SECTION 6
EXCEPTIONS

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 predetermined for each exception. Processing of exceptions occurs in supervisor mode.

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

The PowerPC architecture requires that exceptions be handled in program order; therefore, while exception conditions may be recognized out of order, they are handled strictly in order. When an instruction-caused exception is recognized, any unexecuted instructions that appear earlier in the instruction stream are required to complete before the exception is taken. An instruction is said to have “completed” when the results of that instruction’s execution have been committed to the appropriate registers (i.e., following the writeback stage). If a single instruction encounters multiple exception conditions, those exceptions are taken and handled sequentially.

Asynchronous exceptions (exceptions not associated with a specific instruction) are recognized when they occur, but are not handled until all completed instructions have retired and the instruction remaining at the head of the history buffer is ready to retire.

In many cases, after an 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.

Exception handlers should save the information saved in SRR0 and SRR1 soon after the exception is taken to prevent this information from being lost due to another exception being taken. The information should be saved before enabling any exception that is automatically disabled when an exception is taken.

NOTE

If debug mode is enabled and the appropriate bit in the debug enable register (DER) is set, recognition of an exception results in debug-mode processing rather than normal exception processing. Refer to **SECTION 8 DEVELOPMENT SUPPORT** for details.

6.1 Exception Classes

Exception classes are shown in **Table 6-1**. These classes are described in the following paragraphs.

Table 6-1 RCPU Exception Classes

Type	Exception
Asynchronous, unordered (non-maskable)	System reset Non-maskable data or instruction breakpoint Non-maskable external breakpoint
Asynchronous, ordered (maskable)	External interrupt Decrementer Maskable external breakpoint
Synchronous (precise), ordered	Instruction-caused exceptions (except machine check)
Synchronous (precise), unordered	Machine check

6.1.1 Ordered and Unordered Exceptions

An exception is said to be ordered if, when it is taken, it is guaranteed that no program state is lost (provided proper procedures are followed in the exception handlers). In the RCPU implementation of the PowerPC architecture, all exceptions are ordered except for the following:

- Reset
- Machine check
- Non-maskable internal (instruction and data) breakpoints
- Non-maskable external breakpoints

Unordered exceptions may be reported at any time and are not guaranteed to preserve program state information. The processor can never recover from a reset exception. It can recover from other unordered exceptions in most cases. However, if an unordered exception occurs during the servicing of a previous exception, the machine state information in SRR0 and SRR1 (and, in some cases, the DAR and DSISR) may not be recoverable; the processor may be in the process of saving or restoring these registers.

To determine whether the machine state is recoverable, the user can read the RI (recoverable exception) bit in SRR1. Refer to **6.5 Recovery from Exceptions** for details.

6.1.2 Synchronous, Precise Exceptions

Synchronous exceptions are caused by instructions. They are said to be either precise or imprecise. In the RCPU implementation of the PowerPC architecture, all synchronous exceptions are precise.

When a precise exception occurs, the processor backs the machine up to the in-

struction causing the exception. This ensures that the machine is in its correct architecturally-defined state. The following conditions exist at the point a precise exception occurs:

1. Architecturally, no instruction following the faulting instruction in the code stream has begun execution.
2. All instructions preceding the faulting instruction appear to have completed with respect to the executing processor.
3. SRR0 addresses either the instruction causing the exception or the immediately following instruction. Which instruction is addressed can be determined from the exception type and the status bits.
4. Depending on the type of exception, the instruction causing the exception may not have begun execution, may have partially completed, or may have completed execution. Refer to **Table 6-2** for details.

The precise exception model can simplify and speed up exception processing because software does not have to save the machine's internal pipeline states, unwind the pipelines, and cleanly terminate the faulting instruction, nor does it have to reverse the process to resume execution of the faulting instruction stream.

NOTE

In the RCPU implementation of the PowerPC architecture, the machine-check exception is synchronous, (i.e., it is assigned to the instruction that caused it). In other PowerPC implementations, this exception may be asynchronous.

Table 6-2 shows which precise exceptions are taken before the excepting instruction is executed, which are taken after, and which are taken after the instruction is partially executed.

