure 3-3 illustrates Time Base Upper (TBU).



**Figure 3-2. Time Base Lower (TBL)**

0:31		Time Base Lower	Current count; low-order 32 bits of time base.
------	--	-----------------	--



**Figure 3-3. Time Base Upper (TBU)**

0:31		Time Base Upper	Current count, high-order 32 bits of time base.
------	--	-----------------	---

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 explicit indication (such as an interrupt).

The period of the Time Base depends on the driving frequency. As an order of magnitude example, suppose that the CPU clock is 100 MHz and that the Time Base is driven by this frequency divided by 32. Then the period of the Time Base would be

$$T_{TB} = \frac{2^{64} \times 32}{100 \text{ MHz}} = 5.90 \times 10^{12} \text{seconds}$$

which is approximately 187,000 years.



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.

### **Programming Note**

If the operating system initializes the Time Base at 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.



## Programming Notes

The mnemonic **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. Another way of saying this is that if **mftb** is coded with one operand, then that operand is assumed to be RT, and TBR defaults to the value corresponding to TBL.

The TBR number (TBRN) specified in the assembler language coding of the **mftb** instruction refers to a TBR number listed in the preceding table. The assembler handles the unusual register number encoding to generate the TBRF field.

## Architecture Note

Some implementations may implement **mftb** and **mfspr** identically. Therefore a TBR number must not match an SPR number.

### 3.2.2 Extended Mnemonics

A pair of extended mnemonics is provided for the **mftb** instruction so that it can be coded with the TBR name as part of the mnemonic rather than as a numeric operand (see Table 3-2).

**Table 3-2. Extended Mnemonics for mftb**

Extended Mnemonics	Operands	Function
mftb	RT	Move the contents of TBL into RT. Extended mnemonic for mftb RT, 0x10C
mftbu	RT	Move the contents of TBU into RT. Extended mnemonic for mftb RT, 0x10D

### 3.3 Reading the Time Base

It is not possible to read the entire 64-bit Time Base in a single instruction. The **mftb** extended mnemonic moves from the lower half of the Time Base (TBL) to a GPR, and the **mftbu** extended mnemonic moves from the upper half (TBU) to a GPR.

Because of the possibility of a carry from TBL to TBU occurring between reads of TBL and TBU, a sequence such as the following is necessary to read the Time Base.

loop:

mftbu	Rx	# load from TBU
mftb	Ry	# load from TBL
mftbu	Rz	# load from TBU
cmpw	Rz, Rx	# see if 'old' = 'new'
bne	loop	# loop if carry occurred

The comparison and loop are necessary to ensure that a consistent pair of values has been obtained.

### 3.4 Computing Time of Day from the Time Base

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

As an example, assume that the time base is incremented at a constant rate of once for every 32 cycles of a 100 MHz CPU instruction clock. What is wanted is the pair of 32-bit values comprising a POSIX standard clock: the number of whole seconds that have passed since midnight January 1, 1970, and the remaining fraction of a second expressed as a number of nanoseconds.

Assume that:

- The value 0 in the Time Base represents the start time of the POSIX clock (if this is not true, a simple 64-bit subtraction will make it so).
- The integer constant *ticks\_per\_sec* contains the value

$$\frac{100 \text{ MHz}}{32} = 3,125,000$$

which is the number of times the Time Base is updated each second.

- The integer constant *ns\_adj* contains the value

$$\frac{1,000,000,000}{3,125,000} = 320$$

which is the number of nanoseconds per tick of the time base.

Assume that:

- The operating system maintains the following variables:

*posix\_tb* (64 bits)

*posix\_sec* (32 bits)

*posix\_ns* (32 bits)

These variables hold the value of the Time Base and the computed POSIX second and nanosecond values from the last time the POSIX clock was computed.

- The operating system arranges for an interrupt to occur at least once per second, at which time it recomputes the POSIX clock values.
- The integer constant *billion* contains the value 1,000,000,000.
- The POSIX clock can be computed with an instruction sequence such as this:

loop:

mftbu	Rx	# Rx = TBU
mftb	Ry	# Ry = TBL
mftbu	Rz	# Rz = 'new' TBU value
cmpw	Rz, Rx	# see if 'old' = 'new'

```

    bne        loop          # loop if carry occurred
                                # now have 64-bit TB in Rx and Ry

    lwz        Rz, posix_tb+4

    sub        Rz, Ry, Rz     # Rz = delta in ticks

    lwz        Rw, ns, adj

    mullw      Rz, Rz, Rw     # Rz = delta in ns

    lwz        Rw, posix_ns

    add        Rz, Rz, Rw     # Rz = new ns value

    lwz        Rw, billion

    cmpw       Rz, Rw         # see if past 1 second

    blt        nochange      # branch if not

    sub        Rz, Rz, Rw     # adjust nanoseconds

    lwz        Rw, posix_sec

    addi       Rw, Rw, 1      # adjust seconds

    stw        Rw, posix_sec  # store new seconds
nochange:
    stw        Rz, posix_ns   # store new ns

    stw        Rx, posix_tb   # store new timebase

    stw        Ry, posix_tb+4

```

Note that the upper half of the Time Base does not participate in the calculation to determine the new POSIX time of day. This is correct as long as the time change does not exceed one second.

### 3.4.1 Non-constant update frequency

In a system in which the update frequency of the time base may change over time, it is not possible to convert an isolated time base value into time of day. Instead, a time base value has meaning only with respect to the current update frequency and the time of day that the update frequency was last changed. Each time the update frequency changes, either the system software is notified of the change via an interrupt, or the change was instigated by the system software itself. At each such change, the system software must compute the current time of day using the old update frequency, compute a new value of *ticks\_per\_sec* for the new frequency, and save the time of day, time base value, and tick rate. Subsequent calls to compute time of day use the current time base value and the saved data.

## Instruction Set Summary

### 4.1 Summary of Instructions

Table 4-1 provides a summary of the instructions defined by the IBM PowerPC Embedded Virtual Environment:

I

**Table 4-1. IBM PowerPC Embedded Virtual Environment Instruction Set**

Mnemonic	Operands	Instruction	Function	Other Registers Changed	Page
<b>dcbf</b>	RA, RB	Data Cache Block Flush	Flush (store, then invalidate) the data cache block which contains the effective address (RA 0) + (RB).		2-24
<b>dcbst</b>	RA, RB	Data Cache Block Store	Store the data cache block which contains the effective address (RA 0) + (RB).		2-25
<b>dcbt</b>	RA, RB	Data Cache Block Touch	Load the data cache block which contains the effective address (RA 0) + (RB).		2-26
<b>dcbtst</b>	RA,RB	Data Cache Block Touch for Store	Load the data cache block which contains the effective address (RA 0) + (RB).		2-27
<b>dcbz</b>	RA, RB	Data Cache Block set to Zero	Zero the data cache block which contains the effective address (RA 0) + (RB).		2-28
<b>eieio</b>		Enforce In-order Execution of I/O	Storage synchronization. Provides a barrier between certain kinds of load and store operations that come before the eieio, and those that come after.		2-29

**Table 4-1. IBM PowerPC Embedded Virtual Environment Instruction Set (cont.)**

<b>Mnemonic</b>	<b>Operands</b>	<b>Instruction</b>	<b>Function</b>	<b>Other Registers Changed</b>	<b>Page</b>
<b>icbi</b>	RA, RB	Instruction Cache Block Invalidate	Invalidate the instruction cache block which contains the effective address (RA 0) + (RB).		2-22
<b>isync</b>		Instruction Syn- chronize	Synchronize execution con- text by flushing the prefetch queue, and making sure that all previous instructions have completed before fetching and executing the next instruction.		2-23
<b>mftb</b>		Move From Time Base	Read the Time Base.		3-4



## **Part Three: The IBM PowerPC Embedded Operating Environment**

The IBM PowerPC Embedded Operating Environment describes features of the architecture that permit operating systems to allocate and manage storage, to handle errors encountered by application programs, to support I/O devices, and to provide operating system services. It specifies the resources and mechanisms to which access is privileged, including the memory protection and address translation mechanisms, the exception handling model, debug facilities, and privileged timer facilities.

The chapters that comprise Part Three are:

- Chapter Five: Integer Unit
- Chapter Six: Storage Control
- Chapter Seven: Interrupts and Exceptions
- Chapter Eight: Reset and Initialization
- Chapter Nine: Debug Facilities
- Chapter Ten: Timer Facilities
- Chapter Eleven: Synchronization Requirements
- Chapter Twelve: Instruction Set Summary



This chapter discusses in detail the following topics:

- Special Purpose Registers
- Device Control Registers

## 5.1 Special Purpose Registers

Special Purpose Registers (SPRs) are on-chip registers that are architecturally part of the processor core. They are accessed with the **mtspr** (move to special purpose register) and **mfspr** (move from special purpose register) instructions (see Section 5.1.4, “Special Purpose Register Instructions,” on page 5-6). The descriptions of these instructions list the valid encodings of SPR numbers. Encodings not listed are reserved for future use or for use as implementation-specific registers.

In Figure 5-1, the column “SPRN” lists register numbers, which are used in the instruction mnemonics. The column “SPRF” lists the corresponding fields, which are used in the instruction itself. These are related by  $(\text{SPRN} \leftarrow \text{SPRF}_{5:9} \parallel \text{SPRF}_{0:4})$  as shown in the **mtspr** and **mfspr** instruction descriptions.

Special purpose registers control the use of the debug facilities, the timers, the interrupts, the protection mechanism, real-mode storage attributes, the memory management unit, and other architected processor resources.

Figure 5-1 provides a summary of all special purpose registers and page references for detailed explanations. SPRG’s and the PVR are discussed later in this section. All SPR numbers not listed below are reserved, and should be neither read nor written.

**Figure 5-1. Special Purpose Registers**

Mnemonic	Register Name	SPRN		SPRF	Access	Privileged	Page
		Decimal	Hex				
CTR	Count Register <b>See Table Note 4</b>	9	0x009	0x120	Read / Write	No	-
DAC	Data Address Compare	1014	0x3F6	0x2DF	Read / Write	Yes	9-12
DBCR	Debug Control Register	1010	0x3F2	0x25F	Read / Write	Yes	9-9

**Figure 5-1. Special Purpose Registers (cont.)**

Mnemonic	Register Name	SPRN		SPRF	Access	Privileged	Page
		Decimal	Hex				
DBSR	Debug Status Register	1008	0x3F0	0x21F	Read / Clear	Yes	9-11
DCCR	Data Cache Cacheability Register	1018	0x3FA	0x35F	Read / Write	Yes	6-8
DCWR	Data Cache Write-thru Register	954	0x3BA	0x35D	Read / Write	Yes	6-8
DEAR	Data Exception Address Register	981	0x3D5	0x2BE	Read / Write	Yes	7-24
ESR	Exception Syndrome Register	980	0x3D4	0x29E	Read / Write	Yes	7-24
EVPR	Exception Vector Prefix Register	982	0x3D6	0x2DE	Read / Write	Yes	7-26
IAC	Instruction Address Compare	1012	0x3F4	0x29F	Read / Write	Yes	9-13
ICCR	Instruction Cache Cacheability Register	1019	0x3FB	0x37F	Read / Write	Yes	6-8
LR	Link Register <b>See Table Note 4</b>	8	0x008	0x100	Read / Write	No	-
PID	Process ID <b>See Table Note 1</b>	945	0x3B1	0x23D	Read / Write	Yes	6-16
PIT	Programmable Interval Timer	987	0x3DB	0x37E	Read / Write	Yes	10-5
PVR	Processor Version Register	287	0x11F	0x3E8	Read Only	Yes	5-5
SGR	Storage Guarded Register	953	0x3B9	0x33D	Read / Write	Yes	6-8
SLER	Storage Little-Endian Register	955	0x3BB	0x37D	Read / Write	Yes	6-8
SMR	Storage Memory-Coherent Register <b>See Table Note 2</b>	952	0x3B8	0x31D	Read / Write	Yes	6-8
SPRG0	SPR General 0	272	0x110	0x208	Read / Write	Yes	5-4
SPRG1	SPR General 1	273	0x111	0x228	Read / Write	Yes	5-4
SPRG2	SPR General 2	274	0x112	0x248	Read / Write	Yes	5-4
SPRG3	SPR General 3	275	0x113	0x268	Read / Write	Yes	5-4

**Figure 5-1. Special Purpose Registers (cont.)**

Mnemonic	Register Name	SPRN		SPRF	Access	Privileged	Page
		Decimal	Hex				
SRR0	Save/Restore Register 0	26	0x01A	0x340	Read / Write	Yes	7-21
SRR1	Save/Restore Register 1	27	0x01B	0x360	Read / Write	Yes	7-21
SRR2	Save/Restore Register 2	990	0x3DE	0x3DE	Read / Write	Yes	7-23
SRR3	Save/Restore Register 3	991	0x3DF	0x3FE	Read / Write	Yes	7-23
TBL	Time Base Lower <b>See Table Note 3</b>	268	0x10C	0x188	Read-only	No	3-1
TBU	Time Base Upper <b>See Table Note 3</b>	269	0x10D	0x1A8	Read-only	No	3-1
TBL	Time Base Lower	284	0x11C	0x388	Write-only	Yes	3-1
TBU	Time Base Upper	285	0x11D	0x3A8	Write-only	Yes	3-1
TCR	Timer Control Register	986	0x3DA	0x35E	Read / Write	Yes	10-11
TSR	Timer Status Register	984	0x3D8	0x31E	Read / Clear	Yes	10-12
XER	Fixed Point Exception Register <b>See Table Note 4</b>	1	0x001	0x020	Read / Write	No	-
ZPR	Zone Protection Register <b>See Table Note 1</b>	944	0x3B0	0x21D	Read / Write	Yes	6-17

### Table Notes

1. PID and ZPR are optional and are implemented only if the MMU is implemented.
2. SMR is optional and used only in implementations with support for hardware-enforced memory coherence.
3. TBL and TBH *read* SPR's are Time Base register numbers which are accessed via the **mftb** instruction. They are defined in the IBM PowerPC Embedded Virtual environment.
4. CTR, LR and XER are defined in the PowerPC User Instruction Set Architecture.

### 5.1.1 Privileged SPR Specification

The only SPRs that are not privileged are the Link Register (LR), the Count Register (CTR), and the Fixed Point Exception Register (XER). These registers are defined in the PowerPC User Instruction Set Architecture. All other SPRs are privileged, for both read and write.

The Time Base Upper (TBU) and Time Base Lower (TBL) registers are non-privileged for read access only, and are accessed via the **mftb** instruction, not the **mf spr** instruction. These registers (and the read access to them) are defined in the IBM PowerPC Embedded Virtual Environment. Write access is privileged and is provided via the **mt spr** instruction.

5.1.2 Special Purpose Register General (SPRG0-SPRG3)

SPRG0 through SPRG3 are 32-bit registers provided for operating system use. These four registers are provided as temporary storage locations. As an example, a supervisor routine may save the contents of a GPR to an SPRG, and later restore the GPR from it. This would be faster than the standard save/restore to a memory location. These registers are written using the **mt spr** instruction and read using the **mf spr** instruction. The SPRGs are privileged for both read and write. Figure 5-2 shows the SPRG bit definitions.

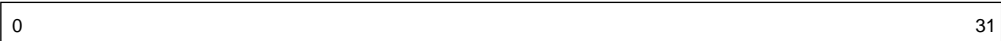


Figure 5-2. Special Purpose Register General (SPRG0-SPRG3)

0-31	General Data	Privileged user-specified; no hardware usage.
------	--------------	---

The following list describes a typical use of SPRG0 through SPRG3.

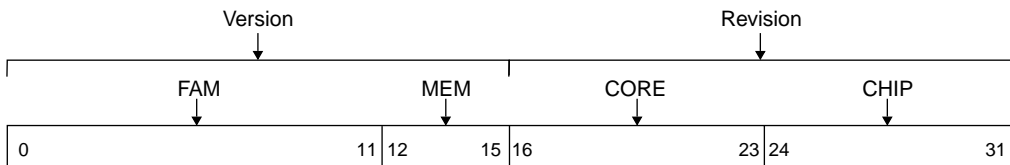
- SPRG0  
Software may load a unique real address in this register to identify an area of storage reserved for use by the first-level interrupt handler. This area must be unique for each processor in the system.
- SPRG1  
This register may be used as a scratch register by the first-level interrupt handler to save the contents of a GPR. That GPR then can be loaded from SPRG0 and used as a base register to save other GPRs to storage.
- SPRG2  
This register may be used by the operating system as needed.
- SPRG3  
This register may be used by the operating system as needed.

### 5.1.3 Processor Version Register (PVR)

The Processor Version Register (PVR) is a 32-bit read-only register that contains a value identifying the specific version (model) and revision level of the PowerPC 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. Software may be written which has features that depend on the processor type. Such software can select the proper features dynamically by examining the PVR. Figure 5-3 shows the PVR bit definitions.

The PVR contains two fields:

1. **Version:** A 16-bit number that uniquely determines a particular processor version and a version of the PowerPC Architecture. This number can be used to determine the version of a processor. The version field is broken into two parts: FAM and MEM (see Figure 5-3 below).
2. **Revision:** A 16-bit number that distinguishes between various releases of a particular version, i.e., an Engineering change level. The revision field is broken into two parts: CORE and CHIP (see Figure 5-3 below).



**Figure 5-3. Processor Version Register (PVR)**

0:11	FAM	Processor Family. Identifies a particular PowerPC processor family, such as 4xx or 6xx.	0x002 for the 4xx family.
12:15	MEM	Family Member. Identifies a specific family member, such as 403 or 601.	
16:23	CORE	Core Revision. Identifies a specific implementation of the processor core, with different values for different revisions of the implementation, whether major or minor.	Indicates the processor core revision of a particular core implementation.
24:31	CHIP	Chip Revision. Identifies a specific configuration of a chip, using the core identified by the FAM/MEM/CORE fields. A different value of this field is used for each configuration or revision of a chip, whether major or minor.	Indicates the chip revision of a particular core+ASIC chip.

The value of the Version portion of the PVR is assigned by the PowerPC Architecture process. The value of the Revision portion of the PVR is implementation-defined.

### 5.1.4 Special Purpose Register Instructions

The Move From Special Purpose Register (**mfspr**) and Move To Special Purpose Register (**mtspr**) instructions are described in the PowerPC User Instruction Set Architecture, but only at the level available to an application programmer. In particular, no mention is made there of registers that can be accessed only in privileged state. A complete description of these instructions appears below.

## Programming Note

For a discussion of software synchronization requirements when altering certain Special Purpose Registers, please refer to Section 11.3, “Synchronization Requirements,” on page 11-2.

#### 5.1.4.1 Move From Special Purpose Register (mfspr)

mf spr RT,SPRN


$$\begin{aligned} \text{SPRN} &\leftarrow \text{SPRF}_{5:9} \parallel \text{SPRF}_{0:4} \\ (\text{RT}) &\leftarrow (\text{SPR}(\text{SPRN})) \end{aligned}$$

The SPR field denotes a Special Purpose Register, encoded as shown in Figure 5-1. The contents of the designated Special Purpose Register are placed into register RT.

SPRF<sub>0</sub>=1 if and only if reading the register is privileged. Execution of this instruction specifying an SPR with SPRF<sub>0</sub>=1 when MSR[PR]=1 will result in a Privileged Instruction type Program interrupt, whether or not the SPR specified is actually defined.

If MSR[PR]=0 or SPRF<sub>0</sub>=0), and the SPR field contains any value that is not shown in Figure 5-1, then either an Illegal Instruction type Program interrupt occurs or the results are boundedly undefined.

A set of extended mnemonics is provided for the **mfspr** instructions so that they can be coded with the SPR name as part of the mnemonic rather than as a numeric operand.

## Registers Altered

- RT



Invalid Instruction Forms

- Reserved fields
- Undefined SPRF values

Compiler and Assembler Note

The SPR number coded in assembler language (SPRN) does not appear directly as a 10-bit binary number in the instruction's SPRF field. 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 this instruction has only a 5-bit SPRF field occupying bits 11:15.

5.1.4.2 Move To Special Purpose Register (mtspr)

mtspr SPRN,RS



$$\text{SPRN} \leftarrow \text{SPRF}_{5:9} \parallel \text{SPRF}_{0:4}$$
$$(\text{SPR}(\text{SPRN})) \leftarrow (\text{RS})$$

The SPR field denotes a Special Purpose Register, encoded as shown in Figure 5-1 on page 6-1. The contents of register RS are placed into the designated Special Purpose Register.

SPRF<sub>0</sub>=1 if and only if writing the register is privileged. Execution of this instruction specifying an SPR with SPRF<sub>0</sub>=1 when MSR[PR]=1 will result in a Privileged Instruction type Program interrupt, whether or not the SPR specified was actually defined.

If MSR[PR]=0 or SPRF<sub>0</sub>=0, and the SPR field contains any value that is not shown in Figure 5-1, then either an Illegal Instruction type Program interrupt occurs or the results are boundedly undefined.

A set of extended mnemonics is provided for the **mtspr** instructions so that they can be coded with the SPR name as part of the mnemonic rather than as a numeric operand.

Registers Altered

- SPR(SPRN)

Invalid Instruction Forms

- Reserved fields
- Undefined SPRF values

Compiler and Assembler Note

The SPR number coded in assembler language (SPRN) does not appear directly as a 10-bit binary number in the instruction's SPRF field. 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 this instruction has only a 5-bit SPRF field occupying bits 11:15.

5.2 Device Control Registers

Device Control Registers (DCRs) are on-chip registers that exist architecturally outside the processor core and thus are not actually part of the IBM PowerPC Embedded Environment. The IBM PowerPC Embedded Environment simply defines the existence of a DCR “address space” and the instructions to access them and does not define any particular DCRs themselves. DCR’s are accessed via the **mtdcr** (Move To Device Control Register) and **mfdcr** (Move From Device Control Register) instructions.

DCRs may control the use of on-chip peripherals, such as memory controllers (see the User’s Manual for the implementation for specific DCR definitions).

All DCRs are privileged for both read and write.

5.2.1 Device Control Register Instructions

This section describes the **mfdcr** and **mtdcr** instructions.

5.2.1.1 Move From Device Control Register (mfdcr)

**mfdcr** RT,DCRN



DCRN ← DCRF<sub>5:9</sub> || DCRF<sub>0:4</sub>  
(RT) ← (DCR(DCRN))

The contents of the DCR specified by the DCRF field are placed into register RT. See the User’s Manual for a listing of DCR names and corresponding DCRN and DCRF values.

Registers Altered

- RT

Invalid Instruction Forms

- Reserved fields
- Undefined DCRF values

### Programming Note

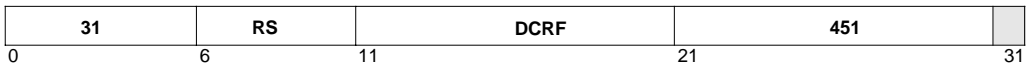
Execution of this instruction is privileged.

### Compiler and Assembler Note

The DCR number coded in assembler language (DCRN) does not appear directly as a 10-bit binary number in the instruction's DCRF field. 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 consistency with **mf spr/mt spr** encoding.

#### 5.2.1.2 Move To Device Control Register (mtdcr)

**mtdcr**                      DCRN,RS



$$\text{DCRN} \leftarrow \text{DCRF}_{5:9} \parallel \text{DCRF}_{0:4}$$
$$(\text{DCR}(\text{DCRN})) \leftarrow (\text{RS})$$

The contents of register RS are placed into the DCR specified by the DCRF field. See the User's Manual for a listing of DCR mnemonics and corresponding DCRN and DCRF values.

### Registers Altered

- DCR(DCRN)

### Invalid Instruction Forms

- Reserved fields
- Undefined DCRF values

### Programming Note

Execution of this instruction is privileged.

### Compiler and Assembler Note

The DCR number coded in assembler language (DCRN) does not appear directly as a 10-bit binary number in the instruction's DCRF field. 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 consistency with **mf spr/mt spr** encoding.



This chapter discusses in detail the following topics:

- Storage Addressing
- Storage Model
- Storage Attributes
- Address Translation and Storage Protection
- Storage Control Instructions

## 6.1 Storage Addressing

A program references storage using the effective address computed by the processor when it executes a load, store, branch, or cache instruction, and when it fetches the next sequential instruction. The effective address is translated to a real address according to procedures described in Section 6.4, “Address Translation and Storage Protection,” on page 6-10. The real address is what is sent to the memory subsystem (see Figure 6-2 on page 6-11).

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

## 6.2 Storage Model

The storage model provides the following features:

1. The architecture allows storage implementations to take advantage of the performance benefits of weak ordering of storage accesses between processors or between processors and devices.
2. The architecture provides the following instructions that allow the programmer to ensure a consistent and ordered storage state: **dcbf**, **dsbtst**, **dcbz**, **eieio**, **icbi**, **isync**, **lwarx**, **stwcx.**, **sync**, **tlbsync**.

3. Processor ordering: storage accesses by a single processor appear to complete sequentially from the view of the programming model but may complete out of order with respect to the ultimate destination in the storage hierarchy. Order is guaranteed at each level of the storage hierarchy for accesses to the same address from the same processor.
4. Storage consistency between processors and between a processor and I/O is controlled by software through mode bits in the TLB entry or real mode storage attribute registers (see Section 6.3.4, “Storage Attribute Control Registers,” on page 6-8).

### 6.2.1 Instruction Fetch

Instructions are fetched under control of MSR[IR]. When any context synchronizing event occurs, any prefetched instructions are discarded and then instructions are refetched using the then-current state of MSR[IR] and the then-current program counter.

- Real Mode

When instruction relocation is off, MSR[IR]=0, the effective address is interpreted as described in Section 6.2.5, “Real Addressing Mode,” on page 6-5.

- Virtual Mode

When instruction relocation is on, MSR[IR]=1, instructions are fetched using the address translated by the TLB mechanism.

When MSR[IR]=1 instructions are not fetched from no-execute (EX=0, See Section 6.4.4.2 on page 6-16.) or guarded storage. If the effective address of the current instruction is mapped to any of these kinds of storage, an Instruction Storage Interrupt (ISI) is generated.

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

### 6.2.2 Implicit Branch

Explicitly altering certain MSR bits (using **mtmsr**), or explicitly altering TLB entries or certain system 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 **mtmsr** instruction that changes the value of MSR[IR] may change the real address from which the current instruction stream is being fetched. The MSR bits and system registers for which alteration can cause an implicit branch are indicated as such in Section 11.3, “Synchronization Requirements,” on page 11-2. Implicit branches are not supported by the IBM PowerPC Embedded Environment. If an implicit branch occurs, the results are boundedly undefined.

### 6.2.3 Data Storage Access

Data accesses are controlled by MSR[DR]. When the state of MSR[DR] changes, subsequent accesses are made using the new state of MSR[DR].

When data translation is off, MSR[DR]=0, the effective address is interpreted as described in Section 6.2.5, “Real Addressing Mode,” on page 6-5.

When data translation is on, MSR[DR]=1, the effective address is translated by the TLB mechanism.

### 6.2.4 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 may be performed out-of-order by the hardware on the expectation that the results will be needed by an instruction that will be required by the sequential execution model. Whether the results are really needed is contingent on anything that might divert the control flow away from the instruction, such as **branch**, **trap**, **system call**, **rfi** and **rfci** instructions, and interrupts, and on anything that might change the context in which the instruction is executed.

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

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

- **Stores**

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

- **Accessing Guarded Storage**

The restrictions for this case are given in Section 6.2.4.2, “Out-of-Order Accesses to Guarded Storage,” on page 6-4.

No error of any kind, other than machine check, may be reported due to an operation that is performed out-of-order, until such time as it is known that the operation is required by the sequential execution model. The only other permitted side effect (other than Machine Check) of performing an operation out-of-order is the following:

- Non-Guarded storage locations that could be fetched into a cache by in-order execution may be fetched out-of-order into that cache.

#### 6.2.4.1 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

- A. the G bit is 1 in the relevant TLB Entry and MSR[IR, DR]=1 for instruction fetches or data accesses respectively; or
- B. the selected G bit from the SGR is 1 and MSR[IR, DR]=0 for instruction fetches or data accesses respectively.

In general, storage that is not well-behaved should be Guarded. Because such storage may represent 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 incorrect operations or may result in a Machine Check.

If an aligned, elementary load or store to caching inhibited guarded storage has accessed main storage and an asynchronous or imprecise mode Floating-Point Enabled exception is pending, the load or store is completed (that is, the SRR must point past the load or store, and the instruction cannot be restarted) before the interrupt is taken.

##### **Architectural Note**

The rules for accessing Guarded storage when an imprecise mode Floating-Point Enabled exception is pending should be revisited when the architecture is clarified with respect to those modes. For example, it may be acceptable to require software synchronization between any instruction that could cause a Floating-Point Enabled exception in imprecise mode and a subsequent instruction that accesses guarded storage. (A floating-point status and control register instruction might provide sufficient synchronization.)

#### 6.2.4.2 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 the location is in a cache then the location may be accessed in the cache or in main storage.

- Instruction Fetch

If MSR[IR]=0 then an instruction may be fetched out of order from Guarded storage if either 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 the same 1KB real page as an instruction that is required by the Sequential Execution Model, or is in the 1KB real page immediately following such a page.

(Note: If MSR[IR]=1 then instructions are not fetched from Guarded storage: see Section 6.2.1, “Instruction Fetch,” on page 6-2).

### **Programming Note**

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

If a real page contains instructions that will be executed by the Sequential Execution Model when MSR[IR]=0, software should ensure that this real page and the next real page contain only well-behaved storage.

## **6.2.5 Real Addressing Mode**

Whether address translation is enabled is controlled by MSR[IR] for instruction fetching and by MSR[DR] for data loads and stores, as well as cache management instructions, including I-cache instructions, (**icbi** and **icbt**). If address translation is disabled for a particular access (fetch, load, store or cache management instruction), the effective address is treated as the real address and is passed directly to the memory subsystem.

The effective address is a 32-bit quantity computed by the CPU. Accesses in real mode bypass all storage protection checks, and are controlled by the real mode storage attribute control registers (SGR, DCWR, DCCR, ICCR, SMR, SLER).

## **6.2.6 Invalid Real Address**

An attempt to fetch from, load from, or store to a real address that is not physically present in the machine may result in a machine check interrupt (see Section 7.9.2, “Machine Check Interrupt,” on page 7-28). This can occur by use of an effective address for a storage access when the corresponding relocate bit in the MSR[IR, DR]=0 or, when the relocate bit in the MSR[IR, DR]=1, by having the relocation mechanism set up in a way that causes nonexistent storage to be addressed.

## 6.3 Storage Attributes

When address translation is enabled and the effective address generated by a storage access is translated by the TLB mechanism, the access is performed under the control of the TLB Entry used to translate the effective address. Each TLB entry contains five storage attribute bits, W, I, M, G and E, that specify the storage attributes for all accesses translated by the entry (see Section 6.4.3, “TLB Fields,” on page 6-13). When address translation is disabled, the WIMGE bits are provided by the real mode storage attribute control registers (see Section 6.3.4, “Storage Attribute Control Registers,” on page 6-8).

The W and I bits control how the processor executing the access uses its own caches. The M bit specifies whether the processor executing the access must use the storage coherence protocol to ensure that all copies of the addressed storage location are made consistent. The G bit controls whether out-of-order fetching of data and instructions is permitted. The E bit controls whether storage accesses are performed in a big endian or little endian fashion.

### 6.3.1 W, I, M, G and E bits

The W, I, M, G and E bits control the way in which the processor accesses cache and main storage. Each bit controls a separate aspect of storage references.

- Write Through (W)

On store operations, including **dcbz**, if W=1 the update must be written to main storage.

Store combining optimizations are allowed except when the store instructions are separated by **sync** or **eieio**. The architecture presumes that data present in the cache are valid and a store may cause any part of that data to be copied back to main storage.

- Caching Inhibited (I)

If I=1, the storage access is completed by referencing the location in main storage. During the access, the accessed location is not brought into the cache nor is the location allocated in the cache.

Load/store combining optimizations are allowed except when the accesses are separated by **sync**, or by **eieio** when the storage access is also guarded.

- Memory Coherence (M)

This Storage attribute is provided to allow improved performance in systems in which accesses to storage kept consistent by hardware are slower than accesses to storage not kept consistent by hardware, and in which software is able to enforce the required consistency. When the Storage attribute is off (M=0), the hardware need not enforce data coherence for storage accesses initiated by the processor. When the Storage attribute is on (M=1), the hardware must enforce data coherence for storage accesses initiated by the processor.

Hardware support for Memory Coherence Required (M=1) storage mode is optional. If the implementation does not support this mode, then all accesses will behave as if M=0, and setting M=1 in a TLB entry will have no effect. Also, such implementations will not provide the real mode storage attribute control for the M bit (SMR), and the results of attempting to access this register will be boundedly undefined.

When an access is performed for which data coherence is required, the processor performing the access must inform the coherence mechanism that the access requires memory coherence. Other processors affected by the access must respond to the coherence mechanism. However since the mode control bits have no direct relation to data or instructions in the cache, processors responding to the coherence request are able to respond without knowledge of the state of this bit. Because instruction storage need not be consistent with data storage, it is permissible for an implementation to ignore the M bit for instruction fetches.

- Guarded (G)

If G=1 the storage is guarded, and accesses to it conform to the restrictions described in Section 6.2.4, “Performing Operations Out-of-Order,” on page 6-3.

- Endian (E)

If E=0, all accesses are performed in a big endian fashion, which means that, for all multi-byte scalar accesses, the byte at the lowest numbered address is treated as most significant. If E=1, all accesses are performed in a little endian fashion, which means that, for all multi-byte scalar accesses, the byte at the lowest numbered address is treated as least significant.

## 6.3.2 Supported Storage Modes

Support for M=1 storage is optional. Storage modes where both W=1 and I=1 (which would represent Write-Thru Required but Caching Inhibited storage) are not supported. For all supported combinations of the W,I and M bits, both G and E may be 0 or 1.

### 6.3.3 Mismatched WIMGE Bits

Accesses to the same storage location using two effective addresses for which the write through Storage attribute (W bit) differs meet the memory coherence requirements described in the IBM PowerPC Embedded Virtual Environment in Chapter 2, “Storage Model”, if the accesses are performed by a single processor. If the accesses are performed by two or more processors, coherence is enforced by the hardware only if the Write Through Storage attribute is the same for all the accesses.

Loads, stores, **dcbz** instructions, and instruction fetches to the same storage location using two effective addresses for which the caching mode (I bit) differs must meet the requirement that a copy of the target location of an access to Caching Inhibited storage not be in the cache. Violation of this requirement is considered a programming error; software must ensure that the location has not previously been brought into the cache or, if it has, that it

has been flushed from the cache. If the programming error occurs, the result of the access is boundedly undefined. It is not considered a programming error if the target location of any other cache management instruction to Caching Inhibited storage is in the cache.

