Chapter 6
Exceptions

The operating environment architecture (OEA) portion of the PowerPC architecture defines the mechanism by which PowerPC processors implement exceptions (referred to as interrupts in the architecture specification). Exception conditions may be defined at other levels of the architecture. For example, the user instruction set architecture (UISA) defines conditions that may cause floating-point exceptions; the OEA defines the mechanism by which the exception is taken.

The PowerPC exception mechanism allows the processor to change to supervisor state as a result of external signals, errors, or unusual conditions arising in the execution of instructions. When exceptions occur, information about the state of the processor is saved to certain registers and the processor begins execution at an address (exception vector) predetermined for each exception. Processing of exceptions begins in supervisor mode.

Although multiple exception conditions can map to a single exception vector, a more specific condition may be determined by examining a register associated with the exception—for example, the DSISR and the floating-point status and control register (FPSCR). Additionally, certain exception conditions can be explicitly enabled or disabled by software.

The PowerPC architecture requires that exceptions be taken in program order; therefore, although a particular implementation may recognize exception conditions out of order, they are handled strictly in order with respect to the instruction stream. When an instruction-caused exception is recognized, any unexecuted instructions that appear earlier in the instruction stream, including any that have not yet entered the execute state, are required to complete before the exception is taken. For example, if a single instruction encounters multiple exception conditions, those exceptions are taken and handled sequentially. Likewise, exceptions that are asynchronous and precise are recognized when they occur, but are not handled until all instructions currently in the execute stage successfully complete execution and report their results.

Note that exceptions can occur while an exception handler routine is executing, and multiple exceptions can become nested. It is up to the exception handler to save the appropriate machine state if it is desired to allow control to ultimately return to the excepting program.
In many cases, after the exception handler handles an exception, there is an attempt to execute the instruction that caused the exception. Instruction execution continues until the next exception condition is encountered. This method of recognizing and handling exception conditions sequentially guarantees that the machine state is recoverable and processing can resume without losing instruction results.

To prevent the loss of state information, exception handlers must save the information stored in SRR0 and SRR1 soon after the exception is taken to prevent this information from being lost due to another exception being taken.

In this chapter, the following terminology is used to describe the various stages of exception processing:

**Recognition**
- Exception recognition occurs when the condition that can cause an exception is identified by the processor.

**Taken**
- An exception is said to be taken when control of instruction execution is passed to the exception handler; that is, the context is saved and the instruction at the appropriate vector offset is fetched and the exception handler routine is begun in supervisor mode.

**Handling**
- Exception handling is performed by the software linked to the appropriate vector offset. Exception handling is begun in supervisor mode (referred to as privileged state in the architecture specification).

### 6.1 Exception Classes

As specified by the PowerPC architecture, all exceptions can be described as either precise or imprecise and either synchronous or asynchronous. *Asynchronous exceptions* are caused by events external to the processor’s execution; *synchronous exceptions* are caused by instructions.
The PowerPC exception types are shown in Table 6-1.

### Table 6-1. PowerPC Exception Classifications

<table>
<thead>
<tr>
<th>Type</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>Asynchronous/nonmaskable</td>
<td>Machine Check</td>
</tr>
<tr>
<td></td>
<td>System Reset</td>
</tr>
<tr>
<td>Asynchronous/maskable</td>
<td>External <em>interrupt</em> Decrementer</td>
</tr>
<tr>
<td>Synchronous/Precise</td>
<td>Instruction-caused exceptions, excluding floating-point <em>imprecise exceptions</em></td>
</tr>
<tr>
<td>Synchronous/Imprecise</td>
<td>Instruction-caused imprecise exceptions</td>
</tr>
<tr>
<td></td>
<td>(Floating-point imprecise exceptions)</td>
</tr>
</tbody>
</table>

Exceptions, and conditions that cause them, are summarized in Table 6-2. Remaining sections in this chapter provide more complete descriptions of the exceptions and of the conditions that cause them.
### Table 6-2. Exceptions and Conditions—Overview

<table>
<thead>
<tr>
<th>Exception Type</th>
<th>Vector Offset (hex)</th>
<th>Causing Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reserved</td>
<td>00000</td>
<td>—</td>
</tr>
<tr>
<td>System reset</td>
<td>00100</td>
<td>The causes of system reset exceptions are <em>implementation-dependent</em>. If the conditions that cause the exception also cause the processor state to be corrupted such that the contents of SRR0 and SRR1 are no longer valid or such that other processor resources are so corrupted that the processor cannot reliably resume execution, the copy of the RI bit copied from the MSR to SRR1 is cleared.</td>
</tr>
<tr>
<td>Machine check</td>
<td>00200</td>
<td>The causes for machine check exceptions are implementation-dependent, but typically these causes are related to conditions such as bus parity errors or attempting to access an invalid physical address. Typically, these exceptions are triggered by an input signal to the processor. Note that not all processors provide the same level of error checking. The machine check exception is disabled when MSR[ME] = 0. If a machine check exception condition exists and the ME bit is cleared, the processor goes into the checkstop state. If the conditions that cause the exception also cause the processor state to be corrupted such that the contents of SRR0 and SRR1 are no longer valid or such that other processor resources are so corrupted that the processor cannot reliably resume execution, the copy of the RI bit written from the MSR to SRR1 is cleared.</td>
</tr>
<tr>
<td>DSI</td>
<td>00300</td>
<td>A DSI exception occurs when a data memory access cannot be performed for any of the reasons described in Section 6.4.3, “DSI Exception (0x00300).” Such accesses can be generated by load/store instructions, certain memory control instructions, and certain cache control instructions.</td>
</tr>
<tr>
<td>ISI</td>
<td>00400</td>
<td>An ISI exception occurs when an instruction fetch cannot be performed for a variety of reasons described in Section 6.4.4, “ISI Exception (0x00400).”</td>
</tr>
<tr>
<td>External interrupt</td>
<td>00500</td>
<td>An external interrupt is generated only when an external interrupt is pending (typically signalled by a signal defined by the implementation) and the interrupt is enabled (MSR[EE] = 1).</td>
</tr>
<tr>
<td>Alignment</td>
<td>00600</td>
<td>An alignment exception may occur when the processor cannot perform a memory access for reasons described in Section 6.4.6, “Alignment Exception (0x00600).” Note that an implementation is allowed to perform the operation correctly and not cause an alignment exception.</td>
</tr>
</tbody>
</table>
Program 00700 A program exception is caused by one of the following exception conditions, which correspond to bit settings in SRR1 and arise during execution of an instruction:

- **Floating-point enabled exception**—A floating-point enabled exception condition is generated when MSR[FE0–FE1] = 00 and FPSCR[FEX] is set. The settings of FE0 and FE1 are described in Table 6-3. FPSCR[FEX] is set by the execution of a floating-point instruction that causes an enabled exception or by the execution of a Move to FPSCR instruction that sets both an exception condition bit and its corresponding enable bit in the FPSCR. These exceptions are described in Section 3.3.6, “Floating-Point Program Exceptions.”

- **Illegal instruction**—An illegal instruction program exception is generated when execution of an instruction is attempted with an illegal opcode or illegal combination of opcode and extended opcode fields or when execution of an optional instruction not provided in the specific implementation is attempted (these do not include those optional instructions that are treated as no-ops). The PowerPC instruction set is described in Chapter 4, “Addressing Modes and Instruction Set Summary.” See Section 6.4.7, “Program Exception (0x00700),” for a complete list of causes for an illegal instruction program exception.

- **Privileged instruction**—A privileged instruction type program exception is generated when the execution of a privileged instruction is attempted and the MSR user privilege bit, MSR[PR], is set. This exception is also generated for mspr or mfspr with an invalid SPR field if spr[0] = 1 and MSR[PR] = 1.

- **Trap**—A trap type program exception is generated when any of the conditions specified in a trap instruction is met. For more information, refer to Section 6.4.7, “Program Exception (0x00700).”

Floating-point unavailable 00800 A floating-point unavailable exception is caused by an attempt to execute a floating-point instruction (including floating-point load, store, and move instructions) when the floating-point available bit is cleared, MSR[FP] = 0.

Decrementer 00900 The decrementer interrupt exception is taken if the exception is enabled (MSR[EE] = 1), and it is pending. The exception is created when the most-significant bit of the decrementer changes from 0 to 1. If it is not enabled, the exception remains pending until it is taken.

Reserved 00A00 This is reserved for implementation-specific exceptions. For example, the 601 uses this vector offset for direct-store exceptions.

Reserved 00B00 —

System call 00C00 A system call exception occurs when a System Call (sc) instruction is executed.

Trace 00D00 Implementation of the trace exception is optional. If implemented, it occurs if either the MSR[SE] = 1 and almost any instruction successfully completed or MSR[BE] = 1 and a branch instruction is completed. See Section 6.4.11, “Trace Exception (0x00D00),” for more information.

Floating-point assist 00E00 Implementation of the floating-point assist exception is optional. This exception can be used to provide software assistance for infrequent and complex floating-point operations such as denormalization.

Reserved 00E10–00FFF —

Reserved 01000–02FFF This is reserved for implementation-specific exceptions.
6.1.1 Precise Exceptions

When any precise exceptions occur, SRR0 is set to point to an instruction such that all prior instructions in the instruction stream have completed execution and no subsequent instruction has begun execution. However, depending on the exception type, the instruction addressed by SRR0 may not have completed execution.

When an exception occurs, instruction dispatch (the issuance of instructions by the instruction fetch unit to any instruction execution mechanism) is halted and the following synchronization is performed:

1. The exception mechanism waits for all previous instructions in the instruction stream to complete to a point where they report all exceptions they will cause.
2. The processor ensures that all previous instructions in the instruction stream complete in the context in which they began execution.
3. The exception mechanism implemented in hardware and the software handler is responsible for saving and restoring the processor state.

The synchronization described conforms to the requirements for context synchronization. A complete description of context synchronization is described in the following section.

6.1.2 Synchronization

The synchronization described in this section refers to the state of activities within the processor that performs the synchronization.

6.1.2.1 Context Synchronization

An instruction or event is context synchronizing if it satisfies all the requirements listed below. Such instructions and events are collectively called context-synchronizing operations. Examples of context-synchronizing operations include the sc and rfid (or rfi) instructions and most exceptions. A context-synchronizing operation has the following characteristics:

- The operation causes instruction dispatching (the issuance of instructions by the instruction fetch mechanism to any instruction execution mechanism) to be halted.
- The operation is not initiated or, in the case of isync, is not completed, until all instructions in execution have completed to a point at which they have reported all exceptions they will cause. If a memory access due to a previously initiated instruction may cause one or more direct-store error exceptions, the determination of whether it does cause such exceptions is made before the operation is initiated.
- Instructions that precede the operation complete execution in the context (for example, the privilege, translation mode, and memory protection) in which they were initiated.
• If the operation either directly causes an exception (for example, the \texttt{sc} instruction causes a system call exception) or is an exception, the operation is not initiated until no exception exists having higher priority than the exception associated with the context-synchronizing operation.