Table 6-2 Handling of Precise Exceptions

Exception Type	Instruction Type	Before/After	Contents of SRR0
Machine check	Any	Before	Faulting instruction
Alignment	Multiple	Partially	Faulting instruction
	Others	Before	
Floating-point enabled	Move to MSR, **rfi**	After	Next instruction to execute
Floating-point enabled	Move to FPSCR	After	Faulting instruction
Privileged instruction, trap, floating-point unavailable	Multiple	Before	Faulting instruction
System call	**sc**	After	Next instruction to execute
Trace	Any	After	Next instruction to execute
Debug I-breakpoint	Any	Before	Faulting instruction
Debug L-breakpoint	Load/store	After	Faulting instruction + 4
Software emulation	NA	Before	Faulting instruction
Floating-point assist	Floating point	Before	Faulting instruction

6.1.3 Asynchronous Exceptions

Asynchronous exceptions are not caused by instructions and are thus not synchronized to internal processor events. When an asynchronous exception occurs, the following conditions exist at the exception point:

- SRR0 addresses the instruction 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.

Asynchronous exceptions can be either ordered or unordered, depending on whether they are maskable.

Maskable exceptions are considered ordered because, if proper software procedures are followed, they are never recognized while the processor is saving or restoring the machine state during a previous exception. Thus, the processor can always recover from one of these exceptions.

Asynchronous, non-maskable exceptions can occur while other exceptions are being processed. If one of these exceptions occurs immediately after another exception, the state information saved by the first exception may be overwritten when the second exception occurs. These exceptions are thus considered unordered. For additional information, refer to **6.5.2 Recovery from Unordered Exceptions**.

6.1.3.1 Asynchronous, Maskable Exceptions

The RCPU supports the following asynchronous, maskable exceptions: external interrupts, decrementer interrupts, and maskable internal and external breakpoint exceptions.

External and decrementer interrupts are masked by the external interrupt enable (EE) bit in the MSR. When MSR[EE] = 0, these exception conditions are latched and are not recognized until MSR[EE] is set. MSR[EE] is cleared automatically when an exception is taken to delay recognition of external and decrementer interrupts.

Maskable internal or external breakpoint exceptions are recognized only when the RI (recoverable exception) bit in the MSR = 1. This ensures that (with proper software safeguards) the processor can always recover from one of these exceptions.

Refer to **SECTION 8 DEVELOPMENT SUPPORT** for details on maskable and non-maskable internal and external breakpoints.

6.1.3.2 Asynchronous, Non-Maskable Exceptions

Asynchronous, non-maskable exceptions include reset and non-maskable internal and external breakpoint exceptions. These exceptions have the highest priority and can occur while other exceptions are being processed. Because these exceptions are non-maskable, they are never delayed; therefore, if an asynchronous, non-maskable exception occurs immediately after another exception, the state information saved by the first exception may be overwritten when the second exception occurs.

For additional information, refer to **6.5.2 Recovery from Unordered Exceptions**. Refer to **SECTION 8 DEVELOPMENT SUPPORT** for details on maskable and non-maskable internal and external breakpoints.

6.2 Exception Vector Table

The setting of the exception prefix (IP) bit in the MSR determines how exceptions are vectored. If the bit is cleared, exceptions are vectored to the physical address 0x0000 0000 plus the vector offset; if IP is set, exceptions are vectored to the physical address 0xFFF0 0000 plus the vector offset. **Table 6-3** shows the exception vector offset of the first instruction of the exception handler routine for each exception type.

NOTE

The exception vectors shown in **Table 6-3**, up to and including the floating-point assist exception (vector offset 0x00E00), are defined by the PowerPC architecture. Exception vectors beginning with offset 0x01000 (software emulation exception in the RCPU) are reserved in the PowerPC architecture for implementation-specific exceptions.