Accesses to the same storage location using two effective addresses for which the guarded Storage attribute mode (G bit) or the endian Storage attribute (E bit) differs are always permitted.

Accesses to the same storage location using two effective addresses for which the memory coherence Storage attribute (M bit) differs may require explicit software synchronization before accessing the location with M=1 if the location has previously been accessed with M=0. Any such requirement is system-dependent. For example, in some "snooping bus" based systems no software synchronization may be required. In some "directory based" systems, software may be required to execute **dcbf** instructions on each processor to flush all storage locations accessed with M=0 before accessing those locations with M=1.

### 6.3.4 Storage Attribute Control Registers

Six SPRs, called storage attribute control registers, control the various storage attributes when address translation is disabled. When address translation is enabled, these registers are ignored, and the storage attributes supplied by the TLB entry are used (see Section 6.4.3, "TLB Fields," on page 6-13).

The storage attribute control registers divide the 4GB real address space into thirty-two 128MB regions. In a storage attribute control register, bit 0 controls the lowest 128MB region, bit 1 the next 128MB region, and so on.

For detailed information on the function of each of these attributes, see Section 6.3.1, "W, I, M, G and E bits," on page 6-6.

The following six 32-bit registers are defined to control storage attributes in real mode:

1. Data Cache Write-thru Register (DCWR) controls the W storage attribute.
2. Data Cache Cacheability Register (DCCR) controls the I storage attribute for data accesses and cache management instructions. Note that the polarity of the bits in this register is opposite to that of the I attribute (DCCR bit =1 means Caching Allowed while I=1 means Caching Inhibited).
3. Instruction Cache Cacheability Register (ICCR) controls the I storage attribute for instruction fetches. Note that the polarity of the bits in this register is opposite to that of the I attribute (ICCR bit =1 means Caching Allowed while I=1 means Caching Inhibited).
4. Storage Memory-coherent Register (SMR) controls the M storage control attribute for instruction and data accesses.
5. Storage Guarded Register (SGR) controls the G storage attribute for instruction and data accesses.
6. Storage Little Endian Register (SLER) controls the E storage attribute for instruction and data accesses.

Figure 6-1 shows a generic storage attribute control register. The actual storage attribute control registers have the same bit numbers and address ranges. Effective address bits 0:4 are used to select one bit from each of the storage attribute control registers. For the DCWR, SMR, SGR, and SLER registers, the selected bits are directly used as the W, M, G, and E storage attributes, respectively. For the DCCR and ICCR, the inverse of the selected bits are used as the I storage attribute for data and instruction accesses, respectively.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
---	---	---	---	---	---	---	---	---	---	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

**Figure 6-1. Storage Attribute Control Registers**

Bit	Address Range	Bit	Address Range
0	0x0000 0000–0x07FF FFFF	16	0x8000 0000–0x87FF FFFF
1	0x0800 0000–0x0FFF FFFF	17	0x8800 0000–0x8FFF FFFF
2	0x1000 0000–0x17FF FFFF	18	0x9000 0000–0x97FF FFFF
3	0x1800 0000–0x1FFF FFFF	19	0x9800 0000–0x9FFF FFFF
4	0x2000 0000–0x27FF FFFF	20	0xA000 0000–0xA7FF FFFF
5	0x2800 0000–0x2FFF FFFF	21	0xA800 0000–0xAFFF FFFF
6	0x3000 0000–0x37FF FFFF	22	0xB000 0000–0xB7FF FFFF
7	0x3800 0000–0x3FFF FFFF	23	0xB800 0000–0xBFFF FFFF
8	0x4000 0000–0x47FF FFFF	24	0xC000 0000–0xC7FF FFFF
9	0x4800 0000–0x4FFF FFFF	25	0xC800 0000–0xCFFF FFFF
10	0x5000 0000–0x57FF FFFF	26	0xD000 0000–0xD7FF FFFF
11	0x5800 0000–0x5FFF FFFF	27	0xD800 0000–0xDFFF FFFF
12	0x6000 0000–0x67FF FFFF	28	0xE000 0000–0xE7FF FFFF
13	0x6800 0000–0x6FFF FFFF	29	0xE800 0000–0xEFFF FFFF
14	0x7000 0000–0x77FF FFFF	30	0xF000 0000–0xF7FF FFFF
15	0x7800 0000–0x7FFF FFFF	31	0xF800 0000–0xFFFF FFFF

## 6.4 Address Translation and Storage Protection

The instruction unit generates all instruction effective addresses; these addresses are both for sequential instruction fetches and addresses that correspond to a change in program flow (branches, interrupts). The integer unit generates effective addresses for data accesses. Both instruction and data effective addresses are translated by the MMU, yielding real addresses which are sent to the storage subsystem.

The MMU supports demand paged virtual memory as well as a variety of other management schemes that depend on precise control of effective to real address mapping and flexible memory protection. Translation misses and protection faults cause precise exceptions. Sufficient information is available to correct the fault and restart the faulting instruction.

The MMU divides the effective address space into pages. The page represents the granularity of effective address translation and protection control. Eight page sizes (1K, 4K, 16K, 64K, 256K, 1M, 4M, 16M) are simultaneously supported. In order for an effective to real translation to exist, a valid entry for the page containing the effective address must be in the Translation Lookaside Buffer (TLB). Addresses for which no TLB entry exists cause TLB-Miss exceptions.

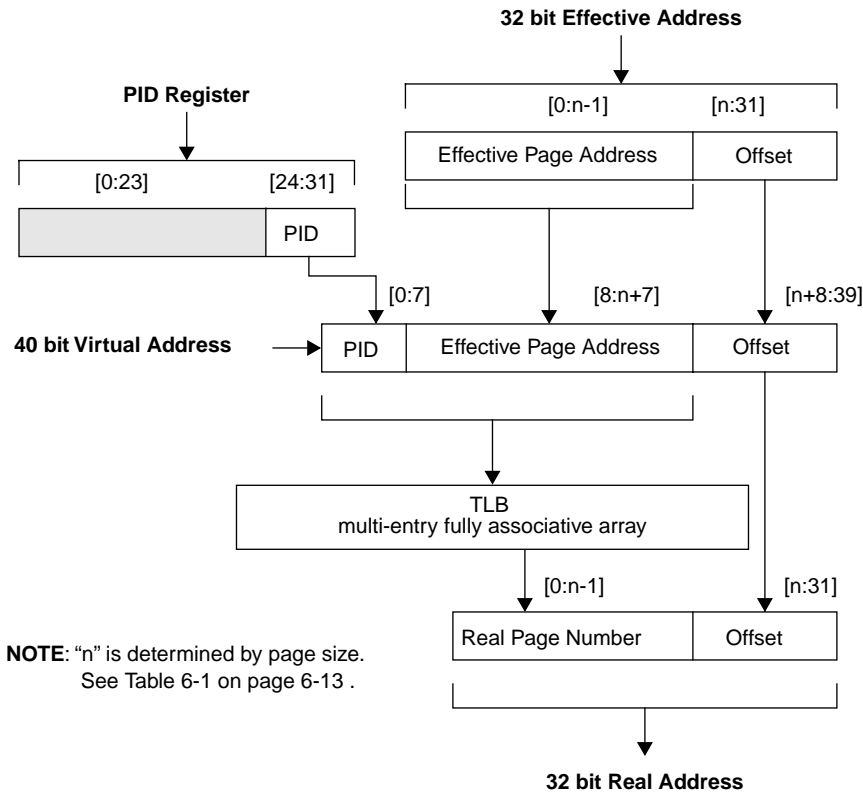
The IBM PowerPC Embedded Environment specifies the TLB entry format and defines that the TLB is managed by software. The exact organization of the TLB (eg. unified versus separate instruction and data, number of entries) is implementation-specific, and thus the software for updating the TLB is also implementation-specific. This document provides guidelines for the update mechanism (instruction semantics) for an assumed unified TLB organization.

Translation is activated via bits in the MSR (Machine State Register). The IR (Instruction Relocate) bit activates the translation mechanism for instruction accesses, and the DR (Data Relocate) bit activates the translation mechanism for data accesses. These bits may be specified independently and changed at any time by a program in supervisor state. Note that IR and DR are cleared and the processor is placed in the supervisor state by any interrupt. Any subsequent discussion about translation or protection will assume that the appropriate IR or DR bit in the MSR is set.

### 6.4.1 Virtual to Real Address Translation

A program references memory using the effective address computed by the processor when it executes a load, store, branch, or cache instruction, and when it fetches the next instruction. The effective address is translated to a real address according to the procedures described in this section. The memory subsystem uses the real address for the access.

In the translation process, the effective address and an 8-bit process ID are used to create a 40-bit virtual address. This address is used to locate the associated entry in the TLB. The TLB contains pointers to the location in physical memory where the data referred to by the virtual address is contained. See Figure 6-2.



**Figure 6-2. Effective to Real Address Translation Flow**

## 6.4.2 Translation Lookaside Buffer (TLB)

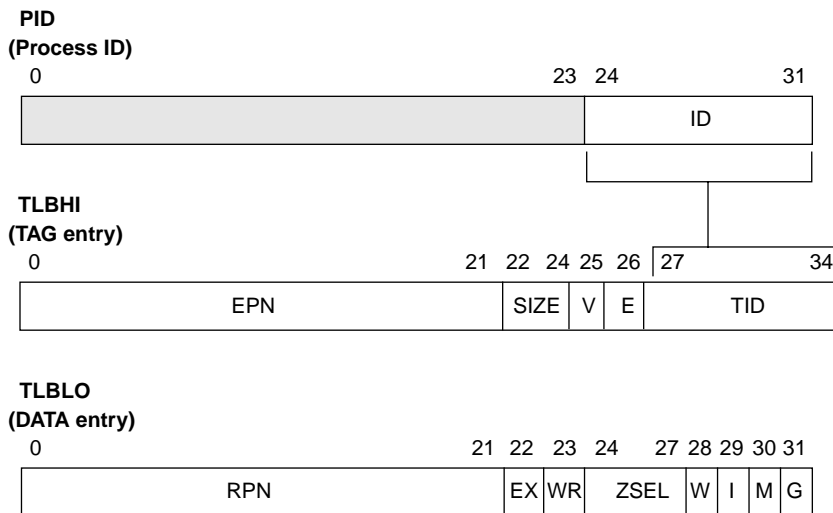
The TLB is the hardware resource that controls translation, protection, and storage attributes. The organization of the TLB is implementation-specific. It can be unified or there can be separate instruction and data TLB's. There can be any number of entries. For the purposes of this discussion, a unified TLB organization is assumed. The differences for an implementation with separate instruction and data TLB's are for the most part obvious (eg. separate instructions for reading and writing each TLB). For details on how to synchronize TLB updates with instruction execution see Section 11.3, "Synchronization Requirements," on page 11-2.

Since the size of a page represented by a given TLB entry is variable, the TLB by necessity is fully-associative. That is, the translation entry for a given effective address can be placed anywhere in the TLB. Maintenance of TLB entries is under software control. System level software completely determines TLB entry replacement strategy and the format and use of

any page state information. The TLB entry contains all the information required to identify the page, to specify the translation and protection controls, and to specify the storage attributes. Subsequent sections will describe the fields in detail.

The TLB contains a high portion and a low portion. The high portion contains 35 bits per entry, and the low portion contains 32 bits per entry. In normal operation with translation enabled, some or all of the incoming effective address bits  $EA_{0:21}$  are compared with some or all of the effective page number bits  $EPN_{0:21}$  based on the size bits  $SIZE_{0:2}$ . All entries are simultaneously checked for a match. If an entry matches, the corresponding bits from the RPN field of the TLB low portion provide the real page number (RPN), and the rest of the TLB low portion provides the access control bits (ZSEL, EX, WR), and storage attributes (W, I, M, G, E). It is a programming error if multiple entries match during a TLB look-up, and the results of such a look-up are undefined.

The virtual address space is extended from the 32-bit effective address space by defining an 8-bit process ID (PID) register and an associated translation ID (TID) field in the TLB which is also compared during the TLB look-up (see Section 6.4.4.1, “TID TLB Field and Process ID (PID) Register,” on page 6-16).



**Figure 6-3. TLB Entries**

The high and low portions of TLB entries are written by copying information from a GPR and the PID, using the **tlbwe** instruction (see Section 6.5.2.5, “TLB Write Entry (tlbwe),” on page 6-28). The high and low entries are read by copying information to a GPR and the PID, using the **tlbre** instruction (see Section 6.5.2.2, “TLB Read Entry (tlbre),” on page 6-26). Software can also search for specific entries using the **tlbsx** instruction (see Section 6.5.2.3, “TLB Search Indexed (tlbsx),” on page 6-27).



### 6.4.3 TLB Fields

Each TLB entry describes a page that is eligible for translation and access controls. Fields in the TLB entry fall into four categories:

1. Page identification fields (information required to identify the page to the hardware translation mechanism).
2. Translation field
3. Access control fields
4. Storage attribute fields

#### 6.4.3.1 Page Identification Fields

When an effective address is presented to the MMU for processing, the MMU applies several selection criteria to each TLB entry to select the appropriate entry. Although it is possible to place multiple entries into the TLB that match a specific effective address and PID, this is a programming error and the results are undefined. The following fields in the TLB entry identify the page. Except as noted, all comparisons must succeed to validate this entry for subsequent processing.

- **EPN — Effective Page Number** (22 bits)

This field is compared to some number of bits 0:21 of the effective address (EA) presented to the MMU. The exact comparison depends on the page size and is specified in Table 6-1 below:

**Table 6-1. TLB Fields Related to Page Size**

Page Size	SIZE Field	“n” High-Order Bits Compared	EPN to EA Comparison	RPN Bits Required = 0
1K	000	22	$EPN_{0:21} \leftrightarrow EA_{0:21}$	none
4K	001	20	$EPN_{0:19} \leftrightarrow EA_{0:19}$	$RPN_{20:21} = 0$
16K	010	18	$EPN_{0:17} \leftrightarrow EA_{0:17}$	$RPN_{18:21} = 0$
64K	011	16	$EPN_{0:15} \leftrightarrow EA_{0:15}$	$RPN_{16:21} = 0$
256K	100	14	$EPN_{0:13} \leftrightarrow EA_{0:13}$	$RPN_{14:21} = 0$
1M	101	12	$EPN_{0:11} \leftrightarrow EA_{0:11}$	$RPN_{12:21} = 0$
4M	110	10	$EPN_{0:9} \leftrightarrow EA_{0:9}$	$RPN_{10:21} = 0$
16M	111	8	$EPN_{0:7} \leftrightarrow EA_{0:7}$	$RPN_{8:21} = 0$

- **SIZE — Page Size** (3 bits)

Selects 1 of 8 page sizes (1K, 4K, 16K, 64K, 256K, 1M, 4M, 16M). The SIZE field controls how many bits of EA and EPN are compared, and correspondingly how many bits from the EA are replaced with bits from the RPN. See Table 6-1.

- **V — Valid** (1 bit)

This bit indicates that this TLB entry is valid and may be used for translation. The presence of a valid TLB entry implies read access unless overridden by zone protection. The valid bit for a given entry can be set or cleared with a TLB write entry (**tlbwe**) instruction; alternatively, the Valid bit for all entries may be simultaneously cleared by a TLB Invalidate All (**tlbia**) instruction.

- **TID — TLB Identifier** (8 bits)

This field is copied from the PID register during a TLB write entry (**tlbwe**) operation. The value of the TID field is compared with the value of the Process ID (PID) register (see Figure 6-4 on page 6-16) during TLB accesses, as an extension to the effective address. This mechanism provides a convenient way to associate a translation with one of 255 unique software entities, typically a process maintained by system level software. A TID value of 0x00 disables the TID/PID comparison and can be used to identify a TLB entry as valid for all processes. As such, the current value of the PID is irrelevant for a TLB entry with a TID value of 0x00.

#### 6.4.3.2 Translation Field

After a TLB entry has been identified as being valid and matching the effective address and (possibly) the PID, the following field defines how the effective address will be translated:

- **RPN — Real Page Number** (22 bits)

The RPN bits are used to replace some (or all) of EA<sub>0:21</sub>, according to page size. For example, a 16K page uses EA<sub>0:17</sub> for comparison. In this example, the translation mechanism replaces EA<sub>0:17</sub> with RPN<sub>0:17</sub> to form the physical address, and EA<sub>18:31</sub> carries forward to become the real page offset. See Figure 6-2 on page 6-11 and Table 6-1 on page 6-13.

NOTE: Software is required to set all unused low-order bits of RPN (as determined by page size) to 0. See Table 6-1 on page 6-13.

#### 6.4.3.3 Access Control Fields

The following fields define the access protection controls. See Section 6.4.4 on page 6-15.

- **ZSEL — Zone Select** (4 bits)

This field selects one of 16 zone fields (Z0-Z15) from the Zone Protection Register (ZPR). The selected ZPR field is used to modify the access protection specified by the V, EX, and WR bits of the TLB entry and is described in detail in Section 6.4.4.4, “ZSEL TLB Field and Zone Protection Register (ZPR),” on page 6-17.

- **EX — Execute Enable** (1 bit)

Controls whether instructions may be fetched and executed from this page.

- **WR — Write Enable** (1 bit)

Controls whether store operations (including **dcbi** and **dcbz**) are permitted to this page.

#### 6.4.3.4 Storage Attribute Fields

These fields define the storage attributes for the page when being accessed with relocation active. In real mode (relocation disabled), these attributes come from the real mode storage attribute control registers. For additional information on the function of the Storage attributes and the real mode Storage attribute control registers see Section 6.3.

- **W — Write Through** (1 bit )

Controls the data cache WriteThrough storage attribute.

- **I — Inhibited** (1 bit )

Controls the Cache Inhibited storage attribute.

- **M — Memory Coherent** (1 bit )

Controls the Memory Coherence storage attribute.

- **G — Guarded** (1 bit )

Controls the Guarded storage attribute.

- **E — Endianness** (1 bit )

Controls the Endian storage attribute.

#### 6.4.4 Storage Protection

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

Since the protection mechanism operates as part of the address translation mechanism, storage protection applies to translated accesses only. Instruction storage access protection is active only when MSR[IR]=1. Data storage access protection is active only when MSR[DR]=1.

If the appropriate relocate bit in the MSR is set to 1, each effective address is translated to a real address before the storage access is performed. A *TLB-Miss exception* occurs if there is no valid entry in the TLB for the page specified by the effective address (Instruction or Data TLB-Miss interrupt). A *storage exception* occurs if the appropriate TLB entry is found but the access is not allowed by the storage protection mechanism (Instruction or Data Storage interrupt). See Section 7.9, “Interrupt Definitions,” on page 7-27 for additional information about these and other interrupt types.

In certain cases, a storage or TLB-Miss exception may result in the restart of (re-execution of at least part of) a load or store instruction (see Section 2.8.1, “Instruction Restart,” on page 2-19).

**6.4.4.1 TID TLB Field and Process ID (PID) Register**

The first level of storage protection is the Translation ID (TID) field of the TLB entry. This 8 bit field, if non-zero, is compared to the contents of the Process ID (PID) register as part of address translation (see Figure 6-4 and Figure 6-2). These must match if access to the page represented by this TLB entry is to be allowed. In typical use, it is assumed that a Supervisor State program (such as an operating system) set the PID register prior to enabling the Problem State program which is subject to access control.

If the TID field of a TLB entry is zero, then the associated memory page is accessible to all programs, regardless of their Process ID. This feature allows for common code or data to be shared by multiple processes. This common area is still subject to all of the other storage protection mechanisms described below.



**Figure 6-4. Process ID (PID)**

0:23		reserved
24:31		Process ID

**6.4.4.2 EX TLB Field**

The EX bit of the TLB entry controls *execute* access to the page.

If the TLB entry corresponding to a particular effective address and Process ID has EX=0, then instructions from that page will not be fetched, and will not be placed into any cache as the result of a fetch request to that page. See Section 6.2.1 on page 6-2. Furthermore, if the Sequential Execution Model calls for the execution of an instruction from that page, an Instruction Storage interrupt (ISI) will be generated.

The zone protection mechanism (see Section 6.4.4.4) can override the EX protection mechanism.

#### 6.4.4.3 WR TLB Field

The WR bit of the TLB entry controls *write* access to the page. *Read* access is implied by the presence of a valid TLB entry matching the requested effective address and Process ID.

If the TLB entry corresponding to a particular effective address and Process ID has WR=0, then any attempted store type access (including **dcbi** and **dcbz**) to the page will cause the instruction execution to be suppressed and a Data Storage interrupt (DSI) will be generated.

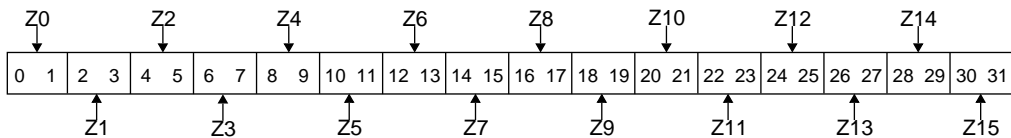
The zone protection mechanism (see Section 6.4.4.4 ) can override the WR protection mechanism.

#### 6.4.4.4 ZSEL TLB Field and Zone Protection Register (ZPR)

Each TLB entry contains a 4-bit Zone Select (ZSEL) field. A zone is an arbitrary identifier for grouping TLB entries and their associated memory pages for purposes of protection. Up to 16 different groups (zones) may be defined. Any one group may have any number of member pages.

Each zone is associated with a 2-bit field in the Zone Protection Register (ZPR). The values of the field define how protection will be applied to every page which is a member of that zone. The protection attributes of every page in the zone may be altered by changing the value in the ZPR. Without the ZPR mechanism, the change would require finding, reading, altering, and rewriting the TLB entry for each page in the zone, individually. The ZPR provides a much faster means of altering the protection for groups of memory pages represented by TLB entries with a common ZSEL value.

The ZSEL values 0 – 15 select ZPR fields Z0 – Z15, respectively. The fields are defined within the ZPR as shown in Figure 6-5:



**Figure 6-5. Zone Protection Register (ZPR)**

0:1	Z0	TLB Page Access Control for all pages in this Zone. EX (Execute Enable) and WR (Write Enable) are bits from the TLB entry associated with the requested effective address and Process ID.	
		In Problem State (MSR[PR] = 1) 00 no access 01 access governed by EX and WR 10 access governed by EX and WR 11 access as if EX and WR both set	In Supervisor State (MSR[PR] = 0) 00 access governed by EX and WR 01 access governed by EX and WR 10 access as if EX and WR both set 11 access as if EX and WR both set
2:3	Z1	see description of Z0	
4:5	Z2	see description of Z0	
6:7	Z3	see description of Z0	
8:9	Z4	see description of Z0	
10:11	Z5	see description of Z0	
12:13	Z6	see description of Z0	
14:15	Z7	see description of Z0	
16:17	Z8	see description of Z0	
18:19	Z9	see description of Z0	
20:21	Z10	see description of Z0	
22:23	Z11	see description of Z0	
24:25	Z12	see description of Z0	
26:27	Z13	see description of Z0	
28:29	Z14	see description of Z0	
30:31	Z15	see description of Z0	

It is not required that the EX and WR fields for all member pages of a group to be identical. The ZPR field alters the protection defined by the EX and WR bits, on a page-by-page basis,

as defined in Figure 6-5. An application program (running in Problem State) may have access as defined by EX and WR of the individual pages, or no access (denies read access as well as write and execute access), or complete access.

A '00' value of a ZPR field is the only available mechanism to deny read access to a page defined by an otherwise valid TLB entry. The EX and WR bit definitions do not allow for read protection.

For a given ZPR field value, a program in Supervisor State always has equal or greater access than a program in Problem State. The Supervisor can never be denied read (load) access for a valid TLB entry.

If any access is attempted while in Problem State to a page whose associated ZPR field is b'00', a Zone exception type of Instruction or Data Storage interrupt will be generated.

#### 6.4.4.5 Protection Applied to Cache Management Instructions

**dcbi** and **dcbz** are treated as stores since they can change data (or cause loss of data by invalidating a dirty line). As such, they both can cause WR = 0 exception type Data Storage interrupts. **dcbz** can cause a Zone exception type Data Storage interrupt; **dcbi** cannot, since it is a privileged instruction and cannot be executed in Problem State.

**dcbt** and **dcbtst** are treated as loads with respect to protection, and therefore cannot cause WR = 0 exception type Data Storage interrupts. If a Zone exception results from the execution of **dcbt** or **dcbtst**, the instruction becomes a no-op and a Data Storage interrupt does not occur.

**dcbf** and **dcbst** are treated as loads with respect to protection. Flushing or storing a line from the cache is not considered a store since the store has already been done to update the cache and the **dcbf** or **dcbst** is only updating the copy in main storage. Therefore, neither **dcbf** nor **dcbst** can cause WR = 0 exception type Data Storage interrupts. Since neither of these instructions are privileged, they can both cause Zone exception type Data Storage interrupts.

**icbi** and **icbt** are considered loads with respect to protection, and so cannot cause WR = 0 exception type Data Storage interrupts. **icbi** can cause a Zone exception type Data Storage interrupt, but **icbt** cannot, since it is a privileged instruction.

**Table 6-2. Protection Applied to Cache Instructions**

Instruction	Zone Exception ?	WR = 0 Exception ?
dcbf	Yes	No
dcbi	No	Yes
dcbst	Yes	No
dcbt	Yes (no DSI)	No
dcbtst	Yes (no DSI)	No
dcbz	Yes	Yes
icbi	Yes	No
icbt	No	No

#### 6.4.4.6 Protection Applied to String Instructions

When the string length is zero, neither **lswx** nor **stswx** can cause Zone exception type Data Storage interrupts, nor can **stswx** cause WR=0 exception type Data Storage interrupts.

#### 6.4.5 Page Reference and Change Recording

When performing physical page management, it is useful to know whether a given physical page has been referenced or altered. Note that this may be more involved than whether a given TLB entry has been used to reference or alter memory, since multiple TLB entries may translate to the same physical page. If it is necessary to replace the contents of some physical page with other contents, a page which has been referenced (accessed for any purpose) is more likely to be maintained than a page which has never been referenced. If the contents of a given physical page are to be replaced, then the contents of that page must be written to the backing store before replacement, if anything in that page has been changed. Software must maintain records to control this process.

Similarly, when performing TLB management, it is useful to know whether a given TLB entry has been referenced. When making a decision about which entry to cast-out of the TLB, an entry which has been referenced is more likely to be maintained in the TLB than an entry which has never been referenced.

TLB-miss exceptions may be used to allow software to maintain reference information for a TLB entry and for its associated physical page. The entry is built, with its valid bit and its WR bit off, and the index and effective page number of the entry are retained by software. The first attempt of application code to use the page will cause a miss exception (because the entry is marked invalid). The miss handler records the reference to the TLB entry and to the associated physical page in a software table, and then turns on the valid bit (note that the



valid bit itself may be regarded as a reference bit for the TLB entry). Subsequent read accesses to the page via this TLB entry will proceed normally.

In a demand-paged environment, when the contents of a physical page are to be replaced, if any storage in that physical page has been altered, then the backing storage must be updated. The information that a physical page is dirty is typically recorded in a 'change' bit for that page.

Data storage exceptions may be used to allow software to maintain change information for a physical page. For the example just given for reference recording, the first write access to the page via the TLB entry, after the entry has been made valid, will create a WR=0 data storage exception. The miss handler records the change status to the physical page in a software table, and then turns on the WR bit. All subsequent accesses to the page via this TLB entry will proceed normally.

### **6.4.6 TLB Management**

The IBM PowerPC Embedded Environment does not imply any format for the page tables or the page table entries. Software has significant flexibility in implementing a custom replacement strategy. For example, software may choose to lock TLB entries that correspond to frequently used storage, so that those entries are never cast out of the TLB and TLB Miss exceptions to those pages never occur.

TLB management is performed in software with some hardware assist. This hardware assist consists of:

- Automatic recording of the effective address causing a TLB Miss exception. For Instruction TLB Miss exceptions, the address is saved in the Save/Restore Register 0 (SRR0). For Data TLB Miss exceptions, the address is saved in the Data Exception Address Register (DEAR).
- Instructions for reading, writing, searching, invalidating, and synchronizing the TLB (see Section 6.5.2, "TLB Management Instructions," on page 6-25).

## 6.5 Storage Control Instructions

Storage control Instructions consist of cache management instructions and translation lookaside buffer management instructions.

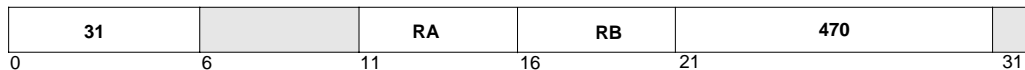
### 6.5.1 Cache Management Instructions

The two privileged cache management instructions consist of:

1. Data Cache Block Invalidate (**dcbi**)
2. Instruction Cache Block Touch (**icbt**).

#### 6.5.1.1 Data Cache Block Invalidate (dcbi)

**dcbi** RA, RB


$$EA \leftarrow (RA|0) + (RB)$$

DCBI(EA)

An effective address is formed by adding an index to a base address. The index is the contents of register RB. The base address is 0 if the RA field is 0 and is the contents of register RA otherwise.

The action taken depends on the storage mode associated with the block containing the byte addressed by EA and on the state of that block. The list below describes the action taken for the various cases.

## Coherence Required

- Unmodified Block

Invalidate copies of the block in the data caches of all processors.

- Modified Block

Invalidate copies of the block in the data caches of all processors (discard the modified contents).

- Absent Block

If copies of the block are in the data caches of other processors, cause them to be invalidated in those data caches (discard any modified contents).

## Coherence Not Required

- Unmodified Block

Invalidate the block in the processor's data cache.

- **Modified Block**  
Invalidate the block in the processor's data cache (discard the modified contents).
- **Absent Block**  
Do nothing.

The function of this instruction is independent of the Write Through Required/Not Required and Caching Inhibited/Allowed modes of the block containing the byte addressed by EA.

**Registers Altered**

- None

**Invalid Instruction Forms**

- Reserved fields

**Exceptions**

This instruction is considered a store with respect to address translation, storage protection, and Data Address Compare (DAC) debug exceptions.

**Programming Note**

Execution of this instruction is privileged.

**6.5.1.2 Instruction Cache Block Touch (icbt)**

**icbt**                    RA,RB



$$EA \leftarrow (RA|0) + (RB)$$

$$ICBT(EA)$$

An effective address is formed by adding an index to a base address. The index is the contents of register RB. The base address is 0 if the RA field is 0 and is the contents of register RA otherwise.

This instruction is a hint that performance will probably be improved if the block containing the byte addressed by EA is fetched into the instruction cache, because the program will probably soon execute code from the addressed location. The hint is ignored if the block is Caching Inhibited. Executing **icbt** will not cause the system error handler to be invoked.

It is permissible in all circumstances for an implementation to treat this instruction as a no-op. If the instruction is not treated as a no-op, it should be handled in the following way:

- If the instruction block at the effective address is not in the instruction cache, and the effective address is marked as Caching Allowed, the block is read from main storage into the instruction cache.
- If the instruction block at the effective address is already in the instruction cache, or if the effective address is marked as Caching Inhibited, no operation is performed.

### **Registers Altered**

- None

### **Invalid Instruction Forms**

- Reserved fields

### **Exceptions**

This instruction is considered a load with respect to address translation and storage protection, and with respect to Data Address Compare (DAC) Debug exceptions. However, this instruction is not allowed to cause a Data Storage interrupt or a Data TLB Miss interrupt. If execution of the instruction causes an exception which would generate such an interrupt, then no operation is performed, and no interrupt occurs. The instruction *is* permitted, however, to cause a Debug interrupt.

### **Programming Notes**

Execution of this instruction is privileged.

This instruction allows a program to begin a cache block fetch from main storage before the program needs the instruction. The program can later branch to the instruction address and fetch the instruction from the cache without incurring the latency of a cache miss.

Instruction cache ops use MSR[DR], not MSR[IR], to determine translation of their operands (since they are treated as loads with respect to translation). When data translation is disabled, cacheability for the effective address of the operand of instruction cache ops is determined by the ICCR, not the DCCR.

## 6.5.2 TLB Management Instructions

While the PowerPC Architecture describes logically separate instruction fetch and integer (including effective address computation) execution units, the programming model is that there is one translation mechanism.

Each IBM PowerPC Embedded Environment implementation that has an MMU also includes the following instructions for managing the TLB. On implementations without an MMU, these instructions will be treated as illegal.

### 6.5.2.1 TLB Invalidate All (tlbia)

**tlbia**



All of the entries in the TLB are invalidated and become unavailable for translation by clearing the valid (V) bit in the TLBHI portion of each TLB entry. The rest of the fields in the TLB entries are unmodified.

The TLB is invalidated regardless of the settings of MSR[IR] and MSR[DR].

#### Registers Altered

- None.

#### Invalid Instruction Forms

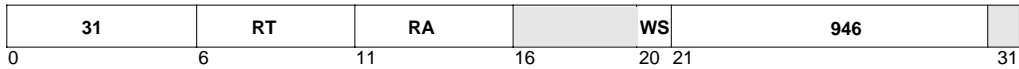
- None.

#### Programming Notes

Execution of this instruction is privileged. Translation is not required to be active during the execution of this instruction.

The effects of the invalidation are not guaranteed to be visible to the programming model until the completion of a context synchronizing operation (see Section 11.1, “Context Synchronization,” on page 11-1).

<b>tlbre</b>	RT, RA, WS
--------------	------------



```

if WS = 1
    (RT)  $\leftarrow$  TLBLO[(RA)]
else
    (RT)  $\leftarrow$  TLBHI[(RA)]
    (PID)  $\leftarrow$  TID from TLB[(RA)]

```

The contents of the selected TLB entry are placed into register RT (and possibly into PID).

The contents of RA are used as an index into the TLB. If this index specifies a TLB entry that does not exist (that is, the index value is greater than the size of the TLB), the results are undefined.

The WS field specifies which portion (TLBHI or TLBLO) of the entry is loaded into RT. If TLBHI is being accessed, the PID SPR is set to the value of the TID field in the TLB entry.

## Registers Altered

- RT
- PID (if WS = 0)

## Invalid Instruction Forms

- Reserved fields

## Programming Notes

Execution of this instruction is privileged. Translation is not required to be active during the execution of this instruction.

