

# Chapter 2

## PowerPC Register Set

This chapter describes the register organization defined by the three levels of the PowerPC architecture—*user instruction set architecture (UISA)*, *virtual environment architecture (VEA)*, and *operating environment architecture (OEA)*. The PowerPC architecture defines register-to-register operations for all computational instructions. Source data for these instructions are accessed from the on-chip registers or are provided as immediate values embedded in the opcode. The three-register instruction format allows specification of a target register distinct from the two source registers, thus preserving the original data for use by other instructions and reducing the number of instructions required for certain operations. Data is transferred between memory and registers with explicit load and store instructions only.



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

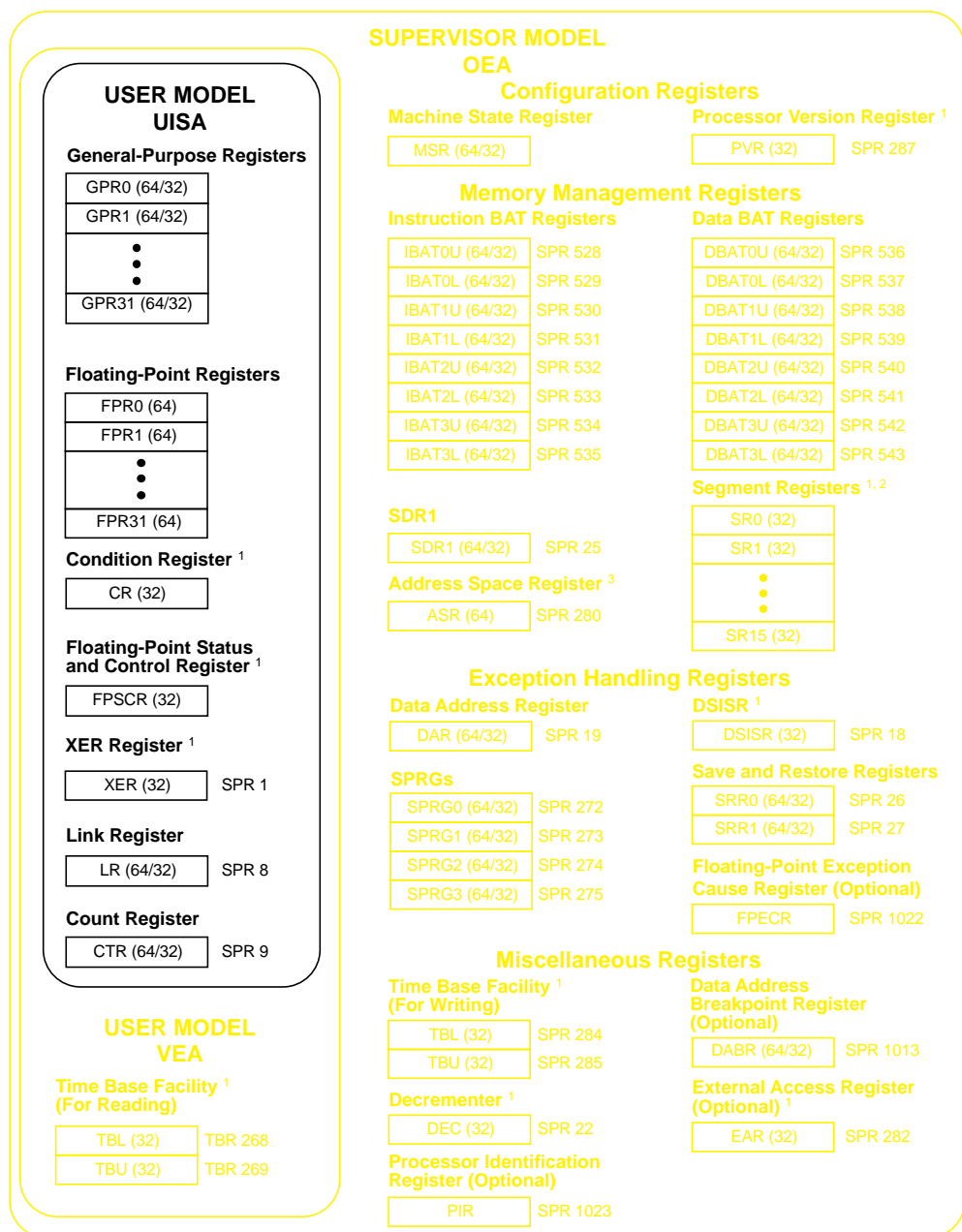
### 2.1 PowerPC UISA Register Set

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



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

Note that the general-purpose registers (GPRs), link register (LR), and count register (CTR) are 64 bits wide on 64-bit implementations and 32 bits wide on 32-bit implementations.



<sup>1</sup> These registers are 32-bit registers only.

<sup>2</sup> These registers are on 32-bit implementations only.

<sup>3</sup> These registers are on 64-bit implementations only.

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

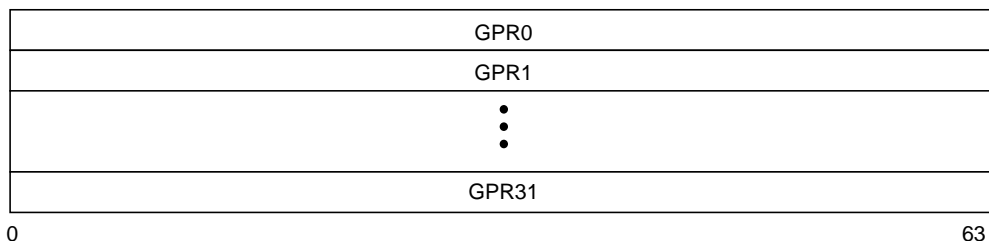


The user-level registers can be accessed by all software with either user or supervisor privileges. The user-level register set includes the following:

- **General-purpose registers (GPRs).** The general-purpose register file consists of 32 GPRs designated as GPR0–GPR31. The GPRs serve as data source or destination registers for all integer instructions and provide data for generating addresses. See Section 2.1.1, “General-Purpose Registers (GPRs),” for more information.
- **Floating-point registers (FPRs).** The floating-point register file consists of 32 FPRs designated as FPR0–FPR31; these registers serve as the data source or destination for all floating-point instructions. While the floating-point model includes data objects of either single- or double-precision floating-point format, the FPRs only contain data in double-precision format. For more information, see Section 2.1.2, “Floating-Point Registers (FPRs).”
- **Condition register (CR).** The CR is a 32-bit register, divided into eight 4-bit fields, CR0–CR7, that reflects the results of certain arithmetic operations and provides a mechanism for testing and branching. For more information, see Section 2.1.3, “Condition Register (CR).”
- **Floating-point status and control register (FPSCR).** The FPSCR contains all floating-point exception signal bits, exception summary bits, exception enable bits, and rounding control bits needed for compliance with the IEEE 754 standard. For more information, see Section 2.1.4, “Floating-Point Status and Control Register (FPSCR).” (Note that the architecture specification refers to exceptions as interrupts.)
- **XER register (XER).** The XER indicates overflows and carry conditions for integer operations and the number of bytes to be transferred by the load/store string indexed instructions. For more information, see Section 2.1.5, “XER Register (XER).”
- **Link register (LR).** The LR provides the branch target address for the Branch Conditional to Link Register (**bclrx**) instructions, and can optionally be used to hold the effective address of the instruction that follows a “branch with link update” instruction in the instruction stream, typically used for loading the return pointer for a subroutine. For more information, see Section 2.1.6, “Link Register (LR).”
- **Count register (CTR).** The CTR holds a loop count that can be decremented during execution of appropriately coded branch instructions. The CTR can also provide the branch target address for the Branch Conditional to Count Register (**bcctrx**) instructions. For more information, see Section 2.1.7, “Count Register (CTR).”

### 2.1.1 General-Purpose Registers (GPRs)

Integer data is manipulated in the processor’s 32 GPRs shown in Figure 2-2. These registers are 64-bit registers in 64-bit implementations and 32-bit registers in 32-bit implementations. The GPRs are accessed as source and destination registers in the instruction syntax.



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

## 2.1.2 Floating-Point Registers (FPRs)

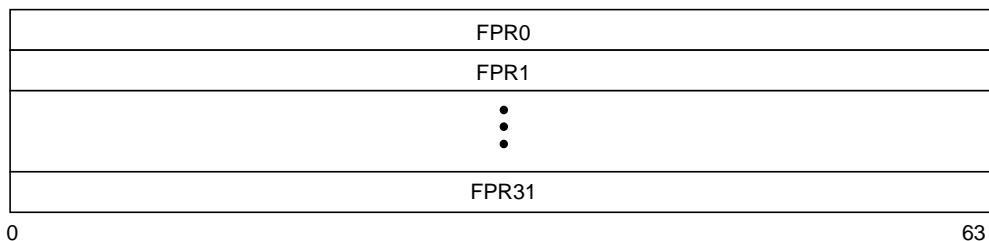
The PowerPC architecture provides **thirty-two 64-bit FPRs** as shown in Figure 2-3. These registers are accessed as source and destination registers for floating-point instructions. Each FPR supports the **double-precision floating-point format**. Every instruction that interprets the contents of an FPR as a floating-point value uses the double-precision floating-point format for this interpretation. Note that FPRs are 64 bits on both 64-bit and 32-bit processor implementations.

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

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

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

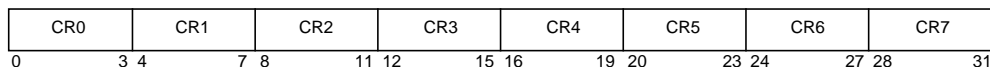
The floating-point arithmetic instructions produce intermediate results that may be regarded as infinitely precise and with unbounded *exponent* range. This intermediate result is *normalized* or *denormalized* if required, and then rounded to the destination format. The final result is then placed into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



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

### 2.1.3 Condition Register (CR)

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



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

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

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

Note that branch instructions are provided to test individual CR bits.

### 2.1.3.1 Condition Register CR0 Field Definition

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

The CR bits are interpreted as shown in Table 2-1. If any portion of the result is undefined, the value placed into the first three bits of CR0 is undefined.