Table 6-3 Exception Vectors and Conditions

Exception Type	Vector Offset	Causing Conditions
Reserved	0x00000	Reserved
System reset	0x00100	A reset exception results when the $\overline{\text{RESET}}$ input to the processor is asserted.
Machine check	0x00200	A machine check exception results when the $\overline{\text{TEA}}$ signal is asserted internally or externally.
—	0x00300	Reserved. (In the PowerPC architecture, this exception vector is reserved for data access exceptions.)
—	0x00400	Reserved. (In the PowerPC architecture, this exception vector is reserved for instruction access exceptions.)
External interrupt	0x00500	An external interrupt occurs when the RCPU $\overline{\text{IRQ}}$ input signal is asserted.
Alignment	0x00600	An alignment exception is caused when the processor cannot perform a memory access for one of the following reasons: • The operand of a floating-point load or store is not word-aligned. • The operand of a load- or store-multiple instruction is not word-aligned. • The operand of **lwarx** or **stwcx.** is not word-aligned. • In little-endian mode, an operand is not properly aligned. • In little-endian mode, the processor attempts to execute a multiple or string instruction.
Program	0x00700	A program exception is caused by one of the following exception conditions: • Floating-point enabled exception — A floating-point enabled program exception condition is generated when the following condition is met as a result of a move to FPSCR instruction, move to MSR instruction, or return from interrupt instruction: (MSR[FE0] | MSR[FE1]) & FPSCR[FEX] = 1. • Privileged instruction — A privileged instruction type program exception is generated when the execution of a privileged instruction is attempted and the MSR register user privilege bit, MSR[PR], is set. This exception is also generated for **mtspr** or **mfspr** with an invalid SPR field if SPR[0]=1 and MSR[PR]=1. • Trap — A trap type program exception is generated when any of the conditions specified in a trap instruction is met.
Floating-point unavailable	0x00800	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 disabled, MSR[FP]=0.
Decrementer	0x00900	The decrementer exception occurs when the most significant bit of the decrementer (DEC) register changes from zero to one.
Reserved	0x00A00	—
Reserved	0x00B00	—
System call	0x00C00	A system call exception occurs when a system call (**sc**) instruction is executed.
Trace	0x00D00	A trace exception occurs if MSR[SE] = 1 and any instruction other than **rfi** is successfully completed, or if MSR[BE] = 1 and a branch is completed.

Table 6-3 Exception Vectors and Conditions (Continued)

Exception Type	Vector Offset	Causing Conditions
Floating-point assist	0x00E00	A floating-point assist exception occurs in the following cases: • When the following condition is true (except in the cases mentioned above for program exceptions): (MSR[FE0] | MSR[FE1]) &FPSCR[FEX] = 1 • When a tiny result is detected and the floating-point underflow exception is disabled (FPSCR[UE] = 0) • In some cases when at least one of the source operands is denormalized.
Software emulation	0x01000	An implementation-dependent software emulation exception occurs when an attempt is made to execute an unimplemented instruction, or to execute a **mtspr** or **mfspr** instruction that specifies an unimplemented register.
Data breakpoint	0x01C00	An implementation-dependent data breakpoint exception occurs when an internal breakpoint match occurs on the load/store bus.
Instruction breakpoint	0x01D00	An implementation-dependent instruction breakpoint exception occurs when an internal breakpoint match occurs on the instruction bus.
Maskable external breakpoint	0x01E00	An implementation-dependent maskable external breakpoint occurs when an external device or on-chip peripheral generates a maskable breakpoint.
Non-maskable external breakpoint	0x01F00	An implementation-dependent non-maskable external breakpoint occurs when an external breakpoint is input to the serial interface of the development port.

6.3 Precise Exception Model Implementation

In order to achieve maximum performance, the RCPU processes many pieces of the instruction stream concurrently. Instructions execute in parallel and may complete out of order. The processor is designed to ensure that this out of order operation never has an effect different from that specified by the program. This requirement is most difficult to ensure when an exception occurs after instructions that logically follow the faulting instruction have already completed. When an exception occurs, the machine state becomes visible to other processes and therefore must be in its correct architecturally specified condition. The processor takes care of this in hardware by automatically backing the machine up to the instruction that caused the interrupt. The processor is therefore said to implement a precise exception model.

To enable the processor to recover from an exception, a history buffer is used. This buffer is a FIFO queue which records relevant machine state at the time of each instruction issue. Instructions are placed on the tail of the queue when they are issued and percolate to the head of the queue while they are in execution. Instructions remain in the queue until they complete execution (i.e., have completed the writeback stage) and all preceding instructions have completed as well. In this way, when an exception occurs, the machine state necessary to recover the architectural state is available. As instructions complete execution, they are retired from the queue, and the buffer storage is reclaimed for new instructions entering the queue.