The contents of RT after the execution of this instruction are interpreted as follows:

If  $WS = 0$  (TLBHI):

```

RT[0:21] ← EPN[0:21]
RT[22:24] ← SIZE[0:2]
RT[25] ← V
RT[26] ← E
RT[27:31] ← 0
PID[24:31] ← TID[0:7]; (note that the TID is copied to the PID, not to RT)

```

If  $WS = 1$  (TLBLO):

RT[0:21] ← RPN[0:21]  
RT[22:23] ← EX,WR  
RT[24:27] ← ZSEL[0:3]  
RT[28:31] ← WIMG

6.5.2.3 TLB Search Indexed (tlbsx)

tlbsx                    RT,RA,RB                    (Rc=0)  
tlbsx.                  RT,RA,RB                    (Rc=1)

31	RT	RA	RB	914	Rc
0	6	11	16	21	31

EA ← (RA|0) + (RB)  
if Rc = 1  
    CR[CR0]<sub>LT</sub> ← 0  
    CR[CR0]<sub>GT</sub> ← 0  
    CR[CR0]<sub>SO</sub> ← XER[SO]  
if Valid TLB entry matching EA and PID is in the TLB then  
    (RT) ← Index of matching TLB Entry  
    if Rc = 1  
        CR[CR0]<sub>EQ</sub> ← 1  
else  
    (RT) Undefined  
    if Rc = 1  
        CR[CR0]<sub>EQ</sub> ← 0

An effective address is formed by adding an index to a base address. The index is the contents of register RB. The base address is 0 if the RA field is 0 and is the contents of register RA otherwise.

The TLB is searched for a valid entry which translates EA and PID. See Section 6.4.3.1, “Page Identification Fields,” on page 13. The record bit (Rc) specifies whether the results of the search will affect CR[CR0] as shown above. The intention is that CR[CR0]<sub>EQ</sub> can be tested after a **tlbsx.** instruction if there is a possibility that the search may fail.

Registers Altered

- CR[CR0]<sub>LT, GT, EQ, SO</sub> if Rc contains 1

Invalid Instruction Forms

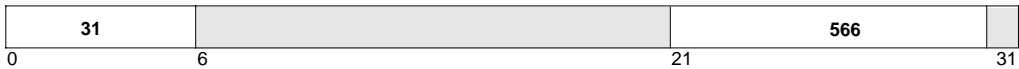
- None.

Programming Note

Execution of this instruction is privileged. Translation is not required to be active during the execution of this instruction.

### 6.5.2.4 TLB Synchronize (tlbsync)

#### tlbsync



The **tlbsync** instruction does not complete until all previous **tlbwe** and **tlbia** instructions executed by the processor executing this instruction have been received and completed by all other processors.

This instruction is optional in the IBM PowerPC Embedded Environment even for implementations that include an MMU.

The operation performed by this instruction is treated as a caching inhibited and guarded data access with respect to the ordering done by **eieio**.

#### Registers Altered

- None.

#### Invalid Instruction Forms

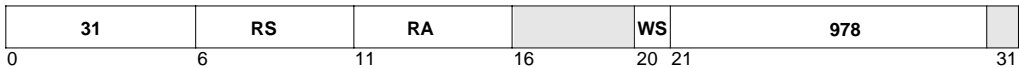
- Reserved fields.

#### Programming Note

Execution of this instruction is privileged. Translation is not required to be active during the execution of this instruction.

### 6.5.2.5 TLB Write Entry (tlbwe)

#### tlbwe            RS , RA, WS



```
if WS = 1
    TLBLO[(RA)] ← (RS)
else
    TLBHI[(RA)] ← (RS)
    TID of TLB[(RA)] ← (PID)
```

The contents of the selected TLB entry is replaced with the contents of register RS (and possibly PID).



The contents of RA are used as an index into the TLB. If this index specifies a TLB entry that does not exist (that is, the index value is larger than the number of the TLB entries), the results are undefined.

The WS field specifies which portion (TLBHI or TLBLO) of the entry is replaced from RS. For instructions that specify TLBHI, the TID field in the TLB entry is supplied from PID.

### Registers Altered

- None.

### Invalid Instruction Forms

- Reserved fields

#### Programming Notes

Execution of this instruction is privileged. Translation is not required to be active during the execution of this instruction.

The effects of the update are not guaranteed to be visible to the programming model until the completion of a context synchronizing operation. For example, updating a zone selection field within the TLB while in supervisor code should be followed by an **isync** instruction (or other context synchronizing operation) to guarantee that the desired translation and protection domains are used. See Section 11.1, “Context Synchronization,” on page 11-1

The TLB fields are written from RS by this instruction as follows:

If WS = 0 (TLBHI):

EPN[0:21]  $\leftarrow$  RS[0:21]  
SIZE[0:2]  $\leftarrow$  RS[22:24]  
V  $\leftarrow$  RS[25]  
E  $\leftarrow$  RS[26]  
TID[0:7]  $\leftarrow$  PID[24:31]; (note that the TID is written from the PID, not RS)

If WS = 1 (TLBLO):

RPN[0:21]  $\leftarrow$  RS[0:21]  
EX,WR  $\leftarrow$  RS[22:23]  
ZSEL[0:3]  $\leftarrow$  RS[24:27]  
WIMG  $\leftarrow$  RS[28:31]



# 7

## Interrupts and Exceptions

---

This chapter discusses in detail the following topics:

- Interrupt Classes
- Interrupt Processing
- Interrupt and Exception Types
- Partially Executed Instructions
- Interrupt Ordering and Masking
- Exception Priorities
- Exception Handling Registers
- Interrupt Definitions
- Interrupt Control Instructions

### 7.1 Overview

An *Interrupt* is the action in which the processor saves its old context (Machine State Register (MSR) and instruction pointer) and begins execution at a pre-determined interrupt-handler address, with a modified MSR. *Exceptions* are the events which will, if enabled, cause the processor to take an interrupt.

In the IBM PowerPC Embedded Environment, exceptions are generated by signals from internal and external peripherals, instructions, the internal timer facility, debug events, or error conditions.

All interrupts, except Machine Check, are ordered within the two categories of non-critical and critical, such that only one interrupt of each category is reported, and when it is processed (taken) no program state is lost. Since Save/Restore Register pairs SRR0/SRR1 and SRR2/SRR3 are serially reusable resources used by all non-critical and critical interrupts respectively, program state may be lost when an unordered interrupt is taken (see Section 7.6, “Interrupt Ordering and Masking,” on page 7-11).

All interrupts, except Machine Check, are context synchronizing as defined in Section 11.1, “Context Synchronization,” on page 11-1. A Machine Check interrupt acts like a context synchronizing operation with respect to subsequent instructions; that is, a

Machine Check interrupt need not satisfy items 1-3 of Section 11.1 but does satisfy items 4-5.

## 7.2 Interrupt Classes

All interrupts, except for Machine Check, can be categorized according to three independent characteristics of the interrupt:

1. Synchronous/Asynchronous Interrupts
2. Precise/Imprecise Interrupts
3. Critical/Non-Critical Interrupts

### 7.2.1 Synchronous/Asynchronous Interrupts

Synchronous interrupts are those which are caused directly by the execution (or attempted execution) of instructions. Synchronous interrupts may be either precise or imprecise.

Asynchronous interrupts are caused by events which are independent of instruction execution. All asynchronous interrupts are precise.

### 7.2.2 Precise/Imprecise Interrupts

Precise interrupts are those which *precisely* indicate the address of the instruction causing the exception which generated the precise, synchronous interrupt; or, for certain precise synchronous interrupts, the address of the immediately following instruction. For asynchronous precise interrupts the address reported to the exception handling routine is the address of the next instruction that would have been executed, had the interrupt not occurred.

#### 7.2.2.1 Precise Interrupts

For synchronous precise interrupts, the following rules apply:

1. The save/restore register 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. However, some storage accesses generated by these preceding instructions may not have been performed with respect to all other processors and mechanisms.
3. The instruction causing the exception may appear not to have begun execution (except for causing the exception), may have partially completed, or may have completed, depending on the interrupt type (see Section 7.5, "Partially Executed Instructions," on page 7-10).

4. Architecturally, no subsequent instruction has begun execution.

For asynchronous precise interrupts, the following rules apply:

1. All instructions prior to the one whose address is reported to the exception handling routine (in the save/restore register) have completed execution with respect to the interrupting processor. However, some storage accesses generated by these preceding instructions may not have been performed with respect to all other processors and mechanisms.
2. No instruction subsequent to one whose address is reported to the exception handling routine has begun execution.
3. The instruction whose address is reported to the exception handling routine may appear not to have begun execution, or may have partially completed (see Section 7.5, “Partially Executed Instructions,” on page 7-10).

#### 7.2.2.2 Imprecise Interrupts

All Imprecise interrupts are synchronous and follow the rules outlined below:

1. The save/restore register addresses either the instruction causing the exception or some instruction following the instruction causing the exception that generated the interrupt.
2. An interrupt is generated such that all instructions preceding the instruction addressed by the save/restore register appear to have completed with respect to the executing processor.
3. If the imprecise interrupt is forced by the context synchronizing mechanism, due to an instruction that causes another exception which generates an interrupt (e.g., Alignment, Data Storage), then the save/restore register addresses the interrupt-forcing instruction, and the interrupt-forcing instruction may have been partially executed (see Section 7.5, “Partially Executed Instructions,” on page 7-10).
4. If the imprecise interrupt is forced by the execution synchronizing mechanism, due to executing an execution synchronizing instruction other than **sync** or **isync**, then the save/restore register addresses the interrupt-forcing instruction, and the interrupt-forcing instruction appears not to have begun execution (except for its forcing the imprecise interrupt). If the imprecise interrupt is forced by a **sync** or **isync** instruction, then the save/restore register may address either the **sync** or **isync** instruction or the following instruction.
5. If the imprecise interrupt is not forced by either the context synchronizing mechanism or the execution synchronizing mechanism, then the instruction addressed by the save/restore register may have been partially executed (see Section 7.5, “Partially Executed Instructions,” on page 7-10).
6. No instruction following the instruction addressed by the save/restore register appears to have begun execution.

### 7.2.3 Critical/Non-Critical Interrupts

Critical interrupts are those which use Save/Restore Register pair SRR2/SRR3. Non-Critical interrupts use Save/Restore Register pair SRR0/SRR1.

### 7.2.4 Machine Check Interrupts

Machine Check interrupts are a special case. They are typically caused by some kind of hardware or storage subsystem failure, or by an attempt to access an invalid address. A Machine Check may be caused indirectly by the execution of an instruction, but not be recognized and/or reported until long after the processor has executed past the instruction which caused the Machine Check. As such, Machine Check interrupts cannot properly be thought of as synchronous or asynchronous, nor as precise or imprecise. They are handled as critical class interrupts however. In the case of Machine Check, the following general rules apply:

1. No instruction after the one whose address is reported to the machine check handler software in the save/restore register has begun execution.
2. The instruction whose address is reported to the machine check handler software in the save/restore register, and all prior instructions, may or may not have completed successfully. All those instructions which are ever going to complete appear to have done so already, and have done so within the context existing prior to the machine check interrupt. No further interrupt (other than possible additional Machine Checks) will occur as a result of those instructions.

## 7.3 Interrupt Processing

Associated with each kind of interrupt is an *Interrupt Vector*, which is the address of the initial instruction that is executed when the corresponding interrupt occurs.

Interrupt processing consists of saving a small part of the processor's state in certain registers, identifying the cause of the interrupt in another register, and continuing execution at the corresponding interrupt vector location. When an exception exists that will cause an interrupt to be generated and it has been determined that the interrupt can be taken, the following actions are performed in order:

1. SRR0 (for non-critical class interrupts) or SRR2 (for critical class interrupts) is loaded with an instruction address that depends on the type of interrupt; see the specific interrupt description for details.
2. The ESR is loaded with information specific to the exception type. Note that many interrupt types can only be caused by a single type of exception event, and thus do not need nor use an ESR setting to indicate to the interrupt handling routine what the cause of the interrupt was.
3. SRR1 (for non-critical class interrupts) or SRR3 (for critical class interrupts) is loaded with a copy of the MSR.

4. The MSR is modified in the following fashion:

MSR[ILE] is copied into MSR[LE], leaving MSR[ILE] unmodified.

MSR[APE, APA, WE, EE, PR, FPA, FE0, FE1, IR, DR] are cleared by all interrupts.

MSR[CE, DE] are cleared by critical class interrupts and left unchanged by non-critical class interrupts.

MSR[ME] is cleared by machine check interrupts and left unchanged by all other interrupts.

Other defined MSR bits are left unchanged by all interrupts.

5. Instruction fetching and execution resumes, using the new MSR value, at a location specific to the interrupt type. The location is determined by concatenating the interrupt vector's offset (see Table 7-1, "Interrupt and Exception Types," on page 7-6) to the right of the high-order 16 bits of the Exception Vector Prefix Register (EVPR). The contents of the EVPR are indeterminate upon reset, and must be initialized by system software via the **mtspr** instruction.

Interrupts do not clear reservations obtained with **lwarx**. The operating system should do so at appropriate points, such as at process switch.

At the end of a non-critical interrupt handling routine, execution of a return from interrupt (**rfi**) instruction causes the contents of SRR0 and SRR1 to be restored to the program counter and the MSR, respectively. Execution then resumes at the address in the program counter. Likewise, execution of a return from critical interrupt (**rfci**) instruction performs the same function at the end of a critical interrupt handling routine, using SRR2 and SRR3.

### Programming Note

In general, at process switch, due to possible process interlocks and possible data availability requirements, the operating system needs to consider executing the following:

**stwcx.**, to clear the reservation if one is outstanding, to ensure that a **lwarx** in the "old" process is not paired with a **stwcx.** in the "new" process.

**sync**, to ensure that all storage operations of an interrupted process are complete with respect to other processors before that process begins executing on another processor.

**isync** or **rfi/rfci**, to ensure that the instructions in the "new" process execute in the "new" context.

## 7.4 Interrupt and Exception Types

Table 7-1 provides a summary of each interrupt type, and the various exception types which may cause that interrupt. The table lists the interrupt types in order by interrupt vector offset, and also indicates the interrupt class, ESR setting (if any), and interrupt mask bits (if any). A cross reference to the section which defines the interrupt type in detail is provided.

**Table 7-1. Interrupt and Exception Types**

Offset	Interrupt Type	Exception Type	Interrupt Class	ESR Setting See Note 6	Mask Bit(s)	Status Bit	Page
0000	Reserved						
0100	Critical Input	Critical Input	Asynchronous, Precise, Critical		MSR[CE] <b>See Note 2</b>	<b>See Note 2</b>	7-27
0200	Machine Check	Data Machine Check	Critical <b>See Note 1</b>	<b>See Note 10</b>	MSR[ME]	<b>See Note 4</b>	7-28
		Instruction Machine Check					
0300	Data Storage	Zone Protection Violation	Synchronous, Precise, Non-Critical	ESR[DIZ]←1 <b>See Note 5</b>			7-28
		Write Protection Violation		ESR[DIZ]←0 ESR[DST]←1			
0400	Instruction Storage	Zone Protection Violation	Synchronous, Precise, Non-Critical	ESR[DIZ]←1			7-30
		Execute Protection Violation		ESR[DIZ]←0			
		Guarded Execute		ESR[DIZ]←0			
0500	External Input	External Input	Asynchronous, Precise, Non-Critical		MSR[EE] <b>See Note 2</b>	<b>See Note 2</b>	7-31
0600	Alignment	Alignment	Synchronous, Precise, Non-Critical				7-32



**Table 7-1. Interrupt and Exception Types (cont.)**

Offset	Interrupt Type	Exception Type	Interrupt Class	ESR Setting See Note 6	Mask Bit(s)	Status Bit	Page
0700	Program	Illegal Instruction	Synchronous, Precise, Non-Critical	ESR[PIL]←1			7-33
		Privileged Instruction	Synchronous, Precise, Non-Critical	ESR[PPR]←1			
		Trap Instruction	Synchronous, Precise, Non-Critical	ESR[PTR]←1			
		Floating Point Enabled	Synchronous, Non-Critical <b>See Note 7</b>	ESR[PFE]←1	MSR[FE0, FE1]	FPSCR[FEX]	
		Floating Point Unimplemented	Synchronous, Precise, Non-Critical	ESR[PFU]←1			
		Auxiliary Processor Unavailable	Synchronous, Precise, Non-Critical	ESR[PAU]←1			
		Auxiliary Processor Enabled	Synchronous, Non-Critical <b>See Note 8</b>	ESR[PAE]←1	MSR[APE]	<b>See Note 9</b>	
0800	Floating Point Unavailable	Floating Point Unavailable	Synchronous, Precise, Non-Critical				7-37
0900	Reserved						
0A00	Reserved						
0B00	Reserved						
0C00	System Call	System Call	Synchronous, Precise, Non-Critical				7-38
0D00	Reserved						
0E00	Reserved						
0E10	Reserved						
....	.....						
0FF0	Reserved						
1000	Programmable Interval Timer	Programmable Interval Timer	Asynchronous, Precise, Non-Critical		MSR[EE] TCR[PIE]	TSR[PIS]	7-38
1010	Fixed Interval Timer	Fixed Interval Timer	Asynchronous, Precise, Non-Critical		MSR[EE] TCR[FIE]	TSR[FIS]	7-39

**Table 7-1. Interrupt and Exception Types (cont.)**

Offset	Interrupt Type	Exception Type	Interrupt Class	ESR Setting See Note 6	Mask Bit(s)	Status Bit	Page
1020	Watchdog Timer	Watchdog Timer	Asynchronous, Precise, Critical		MSR[CE] TCR[WIE]	TSR[WIS]	7-40
1030	Reserved						
....	.....						
10F0	Reserved						
1100	Data TLB Miss	Data TLB Miss	Synchronous, Precise, Non-Critical	See Note 5			7-41
1110	Reserved						
....	.....						
11F0	Reserved						
1200	Instruction TLB Miss	Instruction TLB Miss	Synchronous, Precise, Non-Critical				7-42
1210	Reserved						
....	.....						
1FF0	Reserved						
2000	Debug	Trap, Instruction Address Compare, Data Address Compare	Synchronous, Precise, Critical		MSR[DE] DBCR[IDM]	DBSR[TR, IAC, DAC] respectively	7-42
		Instruction Complete, Branch Taken	Synchronous, Precise, Critical		DBCR[IDM] See Note 3	DBSR[IC, BT] respectively	
		Exception, Unconditional Debug Event	Asynchronous, Precise, Critical		MSR[DE] DBCR[IDM]	DBSR[EXC, UDE] respectively	
2010	Reserved. Implementation Specific						
....	.....						
2FF0	Reserved. Implementation Specific						

## Table 7-1 Notes

1. Machine Check interrupts are a special case, and are not specifically thought of as either synchronous or asynchronous, nor as precise or imprecise. See Section 7.2, “Interrupt Classes,” on page 7-2.
2. Although it is not specified as part of the IBM PowerPC Embedded Environment, it is common for system implementations to provide, as part of the interrupt controller, independent mask and status bits for the various sources of critical input and external input interrupts.
3. The Instruction Complete and Branch Taken Debug Events are only defined for MSR[DE]=1 when in Internal Debug Mode (DBCR[IDM]=1). In other words, when in Internal Debug Mode with MSR[DE]=0, then Instruction Complete and Branch Taken Debug Events cannot occur, and no DBSR status bits are set and no subsequent imprecise debug interrupt will occur (see Section 9.3, “Debug Events,” on page 9-2.)
4. Machine Check status information is commonly provided as part of the system implementation, but is not part of the IBM PowerPC Embedded Environment.
5. On Zone Protection Violation exception type Data Storage interrupts and Data TLB Miss interrupts, ESR[DST] is also set if the instruction causing the interrupt is a store-type instruction (i.e., one that would require WR=1 in order to be performed). This includes all store instructions, as well as **dcbz** and **dcbi**.
6. In general, when an interrupt causes a particular ESR bit to be set (or cleared) as indicated in the table, it also causes all other ESR bits to be cleared. There may be special rules regarding the handling of implementation-specific ESR bits (see the User’s Manual for the implementation for details). Also see Section 7.8.4, “Exception Syndrome Register (ESR),” on page 7-24, for a complete discussion of the ESR bits.
7. The precision of the Floating Point Enabled exception type Program interrupt is controlled by the MSR[FE0, FE1] bits. When MSR[FE0, FE1]=0b01 or 0b10, the interrupt is imprecise. When MSR[FE0, FE1]=0b11, the interrupt is precise. When MSR[FE0, FE1]=0b00, the interrupt is masked, and will subsequently occur as an imprecise interrupt if and when the MSR[FE0, FE1] bits are enabled. See Section 7.9.7.4, “Floating-Point Enabled Exception,” on page 7-35.
8. The precision of the Auxiliary Processor Enabled exception type Program interrupt is implementation dependent (see the User’s Manual for the implementation for more information).
9. Auxiliary Processor Enabled exception status is commonly provided as part of the implementation, but is not part of the IBM PowerPC Embedded Environment.

10. In general the handling of Machine Check interrupts is implementation dependent. Implementations may choose to define a particular ESR setting to differentiate between the different causes of Machine Check interrupts, (for example, whether the machine check occurred on an instruction fetch or on a data access). The specific definition of the ESR setting for Machine Check, along with any other implementation details, may be found in the User's Manual for the implementation.

## 7.5 Partially Executed Instructions

In general, the architecture permits load and store instructions to be partially executed, interrupted, and then to be restarted from the beginning upon return from the interrupt. In order to guarantee that a particular load or store instruction will complete without being interrupted and restarted, software must mark the storage being referred to as Guarded, and must use an elementary (non-string or multiple) load or store that is aligned on an operand-sized boundary.

In order to guarantee that load and store instructions can, in general, be restarted and completed correctly without software intervention, the following rules apply when an execution is partially executed and then interrupted:

1. For an elementary load, no part of the target register (RT) will have been altered.
2. For *with update* forms of load or store, the update register (RA) will not have been altered.
3. In no case will storage protection be violated.

On the other hand, the following effects are permissible when certain instructions are partially executed and then restarted:

1. For any store, some of the bytes at the target storage location may have been altered. In addition, for **stwcx.**, CR0 has been set to an undefined value, and it is undefined whether the reservation has been cleared.
2. For any load, some of the bytes at the addressed storage location may have been accessed.
3. For load multiple or load string, some of the registers in the range to be loaded may have been altered. Including the addressing registers (RA, and possibly RB) in the range to be loaded is a programming error, and thus the rules for partial execution do not protect against overwriting of these registers.

As previously stated, the only load or store instructions that are guaranteed to not be interrupted after being partially executed are elementary, aligned, guarded loads and stores. All others may be interrupted after being partially executed. The following list identifies the specific instruction types for which interruption after partial execution may occur, as well as the specific interrupt types which could cause the interruption:

1. Any load or store (except elementary, aligned, guarded):

- Critical Input
- Machine Check
- External Input
- Program (Imprecise Mode Floating Point Enabled)
- Program (Imprecise Mode Auxiliary Processor Enabled)
- Programmable Interval Timer
- Fixed Interval Timer
- Watchdog Timer
- Debug (UDE)

2. Unaligned elementary load or store, or any multiple or string:

- All of the above listed under item 1, plus the following:
- Alignment
- Data Storage (if the access crosses a protection boundary)
- Debug (DAC)

## 7.6 Interrupt Ordering and Masking

It is possible for multiple exceptions to exist simultaneously, each of which requires the generation of an interrupt. Furthermore, the architecture does not provide for reporting more than one interrupt of the same class (critical or non-critical) at a time. Therefore, the architecture defines that interrupts are ordered with respect to each other, and provides a masking mechanism for certain persistent interrupt types.

When an interrupt type is masked, and an event causes an exception which would normally generate an interrupt of that type, the exception *persists* as a *status* bit in a register (which register depends upon the exception type). However, no interrupt is generated. Later, if the interrupt type is enabled (unmasked), and the exception status has not been cleared by software, the interrupt due to the original exception event will then finally be generated.

All asynchronous interrupt types can be masked. In addition, certain synchronous interrupt types can be masked. An example of such an interrupt type is the Floating Point Enabled exception type Program interrupt. The execution of a Floating Point instruction that causes the FPSCR[FEX] bit to be set to 1 is considered an exception event, regardless of the setting of the MSR[FE0, FE1] bits. If the MSR[FE0, FE1] bits are both 0, then the Floating Point Enabled exception type of Program interrupt is masked, but the exception persists in the FPSCR[FEX] bit. Later, if the MSR[FE0, FE1] bits are enabled, the interrupt will finally be generated.

## Architectural Note

As is the case with this example, when an otherwise synchronous, *precise* class of interrupt is “delayed” in this fashion via masking, and the interrupt type is later enabled, the interrupt which is then generated due to the exception event that occurred while the interrupt type was disabled is then considered a synchronous, *imprecise* class of interrupt. However, this particular category of synchronous, imprecise interrupt is not generally discussed in other sections of this document. Rather, the discussion of synchronous, imprecise interrupts is generally limited to those specific interrupt types which are defined to be imprecise to begin with, and not those which are delayed versions of otherwise synchronous, precise interrupts.

In this fashion, the architecture enables implementations to avoid situations in which an interrupt would cause the state information (saved in Save/Restore Registers) from a previous interrupt to be overwritten and lost. As a first step, upon any non-critical class interrupt, hardware automatically disables any further asynchronous non-critical interrupts (External Input, Programmable Interval Timer, Fixed Interval Timer) by clearing MSR[EE]. Likewise, upon any critical class interrupt, hardware automatically disables any further asynchronous interrupts of either class by clearing MSR[CE, DE] in addition to MSR[EE]. The additional interrupt types that are disabled by the clearing of MSR[CE, DE] are the Critical Input, Watchdog Timer, and Debug interrupts.

This first step of clearing MSR[EE] (and (MSR[CE, DE] for critical class interrupts) prevents any subsequent asynchronous interrupts from overwriting the Save/Restore Registers, prior to software being able to save their contents. Hardware also automatically clears, on any interrupt, MSR[WE, PR, FP, FE0, FE1, APA, APE, IR, DR]. The clearing of these bits assists in the avoidance of subsequent interrupts of certain other types. However, *guaranteeing* that these interrupt types do not occur and thus do not overwrite the Save/Restore Registers also requires the cooperation of system software. Specifically, system software must avoid the execution of instructions which could cause (or enable) a subsequent interrupt, if the Save/Restore Registers’ contents have not yet been saved.

### 7.6.1 Guidelines for System Software

The following list identifies the actions that system software must avoid, prior to having saved the Save/Restore Registers’ contents:

1. Re-enabling of MSR[EE] (or MSR[CE, DE] in critical class interrupt handlers)  
This prevents any asynchronous interrupts, as well as (in the case of MSR[DE]) any Debug interrupts (which include both synchronous and asynchronous types).
2. Re-enabling of MSR[IR]  
This prevents Instruction Storage and Instruction TLB Miss interrupts.
3. Re-enabling of MSR[DR]  
This prevents Data Storage and Data TLB Miss interrupts.

#### 4. Execution of System Call (**sc**) or Trap (**tw**, **twi**) instructions

This prevents System Call and Trap exception type Program interrupts.

#### 5. Execution of *any* Floating Point instructions

This prevents Floating Point Unavailable interrupts. Note that this interrupt would occur upon the execution of any Floating Point instruction, due to the automatic clearing of MSR[FP]. However, even if software were to re-enable MSR[FP], Floating Point instructions must still be avoided in order to prevent Program interrupts due to various possible exceptions (Floating Point Enabled, Floating Point Enabled but Unimplemented).

#### 6. Re-enabling of MSR[PR]

This prevents Privileged Instruction exception type Program interrupts. Alternatively, software could re-enable MSR[PR], but avoid the execution of any privileged instructions.

#### 7. Execution of *any* Auxiliary Processor instructions

This prevents Auxiliary Processor Unavailable exception type Program interrupts. Note that this interrupt would occur upon the execution of any Auxiliary Processor instruction, due to the automatic clearing of MSR[APA]. However, even if software were to re-enable MSR[APA], Auxiliary Processor instructions must still be avoided in order to prevent Program interrupts due to various possible exceptions (Illegal Instruction, Auxiliary Processor Enabled).

#### 8. Execution of any Illegal instructions

This prevents Illegal Instruction exception type Program interrupts.

#### 9. Execution of any instruction which could cause an Alignment interrupt

This prevents Alignment interrupts. Included in this category are any string or multiple instructions, and any unaligned elementary load or store instructions. See Section 7.9.6, “Alignment Interrupt,” on page 7-32, for a complete list of instructions that may cause Alignment interrupts.

Machine Check interrupts are a special case. Machine Checks are critical class interrupts, but *normal* critical class interrupts (Critical Input, Watchdog Timer, Debug) do not automatically disable Machine Checks. Machine Checks are disabled by clearing the MSR[ME] bit, and only a Machine Check interrupt itself automatically clears this bit. Thus there is always the risk that a Machine Check interrupt could occur within a *normal* interrupt handler, prior to the Save/Restore Registers' contents having been saved. In such a case, the interrupt may not be recoverable.

It is not necessary for hardware or software to avoid critical class interrupts from within non-critical class interrupt handlers (and hence hardware does not automatically clear MSR[CE, DE] upon a non-critical interrupt), since the two classes of interrupts use different pairs of Save/Restore Registers to save the program counter and MSR (SRR0/SRR1 for non-critical, and SRR2/SRR3 for critical). The converse, however, is not true. That is, hardware and software must cooperate in the avoidance of both critical *and* non-critical class interrupts from within critical class interrupt handlers, even though the two classes of

interrupts use different Save/Restore Register pairs. This is because the critical class interrupt may have occurred from within a non-critical handler, prior to the non-critical handler having saved the non-critical pair of Save/Restore Registers. Therefore, within the critical class interrupt handler, both pairs of Save/Restore Registers may contain data that is necessary to the system software.

## 7.6.2 Interrupt Order

The following is a prioritized listing of the various enabled interrupt types for which exceptions might exist simultaneously:

### 1. Synchronous (Non Debug) Interrupts:

- Data Storage
- Instruction Storage
- Alignment
- Program
- Floating Point Unavailable
- System Call
- Data TLB Miss
- Instruction TLB Miss

Only one of the above types of synchronous interrupts may have an existing exception generating it at any given time. This is guaranteed by the exception priority mechanism (see Section 7.7, “Exception Priorities,” on page 7-15) and the requirements of the Sequential Execution Model.

- 2. Machine Check
- 3. Debug
- 4. Critical Input
- 5. Watchdog Timer
- 6. External Input
- 7. Fixed Interval Timer
- 8. Programmable Interval Timer

Even though, as indicated above, the non-critical synchronous exception types listed under item 1 are generated with higher priority than the critical interrupt types listed in items 2-5, the fact is that these non-critical interrupts will immediately be followed by the highest priority existing critical interrupt type, without executing any instructions at the non-critical interrupt handler. This is because the non-critical interrupt types do not automatically disable the MSR mask bits for the critical interrupt types. In all other cases, a particular



interrupt type from the above list will automatically disable any subsequent interrupts of the same type, as well as all other interrupt types that are listed below it in the priority order.

## 7.7 Exception Priorities

The IBM PowerPC Embedded Environment requires all synchronous (precise and imprecise) interrupts to be reported in program order, as required by the Sequential Execution Model. The one exception to this rule is the case of multiple synchronous imprecise interrupts. Upon a synchronizing event, all previously executed instructions are required to report any synchronous imprecise interrupt-generating exceptions, and the interrupt will then be generated with all of those exception types reported cumulatively, in both the Exception Syndrome Register (ESR), and any status registers associated with the particular exception type (e.g. the FPSCR).

For any single instruction attempting to cause multiple exceptions for which the corresponding synchronous interrupt types are enabled, this section defines the priority order by which the instruction will be permitted to cause a *single* exception, thus generating a particular synchronous interrupt. Note that it is this exception priority mechanism, along with the requirement that synchronous interrupts be generated in program order, that guarantees that at any given time, there exists for consideration only one of the synchronous interrupt types listed in item 1 of Section 7.6.2, “Interrupt Order,” on page 7-14. The exception priority mechanism also prevents certain debug exceptions from existing in combination with certain other synchronous interrupt-generating exceptions (for example, the Debug Instruction Address Compare (IAC) exception cannot exist simultaneously with any of the other non-debug synchronous interrupt-generating exceptions).

This section does not define the permitted setting of multiple exceptions for which the corresponding interrupt types are disabled. The generation of exceptions for which the corresponding interrupt types are disabled will have no effect on the generation of other exceptions for which the corresponding interrupt types are enabled. Conversely, if a particular exception for which the corresponding interrupt type is enabled is shown in the following sections to be of a higher priority than another exception, it will prevent the setting of that other exception, independent of whether that other exception’s corresponding interrupt type is enabled or disabled.

Except as specifically noted, only one of the exception types listed for a given instruction type will be permitted to be generated at any given time. The priority of the exception types are listed in the following sections ranging from highest to lowest, within each instruction type.

### 7.7.1 Integer Loads and Stores, Cache Management Instructions

1. Debug (IAC)
2. Program (Privileged Instruction) (**dcbi** and **icbt** only)
3. The two items listed below are mutually exclusive, and of equal priority

Data TLB Miss

Data Storage (all types)

4. Alignment
5. Debug (DAC)
6. Debug (ICMP)

If the instruction is causing both a Debug (IAC) and a Debug (DAC), and is not causing any of the exceptions listed in items 2-4, it is permissible for both exceptions to be generated and recorded in the DBSR. A single Debug interrupt will result.

### 7.7.2 Floating Point Loads and Stores

1. Debug (IAC)
2. Floating point Unavailable
3. Program (Floating point Unimplemented)
4. The two items listed below are mutually exclusive and of equal priority

Data TLB Miss

Data Storage (all types)

5. Alignment
6. Debug (DAC)
7. Debug (ICMP)

If the instruction is causing both a Debug (IAC) and a Debug (DAC), and is not causing any of the exceptions listed in items 2-5, it is permissible for both exceptions to be generated and recorded in the DBSR. A single Debug interrupt will result.

### 7.7.3 Floating Point (other)

1. Debug (IAC)
2. Floating point Unavailable
3. The two items listed below are mutually exclusive, and of equal priority.

Program (Floating point Enabled but Unimplemented)

Program (Floating point Enabled)

4. Debug (ICMP)

### **7.7.4 Auxiliary Processor Loads and Stores**

1. Debug (IAC)
2. Program (Auxiliary Processor Unavailable)
3. Program (Privileged Instruction)
4. The two items listed below are mutually exclusive and of equal priority

Data TLB Miss

Data Storage (all types)

5. Alignment
6. Debug (DAC)
7. Debug (ICMP)

If the instruction is causing both a Debug (IAC) and a Debug (DAC), and is not causing any of the exceptions listed in items 2-5, it is permissible for both exceptions to be generated and recorded in the DBSR. A single Debug interrupt will result.

### **7.7.5 Auxiliary Processor (other)**

1. Debug (IAC)
2. Program (Auxiliary Processor Unavailable)
3. Program (Privileged Instruction)
4. Program (Auxiliary Processor Enabled)
5. Debug (ICMP)

### **7.7.6 Illegal Instructions**

1. Debug (IAC)
2. Program (Auxiliary Processor Unavailable)
3. Program (Illegal Instruction)
4. Debug (ICMP)

### 7.7.7 Privileged Instructions

Except **dcbi**, **icbt** and Auxiliary Processor Instructions

1. Debug (IAC)
2. Program (Privileged Instruction)
3. Debug (ICMP)

See Section 7.7.1, Section 7.7.4 and Section 7.7.5.