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

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

Note that CR0 may not reflect the “true” (that is, infinitely precise) result if overflow occurs. Also, CR0 bits 0–2 are undefined if  $R_c = 1$  for the **mulhw**, **mulhww**, **divw**, and **divww** instructions in 64-bit mode.

### 2.1.3.2 Condition Register CR1 Field Definition

In all floating-point instructions when the CR is set to reflect the result of the operation (that is, when the instruction’s record bit,  $R_c$ , is set), CR1 (bits 4–7 of the CR) is copied from bits 0–3 of the FPSCR and indicates the floating-point exception status. For more information about the FPSCR, see Section 2.1.4, “Floating-Point Status and Control Register (FPSCR).” The bit settings for the CR1 field are shown in Table 2-2.

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

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

### 2.1.3.3 Condition Register CR<sub>n</sub> Field—Compare Instruction

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

**Table 2-3. CR<sub>n</sub> Field Bit Settings for Compare Instructions**

CR <sub>n</sub> Bit <sup>1</sup>	Description <sup>2</sup>
0	Less than or floating-point less than (LT, FL). For integer compare instructions: $rA < \text{SIMM or } rB$ (signed comparison) or $rA < \text{UIMM or } rB$ (unsigned comparison). For floating-point compare instructions: $frA < frB$ .
1	Greater than or floating-point greater than (GT, FG). For integer compare instructions: $rA > \text{SIMM or } rB$ (signed comparison) or $rA > \text{UIMM or } rB$ (unsigned comparison). For floating-point compare instructions: $frA > frB$ .
2	Equal or floating-point equal (EQ, FE). For integer compare instructions: $rA = \text{SIMM, UIMM, or } rB$ . For floating-point compare instructions: $frA = frB$ .
3	Summary overflow or floating-point unordered (SO, FU). For integer compare instructions: This is a copy of the final state of XER[SO] at the completion of the instruction. For floating-point compare instructions: One or both of $frA$ and $frB$ is a Not a Number (NaN).

**Notes:**<sup>1</sup>Here, the bit indicates the bit number in any one of the 4-bit subfields, CR0–CR7.

<sup>2</sup>For a complete description of instruction syntax conventions, refer to Table 8-2 on page 8-2.

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

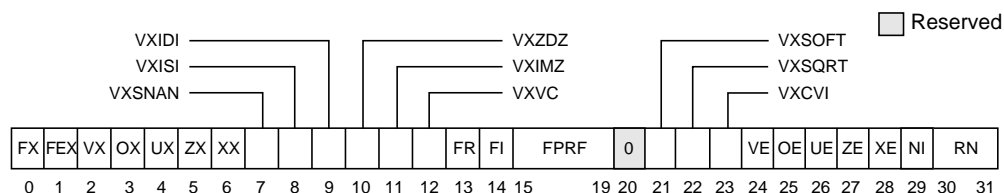
The FPSCR, shown in Figure 2-5, contains bits that do the following:

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

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

Except for the floating-point enabled exception summary (FEX) and floating-point invalid operation exception summary (VX), the exception condition bits in the FPSCR (bits 0–12 and 21–23) are sticky. Once set, *sticky bits* remain set until they are *cleared* by an **mcrfs**, **mtfsfi**, **mtfsf**, or **mtfsb0** instruction.

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



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

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

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

Bit(s)	Name	Description
0	FX	Floating-point exception summary. Every floating-point instruction, except <b>mtfsfi</b> and <b>mtfsf</b> , implicitly sets FPSCR[FX] if that instruction causes any of the floating-point exception bits in the FPSCR to transition from 0 to 1. The <b>mcrfs</b> , <b>mtfsfi</b> , <b>mtfsf</b> , <b>mtfsb0</b> , and <b>mtfsb1</b> instructions can alter FPSCR[FX] explicitly. This is a sticky bit.
1	FEX	Floating-point enabled exception summary. This bit signals the occurrence of any of the enabled exception conditions. It is the logical OR of all the floating-point exception bits masked by their respective enable bits ( $FEX = (VX \& VE) \wedge (OX \& OE) \wedge (UX \& UE) \wedge (ZX \& ZE) \wedge (XX \& XE)$ ). The <b>mcrfs</b> , <b>mtfsf</b> , <b>mtfsfi</b> , <b>mtfsb0</b> , and <b>mtfsb1</b> instructions cannot alter FPSCR[FEX] explicitly. This is not a sticky bit.
2	VX	Floating-point invalid operation exception summary. This bit signals the occurrence of any invalid operation exception. It is the logical OR of all of the invalid operation exceptions. The <b>mcrfs</b> , <b>mtfsf</b> , <b>mtfsfi</b> , <b>mtfsb0</b> , and <b>mtfsb1</b> instructions cannot alter FPSCR[VX] explicitly. This is not a sticky bit.
3	OX	Floating-point overflow exception. This is a sticky bit. See Section 3.3.6.2, "Overflow, Underflow, and Inexact Exception Conditions."
4	UX	Floating-point <i>underflow</i> exception. This is a sticky bit. See Section 3.3.6.2.2, "Underflow Exception Condition."
5	ZX	Floating-point zero divide exception. This is a sticky bit. See Section 3.3.6.1.2, "Zero Divide Exception Condition."
6	XX	Floating-point inexact exception. This is a sticky bit. See Section 3.3.6.2.3, "Inexact Exception Condition." FPSCR[XX] is the sticky version of FPSCR[FI]. The following rules describe how FPSCR[XX] is set by a given instruction: <ul style="list-style-type: none"> <li>If the instruction affects FPSCR[FI], the new value of FPSCR[XX] is obtained by logically ORing the old value of FPSCR[XX] with the new value of FPSCR[FI].</li> <li>If the instruction does not affect FPSCR[FI], the value of FPSCR[XX] is unchanged.</li> </ul>
7	VXSNAN	Floating-point invalid operation exception for SNaN. This is a sticky bit. See Section 3.3.6.1.1, "Invalid Operation Exception Condition."
8	VXISI	Floating-point invalid operation exception for $\infty - \infty$ . This is a sticky bit. See Section 3.3.6.1.1, "Invalid Operation Exception Condition."
9	VXIDI	Floating-point invalid operation exception for $\infty + \infty$ . This is a sticky bit. See Section 3.3.6.1.1, "Invalid Operation Exception Condition."
10	VXZDZ	Floating-point invalid operation exception for $0 \div 0$ . This is a sticky bit. See Section 3.3.6.1.1, "Invalid Operation Exception Condition."



**Table 2-4. FPSCR Bit Settings (Continued)**

Bit(s)	Name	Description
11	VXIMZ	Floating-point invalid operation exception for $\infty * 0$ . This is a sticky bit. See Section 3.3.6.1.1, "Invalid Operation Exception Condition."
12	VXVC	Floating-point invalid operation exception for invalid compare. This is a sticky bit. See Section 3.3.6.1.1, "Invalid Operation Exception Condition."
13	FR	Floating-point <i>fraction</i> rounded. The last arithmetic or rounding and conversion instruction that rounded the intermediate result incremented the fraction. See Section 3.3.5, "Rounding." This bit is not sticky.
14	FI	Floating-point fraction inexact. The last arithmetic or rounding and conversion instruction either rounded the intermediate result (producing an inexact fraction) or caused a disabled overflow exception. See Section 3.3.5, "Rounding." This is not a sticky bit. For more information regarding the relationship between FPSCR[FI] and FPSCR[XX], see the description of the FPSCR[XX] bit.
15–19	FPRF	<p>Floating-point result flags. For arithmetic, rounding, and conversion instructions, the field is based on the result placed into the target register, except that if any portion of the result is undefined, the value placed here is undefined.</p> <p>15 Floating-point result class descriptor (C). Arithmetic, rounding, and conversion instructions may set this bit with the FPCC bits to indicate the class of the result as shown in Table 2-5.</p> <p>16–19 Floating-point condition code (FPCC). Floating-point compare instructions always set one of the FPCC bits to one and the other three FPCC bits to zero. Arithmetic, rounding, and conversion instructions may set the FPCC bits with the C bit to indicate the class of the result. Note that in this case the high-order three bits of the FPCC retain their relational significance indicating that the value is less than, greater than, or equal to zero.</p> <p>16 Floating-point less than or negative (FL or &lt;)</p> <p>17 Floating-point greater than or positive (FG or &gt;)</p> <p>18 Floating-point equal or zero (FE or =)</p> <p>19 Floating-point unordered or NaN (FU or ?)</p> <p>Note that these are not sticky bits.</p>
20	—	Reserved
21	VXSOFT	Floating-point invalid operation exception for software request. This is a sticky bit. This bit can be altered only by the <b>mcrfs</b> , <b>mtfsfi</b> , <b>mtfsf</b> , <b>mtfsb0</b> , or <b>mtfsb1</b> instructions. For more detailed information, refer to Section 3.3.6.1.1, "Invalid Operation Exception Condition."
22	VXSQRT	Floating-point invalid operation exception for invalid square root. This is a sticky bit. For more detailed information, refer to Section 3.3.6.1.1, "Invalid Operation Exception Condition."
23	VXCVI	Floating-point invalid operation exception for invalid integer convert. This is a sticky bit. See Section 3.3.6.1.1, "Invalid Operation Exception Condition."
24	VE	Floating-point invalid operation exception enable. See Section 3.3.6.1.1, "Invalid Operation Exception Condition."
25	OE	IEEE floating-point overflow exception enable. See Section 3.3.6.2, "Overflow, Underflow, and Inexact Exception Conditions."
26	UE	IEEE floating-point underflow exception enable. See Section 3.3.6.2.2, "Underflow Exception Condition."
27	ZE	IEEE floating-point zero divide exception enable. See Section 3.3.6.1.2, "Zero Divide Exception Condition."
28	XE	Floating-point inexact exception enable. See Section 3.3.6.2.3, "Inexact Exception Condition."