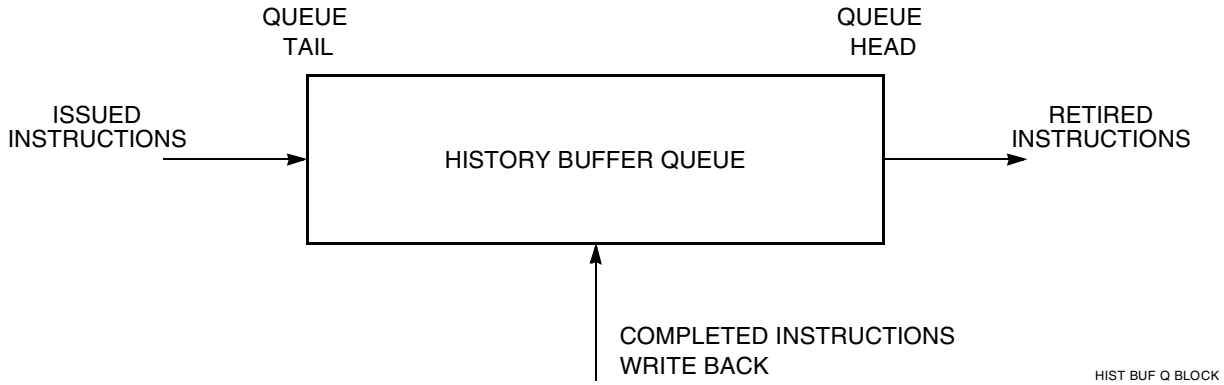

Figure 6-1 History Buffer Queue

An exception can be detected at any time during instruction execution and is recorded in the history buffer when the instruction finishes execution. The exception is not recognized until the faulting instruction reaches the head of the history queue. When the exception is recognized, exception processing begins. The queue is reversed, and the machine is restored to its state at the time the instruction was issued. Machine state is restored at a maximum rate of two floating-point and two integer instructions per clock cycle.

To correctly restore the architectural state, the history buffer must record the value of the destination before the instruction is executed. The destination of a store instruction, however, is in memory. It is not practical for the processor to always read memory before writing it. Therefore, stores issue immediately to store buffers, but do not update memory until all previous instructions have completed execution without exception, i.e., until the store has reached the head of the history buffer.

The history buffer has enough storage to hold a total of six instructions. Of these, a maximum of four can be integer instructions (including integer load or store instructions), and a maximum of three can be floating-point instructions (including floating-point loads or stores). If the buffer includes an instruction with long latency, it is possible (if a data dependency does not occur first) for issued instructions to fill up the history buffer. If so, instruction issue halts until the long-latency operation retires (along with any instructions following it that are ready to retire). Instructions that can cause the history buffer to fill up include floating-point arithmetic instructions, integer divide instructions, and instructions that affect or use resources external to the processor (e.g., load/store instructions).

6.4 Implementation of Asynchronous Exceptions

When an enabled asynchronous exception is detected, the processor attempts to retire as many instructions as possible. That is, all instructions that have completed the writeback stage without generating exceptions are allowed to retire, provided all instructions ahead of them in the history buffer have also completed the writeback stage without generating exceptions.

The asynchronous exception is then assigned to the instruction at the head of the history buffer, which has not yet completed (otherwise, it would have been retired). If this instruction is one of the following, it is allowed to complete execution and retire:

- **mtspr**, **mtmsr**, or **rfi** instruction
- Memory reference that is already on the bus (other than a load or store multiple or string instruction)
- Cache control instruction.

In this case, the exception is assigned to the next instruction in the history buffer. Notice that if the instruction at the head of the history buffer generates an exception before it retires, that exception is treated before the asynchronous exception.

If the instruction is not one of those listed above, it and all subsequent instructions are flushed from the buffer as if they were never executed at all.