For **mtmsr**, **rfi**, **rfci**, **mtspr (dbcr)**, **mtspr (dbsr)**, **mtspr (tcr)** and **mtspr (tsr)**, if they are not causing Debug (IAC) nor Program (Privileged Instruction) exceptions, it is possible that they are simultaneously enabling (via mask bits) multiple existing exceptions (and at the same time possibly causing a Debug (ICMP) exception). When this occurs, the interrupts will be handled in the order defined by Section 7.6.2, "Interrupt Order," on page 7-14.

### 7.7.8 Trap Instructions

1. Debug (IAC)
2. Debug (TR)
3. Program (Trap)
4. Debug (ICMP)

If the instruction is causing both a Debug (IAC) and a Debug (TR) it is permissible for both exceptions to be generated and recorded in the DBSR. A single Debug interrupt will result.

### 7.7.9 System Call Instruction

1. Debug (IAC)
2. The two items listed below are *not* mutually exclusive, and may be set simultaneously, in which case both interrupts will be handled in order, as specified in Section 7.6.2, "Interrupt Order," on page 7-14.

Program (System Call)

Debug (ICMP)

### 7.7.10 Branch Instructions

1. Debug (IAC)
2. Debug (BT)
3. Debug (ICMP)

If the instruction is causing both a Debug (IAC) and a Debug (BT) it is permissible for both exceptions to be generated and recorded in the DBSR. A single Debug interrupt will result.

# 7.8 Exception Handling Registers

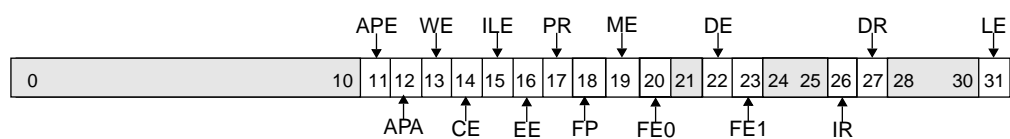
The PowerPC Architecture provides the following registers to handle exceptions and interrupts:

- Machine State Register (MSR)
- Save/Restore Registers (SRR0/SRR1/SRR2/SRR3)
- Data Exception Address Register (DEAR)
- Exception Syndrome Register (ESR)
- Exception Vector Prefix Register (EVPR).

## 7.8.1 Machine State Register (MSR)

The Machine State Register (MSR) is a 32-bit register which controls important chip functions such as the enabling/disabling of interrupts and debugging exceptions. Power management is possible through software control of the Wait State Enable bit within the MSR. The MSR contents are automatically saved, altered, and restored by the interrupt-handling mechanism, as described in Section 7.3, “Interrupt Processing,” on page 7-4.

If a non-critical interrupt is taken, the contents of the MSR are automatically copied into Save/Restore Register 1 (SRR1). If a critical interrupt is taken, the contents of the MSR are automatically copied into Save/Restore Register 3 (SRR3). When a return from interrupt (**rfi**) or return from critical interrupt (**rfci**) instruction is executed, the contents of the MSR are restored from SRR1 or SRR3, respectively. Figure 7-1 illustrates the various bits in the MSR and provides the definition of their function.



**Figure 7-1. Machine State Register (MSR)**

0:10		reserved
11	APE	Auxiliary Processor Exception Enable 0 Auxiliary Processor Enabled Exception Program Interrupts are disabled. 1 Auxiliary Processor Enabled Exception Program Interrupts are enabled.
12	APA	Auxiliary Processor Available 0 The processor cannot execute any Auxiliary Processor instructions. 1 The processor can execute Auxiliary Processor instructions.

**Figure 7-1. Machine State Register (MSR) (cont.)**

13	WE	Wait State Enable 0 The processor is not in the wait state and continues processing. 1 The processor enters the wait state by ceasing to execute instructions and entering low power mode. The details of how the wait state is entered and exited, and how the processor behaves while in the wait state, are implementation dependent. See the User's Manual for the implementation for complete details.	
14	CE	Critical Enable 0 Critical Class interrupts are disabled. 1 Critical Class interrupts are enabled.	CE controls the following interrupts: Critical Input, Watchdog Timer.
15	ILE	Interrupt Little Endian 0 Interrupt handlers execute in Big-Endian mode. 1 Interrupt handlers execute in Little-Endian mode.	MSR(ILE) is copied to MSR(LE) when an interrupt is taken.
16	EE	External Enable 0 External Input interrupts are disabled. 1 External Input interrupts are enabled.	EE controls the following interrupts: External Input, Programmable Interval Timer, Fixed Interval Timer.
17	PR	Problem State 0 Supervisor State — all instructions allowed. 1 Problem State — privileged instructions disallowed.	PR also affects storage protection, as described in Chapter 6, "Storage Control," on p. 6-1.
18	FP	Floating Point Available 0 The processor cannot execute Floating Point instructions 1 The processor can execute Floating Point instructions.	
19	ME	Machine Check Enable 0 Machine Check interrupts are disabled 1 Machine Check interrupts are enabled	
20	FE0	Floating Point Exception Mode 0 FE0, FE1=0b00 - Floating Point Enabled exception type Program interrupts are disabled FE0, FE1=0b01 - Floating Point Enabled exception type Program interrupts are in imprecise non-recoverable mode FE0, FE1=0b10 - Floating Point Enabled exception type Program interrupts are in imprecise recoverable mode FE0, FE1=0b11 - Floating Point Enabled exception type Program interrupts are in precise mode	
21		reserved	
22	DE	Debug Interrupts Enable 0 Debug interrupts are disabled 1 Debug interrupts are enabled	

**Figure 7-1. Machine State Register (MSR) (cont.)**

23	FE1	Floating Point Exception Mode 1 FE0, FE1=0b00 - Floating Point Enabled exception type Program interrupts are disabled FE0, FE1=0b01 - Floating Point Enabled exception type Program interrupts are in imprecise non-recoverable mode FE0, FE1=0b10 - Floating Point Enabled exception type Program interrupts are in imprecise recoverable mode FE0, FE1=0b11 - Floating Point Enabled exception type Program interrupts are in precise mode	
24:25		reserved	
26	IR	Instruction Relocate 0 Instruction Relocation disabled 1 Instruction Relocation enabled	This bit has no function on implementations without the optional memory management support.
27	DR	Data Relocate 0 Data Relocation disabled 1 Data Relocation enabled	This bit has no function on implementations without the optional memory management support.
28:30		reserved	
31	LE	Little Endian 0 Processor executes in Big-Endian mode. 1 Processor executes in Little-Endian mode.	This bit is replaced by the value in MSR[ILE] upon an interrupt.

The contents of the MSR can be read into a general purpose register via move from machine state register (**mfmsr**) instruction. The contents of a general purpose register can be written to the MSR via move to machine state register (**mtmsr**) instruction. The MSR(EE) bit (External Enable) may be set/cleared atomically using the **wrtee** or **wrteei** instructions.

### Programming Note

An MSR bit that is reserved may be altered by **rfi/rfci**.

## 7.8.2 Save Restore Registers

Save/Restore Registers come in two pairs: Save/Restore Registers 0 and 1 (SRR0/SRR1) for non-critical interrupts and Save/Restore Registers 2 and 3(SRR2/SRR3) for critical interrupts.

### 7.8.2.1 Save/Restore Registers 0 and 1 (SRR0 - SRR1)

Save/Restore Register 0 (SRR0) and Save/Restore Register 1 (SRR1) are two 32-bit registers which hold the interrupted context of the machine when a non-critical interrupt is processed. SRR0 is loaded with an address that depends upon the interrupt type (see

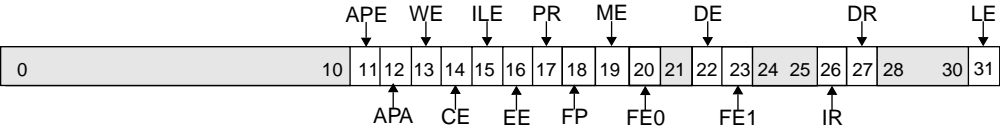
Section 7.9, “Interrupt Definitions,” on page 7-27). SRR1 is loaded with the contents of the MSR. At the end of the exception processing routine, executing a return from interrupt (**rfi**) instruction restores the program counter and the machine state register from SRR0 and SRR1, respectively. Figure 7-2 shows the bit definitions for SRR0.



**Figure 7-2. Save/Restore Register 0 (SRR0)**

0:29	SRR0 is loaded with an instruction address when a non-critical interrupt is taken; the Program Counter is restored from SRR0 when <b>rfi</b> is executed.
30:31	Reserved

Figure 7-3 below and Figure 7-1, on page 7-19 show the bit definitions for SRR1



**Figure 7-3. Save/Restore Register 1 (SRR1)**

0:31	SRR1 receives a copy of the MSR when a non-critical interrupt is taken; the MSR is restored from SRR1 when <b>rfi</b> is executed.
------	--

The contents of the SRR0 and SRR1 can be read into a general purpose register via move from special purpose register (**mfspr**) instruction. The contents of a general purpose register can be written to the SRR0 and SRR1 via move to special purpose register (**mtspr**) instruction.

When a non-critical interrupt occurs, defined bits of the MSR are saved in the corresponding bit positions in SRR1. Bits of SRR1 that correspond to reserved bits in the MSR are also reserved.

**Programming Note**

An MSR bit that is reserved may be altered by **rfi/rfci**.



### 7.8.2.2 Save/Restore Registers 2 and 3 (SRR2 - SRR3)

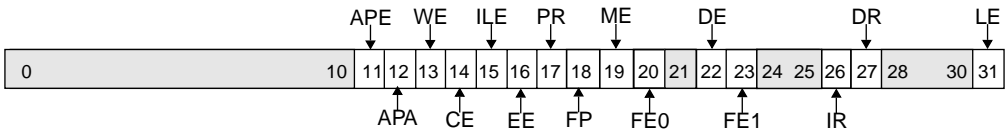
Save/Restore Register 2 (SRR2) and Save/Restore Register 3 (SRR3) are two 32-bit registers which hold the interrupted context of the machine when a critical interrupt occurs. SRR2 is loaded with an address that depends upon the interrupt type (see Section 7.9, “Interrupt Definitions,” on page 7-27). SRR3 is loaded with the contents of the MSR. At the end of the exception processing routine, executing a return from critical interrupt (**rfci**) instruction restores the program counter and the machine state register from SRR2 and SRR3, respectively. Figure 7-4 shows the bit definitions for SRR2.



**Figure 7-4. Save/Restore Register 2 (SRR2)**

0:29		SRR2 is loaded with an instruction address when a critical interrupt is taken; the Program Counter is restored from SRR2 when <b>rfci</b> is executed.
30:31		Reserved

Figure 7-5 below and Figure 7-1, on page 7-19 show the bit definitions for SRR3.



**Figure 7-5. Save/Restore Register 3 (SRR3)**

0:31	SRR3 receives a copy of the MSR when a critical interrupt is taken; the MSR is restored from SRR3 when <b>rfci</b> is executed.
------	---

The contents of the SRR2 and SRR3 can be read into a general purpose register via move from special purpose register (**mf spr**) instruction. The contents of a general purpose register can be written to the SRR2 and SRR3 via move to special purpose register (**mt spr**) instruction.

When a critical interrupt occurs, defined bits of the MSR are saved in the corresponding bit positions in SRR3. Bits of SRR3 that correspond to reserved bits in the MSR are also reserved.

**Programming Note**

An MSR bit that is reserved may be altered by **rfi/rfci**.

**7.8.3 Data Exception Address Register (DEAR)**

The Data Exception Address Register (DEAR) is a 32-bit register which contains the address which was referenced by a load, store or cache management instruction which caused one of the following interrupts:

- alignment,
- data TLB miss,
- or data storage.

Figure 7-6 shows the bit definitions for the DEAR.



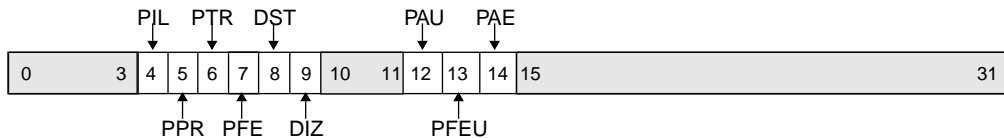
**Figure 7-6. Data Exception Address Register (DEAR)**

0:31	Address of Data Exception
------	---------------------------

The contents of the DEAR can be read into a general purpose register via move from special purpose register (**mfsprr**) instruction. The contents of a general purpose register can be written to the DEAR via move to special purpose register (**mtsprr**) instruction.

**7.8.4 Exception Syndrome Register (ESR)**

The ESR is a 32-bit register that provides a *syndrome* to differentiate between the different kinds of exceptions that can generate the same interrupt type, including program, data storage, data TLB miss, and instruction storage interrupts. Upon the generation of one of these types of interrupts, the bit corresponding to the specific exception that generated the interrupt is set, and the other bits are cleared. Other interrupt types do not affect the contents of the ESR. The ESR does not need to be cleared by software. Figure 7-7 shows the bit definitions for the ESR.



**Figure 7-7. Exception Syndrome Register (ESR)**

0:3		Reserved.
4	PIL	Program - Illegal Instruction exception 0 Illegal Instruction exception type Program interrupt did not occur. 1 Illegal Instruction exception type Program interrupt occurred.
5	PPR	Program - Privileged Instruction exception 0 Privileged Instruction exception type Program interrupt did not occur. 1 Privileged Instruction exception type Program interrupt occurred.
6	PTR	Program - Trap exception 0 Trap exception type Program interrupt did not occur. 1 Trap exception type Program interrupt occurred.
7	PFE	Program - Floating Point Enabled exception 0 Floating Point Enabled exception type Program interrupt did not occur. 1 Floating Point Enabled exception type Program interrupt occurred.
8	DST	Data Storage / Data TLB Miss - Store Operations 0 Excepting instruction was not a store. 1 Excepting instruction was a store (includes <b>dcbz</b> and <b>dcbi</b> ).
9	DIZ	Data / Instruction Storage - Zone exception 0 Zone exception type Data Storage interrupt or Instruction Storage interrupt did not occur. 1 Zone exception type Data Storage interrupt or Instruction Storage interrupt occurred.
10:11		Reserved
12	PAU	Program - Auxiliary Processor Unavailable exception 0 Auxiliary Processor Unavailable exception type Program interrupt did not occur. 1 Auxiliary Processor Unavailable exception type Program interrupt occurred.
13	PFEU	Program - Floating Point Enabled but Unimplemented exception 0 Floating Point Enabled but Unimplemented exception type Program interrupt did not occur. 1 Floating Point Enabled but Unimplemented exception type Program interrupt occurred.
14	PAE	Program - Auxiliary Processor Enabled exception 0 Auxiliary Processor Enabled exception type Program interrupt did not occur. 1 Auxiliary Processor Enabled exception type Program interrupt occurred.
15:31		reserved

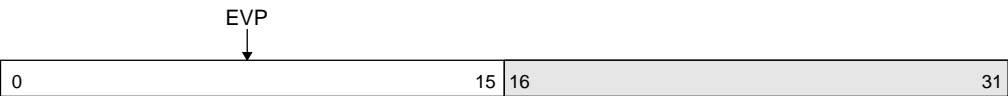
The contents of the ESR can be written into a general purpose register via move from special purpose register (**mfsprr**) instruction. The contents of a general purpose register can be written to the ESR via move to special purpose register (**mtsprr**) instruction.

**Engineering Note**

An implementation may choose to implement additional ESR bits to identify implementation specific exception types. Also, some implementations may choose to use the ESR to record information about the cause of a machine check interrupt.

**7.8.5 Exception Vector Prefix Register (EVPR)**

The EVPR is a 32-bit register that contains the high-order 16 bits that are used as the prefix in the generation of the address of the exception processing routines. The 16-bit exception vector offsets (shown in Table 7-1, “Interrupt and Exception Types,” on page 7-6) are concatenated to the right of the high-order 16-bits of the EVPR to form the 32-bit address of the exception processing routine. Figure 7-8 shows the bit definitions for the EVPR.



**Figure 7-8. Exception Vector Prefix Register (EVPR)**

0:15	EVP	Exception Vector Prefix
16:31		Reserved

The contents of the EVPR can be read into a general purpose register via move from special purpose register (**mfsprr**) instruction. The contents of a general purpose register can be written to the EVPR via move to special purpose register (**mtsprr**) instruction.

# 7.9 Interrupt Definitions

This section describes in detail all types of interrupts and the exception types which cause a particular interrupt. For a summary table of the interrupts and exceptions defined in this section, see Section 7.4, “Interrupt and Exception Types,” on page 7-6.

Implementations may define certain additional, implementation-specific interrupt types and/or exception types (see the User’s Manual for the implementation).

## 7.9.1 Critical Input Interrupt

A Critical Input interrupt occurs when no higher priority exception exists, a Critical Input exception is presented to the interrupt mechanism, and MSR[CE]=1. While the specific definition of a Critical Input exception is implementation dependent, it would typically be caused by the activation of an asynchronous signal that is part of the processing system. Also, implementations may provide an alternative means (in addition to MSR[CE]) for masking the Critical Input interrupt (see the User’s Manual for the implementation for details).

**Note:** MSR[CE] also enables the Watchdog Timer (WDT) interrupt.

When a Critical Input interrupt occurs, the address of the next instruction to be executed is saved in SRR2. The current contents of the MSR are saved in SRR3 and the MSR is updated as shown in Table 7-2. The high-order 16 bits of the program counter are then loaded with the contents of the EVPR and the low-order 16 bits of the program counter are loaded with 0x0100. Exception processing begins at the address in the program counter.

Executing the return from critical interrupt (**rfci**) instruction restores the contents of the program counter and the MSR from SRR2 and SRR3, respectively, and the processor resumes execution at the address contained in the program counter. Software is responsible for taking whatever action(s) required by the implementation in order to clear any Critical Input exception status prior to re-enabling MSR[CE] in order to avoid another redundant Critical Input interrupt.

**Table 7-2. Register Settings during Critical Input Interrupts**

SRR2	Set to the effective address of the next instruction to be executed	
SRR3	Set to the value of the MSR at the time of the interrupt	
MSR	APE, APA, WE, CE, EE, PR, FP, FE0, DE, FE1, IR, DR← 0 LE ← ILE ILE, ME ← unchanged	
PC	EVPR[0:15]    0x0100	

## 7.9.2 Machine Check Interrupt

A Machine Check interrupt occurs when no higher priority exception exists, a Machine Check exception is presented to the interrupt mechanism, and MSR[ME]=1. The specific cause or causes of Machine Check exceptions are implementation dependent, as are the details of the actions taken on a Machine Check interrupt (see the User's Manual for the implementation for details).

When a Machine Check interrupt occurs, SRR2 is set on a 'best effort' basis to the effective address of an instruction that was executing or about to be executed when the Machine Check exception occurred. The current contents of the MSR are saved in SRR3, and the MSR is updated as shown in Table 7-3. The high-order 16 bits of the program counter are then loaded with the contents of the EVPR and the low-order 16 bits of the program counter are loaded with 0x0200. Exception processing begins at the new address in the program counter.

Executing the return from critical interrupt (**rfci**) instruction restores the contents of the program counter and the MSR from SRR2 and SRR3, respectively, and the processor resumes execution at the address contained in the program counter.

**Table 7-3. Register Settings during Machine Check Interrupts**

SRR2	Set to an instruction address; see explanation above
SRR3	Set to the value of the MSR at the time of the interrupt
MSR	APE, APA, WE, CE, EE, PR, FP, ME, FE0, DE, FE1, IR, DR, $\leftarrow 0$ LE $\leftarrow$ ILE ILE $\leftarrow$ unchanged
PC	EVPR[0:15]    0x0200
ESR	Implementation dependent (see User's Manual for the implementation).

## 7.9.3 Data Storage Interrupt

A Data Storage interrupt occurs when no higher priority exception exists and a Data Storage exception is presented to the interrupt mechanism. A Data Storage exception is caused when a data storage access cannot be performed for any of the following reasons:

- In Problem State (MSR[PR]=1) with data translation enabled (MSR[DR]=1):
  - A *zone fault*, which is any user-mode data storage access (load, store, **icbi**, **dcbz**, **dcbst**, or **dcbf**) with an effective address for which the ZSEL field supplied by the matching TLB entry selects a ZP value of 0b00 from the ZPR. **dcbt** and **dcbst** will no-op in this situation, rather than generate an interrupt. The instructions **dcbi** and **icbt**, being privileged, cannot cause Zone Fault Data Storage exceptions.

- Store or **dcbz** to an effective address for which the matching TLB entry supplies a WR value of 0b0 and a ZSEL field that selects a ZP value other than 0b11 from the ZPR. The privileged instruction **dcbi** is treated as a “store”, but will cause a Privileged Instruction exception rather than a Data Storage exception.)
- In Supervisor State (MSR[PR]=0) with data translation enabled (MSR[DR=1]):
  - Data store, **dcbi**, or **dcbz** to an effective address for which the matching TLB entry supplies a WR value of 0b0 and a ZSEL field that selects a ZP value other than 0b11 or 0b10 from the ZPR.
- In either Problem State or Supervisor State:
  - **lwarx** or **stwcx**. to a location that is Write Through Required (if the interrupt does not occur then the instruction executes correctly: see Section 2.7.2, “Atomic Update Primitives,” on page 2-15).

If a **stwcx**. has an effective address for which a normal *Store* would cause a Data Storage interrupt, but the processor does not have the reservation from **lwarx**, then it is implementation-dependent whether a Data Storage interrupt occurs.

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

### Programming Note

The **icbi** and **icbt** instructions are treated as loads from the addressed byte with respect to address translation and protection. These instruction cache ops use MSR[DR], not MSR[IR], to determine translation of their operands. Instruction Storage exceptions and Instruction-side TLB Miss exceptions are associated with the *fetching* of instructions, not with the *execution* of instructions. Data Storage exceptions and Data TLB Miss exceptions are associated with the execution of instruction cache ops.

When a Data Storage interrupt occurs, the processor suppresses the execution of the instruction causing the Data Storage exception and saves the instruction address in SRR0. The Data Exception Address Register (DEAR) is loaded with the data address that caused the Data Storage exception. The Exception Syndrome Register (ESR) is set as shown in Table 7-4 to provide further information about the exception. The current contents of the MSR are saved in SRR1, and the MSR is updated as shown in Table 7-4. The high-order 16 bits of the program counter are then loaded with the contents of the EVPR and the low-order 16 bits of the program counter are loaded with 0x0300. Exception processing begins at the new address in the program counter.

Executing the return from interrupt (**rfi**) instruction restores the contents of the program counter and the MSR from SRR0 and SRR1, respectively, and the processor resumes execution at the address contained in the program counter.

**Table 7-4. Register Settings during Data Storage Interrupts**

SRR0	Set to the effective address of the instruction causing the Data Storage interrupt
SRR1	Set to the value of the MSR at the time of the interrupt
MSR	APE, APA, WE, EE, PR, FP, FE0, FE1, IR, DR $\leftarrow$ 0 LE $\leftarrow$ ILE CE, ILE, ME, DE $\leftarrow$ unchanged
PC	EVPR[0:15]    0x0300
DEAR	Set to the effective address of the failed access
ESR	DST $\leftarrow$ 1 If excepting operation is a store operation (includes <b>dcbi</b> and <b>dcbz</b> ). DIZ $\leftarrow$ 1 If access failure due to a zone protection fault (ZPR field value of 00 in Problem State)

### 7.9.4 Instruction Storage Interrupt

An Instruction Storage interrupt occurs when no higher priority exception exists and an Instruction Storage exception is presented to the interrupt mechanism. An Instruction Storage exception is caused when an instruction fetch cannot be performed for any of the following reasons:

- In Problem State (MSR[PR]=1) with instruction translation enabled (MSR[IR]=1):
  - A *zone fault*, which is a user-mode instruction fetch with an effective address for which the ZSEL field supplied by the matching TLB entry selects a ZP value of 0b00 from the ZPR.
  - Instruction fetch from an effective address for which the matching TLB entry supplies an EX value of 0b0 and a ZSEL field that selects a ZP value other than 0b11 from the ZPR.
  - Instruction fetch from an effective address which is marked as Guarded (G=1) in the matching TLB entry.
- In Supervisor State (MSR[PR]=0) with instruction translation enabled (MSR[IR]=1):
  - Instruction fetch from an effective address for which the matching TLB entry supplies an EX value of 0b0 and a ZSEL field that selects a ZP value other than 0b11 or 0b10 from the ZPR.
  - Instruction fetch from an effective address which is marked as Guarded (G=1) in the matching TLB entry.

When an Instruction Storage interrupt occurs, the processor suppresses the execution of the instruction causing the Instruction Storage exception and saves the instruction address in SRR0. The Exception Syndrome Register (ESR) is set as shown in Table 7-5 to provide further information about the exception. The current contents of the MSR are saved in



SRR1, and the MSR is updated as shown in Table 7-5. The high-order 16 bits of the program counter are then loaded with the contents of the EVPR and the low-order 16 bits of the program counter are loaded with 0x0400. Exception processing begins at the new address in the program counter.

Executing the return from interrupt (**rfi**) instruction restores the contents of the program counter and the MSR from SRR0 and SRR1, respectively, and the processor resumes execution at the address contained in the program counter.

**Table 7-5. Register Settings during Instruction Storage Interrupts**

SRR0	Set to the effective address of the instruction causing the Instruction Storage interrupt
SRR1	Set to the value of the MSR at the time of the interrupt
MSR	APE, APA, WE, EE, PR, FP, FE0, FE1, IR, DR $\leftarrow$ 0 LE $\leftarrow$ ILE CE, ILE, ME, DE $\leftarrow$ unchanged
PC	EVPR[0:15]    0x0400
ESR	DIZ $\leftarrow$ 1 If access failure due to a zone protection fault (ZPR field value of 00 in Problem State)

## 7.9.5 External Input Interrupt

An External Input interrupt occurs when no higher priority exception exists, an External Input exception is presented to the interrupt mechanism, and MSR[EE]=1. While the specific definition of an External Input exception is implementation dependent, it would typically be caused by the activation of an asynchronous signal that is part of the processing system. Also, implementations may provide an alternative means (in addition to MSR[EE]) for masking the External Input interrupt (see the User's Manual for the implementation for details).

### Note

MSR[EE] also enables the Programmable Interval Timer (PIT) and the Fixed Interval Timer interrupts.

When an External Input interrupt occurs, the address of the next instruction to be executed is saved in SRR0. The current contents of the MSR are saved in SRR1 and the MSR is updated as shown in Table 7-6. The high-order 16 bits of the program counter are then loaded with the contents of the EVPR and the low-order 16 bits of the program counter are loaded with 0x0500. Exception processing begins at the address in the program counter.

Executing the return from interrupt (**rfi**) instruction restores the contents of the program counter and the MSR from SRR0 and SRR1, respectively, and the processor resumes execution at the address contained in the program counter. Software is responsible for taking whatever action(s) required by the implementation in order to clear any External Input

exception status prior to re-enabling the MSR[EE] bit in order to avoid another redundant External Input interrupt.

**Table 7-6. Register Settings during External Input Interrupts**

SRR0	Set to the effective address of the next instruction to be executed	
SRR1	Set to the value of the MSR at the time of the interrupt	
MSR	APE, APA, WE, EE, PR, FP, FE0, FE1, IR, DR ← 0 LE ← ILE DE, CE, ILE, ME ← unchanged	
PC	EVPR[0:15]    0x0500	

### 7.9.6 Alignment Interrupt

An Alignment interrupt occurs when no higher priority exception exists and an Alignment exception is presented to the interrupt mechanism. An Alignment exception is caused when the implementation cannot perform a data storage access for any of the following reasons:

- Required
  - The instruction is a string or multiple and the processor is in Little Endian mode (MSR[LE]=1).
- Permitted (implementation-dependent)
  - The instruction is a string or multiple and the processor is in Big Endian mode (MSR[LE]=0).
  - The operand of any elementary (non-string or -multiple) load or store, including **lwarx** and **stwcx.**, is not aligned.
  - The operand of **dcbz** is in storage that is Write Through Required or Caching Inhibited, or **dcbz** is executed in an implementation that has either no data cache or a Write Through data cache.
  - The operand of any store instruction is in storage that is Write Through Required.

For multiple and string instructions in Little-Endian mode, an Alignment interrupt always occurs. For multiple instructions in Big-Endian mode, and for **lwarx** or **stwcx.** with an operand that is not aligned in either Endian mode, an implementation may yield boundedly undefined results instead of causing an Alignment interrupt. For all other cases listed above, an implementation may execute the instruction correctly instead of causing an Alignment interrupt. (For **dcbz** to Caching Inhibited or Write Through Required storage, “correct” execution means setting each byte of the block in main storage to 0x00.)

#### Programming note

The architecture does not support the use of an unaligned effective address by **lwarx** or **stwcx.** If an Alignment interrupt occurs because one of these instructions specifies an unaligned effective address, the Alignment interrupt

handler must not attempt to emulate the instruction, but instead should treat the instruction as a programming error.

When an Alignment interrupt occurs, the processor suppresses the execution of the instruction causing the Alignment exception and saves the instruction address in SRR0. The Data Exception Address Register (DEAR) is loaded with the data address that caused the Alignment exception. The current contents of the MSR are saved in SRR1, and the MSR is updated as shown in Table 7-7. The high-order 16 bits of the program counter are then loaded with the contents of the EVPR and the low-order 16 bits of the program counter are loaded with 0x0600. Exception processing begins at the new address in the program counter.

Executing the return from interrupt (**rfi**) instruction restores the contents of the program counter and the MSR from SRR0 and SRR1, respectively, and the processor resumes execution at the address contained in the program counter.

**Table 7-7. Register Settings during Alignment Interrupts**

SRR0	Set to the effective address of the instruction causing the Alignment interrupt
SRR1	Set to the value of the MSR at the time of the interrupt
MSR	APE, APA, WE, EE, PR, FP, FE0, FE1, IR, DR ← 0 LE ← ILE CE, ILE, ME, DE ← unchanged
PC	EVPR[0:15]    0x0600
DEAR	Set to the effective address of the failed access

### 7.9.7 Program Interrupt

A Program interrupt occurs when no higher priority exception exists and a program exception is presented to the interrupt mechanism. A Program exception is caused by any of the events described in Section 7.9.7.1-Section 7.9.7.7.

For all types of exceptions causing a Program interrupt, the current contents of the MSR are saved in SRR1, and the MSR bit is updated as shown in Table 7-8. The high-order 16 bits of the program counter are then loaded with the contents of the EVPR and the low-order 16 bits of the program counter are loaded with 0x0700. Exception processing begins at the new address in the program counter.

Executing the return from interrupt (**rfi**) instruction restores the contents of the program counter and the MSR from SRR0 and SRR1, respectively, and the processor resumes execution at the address contained in the program counter.

**Table 7-8. Register Settings during Program Interrupts**

SRR0	Set to the effective address of the instruction causing the Program interrupt	
SRR1	Set to the value of the MSR at the time of the interrupt	
MSR	APE, APA, WE, EE, PR, FP, FE0, FE1, IR, DR, $\leftarrow$ 0 LE $\leftarrow$ ILE CE, ILE, ME, DE $\leftarrow$ unchanged	
PC	EVPR[0:15]    0x0700	
ESR	Set according to the exception type that is causing the Program interrupt. The ESR bit corresponding to the exception type is set to 1 and all other bits are cleared.	
	Exception Type	ESR Bit
	Illegal Instruction	PIL(4)
	Privileged Instruction	PPR(5)
	Trap	PTR(6)
	FP Enabled	PFE(7)
	FP Enabled Unimplemented	PFEU(13)
	AP Unavailable	PAU(12)
	AP Enabled	PAE(14)

### 7.9.7.1 Illegal Instruction Exception

An Illegal Instruction exception occurs 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 exception *may* occur when execution is attempted of any of the following kinds of instruction. If the exception does not occur, the alternative is shown in parentheses.

- an instruction that is in invalid form (boundedly undefined results)
- an **lswx** instruction for which RA or RB is in the range of registers to be loaded (boundedly undefined results)
- an **mtspr** or **mfspr** instruction with an SPRF value that is not defined:

MSR<sub>PR</sub>=1 and SPRF<sub>0</sub>=1 (Privileged Instruction exception)

MSR<sub>PR</sub>=0 or SPRF<sub>0</sub>=0 (boundedly undefined results)

When an Illegal Instruction exception type Program interrupt occurs, the processor suppresses the execution of the instruction, and saves the instruction address in SRR0. The ESR[PIL] bit is set to “1” to identify the cause of the Program interrupt.

### 7.9.7.2 Privileged Instruction Exception

The following applies when MSR[PR]=1:

A Privileged Instruction exception occurs when execution is attempted of a privileged instruction, or of an **mtspr** or **mfspr** instruction with an SPRF value that is defined and has SPRF<sub>0</sub>=1. It *may* be generated when execution is attempted of an **mtspr** or **mfspr** instruction with an SPRF value that is not defined but has SPRF<sub>0</sub>=1; in this case an Illegal Instruction exception may occur instead.

When a Privileged Instruction exception type Program interrupt occurs, the processor suppresses the execution of the instruction, and saves the instruction address in SRR0. The ESR[PPR] bit is set to 1 to identify the cause of the Program interrupt.

### 7.9.7.3 Trap Exception

A Trap exception occurs when any of the conditions specified in a Trap (**twi**, **tw**) instruction are met, and Trap is either not enabled as a debug event (DBCR[TR]=0) or Debug interrupts are not enabled (MSR[DE]=0 or DBCR[IDM]=0). See Section 9.3, “Debug Events,” on page 9-2.

When a Trap exception type Program interrupt occurs, the processor saves the instruction address in SRR0. The ESR[PTR] bit is set to 1 to identify the cause of the Program interrupt.

### 7.9.7.4 Floating-Point Enabled Exception

A Floating-Point Enabled exception occurs when a floating point instruction is executed and causes an arithmetic exception which is enabled in the FPSCR. Alternatively, the exception can be caused by the execution of a move to FPSCR instruction which results in the FPSCR[FEX] bit being 1. In either case, the associated Program interrupt is disabled if MSR[FE0, FE1]=0b00.

The precision of the Floating Point Enabled exception type Program interrupt is controlled by MSR[FE0, FE1]. If MSR[FE0, FE1]=0b11, the interrupt is precise; otherwise, the interrupt is either imprecise or disabled.

For a Precise Mode (MSR[FE0, FE1]=0b11) Floating Point Enabled exception type Program interrupt, SRR0 is set to the address of the instruction causing the exception.