**Table 2-4. FPSCR Bit Settings (Continued)**

Bit(s)	Name	Description
29	NI	Floating-point non-IEEE mode. If this bit is set, results need not conform with IEEE standards and the other FPSCR bits may have meanings other than those described here. If the bit is set and if all <i>implementation-specific</i> requirements are met and if an IEEE-conforming result of a floating-point operation would be a <i>denormalized number</i> , the result produced is zero (retaining the sign of the denormalized number). Any other effects associated with setting this bit are described in the user's manual for the implementation (the effects are implementation-dependent).
30–31	RN	Floating-point rounding control. See Section 3.3.5, “Rounding.” 00 Round to nearest 01 Round toward zero 10 Round toward +infinity 11 Round toward –infinity

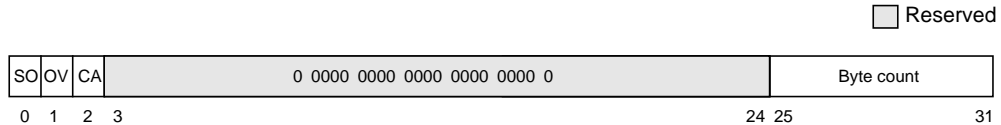
Table 2-5 illustrates the floating-point result flags used by PowerPC processors. The result flags correspond to FPSCR bits 15–19.

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

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

## 2.1.5 XER Register (XER)

The XER register (XER) is a 32-bit, user-level register shown in Figure 2-6.



**Figure 2-6. XER Register**

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

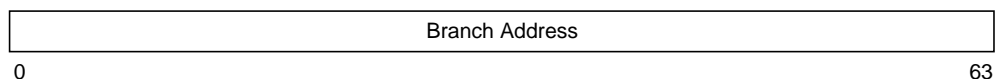
**Table 2-6. XER Bit Definitions**

Bit(s)	Name	Description
0	SO	Summary overflow. The summary overflow bit (SO) is set whenever an instruction (except <b>mtspr</b> ) sets the overflow bit (OV). Once set, the SO bit remains set until it is cleared by an <b>mtspr</b> instruction (specifying the XER) or an <b>mcrxr</b> instruction. It is not altered by compare instructions, nor by other instructions (except <b>mtspr</b> to the XER, and <b>mcrxr</b> ) that cannot overflow. Executing an <b>mtspr</b> instruction to the XER, supplying the values zero for SO and one for OV, causes SO to be cleared and OV to be set.
1	OV	Overflow. The overflow bit (OV) is set to indicate that an overflow has occurred during execution of an instruction. Add, subtract from, and negate instructions having OE = 1 set the OV bit if the carry out of the msb is not equal to the carry out of the msb + 1, and clear it otherwise. Multiply low and divide instructions having OE = 1 set the OV bit if the result cannot be represented in 64 bits ( <b>mullw</b> , <b>divw</b> , <b>divdu</b> ) or in 32 bits ( <b>mullw</b> , <b>divw</b> , <b>divwu</b> ), and clear it otherwise. The OV bit is not altered by compare instructions that cannot overflow (except <b>mtspr</b> to the XER, and <b>mcrxr</b> ).
2	CA	Carry. The carry bit (CA) is set during execution of the following instructions: <ul style="list-style-type: none"> <li>Add carrying, subtract from carrying, add extended, and subtract from extended instructions set CA if there is a carry out of the msb, and clear it otherwise.</li> <li>Shift right algebraic instructions set CA if any 1 bits have been shifted out of a negative operand, and clear it otherwise.</li> </ul> The CA bit is not altered by compare instructions, nor by other instructions that cannot carry (except shift right algebraic, <b>mtspr</b> to the XER, and <b>mcrxr</b> ).
3–24	—	Reserved
25–31		This field specifies the number of bytes to be transferred by a Load String Word Indexed ( <b>lswx</b> ) or Store String Word Indexed ( <b>stswx</b> ) instruction.

## 2.1.6 Link Register (LR)

The link register (LR) is a 64-bit register in 64-bit implementations and a 32-bit register in 32-bit implementations. The LR supplies the branch target address for the Branch Conditional to Link Register (**bclrx**) instructions, and in the case of a branch with link update instruction, can be used to hold the logical address of the instruction that follows the

branch with link update instruction (for returning from a subroutine). The format of LR is shown in Figure 2-7.



**Figure 2-7. Link Register (LR)**

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

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

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

### 2.1.7 Count Register (CTR)

The count register (CTR) is a 64-bit register in 64-bit implementations and a 32-bit register in 32-bit implementations. The CTR can hold a loop count that can be decremented during execution of branch instructions that contain an appropriately coded BO field. If the value in CTR is 0 before being decremented, it is 0xFFFF\_FFFF\_FFFF\_FFFF ( $2^{64}-1$ ) afterward in 64-bit implementations and 0xFFFF\_FFFF ( $2^{32}-1$ ) in 32-bit implementations. The CTR can also provide the branch target address for the Branch Conditional to Count Register (**bctrx**) instruction. The CTR is shown in Figure 2-8.



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

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

The count register can also be accessed by the **mtspr** and **mfspr** instructions by specifying SPR 9. In branch conditional instructions, the BO field specifies the conditions under which

the branch is taken. The first four bits of the BO field specify how the branch is affected by or affects the CR and the CTR. The encoding for the BO field is shown in Table 2-7.

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

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

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

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

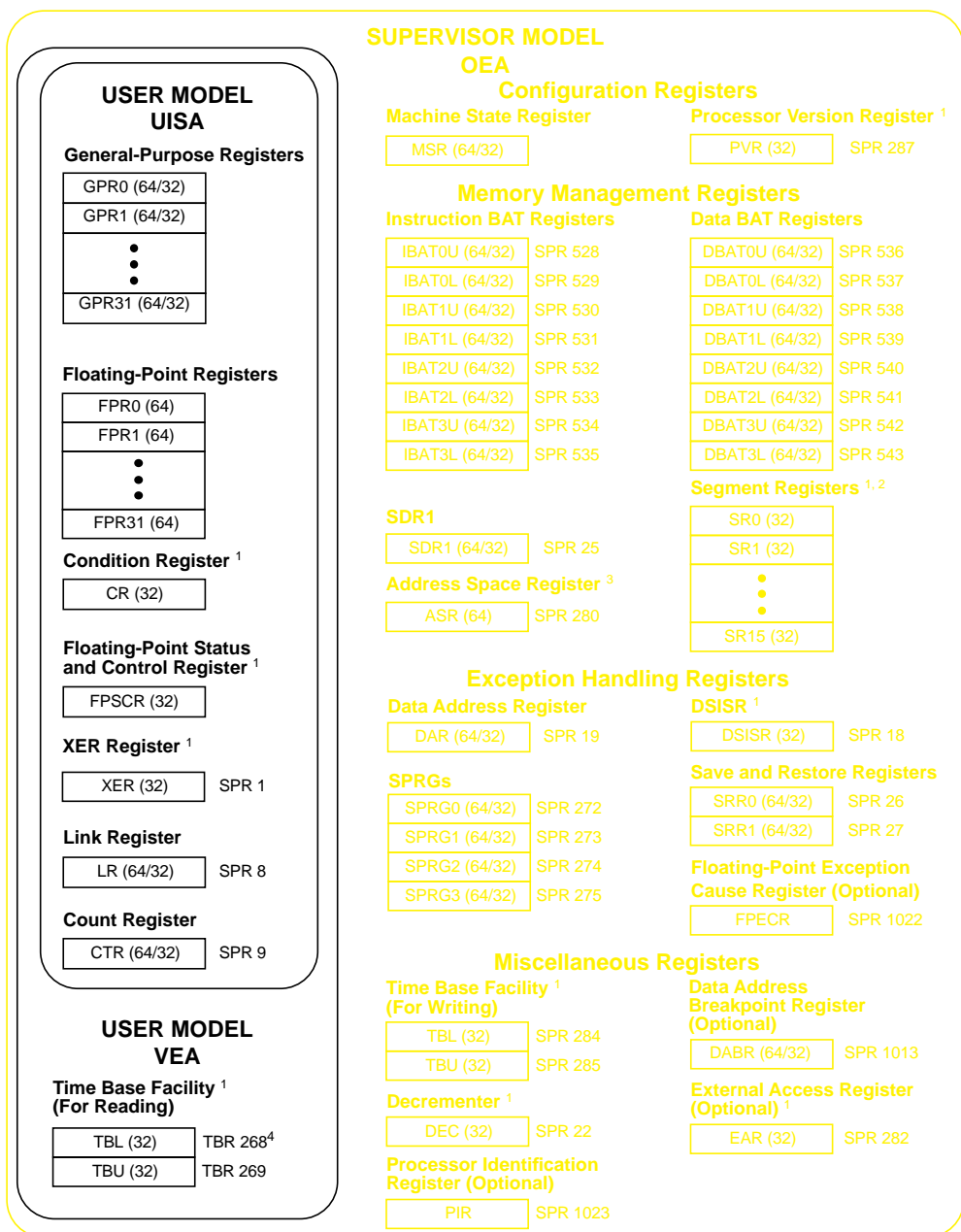
## 2.2 PowerPC VEA Register Set—Time Base

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

The PowerPC VEA introduces the time base facility (TB), a 64-bit structure that consists of two 32-bit registers—time base upper (TBU) and time base lower (TBL). Note that the time base registers can be accessed by both user- and supervisor-level instructions. In the context of the VEA, user-level applications are permitted read-only access to the TB. The OEA defines supervisor-level access to the TB for writing values to the TB. See Section 2.3.13, “Time Base Facility (TB)—OEA,” for more information.

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

Note that the general-purpose registers (GPRs), link register (LR), and count register (CTR) are 64 bits on 64-bit implementations and 32 bits on 32-bit implementations. These registers are described fully in Section 2.1, “PowerPC UISA Register Set.”



<sup>1</sup> These registers are 32-bit registers only.

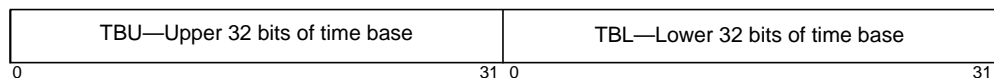
<sup>2</sup> These registers are on 32-bit implementations only.

<sup>3</sup> These registers are on 64-bit implementations only.

<sup>4</sup> In 64-bit implementations, TBR268 is read as a 64-bit value.

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

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



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

The TB increments until its value becomes 0xFFFF\_FFFF\_FFFF\_FFFF ( $2^{64} - 1$ ). At the next increment its value becomes 0x0000\_0000\_0000\_0000. Note that there is no explicit indication that this has occurred (that is, no exception is generated).