A context-synchronizing operation is necessarily \textit{execution synchronizing}. Unlike the \texttt{sync} instruction, a context-synchronizing operation need not wait for memory-related operations to complete on other processors, nor for \textit{referenced} and \textit{changed bits} in the \textit{page table} to be updated.

\subsection*{6.1.2.2 Execution Synchronization}

An instruction is execution synchronizing if all previously initiated instructions appear to have completed before the instruction is initiated or, in the case of the \texttt{sync} and \texttt{isync} instructions, before the instruction completes. The \texttt{sync} and \texttt{mtmsr} instructions are execution-synchronizing.

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

\subsection*{6.1.2.3 Synchronous/Precise Exceptions}

When instruction execution causes a precise exception, the following conditions exist at the exception point:

• Depending on the type of exception, SRR0 addresses either the instruction causing the exception or the immediately following instruction. The instruction addressed can be determined from the exception type and status bits, which are defined in the description of each exception.

• All instructions that precede the excepting instruction complete before the exception is processed. However, some memory accesses generated by these preceding instructions may not have been performed with respect to all other processors or system devices.

• The instruction causing the exception may not have begun execution, may have partially completed, or may have completed, depending on the exception type. Handling of partially executed instructions is described in Section 6.1.4, “Partially Executed Instructions.”

• Architecturally, no subsequent instruction has begun execution.

While \textit{instruction parallelism} allows the possibility of multiple instructions reporting exceptions during the same cycle, they are handled one at a time in program order. Exception priorities are described in Section 6.1.5, “Exception Priorities.”
6.1.2.4 Asynchronous Exceptions
There are four asynchronous exceptions—system reset and machine check, which are nonmaskable and highest-priority exceptions, and external interrupt and decremente
exceptions which are maskable and low-priority. These two types of asynchronous exceptions are discussed separately.

6.1.2.4.1 System Reset and Machine Check Exceptions
System reset and machine check exceptions have the highest priority and can occur while other exceptions are being processed. Note that nonmaskable, asynchronous exceptions are never delayed; therefore, if two of these exceptions occur in immediate succession, the state information saved by the first exception may be overwritten when the subsequent exception occurs. Note that these exceptions are context-synchronizing if they are recoverable (MSR[RI] is copied from the MSR to SRR1 if the exception does not cause loss of state.) If the RI bit is clear (nonrecoverable), the exception is context-synchronizing only with respect to subsequent instructions.

These exceptions cannot be masked by using the MSR[EE] bit. However, if the machine check enable bit, MSR[ME], is cleared and a machine check exception condition occurs, the processor goes directly into checkstop state as the result of the exception condition. When one of these exceptions occur, the following conditions exist at the exception point:

- For system reset exceptions, SRR0 addresses the instruction that would have attempted to execute next if the exception had not occurred.
- For machine check exceptions, SRR0 holds either an instruction that would have completed or some instruction following it that would have completed if the exception had not occurred.
- An exception is generated such that all instructions preceding the instruction addressed by SRR0 appear to have completed with respect to the executing processor.

Note that a bit in the MSR (MSR[RI]) indicates whether enough of the machine state was saved to allow the processor to resume processing.

6.1.2.4.2 External Interrupt and Decrementer Exceptions
For the external interrupt and decremente exceptions, the following conditions exist at the exception point (assuming these exceptions are enabled (MSR[EE] bit is set)):

- All instructions issued before the exception is taken and any instructions that precede those instructions in the instruction stream appear to have completed before the exception is processed.
- No subsequent instructions in the instruction stream have begun execution.
- SRR0 addresses the instruction that would have been executed had the exception not occurred.
That is, these exceptions are context-synchronizing. The external interrupt and decrementer exceptions are maskable. When the machine state register external interrupt enable bit is cleared (MSR[EE] = 0), these exception conditions are not recognized until the EE bit is set. MSR[EE] is cleared automatically when an exception is taken, to delay recognition of subsequent exception conditions. No two precise exceptions can be recognized simultaneously. Exception handling does not begin until all currently executing instructions complete and any synchronous, precise exceptions caused by those instructions have been handled. Exception priorities are described in Section 6.1.5, “Exception Priorities.”

6.1.3 Imprecise Exceptions
The PowerPC architecture defines one imprecise exception, the imprecise floating-point enabled exception. This is implemented as one of the conditions that can cause a program exception.

6.1.3.1 Imprecise Exception Status Description
When the execution of an instruction causes an imprecise exception, SRR0 contains information related to the address of the excepting instruction as follows:

- SRR0 contains the address of either the instruction that caused the exception or of some instruction following that instruction.
- The exception is generated such that all instructions preceding the instruction addressed by SRR0 have completed with respect to the processor.
- If the imprecise exception is caused by the context-synchronizing mechanism (due to an instruction that caused another exception—for example, an alignment or DSI exception), then SRR0 contains the address of the instruction that caused the exception, and that instruction may have been partially executed (refer to Section 6.1.4, “Partially Executed Instructions”).
- If the imprecise exception is caused by an execution-synchronizing instruction other than sync or isync, SRR0 addresses the instruction causing the exception. Additionally, besides causing the exception, that instruction is considered not to have begun execution. If the exception is caused by the sync or isync instruction, SRR0 may address either the sync or isync instruction, or the following instruction.
- If the imprecise exception is not forced by either the context-synchronizing mechanism or the execution-synchronizing mechanism, the instruction addressed by SRR0 is considered not to have begun execution if it is not the instruction that caused the exception.
- When an imprecise exception occurs, no instruction following the instruction addressed by SRR0 is considered to have begun execution.
6.1.3.2 Recoverability of Imprecise Floating-Point Exceptions

The enabled IEEE floating-point exception mode bits in the MSR (FE0 and FE1) together define whether IEEE floating-point exceptions are handled precisely, imprecisely, or whether they are taken at all. The possible settings are shown in Table 6-3. For further details, see Section 3.3.6, “Floating-Point Program Exceptions.”

As shown in the table, the imprecise floating-point enabled exception has two modes—nonrecoverable and recoverable. These modes are specified by setting the MSR[FE0] and MSR[FE1] bits and are described as follows:

- Imprecise nonrecoverable floating-point enabled mode. MSR[FE0] = 0; MSR[FE1] = 1. When an exception occurs, the exception handler is invoked at some point at or beyond the instruction that caused the exception. It may not be possible to identify the excepting instruction or the data that caused the exception. Results from the excepting instruction may have been used by or affected subsequent instructions executed before the exception handler was invoked.

- Imprecise recoverable floating-point enabled mode. MSR[FE0] = 1; MSR[FE1] = 0. When an exception occurs, the floating-point enabled exception handler is invoked at some point at or beyond the instruction that caused the exception. Sufficient information is provided to the exception handler that it can identify the excepting instruction and correct any faulty results. In this mode, no incorrect results caused by the excepting instruction have been used by or affected subsequent instructions that are executed before the exception handler is invoked.

Although these exceptions are maskable with these bits, they differ from other maskable exceptions in that the masking is usually controlled by the application program rather than by the operating system.

Table 6-3. IEEE Floating-Point Program Exception Mode Bits

<table>
<thead>
<tr>
<th>FE0</th>
<th>FE1</th>
<th>Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Floating-point exceptions ignored</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Floating-point imprecise nonrecoverable</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Floating-point imprecise recoverable</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Floating-point precise mode</td>
</tr>
</tbody>
</table>
6.1.4 Partially Executed Instructions

The architecture permits certain instructions to be partially executed when an alignment exception or DSI exception occurs, or an imprecise floating-point exception is forced by an instruction that causes an alignment or DSI exception. They are as follows:

- Load multiple/string instructions that cause an alignment or DSI exception—Some registers in the range of registers to be loaded may have been loaded.
- Store multiple/string instructions that cause an alignment or DSI exception—Some bytes in the addressed memory range may have been updated.
- Non-multiple/string store instructions that cause an alignment or DSI exception—Some bytes just before the boundary may have been updated. If the instruction normally alters CR0 (stwcx. or stdcx.), CR0 is set to an undefined value. For instructions that perform register updates, the update register (rA) is not altered.
- Floating-point load instructions that cause an alignment or DSI exception—The target register may be altered. For update forms, the update register (rA) is not altered.
- A load or store to a direct-store segment that causes a DSI exception due to a direct-store interface error exception—Some of the associated address/data transfers may not have been initiated. All initiated transfers are completed before the exception is reported, and the transfers that have not been initiated are aborted. Thus the instruction completes before the DSI exception occurs.

In the cases above, the number of registers and the amount of memory altered are implementation-, instruction-, and boundary-dependent. However, memory protection is not violated. Furthermore, if some of the data accessed is in a direct-store segment and the instruction is not supported for use in such memory space, the locations in the direct-store segment are not accessed.

Partial execution is not allowed when integer load operations (except multiple/string operations) cause an alignment or DSI exception. The target register is not altered. For update forms of the integer load instructions, the update register (rA) is not altered.

6.1.5 Exception Priorities

Exceptions are roughly prioritized by exception class, as follows:

1. Nonmaskable, asynchronous exceptions have priority over all other exceptions—system reset and machine check exceptions (although the machine check exception condition can be disabled so that the condition causes the processor to go directly into the checkstop state). These two types of exceptions in this class cannot be delayed by exceptions in other classes, and do not wait for the completion of any precise exception handling.
2. Synchronous, precise exceptions are caused by instructions and are taken in strict program order.
3. If an imprecise exception exists (the instruction that caused the exception has been completed and is required by the sequential execution model), exceptions signaled by instructions subsequent to the instruction that caused the exception are not permitted to change the architectural state of the processor. The exception causes an imprecise program exception unless a machine check or system reset exception is pending.

4. Maskable asynchronous exceptions (external interrupt and decrementer exceptions) have lowest priority.

The exceptions are listed in Table 6-4 in order of highest to lowest priority.