For an Imprecise Mode (MSR[FE0, FE1]=0b01 or 0b10) Floating-Point Enabled Exception type Program interrupt, SRR0 is set to the effective address of the excepting instruction or to the effective address of some subsequent instruction. If it points to a subsequent instruction, that instruction may have been partially executed (see Section 7.5, “Partially Executed Instructions,” on page 7-10). If a subsequent instruction is Synchronize (**sync**) or Instruction Synchronize (**isync**), SRR0 will not point more than four bytes beyond the **sync** or **isync** instruction.

If FPSCR[FEX]=1 but Floating-Point Enabled exception type Program interrupts are disabled by having MSR[FE0, FE1]=0b00, a Floating-Point Enabled exception type Program interrupt will occur prior to or at the next synchronizing event if these MSR bits are altered by any instruction that can set the MSR so that the expression

$(\text{MSR}[\text{FE0}] \mid \text{MSR}[\text{FE1}]) \ \& \ \text{FPSCR}[\text{FEX}]$

is '1'. When this occurs, SRR0 is set to the address of the instruction that would have executed next, not with the address of the instruction that modified the MSR causing the interrupt.

The ESR[PFE] bit is set to '1' to identify the cause of the Program interrupt.

#### **7.9.7.5 Floating Point Enabled but Unimplemented Exception**

A Floating Point Enabled but Unimplemented exception occurs when execution of floating point instructions is enabled (MSR[FP]=1), and the execution of a defined (non-optional) floating point instruction which the implementation does not provide is attempted.

When a Floating Point Enabled but Unimplemented exception type Program interrupt occurs, the processor suppresses the execution of the instruction and saves the instruction address in SRR0. The ESR[PFEU] bit is set to 1 to identify the cause of the Program interrupt.

#### **7.9.7.6 Auxiliary Processor Unavailable Exception**

An Auxiliary Processor Unavailable exception occurs when MSR[APA]=0 and execution is attempted of an instruction which is within the opcode space defined by the implementation to be reserved for potential use by an Auxiliary Processor (see the User's Manual for the implementation for details). Note that the particular instruction opcode causing the exception may not in fact be implemented by any Auxiliary Processor included as part of the implementation. In such a case, if MSR[APA] is subsequently set to '1' and execution of the instruction re-attempted, an Illegal Instruction exception will result.

When an Auxiliary Processor Unavailable exception type Program interrupt occurs, the processor suppresses the execution of the instruction and saves the instruction address in SRR0. The ESR[PAU] bit is set to '1' to identify the cause of the program interrupt.

#### **7.9.7.7 Auxiliary Processor Enabled Exception**

An Auxiliary Processor Enabled exception occurs when the execution of an Auxiliary Processor Unit (APU) instruction causes the APU to signal an exception to the processor. The associated Program interrupt is disabled if MSR[APE]=0.

The precision of the Auxiliary Processor Enabled exception type Program interrupt is controlled by the APU, and is implementation-dependent (see the User's Manual for the implementation for details).

For a Precise Mode Auxiliary Processor Enabled exception type Program interrupt, SRR0 is set to the address of the instruction causing the exception. For an Imprecise Mode Auxiliary Processor Enabled exception type Program interrupt, SRR0 is set to the effective address of the excepting instruction or to the effective address of some subsequent instruction. If it points to a subsequent instruction, that instruction may have been partially executed (see Section 7.5, “Partially Executed Instructions,” on page 7-10). If a subsequent instruction is Synchronize (**sync**) or Instruction Synchronize (**isync**), SRR0 will not point more than four bytes beyond the **sync** or **isync** instruction.

If the APU is signalling an exception to the processor, but the associated Program interrupt is disabled by having MSR[APE]=0, then an Auxiliary Processor Enabled exception type Program interrupt will occur prior to or at the next synchronizing event if the MSR[APE] bit is set to ‘1’ by any instruction, and the APU is still signalling the exception. When this occurs, SRR0 is set to the address of the instruction that would have executed next, not with the address of the instruction that modified the MSR causing the interrupt.

The ESR[PAE] bit is set to ‘1’ to identify the cause of the Program interrupt.

## 7.9.8 Floating-Point Unavailable Interrupt

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

When a Floating-Point Unavailable interrupt occurs, the processor suppresses the execution of the instruction causing the Floating-Point Unavailable exception and saves the instruction address in SRR0. The current contents of the MSR are saved in SRR1 and the MSR is updated as shown in Table 7-9. The high-order 16 bits of the program counter are then loaded with the contents of the EVPR and the low-order 16 bits of the program counter are loaded with 0x0800. Exception processing begins at the address in the program counter.

Executing the return from interrupt (**rfi**) instruction restores the contents of the program counter and the MSR from SRR0 and SRR1, respectively, and the processor resumes execution at the address contained in the program counter.

**Table 7-9. Register Settings during Floating-Point Unavailable Interrupts**

SRR0	Set to the effective address of the instruction causing the Floating-Point Unavailable interrupt
SRR1	Set to the value of the MSR at the time of the interrupt
MSR	APE, APA, WE, EE, PR, FP, FE0, FE1, IR, DR ← 0 LE ← ILE CE, ILE, ME, DE ← unchanged
PC	EVPR[0:15]    0x0800

### 7.9.9 System Call Interrupt

A System Call interrupt occurs when no higher priority exception exists, and a system call (**sc**) instruction is executed.

When a System Call interrupt occurs, the address of the instruction after the **sc** instruction is saved in SRR0. The current contents of the MSR are saved in SRR1 and the MSR is updated as shown in Table 7-10. The high-order 16 bits of the program counter are then loaded with the contents of the EVPR and the low-order 16 bits of the program counter are loaded with 0x0C00. Exception processing begins at the address in the program counter.

Executing the return from interrupt (**rfi**) instruction restores the contents of the program counter and the MSR from SRR0 and SRR1, respectively, and the processor resumes execution at the address contained in the program counter.

**Table 7-10. Register Settings during System Call Interrupts**

SRR0	Set to the effective address of the instruction after the <b>sc</b> instruction
SRR1	Set to the value of the MSR at the time of the interrupt
MSR	APE, APA, WE, EE, PR, FP, FE0, FE1, IR, DR ← 0 LE ← ILE CE, ILE, ME, DE ← unchanged
PC	EVPR[0:15]    0x0C00

### 7.9.10 Programmable Interval Timer Interrupt

A Programmable Interval Timer interrupt occurs when no higher priority exception exists, a Programmable Interval Timer exception exists (TSR[PIS]=1), and the interrupt is enabled (TCR[PIE]=1 and MSR[EE]=1). See Section 10.3, “Programmable Interval Timer (PIT),” on page 10-5.

**Note:** MSR[EE] also enables the External Input and Fixed Interval Timer interrupts.

When a Programmable Interval Timer interrupt occurs, the address of the next instruction to be executed is saved in SRR0. The current contents of the MSR are saved in SRR1 and the MSR is updated as shown in Table 7-11. The high-order 16 bits of the program counter are then loaded with the contents of the EVPR and the low-order 16 bits of the program counter are loaded with 0x1000. Exception processing begins at the address in the program counter.

Executing the return from interrupt (**rfi**) instruction restores the contents of the program counter and the MSR from SRR0 and SRR1, respectively, and the processor resumes execution at the address contained in the program counter. Software is responsible for clearing the Programmable Interval Timer exception status prior to re-enabling the MSR[EE] bit in order to avoid another redundant Programmable Interval Timer interrupt. To clear the Programmable Interval Timer exception, the interrupt handling routine must clear the PIT Interrupt Status bit in the Timer Status Register (TSR[PIS]). Clearing is done by writing a word to TSR using **mtspr** with a ‘1’ in any bit position that is to be cleared and ‘0’ in all other



bit positions. The write-data to the TSR is not direct data, but a mask. A '1' causes the bit to be cleared, and a '0' has no effect.

**Table 7-11. Register Settings during Programmable Interval Timer Interrupts**

SRR0	Set to the effective address of the next instruction to be executed
SRR1	Set to the value of the MSR at the time of the interrupt
MSR	APE, APA, WE, EE, PR, FP, FE0, FE1, IR, DR $\leftarrow$ 0 LE $\leftarrow$ ILE CE, ILE, ME, DE $\leftarrow$ unchanged
PC	EVPR[0:15]    0x1000
TSR	TSR[PIS] $\leftarrow$ 1

### 7.9.11 Fixed Interval Timer Interrupt

A Fixed Interval Timer interrupt occurs when no higher priority exception exists, a Fixed Interval Timer exception exists (TSR[FIS]=1), and the interrupt is enabled (TCR[FIE]=1 and MSR[EE]=1). See Section 10.4, "Fixed Interval Timer (FIT)," on page 10-7.

**Note:** MSR[EE] also enables the External Input and Programmable Interval Timer interrupts.

When a Fixed Interval Timer interrupt occurs, the address of the next instruction to be executed is saved in SRR0. The current contents of the MSR are saved in SRR1 and the MSR is updated as shown in Table 7-12. The high-order 16 bits of the program counter are then loaded with the contents of the EVPR and the low-order 16 bits of the program counter are loaded with 0x1010. Exception processing begins at the address in the program counter.

Executing the return from interrupt (**rfi**) instruction restores the contents of the program counter and the MSR from SRR0 and SRR1, respectively, and the processor resumes execution at the address contained in the program counter. Software is responsible for clearing the Fixed Interval Timer exception status prior to re-enabling the MSR[EE] bit in order to avoid another redundant Fixed Interval Timer interrupt. To clear the Fixed Interval Timer exception, the interrupt handling routine must clear the FIT Interrupt Status bit in the Timer Status Register (TSR[FIS]). Clearing is done by writing a word to TSR using **mtspr** with a '1' in any bit position that is to be cleared and '0' in all other bit positions. The write-data to the TSR is not direct data, but a mask. A '1' causes the bit to be cleared, and a '0' has no effect.

**Table 7-12. Register Settings during Fixed Interval Timer Interrupts**

SRR0	Set to the effective address of the next instruction to be executed
SRR1	Set to the value of the MSR at the time of the interrupt
MSR	APE, APA, WE, EE, PR, FP, FE0, FE1, IR, DR $\leftarrow$ 0 LE $\leftarrow$ ILE CE, ILE, ME, DE $\leftarrow$ unchanged
PC	EVPR[0:15]    0x1010
TSR	TSR[FIS] $\leftarrow$ 1

### 7.9.12 Watchdog Timer Interrupt

A Watchdog Timer interrupt occurs when no higher priority exception exists, a Watchdog Timer exception exists (TSR[WIS]=1), and the interrupt is enabled (TCR[WIE]=1 and MSR[CE]=1). See Section 10.5, “Watch Dog Timer (WDT),” on page 10-8.

**Note:** MSR[CE] also enables the Critical Input interrupt.

When a Watchdog Timer interrupt occurs, the address of the next instruction to be executed is saved in SRR2. The current contents of the MSR are saved in SRR3 and the MSR is updated as shown in Table 7-13. The high-order 16 bits of the program counter are then loaded with the contents of the EVPR and the low-order 16 bits of the program counter are loaded with 0x1020. Exception processing begins at the address in the program counter.

Executing the return from critical interrupt (**rfci**) instruction restores the contents of the program counter and the MSR from SRR2 and SRR3, respectively, and the processor resumes execution at the address contained in the program counter. Software is responsible for clearing the Watchdog Timer exception status prior to re-enabling the MSR[CE] bit in order to avoid another redundant Watchdog Timer interrupt. To clear the Watchdog Timer exception, the interrupt handling routine must clear the WDT interrupt Status bit in the Timer Status Register (TSR[WIS]). Clearing is done by writing a word to TSR using **mtspr** with a ‘1’ in any bit position that is to be cleared and ‘0’ in all other bit positions. The write-data to the TSR is not direct data, but a mask. A ‘1’ causes the bit to be cleared, and a ‘0’ has no effect.

**Table 7-13. Register Settings during Watchdog Interrupts**

SRR2	Set to the effective address of the next instruction to be executed
SRR3	Set to the value of the MSR at the time of the interrupt
MSR	APE, APA, WE, CE, EE, PR, FP, FE0, DE, FE1, IR, DR ← 0 LE ← ILE ILE, ME ← unchanged
PC	EVPR[0:15]    0x1020
TSR	TSR[WIS] ← 1

### 7.9.13 Data TLB Miss Interrupt

A Data TLB Miss interrupt occurs when no higher priority exception exists and a Data TLB Miss exception is presented to the interrupt mechanism. A Data TLB Miss exception is caused when Data Relocation is enabled (MSR[DR]=1) and the address and PID associated with a data storage access does not have a valid matching entry in the TLB.

When a Data TLB Miss interrupt occurs, the processor suppresses the execution of the instruction causing the Data TLB Miss exception and saves the instruction address in SRR0. The Data Exception Address Register (DEAR) is loaded with the data address that caused the Data TLB Miss exception. The Exception Syndrome Register (ESR) is set as shown in Table 7-14 to provide further information about the exception. The current contents of the MSR are saved in SRR1, and the MSR is updated as shown in Table 7-4. The high-order 16 bits of the program counter are then loaded with the contents of the EVPR and the low-order 16 bits of the program counter are loaded with 0x1100. Exception processing begins at the new address in the program counter.

Executing the return from interrupt (**rfi**) instruction restores the contents of the program counter and the MSR from SRR0 and SRR1, respectively, and the processor resumes execution at the address contained in the program counter.

**Table 7-14. Register Settings during Data TLB Miss Interrupts**

SRR0	Set to the effective address of the instruction causing the Data TLB Miss interrupt
SRR1	Set to the value of the MSR at the time of the interrupt
MSR	APE, APA, WE, EE, PR, FP, FE0, FE1, IR, DR ← 0 LE ← ILE CE, ILE, ME, DE ← unchanged
PC	EVPR[0:15]    0x1100
DEAR	Set to the effective address of the failed access
ESR	DST ← 1 If excepting operation is a store operation (includes <b>dcbi</b> and <b>dcbz</b> ).

### 7.9.14 Instruction TLB Miss Interrupt

An Instruction TLB Miss interrupt occurs when no higher priority exception exists and an Instruction TLB Miss exception is presented to the interrupt mechanism. An Instruction TLB Miss exception is caused when Instruction Relocation is enabled (MSR[IR]=1) and the address and PID associated with an instruction fetch does not have a valid matching entry in the TLB.

When an Instruction TLB Miss interrupt occurs, the processor suppresses the execution of the instruction causing the Instruction TLB Miss exception and saves the instruction address in SRR0. The current contents of the MSR are saved in SRR1, and the MSR is updated as shown in Table 7-15. The high-order 16 bits of the program counter are then loaded with the contents of the EVPR and the low-order 16 bits of the program counter are loaded with 0x1200. Exception processing begins at the new address in the program counter.

Executing the return from interrupt (**rfi**) instruction restores the contents of the program counter and the MSR from SRR0 and SRR1, respectively, and the processor resumes execution at the address contained in the program counter.

**Table 7-15. Register Settings during Instruction TLB Miss Interrupts**

SRR0	Set to the effective address of the instruction causing the Instruction TLB Miss interrupt
SRR1	Set to the value of the MSR at the time of the interrupt
MSR	APE, APA, WE, EE, PR, FP, FE0, FE1, IR, DR ← 0 LE ← ILE CE, ILE, ME, DE ← unchanged
PC	EVPR[0:15]    0x1200

### 7.9.15 Debug Interrupt

A Debug interrupt occurs when no higher priority exception exists, a Debug exception exists in the DBSR, and Debug interrupts are enabled (DBCR[IDM]=1 and MSR[DE]=1). A Debug exception occurs when a Debug Event causes a corresponding bit in the DBSR to be set. See Chapter 9, “Debug Facilities,” on p. 9-1.

The instruction address that is saved in SRR2 depends upon the type of Debug exception that is causing the Debug interrupt, and upon whether or not Debug interrupts were enabled at the time of the Debug exception. For Debug exceptions that occur while Debug interrupts are enabled (DBCR[IDM]=1 and MSR[DE]=1), SRR2 is set as follows:

- Debug exception is Instruction Address Compare (IAC), Data Address Compare (DAC), Trap (TR), or Branch Taken (BT).
  - SRR2 is set to the address of the instruction causing the Debug interrupt.
- Debug exception is Instruction Complete (ICMP)
  - SRR2 is set to the address of the instruction which would have executed after the one which caused the Debug interrupt.

- Debug exception is Unconditional Debug Event (UDE)
  - SRR2 is set to the address of the instruction which would have executed next if the Debug interrupt had not occurred.
- Debug exception is Exception Event (EXC)
  - SRR2 is set to the Interrupt vector value of the interrupt which caused the Exception Debug Event.

For Debug exceptions that occur while debug interrupts are disabled (DBCR[IDM]=0 or MSR[DE]=0), a debug interrupt will occur at the next synchronizing event if the DBCR[IDM] and/or the MSR[DE] bit(s) are modified such that they are both '1' and if the Debug exception Status is still set in the DBSR. When this occurs, SRR2 is set to the address of the instruction that would have executed next, not with the address of the instruction that modified the DBCR or MSR and thus caused the interrupt.

For all cases, when a Debug interrupt occurs, the current contents of the MSR are saved in SRR3, and the MSR is updated as shown in Table 7-16. The high order 16 bits of the program counter are then loaded with the contents of the EVPR and the low order 16 bits of the program counter are loaded with 0x2000. Instruction execution begins at the new address in the program counter.

Executing the return from critical interrupt (**rfci**) instruction restores the contents of the program counter and the MSR from SRR2 and SRR3, respectively, and the processor resumes execution at the address contained in the program counter.

Software is responsible for clearing any Debug exceptions from the DBSR, prior to re-enabling Debug interrupts via the MSR[DE] and/or the DBCR[IDM] bits, in order to avoid another redundant Debug interrupt. To clear Debug exceptions, software must write a word to the DBSR using **mtspr** with a '1' in any bit position that is to be cleared and '0' in all other bit positions. The write data to the DBSR is not direct data, but a mask. A '1' causes the corresponding bit to be cleared, while a '0' has no effect.

**Table 7-16. Register Settings during Debug Interrupt**

SRR2	Set as indicated above, depending on the Debug exception type
SRR3	Set to the value of the MSR at the time of the interrupt
MSR	APE, APA, WE, CE, EE, PR, FP, FE0, DE, FE1, IR, DR ← 0 LE ← ILE ILE, ME ← unchanged
PC	EVPR[0:15]    0x2000
DBSR	Set to indicate type of Debug Event (see Chapter 9, "Debug Facilities," on p. 9-1)

## 7.10 Interrupt Control Instructions

The IBM PowerPC Embedded Operating Environment provides the following interrupt control instructions:

- Move From Machine State Register (**mfmsr**)
- Move To Machine State Register (**mtmsr**)
- Return From Critical Interrupt (**rfdi**)
- Return From Interrupt (**rfi**)
- System Call (**sc**)
- Write External Enable (**wrtee**)
- Write External Enable Immediate (**wrteei**)

The **mfmsr** and **mtmsr** instructions allow the contents of the MSR to be read into and written from a General Purpose Register (GPR), respectively.

**rfci**, **rfi**, and **sc** are system linkage instructions which enable the program to 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 **sc** instruction is described in the PowerPC User Instruction Set Architecture, but only at the level required by an application programmer. System linkage instructions are context synchronizing, as defined in Section 11.1, "Context Synchronization," on page 11-1.

**wrt<sub>ee</sub>** and **wrt<sub>eei</sub>** enable atomic update of the MSR[EE] bit (External Enable).

The interrupt control instructions are defined in detail in the following sections.

### 7.10.1 Move From Machine State Register (mfmsr)

mfmsr RT


$$(RT) \leftarrow (MSR)$$

The contents of the MSR are placed into register RT.

## Registers Altered

- RT

## Invalid Instruction Forms

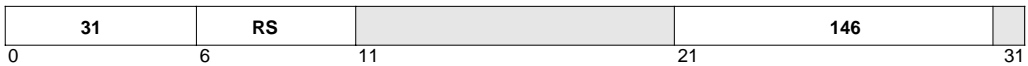
- Reserved fields

### Programming Note

Execution of this instruction is privileged.

## 7.10.2 Move to Machine State Register (mtmsr)

**mtmsr**                      RS



(MSR) ← (RS)

The contents of register RS are placed into the MSR.

Alteration of the EE or CE bit(s) is effective as soon as the instruction completes. Thus if MSR[EE]=0 and an External Input, PIT or FIT interrupt is pending, executing an **mtmsr** instruction that sets MSR[EE] to 1 will cause the External, PIT or FIT interrupt to be taken before the next instruction is executed if no higher priority exception exists. Likewise, if MSR[CE]=0 and a Critical Input or Watchdog Timer interrupt is pending, executing an **mtmsr** instruction that sets MSR[CE] to 1 will cause the Critical Input or Watchdog Timer interrupt to be taken before the next instruction is executed if no higher priority exception exists.(see Section 7.6.2, “Interrupt Order,” on page 7-14).

### Registers Altered

- MSR

### Invalid Instruction Forms

- Reserved fields

### Programming Note

The **mtmsr** instruction is privileged and execution synchronizing.

## 7.10.3 Return From Critical Interrupt (rfci)

**rfci**



(PC) ← (SRR2)

(MSR) ← (SRR3)

The **rfci** instruction is used to return from a critical class interrupt, or as a precise means of establishing a new context and synchronizing on that new context simultaneously. The

program counter (PC) is restored with the contents of SRR2 and the MSR is restored with the contents of SRR3.

Instruction execution returns to the address contained in the PC.

**Registers Altered**

- MSR

**Invalid Instruction Forms**

- Reserved fields

**Programming Note**

The **rfci** instruction is privileged and context synchronizing.

**7.10.4 Return From Interrupt (rfi)**

**rfi**



(PC) ← (SRR0)  
(MSR) ← (SRR1)

The **rfi** instruction is used to return from a non-critical class interrupt, or as a precise means of establishing a new context and synchronizing on that new context simultaneously. The program counter (PC) is restored with the contents of SRR0 and the MSR is restored with the contents of SRR1.

Instruction execution returns to the address contained in the PC.

**Registers Altered**

- MSR

**Invalid Instruction Forms**

- Reserved fields

**Programming Note**

The **rfi** instruction is privileged and context synchronizing.



# 7.10.5 System Call (sc)

sc



```
(SRR1) ← (MSR)
(SRR0) ← (PC)
PC ← EVPR0:15 || x'0C00'
(MSR[APE, APA, WE, EE, PR, FP, FE0, FE1, IR, DR]) ← 0
(MSR[LE]) ← (MSR[ILE])
```

The **sc** instruction is used to request a system service. A System Call interrupt is generated. The contents of the MSR are copied into SRR1 and the address of the instruction after the **sc** instruction is placed into SRR0.

The program counter is then loaded with the exception vector address. The exception vector address is calculated by concatenating the high-order 16 bits of the Exception Vector Prefix Register (EVPR) to the left of 0x0C00.

The MSR[APE, APA, WE, EE, PR, FP, FE0, FE1, IR, DR] bits are set to 0, and MSR[ILE] is copied to MSR[LE].

Program execution continues at the new address in the PC.

## Registers Altered

- SRR0
- SRR1
- MSR[APA, APE, DR, EE, FE0, FE1, FP, IR, LE, PR, WE]

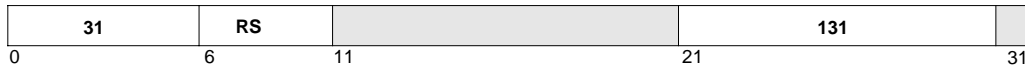
## Invalid Instruction Forms

- Reserved fields

### Programming Note

The **sc** instruction is context synchronizing.

<b>wrtree</b>	RS
---------------	----


$$\text{MSR}[\text{EE}] \leftarrow (\text{RS})_{16}$$

MSR[EE] is set to the value specified by RS[16].

## Registers Altered

- MSR[EE]

### Invalid Instruction Forms:

- Reserved fields

## Programming Note

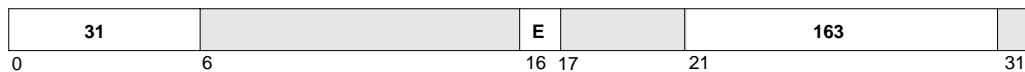
Execution of this instruction is privileged.

This instruction is used to provide atomic update of MSR[EE]. Typical usage is:

<b>mfmsr</b>	Rn	#save EE in Rn[16]
<b>wrttee</b>	0	#turn off EE
<b>.....</b>		#code with EE disabled
<b>wrttee</b>	Rn	#restore EE without affecting other MSR changes that may have occurred during the disabled code

### 7.10.7 Write External Enable Immediate (wrteei)

**wrteei** **E**


$$\text{MSR}[\text{EE}] \leftarrow \text{E}$$

MSR[EE] is set to the value specified by the E field.

## Registers Altered

- MSR[EE]

### Invalid Instruction Forms:

- Reserved fields

## Programming Note

Execution of this instruction is privileged.

This instruction is used to provide atomic update of MSR[EE]. Typical usage is:

<b>mfmsr</b>	Rn	#save EE in Rn[16]
<b>wrteei</b>	0	#turn off EE
• • • • •		#code with EE disabled
<b>wrtee</b>	Rn	#restore EE without affecting other MSR changes that may have occurred during the disabled code



# 8

## Reset and Initialization

---

This chapter discusses in detail the following topics:

- Reset and Initialization
- Reset Mechanisms
- Processor State After Reset
- Initialization Code Example

### 8.1 Reset and Initialization

This chapter describes the requirements for processor reset. This includes both the means of causing reset, and the specific initialization that is required to be performed automatically by the processor hardware. This chapter also provides an overview of the operations that should be performed by initialization software, in order to fully initialize the processor.

In general, the specific actions taken by a processor upon reset are implementation dependent, and are described in the User's Manual for the implementation. Also, it is the responsibility of system initialization software to initialize the majority of processor and system resources after reset. Implementations are required to provide a minimum processor initialization such that this system software may be fetched and executed, thereby accomplishing the rest of system initialization.

### 8.2 Reset Mechanisms

This specification defines two processor mechanisms for internally invoking a reset operation. In addition, implementations will typically provide additional means for invoking a reset operation, via an external mechanism such as a signal pin which when activated will cause the processor to reset.

The two internal mechanisms for invoking a reset operation are:

1. Debug Control Register (DBCR)
2. Timer Control Register (TCR)

## 8.2.1 Debug Control Register (DBCR)

The DBCR[RST] field may be written by software to a non-zero value in order to cause an immediate processor reset. The exact behavior which results from specific non-zero values written to this field is implementation-dependent. Writing a value of zero to this field will have no effect on the processor.

The Most Recent Reset field of the Debug Status Register (DBSR[MRR]) records information about the most recent reset operation which occurred, regardless of the mechanism which invoked the reset. See the User's Manual for the implementation for a definition of this field.

## 8.2.2 Timer Control Register (TCR)

The Watchdog Reset Control field of the Timer Control Register (TCR[WRC]) may be set to a non-zero value by software, in order to allow a Watchdog timeout event to automatically invoke a processor reset. The exact behavior which results from specific non-zero values written to this field is implementation dependent.

Once set to a non-zero value, this field may not be restored to a zero value by software, and will only be restored upon an actual reset operation. This is to prevent errant software from disabling the Watchdog reset safeguard, once it has been established by software.

The Watchdog Reset Status field of the Timer Status Register (TSR[WRS]) records information about only those reset operations which were actually invoked by a Watchdog timeout. See the User's Manual for the implementation for a definition of this field.

For additional information see Section 10.5, "Watch Dog Timer (WDT)," on p. 10-8.

## 8.3 Processor State After Reset

The initial processor state is controlled by the register contents after reset. In general, the contents of most registers are undefined after reset.

The processor hardware only initializes those registers (or specific bits in registers) which must be initialized in order for software to be able to reliably perform the rest of system initialization. Table 8-1 identifies those specific resources which are initialized automatically by hardware upon reset:

**Table 8-1. Contents of Processor Resources After Reset**

Register	Bit Number	Bit Name	Reset Value	Comment
DBCR	0:31	Various	0x0000000	Debug Events disabled
DBSR	22:23	MRR	Implementation Dependent	Indicates information about the most recent reset operation
DCCR	0:31		0x0000000	Data Cacheability disabled

**Table 8-1. Contents of Processor Resources After Reset (cont.)**

Register	Bit Number	Bit Name	Reset Value	Comment
ICCR	0:31		0x0000000	Instruction Cacheability disabled
MSR	11	APE	0	Auxilliary Processor exception type Program interrupts disabled.
	12	APA	0	Auxilliary Processor unavailable
	13	WE	0	Wait State disabled
	14	CE	0	Critical Input and Watchdog interrupts disabled
	15	ILE	0	Interrupt Endianness set to Big Endian
	16	EE	0	External Input and PIT/FIT interrupts disabled
	17	PR	0	Supervisor Mode
	18	FPA	0	Floating Point unavailable
	19	ME	0	Machine Check interrupts disabled
	20, 23	FE0, FE1	00	Floating Point exception type Program interrupts disabled
	22	DE	0	Debug interrupts disabled
	26	IR	0	Instruction Relocation disabled
	27	DR	0	Data Relocation disabled
	31	LE	0	Endianness set to Big Endian
PC	0:31		0xFFFFFFFFC	Processor begins fetching instructions at the last word of real memory
PVR	0:31		Implementation Dependent	Processor Version. This register is read-only, and contains a value which identifies the specific implementation
SGR	0:31		0xFFFFFFFF	All of storage marked as Guarded
TCR	2:3	WRC	00	Watchdog Timer Reset disabled
TSR	2:3	WRS	Implementation Dependent	Indicates information about the most recent reset, if invoked by the Watchdog Timer; otherwise 0b00

## 8.4 Software Initialization Requirements

When reset occurs, the processor is initialized to a minimum configuration to start executing initialization code. Initialization code is necessary to complete the processor and system configuration. The initialization code described in this section is the minimum recommended for configuring the processor to run application code.

Initialization code should configure the following processor resources:

- Initialize DCWR to non-write-thru, to avoid potential alignment interrupts on store operations (must be done before first store).
- Invalidate the i-cache and d-cache (implementation dependent).
- Enable cacheability for appropriate memory regions.
- Turn off guarded attribute (SGR) as appropriate for memory regions (to enable pre-fetching for improved performance).
- Initialize system memory as required by the operating system or application code.
- Initialize processor registers as needed by the system.
- Initialize off-chip system facilities.
- Dispatch the operating system or application code.

### 8.4.1 Initialization Code Example

This section presents an example of initialization code to illustrate the steps that should be taken to initialize the processor before the operating system or user programs are executed. It is presented in pseudo-code with function calls similar to PowerPC instruction mnemonics. Specific implementations may require different ordering of these sections to ensure proper operation.



```

/* _____ */
/* Initialization Pseudo Code */
/* _____ */
@0xFFFFFFFFC: /* Initial instruction fetch from 0xFFFFFFFFC */
    ba(init_code); /* branch from initial fetch address to init_code */

@init_code: /* Start of initialization psuedo code */

/* _____ */
/* Clear DCWR to avoid potential alignment exceptions on stores. */
/* _____ */

    mtspr(DCWR, 0);

/* _____ */
/* Invalidate both caches */
/* _____ */

/* Implementation dependent code sequence to invalidate all lines in instruction cache*/
/* and data cache. */

/* _____ */
/* Enable cacheability for appropriate regions of real storage */
/* _____ */

    mtspr(DCCR, d_cache_cacheability); /* enable data cacheability */
    mtspr(ICCR, i_cache_cacheability); /* enable instruction cacheability */

/* _____ */
/* Clear guarded attribute to allow improved performance via instruction prefetch, as */
/* appropriate. */
/* Can do as early as desired (e.g., before caches invalidated). */
/* _____ */

    mtspr(SGR, guarded_configuration); /* mark appropriate regions as unguarded */

```

```

/*----- */
/* Load operating system and/or application code, including exception handlers, */
/* into memory. */
/* */
/* The example assumes that the system and/or application code is loaded */
/* immediately after the cache is initialized. */
/* The example assumes that the source and destination regions do not overlap and */
/* are aligned on cache line boundaries, and that cache lines consist of four words */
/* (16 bytes) */
/*----- */

while (not_done) /* repeat until all code has been loaded. */
{
    lmw(4, &code); /* load 4 words into 4 registers */
    stmw(4, &new_location); /* store 4 words to d-cache */
    dcbst(&new_location); /* store cache block to physical memory */
    sync(); /* allow store to complete */
    icbi(&new_location); /* clear any obsolete code from i-cache */
    inc(&code); /* increment the code address by 4 words */
    inc(&new_location); /* increment the new_location addr by 4 words */
}

isync(); /* discard prefetched instructions and re-fetch */

/*----- */
/* set the exception vector prefix */
/*----- */

mtspr(EVPR, prefix_addr); /* initialize exception vector prefix */

/*----- */
/* initialize and configure timer facilities */
/*----- */

mtspr(TBL, 0); /* reset time base lower first to avoid ripple */
mtspr(PIT, 0); /* clear PIT so no PIT indication after TSR */
cleared /* */
mtspr(TSR, 0xFFFFFFFF); /* clear Timer Status Register */
mtspr(TCR, timer_enable); /* enable desired timers */
mtspr(TBU, time_base_u); /* set time base, upper first to catch possible */
ripple /* */
mtspr(TBL, time_base_l); /* set time base, lower */
mtspr(PIT, pit_count); /* set desired pit count */

```

```

/* _____ */
/* Enable interrupts in the Machine State Register */
/* _____ */
/* Interrupts should be enabled immediately after timer */
/* facilities to avoid missing a timer exception. */
/* _____ */
/* The MSR also controls the user/supervisor mode, */
/* translation, and the wait state. */
/* These modes must be initialized by the operating */
/* system or application code. */
/* If enabling translation, code must initialize the TLB. */
/* _____ */
/* _____ */

mtmsr(machine_state);

/* _____ */
/* initialization of non-processor facilities should be performed at this time */
/* _____ */

/* _____ */
/* branch to operating system or application code */
/* _____ */
ba(&code_load_address);

```



This chapter discusses in detail the following topics:

- Internal Debug Mode
- Debug Events
- Debug Registers

## 9.1 Background

The IBM PowerPC Embedded Environment provides debug facilities to enable hardware and software debug functions, such as instructions and data breakpoints and program single stepping. The debug facilities consist of special purpose registers (SPR's) for enabling and recording various kinds of debug events, as well as a special Debug interrupt type built into the interrupt mechanism (see Section 7.9.15, “Debug Interrupt,” on page 7-42). The debug facilities also provide a mechanism for software-controlled processor reset, and for controlling the operation of the timers in a debug environment.