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

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

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

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

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

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

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

## 2.2.1 Reading the Time Base

The **mftb** instruction is used to read the time base. The following sections discuss reading the time base on 64-bit and 32-bit implementations. For specific details on using the **mftb** instruction, see Chapter 8, “Instruction Set.” For information on writing the time base, see Section 2.3.13.1, “Writing to the Time Base.”

### 2.2.1.1 Reading the Time Base on 64-Bit Implementations

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

```
mftb    rD
```

The above example uses the *simplified mnemonic* (referred to as extended mnemonic in the architecture specification) form of the **mftb** instruction (equivalent to **mftb rA,268**). Using this instruction on a 64-bit implementation copies the entire time base (TBU || TBL) into **rA**. Note that if the simplified mnemonic form **mftbu rA** (equivalent to **mftb rA,269**) is used on a 64-bit implementation, the contents of TBU are copied to the low-order 32 bits of **rA**, and the high-order 32 bits of **rA** are cleared (0 || TBU).

Reading the time base has no effect on the value it contains or the periodic incrementing of that value.

### 2.2.1.2 Reading the Time Base on 32-Bit Implementations

On 32-bit implementations, it is not possible to read the entire 64-bit time base in a single instruction. The **mftb** simplified mnemonic moves from the lower half of the time base register (TBL) to a GPR, and the **mftbu** simplified mnemonic moves from the upper half of the time base (TBU) to a GPR.

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

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

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



## 2.2.2 Computing Time of Day from the Time Base

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

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

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

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

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

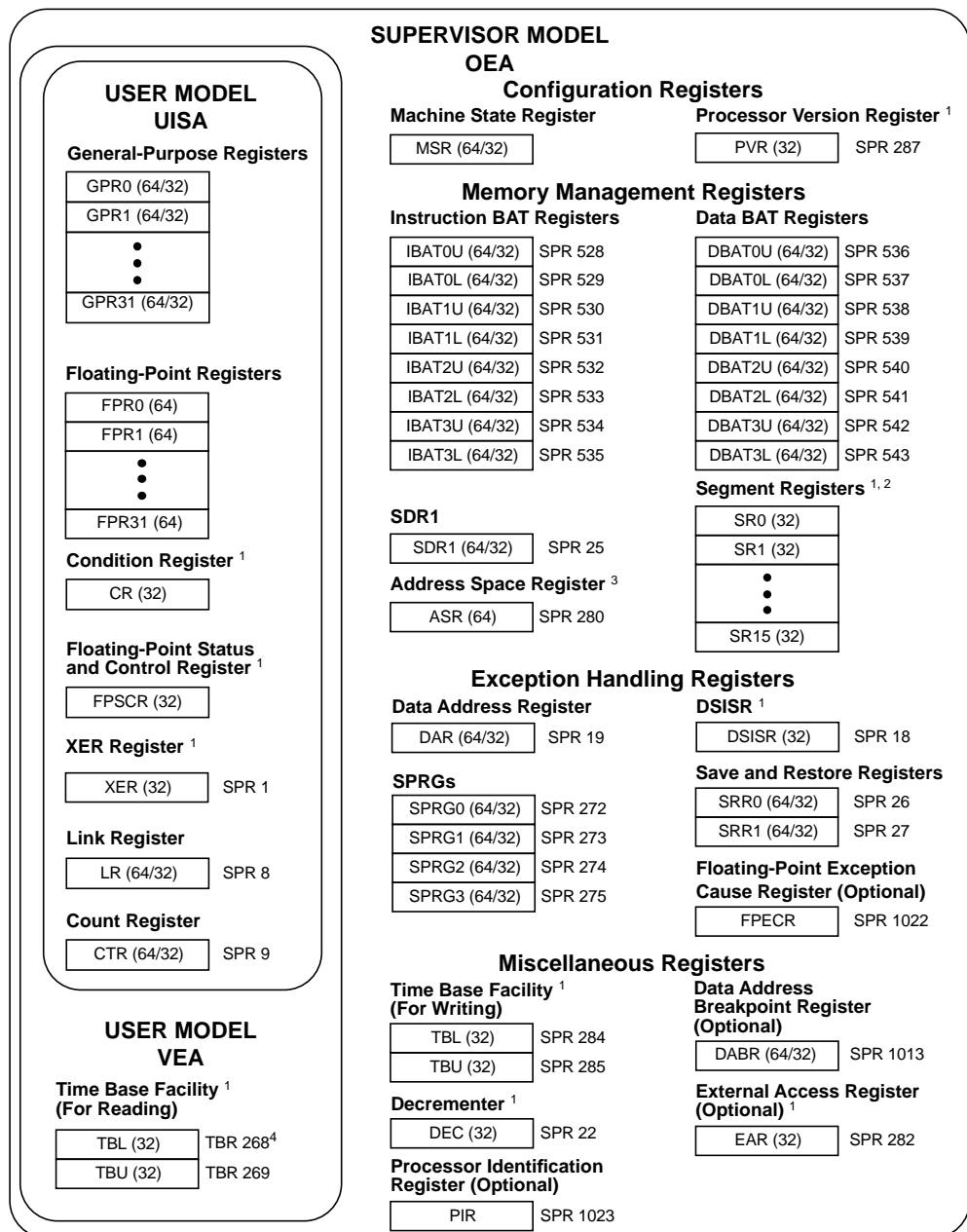
## 2.3 PowerPC OEA Register Set

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

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

If a PowerPC processor executes an **mtspr/mfspr** instruction with an undefined SPR encoding, it takes (depending on the implementation) an *illegal instruction* program exception, a privileged instruction program exception, or the results are *boundedly undefined*. See Section 6.4.7, “Program Exception (0x00700),” for more information.

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



<sup>1</sup> These registers are 32-bit registers only.

<sup>2</sup> These registers are on 32-bit implementations only.

<sup>3</sup> These registers are on 64-bit implementations only.

<sup>4</sup> In 64-bit implementations, TBR268 is read as a 64-bit value

Figure 2-11. OEA Programming Model—All Registers

A description of the PowerPC OEA supervisor-level registers follows:

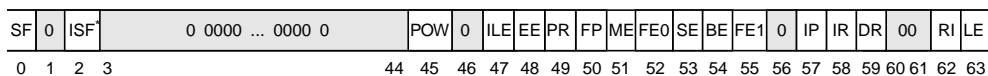
- **Configuration registers**
  - **Machine state register (MSR).** The MSR defines the state of the processor. The MSR can be modified by the Move to Machine State Register (**mtmsrd** [or **mtmsr**]), System Call (**sc**), and Return from Exception (**rfd** [or **rfi**]) instructions. It can be read by the Move from Machine State Register (**mfmsr**) instruction. For more information, see Section 2.3.1, “Machine State Register (MSR).”
  - Processor version register (PVR). This register is a **read-only register** that **identifies the version (model)** and revision level of the PowerPC processor. For more information, see Section 2.3.2, “Processor Version Register (PVR).”
- **Memory management registers**
  - Block-address translation (BAT) registers. The PowerPC OEA includes eight **block-address translation registers (BATs)**, consisting of four pairs of **instruction BATs** (IBAT0U–IBAT3U and IBAT0L–IBAT3L) and four pairs of **data BATs** (DBAT0U–DBAT3U and DBAT0L–DBAT3L). See Figure 2-11 for a list of the SPR numbers for the BAT registers. Refer to Section 2.3.3, “BAT Registers,” for more information.
  - SDR1. The **SDR1 register specifies the *page table* base** address used in virtual-to-physical address translation. For more information, see Section 2.3.4, “SDR1.” (Note that physical address is referred to as real address in the architecture specification.)
  - Address space register (ASR). The **ASR holds the physical address of the *segment table*. It is found only on 64-bit implementations.** For more information, see Section 2.3.5, “Address Space Register (ASR).”
  - **Segment registers (SR).** The PowerPC OEA defines **sixteen 32-bit segment registers (SR0–SR15).** Note that the **SRs are implemented on 32-bit implementations only.** The fields in the segment register are interpreted differently depending on the value of **bit 0.** For more information, see Section 2.3.6, “Segment Registers.” Note that the 64-bit bridge facility defines a way in which 64-bit implementations can use 16 *SLB* entries as if they were segment registers. See Chapter 7, “Memory Management,” for more detailed information about the bridge facility.
- **Exception handling registers**
  - Data address register (DAR). After a DSI or an alignment exception, DAR is set to the **effective address generated by the faulting instruction.** For more information, see Section 2.3.7, “Data Address Register (DAR).”
  - SPRG0–SPRG3. The SPRG0–SPRG3 registers are provided for operating system use. For more information, see Section 2.3.8, “SPRG0–SPRG3.”
  - DSISR. The DSISR defines the cause of DSI and alignment exceptions. For more information, refer to Section 2.3.9, “DSISR.”

- Machine status save/restore register 0 (**SRR0**). The SRR0 register is used to save **machine status on exceptions** and to restore machine status when an **rfid** [or **rfi**] instruction is executed. For more information, see Section 2.3.10, “Machine Status Save/Restore Register 0 (SRR0).”
- Machine status save/restore register 1 (**SRR1**). The SRR1 register is used to save **machine status on exceptions** and to restore machine status when an **rfid** [or **rfi**] instruction is executed. For more information, see Section 2.3.11, “Machine Status Save/Restore Register 1 (SRR1).”
- Floating-point exception cause register (**FPECR**). This *optional* register is used to identify the cause of a floating-point exception.
- **Miscellaneous registers**
  - **Time base (TB)**. The TB is a 64-bit structure that maintains the time of day and operates interval timers. The TB consists of two 32-bit registers—time base upper (TBU) and time base lower (TBL). Note that the time base registers can be accessed by both user- and supervisor-level instructions. For more information, see Section 2.3.13, “Time Base Facility (TB)—OEA” and Section 2.2, “PowerPC VEA Register Set—Time Base.”
  - **Decrementer register (DEC)**. This register is a 32-bit decrementing counter that provides a mechanism for causing a decrementer exception after a programmable delay; the frequency is a subdivision of the processor clock. For more information, see Section 2.3.14, “Decrementer Register (DEC).”
  - **External access register (EAR)**. This optional register is used in conjunction with the **eciwx** and **ecowx** instructions. Note that the EAR register and the **eciwx** and **ecowx** instructions are optional in the PowerPC architecture and may not be supported in all PowerPC processors that implement the OEA. For more information about the external control facility, see Section 4.3.4, “External Control Instructions.”
  - Data address breakpoint register (**DABR**). This optional register is used to control the data address breakpoint facility. Note that the DABR is optional in the PowerPC architecture and may not be supported in all PowerPC processors that implement the OEA. For more information about the data address breakpoint facility, see Section 6.4.3, “DSI Exception (0x00300).”
  - Processor identification register (**PIR**). This optional register is used to hold a value that **distinguishes an individual processor in a multiprocessor** environment.