Table 6-4. Exception Priorities

<table>
<thead>
<tr>
<th>Exception Class</th>
<th>Priority</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nonmaskable, asynchronous</td>
<td>1</td>
<td>System reset—The system reset exception has the highest priority of all exceptions. If this exception exists, the exception mechanism ignores all other exceptions and generates a system reset exception. When the system reset exception is generated, previously issued instructions can no longer generate exception conditions that cause a nonmaskable exception.</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>Machine check—The machine check exception is the second-highest priority exception. If this exception occurs, the exception mechanism ignores all other exceptions (except reset) and generates a machine check exception. When the machine check exception is generated, previously issued instructions can no longer generate exception conditions that cause a nonmaskable exception.</td>
</tr>
</tbody>
</table>
Synchronous, precise

When an instruction causes an exception, the exception mechanism waits for any instructions prior to the excepting instruction in the instruction stream to complete. Any exceptions caused by these instructions are handled first. It then generates the appropriate exception if no higher priority exception exists when the exception is to be generated.

Note that a single instruction can cause multiple exceptions. When this occurs, those exceptions are ordered in priority as indicated in the following:

A. Integer loads and stores
   a. Alignment
   b. DSI
   c. Trace (if implemented)
B. Floating-point loads and stores
   a. Floating-point unavailable
   b. Alignment
   c. DSI
   d. Trace (if implemented)
C. Other floating-point instructions
   a. Floating-point unavailable
   b. Program—Precise-mode floating-point enabled exception
   c. Floating-point assist (if implemented)
   d. Trace (if implemented)
D. \texttt{rfid} (or \texttt{rfi}) and \texttt{mtmsrd} (or \texttt{mtmsr})
   a. Program—Privileged Instruction
   b. Program—Precise-mode floating-point enabled exception
   c. Trace (if implemented), for \texttt{mtmsrd} (or \texttt{mtmsr}) only
   If precise-mode IEEE floating-point enabled exceptions are enabled and the FPSCR[FEX] bit is set, a program exception occurs no later than the next synchronizing event.
E. Other instructions
   a. These exceptions are mutually exclusive and have the same priority:
      —Program: Trap
      —System call (\texttt{sc})
      —Program: Privileged Instruction
      —Program: Illegal Instruction
   b. Trace (if implemented)
F. ISI exception
   The ISI exception has the lowest priority in this category. It is only recognized when all instructions prior to the instruction causing this exception appear to have completed and that instruction is to be executed. The priority of this exception is specified for completeness and to ensure that it is not given more favorable treatment. An implementation can treat this exception as though it had a lower priority.

Imprecise

Program imprecise floating-point mode enabled exceptions—When this exception occurs, the exception handler is invoked at or beyond the floating-point instruction that caused the exception. The PowerPC architecture supports recoverable and nonrecoverable imprecise modes, which are enabled by setting MSR[FE0] ≠ MSR[FE1]. For more information see, Section 6.1.3, “Imprecise Exceptions.”

<table>
<thead>
<tr>
<th>Exception Class</th>
<th>Priority</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synchronous, precise</td>
<td>3</td>
<td>Instruction dependent—When an instruction causes an exception, the exception mechanism waits for any instructions prior to the excepting instruction in the instruction stream to complete. Any exceptions caused by these instructions are handled first. It then generates the appropriate exception if no higher priority exception exists when the exception is to be generated. Note that a single instruction can cause multiple exceptions. When this occurs, those exceptions are ordered in priority as indicated in the following: A. Integer loads and stores a. Alignment b. DSI c. Trace (if implemented) B. Floating-point loads and stores a. Floating-point unavailable b. Alignment c. DSI d. Trace (if implemented) C. Other floating-point instructions a. Floating-point unavailable b. Program—Precise-mode floating-point enabled exception c. Floating-point assist (if implemented) d. Trace (if implemented) D. \texttt{rfid} (or \texttt{rfi}) and \texttt{mtmsrd} (or \texttt{mtmsr}) a. Program—Privileged Instruction b. Program—Precise-mode floating-point enabled exception c. Trace (if implemented), for \texttt{mtmsrd} (or \texttt{mtmsr}) only If precise-mode IEEE floating-point enabled exceptions are enabled and the FPSCR[FEX] bit is set, a program exception occurs no later than the next synchronizing event. E. Other instructions a. These exceptions are mutually exclusive and have the same priority: —Program: Trap —System call (\texttt{sc}) —Program: Privileged Instruction —Program: Illegal Instruction b. Trace (if implemented) F. ISI exception The ISI exception has the lowest priority in this category. It is only recognized when all instructions prior to the instruction causing this exception appear to have completed and that instruction is to be executed. The priority of this exception is specified for completeness and to ensure that it is not given more favorable treatment. An implementation can treat this exception as though it had a lower priority.</td>
</tr>
<tr>
<td>Imprecise</td>
<td>4</td>
<td>Program imprecise floating-point mode enabled exceptions—When this exception occurs, the exception handler is invoked at or beyond the floating-point instruction that caused the exception. The PowerPC architecture supports recoverable and nonrecoverable imprecise modes, which are enabled by setting MSR[FE0] ≠ MSR[FE1]. For more information see, Section 6.1.3, “Imprecise Exceptions.”</td>
</tr>
</tbody>
</table>
Nonmaskable, asynchronous exceptions (namely, system reset or machine check exceptions) may occur at any time. That is, these exceptions are not delayed if another exception is being handled (although machine check exceptions can be delayed by system reset exceptions). As a result, state information for the interrupted exception handler may be lost.

All other exceptions have lower priority than system reset and machine check exceptions, and the exception may not be taken immediately when it is recognized. Only one synchronous, precise exception can be reported at a time. If a maskable, asynchronous or an imprecise exception condition occurs while instruction-caused exceptions are being processed, its handling is delayed until all exceptions caused by previous instructions in the program flow are handled and those instructions complete execution.

### 6.2 Exception Processing

When an exception is taken, the processor uses the save/restore registers, SRR1 and SRR0, respectively, to save the contents of the MSR for the interrupted process and to help determine where instruction execution should resume after the exception is handled.

When an exception occurs, the address saved in SRR0 is used to help calculate where instruction processing should resume when the exception handler returns control to the interrupted process. Depending on the exception, this may be the address in SRR0 or at the next address in the program flow. All instructions in the program flow preceding this one will have completed execution and no subsequent instruction will have begun execution. This may be the address of the instruction that caused the exception or the next one (as in the case of a system call or trap exception). The SRR0 register is shown in Figure 6-1.
Chapter 6. Exceptions

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

This register is 32 bits wide in 32-bit implementations.

The save/restore register 1 (SRR1) is used to save machine status (selected bits from the MSR and possibly other status bits as well) on exceptions and to restore those values when \texttt{rfid} (or \texttt{rfi}) is executed. SRR1 is shown in Figure 6-2.

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

This register is 32 bits wide in 32-bit implementations. Typically, 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. Note that depending on the implementation, reserved bits in the MSR may not be copied to SRR1.

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

The MSR bits for 64-bit implementations are shown in Figure 6-3.

Figure 6-3. Machine State Register (MSR)—64-Bit Implementation

In 32-bit PowerPC implementations, the MSR is 32 bits wide as shown in Figure 6-4. Note that the 32-bit implementation of the MSR is comprised of the 32 least-significant bits of the 64-bit MSR.
Table 6-5 shows the bit definitions for the MSR.

### Table 6-5. MSR Bit Settings

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>64 Bit</td>
<td>32 Bit</td>
<td></td>
</tr>
</tbody>
</table>
| 0      | SF   | Sixty-four bit mode  
       |      | 0: The 64-bit processor runs in 32-bit mode.  
       |      | 1: The 64-bit processor runs in 64-bit mode. Note that this is the default setting. |
| 1      | —    | Reserved |
| 64-BIT | ISF  | Exception sixty-four 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 function is not implemented, this bit is treated as reserved. |
| 3–44   | 0–12 | Reserved |
| 45     | POW  | Power management enable  
       |      | 0: Power management disabled (normal operation mode)  
       |      | 1: Power management enabled (reduced power mode)  
       |      | Note: Power management functions are implementation-dependent. If the function is not implemented, this bit is treated as reserved. |
| 46     | 14   | Reserved |
| 47     | ILE  | Exception little-endian mode. When an exception occurs, this bit is copied into MSR[LE] to select the endian mode for the context established by the exception. |
| 48     | EE   | External interrupt enable  
       |      | 0: While the bit is cleared the processor delays recognition of external interrupts and decrementer exception conditions.  
       |      | 1: The processor is enabled to take an external interrupt or the decrementer exception. |
| 49     | PR   | Privilege level  
       |      | 0: The processor can execute both user- and supervisor-level instructions.  
       |      | 1: The processor can only execute user-level instructions. |
| 50     | FP   | Floating-point available  
       |      | 0: The processor prevents dispatch of floating-point instructions, including floating-point loads, stores, and moves.  
       |      | 1: The processor can execute floating-point instructions. |
Table 6-5. MSR Bit Settings (Continued)

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>64 Bit</td>
<td>32 Bit</td>
<td></td>
</tr>
</tbody>
</table>
| 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 2-10 on page 23). |
| 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 2-10 on page 23). |
| 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, nnnnn 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_0000n_nnnn in 64-bit implementations.  
1 Exceptions are vectored to the physical address 0xFFFn_nnnn in 32-bit implementations and 0xFFFF_FFFF_FFFnnn 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.” |
| 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 Section 6.4.1, “System Reset Exception (0x00100),” and Section 6.4.2, “Machine Check Exception (0x00200).” |
| 63 | 31 | LE | Little-endian mode enable  
0 The processor runs in big-endian mode.  
1 The processor runs in little-endian mode. |
Those MSR bits that are written to SRR1 are written when the first instruction of the exception handler is encountered. The data address register (DAR) is used by several exceptions (for example, DSI and alignment exceptions) to identify the address of a memory element.

### 6.2.1 Enabling and Disabling Exceptions

When a condition exists that may cause an exception to be generated, it must be determined whether the exception is enabled for that condition as follows:

- IEEE floating-point enabled exceptions (a type of program exception) are ignored when both MSR[FE0] and MSR[FE1] are cleared. If either of these bits is set, all IEEE enabled floating-point exceptions are taken and cause a program exception.
- Asynchronous, maskable exceptions (that is, the external and decremener interrupts) are enabled by setting the MSR[EE] bit. When MSR[EE] = 0, recognition of these exception conditions is delayed. MSR[EE] is cleared automatically when an exception is taken, to delay recognition of conditions causing those exceptions.
- A machine check exception can only occur if the machine check enable bit, MSR[ME], is set. If MSR[ME] is cleared, the processor goes directly into checkstop state when a machine check exception condition occurs.