6.5 Recovery from Exceptions

The processor should always be able to recover from an ordered exception. Provided no machine state information is lost, the processor can recover from unordered exceptions, except reset, as well.

6.5.1 Recovery from Ordered Exceptions

The RCPU can always recover from an ordered exception, provided the exception-handling software follows proper procedures. Exception handlers must ensure that no exception-generating instruction is executed during the prologue (before appropriate registers are saved) or epilogue (between restore of these registers and the execution of the **rfi** instruction). Registers that need to be saved are the machine status save/restore registers (SRR0 and SRR1) and, for certain exceptions, the DAR (data address register) and DSISR (data storage interrupt status register).

Hardware automatically clears MSR[EE] during exception processing in order to disable external and decrementer interrupts. If desired, the user can set this bit at the end of the exception handler prologue, after saving the machine state. In this case, the user must clear the bit (along with the RI bit) before the start of the exception handler epilog. Refer to **6.5.3 Commands to Alter MSR[EE] and MSR[RI]** for instructions on altering these bits.

6.5.2 Recovery from Unordered Exceptions

Unless it is in the process of saving or restoring machine state, the processor can recover from the following unordered exceptions:

- Machine check
- Non-maskable external breakpoint
- Non-maskable internal instruction or data breakpoint

The RI bit (recoverable exception) in the MSR and its shadow in SRR1 enable an exception handler to determine whether the processor can recover from an exception. During exception processing, the RI bit in the MSR is copied to SRR1; the bit

in the MSR is then cleared. Each exception handler should set the RI bit in the MSR (using the **mtmsr** instruction) at the end of its prologue, after saving the program state (SRR0, SRR1, and, in some cases, DSISR and DAR). At the start of its epilogue (before saving the machine state), each exception handler should clear the RI bit in the MSR.

In this way, the exception handler for an unordered exception can read the RI bit in SRR1 to determine whether the processor can recover from the exception. If the exception occurs while the machine state is being saved or restored during the processing of a previous exception, the RI bit in SRR1 will be cleared, indicating that the processor cannot recover from the exception. If the exception occurs at any other time, the RI bit in SRR1 will be set, indicating the processor can recover from the exception.

In critical code sections where MSR[EE] is negated but SRR0 and SRR1 are not busy, MSR[RI] should be left asserted. In these cases if an exception occurs,the processor can be restarted.

6.5.3 Commands to Alter MSR[EE] and MSR[RI]

The processor includes special commands to facilitate the software manipulation of the MSR[RI] and MSR[EE] bits. These commands are executed by issuing the **mtspr** instruction with one of the pseudo-SPRs shown in **Table 6-4**. Writing any data to one of these locations performs the operation specified in the table. A read (**mfspr**) of any of these locations is treated as an unimplemented instruction, resulting in a software emulation exception.

Table 6-4 Manipulating EE and RI Bits

SPR # (Decimal)	Mnemonic	MSR[EE]	MSR[RI]	Use
80	EIE	1	1	External Interrupt Enable: • End of exception handler's prologue, to enable nested external interrupts; • End of critical code segment in which external interrupts were disabled
81	EID	0	1	External Interrupt Disable, but other interrupts are recoverable: • End of exception handler's prologue, to keep external nested interrupts disabled; • Start of critical code segment in which external interrupts are disabled
82	NRI	0	0	Non-Recoverable Interrupt: • Start of exception handler's epilogue

6.6 Exception Order and Priority

When multiple conditions that can cause an exception are present, the highest-priority exception is taken. Exceptions are roughly prioritized by exception class, as follows:

1. Asynchronous, non-maskable exceptions have priority over all other exceptions. These exceptions cannot be delayed and do not wait for the completion of any precise exception handling.
2. Synchronous exceptions are caused by instructions and are handled in strict program order.
3. Asynchronous, maskable exceptions (external interrupt, decrementer exceptions, and maskable breakpoint exceptions) are delayed until exceptions caused by the instruction at the head of the history buffer (after instructions that have already completed have retired) are taken.

The exceptions are listed in **Table 6-5** in order of highest to lowest priority.