### 2.3.1 Machine State Register (MSR)

The machine state register (MSR) is a 64-bit register on 64-bit implementations (see Figure 2-12) and a 32-bit register in 32-bit implementations (see Figure 2-13). The MSR defines the state of the processor. When an exception occurs, MSR bits, as described in Table 2-8, are altered as determined by the exception. The MSR can also be modified by

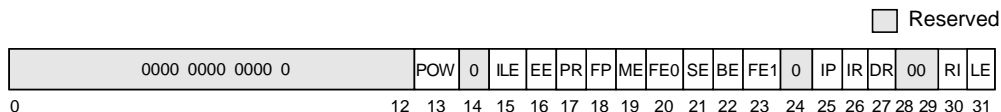
the **mtmsrd** [or **mtmsr**], **sc**, and **rfd** [or **rfi**] instructions. It can be read by the **mfmsr** instruction.



Temporary 64-Bit Bridge

\* Note that the ISF bit is optional and implemented only as part of the 64-bit bridge. For information see Table 2-8.

**Figure 2-12. Machine State Register (MSR)—64-Bit Implementations**



**Figure 2-13. Machine State Register (MSR)—32-Bit Implementations**

Table 2-8 shows the bit definitions for the MSR.

**Table 2-8. MSR Bit Settings**

Bit(s)		Name	Description
64 Bit	32 Bit		
0	—	SF	Sixty-four bit mode 0 The 64-bit processor runs in 32-bit mode. 1 The 64-bit processor runs in 64-bit mode. Note that this is the default setting.
1	—	—	Reserved
TEMPORARY 64-BIT BRIDGE 2	—	ISF	Exception 64-bit mode (Optional). When an exception occurs, this bit is copied into MSR[SF] to select 64- or 32-bit mode for the context established by the exception. Note: If the bridge function is not implemented, this bit is treated as reserved.
3–44	0–12	—	Reserved
45	13	POW	Power management enable 0 Power management disabled (normal operation mode) 1 Power management enabled (reduced power mode) Note: Power management functions are implementation-dependent. If the function is not implemented, this bit is treated as reserved.
46	14	—	Reserved
47	15	ILE	Exception <i>little-endian</i> mode. When an exception occurs, this bit is copied into MSR[LE] to select the endian mode for the context established by the exception.

**Table 2-8. MSR Bit Settings (Continued)**

Bit(s)		Name	Description
64 Bit	32 Bit		
48	16	EE	External interrupt enable 0 While the bit is cleared, the processor delays recognition of external <i>interrupts</i> and decrementer exception conditions. 1 The processor is enabled to take an external interrupt or the decrementer exception.
49	17	PR	<b>Privilege level</b> 0 The processor can execute both user- and supervisor-level instructions. 1 The processor can only execute user-level instructions.
50	18	FP	Floating-point available 0 The processor prevents dispatch of floating-point instructions, including floating-point loads, stores, and moves. 1 The processor can execute floating-point instructions.
51	19	ME	Machine check enable 0 Machine check exceptions are disabled. 1 Machine check exceptions are enabled.
52	20	FE0	Floating-point exception mode 0 (see Table 2-9).
53	21	SE	Single-step trace enable (Optional) 0 The processor executes instructions normally. 1 The processor generates a single-step trace exception upon the successful execution of the next instruction. Note: If the function is not implemented, this bit is treated as reserved.
54	22	BE	Branch trace enable (Optional) 0 The processor executes branch instructions normally. 1 The processor generates a branch trace exception after completing the execution of a branch instruction, regardless of whether the branch was taken. Note: If the function is not implemented, this bit is treated as reserved.
55	23	FE1	Floating-point exception mode 1 (See Table 2-9).
56	24	—	Reserved
57	25	IP	Exception prefix. The setting of this bit specifies whether an exception vector offset is prepended with Fs or 0s. In the following description, <i>nnnn</i> is the offset of the exception vector. See Table 6-2. 0 Exceptions are vectored to the physical address 0x000n_nnnn in 32-bit implementations and 0x0000_0000_000n_nnnn in 64-bit implementations. 1 Exceptions are vectored to the physical address 0xFFFFn_nnnn in 32-bit implementations and 0xFFFF_FFFF_FFFn_nnnn in 64-bit implementations.
58	26	IR	Instruction address translation 0 Instruction address translation is disabled. 1 Instruction address translation is enabled. For more information, see Chapter 7, “Memory Management.”
59	27	DR	Data address translation 0 Data address translation is disabled. 1 Data address translation is enabled. For more information, see Chapter 7, “Memory Management.”

**Table 2-8. MSR Bit Settings (Continued)**

Bit(s)		Name	Description
64 Bit	32 Bit		
60–61	28–29	—	Reserved
62	30	RI	Recoverable exception (for system reset and machine check exceptions). 0 Exception is not recoverable. 1 Exception is recoverable. For more information, see Chapter 6, “Exceptions.”
63	31	LE	Little-endian mode enable 0 The processor runs in <i>big-endian</i> mode. 1 The processor runs in little-endian mode.

The floating-point exception mode bits (FE0–FE1) are interpreted as shown in Table 2-9.

**Table 2-9. Floating-Point Exception Mode Bits**

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

Table 2-10 indicates the initial state of the MSR.

**Table 2-10. State of MSR at Power Up**

Bit(s)		Name	64-Bit Default Value	32-Bit Default Value
64 Bit	32 Bit			
0	—	SF	1	—
1	—	—	Unspecified <sup>1</sup>	—
TEMPORARY 64-BIT BRIDGE 2	—	ISF	1	—
3–44	0–12	—	Unspecified <sup>1</sup>	Unspecified <sup>1</sup>
45	13	POW	0	0
46	14	—	Unspecified <sup>1</sup>	Unspecified <sup>1</sup>
47	15	ILE	0	0
48	16	EE	0	0
49	17	PR	0	0
50	18	FP	0	0

**Table 2-10. State of MSR at Power Up (Continued)**

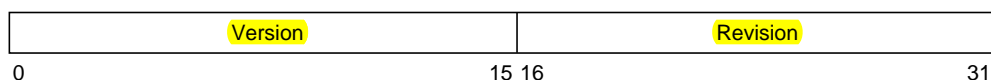
Bit(s)		Name	64-Bit Default Value	32-Bit Default Value
64 Bit	32 Bit			
51	19	ME	0	0
52	20	FE0	0	0
53	21	SE	0	0
54	22	BE	0	0
55	23	FE1	0	0
56	24	—	Unspecified <sup>1</sup>	Unspecified <sup>1</sup>
57	25	IP	1 <sup>2</sup>	1 <sup>2</sup>
58	26	IR	0	0
59	27	DR	0	0
60–61	28–29	—	Unspecified <sup>1</sup>	Unspecified <sup>1</sup>
62	30	RI	0	0
63	31	LE	0	0

<sup>1</sup> Unspecified can be either 0 or 1

<sup>2</sup> 1 is typical, but might be 0

## 2.3.2 Processor Version Register (PVR)

The processor version register (PVR) is a 32-bit, read-only register that contains a value identifying the **specific version (model) and revision level of the PowerPC processor** (see Figure 2-14). The contents of the PVR can be copied to a GPR by the **mfspr** instruction. Read access to the PVR is supervisor-level only; write access is not provided.



**Figure 2-14. Processor Version Register (PVR)**

The PVR consists of **two 16-bit fields**:

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

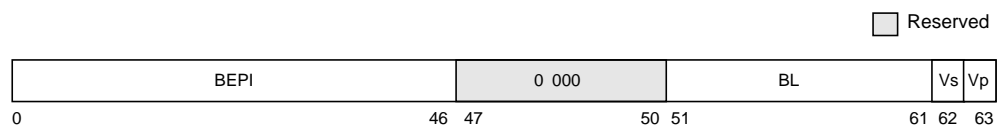


### 2.3.3 BAT Registers

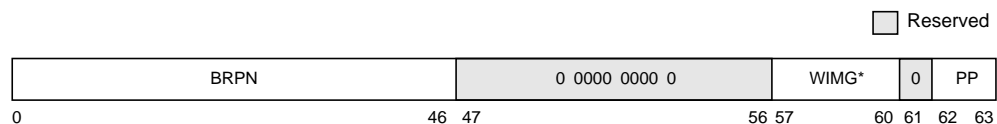
The BAT registers (BATs) maintain the address translation information for eight blocks of memory. The BATs are maintained by the system software and are implemented as eight pairs of special-purpose registers (SPRs). Each block is defined by a pair of SPRs called upper and lower BAT registers. These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

The PowerPC OEA defines the BAT registers as eight instruction block-address translation (IBAT) registers, consisting of four pairs of instruction BATs, or IBATs (IBAT0U–IBAT3U and IBAT0L–IBAT3L) and eight data BATs, or DBATs, (DBAT0U–DBAT3U and DBAT0L–DBAT3L). See Figure 2-11 for a list of the SPR numbers for the BAT registers.

Figure 2-15 and Figure 2-16 show the format of the upper and lower BAT registers for 64-bit PowerPC processors.



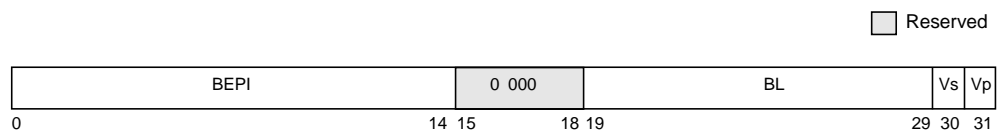
**Figure 2-15. Upper BAT Register—64-Bit Implementations**



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

**Figure 2-16. Lower BAT Register—64-Bit Implementations**

Figure 2-17 and Figure 2-18 show the format of the upper and lower BAT registers for 32-bit PowerPC processors.