#### TEMPORARY 64-BIT BRIDGE

Bit 2 of the MSR (MSR[ISF]) may optionally be used by a 64-bit implementation to control the mode (64-bit or 32-bit) that is entered when an exception is taken. If this bit is implemented, it has the following properties:

- When an exception is taken, the value of MSR[ISF] is copied to MSR[SF].
- When an exception is taken, MSR[ISF] is not altered.
- No software synchronization is required before or after altering MSR[ISF]. Refer to Section 2.3.18, “Synchronization Requirements for Special Registers and for Lookaside Buffers,” for more information on synchronization requirements for altering other bits in the MSR.

If the MSR[ISF] bit is not implemented, it is treated as reserved except that the value is assumed to be 1 for exception processing.
6.2.2 Steps for Exception Processing

After it is determined that the exception can be taken (by confirming that any instruction-caused exceptions occurring earlier in the instruction stream have been handled, and by confirming that the exception is enabled for the exception condition), the processor does the following:

1. The machine status save/restore register 0 (SRR0) is loaded with an instruction address that depends on the type of exception. See the individual exception description for details about how this register is used for specific exceptions.

2. Bits 33–36 and 42–47 of SRR1 are loaded with information specific to the exception type.

3. Bits 0–32, 37–41, and 48–63 of SRR1 are loaded with a copy of the corresponding bits of the MSR. Note that depending on the implementation, reserved bits may not be copied.

4. The MSR is set as described in Table 6-6. The new values take effect beginning with the fetching of the first instruction of the exception-handler routine located at the exception vector address.
   
   Note that MSR[IR] and MSR[DR] are cleared for all exception types; therefore, address translation is disabled for both instruction fetches and data accesses beginning with the first instruction of the exception-handler routine.

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

5. Instruction fetch and execution resumes, using the new MSR value, at a location specific to the exception type. The location is determined by adding the exception's vector offset (see Table 6-2) to the base address determined by MSR[IP]. If IP is cleared, exceptions are vectored to the physical address 0x0000_0000_000n_nnnn in 64-bit implementations and 0x000n_nnnn in 32-bit implementations. If IP is set, exceptions are vectored to the physical address 0xFFFF_FFFF_FFFn_nnnn in 64-bit implementations and 0xFFFn_nnnn in 32-bit implementations. For a machine check exception that occurs when MSR[ME] = 0 (machine check exceptions are disabled), the checkstop state is entered (the machine stops executing instructions). See Section 6.4.2, “Machine Check Exception (0x00200).”

In some implementations, any instruction fetch with MSR[IR] = 1 and any load or store with MSR[DR] = 1 may cause SRR0 and SRR1 to be modified.
6.2.3 Returning from an Exception Handler

The Return from Interrupt (\texttt{rfid} [or \texttt{rfi}]) instruction performs context synchronization by allowing previously issued instructions to complete before returning to the interrupted process. Execution of the \texttt{rfid} (or \texttt{rfi}) instruction ensures the following:

- All previous instructions have completed to a point where they can no longer cause an exception. If a previous instruction causes a direct-store interface error exception, the results are determined before this instruction is executed.
- Previous instructions complete execution in the context (privilege, protection, and address translation) under which they were issued.
- The \texttt{rfid} (or \texttt{rfi}) instruction copies SRR1 bits back into the MSR.
- The instructions following this instruction execute in the context established by this instruction.

For a complete description of context synchronization, refer to Section 6.1.2.1, “Context Synchronization.”

**TEMPORARY 64-BIT BRIDGE**

The 64-bit bridge facility affects the operation of the return from exception mechanism in that the \texttt{rfi} instruction can optionally be allowed to execute in 64-bit implementations. In this case, the \texttt{mtmsr} instruction must also be implemented. When these instructions are implemented on a 64-bit implementation, their operation is identical to their operation in a 32-bit implementation. For an \texttt{rfi} instruction, in addition to the actions described above, the following occurs:

- The SRR1 bits that are copied to the corresponding bits of the MSR are bits 32, 37–41, and 48–63. The remaining bits of the MSR, including the high-order 32 bits are unchanged.
- If the new MSR value does not enable any pending exceptions, then the next instruction is fetched, under control of the new MSR value from the address specified in SRR0[0–61] concatenated with 0b00 (when MSR[SF] = 1 in the new MSR value). Alternately, when MSR[SF] = 0 in the new MSR value, the next instruction is fetched from the address specified by thirty-two 0’s concatenated with SRR0[32–61], concatenated with 0b00.
6.3 Process Switching

The operating system should execute the following when processes are switched:

- The **sync** instruction, which orders the effects of instruction execution. All instructions previously initiated appear to have completed before the **sync** instruction completes, and no subsequent instructions appear to be initiated until the **sync** instruction completes.

- The **isync** instruction, which waits for all previous instructions to complete and then discards any fetched instructions, causing subsequent instructions to be fetched (or refetched) from memory and to execute in the context (privilege, translation, protection, etc.) established by the previous instructions.

- The **stwcx/**stdcx. instruction, to clear any outstanding reservations, which ensures that an **lwarx/**ldarx instruction in the old process is not paired with an **stwcx/**stdcx. instruction in the new process.

The operating system should handle MSR[RI] as follows:

- In machine check and system reset exception handlers—If the SRR1 bit corresponding to MSR[RI] is cleared, the exception is not recoverable.

- In each exception handler—When enough state information has been saved that a machine check or system reset exception can reconstruct the previous state, set MSR[RI].

- At the end of each exception handler—Clear MSR[RI], set the SRR0 and SRR1 registers appropriately, and then execute **rfid** (or **rfi**).

Note that the RI bit being set indicates that, with respect to the processor, enough processor state data is valid for the processor to continue, but it does not guarantee that the interrupted process can resume.
6.4 Exception Definitions

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

<table>
<thead>
<tr>
<th>Exception Type</th>
<th>MSR Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SF1,2 ISP2 POW ILE EE PR FP ME FE0 SE BE FE1 IP IR DR RI LE</td>
</tr>
<tr>
<td>System reset</td>
<td>1 — 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ILE</td>
</tr>
<tr>
<td>Machine check</td>
<td>1 — 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ILE</td>
</tr>
<tr>
<td>DSI</td>
<td>1 — 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ILE</td>
</tr>
<tr>
<td>ISI</td>
<td>1 — 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ILE</td>
</tr>
<tr>
<td>External</td>
<td>1 — 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ILE</td>
</tr>
<tr>
<td>Alignment</td>
<td>1 — 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ILE</td>
</tr>
<tr>
<td>Program</td>
<td>1 — 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ILE</td>
</tr>
<tr>
<td>Floating-point</td>
<td>1 — 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ILE</td>
</tr>
<tr>
<td>unavailable</td>
<td>1 — 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ILE</td>
</tr>
<tr>
<td>Decrementer</td>
<td>1 — 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ILE</td>
</tr>
<tr>
<td>System call</td>
<td>1 — 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ILE</td>
</tr>
<tr>
<td>Trace exception</td>
<td>1 — 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ILE</td>
</tr>
<tr>
<td>Floating-point</td>
<td>1 — 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ILE</td>
</tr>
<tr>
<td>assist exception</td>
<td>1 — 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ILE</td>
</tr>
</tbody>
</table>

0 Bit is cleared.
1 Bit is set.
ILE Bit is copied from the ILE bit in the MSR.
— Bit is not altered.
Reading of reserved bits may return 0, even if the value last written to it was 1.

\[1^{64}\text{-bit implementations only.}\]

TEMPORARY 64-BIT BRIDGE

2 When the 64-bit bridge is implemented in a 64-bit processor and the MSR[ISF] bit is implemented, the value of the MSR[ISF] bit is copied to the MSR[SF] bit when an exception is taken.
6.4.1 System Reset Exception (0x00100)

The system reset exception is a nonmaskable, asynchronous exception signaled to the processor typically through the assertion of a system-defined signal; see Table 6-7.

Table 6-7. System Reset Exception—Register Settings

<table>
<thead>
<tr>
<th>Register</th>
<th>Setting Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRR0</td>
<td>Set to the effective address of the instruction that the processor would have attempted to execute next if no exception conditions were present.</td>
</tr>
<tr>
<td>SRR1</td>
<td></td>
</tr>
<tr>
<td>0–32</td>
<td>Loaded with equivalent bits from the MSR</td>
</tr>
<tr>
<td>33–36</td>
<td>Cleared</td>
</tr>
<tr>
<td>37–41</td>
<td>Loaded with equivalent bits from the MSR</td>
</tr>
<tr>
<td>42–47</td>
<td>Cleared</td>
</tr>
<tr>
<td>48–61</td>
<td>Loaded with equivalent bits from the MSR</td>
</tr>
<tr>
<td>62</td>
<td>Loaded from the equivalent MSR bit, MSR[RI], if the exception is recoverable; otherwise cleared.</td>
</tr>
<tr>
<td>63</td>
<td>Loaded with equivalent bit from the MSR</td>
</tr>
<tr>
<td>Note that depending on the implementation, reserved bits in the MSR may not be copied to SRR1. If the processor state is corrupted to the extent that execution cannot resume reliably, the bit corresponding to MSR[RI], (SRR1[62]), is cleared.</td>
<td></td>
</tr>
</tbody>
</table>

MSR | SF * 1  | PR 0  | SE 0  | IR 0  |
| ISF — | FP 0  | BE 0  | DR 0  |
| POW 0 | ME —  | FE1 0 | RI 0  |
| ILE — | FE0 0 | IP —  | LE Set to value of ILE |

*2 If the MSR[ISF] bit is implemented, the value of the MSR[ISF] bit is copied to the MSR[SF] bit when an exception is taken.

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

If the exception is recoverable, the value of the MSR[RI] bit is copied to the corresponding SRR1 bit. The exception functions as a context-synchronizing operation. If a reset exception causes the loss of:

- an external exception (interrupt or decremter),
- direct-store error type DSI, or
- floating-point enabled type program exception,

then the exception is not recoverable. If the SRR1 bit corresponding to MSR[RI] is cleared, the exception is context-synchronizing only with respect to subsequent instructions. Note that each implementation provides a means for software to distinguish between power-on reset and other types of system resets (such as soft reset).
6.4.2 Machine Check Exception (0x00200)

If no higher-priority exception is pending (namely, a system reset exception), the processor initiates a machine check exception when the appropriate condition is detected. Note that the causes of machine check exceptions are implementation- and system-dependent, and are typically signalled to the processor by the assertion of a specified signal on the processor interface.

When a machine check condition occurs and MSR[ME] = 1, the exception is recognized and handled. If MSR[ME] = 0 and a machine check occurs, the processor generates an internal checkstop condition. When a processor is in checkstop state, instruction processing is suspended and generally cannot continue without resetting the processor. Some implementations may preserve some or all of the internal state of the processor when entering the checkstop state, so that the state can be analyzed as an aid in problem determination.