Access to the debug facilities via the move from/to special purpose register (**mf spr/mt spr**) instructions, as well as the debug interrupt mechanism, are defined as part of the IBM PowerPC Embedded Environment, and are included in this chapter. In addition, implementations will typically include debug facilities, modes, and access mechanisms which are implementation-specific and defined as part of the User's Manual for the implementation. For example, implementations will typically provide access to the debug facilities via a dedicated Test Access Port, such as JTAG.

## 9.2 Internal Debug Mode

Debug events include such things as instruction and data breakpoints. These debug events cause status bits to be set in the Debug Status Register (DBSR). The existence of a set bit in the DBSR is considered a Debug exception. Debug exceptions, if enabled, will cause Debug interrupts.

There are two different mechanisms that control whether Debug interrupts are enabled. The first is the MSR[DE] bit, and this bit must be set to 1 to enable Debug interrupts. The second mechanism is an enable bit in the Debug Control Register (DBCR). This bit is the Internal Debug Mode bit (IDM), and it must also be set to 1 to enable Debug interrupts.

When DBCR[IDM]=1, the processor is in Internal Debug Mode. In this mode, debug events will (if also enabled by MSR[DE]) cause Debug interrupts. Software at the Debug interrupt vector location will thus be given control upon the occurrence of a debug event, and can access (via the normal instructions) all architected processor resources. In this fashion, debug monitor software can control the processor and gather status, and interact with debugging hardware connected to the processor.

When the processor is not in Internal Debug Mode (DBCR[IDM]=0), debug events may still occur and be recorded in the DBSR. These exceptions may be monitored via software by reading the DBSR (using **mfspr**), or may eventually cause a Debug interrupt if later enabled by setting DBCR[IDM]=1 (and MSR[DE]=1).

### 9.3 Debug Events

Debug events are used to cause Debug exceptions to be recorded in the Debug Status Register (DBSR). In order for a debug event to be enabled to set a DBSR bit and thereby cause a Debug exception, the specific event type must be enabled by a corresponding bit in the Debug Control Register (DBCR) (in most cases; the Unconditional Debug Event (UDE) is an exception to this rule). Once a DBSR bit is set, if Debug interrupts are enabled by both DBCR[IDM] and MSR[DE], a Debug interrupt will be generated.

Certain debug events are not allowed to occur when DBCR[IDM]=1 and MSR[DE]=0. In such situations, no Debug exception occurs and thus no DBSR bit is set. Other debug events may cause Debug exceptions and set DBSR bits regardless of the state of MSR[IDM] and/or MSR[DE]. The associated Debug interrupts that result from such Debug exceptions will be delayed until such time that both DBCR[IDM] and MSR[DE]=1, provided the exceptions have not been cleared from the DBSR in the meantime.

Anytime that a DBSR bit is allowed to be set while MSR[DE]=0, a special DBSR bit -- Imprecise Debug Event (IDE) -- will also be set. The DBSR[IDE] bit indicates that the associated Debug exception bit in the DBSR was set while Debug interrupts were disabled via the MMSR[DE] bit. Debug interrupt handler software can use this bit to determine whether the address recorded in the Save/Restore Register 2 (SRR2) should be interpreted as the address associated with the instruction causing the Debug exception, or simply the address of the instruction after the one which set the MSR[DE] bit, thereby enabling the delayed Debug interrupt.

Debug interrupts are ordered with respect to other interrupt types (see Section 7.6, "Interrupt Ordering and Masking," on page 7-11). Debug exceptions are prioritized with respect to other exceptions (see Section 7.7, "Exception Priorities," on page 7-15).

There are seven types of debug events defined:

1. Instruction Address Compare (IAC)
2. Data Address Compare (DAC)
3. Trap (TR)

4. Branch Taken (BT)
5. Instruction Complete (ICMP)
6. Exception (EXC)
7. Unconditional Debug Event (UDE)

The following sections provide detailed descriptions of the debug event types.

### 9.3.1 Instruction Address Compare (IAC)

This event occurs if Instruction Address Compare debug events are enabled by DBCR[IAC] and execution is attempted of the instruction at the address contained in the Instruction Address Compare Register (IACR). The event can occur regardless of the setting of MSR[DE] and/or DBCR[IDM]. When an Instruction Address Compare debug event occurs, the DBSR[IAC] bit is set to record the Debug exception. A Debug interrupt will occur if and when DBSR[IAC]=1 and both MSR[DE]=1 and DBCR[IDM]=1.

If Debug interrupts are enabled via MSR[DE] and DBCR[IDM] at the time of the Instruction Address Compare Debug exception, a Debug interrupt will occur immediately (provided there exists no higher priority exception which is enabled to cause an interrupt). The execution of the instruction causing the exception will be suppressed, and SRR2 will be set to the address of the excepting instruction.

If Debug interrupts are not enabled at the time of the Instruction Address Compare Debug exception, a Debug interrupt will not occur, and the instruction will complete execution (provided the instruction is not causing some other exception which will generate an enabled interrupt). Also, if MSR[DE]=0 at the time of the Debug exception, the Imprecise Debug Event (IDE) bit of the DBSR will be set, to indicate that the Debug exception occurred while Debug interrupts were disabled by MSR[DE].

Later, if the Debug exception has not been reset by clearing DBSR[IAC], and both MSR[DE] and DBCR[IDM] are set to '1', a delayed Debug interrupt will occur. In this case, SRR2 will contain the address of the instruction after the one which enabled the Debug interrupt by updating either MSR[DE] or DBCR[IDM] such that both bits were set to 1. Software in the Debug interrupt handler can observe DBSR[IDE] to determine how to interpret the value in SRR2.

### 9.3.2 Data Address Compare (DAC)

This event occurs if one of the Data Address Compare debug events is enabled in the DBCR, execution is attempted of a storage access instruction of the type which is enabled in the DBCR, and the address being accessed by the instruction matches the address contained in the Data Address Compare Register. The event can occur regardless of the setting of MSR[DE] and/or DBCR[IDM]. When a Data Address Compare (DAC) debug event occurs, a DBSR bit is set to record the specific type of DAC Debug exception. A Debug interrupt will occur if and when the DBSR bit is set to 1 and both MSR[DE]=1 and DBCR[IDM]=1.

There are two different types of DAC debug events: those caused by read-type accesses (eg. load instructions), and those caused by write-type accesses (eg. store instructions). There are separate bits in the DBCR for enabling read and write type DAC debug events (DBCR[DRD, DWR]). Likewise, there are separate bits in the DBSR for recording the two types of Debug exceptions (DBSR[DRD, DWR]).

In addition, there are two bits in the DBCR for controlling the “precision” of the address comparison that must match in order for the event to be recognized. These bits are the DAC size bits, DBCR[DS0, DS1], and they are used to specify how many (if any) low-order address bits should be ignored when comparing the data access address to the DACR (see Section 9.4.1, “Debug Control Register (DBCR),” on page 9-9 for the full definition of these bits).

As an example of how the DBCR[DS0, DS1] bits are used, consider a debug situation in which it is desired to detect any access to any of the bytes in the 4-byte word at address X'0000 0000'. If the DACR is programmed with the value x'0000 0000' and a word load instruction to that address is executed, this is of course recognized as a match and will cause a debug event, if enabled. However, if a *byte* load instruction to address x'0000 0003' is executed, this instruction also accesses one of the bytes of interest, but the address does not match the DACR, unless two low-order address bits are ignored as part of the address comparison. The DBCR[DS0, DS1] bits allow DACR addresses to be programmed as specific byte addresses, or as multi-byte halfword, word, or cache block addresses.

All load instructions are considered *reads* with respect to debug events, while all store instructions are considered *writes* with respect to debug events. In addition, the cache management instructions, and certain special cases, are handled as follows:

- **dcbt**, **dcbtst**, **icbt**, and **icbi** are all considered *reads* with respect to debug events. Note that **dcbt**, **dcbtst**, and **icbt** are treated as *no-ops* when they report Data Storage or Data TLB Miss exceptions, instead of being allowed to cause interrupts. However, these instructions *are* allowed to cause Debug interrupts, even when they would otherwise have been no-op'ed due to a Data Storage or Data TLB Miss exception.
- **dcbz**, **dcbi**, **dcbf**, and **dcbst** are all considered *writes* with respect to debug events. Note that **dcbf** and **dcbst** are considered *reads* with respect to Data Storage exceptions, since they do not actually change the data at a given address. However, since the execution of these instructions may result in write activity on the processor's data bus, they are treated as *writes* with respect to debug events.
- indexed-string instructions (**lswx**, **stswx**) for which the XER field specifies zero bytes as the length of the string are treated as no-ops, and are not allowed to cause DAC debug events.

If Debug interrupts are enabled via MSR[DE] and DBCR[IDM] at the time of the Data Address Compare Debug exception, a Debug interrupt will occur immediately (provided there exists no higher priority exception which is enabled to cause an interrupt), the execution of the instruction causing the exception will be suppressed, and SRR2 will be set to the address of the excepting instruction. Depending on the type of instruction and/or the



alignment of the data access, the instruction causing the exception may have been partially executed (see Section 7.5, “Partially Executed Instructions,” on page 7-10).

If Debug interrupts are not enabled at the time of the Data Address Compare Debug exception, a Debug interrupt will not occur, and the instruction will complete execution (provided the instruction is not causing some other exception which will generate an enabled interrupt). Also if MSR[DE]=0 at the time of the Debug exception, the Imprecise Debug Event (IDE) bit of the DBSR will be set, to indicate that the Debug exception occurred while Debug interrupts were disabled by MSR[DE].

Later, if the Debug exception has not been reset by clearing DBSR[DAC], and both MSR[DE] and DBCR[IDM] are set to 1, a delayed Debug interrupt will occur. In this case, SRR2 will contain the address of the instruction after the one which enabled the Debug interrupt by updating either MSR[DE] or DBCR[IDM] such that both bits were set to 1. Software in the Debug interrupt handler can observe DBSR[IDE] to determine how to interpret the value in SRR2.

### 9.3.3 Trap (TR)

This event occurs if Trap debug events are enabled by DBCR[TR] and a Trap instruction (**tw**, **twi**) is executed. The event can occur regardless of the setting of MSR[DE] and/or DBCR[IDM]. When a Trap debug event occurs, DBSR[TR] is set to record the Debug exception. A Debug interrupt will occur if and when DBSR[TR]=1 and both MSR[DE]=1 and DBCR[IDM]=1.

If Debug interrupts are enabled via MSR[DE] and DBCR[IDM] at the time of the Trap debug exception, a Debug interrupt will occur immediately (provided there exists no higher priority exception which is enabled to cause an interrupt), and SRR2 will be set to the address of the excepting instruction.

If Debug interrupts are not enabled at the time of the Trap Debug exception, a Debug interrupt will not occur, and a Trap exception type Program interrupt will occur instead. Also if MSR[DE]=0 at the time of the Debug exception, the Imprecise Debug Event (IDE) bit of the DBSR will be set, to indicate that the Debug exception occurred while Debug interrupts were disabled by MSR[DE].

Later, if the Debug exception has not been reset by clearing DBSR[TR], and both MSR[DE] and DBCR[IDM] are set to ‘1’, a delayed Debug interrupt will occur. In this case, SRR2 will contain the address of the instruction after the one which enabled the Debug interrupt by updating either MSR[DE] or DBCR[IDM] such that both bits were set to ‘1’. Software in the debug interrupt handler can observe DBSR[IDE] to determine how to interpret the value in SRR2.

### 9.3.4 Branch Taken (BT)

This event occurs if Branch Taken Debug events are enabled by DBCR[BT] and execution is attempted of a branch instruction whose direction will be *taken* (that is, either an unconditional branch, or a conditional branch whose branch condition is met).

This debug event is not recognized if MSR[DE]=0 and DBCR[IDM]=1 at the time of the execution of the branch instruction. This is because the execution of branch instructions is very common, and allowing these common events to be recorded as exceptions in the DBSR while debug interrupts are disabled via MSR[DE] would mean that the debug interrupt handler software would receive an inordinate number of *imprecise* Debug interrupts every time Debug interrupts were re-enabled via MSR[DE].

However, MSR[DE] being 0 prevents the recognition of Branch Taken debug events if and only if the processor is actually in Internal Debug Mode (DBCR[IDM]=1). If DBCR[IDM]=0, then the assumption is that debug events are not being used to cause interrupts (software may instead be polling the DBSR), and therefore it is proper to record the exception in the DBSR regardless of the setting of MSR[DE].

When a Branch Taken debug event occurs, the DBSR[BT] bit is set to record the Debug exception. A Debug interrupt will occur if and when DBSR[BT]=1 and both MSR[DE]=1 and DBCR[IDM]=1.

If Debug interrupts are enabled via MSR[DE] and DBCR[IDM] at the time of the Branch Taken Debug exception, a Debug interrupt will occur immediately (provided there exists no higher priority exception which is enabled to cause an interrupt), the execution of the instruction causing the exception will be suppressed, and SRR2 will be set to the address of the excepting instruction.

If Debug interrupts are not enabled at the time of the Branch Taken Debug exception (which implies that MSR[DE]=0 and DBCR[IDM]=0, since the event would not be recognized if only MSR[DE]=0), a Debug interrupt will not occur, and the instruction will complete execution (provided the instruction is not causing some other exception which will generate an enabled interrupt).

Later, if the Debug exception has not been reset by clearing DBSR[BT], and both MSR[DE] and DBCR[IDM] are set to '1', a delayed Debug interrupt will occur. In this case, SRR2 will contain the address of the instruction after the one which enabled the Debug interrupt by updating either MSR[DE] or DBCR[IDM] such that both bits were set to 1. Software in the debug interrupt handler can observe DBSR[IDE] to determine how to interpret the value in SRR2.

### 9.3.5 Instruction Complete (ICMP)

This event occurs if Instruction Complete debug events are enabled by DBCR[ICMP] and execution of any instruction is completed. Note that if execution of an instruction is suppressed due to the instruction causing some other exception which is enabled to generate an interrupt, then the attempted execution of that instruction does not cause an Instruction Complete debug event. The System Call (**sc**) instruction does not fall into the category of an instruction whose execution is suppressed, since the instruction actually completes execution and then generates a System Call interrupt. In this case, the Instruction Complete Debug exception will also be set.

This debug event is not recognized if  $MSR[DE]=0$  and  $DBCR[IDM]=1$  at the time of the execution of the instruction. This is because allowing the common event of Instruction Completion to be recorded as an exception in the DBSR while Debug interrupts are disabled via  $MSR[DE]$  would mean that the Debug interrupt handler software would receive an inordinate number of imprecise Debug interrupts every time Debug interrupts were re-enabled via  $MSR[DE]$ . Moreover it would be impossible to return from the Debug interrupt handler if the instruction trying to return (typically **rfci**) were itself allowed to cause an Instruction Complete Debug exception, even though  $MSR[DE]=0$  at the time of execution of the return instruction.

However,  $MSR[DE]$  being 0 prevents the recognition of Instruction Complete debug events if and only if the processor is actually in Internal Debug Mode ( $DBCR[IDM]=1$ ). If  $DBCR[IDM]=0$ , then the assumption is that debug events are not being used to cause interrupts (software may instead be polling the DBSR), and therefore it is proper to record the exception in the DBSR regardless of the setting of  $MSR[DE]$ .

When an Instruction Complete debug event occurs,  $DBSR[ICMP]$  is set to record the Debug exception. A Debug interrupt will occur if and when  $DBSR[ICMP]=1$  and both  $MSR[DE]=1$  and  $DBCR[IDM]=1$ .

If Debug interrupts are enabled via  $MSR[DE]$  and  $DBCR[IDM]$  at the time of the Instruction Complete Debug exception, a Debug interrupt will occur immediately (provided there exists no higher priority exception which is enabled to cause an interrupt), and  $SRR2$  will be set to the address of the instruction after the one causing the Debug exception.

If Debug interrupts are not enabled at the time of the Instruction Complete Debug exception (which implies that both  $MSR[DE]=0$  and  $DBCR[IDM]=0$ , since the event would not be recognized if only  $MSR[DE]=0$ ), a Debug interrupt will not occur.

Later, if the Debug exception has not been reset by clearing  $DBSR[ICMP]$ , and both  $MSR[DE]$  and  $DBCR[IDM]$  are set to 1, a delayed Debug interrupt will occur. In this case,  $SRR2$  will contain the address of the instruction after the one which enabled the Debug interrupt by updating either  $MSR[DE]$  or  $DBCR[IDM]$  such that both bits were set to 1. Software in the Debug interrupt handler can observe  $DBSR[IDE]$  to determine how to interpret the value in  $SRR2$ .

### 9.3.6 Exception (EXC)

This event occurs if Exception debug events are enabled by  $DBCR[EXC]$  and an interrupt occurs. The event can occur regardless of the setting of  $MSR[DE]$  and/or  $DBCR[IDM]$ . When an Exception debug event occurs,  $DBSR[EXC]$  is set to record the Debug exception. A Debug interrupt will occur if and when  $DBSR[EXC]=1$  and both  $MSR[DE]=1$  and  $DBCR[IDM]=1$ .

Note that if  $DBCR[IDM]=1$ , then only non-critical class interrupts will be allowed to cause an Exception debug event; critical class interrupts will not cause the event. This is because all critical interrupts automatically clear  $MSR[DE]$ , and thus would always prevent the associated Debug interrupt from occurring precisely. Also, Debug interrupts themselves are

critical class interrupts, and thus any Debug interrupt (for any other debug event) would always end up setting the additional exception of DBSR[EXC] upon entry to the Debug interrupt handler. At this point, the Debug interrupt handler would be unable to determine whether or not the Exception debug event was related to the original debug event.

If DBCR[IDM]=0, then critical class interrupts *will* be allowed to cause the Exception debug event, and set DBSR[EXC]. In this case, the assumption is that debug events are not being used to cause interrupts (software may instead be polling the DBSR), and therefore it is proper to record the exception in the DBSR even though the critical interrupt that is causing the Exception debug event will be clearing MSR[DE].

If Debug interrupts are enabled via MSR[DE] and DBCR[IDM] at the time of the Exception debug event, a Debug interrupt will occur immediately (provided there exists no higher priority exception which is enabled to cause an interrupt), and SRR2 will be set to the address of the non-critical interrupt vector which caused the Exception debug event. No instruction at the non-critical interrupt handler will have been executed.

If Debug interrupts are not enabled at the time of the Exception debug event, a Debug interrupt will not occur, and the handler for the interrupt which caused the Exception debug event will be allowed to execute. Also, if MSR[DE]=0 at the time of the Exception debug event, the Imprecise Debug Event (IDE) bit of the DBSR will be set, to indicate that the debug event occurred while Debug interrupts were disabled by MSR[DE].

Later, if the Debug exception has not been reset by clearing DBSR[EXC], and both MSR[DE] and DBCR[IDM] are set to '1', a delayed Debug interrupt will occur. In this case, SRR2 will contain the address of the instruction after the one which enabled the Debug interrupt by updating either MSR[DE] or DBCR[IDM] such that both bits were set to 1. Software in the Debug interrupt handler can observe the DBSR[IDE] bit to determine how to interpret the value in SRR2.

### 9.3.7 Unconditional Debug Event (UDE)

This event occurs when the Unconditional Debug Event (UDE) signal is activated by the debug mechanism. The exact definition of the UDE signal and how it is activated is implementation-dependent (see the User's Manual for the implementation for more details). Note that UDE is the only debug event which does not have a corresponding enable bit for the event in the DBCR (hence the name of the event).

The UDE event can occur regardless of the setting of MSR[DE] and/or DBCR[IDM]. When a UDE event occurs, the DBSR[UDE] bit is set to record the Debug exception. A Debug interrupt will occur if and when DBSR[UDE]=1 and both MSR[DE]=1 and DBCR[IDM]=1.

If Debug interrupts are enabled via MSR[DE] and DBCR[IDM] at the time of the UDE Debug exception, a Debug interrupt will occur immediately (provided there exists no higher priority exception which is enabled to cause an interrupt), and SRR2 will be set to the address of the instruction which would have executed next had the interrupt not occurred.

If Debug interrupts are not enabled at the time of the UDE Debug exception, a Debug interrupt will not occur. Also, if MSR[DE]=0 at the time of the Debug exception, the

Imprecise Debug Event (IDE) bit of the DBSR will be set, to indicate that the Debug exception occurred while Debug interrupts were disabled by MSR[DE].

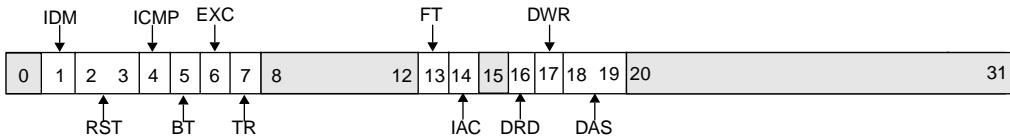
Later, if the Debug exception has not been reset by clearing DBSR[UDE], and both MSR[DE] and DBCR[IDM] are set to 1, a delayed Debug interrupt will occur. In this case, SRR2 will contain the address of the instruction after the one which enabled the Debug interrupt by updating either MSR[DE] or DBCR[IDM] such that both bits were set to 1. Software in the Debug interrupt handler can observe DBSR[IDE] to determine how to interpret the value in SRR2.

## 9.4 Debug Registers

This section describes debug-related registers that are accessible to software running on the processor. These registers are intended for use by special debug tools and debug software, and not by general application or operating system code.

### 9.4.1 Debug Control Register (DBCR)

The Debug Control Register (DBCR) is used to enable debug events, reset the processor, control timer operation during debug events, and set the debug mode of the processor. The contents of the DBCR can be read into a general purpose register via the Move From Special Purpose Register (**mfspr**) instruction. The contents of a general purpose register can be written to the DBCR via the Move To Special Purpose Register (**mtspr**) instruction. Figure 9-1 below provides bit definitions for the DBCR.



**Figure 9-1. Debug Control Register (DBCR)**

0		Reserved
1	IDM	Internal Debug Mode 0 Disable 1 Enable
2:3	RST	Reset 00 No action 01 10 11 <div>The exact function of 01, 10, and 11 settings is implementation-dependent. See the User's Manual for the implementation for details.</div> <div><b>Warning:</b> Writing 01, 10, or 11 to these bits will cause a processor reset to occur</div>

**Figure 9-1. Debug Control Register (DBCR) (cont.)**

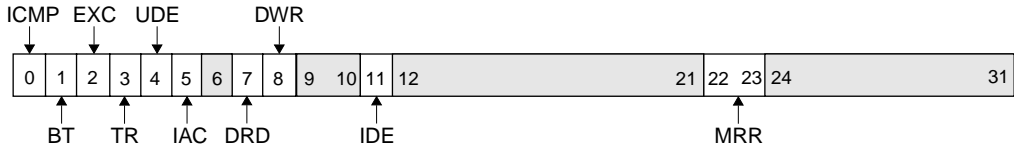
4	ICMP	Instruction Completion Debug Event 0 Disable 1 Enable	(Instruction Completion will not cause a debug event if MSR[DE] = 0 in Internal Debug Mode)
5	BT	Branch Taken Debug Event 0 Disable 1 Enable	(Branch Taken will not cause a debug event if MSR[DE] = 0 in Internal Debug Mode)
6	EXC	Exception Debug Event 0 Disable 1 Enable	(Critical interrupts will not cause a debug event if MSR[DE] = 0 in Internal Debug Mode)
7	TR	TRAP Debug Event 0 Disable 1 Enable	
8:12		Reserved	
13	FT	Freeze Timers on Debug Event 0 Enable clocking of timers 1 Disable clocking of timers	
14	IAC	Instruction Address Compare Debug Event 0 Disable 1 Enable	
15		Reserved	
16	DRD	Data Address Compare Read Debug Event 0 Disable 1 Enable	
17	DWR	Data Address Compare Write Debug Event 0 Disable 1 Enable	
18:19	DAS	Data Address Size 00 Compare all bits 01 Ignore 1 least significant bit 10 Ignore 2 least significant bits 11 Ignore 'n' least significant bits	(see explanation below table)    'n' is determined by the cache block size
20:31		Reserved	

The Data Address Compare (DAC) debug event can be set to react to any byte in a larger block of memory, in addition to reacting to a precise address match. The Data Address Size (DAS) field allows DAC debug events to react to any byte in a halfword (field = 01 — Ignore 1 least significant bit), any byte in a word (field = 10 — Ignore 2 least significant bits), or any byte in a cache block (field = 11 — Ignore 'n' least significant bits, where 'n' is determined by the cache block size and is implementation-dependent). These capabilities are in addition to exact address compare (field = 00 — Compare All Bits).

The addresses for DAC are the operand addresses (effective addresses) of storage operations. As an example, suppose that it is desired to react precisely to byte 3 of a word-aligned target. DAC set for precise compare would fail to recognize access to that byte via aligned word or halfword operations, since that byte address is not the effective address of the operation. In such a case, the Data Address Size field must be set for a wider capture range (for example, ignore 2 least significant bits if word operations to the non-word-aligned byte are to be detected). This wider capture range may result in excess debug events (events that are within the specified capture range, but reflect operations to bytes in addition to the desired byte). These excess debug events must be handled by software.

### 9.4.2 Debug Status Register (DBSR)

The Debug Status Register (DBSR) contains status on debug events and the most recent processor reset. The DBSR is set via hardware, and read and cleared via software. The contents of the DBSR can be read into a general purpose register via the Move From Special Purpose Register (**mfspir**) instruction. Bits in the DBSR can be cleared using the Move To Special Purpose Register (**mtspir**) instruction. Clearing is done by writing a word to the DBSR using **mtspir** with a '1' in any bit position that is to be cleared and '0' in all other bit positions. The write-data to the status register is not direct data, but a mask. A '1' causes the bit to be cleared, and a '0' has no effect. Figure 9-2 provides bit definitions for the DBSR.



**Figure 9-2. Debug Status Register (DBSR)**

0	ICMP	Instruction Completion Debug Event 0 Event didn't occur 1 Event occurred
1	BT	Branch Taken Debug Event 0 Event didn't occur 1 Event occurred
2	EXC	Exception Debug Event 0 Event didn't occur 1 Event occurred
3	TR	TRAP Instruction Debug Event 0 Event didn't occur 1 Event occurred

**Figure 9-2. Debug Status Register (DBSR) (cont.)**

4	UDE	Unconditional Debug Event 0 Event didn't occur 1 Event occurred	
5	IAC	Instruction Address Compare Debug Event 0 Event didn't occur 1 Event occurred	
6		Reserved	
7	DRD	Data Address Compare Read Debug Event 0 Event didn't occur 1 Event occurred	
8	DWR	Data Address Compare Write Debug Event 0 Event didn't occur 1 Event occurred	
9:10		Reserved	
11	IDE	Imprecise Debug Event 0 Event didn't occur 1 Debug Event occurred while MSR[DE]=0	
12:21		Reserved	
22:23	MRR	Most Recent Reset 00 No reset occurred since these bits last cleared by software 01 10 11	These two bits are set to one of three values when a reset occurs. These two bits are undefined at power-up. Implementation-dependent reset information. See the User's Manual for the implementation for further details.
24:31		Reserved	

### 9.4.3 Data Address Compare Register (DACR)

A debug event may be enabled to occur upon loads, stores, or cache operations to an address defined in the DACR register. The DRD, DWR, and DAS fields of the DBCR control the Data Address Compare debug event. (See Section 9.4.1, "Debug Control Register (DBCR)," on page 9-9 for more information on these fields).

The contents of the DACR can be read into a general purpose register via the Move From Special Purpose Register (**mfspr**) instruction. The contents of a general purpose register can be written to the DACR via the move to special purpose register (**mtspr**) instruction.

The contents of the DACR register are compared to the address generated by a storage access instruction. Figure 9-3 provides bit definitions for the DACR.



0	31
---	----

**Figure 9-3. Data Address Compare Register (DACR)**

0:31		Data Address Compare, Byte Address
------	--	------------------------------------

#### 9.4.4 Instruction Address Compare Registers (!ACR)

A debug event may be enabled to occur upon an attempt to execute an instruction from an address defined in the IACR register. Since all instruction addresses are required to be word-aligned, the two low-order bits of the IACR are reserved and do not participate in the comparison to the instruction address. The contents of the IACR can be read into a general purpose register via the Move From Special Purpose Register (**mfspr**) instruction. The contents of a general purpose register can be written to the IACR via the Move To Special Purpose Register (**mtspr**) instruction. Figure 9-4 provides bit definitions for the IACR.

0	29	30	31
---	----	----	----

**Figure 9-4. Instruction Address Compare Registers (!ACR)**

0:29		Instruction Address Compare, Word Address	(omit 2 low-order bits of complete address)
30:31		reserved	



This chapter discusses in detail the following topics:

- Time Base
- Programmable Interval Timer (PIT)
- Fixed Interval Timer (FIT)
- Watch Dog Timer (WDT)
- Timer Control Register (TCR)
- Timer Status Register (TSR)
- Freezing the Timer Facilities

## 10.1 Background and Information

The Time Base, Programmable Interval Timer (PIT), Fixed Interval Timer (FIT), and Watchdog Timer (WDT) provide timing functions for the system. All are volatile resources and must be initialized during start-up. The **mtfb** instruction is used to read the Time Base; the **mtspr** and **mfspr** instructions are used to write the Time Base and Programmable Interval Timer and to read the Programmable Interval Timer.

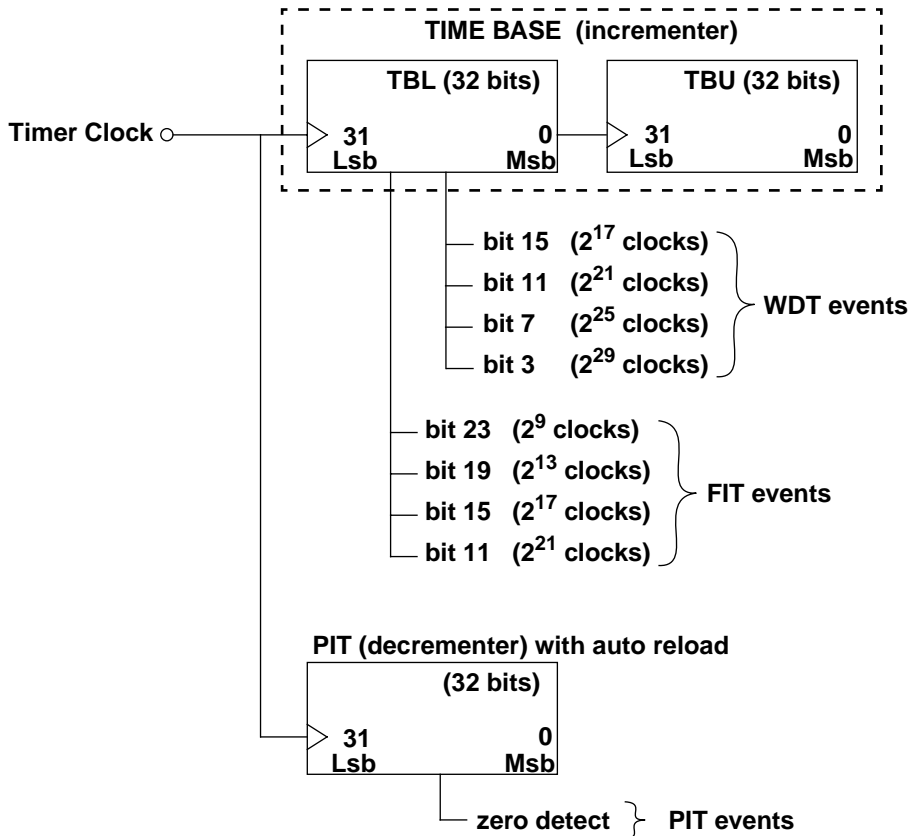
- The Time Base provides a long-period counter driven by an implementation-dependent frequency.
- The PIT, a counter that is updated at the same rate as the Time Base, provides a means of signaling an exception after a specified amount of time has elapsed unless:
  - the PIT is altered by software in the interim, or
  - the Time Base update frequency changes

The PIT is typically used as a general purpose software timer.

- The FIT is really a selected bit of the Time Base, which provides a means of signalling an exception whenever the selected bit transitions from 0 to 1, in a repetitive fashion. The FIT is typically used to trigger periodic system maintenance functions.

- The WDT is also a selected bit of the Time Base, which provides a means of signalling a critical class exception whenever the selected bit transitions from 0 to 1. In addition, if software does not respond in time to the initial exception (by clearing the associated status in the Timer Status Register (TSR) prior to the next expiration of the WDT interval), then a WDT-generated processor reset may result, if so enabled. The WDT is typically used to provide a system error recovery function.

The relationship of these Timer facilities to each other is illustrated in Figure 10-1 below:



**Figure 10-1. Relationship of Timer Facilities to Time Base**

# 10.2 Time Base

The Time Base is a 64-bit register (see Figure 10-2) 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.

Figure 10-2. Time Base (TB)

0	TBU	31	32	TBL	63
---	-----	----	----	-----	----

There is no automatic initialization of the Time Base; system software must perform this initialization.

The Time Base increments until its value becomes 0xFFFF\_FFFF\_FFFF\_FFFF (2<sup>64</sup>–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 frequency is 100 MHz and that the Time Base is driven by this frequency divided by 32. Then the period of the Time Base would be

$$T_{TB} = \frac{2^{64} \times 32}{100 \text{ MHz}} = 5.90 \times 10^{12} \text{ seconds}$$

which is approximately 187,000 years.

The Time Base must be implemented such that the following requirements are satisfied:

- Reading the Time Base into a GPR shall have no effect on the accuracy of the Time Base.
- Writing the Time Base from a GPR shall replace the value in the Time Base with the value in 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 system environment. 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 supervisor state (MSR[PR]=0).

#### Architecture Note

Disabling the Time Base or making the **mftb** instruction privileged prevents the Time Base from being used to implement a "covert channel" in a secure system.

#### Programming Notes

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

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

Successive readings of the Time Base may return identical values.

See Section 3.4, "Computing Time of Day from the Time Base," on page 3-6 for ways to compute time of day in POSIX format from the Time Base.

#### Architecture Note

It is intended that the Time Base be useful for timing reasonably short sequences of code (a few hundred instructions) and for low-overhead time stamps for tracing. The Time Base should not "tick" faster than the CPU instruction clock. Driving the Time Base directly from the CPU instruction clock is probably finer granularity than necessary; the instruction clock divided by 8, 16, or 32 would be more appropriate.

The Time Base driving frequency is also used to update the Programmable Interval Timer, which is used by system software to set interval timers ("alarms"), as well as affecting the period of the Fixed Interval Timer and Watchdog Timer. The update frequency chosen should be appropriate for these purposes as well.

### 10.2.1 Writing the Time Base

Writing the Time Base is privileged. Reading the Time Base is not privileged. It is discussed in the IBM PowerPC Embedded Virtual Environment, Section 3.3, "Reading the Time Base," on page 3-6.

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 Figure 5-1 on page 6-1 for a summary of SPRs, including the Time Base registers).

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	# force 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.