**Figure 2-17. Upper BAT Register—32-Bit Implementations**

Reserved

BRPN														0 0000 0000 0										WIMG*		0	PP						
0														14 15														24 25		28 29		30 31	

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

## Figure 2-18. Lower BAT Register—32-Bit Implementations

Table 2-11 describes the bits in the BAT registers.

**Table 2-11. BAT Registers—Field and Bit Descriptions**

Upper/ Lower BAT Register	Bits		Name	Description
	64 Bit	32 Bit		
Upper BAT Register	0–46	0–14	BEPI	Block effective page index. This field is compared with high-order bits of the logical address to determine if there is a hit in that BAT array entry. (Note that the architecture specification refers to logical address as effective address.)
	46–50	15–18	—	Reserved
	51–61	19–29	BL	<b>Block length.</b> BL is a mask that encodes the size of the block. Values for this field are listed in Table 2-12.
	62	30	Vs	<b>Supervisor mode</b> valid bit. This bit interacts with MSR[PR] to determine if there is a match with the logical address. For more information, see Section 7.4.2, “Recognition of Addresses in BAT Arrays.”
	63	31	Vp	<b>User mode valid</b> bit. This bit also interacts with MSR[PR] to determine if there is a match with the logical address. For more information, see Section 7.4.2, “Recognition of Addresses in BAT Arrays.”
Lower BAT Register	0–46	0–14	BRPN	This field is used in conjunction with the BL field to generate high-order bits of the physical address of the block.
	47–56	15–24	—	Reserved
	57–60	25–28	WIMG	<b>Memory/cache access mode bits</b> <b>W</b> <b>Write-through</b> <b>I</b> <b>Caching-inhibited</b> <b>M</b> <b>Memory coherence</b> <b>G</b> <b>Guarded</b> Attempting to write to the W and G bits in IBAT registers causes boundedly-undefined results. For detailed information about the WIMG bits, see Section 5.2.1, “Memory/Cache Access Attributes.”
	61	29	—	Reserved
	62–63	30–31	PP	<b>Protection bits for block.</b> This field determines the protection for the block as described in Section 7.4.4, “Block Memory Protection.”

Table 2-12 lists the BAT area lengths encoded in BAT[BL].

**Table 2-12. BAT Area Lengths**

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

Only the values shown in Table 2-12 are valid for the BL field. The rightmost bit of BL is aligned with bit 46 (bit 14 for 32-bit implementations) of the logical address. A logical address is determined to be within a BAT area if the logical address matches the value in the BEPI field.

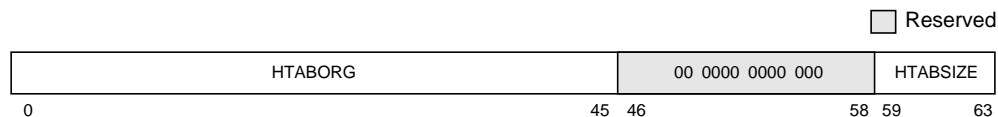
The boundary between the cleared bits and set bits (0s and 1s) in BL determines the bits of logical address that participate in the comparison with BEPI. Bits in the logical address corresponding to set bits in BL are cleared for this comparison. Bits in the logical address corresponding to set bits in the BL field, concatenated with the 17 bits of the logical address to the right (less significant bits) of BL, form the offset within the BAT area. This is described in detail in Chapter 7, “Memory Management.”

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

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

### 2.3.4 SDR1

The SDR1 is a 64-bit register in 64-bit implementations and a 32-bit register in 32-bit implementations. The 64-bit implementation of SDR1 is shown in Figure 2-19.



**Figure 2-19. SDR1—64-Bit Implementations**

The bits of the 64-bit implementation of SDR1 are described in Table 2-15.

**Table 2-13. SDR1 Bit Settings—64-Bit Implementations**

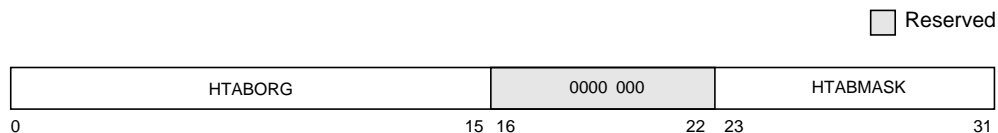
Bits	Name	Description
0–45	HTABORG	Physical base address of page table
46–58	—	Reserved
59–63	HTABSIZE	Encoded size of page table (used to generate mask)

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

The page table can be any size  $2^n$  where  $18 \leq n \leq 46$ . As the table size is increased, more bits are used from the hash to index into the table and the value in HTABORG must have more of its low-order bits equal to 0. The HTABSIZE field in SDR1 contains an integer value that determines how many bits from the hash are used in the page table index. This mask must be of the form 0b00...011...1; that is, a string of 0 bits followed by a string of 1 bits. The 1 bits determine how many additional bits (at least 11) from the hash are used in the index; HTABORG must have this same number of low-order bits equal to 0. See Figure 7-35 for an example of the primary PTEG address generation in a 64-bit implementation.

For example, suppose that the page table is 16,384 ( $2^{14}$ ), 128-byte PTEGs, for a total size of  $2^{21}$  bytes (2 Mbytes). Note that a 14-bit index is required. Eleven bits are provided from the hash initially, so three additional bits from the hash must be selected. The value in HTABSIZE must be 3 and the value in HTABORG must have its low-order three bits (bits 31–33 of SDR1) equal to 0. This means that the page table must begin on a  $2^{3+11+7} = 2^{21} = 2$  Mbytes boundary.

The 32-bit implementation of SDR1 is shown in Figure 2-20.



**Figure 2-20. SDR1—32-Bit Implementations**

The bits of the 32-bit implementation of SDR1 are described in Table 2-14.

**Table 2-14. SDR1 Bit Settings—32-Bit Implementations**

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

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

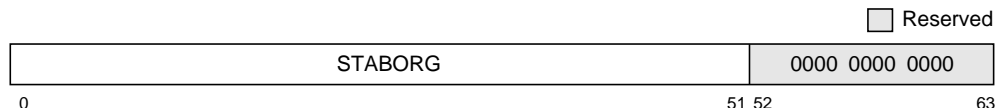
The page table can be any size  $2^n$  where  $16 \leq n \leq 25$ . As the table size is increased, more bits are used from the hash to index into the table and the value in HTABORG must have more of its low-order bits equal to 0. The HTABMASK field in SDR1 contains a mask value that determines how many bits from the hash are used in the page table index. This mask must be of the form 0b00...011...1; that is, a string of 0 bits followed by a string of 1 bits. The 1 bits determine how many additional bits (at least 10) from the hash are used in the index; HTABORG must have this same number of low-order bits equal to 0. See Figure 7-37 for an example of the primary PTEG address generation in a 32-bit implementation.

For example, suppose that the page table is 8,192 ( $2^{13}$ ), 64-byte PTEGs, for a total size of  $2^{19}$  bytes (512 Kbytes). Note that a 13-bit index is required. Ten bits are provided from the hash initially, so 3 additional bits from the hash must be selected. The value in HTABMASK must be 0x007 and the value in HTABORG must have its low-order 3 bits (bits 13–15 of SDR1) equal to 0. This means that the page table must begin on a  $2^{3+10+6} = 2^{19} = 512$  Kbytes boundary.

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

### 2.3.5 Address Space Register (ASR)

The ASR, shown in Figure 2-21, is a 64-bit SPR that holds bits 0–51 of the segment table’s physical address. The segment table contains the *segment table entries* for 64-bit implementations. The segment table defines the set of *segments* that can be addressed at any one time. Note that the ASR is defined only for 64-bit implementations.



**Figure 2-21. Address Space Register (ASR)—64-Bit Implementations Only**

The bits of the ASR are described in Table 2-15.

**Table 2-15. ASR Bit Settings**

Bits	Name	Description
0–51	STABORG	Physical address of segment table
52–63	—	Reserved

The following values, 0x0000\_0000\_0000\_0000, 0x0000\_0000\_0000\_1000, and 0x0000\_0000\_0000\_2000, cannot be used as segment table addresses, since these pages correspond to areas of the exception vector table reserved for implementation-specific purposes. For more information, see Chapter 7, “Memory Management.”

## TEMPORARY 64-BIT BRIDGE

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

If the ASR[V] bit is implemented and is set, the ASR[STABORG] field is valid and functions as described for the 64-bit architecture. However, if the ASR[V] bit is implemented and ASR[V] and MSR[SF] are cleared, an operating system can use 16 SLB entries similarly to the way 32-bit implementations use the segment registers, which are otherwise not supported in the 64-bit architecture. Note that if ASR[V] = 0, a reference to a nonexistent address in the STABORG field does not cause a machine check exception. For more information, see Section 7.7.1.1, “Address Space Register (ASR).”

The ASR, with the optional V bit implemented, is shown in Figure 2-22.



**Figure 2-22. Address Space Register (ASR)—64-Bit Bridge**

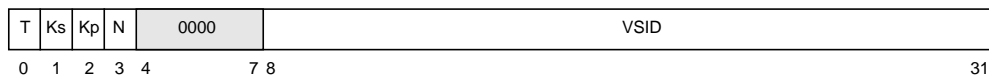
The bits of the ASR, including the optional V bit, are described in Table 2-16.

**Table 2-16. ASR Bit Settings—64-Bit Bridge**

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

### 2.3.6 Segment Registers

The segment registers contain the *segment descriptors* for 32-bit implementations. For 32-bit processors, the OEA defines a segment register file of sixteen 32-bit registers. Segment registers can be accessed by using the **mtsr/mfsr** and **mtsrin/mfsrin** instructions. The value of bit 0, the T bit, determines how the remaining register bits are interpreted. Figure 2-23 shows the format of a segment register when T = 0.