In general, it is expected that a bus error signal would be used by a memory controller to indicate a memory parity error or an uncorrectable memory ECC error. Note that the resulting machine check exception has priority over any exceptions caused by the instruction that generated the bus operation.

If a machine check exception causes an exception that is not context-synchronizing, the exception is not recoverable. Also, if a machine check exception causes the loss of:

- an external exception (interrupt or decrementer),
- direct-store error type DSI, or
- floating-point enabled type program exception,

then the exception is not recoverable. If the SRR1 bit corresponding to MSR[RI] is cleared, the exception is context-synchronizing only with respect to subsequent instructions. If the exception is recoverable, the SRR1 bit corresponding to MSR[RI] is set and the exception is context-synchronizing.

On some implementations, a machine check exception may be caused by referring to a nonexistent physical (real) address, either because translation is disabled (MSR[IR] or MSR[DR] = 0) or through an invalid translation. On such a system, execution of the `dcbz` instruction can cause a delayed machine check exception by introducing a block into the data cache that is associated with an invalid physical (real) address. A machine check exception could eventually occur when and if a subsequent attempt is made to store that block to memory.
When a machine check exception is taken, registers are updated as shown in Table 6-8.

<table>
<thead>
<tr>
<th>Register</th>
<th>Setting Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRR0</td>
<td>On a best-effort basis, implementations can set this to an EA of some instruction that was executing or about to be executing when the machine check condition occurred.</td>
</tr>
<tr>
<td>SRR1</td>
<td>Bit 62 is loaded from MSR[RI] if the processor is in a recoverable state. Otherwise cleared. The setting of all other SRR1 bits is implementation-dependent.</td>
</tr>
<tr>
<td>MSR</td>
<td>SF (^1) 1 PR 0 SE 0 IR 0 ISF (^1) — FP 0 BE 0 DR 0 POW 0 ME (^2) — FE1 0 RI 0 ILE — FE0 0 IP — LE Set to value of ILE EE 0</td>
</tr>
</tbody>
</table>

\(^1\) If the MSR[ISF] bit is implemented, the value of the MSR[ISF] bit is copied to the MSR[SF] bit when an exception is taken.

\(^2\) Note that when a machine check exception is taken, the exception handler should set MSR[ME] as soon as it is practical to handle another machine check exception. Otherwise, subsequent machine check exceptions cause the processor to automatically enter the checkstop state.

If MSR[RI] is set, the machine check exception may still be unrecoverable in the sense that execution cannot resume in the same context that existed before the exception.

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

### 6.4.3 DSI Exception (0x00300)

A DSI exception occurs when no higher priority exception exists and a data memory access cannot be performed. The condition that caused the DSI exception can be determined by reading the DSISR, a supervisor-level SPR (SPR18) that can be read by using the \texttt{mfspr} instruction. Bit settings are provided in Table 6-9. Table 6-9 also indicates which memory element is pointed to by the DAR. DSI exceptions can be generated by load/store instructions, cache-control instructions (\texttt{icbi}, \texttt{dcbi}, \texttt{dcbz}, \texttt{dcbst}, and \texttt{dcbf}), or the \texttt{eciwx/ecowx} instructions for any of the following reasons:

- A load or a store instruction results in a direct-store error exception.
- The effective address cannot be translated. That is, there is a page fault for this portion of the translation, so a DSI exception must be taken to retrieve the translation, for example from a storage device such as a hard disk drive.
• The instruction is not supported for the type of memory addressed.
  — For `lwarx/stwcx` and `ldarx/stdcx`, instructions that reference a memory location that is write-through required. If the exception is not taken, the instructions execute correctly.
  — For `lwarx/stwcx`, `ldarx/stdcx`, or `eciwx/ecowx` instructions that attempt to access direct-store segments. If the exception does not occur, the results are boundedly undefined.

• The access violates memory protection.

• The execution of an `eciwx` or `ecowx` instruction is disallowed because the external access register enable bit (EAR[E]) is cleared.

• A data address breakpoint register (DABR) match occurred. The DABR facility is optional to the PowerPC architecture, but if one is implemented, it is recommended, but not required, that it be implemented as follows. A data address breakpoint match is detected for a load or store instruction if the three following conditions are met for any byte accessed:
  — EA[0–60] = DABR[DAB]
  — MSR[DR] = DABR[BT]
  — The instruction is a store and DABR[DW] = 1, or the instruction is a load and DABR[DR] = 1.

The DABR is described in Section 2.3.15, “Data Address Breakpoint Register (DABR).” In 32-bit mode of 64-bit implementations, the high-order 32 bits of the EA are treated as zero for the purpose of detecting a match; the DAR settings are described in Table 6-9. If the above conditions are satisfied, it is undefined whether a match occurs in the following cases:
  — The instruction is store conditional but the store is not performed.
  — The instruction is a load/store string of zero length.
  — The instruction is `dcbz`, `eciwx`, or `ecowx`.

The cache management instructions other than `dcbz` never cause a match. If `dcbz` causes a match, some or all of the target memory locations may have been updated. For the purpose of determining whether a match occurs, `eciwx` is treated as a load, and `ecowx` and `dcbz` are treated as stores.

If an `stwcx/stdcx` instruction has an EA for which a normal store operation would cause a DSI exception but the processor does not have the reservation from `lwarx/ldarx`, whether a DSI exception is taken is implementation-dependent.

If the value in XER[25–31] indicates that a load or store string instruction has a length of zero, a DSI exception does not occur, regardless of the effective address.
The condition that caused the exception is defined in the DSISR. As shown in Table 6-9, this exception also sets the data address register (DAR).

### Table 6-9. DSI Exception—Register Settings

<table>
<thead>
<tr>
<th>Register</th>
<th>Setting Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRR0</td>
<td>Set to the effective address of the instruction that caused the exception.</td>
</tr>
<tr>
<td>SRR1</td>
<td>0–32: Loaded with equivalent bits from the MSR. 33–36: Cleared. 37–41: Loaded with equivalent bits from the MSR. 42–47: Cleared. 48–63: Loaded with equivalent bits from the MSR. Note that depending on the implementation, reserved bits in the MSR may not be copied to SRR1.</td>
</tr>
<tr>
<td>MSR</td>
<td>SF *</td>
</tr>
<tr>
<td></td>
<td>ISF  —</td>
</tr>
<tr>
<td></td>
<td>POW  0</td>
</tr>
<tr>
<td></td>
<td>ILE  —</td>
</tr>
<tr>
<td></td>
<td>EE    0</td>
</tr>
</tbody>
</table>

**TEMPORARY 64-BIT BRIDGE**

* If the MSR[ISF] bit is implemented, the value of the MSR[ISF] bit is copied to the MSR[SF] bit when an exception is taken.

| DSISR | 0 | Set if a load or store instruction results in a direct-store error exception; otherwise cleared. |
|       | 1 | Set if the translation of an attempted access is not found in the primary hash table entry group (HTEG), or in the rehashed secondary HTEG, or in the range of a DBAT register; otherwise cleared. |
|       | 2–3 | Cleared |
|       | 4 | Set if a memory access is not permitted by the page or DBAT protection mechanism; otherwise cleared. |
|       | 5 | Set if the ecixw, ecowx, lwarx/ldatrx, or stwcx/stdcx, instruction is attempted to direct-store interface space, or if the lwarx/ldatrx or stwcx/stdcx, instruction is used with addresses that are marked as write-through. Otherwise cleared to 0. |
|       | 6 | Set for a store operation and cleared for a load operation. |
|       | 7–8 | Cleared |
|       | 9 | Set to 1 if a DABR match occurred. Otherwise cleared to 0. |
|       | 10 | For 64-bit implementations, set if the segment table search fails to find a translation for the effective address; otherwise cleared. Cleared in 32-bit implementations. |
|       | 11 | Set if the instruction was an ecixw or ecowx and EAR[E] = 0; otherwise cleared. |
|       | 12–31 | Cleared |

Due to the multiple exception conditions possible from the execution of a single instruction, the following combinations of bits of DSISR may be set concurrently:

- Bits 1 and 11
- Bits 4 and 5
- Bits 4 and 11
- Bits 5 and 11
- Bits 10 and 11

Additionally, bit 6 is set if the instruction that caused the exception is a store, ecowx, dcbz, or dcbi and bit 6 would otherwise be cleared. Also, bit 9 (DABR match) may be set alone, or in combination with any other bit, or with any of the other combinations shown above.
When a DSI exception is taken, instruction execution resumes at offset 0x00300 from the physical base address indicated by MSR[IP].

### 6.4.4 ISI Exception (0x00400)

An ISI exception occurs when no higher priority exception exists and an attempt to fetch the next instruction to be executed fails for any of the following reasons:

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

---

<table>
<thead>
<tr>
<th>Register</th>
<th>Setting Description</th>
</tr>
</thead>
</table>
| DAR      | Set to the effective address of a memory element as described in the following list:  
  - A byte in the first word accessed in the page that caused the DSI exception, for a byte, half word, or word memory access to an ordinary memory segment.  
  - A byte in the first double word accessed in the page that caused the DSI exception, for a double-word memory access to an ordinary memory segment.  
  - A byte in the first word accessed in the BAT area that caused the DSI exception for a byte, half word, or word access to a BAT area.  
  - A byte in the first double word accessed in the BAT area that caused the DSI exception, or a double-word access to a BAT area.  
  - A byte in the block that caused the exception for icbi, dcbz, dcbst, dcbf, or dcbi instructions.  
  - Any EA in the memory range addressed (for direct-store error exceptions).  
  - The EA computed by the instruction for the attempted execution of an eciwx or ecowx instruction when EAR[E] is cleared.  
  - If the exception is caused by a DABR match, the DAR is set to the effective address of any byte in the range from A to B inclusive, where A is the effective address of the word (for a byte, half word, or word access) or double word (for a double word access) specified by the EA computed by the instruction, and B is the EA of the last byte in the word or double word in which the match occurred.  
  Note that if the exception occurs when a 64-bit processor is running in 32-bit mode, the 32 high-order bits are cleared. |

---

Table 6-9. DSI Exception—Register Settings (Continued)
Register settings for ISI exceptions are shown in Table 6-10.