### 10.3 Programmable Interval Timer (PIT)

The PIT is a 32-bit register that decrements at the same rate that the Time Base increments. The PIT is read/written via **mfspir/mtspir**. Writing to the PIT using **mtspir** simultaneously writes to a hidden reload register. Reading the PIT using **mfspir** returns the current PIT contents; there is no mechanism to read the content of the hidden reload register. When written to a non-zero value, the PIT begins decrementing. A PIT event occurs when a decrement occurs on a PIT count of one. When the PIT event occurs, the following things happen:

1. If auto-reload mode is enabled in the Timer Control Register (TCR[ARE]=1), the PIT reloads with the last value that was written to the PIT using a **mtspir** instruction. The decrement from one immediately causes the reload; an intermediate PIT content of zero does not occur.  
  
If auto-reload mode is not enabled (TCR[ARE]=0), decrement from one simply causes a PIT content of zero, and the PIT stops decrementing.
2. The PIT Interrupt Status bit in the Timer Control Register (TCR[PIS]) is set. This constitutes a PIT exception.
3. If enabled by TCR[PIE] and MSR[EE], a PIT interrupt is generated. See Section 7.9.10, “Programmable Interval Timer Interrupt,” on page 7-38 for details of register behavior caused by the PIT interrupt.

The interrupt handler must reset TSR[PIS] via software, in order to avoid another PIT interrupt once MSR[EE] is re-enabled (assuming TCR[PIE] is not cleared instead). This is done by writing a word to TSR using **mtspir** with a ‘1’ in the bit corresponding to TSR[PIS] (and any other bits that are to be cleared) and 0 in all other bits. The write-data to the TSR is not direct data, but a mask. A ‘1’ causes the bit to be cleared, and a ‘0’ has no effect.

Forcing the PIT to 0 using the **mtspr** instruction will *not* cause a PIT exception; however, decrementing which was in progress at the instant of the **mtspr** instruction may cause the exception. To eliminate the PIT as a source of exceptions, set TCR[PIE]=0 (clear the PIT Interrupt Enable bit).

If it is desired to eliminate all PIT activity, the procedure is as follows:

1. Write 0 to TCR[PIE]. This will prevent PIT activity from causing exceptions.
2. Write 0 to TCR[ARE] to disable the PIT auto-reload feature.
3. Write 0 to PIT. This will halt PIT decrementing. While this action will not cause a PIT exception to be set in TSR[PIS], a near simultaneous decrement may have done so.
4. Write 1 to TSR[PIS]. This action will clear TSR[PIS] to 0 (see Section 10.7, "Timer Status Register (TSR)," on page 10-12). This will clear any PIT exception which may be pending. Because the PIT is frozen at zero, no further PIT events are possible.

If the auto-reload feature is disabled (TCR[ARE]=0), then once the PIT decrements to zero, it will stay there until software reloads it using the **mtspr** instruction.

On reset, TCR[ARE] is cleared to zero. This disables the auto-reload feature.

Figure 10-3 illustrates the PIT.

0	31
---	----

**Figure 10-3. Programmable Interval Timer (PIT)**

0:31		Programmed Interval Remaining	(The number of clocks remaining until the PIT event)
------	--	-------------------------------	--



## 10.4 Fixed Interval Timer (FIT)

The FIT is a mechanism for providing timer interrupts with a repeatable period, to facilitate system maintenance. It is similar in function to an auto-reload PIT, except that there are fewer selections of interrupt period available. The FIT exception occurs on 0→1 transitions of a selected bit from the Time Base, as per Table 10-1:

**Table 10-1. Fixed Interval Timer**

TCR[FP]	TBL Bit	Period (Time Base clocks)	Period (100 MHz clock)
0b00	23	$2^9$ clocks	5.12 $\mu$ sec
0b01	19	$2^{13}$ clocks	81.92 $\mu$ sec
0b10	15	$2^{17}$ clocks	1.311 msec
0b11	11	$2^{21}$ clocks	20.97 msec

The FIT exception is logged by TSR[FIS]. A FIT interrupt will occur if TCR[FIE] and MSR[EE] are enabled. See Section 7.9.11, “Fixed Interval Timer Interrupt,” on page 7-39 for details of register behavior caused by the FIT interrupt.

The interrupt handler must reset TSR[FIS] via software, in order to avoid another FIT interrupt once MSR[EE] is re-enabled (assuming TCR[FIE] is not cleared instead). This is done by writing a word to TSR using **mtspr** with a ‘1’ in the bit corresponding to TSR[FIS] (and any other bits that are to be cleared) and ‘0’ in all other bits. The write-data to the status register is not direct data, but a mask. A ‘1’ causes the bit to be cleared, and a ‘0’ has no effect.

Note that a FIT exception will also occur if the selected Time Base bit transitions from 0-1 due to **mttbl** that writes a ‘1’ to the bit when its previous value was ‘0’.

## 10.5 Watch Dog Timer (WDT)

The Watchdog Timer is a facility intended to aid system recovery from faulty software or hardware. Watchdog timeouts occur on 0→1 transitions of selected bits from the Time Base, as per Table 10-2:

**Table 10-2. Watch Dog Timer**

TCR[WP]	TBL Bit	Period (Time Base clocks)	Period (100 Mhz clock)
0b00	15	$2^{17}$ clocks	1.311 msec
0b01	11	$2^{21}$ clocks	20.97 msec
0b10	7	$2^{25}$ clocks	335.5 msec
0b11	3	$2^{29}$ clocks	5.369 sec

When a WDT timeout occurs while Watchdog Interrupt Status is clear (TSR[WIS] = 0) and the next Watchdog Timeout is enabled (TSR[ENW] = 1), a WDT exception is generated and logged by setting TSR[WIS] to 1. This is referred to as a *WDT First Time Out*. A Watchdog Timer interrupt will occur if enabled by TCR[WIE] and MSR[CE]. See

Section 7.9.12, “Watchdog Timer Interrupt,” on page 7-40 for details of register behavior caused by the Watchdog interrupt.

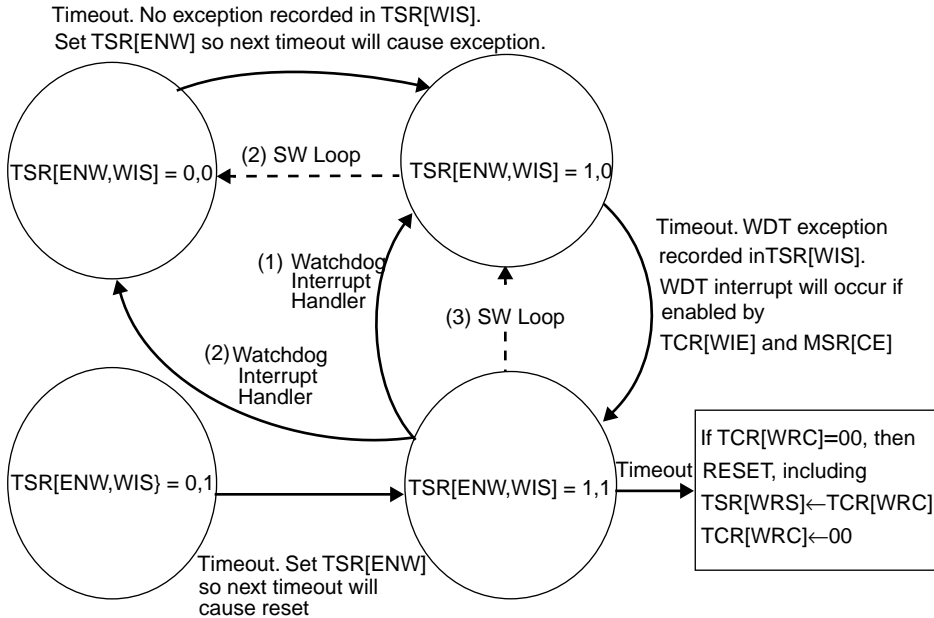
The interrupt handler must reset TSR[WIS] via software, in order to avoid another WDT interrupt once MSR[CE] is re-enabled (assuming TCR[WIE] is not cleared instead). This is done by writing a word to the TSR using **mtspr** with a ‘1’ in the bit corresponding to TSR[WIS] (and any other bits that are to be cleared) and a ‘0’ in all other bits. The write-data to the TSR is not direct data, but a mask. A ‘1’ causes the bit to be cleared, and a ‘0’ has no effect.

Note that a WDT exception will also occur if the selected Time Base bit transitions from 0-1 due to a **mttbl** that writes a ‘1’ to the bit when its previous value was ‘0’.

When a WDT timeout occurs while TSR[WIS] = 1 and TSR[ENW] = 1, a processor reset occurs if enabled by a non-zero value of the Watchdog Reset Control field in the TCR (TCR[WRC]). This is referred to as a *WDT Second Time Out*. The assumption is that TSR[WIS] was not cleared because the processor was unable to execute the Watchdog interrupt handler, leaving reset as the only available means to restart the system. Note that once TCR[WRC] has been set to a non-zero value, it cannot be reset by software; this feature prevents errant software from disabling the WDT reset capability.

## Programming Note

A more thorough view of Watchdog behavior is afforded by Figure 10-4 and Table 10-3, which describe the Watchdog state machine and Watchdog timer controls. The numbers in parentheses in the figure refer to the discussion of modes of operation which follow the table.



**Figure 10-4. Watchdog State Machine**

**Table 10-3. Watchdog Timer Controls**

Enable Next WDT TSR[ENW]	WDT Status TSR[WIS]	Action when timer interval expires
0	0	Set Enable Next Watchdog (TSR[ENW]=1).
0	1	Set Enable Next Watchdog (TSR[ENW]=1).
1	0	Set Watchdog interrupt status bit (TSR[WIS]=1). If Watchdog interrupt is enabled (TCR[WIE]=1 and MSR[CE]=1), then interrupt.
1	1	Cause Watchdog reset action specified by TCR[WRC]. Reset will copy pre-reset TCR[WRC] into TSR[WRS], then clear TCR[WRC].

The controls described in the above table imply three different modes of operation that a programmer might select for the Watchdog Timer. Each of these modes assumes that TCR[WRC] has been set to allow processor reset by the Watchdog facility:

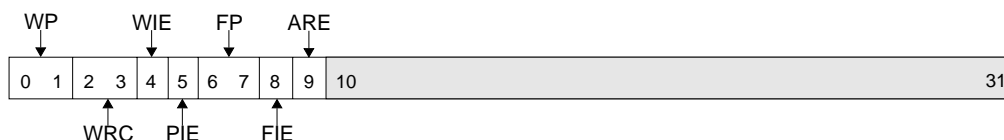
1. Always take the Watchdog Timer interrupt when pending, and never attempt to prevent its occurrence. In this mode, the Watchdog Timer interrupt caused by a first timeout is used to clear TSR[WIS] so a second timeout never occurs. TSR[ENW] is not cleared, thereby allowing the next timeout to cause another interrupt.
2. Always take the Watchdog Timer interrupt when pending, but avoid when possible. In this mode a recurring code loop of reliable duration (or perhaps a periodic interrupt handler such as the FIT interrupt handler) is used to repeatedly clear TSR[ENW] such that a first timeout exception is avoided, and thus no Watchdog Timer interrupt occurs. Once TSR[ENW] has been cleared, software has between one and two full Watchdog periods before a Watchdog exception will be posted in TSR[WIS]. If this occurs before the software is able to clear TSR[ENW] again, a Watchdog Timer interrupt will occur. In this case, the Watchdog Timer interrupt handler will then clear both TSR[ENW] and TSR[WIS], in order to (hopefully) avoid the next Watchdog Timer interrupt.
3. Never take the Watchdog Timer interrupt. In this mode, Watchdog Timer interrupts are disabled (via TCR[WIE]=0), and the system depends upon a recurring code loop of reliable duration (or perhaps a periodic interrupt handler such as the FIT interrupt handler) to repeatedly clear TSR[WIS] such that a second timeout is avoided, and thus no reset occurs. TSR[ENW] is not cleared, thereby allowing the next timeout to set TSR[WIS] again. The recurring code loop must have a period which is less than one WDT period in order to guarantee that a WDT reset will not occur.

## 10.6 Timer Control Register (TCR)

The Timer Control Register (TCR) controls PIT, FIT, and WDT options. The contents of the TCR can be read into a general purpose register via the Move From Special Purpose Register (**mf spr**) instruction. The contents of a general purpose register can be written to the TCR via the Move To Special Purpose Register (**mt spr**) instruction.

The Watchdog Reset Control field (TCR[WRC]) is cleared to zero by processor reset (see Table 8-1 on page 8-2). These bits are set only by software; however, hardware does not allow software to clear these bits once they have been set. Once software has written a '1' to one of these bits, that bit remains a '1' until a reset occurs. This is to prevent errant code from disabling the Watchdog reset function.

TCR[ARE] is cleared to zero by processor reset. This disables the auto-reload feature of the PIT.



**Figure 10-5. Timer Control Register (TCR)**

0:1	WP	Watchdog Period 00 $2^{17}$ clocks 01 $2^{21}$ clocks 10 $2^{25}$ clocks 11 $2^{29}$ clocks	
2:3	WRC	Watchdog Reset Control 00 No Watchdog reset will occur  01 10 11	TCR[WRC] resets to 00. This field may be set by software, but cannot be cleared by software (except by a software-induced reset).  Force processor to be reset on second timeout of WDT. The exact function of any of these settings is implementation-dependent. See the User's Manual for the implementation for further details.
4	WIE	Watchdog Interrupt Enable 0 Disable WDT interrupt 1 Enable WDT interrupt	
5	PIE	PIT Interrupt Enable 0 Disable PIT interrupt 1 Enable PIT interrupt	

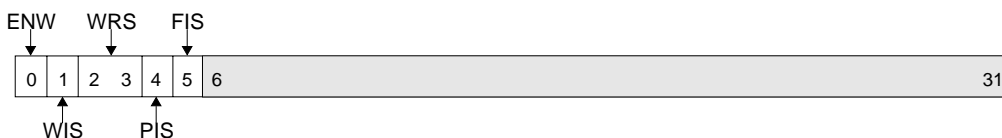
**Figure 10-5. Timer Control Register (TCR) (cont.)**

6:7	FP	FIT Period 00 $2^9$ clocks 01 $2^{13}$ clocks 10 $2^{17}$ clocks 11 $2^{21}$ clocks
8	FIE	FIT Interrupt Enable 0 Disable FIT interrupt 1 Enable FIT interrupt
9	ARE	Auto Reload Enable (disables on reset) 0 Disable auto reload 1 Enable auto reload
10:31		reserved

## 10.7 Timer Status Register (TSR)

The Timer Status Register (TSR) contains status on timer events and the most recent Watchdog Timer-initiated processor reset.

The TSR is set via hardware, and read and cleared via software. The contents of the TSR can be read into a general purpose register via the Move From Special Purpose Register (**mf spr**) instruction. Bits in the TSR can be cleared using the Move To Special Purpose Register (**mt spr**) instruction. Clearing is done by writing a word to the TSR using **mt spr** with a '1' in any bit position that is to be cleared and '0' in all other bit positions. The write-data to the status register is not direct data, but a mask. A '1' causes the bit to be cleared, and a '0' has no effect.



**Figure 10-6. Timer Status Register (TSR)**

0	ENW	Enable Next Watchdog (See Section 10.5 on page 10-8) 0 Action on next Watchdog time-out is to set TSR[0]. 1 Action on next Watchdog time-out is governed by TSR[WIS].
---	-----	---

**Figure 10-6. Timer Status Register (TSR) (cont.)**

1	WIS	Watchdog Interrupt Status 0 No Watchdog interrupt is pending 1 Watchdog interrupt is pending	
2:3	WRS	Watchdog Reset Status 00 No Watchdog reset has occurred  01 10 11	These two bits are set to one of three values when a reset is caused by the Watchdog Timer. These bits are undefined at power-up.  Implementation-dependent reset information. See the User's Manual for the implementation for further details.
4	PIS	PIT Interrupt Status 0 No PIT interrupt is pending 1 PIT interrupt is pending	
5	FIS	FIT Interrupt Status 0 No FIT interrupt is pending 1 FIT interrupt is pending	
6:31		reserved	

## 10.8 Freezing the Timer Facilities

The debug mechanism provides a means of temporarily freezing the timers upon a debug event. Specifically, the Time Base and Programmable Interval Timer can be frozen and prevented from incrementing/decrementing, respectively, whenever a debug event is set in the DBSR. This allows a debugger to simulate the appearance of 'real time', even though the application has been temporarily 'halted' to service the debug event.





## Synchronization Requirements

---

This chapter discusses in detail the following topics:

- Context Synchronization
- Execution Synchronization
- Synchronization Requirements

### 11.1 Context Synchronization

An instruction or event is *context synchronizing* if it satisfies the requirements listed below. Such instructions and events are collectively called *context synchronizing operations*. Examples of context synchronizing operations include the **sc** instruction, the **rfi/rfci** instructions, 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**, is not completed, until all instructions already in execution have completed to a point at which they have reported all exceptions they will cause.
3. The instructions that precede the operation will complete execution in the context (privilege, relocation, storage protection, etc.) in which they were initiated.
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 7.7, “Exception Priorities,” on page 7-15).
5. 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, which in turn requires that any effects and side effects of speculatively executing them also be discarded. The only side effects of these instructions that are permitted to survive are those specified in Section 6.2.4, “Performing Operations Out-of-Order,” on page 6-3.)

A context synchronizing operation is necessarily execution synchronizing; (see Section 11.2, “Execution Synchronization,” on page 11-2). Unlike the **sync** instruction (see

Section 2.7.1.2, “Synchronize (sync),” on page 2-14), a context synchronizing operation need not wait for storage-related operations to complete on other processors.

## 11.2 Execution Synchronization

An instruction is *execution synchronizing* if all previously initiated instructions appear to have completed before the instruction is initiated or, in the case of **sync** and **isync**, before the instruction completes. Examples of execution synchronizing instructions are **sync** (see the PowerPC User Instruction Set Architecture and the IBM PowerPC Embedded Virtual Environment Section 2.7.1.2, “Synchronize (sync),” on page 2-14) and **mtmsr**. Also, all context synchronizing instructions (see Section 11.1, “Context Synchronization,” on page 11-1) are execution synchronizing.

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

## 11.3 Synchronization Requirements

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

An instruction that alters the context in which data addresses or instruction addresses are interpreted, or in which instructions are executed, is called a “context-altering instruction.” This chapter covers all the context-altering instructions. The software synchronization required for them is shown in Table 11-1, “Data Access,” on p. 11-4 and Table 11-2, “Instruction Fetch And/Or Execution,” on p. 11-5.

The notation “CSI” in the tables means any context synchronizing instruction (i.e., **sc**, **isync**, **rftci** or **rfti**). A context synchronizing interrupt (that is, any interrupt except non-recoverable Machine Check) can be used instead of a context synchronizing instruction. If it is, phrases like “the synchronizing instruction,” below, should be interpreted as meaning the instruction at which the interrupt occurs. If no software synchronization is required before (after) a context-altering instruction, “the synchronizing instruction before (after) the context-altering instruction” should be interpreted as meaning the context-altering instruction itself.

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

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

### **Programming Note**

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

No software synchronization is required before altering the MSR (except perhaps when altering the WE or LE bits: see the tables), because **mtmsr** is execution synchronizing. No software synchronization is required before most of the other alterations shown in Table 11-2, because all instructions before the context-altering instruction are fetched and decoded before the context-altering instruction is executed (the processor must determine whether any of the preceding instructions are context synchronizing)

Table 11-1 below identifies the software synchronization requirements for data access for all context-altering instructions.

**Table 11-1. Data Access**

Context Altering Instruction or Event	Required Before	Required After	Notes
interrupt	none	none	
<b>rfi</b>	none	none	
<b>rfci</b>	none	none	
<b>sc</b>	none	none	
<b>mtmsr</b> (ILE)	none	none	
<b>mtmsr</b> (PR)	none	CSI	
<b>mtmsr</b> (ME)	none	CSI	1
<b>mtmsr</b> (DR)	none	CSI	
<b>mtmsr</b> (DE)	none	CSI	
<b>mtmsr</b> (LE)	—	—	3
<b>mtspr</b> (DAC)	—	—	5
<b>mtspr</b> (DBCR)	—	—	5
<b>mtspr</b> (DBSR)	—	—	5
<b>mtspr</b> (PID)	CSI	CSI	
<b>mtspr</b> (ZPR)	CSI	CSI	
<b>tlbia</b>	CSI	CSI or <b>sync</b>	6, 7
<b>tlbwe</b>	CSI	CSI or <b>sync</b>	6, 7

Table 11-2 below identifies the software synchronization requirements for instruction fetch and/or execution for all context-altering instructions.

**Table 11-2. Instruction Fetch And/Or Execution**

Context Altering Instruction or Event	Required Before	Required After	Notes
interrupt	none	none	
<b>rfi</b>	none	none	
<b>rfci</b>	none	none	
<b>sc</b>	none	none	
<b>mtmsr</b> (WE)	—	—	2
<b>mtmsr</b> (ILE)	none	none	
<b>mtmsr</b> (CE)	none	none	4
<b>mtmsr</b> (EE)	none	none	4
<b>wrtee</b>	none	none	4
<b>wrteei</b>	none	none	4
<b>mtmsr</b> (PR)	none	CSI	
<b>mtspr</b> (ZPR)	none	CSI	
<b>mtmsr</b> (FP)	none	CSI	
<b>mtmsr</b> (APA)	none	CSI	
<b>mtmsr</b> (ME)	none	CSI	1
<b>mtmsr</b> (FE0)	none	CSI	
<b>mtmsr</b> (FE1)	none	CSI	
<b>mtmsr</b> (APE)	none	CSI	
<b>mtmsr</b> (DE)	none	CSI	
<b>mtspr</b> (EVPR)	none	none	
<b>mtmsr</b> (IR)	none	CSI	8
<b>mtmsr</b> (LE)	—	—	3
<b>mtspr</b> (PID)	none	CSI	8
<b>mtspr</b> (PIT)	none	none	9
<b>mtspr</b> (TCR)	none	none	9
<b>mtspr</b> (TSR)	none	none	9
<b>mtspr</b> (IAC)	—	—	5
<b>mtspr</b> (DBCR)	—	—	5
<b>mtspr</b> (DBSR)	—	—	5
<b>tlbia</b>	none	CSI or sync	6, 7
<b>tlbwe</b>	none	CSI or sync	6, 7

## Notes for Tables 11-1 and 11-2

1. A context synchronizing instruction is required after altering MSR[ME] to ensure that the alteration takes effect for subsequent Machine Check interrupts, which may not be recoverable and therefore may not be context synchronizing.
2. Synchronization requirements for changing the Wait State Enable are implementation-dependent, and are specified in the User's Manual for the implementation.
3. Synchronization requirements for changing from one Endian mode to the other using the **mtmsr** instruction are implementation-dependent, and are specified in an implementation's User's Manual.
4. The effect of changing MSR[EE] on MSR[CE] is immediate.

If an **mtmsr**, **wrtee**, or **wrteei** instruction sets MSR[EE] to '0', an External Input, PIT or FIT interrupt does not occur after the instruction is executed.

If an **mtmsr**, **wrtee**, or **wrteei** instruction changes MSR[EE] from '0' to '1' when an External Input, PIT, FIT, or higher priority enabled exception exists, the corresponding interrupt occurs immediately after the **mtmsr**, **wrtee**, or **wrteei** is executed, and before the next instruction is executed in the program that set MSR[EE] to '1'.

If an **mtmsr** instruction sets MSR[CE] to '0', a Critical Input or WDT interrupt does not occur after the instruction is executed.

If an **mtmsr** instruction changes MSR[CE] from '0' to '1' when a Critical Input, WDT, or higher priority enabled exception exists, the corresponding interrupt occurs immediately after the **mtmsr** is executed, and before the next instruction is executed in the program that set MSR[CE] to '1'.

5. Synchronization requirements for changing any of the Debug Facility registers are implementation-dependent, and are specified in the User's Manual for the implementation.
6. For data accesses, the context synchronizing instruction before the **tlbwe** or **tlbia** instruction ensures that all storage accesses due to preceding instructions have completed to a point at which they have reported all exceptions they will cause.

The context synchronizing instruction after the **tlbwe** or **tlbia** ensures that subsequent storage accesses (data and instruction) will use the updated value in the TLB entry(s) being affected. It does *not* ensure that all storage accesses previously translated by the TLB entry(s) being updated have completed with respect to storage; if these completions must be ensured, the **tlbwe** or **tlbia** must be followed by a **sync** instruction as well as by a context synchronizing instruction.

### Programming Note

The following sequence illustrates why it is necessary, for data accesses, to ensure that all storage accesses due to instructions before the **tlbwe** or **tlbia** have completed to a point at which they have reported all exceptions they will

cause. Assume that valid TLB entries exist for the target storage location when the sequence starts.

1. A program issues a load or store to a page.
2. The same program executes a **tlbwe** or **tlbia** that invalidates the corresponding TLB entry.
3. The load or store instruction finally executes, and gets a TLB Miss exception.

The TLB Miss exception is semantically incorrect. In order to prevent it, a context synchronizing instruction must be executed between steps 1 and 2.

7. Multiprocessor systems have other requirements to synchronize “TLB shoot down” (i.e., to invalidate one or more TLB entries on all processors in the multiprocessor system and to be able to determine that the invalidations have completed and that all side effects of the invalidations have taken effect).
8. 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.
9. The elapsed time between the PIT reaching zero -- or the transition of the selected Time Base bit for the FIT or WDT -- and the signaling of the PIT, FIT, or WDT exception is not defined.





# 12

## Instruction Set Summary

### 12.1 Summary of Instructions

Table 12-1 provides a description of the IBM PowerPC Embedded Operating Environment Instructions which includes a mnemonic, operands, instruction, function and page reference for detailed information.

**Table 12-1. IBM PowerPC Embedded Operating Environment Instruction Set**

Mnemonic	Operands	Instruction	Function	Other Registers Changed	Page
<b>dcbi</b>	RA, RB	Data Cache Block Invalidate	Invalidate the data cache block which contains the effective address (RA 0) + (RB).		6-22
<b>icbt</b>	RA, RB	Instruction Cache Block Touch	Fetch the instruction cache block which contains the effective address (RA 0) + (RB).		6-23
<b>mfdcr</b>	RT, DCRN	Move From Device control Register	Move from DCR to RT, (RT) $\leftarrow$ (DCR(DCRN)).		5-8
<b>mfmsr</b>	RT	Move From Machine State Register	Move from MSR to RT, (RT) $\leftarrow$ (MSR).		7-44
<b>mf spr</b>	RT, SPRN	Move From Special Purpose Register	Move from SPR to RT, (RT) $\leftarrow$ (SPR(SPRN)).		5-6
<b>mtdcr</b>	DCRN, RS	Move To Device Control Register	Move to DCR from RS, (DCR(DCRN)) $\leftarrow$ (RS).		5-9
<b>mtmsr</b>	RS	Move To Machine State Register	Move to MSR from RS, (MSR) $\leftarrow$ (RS).		7-45

**Table 12-1. IBM PowerPC Embedded Operating Environment Instruction Set (cont.)**

Mnemonic	Operands	Instruction	Function	Other Registers Changed	Page
<b>mtspr</b>	SPRN, RS	Move To Special Purpose Register	Move to SPR from RS, (SPR(SPRN)) $\leftarrow$ (RS).		5-7
<b>rfci</b>		Return From Critical Interrupt	Return from critical interrupt, (PC) $\leftarrow$ (SRR2). (MSR) $\leftarrow$ (SRR3).		7-45
<b>rfi</b>		Return From Interrupt	Return from interrupt, (PC) $\leftarrow$ (SRR0). (MSR) $\leftarrow$ (SRR1).		7-46
<b>tlbia</b>		TLB Invalidate All	All of the entries in the TLB are invalidated and become unavailable for translation by clearing the valid (V) bit in the TLBHI portion of each TLB entry. The rest of the fields in the TLB entries are unmodified.		6-25
<b>tlbre</b>	RT, RA, WS	TLB Read Entry	If WS = 0: Load TLBHI portion of the selected TLB entry into RT. Load the PID register with the contents of the TID field of the selected TLB entry. (RT) $\leftarrow$ TLBHI[(RA)] (PID) $\leftarrow$ TLB[(RA)] <sub>TID</sub>  If WS = 1: Load TLBLO portion of the selected TLB entry into RT. (RT) $\leftarrow$ TLBLO[(RA)]		6-26
<b>tlbsx</b> <b>tlbsx.</b>	RT, RA, RB	TLB Search Indexed	Search the TLB array for a valid entry which translates the effective address EA = (RA)0 + (RB). If found, (RT) $\leftarrow$ Index of TLB entry. CR[CR0] <sub>EQ</sub> $\leftarrow$ 1 (if tlbsx.) If not found, (RT) $\leftarrow$ Undefined. CR[CR0] <sub>EQ</sub> $\leftarrow$ 1 (if tlbsx.)	CR[CR0] LT,GT,SO (if tlbsx.)	6-27

**Table 12-1. IBM PowerPC Embedded Operating Environment Instruction Set (cont.)**

Mnemonic	Operands	Instruction	Function	Other Registers Changed	Page
<b>tlbsync</b>		TLB Synchronize	<b>tlbsync</b> does not complete until all previous TLB-update instructions executed by this processor have been received and completed by all other processors.		6-28
<b>tlbwe</b>	RS, RA, WS	TLB Write Entry	<p>If WS = 0:  Write TLBHI portion of the selected TLB entry from RS.  Write the TID field of the selected TLB entry from the PID register.  <math>TLBHI[(RA)] \leftarrow (RS)</math>  <math>TLB[(RA)]_{TID} \leftarrow (PID)_{24:31}</math></p> <p>If WS = 1:  Write TLBLO portion of the selected TLB entry from RS.  <math>TLBLO[(RA)] \leftarrow (RS)</math></p>		6-28
<b>wrtee</b>	RS	Write External Enable	Write value of $(RS)_{16}$ to the External Enable bit (MSR[EE]).		7-48
<b>wrteei</b>	E	Write External Enable Immediate	Write value of E to the External Enable bit (MSR[EE]).		7-48

I





## Byte Ordering

---

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

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

For the PowerPC Architecture, as for most computer architectures currently implemented, the smallest addressable unit of storage is the 8-bit byte. Many scalars are halfwords, words, or doublewords, which consist of groups of bytes. When a word-length scalar is moved from a register to storage, the scalar occupies four consecutive byte addresses. It thus becomes meaningful to discuss the order of the byte addresses with respect to the value of the scalar: which byte contains the highest-order eight bits of the scalar, which byte contains the next-highest-order eight bits, and so on.

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

- The ordering that assigns the lowest address to the highest-order (“leftmost”) eight bits of the scalar, the next sequential address to the next-highest-order eight bits, and so on.

This ordering is called *Big Endian* because the “big end” of the scalar, considered as a binary number, comes first in storage. IBM RISC System/6000, IBM System/390, and Motorola 680x0 are examples of computers using this byte ordering.

- The ordering that assigns the lowest address to the lowest-order (“rightmost”) eight bits of the scalar, the next sequential address to the next-lowest-order eight bits, and so on.

This ordering is called *Little Endian* because the “little end” of the scalar, considered as a binary number, comes first in storage. DEC VAX and Intel x86 are examples of computers using this byte ordering.

## A.1 Structure Mapping Examples

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

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

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

### A.1.1 Big-Endian Mapping

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

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

## A.1.2 Little Endian Mapping

Structure *s* is shown mapped Little Endian.

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

## A.2 PowerPC Byte Ordering

By default, the PowerPC Architecture is Big Endian. This book describes the processor as if it operated only in a Big Endian fashion. In fact, the PowerPC Architecture and the IBM PowerPC Embedded Environment support Little Endian operation as well.

Two independent mechanisms support Little Endian operation. The first, defined by the PowerPC Architecture, provides Endian mode control using bits in the MSR. The second is an Endian storage attribute. It is defined by the IBM PowerPC Embedded Environment, and is not part of the PowerPC Architecture. Subsequent sections explain both mechanisms in more detail. For more information, see *The PowerPC Architecture: A Specification for a New Family of RISC Processors* and *The IBM PowerPC Embedded Environment*.

## A.3 PowerPC Endian Mode

PowerPC Endian mode is useful for system environments in which some processes (including the associated data structures) are written as Little Endian, and other processes are written as Big Endian. The PowerPC Endian mode mechanism handles such bi-Endian systems and manages communications and data sharing between processes running in the system. However, because of how PowerPC Endian mode operates, it does not provide for direct processor connections to Little Endian hardware, nor for operating a PowerPC processor in a hardware system environment that is connected in a Little Endian manner. Instead, for such environments, one should use the Endian (E) storage attribute described in Section A.4, “Endian Storage Attribute,” on p. A-9.

When a PowerPC processor operates with the PowerPC Endian mode set to Little Endian, the instructions and data in memory *appear*, from the programmer's point of view, to be

arranged in Little Endian format. However, this is not the case. Instead, instructions and data in memory are arranged in a unique order that is neither Big Endian nor Little Endian. In addition, the processor manipulates the low-order address bits used for all instruction fetches and data references such that, when combined with the unique ordering of the bytes in memory, the instructions and data appear to the executing program to be arranged in true Little Endian order. Section A.3.1 describes this unique byte arrangement and the address manipulation in detail, while Section A.3.2 explains how the PowerPC Endian mode is controlled.

### A.3.1 Byte Ordering in PowerPC Little Endian Mode

When the processor operates in PowerPC Little Endian mode, bytes, *in memory*, are rearranged from the order in which they would appear in a true Little Endian environment. Specifically, for each aligned doubleword (eight bytes) of memory, the eight bytes are reversed across the doubleword. For example, for the aligned doubleword at addresses A0–A7, the byte at A0 in Little Endian format is instead placed at A7 for PowerPC Little Endian mode. Likewise, the byte from A1 is moved to A6, A2 to A5, A3 to A4, A4 to A3, A5 to A2, A6 to A1, and A7 to A0. This is repeated for the next doubleword at addresses A8–A15, and so on.

Structure *s* (defined in Section A.1, “Structure Mapping Examples,” on p. A-2) would appear *in memory* as follows after being rearranged as described.

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

Note that this arrangement of bytes is neither Big Endian nor Little Endian, but is rather the result of taking the bytes from the Little Endian mapping and “swapping” them byte-for-byte across each doubleword. For this unique arrangement of bytes to appear to the executing program as equivalent to a true Little Endian arrangement, the address of each storage reference (whether for instruction fetches or for data accesses from load and store instructions) must be modified.



Specifically, the address of each storage access is modified by exclusive-ORing the low-order three bits of the address (addresses for instruction fetches are modified as for word accesses, because all PowerPC instructions are words).