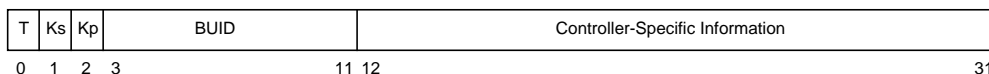
**Figure 2-23. Segment Register Format (T = 0)**

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

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

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

Figure 2-24 shows the bit definition when T = 1.



**Figure 2-24. Segment Register Format (T = 1)**

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

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

Bits	Name	Description
0	T	T = 1 selects this format.
1	Ks	Supervisor-state protection key
2	Kp	User-state protection key
3–11	BUID	Bus unit ID
12–31	CNTRLR_SPEC	Device-specific data for I/O controller



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

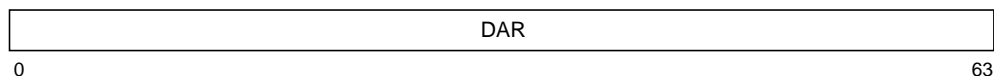
- The 24-bit VSID field from the segment register
- The 16-bit page index, EA[4–19]
- The 12-bit byte offset, EA[20–31]

The VA is then translated to a physical address as described in Section 7.5, “Memory Segment Model.”

If  $T = 1$  in the selected segment register (and the access is not translated by a BAT), the effective address is a reference to a *direct-store* segment. No reference is made to the page tables. For further discussion of address translation when  $T = 1$ , see Section 7.8, “Direct-Store Segment Address Translation.”

### 2.3.7 Data Address Register (DAR)

The DAR is a 64-bit register in 64-bit implementations and a 32-bit register in 32-bit implementations. The DAR is shown in Figure 2-25.



**Figure 2-25. Data Address Register (DAR)**

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

### 2.3.8 SPRG0–SPRG3

SPRG0–SPRG3 are 64-bit or 32-bit registers, depending on the type of PowerPC processor. They are provided for general operating system use, such as performing a fast state save or for supporting multiprocessor implementations. The formats of SPRG0–SPRG3 are shown in Figure 2-26.

	SPRG0	
	SPRG1	
	SPRG2	
	SPRG3	

0 63

**Figure 2-26. SPRG0–SPRG3**

Table 2-19 provides a description of conventional uses of SPRG0 through SPRG3.

**Table 2-19. Conventional Uses of SPRG0–SPRG3**

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

## 2.3.9 DSISR

The 32-bit DSISR, shown in Figure 2-27, identifies the cause of DSI and alignment exceptions.

	DSISR	
--	-------	--

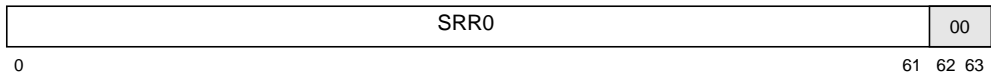
0 31

**Figure 2-27. DSISR**

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

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

The SRR0 is a 64-bit register in 64-bit implementations and a 32-bit register in 32-bit implementations. SRR0 is used to save machine status on exceptions and restore machine status when an **rfd** [or **rfi**] instruction is executed. It also holds the EA for the instruction that follows the System Call (**sc**) instruction. The format of SRR0 is shown in Figure 2-28. For 32-bit implementations, the format of SRR0 is that of the low-order bits (32–63) of Figure 2-28.



**Figure 2-28. Machine Status Save/Restore Register 0 (SRR0)**

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

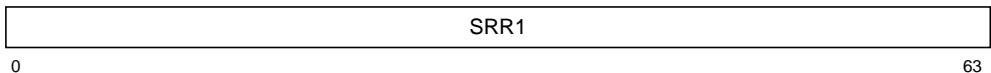
If the exception occurs in 32-bit mode of a 64-bit implementation, the high-order 32 bits of the NIA are cleared, NIA[32–61] are set from SRR0[32–61], and the two least significant bits of NIA are cleared.

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

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

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

The SRR1 is a 64-bit register in 64-bit implementations and a 32-bit register in 32-bit implementations. SRR1 is used to save machine status on exceptions and to restore machine status when an **rfd** [or **rfi**] instruction is executed. The format of SRR1 is shown in Figure 2-29.



**Figure 2-29. Machine Status Save/Restore Register 1 (SRR1)**

In 64-bit implementations, when an exception occurs, bits 33–36 and 42–47 of SRR1 are loaded with exception-specific information and bits 0–32, 37–41, and 48–63 of MSR are placed into the corresponding bit positions of SRR1. When **rfd** is executed, MSR[0–32, 37–41, 48–63] are loaded from SRR1[0–32, 37–41, 48–63].

For 32-bit implementations, when an exception occurs, bits 1–4 and 10–15 of SRR1 are loaded with exception-specific information and bits 0, 5–9, and 16–31 of MSR are placed into the corresponding bit positions of SRR1. When **rfi** is executed, MSR[0, 5–9, 16–31] are loaded from SRR1[0, 5–9, 16–31].

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

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

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

The FPECR register may be used to identify the cause of a floating-point exception. Note that the FPECR is an optional register in the PowerPC architecture and may be implemented differently (or not at all) in the design of each processor. The user’s manual of a specific processor will describe the functionality of the FPECR, if it is implemented in that processor.

### 2.3.13 Time Base Facility (TB)—OEA

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

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

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

#### 2.3.13.1 Writing to the Time Base

Note that writing to the TB is reserved for supervisor-level software.

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

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

The TB can be written by a sequence such as:

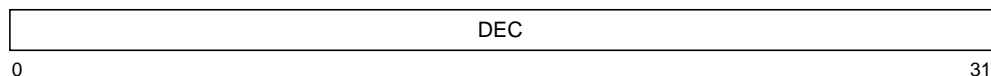
<b>lwz</b>	<b>rx, upper</b>	<b>#load 64-bit value for</b>
<b>lwz</b>	<b>ry, lower</b>	<b># TB into rx and ry</b>
<b>li</b>	<b>rz, 0</b>	
<b>mttbl</b>	<b>rz</b>	<b>#force TBL to 0</b>
<b>mttbu</b>	<b>rx</b>	<b>#set TBU</b>
<b>mttbl</b>	<b>ry</b>	<b>#set TBL</b>

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

For information on reading the time base, refer to Section 2.2.1, “Reading the Time Base.”

### 2.3.14 Decrementer Register (DEC)

The decrementer register (DEC), shown in Figure 2-30, is a 32-bit decrementing counter that provides a mechanism for causing a decrementer exception after a programmable delay. The DEC frequency is based on the same implementation-dependent frequency that drives the time base.



**Figure 2-30. Decrementer Register (DEC)**

#### 2.3.14.1 Decrementer Operation

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

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

### 2.3.14.2 Writing and Reading the DEC

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

```
mtdec  rA
```

Using a simplified mnemonic for the **mf spr** instruction, the DEC may be read into GPR **rA** with the following:

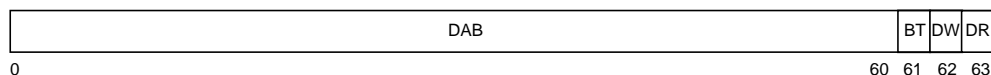
```
mfdec  rA
```

### 2.3.15 Data Address Breakpoint Register (DABR)

The optional data address breakpoint facility is controlled by an optional SPR, the DABR. The DABR is a 64-bit register in 64-bit implementations and a 32-bit register in 32-bit implementations. The data address breakpoint facility is optional to the PowerPC architecture. However, if the data address breakpoint facility is implemented, it is recommended, but not required, that it be implemented as described in this section.

The data address breakpoint facility provides a means to detect accesses to a designated double *word*. The address comparison is done on an effective address, and it applies to data accesses only. It does not apply to instruction fetches.

The DABR is shown in Figure 2-31.



**Figure 2-31. Data Address Breakpoint Register (DABR)**

Table 2-20 describes the fields in the DABR.

**Table 2-20. DABR—Bit Settings**

Bits		Name	Description
64 Bit	32 Bit		
0–60	0–28	DAB	Data address breakpoint
61	29	BT	Breakpoint translation enable
62	30	DW	Data write enable
63	31	DR	Data read enable

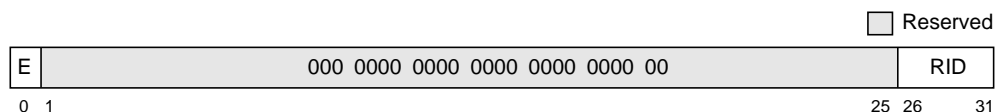
A data address breakpoint match is detected for a load or store instruction if the three following conditions are met for any byte accessed:

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

A match generates a DSI exception. Note that in 32-bit mode of a 64-bit implementation, the high-order 32 bits of the EA are treated as zero for the purpose of detecting a match. Refer to Section 6.4.3, “DSI Exception (0x00300),” for more information on the data address breakpoint facility.

### 2.3.16 External Access Register (EAR)

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



**Figure 2-32. External Access Register (EAR)**

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

The EAR register is provided to support the External Control In Word Indexed (**eciwx**) and External Control Out Word Indexed (**ecowx**) instructions, which are described in Chapter 8, “Instruction Set.” Although access to the EAR is supervisor-level, the operating system can determine which tasks are allowed to issue external access instructions and when they are allowed to do so. The bit settings for the EAR are described in Table 2-21. Interpretation of the physical address transmitted by the **eciwx** and **ecowx** instructions and the 32-bit value transmitted by the **ecowx** instruction is not prescribed by the PowerPC OEA but is determined by the target device. The data access of **eciwx** and **ecowx** is performed as though the memory access mode bits (WIMG) were 0101.

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

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

Bit	Name	Description
0	E	<p>Enable bit</p> <p>1 Enabled</p> <p>0 Disabled</p> <p>If this bit is set, the <b>eciwx</b> and <b>ecowx</b> instructions can perform the specified external operation. If the bit is cleared, an <b>eciwx</b> or <b>ecowx</b> instruction causes a DSI exception.</p>
1–25	—	Reserved
26–31	RID	Resource ID

This register can also be accessed by using the **mtspr** and **mfspir** instructions. *Synchronization* requirements for the EAR are shown in Table 2-22 and Table 2-23.