### Table 6-10. ISI Exception—Register Settings

<table>
<thead>
<tr>
<th>Register</th>
<th>Setting Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRR0</td>
<td>Set to the effective address of the instruction that the processor would have attempted to execute next if no exception conditions were present (if the exception occurs on attempting to fetch a branch target, SRR0 is set to the branch target address).</td>
</tr>
<tr>
<td>SRR1</td>
<td>0–32 Cleared &lt;br&gt;33 Set if the translation of an attempted access is not found in the primary hash table entry group (HTEG), or in the rehashed secondary HTEG, or in the range of an IBAT register; otherwise cleared. &lt;br&gt;34 Cleared &lt;br&gt;35 Set if the fetch access was to a direct-store segment (SR[T] = 1), to a no-execute segment (N bit set in segment descriptor), or to guarded memory when MSR[IR] = 1. Otherwise, cleared. &lt;br&gt;36 Set if a memory access is not permitted by the page or IBAT protection mechanism, described in Chapter 7, &quot;Memory Management&quot;; otherwise cleared. &lt;br&gt;37–41 Cleared &lt;br&gt;42 For 64-bit implementations, set if the segment table search fails to find a translation for the effective address; otherwise cleared. &lt;br&gt;43–47 Cleared &lt;br&gt;48–63 Loaded from 16 least-significant bits of the MSR</td>
</tr>
<tr>
<td>MSR</td>
<td>SF * 1 PR 0 SE 0 IR 0 &lt;br&gt;ISF * — FP 0 BE 0 DR 0 &lt;br&gt;POW 0 ME — FE1 0 RI 0 &lt;br&gt;ILE — FE0 0 IP — LE Set to value of ILE</td>
</tr>
</tbody>
</table>

> TEMPORARY 64-BIT BRIDGE

* If the MSR[ISF] bit is implemented, the value of the MSR[ISF] bit is copied to the MSR[SF] bit when an exception is taken.

When an ISI exception is taken, instruction execution resumes at offset 0x00400 from the physical base address indicated by MSR[IP].

### 6.4.5 External Interrupt (0x00500)

An external interrupt exception is signaled to the processor by the assertion of the external interrupt signal. The exception may be delayed by other higher priority exceptions or if the MSR[EE] bit is zero when the exception is detected. Note that the occurrence of this exception does not cancel the external request.
The register settings for the external interrupt exception are shown in Table 6-11.

Table 6-11. External Interrupt—Register Settings

<table>
<thead>
<tr>
<th>Register</th>
<th>Setting Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRR0</td>
<td>Set to the effective address of the instruction that the processor would have attempted to execute next if no interrupt conditions were present.</td>
</tr>
<tr>
<td>SRR1</td>
<td>0–32: Loaded with equivalent bits from the MSR 33–36: Cleared 37–41: Loaded with equivalent bits from the MSR 42–47: Cleared 48–63: Loaded with equivalent bits from the MSR</td>
</tr>
<tr>
<td>Note that depending on the implementation, reserved bits in the MSR may not be copied to SRR1.</td>
<td></td>
</tr>
<tr>
<td>MSR</td>
<td>SF: Set to 1, if the MSR[ISF] bit is implemented, the value of the MSR[ISF] bit is copied to the MSR[SF] bit when an exception is taken. PR: Set to value of IPR SE: Set to value of ISE IR: Set to value of IIR</td>
</tr>
</tbody>
</table>

When an external interrupt exception is taken, instruction execution resumes at offset 0x00500 from the physical base address indicated by MSR[IP].

6.4.6 Alignment Exception (0x00600)

This section describes conditions that can cause alignment exceptions in the processor. Similar to DSI exceptions, alignment exceptions use the SRR0 and SRR1 to save the machine state and the DSISR to determine the source of the exception. An alignment exception occurs when no higher priority exception exists and the implementation cannot perform a memory access for one of the following reasons:

- The operand of a floating-point load or store instruction is not word-aligned.
- The operand of an integer double-word load or store instruction is not word-aligned.
- The operand of \texttt{lmw, stmw, lwarx, ldarx, stwcx, stdcx, eciwx, or ecowx} is not aligned.
- The instruction is \texttt{lmw, stmw, lswi, lsrx, stswi, or stswx} and the processor is in little-endian mode.
- The operand of a floating-point load or store instruction is in a direct-store segment (T = 1).
- The operand of an elementary or string load or store crosses a protection boundary.
- The operand of \texttt{lmw} or \texttt{stmw} crosses a segment or BAT boundary.
- The operand of \texttt{dcbz} is in memory that is write-through-required or caching inhibited, or \texttt{dcbz} is executed in an implementation that has either no data cache or a write-through data cache.
For **lmw**, **stmw**, **lswi**, **stswi**, and **stswx** instructions in little-endian mode, an alignment exception always occurs. For **lmw** and **stmw** instructions with an operand that is not aligned in big-endian mode, and for **lwarx**, **ldarx**, **stwcx**, **stdcx**, **eciwx**, and **ecowx** with an operand that is not aligned in either endian mode, an implementation may yield boundedly-undefined results instead of causing an alignment exception (for **eciwx** and **ecowx** when EAR[E] = 0, a third alternative is to cause a DSI exception). For all other cases listed above, an implementation may execute the instruction correctly instead of causing an alignment exception. For the **dcbz** instruction, correct execution means clearing each byte of the block in main memory. See Section 3.1, “Data Organization in Memory and Data Transfers,” for a complete definition of alignment in the PowerPC architecture.

The term, ‘protection boundary’, refers to the boundary between protection domains. A protection domain is a segment, a block of memory defined by a BAT entry, a virtual 4-Kbyte page, or a range of unmapped effective addresses. Protection domains are defined only when the corresponding address translation (instruction or data) is enabled (MSR[IR] or MSR[DR] = 1).

The register settings for alignment exceptions are shown in Table 6-12.

**Table 6-12. Alignment Exception—Register Settings**

<table>
<thead>
<tr>
<th>Register</th>
<th>Setting Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRR0</td>
<td>Set to the effective address of the instruction that caused the exception.</td>
</tr>
<tr>
<td>SRR1 0–32</td>
<td>Loaded with equivalent bits from the MSR</td>
</tr>
<tr>
<td>SRR1 33–36</td>
<td>Cleared</td>
</tr>
<tr>
<td>SRR1 37–41</td>
<td>Loaded with equivalent bits from the MSR</td>
</tr>
<tr>
<td>SRR1 42–47</td>
<td>Cleared</td>
</tr>
<tr>
<td>SRR1 48–63</td>
<td>Loaded with equivalent bits from the MSR</td>
</tr>
</tbody>
</table>

Note that depending on the implementation, reserved bits in the MSR may not be copied to SRR1.

| MSR SF 0 | PR 0 | SE 0 | IR 0 |
| MSR SF 1 | PR 0 | SE 0 | IR 0 |
| MSR ISF — | FP 0 | BE 0 | DR 0 |
| MSR POW 0 | ME — | FE1 0 | RI 0 |
| MSR ILE — | FE0 0 | IP — | LE Set to value of ILE |
| MSR EE 0 | — | — | — |
The architecture does not support the use of a misaligned EA by load/store with reservation instructions or by the `eciwx` and `ecowx` instructions. If one of these instructions specifies a misaligned EA, the exception handler should not emulate the instruction but should treat the occurrence as a programming error.

The instruction pairs that can use the same DSISR values are as follows:

<table>
<thead>
<tr>
<th>Instruction Pairs</th>
<th>DSISR Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>lbz/lbzx</td>
<td>lbz/lbzux</td>
</tr>
<tr>
<td>lhz/lhux</td>
<td>lwa/lwax</td>
</tr>
<tr>
<td>ld/ldx</td>
<td>ldu/ldux</td>
</tr>
<tr>
<td>stb/stbx</td>
<td>sth/sthx</td>
</tr>
<tr>
<td>std/stdux</td>
<td>lfs/lf sx</td>
</tr>
<tr>
<td>stfs/stf sx</td>
<td>stfd/stfx</td>
</tr>
</tbody>
</table>

Note that for load or store instructions that use register indirect with index addressing, the DSISR can be set to the same value that would have resulted if the corresponding instruction uses register indirect with immediate index addressing had caused the exception. Similarly, for load or store instructions that use register indirect with immediate index addressing, DSISR can hold a value that would have resulted from an instruction that uses register indirect with index addressing. For example, a misaligned `lwarx` instruction that crosses a protection boundary would normally cause the DSISR to be set to the following binary value:

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

The value ttttt refers to the destination and ???? indicates undefined bits. However, this register may be set as if the instruction were `lwa`, as follows:

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

If there is no corresponding instruction (such as for the `lwaux` instruction), no alternative value can be specified.

The instruction pairs that can use the same DAR values are as follows:

<table>
<thead>
<tr>
<th>Instruction Pairs</th>
<th>DAR Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>lbz/lbzx</td>
<td>lbz/lbzux</td>
</tr>
<tr>
<td>lhz/lhux</td>
<td>lwa/lwax</td>
</tr>
<tr>
<td>ld/ldx</td>
<td>ldu/ldux</td>
</tr>
<tr>
<td>stb/stbx</td>
<td>sth/sthx</td>
</tr>
<tr>
<td>std/stdux</td>
<td>lfs/lf sx</td>
</tr>
<tr>
<td>stfs/stf sx</td>
<td>std/stdx</td>
</tr>
</tbody>
</table>

Note that if a 64-bit processor is running in 32-bit mode, the 32 high-order bits are cleared.

The architecture does not support the use of a misaligned EA by load/store with reservation instructions or by the `eciwx` and `ecowx` instructions. If one of these instructions specifies a misaligned EA, the exception handler should not emulate the instruction but should treat the occurrence as a programming error.
6.4.6.1 Integer Alignment Exceptions
Operations that are not naturally aligned may suffer performance degradation, depending on the processor design, the type of operation, the boundaries crossed, and the mode that the processor is in during execution. More specifically, these operations may either cause an alignment exception or they may cause the processor to break the memory access into multiple, smaller accesses with respect to the cache and the memory subsystem.

6.4.6.1.1 Direct-Store Interface Access Considerations
The following apply for direct-store interface accesses:

- If a 256-Mbyte boundary will be crossed by any portion of the direct-store interface space accessed by an instruction (the entire string for strings/multiples), an alignment exception is taken.
- Floating-point loads and stores to direct-store segments may cause an alignment exception, regardless of operand alignment.
- The load/store word/double word with reservation instructions that map into a direct-store segment always cause a DSI exception. However, if the instruction crosses a segment boundary an alignment exception is taken instead.

6.4.6.1.2 Page Address Translation Access Considerations
A page address translation access occurs when MSR[DR] is set, SR[T] is cleared, and there is no BAT match. Note that a dcbz instruction causes an alignment exception if the access is to a page or block with the W (write-through) or I (cache-inhibit) bit set.