Access Type	Address Modification
Byte	XOR with 0b111
Halfword	XOR with 0b110
Word	XOR with 0b100
Doubleword	No Modification

To see how this address modification, combined with the unique ordering of bytes in memory, results in the appearance to the executing program of a true Little Endian byte arrangement, consider the following example, using the value of the doubleword *b* from structure *s*. If *b* were stored, in Little Endian format, to address 08, it would appear as follows:

<b>28</b>	<b>27</b>	<b>26</b>	<b>25</b>	<b>24</b>	<b>23</b>	<b>22</b>	<b>21</b>
0x08	0x09	0x0A	0x0B	0x0C	0x0D	0x0E	0x0F

This memory could be accessed using doubleword, word, halfword, or byte accesses in a true Little Endian system with the following results:

Doubleword load from address 08	0x21222324_25262728
Word load from address 08	0x25262728
Word load from address 0C	0x21222324
Halfword load from address 08	0x2228
Halfword load from address 0A	0x2526
Halfword load from address 0C	0x2324
Halfword load from address 0E	0x2122
Byte load from address 08	0x28
Byte load from address 09	0x27
Byte load from address 0A	0x26
Byte load from address 0B	0x25
Byte load from address 0C	0x24
Byte load from address 0D	0x23
Byte load from address 0E	0x22
Byte load from address 0F	0x21

For programmers to view memory in PowerPC Little Endian mode as equivalent to memory in a true Little Endian system, the values observed for each kind of access must match those shown.

The following example shows how *b* is arranged in memory when stored, in PowerPC Little Endian mode, to address 08:

21	22	23	24	25	26	27	28
0x08	0x09	0x0A	0x0B	0x0C	0x0D	0x0E	0x0F

This doubleword could then be accessed using doubleword, word, halfword, or byte accesses in PowerPC Little Endian mode with the following results:

Doubleword load from (processor) address 08	converts to (memory) address 08	0x21222324_25262728
Word load from (processor) address 08	converts to (memory) address 0C	0x25262728
Word load from (processor) address 0C	converts to (memory) address 08	0x21222324
Halfword load from (processor) address 08	converts to (memory) address 0E	0x2728
Halfword load from (processor) address 0A	converts to (memory) address 0C	0x2526
Halfword load from (processor) address 0C	converts to (memory) address 0A	0x2324
Halfword load from (processor) address 0E	converts to (memory) address 08	0x2122
Byte load from (processor) address 08	converts to (memory) address 0F	0x28
Byte load from (processor) address 09	converts to (memory) address 0E	0x27
Byte load from (processor) address 0A	converts to (memory) address 0D	0x26
Byte load from (processor) address 0B	converts to (memory) address 0C	0x25
Byte load from (processor) address 0C	converts to (memory) address 0B	0x24
Byte load from (processor) address 0D	converts to (memory) address 0A	0x23
Byte load from (processor) address 0E	converts to (memory) address 09	0x22
Byte load from (processor) address 0F	converts to (memory) address 08	0x21

This example shows that a program, running on a PowerPC processor operating in PowerPC Little Endian mode, views doubleword *b* in memory as if it were instead arranged in true Little Endian format. Similar results are obtained for the other members of structure *s*.

It should be recognized that because *instructions* in PowerPC Architecture are defined as aligned words, their addressing is also affected by Endian mode. Specifically, each pair of words in an aligned doubleword of memory are reversed with respect to each other when operating in PowerPC Little Endian mode. So, even though both a Big Endian and a Little Endian program may have a sequence of instructions at, say, addresses 00, 04, 08, 12, and so on, and the executing program will request these instructions in order, the address modification causes the Little Endian program executing in PowerPC Little Endian mode to receive the instructions *from memory* in the following order of addresses: 04, 00, 12, 08, and so on.

Care must be taken when loading Little Endian programs into memory to ensure that the instructions are arranged in the proper order. See Section A.3.5, “Switching Endian Modes,” on p. A-9, for more detailed information.

### A.3.2 Control of PowerPC Endian Mode

The selection of the PowerPC Endian mode is controlled by two bits in the MSR: the Little Endian mode bit (MSR[LE]) and the Interrupt Little Endian bit (MSR[ILE]).

MSR[LE] describes the current Endian mode. If MSR[LE] = 1, the processor is executing in PowerPC Little Endian mode. Otherwise, the processor executes in Big Endian mode.

When a PowerPC processor takes an interrupt, the MSR contents are saved in either Save/Restore Register 1 (SRR1) or Save/Restore Register 3 (SRR3), depending on the interrupt type. The content of MSR[ILE] replaces the content of MSR[LE]. The processor can switch Endian modes in this fashion when entering an interrupt handler. The original value of MSR[LE] is restored from SRR1 or SRR3 upon leaving the interrupt handler (using an **rfi** or **rfci** instruction as appropriate) and returning to the previously executing program. Hence, the processor can also switch Endian modes when leaving an interrupt handler. This mode-switching capability enables an operating system written in one Endian mode to support application programs written in the other mode.

PowerPC processors reset to Big Endian mode, MSR[LE] = 0 and MSR[ILE] = 0.

### A.3.3 Addressing in PowerPC Little Endian Mode

The address modification performed in PowerPC Little Endian mode affects only those addresses that are presented to the storage subsystem (including the caches). Specifically, it does *not* affect the original calculation of addresses, nor the value of addresses saved in registers as part of the semantics of instruction execution.

For example, the following address values are calculated independently of Endian mode, and are stored in the appropriate registers without modification:

- The address placed into the LR by a branch with link update instruction, which is equal to the Program Counter (PC) + 4
- The offset in a relative branch instruction, which reflects the difference between the addresses of the branch and target instructions as they appear to the executing program (*not necessarily* as they appear in the actual memory arrangement)
- The address placed into RA by a load/store with update instruction, which is the value computed as described in the instruction description
- The address saved in system registers, such as SRR0, SRR2, and the DEAR, as computed by the executing program and as defined for these registers

These examples do not include all addresses that are not affected by the Little Endian address modification.

The cache management instructions (**dcbi**, **icbi**, and others) are unaffected by Endian mode, because the addresses used by these instructions refer to an entire cache block which is assumed to be larger than eight bytes, and thus the low-order three address bits are not used.

### A.3.4 Little Endian Mode Alignment Requirements

The “trick” of Exclusive OR-ing the low-order three bits of the address of an individual scalar does not work unless the scalar is aligned in memory to the size of the scalar. To illustrate, consider the following example of a word *w* (containing 0x1112\_1314) stored in memory at address 05, and arranged in Little Endian format:

					<b>14</b>	<b>13</b>	<b>12</b>
0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07
<b>11</b>							
0x08	0x09	0x0A	0x0B	0x0C	0x0D	0x0E	0x0F

In PowerPC Little Endian mode, word *w* would be arranged in memory as follows (remember that the bytes in each aligned doubleword are reversed in the format used by PowerPC Little Endian mode):

<b>12</b>	<b>13</b>	<b>14</b>					
0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07
							<b>11</b>
0x08	0x09	0x0A	0x0B	0x0C	0x0D	0x0E	0x0F

Note that the unaligned word *w* spans two doublewords. The two parts of the unaligned word are not contiguous in memory. Applying the address modification to a load word from address 0x05 would result in address 0x01, causing the four bytes from addresses 0x01, 0x02, 0x03, and 0x04 to be loaded — clearly an incorrect result. Instead, in order for the processor to properly load the four bytes from addresses 0x0F, 0x00, 0x01, and 0x02, an alternative, implementation-dependent mechanism must be used. Because of the complexity of dealing with this unusual arrangement of unaligned scalars when operating in PowerPC Little Endian mode, some PowerPC processors may generate Alignment exceptions when attempting to execute any unaligned scalar load or store instruction while in PowerPC Little Endian mode. Such an exception is not *required*, however. On the other hand, it *is* required that any string or multiple instruction (**lmw**, **lswi**, **lswx**, **stmw**, **stswi**, **stswx**) cause an Alignment exception when execution is attempted while in PowerPC Little Endian mode.

Note that although there are other conditions that can result in Alignment exceptions, the Alignment exceptions caused by those conditions occur regardless of Endian mode. See the

IBM PowerPC Embedded Operating Environment, Section 7.9.6, “Alignment Interrupt,” on p. 7-32 for more information.

### **A.3.5 Switching Endian Modes**

Because bytes in memory are arranged differently when operating in the different Endian modes, care must be taken, when switching modes, to convert programs and data structures to the new mode. The operating system must understand the differences in the two memory formats, and must reorder the bytes in memory, as appropriate, before dispatching a new process that accesses these memory structures in the new Endian mode.

For example, if a process executing in Big Endian mode creates a data structure, and a new process executing in Little Endian mode will access this data structure, the operating system must reverse the eight bytes within each aligned doubleword in the data structure before passing control to the new process.

### **A.3.6 Direct Memory Access in PowerPC Little Endian Mode**

Another aspect of the unique arrangement of bytes used by PowerPC Little Endian mode that must be considered is that of access to memory by devices other than the PowerPC processor. Because these other devices, such as a direct memory access (DMA) device or another non-PowerPC processor, are not likely to handle the special address modifications associated with PowerPC Little Endian mode, they must be aware of the special arrangement of bytes used by the PowerPC processor when it operates in Little Endian mode.

For example, if an I/O device loads a Little Endian program and data structure from a disk and places it into memory so that the PowerPC Processor can execute the program in Little Endian mode, the I/O device must reverse the eight bytes in each aligned doubleword after reading the data from the disk and before writing it to memory. Alternatively, the operating system running on the PowerPC processor must understand that the program image loaded from the disk was placed into memory in true Little Endian format, in which case the operating system must rearrange the bytes before executing the program.

## **A.4 Endian Storage Attribute**

The Endian storage attribute (E bit), defined in the IBM PowerPC Embedded Environment, also supports using a PowerPC processor in a Little Endian system. For every storage reference (instruction fetch or load/store access), an E bit is associated with the page or address region of the storage reference. The E bit specifies whether that page or region is organized as Big Endian (E = 0) or Little Endian (E = 1).

Unlike the organization of memory when using the PowerPC Little Endian mode, bytes in storage regions that are programmed as Little Endian using the E bit are arranged in true Little Endian format. Furthermore, no address modification is performed when accessing storage regions programmed as E = 1. Instead, when accessing storage regions with E = 1,

the PowerPC processor reorders the bytes as they are transferred between the processor and memory. Unlike PowerPC Little Endian mode, the E storage attribute supports direct connections of a PowerPC processor to Little Endian hardware and to memory containing Little Endian programs and data structures that may be shared with other Little Endian devices.

The on-the-fly reversal of bytes accessed in Little Endian storage regions is handled in one of two ways, depending on whether the storage access is an instruction fetch or a data (load/store) access. The following sections describe byte reversal for the two kinds of storage accesses.

#### A.4.1 Fetching Instructions from Little Endian Storage Regions

The PowerPC Architecture defines instructions as aligned words (four bytes) in memory. As such, instructions in a Big Endian program image are arranged with the most significant byte (MSB) of the instruction word at the lowest numbered address.

Consider the Big Endian mapping of instruction  $p$  at address 00, where, for example,  $p = \text{add } r7, r7, r4$ :

MSB			LSB
0x00	0x01	0x02	0x03

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

LSB			MSB
0x00	0x01	0x02	0x03

When an instruction is fetched from memory, the instruction must be placed in the pipeline in the proper order. Otherwise, the instruction decoder cannot recognize it. Because the PowerPC Architecture, by default, is Big Endian, the MSB of an instruction word is assumed to be at the lowest address. Therefore, when instructions are fetched from Little Endian storage regions, the four bytes of an instruction word must be reversed before the instruction is decoded. This reversal may occur between the memory interface and an instruction cache, or between the cache and the instruction decoder.

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

(or for Big Endian, no byte reversal) occurs in the event that this byte reversal is performed between the memory interface and the cache.

## A.4.2 Accessing Data in Little Endian Storage Regions

Unlike instruction fetches from Little Endian storage regions, data accesses from Little Endian storage regions can not be byte-reversed between memory and a data cache. Data byte ordering, in memory, depends on the data type (byte, halfword, or word) of a specific data item. It is only when moving a data item *of a specific type* from or to a GPR that it becomes known whether byte reversal is required due to the Endian format of the data item. Therefore, byte reversal during load/store accesses is performed between memory (or the data cache) and the GPR file, depending on whether the load/store was for a byte, halfword, or word.

Referring to the Big Endian and Little Endian mappings of structure *s*, as shown in Section A.1, “Structure Mapping Examples,” on p. A-2, the differences between the byte locations of any data item in the structure depends upon the size of the particular data item. For example (again referring to the Big Endian and Little Endian mappings of structure *s*):

- The word *a* has its four bytes reversed within the word spanning addresses 00–03.
- The halfword *e* has its two bytes reversed within the halfword spanning addresses 1C–1D.

Note that the array of bytes *d*, where each data item is a byte, is not reversed when the Big Endian and Little Endian mappings are compared. For example, the character 'A' is located at address 14 in both the Big Endian and Little Endian mappings.

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

When accessing data in a Little Endian storage region:

- For byte loads/stores, no reordering of bytes occurs.
- For halfword loads/stores, bytes are reversed within the halfword.
- For word loads/stores, bytes are reversed within the word.

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

For example, when loading a data word from a Little Endian storage region, all four bytes of the word are retrieved from memory (or the data cache). Then, the bytes are placed in the GPR so that the byte from the lowest address is placed in the LSB of the GPR.

In Little Endian storage regions, the alignment of data is treated as it is in Big Endian storage regions. Unlike PowerPC Little Endian mode, no special Alignment exceptions occur when accessing data in Little Endian storage regions. Note that the Alignment exceptions that apply to Big Endian region accesses also apply to Little Endian storage region accesses. See the IBM PowerPC Embedded Operating Environment,

Section 7.9.6, “Alignment Interrupt,” on p. 7-32 for detailed descriptions of conditions causing Alignment exceptions.

### **A.4.3 Control of the Endian Storage Attribute**

The control of the Endian (E) storage attribute, for a given access, depends upon whether the processor is operating with the associated MSR relocation bit on or off (MSR[IR] for instruction fetches and MSR[DR] for data accesses).

In real mode (MSR[IR] = 0 or MSR[DR] = 0), the E storage attribute for the access is controlled by the Storage Little Endian Register (SLER), a storage attribute control register similar to those controlling the other storage attributes in real mode, such as Caching Inhibited (I storage attribute), Write-Through Required (W storage attribute), and Guarded (G storage attribute).

The SLER is a 32-bit register that provides a separate E storage attribute for each 128MB address region in the 4GB real address space of the processor. The high-order five bits of the storage address select, from the SLER, the E storage attribute associated with the address region. Setting a bit to 1 in the SLER specifies that the associated storage region is Little Endian. See the IBM PowerPC Embedded Operating Environment Chapter 6, “Storage Control” on page 6-1 for additional discussion of storage attributes, the TLB, and the Real Mode Storage Attribute Control Registers.

### **A.4.4 PowerPC Byte-Reverse Instructions**

The PowerPC Architecture defines byte-reverse load/store instructions, which can perform a function similar to the action taken automatically by the PowerPC processor when it accesses data in Little Endian storage regions using the normal load/store instructions. However, the byte-reverse load/store instructions are not as generally useful as the Endian storage attribute mechanism.

For Big Endian storage regions, the normal (non-byte-reverse) load/store instructions operate as defined in the instruction descriptions, moving the more significant bytes of the register to and from the lower-numbered memory addresses. The load/store with byte-reverse instructions move the more significant bytes of the register to and from the higher numbered memory addresses.

The opposite is true for Little Endian storage regions, where the normal load/store instructions give the same results that load/store with byte-reverse instructions do in Big Endian storage regions. Load/store with byte-reverse instructions give the same results that normal load/store instructions do in Big Endian storage regions.

As Figures A-1 through A-4 illustrate, a normal store to a Big Endian storage region is the same as a byte-reverse store to a Little Endian storage region, while a normal store to a Little Endian storage region is the same as a byte-reverse store to a Big Endian storage region.



Figure A-1 illustrates the contents of a GPR and memory (starting at address 00) after a normal load/store in a Big Endian storage region.

MSB		LSB		
11	12	13	14	GPR
11	12	13	14	Memory
0x00	0x01	0x02	0x03	

**Figure A-1. Normal Word Load or Store (Big Endian Storage Region)**

Note that the results are identical to the results of a load/store with byte-reverse in a Little Endian storage region, as illustrated in Figure A-2.

MSB		LSB		
11	12	13	14	GPR
11	12	13	14	Memory
0x00	0x01	0x02	0x03	

**Figure A-2. Byte-reverse Word Load or Store (Little Endian Storage Region)**

Figure A-3 illustrates the contents of a GPR and memory (starting at address 00) after a load/store with byte-reverse in a Big Endian storage region.

MSB		LSB		GPR
11	12	13	14	

14	13	12	11	Memory
0x00	0x01	0x02	0x03	

**Figure A-3. Byte-reverse Word Load or Store (Big Endian Storage Region)**

Note that the results are identical to the results of a normal load/store in a Little Endian storage region, as illustrated in Figure A-4.

MSB		LSB		GPR
11	12	13	14	

14	13	12	11	Memory
0x00	0x01	0x02	0x03	

**Figure A-4. Normal Word Load or Store (Little Endian Storage Region)**

The E storage attribute is provided in the processor to augment the byte-reverse load/store instructions in two important ways:

- The load/store with byte-reverse instructions do not solve the problem of fetching instructions from a program image in true Little Endian format.  
Only the Endian storage attribute mechanism supports the fetching of true Little Endian program images.
- Typical compilers cannot make general use of the byte-reverse load/store instructions, so these instructions are ordinarily used only in special, hand-coded device drivers.  
Compilers can, however, take full advantage of the Endian storage attribute mechanism, enabling application programmers working in a high-level language, such as C, to compile programs and data structures into Little Endian format.

# Index

## A

- access
  - byte access 2-3
  - halfword access 2-3
  - single copy atomicity 2-3
  - word access 2-3
- access atomicity 2-19
- access control field 6-14
- access order 2-19
- access ordering 2-13
- address translation 6-10
  - TLB fields 6-13
    - access control 6-14
    - page identification 6-13
    - storage attribute 6-15
    - translation 6-14
  - translation lookaside buffer 6-11
  - virtual to real 6-10
- architecture 1-2
  - embedded environment 1-3
  - organization 1-2
- atomic access
  - byte access 2-3
  - floating-point access 2-3
  - halfword access 2-3
  - multiple-register loads 2-3
  - word access 2-3
- atomic update primitives 2-15
- auxiliary processor loads and stores 7-17

## B

- big endian mapping A-2
- branch instructions 7-18
- branch taken 9-5
- BT 9-5
- byte ordering A-1
  - accessing data A-11
  - addressing in PowerPC little endian mode A-7
  - control of PowerPC endian mode A-7
  - endian storage attribute A-9

- fetching instructions A-10
- little endian mode
  - alignment requirements A-8
- PowerPC byte ordering A-3
- PowerPC byte reverse instructions A-12
- PowerPC endian mode A-3
- PowerPC little endian mode A-4
- structure mapping A-2
  - big-endian mapping A-2
  - little endian mapping A-3
- switching endian modes A-9

## C

- cache
  - cache model 2-8
  - combined cache 2-12
  - dual cache 2-9
  - instruction
    - dcbf 2-12
    - dcbi 6-22
    - dcbst 2-11
    - dcbt 2-10
    - dcbtst 2-10
    - dcbz 2-11
    - icbi 2-9
    - icbt 6-23
  - split cache 2-9
  - write-through data cache 2-12
- cache instruction 6-19
- cache management instruction 6-22
- cache model 2-8
- caching 2-6, 6-6
- change recording 6-20
- combined cache 2-12
- context synchronization 11-1

## D

- DAC 9-3
- DACR 9-12
- data access 6-3, 11-4
- data address compare 9-3

- data address compare register 9-12
- data cache block flush 2-12, 2-24
- data cache block invalidate 6-22
- data cache block set to zero 2-11, 2-28
- data cache block store 2-11, 2-25
- data cache block touch 2-10, 2-26
- data cache block touch for store 2-10, 2-27
- data cache cacheability register 6-8
- data cache write-through register 6-8
- data exception address register 7-24
- data TLB miss exception 7-41
- DBCR 8-2, 9-9
- dbsr 9-11
- dcbf 2-12, 2-24
- dcbi 6-22
- dcbst 2-11, 2-25
- dcbt 2-10, 2-26
- dcbtst 2-10, 2-27
- dcbz 2-11, 2-28
- DCCR 6-8
- DCR 5-8
- DCR instruction
  - mfdcr 5-8
  - mtdcr 5-9
- DCWR 6-8
- DEAR 7-24
- debug
  - internal debug mode 9-1
- debug control register 8-2, 9-9
- debug events 9-2
  - branch taken 9-5
  - data address compare 9-3
  - exception 9-7
  - instruction address compare 9-3
  - instruction complete 9-6
  - trap 9-5
  - unconditional debug event 9-8
- debug registers 9-9
  - DACR 9-12
  - DBCR 9-9
  - dbsr 9-11
  - IACR 9-13
- debug status register 9-11
- device control register 5-8
- direct memory access A-9
- dual cache 2-9

## E

- eieio 2-14, 2-29
- embedded environment 1-3
  - operating 1-4
  - virtual 1-3
- endian storage attribute A-9
  - control A-12
- endianness 2-8, 6-7
- enforce in order execution of I/O 2-14, 2-29
- ESR 7-24
- evpr 7-26
- EX 6-16
- EXC 9-7
- exception 7-1, 7-41, 9-7
  - alignment exception 7-32
  - auxiliary processor enabled exception 7-36
  - auxiliary processor unavailable exception 7-36
  - critical input exception 7-27
  - data storage exception 7-28
  - debug exception 7-42
  - exception types 7-6
  - external input exception 7-31
  - fixed interval timer exception 7-39
  - floating-point enabled but unimplemented exception 7-36
  - floating-point enabled exception 7-35
  - floating-point unavailable exception 7-37
  - illegal instruction exception 7-34
  - instruction storage exception 7-30
  - instruction TLB miss exception 7-42
  - machine check exception 7-28
  - privileged instruction exception 7-35
  - program exception 7-33
  - programmable interval timer exception 7-38
  - system call exception 7-38
  - trap exception 7-35
  - watchdog timer exception 7-40
- exception priorities 7-15
  - auxiliary processor loads and stores 7-17
  - branch instructions 7-18
  - floating point loads and stores 7-16
  - illegal instructions 7-17
  - integer loads and stores 7-16
  - privileged instructions/privileged instructions 7-18

- system call instruction 7-18
- trap instructions 7-18
- exception syndrome register 7-24
- exception vector prefix register 7-26
- execution synchronization 11-2

## F

- FIT 10-7
- floating-point loads and stores 7-16
- forward progress 2-17
- freezing the timer 10-13

## G

- granularity 2-18
- guarded storage 2-7, 6-4, 6-7

## I

- IAC 9-3
- IACR 9-13
- icbi 2-9, 2-22
- icbt 6-23
- ICCR 6-8
- ICMP 9-6
- illegal instructions 7-17
- implicit branch 6-2
- initialization 8-1
- initialization code 8-4
- instruction
  - data cache 2-24
  - dcbf 2-12, 2-24
  - dcbi 6-22
  - dcbst 2-25
  - dcbt 2-10, 2-26
  - dcbtst 2-10, 2-27
  - dcbz 2-11, 2-28
  - dsbst 2-11
  - eleio 2-14, 2-29
  - formats xix
  - forms xix
  - icbi 2-9, 2-22
  - icbt 6-23
  - instruction cache 2-21
  - interrupt control 7-44
  - isync 2-23

- mfdcr 5-8
- mfmshr 7-44
- mfspr 5-6
- mftb 3-4
- mtdcr 5-9
- mtmsr 7-45
- mtspr 5-7
- partially executed 7-10
- rfci 7-45
- rfi 7-46
- sc 7-47
- summary 12-1
- sync 2-14
- tlbia 6-25
- tlbre 6-26
- tlbsx 6-27
- tlbsync 6-28
- tlbwe 6-28
- virtual environment 4-1
- wrttee 7-48
- wrtteei 7-48

- instruction address compare 9-3
- instruction address compare register 9-13
- instruction cache block invalidate 2-9, 2-22
- instruction cache block touch 6-23
- instruction cache cacheability register 6-8
- instruction complete 9-6
- instruction fetch 6-2, 11-5
- instruction restart 2-19
- instruction set summary 4-1
- instruction synchronize 2-23
- integer loads and stores 7-16
- interger unit 5-1
- internal debug mode 9-1
- interrupt 7-1
  - alignment interrupt 7-32
  - critical input interrupt 7-27
  - data storage interrupt 7-28
  - data TLB miss interrupt 7-41
  - debug interrupt 7-42
  - external input interrupt 7-31
  - fixed interval timer interrupt 7-39
  - floating-point unavailable interrupt 7-37
  - instruction
    - partially executed 7-10
  - instruction storage interrupt 7-30
  - instruction TLB miss interrupt 7-42

- interrupt types 7-6
- machine check interrupt 7-28
- masking 7-11
  - guidelines for system software 7-12
- ordering 7-11, 7-14
  - guidelines for system software 7-12
- program interrupt 7-33
  - auxiliary processor enabled exception 7-36
  - auxiliary processor unavailable exception 7-36
  - floating point enabled but unimplemented exception 7-36
  - floating-point enabled exception 7-35
  - illegal instruction exception 7-34
  - privileged instruction exception 7-35
  - trap exception 7-35
- programmable interval timer interrupt 7-38
- register setting
  - alignment interrupt 7-33
  - critical input interrupt 7-27
  - data storage interrupt 7-30
  - data TLB miss interrupt 7-41
  - debug interrupt 7-43
  - external input interrupt 7-32
  - fixed interval timer interrupt 7-40
  - floating-point unavailable interrupt interrupt 7-37
  - instruction storage interrupt 7-31
  - instruction TLB miss interrupt 7-42
  - machine check interrupt 7-28
  - program interrupt 7-34
  - programmable interval timer interrupt 7-39
  - system call interrupt 7-38
  - watchdog interrupt 7-41
- system call interrupt 7-38
- vector offset 7-6
- watchdog timer interrupt 7-40
- interrupt and exception handling registers 7-19
  - DEAR 7-24
  - ESR 7-24
  - evpr 7-26
  - MSR 7-19
  - SRR0 7-21
  - SRR1 7-21
  - SRR2 7-23

- SRR3 7-23
- interrupt classes 7-2
  - critical,non-critical 7-4
  - machine check 7-4
  - precise,imprecise 7-2
  - synchronous,asynchronous 7-2
- interrupt control instructions 7-44
  - mfmsr 7-44
  - mtmsr 7-45
  - rfci 7-45
  - rfi 7-46
  - sc 7-47
  - wrtee 7-48
  - wrteei 7-48
- interrupt processing 7-4
  - interrupt vector 7-4
- interrupt vector 7-4
- invalid real address 6-5
- isync 2-23

## L

- little endian mapping A-3
- little endian mode
  - alignment requirements A-8

## M

- machine state register 7-19
- memory coherence 2-4, 2-7, 6-6
  - coherence blocks 2-4
  - coherence not required mode 2-4
  - coherence required mode 2-4
- mfcdcr 5-8
- mfmsr 7-44
- mfsprr 5-6
- mftb 3-4
- mismatched WIMGE bits 6-7
- move from device control register 5-8
- move from machine state register 7-44
- move from special purpose register 5-6
- move from time base 3-4
- move to device control register 5-9
- move to machine state register 7-45
- move to special purpose register 5-7
- MSR 7-19
- mtcdcr 5-9

mtmsr 7-45  
mtspr 5-7

## O

oea 1-2  
operand  
    access atomicity 2-19  
    access order 2-19  
    instruction restart 2-19  
    placement 2-18  
operating environment  
    summary of instructions 12-1  
out-of-order access 6-4  
out-of-order operation 6-3

## P

page identification field 6-13  
page reference 6-20  
parameters 2-20  
partially executed instructions 7-10  
PID 6-16  
PIT 10-5  
PowerPC 1-2  
PowerPC byte ordering A-3  
PowerPC endian mode A-3  
    control A-7  
PowerPC little endian mode A-4  
    direct memory access A-9  
processor version register 5-5  
programmable interval timer 10-5  
PVR 5-5

## R

real address 6-10  
real addressing mode 6-5  
register  
    DACR 9-12  
    DBCR 8-2, 9-9  
    dbsr 9-11  
    DCCR 6-8  
    DCWR 6-8  
    DEAR 7-24  
    device control 5-8  
    ESR 7-24

evpr 7-26  
FIT 10-7  
IACR 9-13  
ICCR 6-8  
MSR 7-19  
PID 6-16  
PIT 10-5  
PVR 5-5  
SGR 6-8  
SLER 6-8  
SMR 6-8  
SPR 5-1  
SPR instruction 5-6  
SPRG0 5-4  
SPRG1 5-4  
SPRG2 5-4  
SPRG3 5-4  
SRR0 7-21  
SRR1 7-21  
SRR2 7-23  
SRR3 7-23  
TB 3-1, 10-3  
TBL 3-2  
TBU 3-2  
TCR 8-2, 10-11  
TSR 10-12  
WDT 10-8  
ZPR 6-17

### register setting

alignment interrupt 7-33  
critical input interrupt 7-27  
data storage interrupt 7-30  
data TLB miss interrupt 7-41  
debug interrupt 7-43  
external input interrupt 7-32  
fixed interval timer interrupt 7-40  
floating-point unavailable interrupt 7-37  
instruction storage interrupt 7-31  
instruction TLB miss interrupt 7-42  
machine check interrupt 7-28  
program interrupt 7-34  
programmable interval timer interrupt 7-39  
system call interrupt 7-38  
watchdog interrupt 7-41

### reservations 2-16

reset 8-1  
    initialization code example 8-4

- processor state after reset 8-2
- software initialization 8-4
- reset mechanism 8-1
  - DBCR 8-2
  - TCR 8-2
- return from critical interrupt 7-45
- return from interrupt 7-46
- rfci 7-45
- rfi 7-46
- fixed interval timer 10-7

## S

- save restore register 7-21, 7-23
- sc 7-47
- SGR 6-8
- shared storage 2-13
- single copy atomicity 2-3
- SLER 6-8
- SMR 6-8
- software initialization 8-4
- special purpose register 5-1
- special purpose register general 5-4
- split cache 2-9
- SPR 5-1
- SPR instruction 5-6
  - mf spr 5-6
  - mt spr 5-7
- SPRG0 5-4
- SPRG1 5-4
- SPRG2 5-4
- SPRG3 5-4
- SRR0 7-21
- SRR1 7-21
- SRR2 7-23
- SRR3 7-23
- storage
  - access ordering 2-13
  - addressing 6-1
  - atomic update primitives 2-15
    - forward progress 2-17
    - granularity 2-18
    - reservations 2-16
  - data access 6-3
  - guarded 6-4
  - implicit branch 6-2
  - instruction

- eieio 2-14
- sync 2-14
- instruction fetch 6-2
- invalid real address 6-5
- out-of-order access 6-4
- out-of-order operation 6-3
- real addressing mode 6-5
- shared storage 2-13
- virtual storage 2-2
- storage attribute control register
  - DCCR 6-8
  - DCWR 6-8
  - ICCR 6-8
  - SGR 6-8
  - SLER 6-8
  - SMR 6-8
- storage attribute field 6-15
- storage control
  - data cache instruction 2-24
    - dcbf 2-24
    - dcbst 2-25
    - dcbt 2-26
    - dcbtst 2-27
    - dcbz 2-28
  - instruction
    - eieio 2-29
  - instruction cache instruction 2-21
    - icbi 2-22
    - isync 2-23
  - parameters 2-20
- storage control attribute 2-5
  - caching 2-6
  - endianness 2-8
  - guarded storage 2-7
  - memory coherence 2-7
  - write-through 2-6
- storage control attributes 6-6
  - caching 6-6
  - endianness 6-7
  - guarded 6-7
  - memory coherence 6-6
  - mismatched WIMGE bits 6-7
  - write-through 6-6
- storage control instruction 2-20, 6-22
- storage guarded register 6-8
- storage little endian register 6-8
- storage memory-coherent register 6-8



- storage model 2-1, 6-1
- storage protection 6-10, 6-15
  - cache instruction 6-19
  - change recording 6-20
  - execute enable bit 6-16
  - page reference 6-20
  - process ID 6-16
  - string instruction 6-20
  - TLB management 6-21
  - translation ID 6-16
  - write enable bit 6-17
  - zone select
    - storage protection
    - zone protection 6-17
- string instruction 6-20
- structure mapping A-2
- switching endian modes A-9
- sync 2-14
- synchronization
  - context synchronization 11-1
  - data access 11-4
  - execution 11-5
  - execution synchronization 11-2
  - instruction fetch 11-5
  - requirements 11-2
- synchronize 2-14
- system call 7-47
- system call instruction 7-18

## T

- TB 3-1, 10-3
- TBL 3-2
- TBU 3-2
- TCR 8-2, 10-11
- TID 6-16
- time base 3-1, 10-3
  - computing time of day 3-6
  - extended mnemonics 3-5
  - instruction
    - mftb 3-4
  - TBL 3-2
  - TBU 3-2
  - update frequency 3-8
- timer 3-1, 10-2
  - FIT 10-7
  - freezing the timer 10-13

- PIT 10-5
- TB 10-3
- TBL 3-2
- TBU 3-2
- TCR 10-11
- TSR 10-12
- WDT 10-8
  - writing the time base 10-4
- timer control register 8-2, 10-11
- timer status register 10-12
- TLB 6-10, 6-11
  - instruction
    - tlbia 6-25
    - tlbre 6-26
    - tlbsx 6-27
    - tlbsync 6-28
    - tlbwe 6-28
- TLB fields 6-13
  - page identification 6-13
  - translation 6-14
- TLB invalidate all 6-25
- TLB management 6-21
- TLB management instruction 6-25
- TLB read entry 6-26
- TLB search indexed 6-27
- TLB synchronize 6-28
- TLB write entry 6-28
- tlbia 6-25
- tlbre 6-26
- tlbsx 6-27
- tlbsync 6-28
- tlbwe 6-28
- TR 9-5
- translation field 6-14
- trap 9-5
- trap instructions 7-18
- TSR 10-12

## U

- UDE 9-8
- unconditional debug event 9-8
- user instruction set architecture 1-2

## V

- virtual address 6-10

- virtual environment
  - instruction set summary 4-1
- virtual environment architecture 1-2
- virtual storage 2-2

## **W**

- watchdog timer 10-8
- WDT 10-8
- WIMGE 6-6
- WR 6-17
- write external enable 7-48
- write external enable immediate 7-48
- write-through 2-6, 6-6
- write-through data cache 2-12
- writing the time base 10-4
- wrtee 7-48
- wrteei 7-48

## **Z**

- ZPR 6-17
- ZSEL 6-17