### 2.3.17 Processor Identification Register (PIR)

The **PIR** register is used to differentiate between individual processors in a multiprocessor environment. Note that the PIR is an optional register in the PowerPC architecture and may be implemented differently (or not at all) in the design of each processor. The user's manual of a specific processor will describe the functionality of the PIR, if it is implemented in that processor.

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

Changing the value in certain system registers, and invalidating SLB and *TLB* entries, can cause alteration of the context in which data addresses and instruction addresses are interpreted, and in which instructions are executed. An instruction that alters the context in which data addresses or instruction addresses are interpreted, or in which instructions are executed, is called a context-altering instruction. The *context synchronization* required for context-altering instructions is shown in Table 2-22 for data access and Table 2-23 for instruction fetch and execution.

A context-synchronizing exception (that is, any exception except nonrecoverable system reset or nonrecoverable machine check) can be used instead of a context-synchronizing instruction. In the tables, if no software synchronization is required before (after) a context-altering instruction, the synchronizing instruction before (after) the context-altering instruction should be interpreted as meaning the context-altering instruction itself.

A synchronizing instruction before the context-altering instruction ensures that all instructions up to and including that synchronizing instruction are fetched and executed in the context that existed before the alteration. A synchronizing instruction after the context-altering instruction ensures that all instructions after that synchronizing instruction are fetched and executed in the context established by the alteration. Instructions after the first



synchronizing instruction, up to and including the second synchronizing instruction, may be fetched or executed in either context.

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

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

No software synchronization is required before altering the MSR (except when altering the MSR[POW] or MSR[LE] bits; see Table 2-22 and Table 2-23), because **mtmsrd** [or **mtmsr**] is *execution synchronizing*. No software synchronization is required before most of the other alterations shown in Table 2-23, because all instructions before the context-altering instruction are fetched and decoded before the context-altering instruction is executed (the processor must determine whether any of the preceding instructions are context synchronizing).

Table 2-22 provides information on data access synchronization requirements.

**Table 2-22. Data Access Synchronization**

Instruction/Event	Required Prior	Required After
Exception <sup>2</sup>	None	None
<b>rfid</b> [or <b>rfi</b> ] <sup>2</sup>	None	None
<b>sc</b> <sup>2</sup>	None	None
Trap <sup>2</sup>	None	None
<b>mtmsrd</b> (SF)	None	Context-synchronizing instruction
<b>mtmsrd</b> [or <b>mtmsr</b> ] (ILE)	None	None
<b>mtmsrd</b> [or <b>mtmsr</b> ] (PR)	None	Context-synchronizing instruction
<b>mtmsrd</b> [or <b>mtmsr</b> ] (ME) <sup>1</sup>	None	Context-synchronizing instruction
<b>mtmsrd</b> [or <b>mtmsr</b> ] (DR)	None	Context-synchronizing instruction
<b>mtmsrd</b> [or <b>mtmsr</b> ] (LE) <sup>3</sup>	—	—
<b>mtsr</b> [or <b>mtsrin</b> ]	Context-synchronizing instruction	Context-synchronizing instruction
<b>mtspr</b> (ASR)	Context-synchronizing instruction	Context-synchronizing instruction
<b>mtspr</b> (SDR1) <sup>5, 6</sup>	<b>sync</b>	Context-synchronizing instruction
<b>mtspr</b> (DBAT)	Context-synchronizing instruction	Context-synchronizing instruction
<b>mtspr</b> (DABR) <sup>4</sup>	—	—
<b>mtspr</b> (EAR)	Context-synchronizing instruction	Context-synchronizing instruction
<b>slbie</b> <sup>7</sup>	Context-synchronizing instruction	Context-synchronizing instruction or <b>sync</b>

**Table 2-22. Data Access Synchronization (Continued)**

Instruction/Event	Required Prior	Required After
<b>slbia</b> <sup>7</sup>	Context-synchronizing instruction	Context-synchronizing instruction or <b>sync</b>
<b>tlbie</b> <sup>7, 8</sup>	Context-synchronizing instruction	Context-synchronizing instruction or <b>sync</b>
<b>tlbia</b> <sup>7, 8</sup>	Context-synchronizing instruction	Context-synchronizing instruction or <b>sync</b>

**Notes:**

<sup>1</sup> A context synchronizing instruction is required after modification of the MSR[ME] bit to ensure that the modification takes effect for subsequent machine check exceptions, which may not be recoverable and therefore may not be context synchronizing.

<sup>2</sup> Synchronization requirements for changing the power conserving mode are implementation-dependent.

<sup>3</sup> Synchronization requirements for changing from one endian mode to the other are implementation-dependent.

<sup>4</sup> Synchronization requirements for changing the DABR are implementation-dependent.

<sup>5</sup> SDR1 must not be altered when MSR[DR] = 1 or MSR[IR] = 1; if it is, the results are undefined.

<sup>6</sup> A **sync** instruction is required before the **mtspr** instruction because SDR1 identifies the page table and thereby the location of the referenced and changed (R and C) bits. To ensure that R and C bits are updated in the correct page table, SDR1 must not be altered until all R and C bit updates due to instructions before the **mtspr** have completed. A **sync** instruction guarantees this synchronization of R and C bit updates, while neither a context synchronizing operation nor the instruction fetching mechanism does so.

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

<sup>8</sup> Multiprocessor systems have other requirements to synchronize TLB invalidate.

For information on instruction access synchronization requirements, see Table 2-23.

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

Instruction/Event	Required Prior	Required After
Exception <sup>2</sup>	None	None
<b>rfid</b> [or <b>rfi</b> ] <sup>2</sup>	None	None
<b>sc</b> <sup>2</sup>	None	None
Trap <sup>2</sup>	None	None
<b>mtmsrd</b> (SF) <sup>9</sup>	None	Context-synchronizing instruction
<b>mtmsrd</b> [or <b>mtmsr</b> ] (POW) <sup>2</sup>	—	—
<b>mtmsrd</b> [or <b>mtmsr</b> ] (ILE)	None	None
<b>mtmsrd</b> [or <b>mtmsr</b> ] (EE) <sup>4</sup>	None	None
<b>mtmsrd</b> [or <b>mtmsr</b> ] (PR)	None	Context-synchronizing instruction
<b>mtmsrd</b> [or <b>mtmsr</b> ] (FP)	None	Context-synchronizing instruction
<b>mtmsrd</b> [or <b>mtmsr</b> ] (ME) <sup>1</sup>	None	Context-synchronizing instruction
<b>mtmsrd</b> [or <b>mtmsr</b> ] (FE0, FE1)	None	Context-synchronizing instruction
<b>mtmsrd</b> [or <b>mtmsr</b> ] (SE, BE)	None	Context-synchronizing instruction
<b>mtmsrd</b> [or <b>mtmsr</b> ] (IP)	None	None
<b>mtmsrd</b> [or <b>mtmsr</b> ] (IR) <sup>10</sup>	None	Context-synchronizing instruction
<b>mtmsrd</b> [or <b>mtmsr</b> ] (RI)	None	None
<b>mtmsrd</b> [or <b>mtmsr</b> ] (LE) <sup>3</sup>	—	—
<b>mtsr</b> [or <b>mtsrin</b> ] <sup>10</sup>	None	Context-synchronizing instruction
<b>mtspr</b> (ASR) <sup>10</sup>	None	Context-synchronizing instruction
<b>mtspr</b> (SDR1) <sup>5, 6</sup>	<b>sync</b>	Context-synchronizing instruction
<b>mtspr</b> (IBAT) <sup>10</sup>	None	Context-synchronizing instruction
<b>mtspr</b> (DEC) <sup>11</sup>	None	None
<b>slbie</b> <sup>7</sup>	None	Context-synchronizing instruction or <b>sync</b>
<b>slbia</b> <sup>7</sup>	None	Context-synchronizing instruction or <b>sync</b>

**Table 2-23. Instruction Access Synchronization (Continued)**

Instruction/Event	Required Prior	Required After
<b>tlbie</b> <sup>7, 8</sup>	None	Context-synchronizing instruction or <b>sync</b>
<b>tlbia</b> <sup>7, 8</sup>	None	Context-synchronizing instruction or <b>sync</b>

**Notes:**

<sup>1</sup> A context synchronizing instruction is required after modification of the MSR[ME] bit to ensure that the modification takes effect for subsequent machine check exceptions, which may not be recoverable and therefore may not be context synchronizing.

<sup>2</sup> Synchronization requirements for changing the power conserving mode are implementation-dependent.

<sup>3</sup> Synchronization requirements for changing from one endian mode to the other are implementation-dependent.

<sup>4</sup> The effect of altering the EE bit is immediate as follows:

- If an **mtmsrd** [or **mtmsr**] sets the EE bit to 0, neither an external interrupt nor a decremter exception can occur after the instruction is executed.
- If an **mtmsrd** [or **mtmsr**] sets the EE bit to 1 when an external interrupt, decremter exception, or higher priority exception exists, the corresponding exception occurs immediately after the **mtmsrd** [or **mtmsr**] is executed, and before the next instruction is executed in the program that set MSR[EE] to 1.

<sup>5</sup> SDR1 must not be altered when MSR[DR] = 1 or MSR[IR] = 1; if it is, the results are undefined.

<sup>6</sup> A **sync** instruction is required before the **mtspr** instruction because SDR1 identifies the page table and thereby the location of the referenced and changed (R and C) bits. To ensure that R and C bits are updated in the correct page table, SDR1 must not be altered until all R and C bit updates due to instructions before the **mtspr** have completed. A **sync** instruction guarantees this synchronization of R and C bit updates, while neither a context synchronizing operation nor the instruction fetching mechanism does so.

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

<sup>8</sup> Multiprocessor systems have other requirements to synchronize TLB invalidate.

<sup>9</sup> The alteration must not cause an implicit branch in effective address space. The **mtmsrd** (SF) instruction and all subsequent instructions, up to and including the next context-synchronizing instruction, must have effective addresses that are less than 2<sup>32</sup>.

<sup>10</sup> The alteration must not cause an implicit branch in physical address space. The physical address of the context-altering instruction and of each subsequent instruction, up to and including the next context synchronizing instruction, must be independent of whether the alteration has taken effect.

<sup>11</sup> The elapsed time between the content of the decremter becoming negative and the signaling of the decremter exception is not defined.