Misaligned memory accesses that do not cause an alignment exception may not perform as well as an aligned access of the same type. The resulting performance degradation due to misaligned accesses depends on how well each individual access behaves with respect to the memory hierarchy.

Particular details regarding page address translation is implementation-dependent; the reader should consult the user’s manual for the appropriate processor for more information.

6.4.6.2 Little-Endian Mode Alignment Exceptions
The OEA allows implementations to take alignment exceptions on misaligned accesses (as described in Section 3.1.4, “PowerPC Byte Ordering”) in little-endian mode but does not require them to do so. Some implementations may perform some misaligned accesses without taking an alignment exception.
6.4.6.3  Interpretation of the DSISR as Set by an Alignment Exception

For most alignment exceptions, an exception handler may be designed to emulate the instruction that causes the exception. To do this, the handler requires the following characteristics of the instruction:

- Load or store
- Length (half word, word, or double word)
- String, multiple, or normal load/store
- Integer or floating-point
- Whether the instruction performs update
- Whether the instruction performs byte reversal
- Whether it is a `dcbz` instruction

The PowerPC architecture provides this information implicitly, by setting opcode bits in the DSISR that identify the excepting instruction type. The exception handler does not need to load the excepting instruction from memory. The mapping for all exception possibilities is unique except for the few exceptions discussed below.

Table 6-13 shows the inverse mapping—how the DSISR bits identify the instruction that caused the exception.

The alignment exception handler cannot distinguish a floating-point load or store that causes an exception because it is misaligned, or because it addresses the direct-store interface space. However, this does not matter; in either case it is emulated with integer instructions.

<table>
<thead>
<tr>
<th>DSISR[15–21]</th>
<th>Instruction</th>
<th>DSISR[15–21]</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 0 0000</td>
<td>lwarx, lwz, special cases¹</td>
<td>01 1 0010</td>
<td>stdux</td>
</tr>
<tr>
<td>00 0 0010</td>
<td>ldarx</td>
<td>01 1 0101</td>
<td>lwaux</td>
</tr>
<tr>
<td>00 0 0010</td>
<td>stw</td>
<td>10 0 0010</td>
<td>stwcx</td>
</tr>
<tr>
<td>00 0 0100</td>
<td>lhz</td>
<td>10 0 0011</td>
<td>stdcx</td>
</tr>
<tr>
<td>00 0 0101</td>
<td>lha</td>
<td>10 0 1000</td>
<td>lwbrix</td>
</tr>
<tr>
<td>00 0 0110</td>
<td>sth</td>
<td>10 0 1010</td>
<td>stwbrx</td>
</tr>
<tr>
<td>00 0 0111</td>
<td>lmw</td>
<td>10 0 1100</td>
<td>lhbrix</td>
</tr>
<tr>
<td>00 0 1000</td>
<td>lfs</td>
<td>10 0 1110</td>
<td>stbbrx</td>
</tr>
<tr>
<td>00 0 1001</td>
<td>lfd</td>
<td>10 1 0100</td>
<td>eciwx</td>
</tr>
<tr>
<td>00 0 1010</td>
<td>stfs</td>
<td>10 1 0110</td>
<td>ecowx</td>
</tr>
<tr>
<td>00 0 1011</td>
<td>std</td>
<td>10 1 1111</td>
<td>dcbz</td>
</tr>
<tr>
<td>00 0 1101</td>
<td>ld, ldu, lwa²</td>
<td>11 0 0000</td>
<td>lwzx</td>
</tr>
</tbody>
</table>
### Table 6-13. DSISR(15–21) Settings to Determine Misaligned Instruction (Continued)

<table>
<thead>
<tr>
<th>DSISR[15–21]</th>
<th>Instruction</th>
<th>DSISR[15–21]</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 0 1111</td>
<td>std, stdu (^2)</td>
<td>11 0 0010</td>
<td>stwx</td>
</tr>
<tr>
<td>00 1 0000</td>
<td>lwzu</td>
<td>11 0 0100</td>
<td>lhzx</td>
</tr>
<tr>
<td>00 1 0010</td>
<td>stwu</td>
<td>11 0 0101</td>
<td>lhax</td>
</tr>
<tr>
<td>00 1 0100</td>
<td>lhzu</td>
<td>11 0 0110</td>
<td>sthx</td>
</tr>
<tr>
<td>00 1 0101</td>
<td>lhau</td>
<td>11 0 1000</td>
<td>lfsx</td>
</tr>
<tr>
<td>00 1 0110</td>
<td>sthu</td>
<td>11 0 1001</td>
<td>lfdx</td>
</tr>
<tr>
<td>00 1 0111</td>
<td>stmw</td>
<td>11 0 1010</td>
<td>stfsx</td>
</tr>
<tr>
<td>00 1 1000</td>
<td>lfsu</td>
<td>11 0 1011</td>
<td>stfdx</td>
</tr>
<tr>
<td>00 1 1001</td>
<td>lfdu</td>
<td>11 0 1111</td>
<td>stfiwx</td>
</tr>
<tr>
<td>00 1 1010</td>
<td>stfsu</td>
<td>11 1 0000</td>
<td>lwzux</td>
</tr>
<tr>
<td>00 1 1011</td>
<td>stfdu</td>
<td>11 1 0010</td>
<td>stwux</td>
</tr>
<tr>
<td>01 0 0000</td>
<td>ldx</td>
<td>11 1 0100</td>
<td>lhzux</td>
</tr>
<tr>
<td>01 0 0010</td>
<td>stdx</td>
<td>11 1 0101</td>
<td>lhaux</td>
</tr>
<tr>
<td>01 0 0101</td>
<td>lswap</td>
<td>11 1 0110</td>
<td>sthux</td>
</tr>
<tr>
<td>01 0 1000</td>
<td>lswx</td>
<td>11 1 1000</td>
<td>lfsux</td>
</tr>
<tr>
<td>01 0 1001</td>
<td>lswi</td>
<td>11 1 1001</td>
<td>lfdux</td>
</tr>
<tr>
<td>01 0 1010</td>
<td>stswx</td>
<td>11 1 1010</td>
<td>stfsux</td>
</tr>
<tr>
<td>01 0 1011</td>
<td>stswi</td>
<td>11 1 1011</td>
<td>stfdux</td>
</tr>
<tr>
<td>01 1 0000</td>
<td>ldux</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

\(^1\)The instructions lwz and lwarx give the same DSISR bits (all zero). But if lwarx causes an alignment exception, it is an invalid form, so it need not be emulated in any precise way. It is adequate for the alignment exception handler to simply emulate the instruction as if it were an lwz. It is important that the emulator use the address in the DAR, rather than computing it from rA/rB/D, because lwz and lwarx use different addressing modes.

If opcode 0 ("illegal or reserved") can cause an alignment exception, it will be indistinguishable to the exception handler from lwarx and lwz.

\(^2\)These instructions are distinguished by DSISR[12–13], which are not shown in this table.
6.4.7 Program Exception (0x00700)

A program exception occurs when no higher priority exception exists and one or more of the following exception conditions, which correspond to bit settings in SRR1, occur during execution of an instruction:

- System IEEE floating-point enabled exception—A system IEEE floating-point enabled exception can be generated when FPSCR[FEX] is set and either (or both) of the MSR[FE0] or MSR[FE1] bits is set.
  
  FPSCR[FEX] is set by the execution of a floating-point instruction that causes an enabled exception or by the execution of a “move to FPSCR” type instruction that sets an exception bit when its corresponding enable bit is set. Floating-point exceptions are described in Section 3.3.6, “Floating-Point Program Exceptions.”

- Illegal instruction—An illegal instruction program exception is generated when execution of an instruction is attempted with an illegal opcode or illegal combination of opcode and extended opcode fields (these include PowerPC instructions not implemented in the processor), or when execution of an optional or a reserved instruction not provided in the processor is attempted.
  
  Note that implementations are permitted to generate an illegal instruction program exception when encountering the following instructions. If an illegal instruction exception is not generated, then the alternative is shown in parenthesis.
  
  — An instruction corresponds to an invalid class (the results may be boundedly undefined)
  
  — An **lswx** instruction for which **rA** or **rB** is in the range of registers to be loaded (may cause results that are boundedly undefined)
  
  — A move to/from SPR instruction with an SPR field that does not contain one of the defined values
    
    - MSR[PR] = 1 and spr[0] = 1 (this can cause a privileged instruction program exception)
    
    - MSR[PR] = 0 or spr[0] = 0 (may cause boundedly-undefined results.)
  
  — An unimplemented floating-point instruction that is not optional (may cause a floating-point assist exception)
• Privileged instruction—A privileged instruction type program exception is generated when the execution of a privileged instruction is attempted and the processor is operating in user mode (MSR[PR] is set). It is also generated for mtspr or mfspr instructions that have an invalid SPR field that contain one of the defined values having spr[0] = 1 and if MSR[PR] = 1. Some implementations may also generate a privileged instruction program exception if a specified SPR field (for a move to/from SPR instruction) is not defined for a particular implementation, but spr[0] = 1; in this case, the implementation may cause either a privileged instruction program exception, or an illegal instruction program exception may occur instead.

• Trap—A trap program exception is generated when any of the conditions specified in a trap instruction is met. Trap instructions are described in Section 4.2.4.6, “Trap Instructions.”

The register settings when a program exception is taken are shown in Table 6-14.

**Table 6-14. Program Exception—Register Settings**

<table>
<thead>
<tr>
<th>Register</th>
<th>Setting Description</th>
</tr>
</thead>
</table>
| SRR0     | The contents of SRR0 differ according to the following situations:  
  • For all program exceptions except floating-point enabled exceptions when operating in imprecise mode (MSR[FE0] = MSR[FE1]), SRR0 contains the EA of the excepting instruction.  
  • When the processor is in floating-point imprecise mode, SRR0 may contain the EA of the excepting instruction or that of a subsequent unexecuted instruction. If the subsequent instruction is sync or isync, SRR0 points no more than four bytes beyond the sync or isync instruction.  
  • If FPSCR[FEX] = 1, but IEEE floating-point enabled exceptions are disabled (MSR[FE0] = MSR[FE1] = 0), the program exception occurs before the next synchronizing event if an instruction alters those bits (thus enabling the program exception). When this occurs, SRR0 points to the instruction that would have executed next and not to the instruction that modified MSR. |
| SRR1     | 0–32 Loaded with equivalent bits from the MSR  
  33–36 Cleared  
  37–41 Loaded with equivalent bits from the MSR  
  42 Cleared  
  Note that only one of bits 43–46 of SRR1 can be set.  
  43 Set for an IEEE floating-point enabled program exception; otherwise cleared.  
  44 Set for an illegal instruction program exception; otherwise cleared.  
  45 Set for a privileged instruction program exception; otherwise cleared.  
  46 Set for a trap program exception; otherwise cleared.  
  47 Cleared if SRR0 contains the address of the instruction causing the exception, and set if SRR0 contains the address of a subsequent instruction.  
  48–63 Loaded with equivalent bits from the MSR  
  Note that depending on the implementation, reserved bits in the MSR may not be copied to SRR1. |
| MSR      | SF * 1  
  PR 0  
  SE 0  
  IR 0  
  ISF —  
  FP 0  
  BE 0  
  DR 0  
  POW 0  
  ME —  
  FE1 0  
  RI 0  
  ILE —  
  FE0 0  
  IP —  
  LE Set to value of ILE  
  EE 0 |

* If the MSR[ISF] bit is implemented, the value of the MSR[ISF] bit is copied to the MSR[SF] bit when an exception is taken.

---

Chapter 6. Exceptions 6-37
When a program exception is taken, instruction execution resumes at offset 0x00700 from the physical base address indicated by MSR[IP].

### 6.4.8 Floating-Point Unavailable Exception (0x00800)

A floating-point unavailable exception occurs when no higher priority exception exists, an attempt is made to execute a floating-point instruction (including floating-point load, store, or move instructions), and the floating-point available bit in the MSR is cleared, (MSR[FP] = 0).

The register settings for floating-point unavailable exceptions are shown in Table 6-15.

<table>
<thead>
<tr>
<th>Register</th>
<th>Setting Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRR0</td>
<td>Set to the effective address of the instruction that caused the exception.</td>
</tr>
</tbody>
</table>
| SRR1     | 0–32 Loaded with equivalent bits from the MSR  
33–36 Cleared  
37–41 Loaded with equivalent bits from the MSR  
42–47 Cleared  
48–63 Loaded with equivalent bits from the MSR  
Note that depending on the implementation, reserved bits in the MSR may not be copied to SRR1. |
| MSR      | SF 1  
IF 0  
POW 0  
ILE —  
EE 0  
PR 0  
SE 0  
IR 0  
FP 0  
BE 0  
DR 0  
ME —  
FE1 0  
RI 0  
FE0 0  
IP —  
LE Set to value of ILE |

When a floating-point unavailable exception is taken, instruction execution resumes at offset 0x00800 from the physical base address indicated by MSR[IP].

### 6.4.9 Decrementer Exception (0x00900)

A decrementer exception occurs when no higher priority exception exists, a decrementer exception condition occurs (for example, the decrementer register has completed decrementing), and MSR[EE] = 1. The decrementer register counts down, causing an exception request when it passes through zero. A decrementer exception request remains pending until the decrementer exception is taken and then it is cancelled. The decrementer implementation meets the following requirements:

- The counters for the decrementer and the time-base counter are driven by the same fundamental time base.
- Loading a GPR from the decrementer does not affect the decrementer.
- Storing a GPR value to the decrementer replaces the value in the decrementer with the value in the GPR.
- Whenever bit 0 of the decrementer changes from 0 to 1, a decrementer exception request is signaled. If multiple decrementer exception requests are received before the first can be reported, only one exception is reported. The occurrence of a decrementer exception cancels the request.
- If the decrementer is altered by software and if bit 0 is changed from 0 to 1, an exception request is signaled.

The register settings for the decrementer exception are shown in Table 6-16.

**Table 6-16. Decrementer Exception—Register Settings**

<table>
<thead>
<tr>
<th>Register</th>
<th>Setting Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRR0</td>
<td>Set to the effective address of the instruction that the processor would have attempted to execute next if no exception conditions were present.</td>
</tr>
</tbody>
</table>
| SRR1     | 0–32 Loaded with equivalent bits from the MSR  
33–36 Cleared  
37–41 Loaded with equivalent bits from the MSR  
42–47 Cleared  
48–63 Loaded with equivalent bits from the MSR  
Note that depending on the implementation, reserved bits in the MSR may not be copied to SRR1. |
| MSR      | SF 1  
ISF  
POW 0  
ILE —  
EE 0  
PR 0  
FP 0  
ME —  
FE0 0  
SE 0  
BE 0  
FE1 0  
IR 0  
DR 0  
RI 0  
IP —  
LE Set to value of ILE |

When a decrementer exception is taken, instruction execution resumes at offset 0x00900 from the physical base address indicated by MSR[IP].

**6.4.10 System Call Exception (0x00C00)**

A system call exception occurs when a System Call (sc) instruction is executed. The effective address of the instruction following the sc instruction is placed into SRR0. MSR bits are saved in SRR1, as shown in Table 6-17. Then a system call exception is generated.
The system call exception causes the next instruction to be fetched from offset 0x00C00 from the physical base address indicated by the new setting of MSR[IP]. As with most other exceptions, this exception is context-synchronizing. Refer to Section 6.1.2.1, “Context Synchronization,” for more information on the actions performed by a context-synchronizing operation. Register settings are shown in Table 6-17.

6.4.11 Trace Exception (0x00D00)

The trace exception is optional to the PowerPC architecture, and specific information about how it is implemented can be found in user’s manuals for individual processors.

The trace exception provides a means of tracing the flow of control of a program for debugging and performance analysis purposes. It is controlled by MSR bits SE and BE as follows:

- MSR[SE] = 1: the processor generates a single-step type trace exception after each instruction that completes without causing an exception or context change (such as occurs when an `sc`, `rfid` (or `rfi`), or a load instruction that causes an exception, for example, is executed).
- MSR[BE] = 1: the processor generates a branch-type trace exception after completing the execution of a branch instruction, whether or not the branch is taken.

---

### Table 6-17. System Call Exception—Register Settings

<table>
<thead>
<tr>
<th>Register</th>
<th>Setting Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRR0</td>
<td>Set to the effective address of the instruction following the System Call instruction</td>
</tr>
<tr>
<td>SRR1</td>
<td>0–32: Loaded with equivalent bits from the MSR 33–36: Cleared 37–41: Loaded with equivalent bits from the MSR 42–47: Cleared 48–63: Loaded with equivalent bits from the MSR Note that depending on the implementation, reserved bits in the MSR may not be copied to SRR1.</td>
</tr>
<tr>
<td>MSR SF *</td>
<td>1</td>
</tr>
<tr>
<td>MSR ISF</td>
<td>—</td>
</tr>
<tr>
<td>MSR POW</td>
<td>0</td>
</tr>
<tr>
<td>MSR ILE</td>
<td>0</td>
</tr>
<tr>
<td>MSR SE</td>
<td>0</td>
</tr>
<tr>
<td>MSR IR</td>
<td>0</td>
</tr>
<tr>
<td>MSR FP</td>
<td>0</td>
</tr>
<tr>
<td>MSR BE</td>
<td>0</td>
</tr>
<tr>
<td>MSR DR</td>
<td>0</td>
</tr>
<tr>
<td>MSR FE</td>
<td>0</td>
</tr>
<tr>
<td>MSR IP</td>
<td>—</td>
</tr>
<tr>
<td>MSR LE</td>
<td>Set to value of ILE</td>
</tr>
</tbody>
</table>

* If the MSR[ISF] bit is implemented, the value of the MSR[ISF] bit is copied to the MSR[SF] bit when an exception is taken.
If this facility is implemented, a trace exception occurs when no higher priority exception exists and either of the conditions described above exist. The following are not traced:

- `rfid` (or `rfi`) instruction
- `sc`, and trap instructions that trap
- Other instructions that cause exceptions (other than trace exceptions)
- The first instruction of any exception handler
- Instructions that are emulated by software

MSR[SE, BE] are both cleared when the trace exception is taken. In the normal use of this function, MSR[SE, BE] are restored when the exception handler returns to the interrupted program using an `rfid` (or `rfi`) instruction.

Register settings for the trace mode are described in Table 6-18.

<table>
<thead>
<tr>
<th>Register</th>
<th>Setting Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRR0</td>
<td>Set to the effective address of the next instruction to be executed in the program for which the trace exception was generated.</td>
</tr>
<tr>
<td>SRR1</td>
<td>0–32: Loaded with equivalent bits from the MSR 33–36: Implementation-specific information 37–41: Loaded with equivalent bits from the MSR 42–47: Implementation-specific information 48–63: Loaded with equivalent bits from the MSR</td>
</tr>
<tr>
<td>MSR</td>
<td>SF: 1  PR: 0  SE: 0  IR: 0  ISF: —  FP: 0  BE: 0  DR: 0  POW: 0  ME: —  FE1: 0  RI: 0  ILE: —  FE0: 0  IP: —  LE: Set to value of ILE  EE: 0</td>
</tr>
</tbody>
</table>

Temporary 64-Bit Bridge
* If the MSR[ISF] bit is implemented, the value of the MSR[ISF] bit is copied to the MSR[SF] bit when an exception is taken.

When a trace exception is taken, instruction execution resumes at offset 0x00D00 from the base address indicated by MSR[IP].

**6.4.12 Floating-Point Assist Exception (0x00E00)**

The floating-point assist exception is optional to the PowerPC architecture. It can be used to allow software to assist in the following situations:

- Execution of floating-point instructions for which an implementation uses software routines to perform certain operations, such as those involving denormalization.
- Execution of floating-point instructions that are not optional and are not implemented in hardware. In this case, the processor may generate an illegal instruction type program exception instead.
Register settings for the floating-point assist exceptions are described in Table 6-19.

**Table 6-19. Floating-Point Assist Exception—Register Settings**

<table>
<thead>
<tr>
<th>Register</th>
<th>Setting Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRR0</td>
<td>Set to the address of the next instruction to be executed in the program for which the floating-point assist exception was generated.</td>
</tr>
</tbody>
</table>
| SRR1     | 0–32: Loaded with equivalent bits from the MSR  
37–41: Loaded with equivalent bits from the MSR  
48–63: Loaded with equivalent bits from the MSR  
Note that depending on the implementation, reserved bits in the MSR may not be copied to SRR1. |
| MSR      | SF * 1  
PR 0  
SE 0  
IR 0  
ISF * —  
FP 0  
BE 0  
DR 0  
POW 0  
ME —  
FE1 0  
RI 0  
ILE —  
FE0 0  
IP —  
LE Set to value of ILE  
TEMPORARY 64-BIT BRIDGE |

* If the MSR[ISF] bit is implemented, the value of the MSR[ISF] bit is copied to the MSR[SF] bit when an exception is taken.

When a floating-point assist exception is taken, instruction execution resumes as offset 0x00E00 from the base address indicated by MSR[IP].