Table 6-5 Exception Priorities

Class	Priority	Exception
Asynchronous, non-maskable	1	Non-maskable external breakpoint — This exception has the highest priority and is taken immediately, regardless of other pending exceptions or whether the machine state is recoverable.
	2	Reset —The reset exception has the second-highest priority and is taken immediately, regardless of other pending exceptions (except for the non-maskable external breakpoint exception) or whether the machine state is recoverable.
Synchronous	3	Instruction dependent — When an instruction causes an exception, the exception mechanism waits for any instructions prior to the exception instruction in the instruction stream to execute. 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.

Table 6-5 Exception Priorities (Continued)

Class	Priority	Exception
Asynchronous, maskable	4	Peripheral or external maskable breakpoint request — When this exception type occurs, the processor retires as many instructions as possible (i.e., all instructions that have completed the writeback stage without generating an instruction, provided all instructions ahead of it in the history buffer have also completed the writeback stage without generating an exception). Then, depending on the instruction currently at the head of the history buffer, the processor either flushes the history buffer or allows the instruction at the head of the buffer to retire before generating an exception. Refer to **6.4 Implementation of Asynchronous Exceptions**.
	5	External interrupt — When this exception type occurs, the processor retires as many instructions as possible (i.e., all instructions that have completed the writeback stage without generating an instruction, provided all instructions ahead of it in the history buffer have also completed the writeback stage without generating an exception). Then, depending on the instruction currently at the head of the history buffer, the processor either flushes the history buffer or allows the instruction at the head of the buffer to retire before generating an exception (provided a higher priority exception does not exist). Refer to **6.4 Implementation of Asynchronous Exceptions**. This exception is delayed if MSR[EE] is cleared.
	6	Decrementer — This exception is the lowest priority exception. When this exception type occurs, the processor retires as many instructions as possible (i.e., all instructions that have completed the writeback stage without generating an instruction, provided all instructions ahead of it in the history buffer have also completed the writeback stage without generating an exception). Then, depending on the instruction currently at the head of the history buffer, the processor either flushes the history buffer or allows the instruction at the head of the buffer to retire before generating an exception (provided a higher priority exception does not exist). Refer to **6.4 Implementation of Asynchronous Exceptions**. This exception is delayed if MSR[EE] is cleared.

6.7 Ordering of Synchronous, Precise Exceptions

Synchronous exceptions are handled in strict program order, even though instructions can execute and exceptions can be detected out of order. Therefore, before the RCPU processes an instruction-caused exception, it executes all instructions and handles any resulting exceptions that appear earlier in the instruction stream.

Only one synchronous, precise exception can be reported at a time. If single instructions generate multiple exception conditions, the processor handles the exception it encounters first; then the execution of the excepting instruction continues until the next excepting condition is encountered. **Table 6-6** lists the order in which synchronous exceptions are detected.

Table 6-6 Detection Order of Synchronous Exceptions

Order of Detection	Exception Type
1	Trace[1]
2	Machine check during instruction fetch
3	I-bus breakpoint
4	Software emulation exception
5	Floating-point unavailable
6[2]	Privileged instruction
	Alignment exception
	Floating-point enabled exception
	System call
	Trap
7	Floating-point assist exception detected by floating-point unit, or by load/store unit during a store
8	Machine check during load or store
9	Floating-point assist exception detected by load/store unit during a load
10	L-bus breakpoint

NOTES:
1. The trace mechanism is implemented by letting one instruction complete as if no trace were enabled and then trapping the second instruction. Trace has the highest priority of exceptions associated with this second instruction.
2. All of these cases are mutually exclusive for any one instruction.

6.8 Exception Processing

When an exception is taken, the processor uses the save/restore registers, SRR0 and SRR1, to save the contents of the machine state register and to identify where instruction execution should resume after the exception is handled.

When an exception occurs, SRR0 is set to point to the instruction at which instruction processing should resume when the exception handler returns control to the interrupted process. All instructions in the program flow preceding this one will have completed, and no subsequent instruction will have completed. The address may be of the instruction that caused the exception or of the next one (as in the case of a system call exception, for example). The instruction addressed can be determined from the exception type and status bits.

SRR1 is a 32-bit register used to save machine status (the contents of the MSR) on exceptions and to restore machine status when **rfi** is executed.

The data address register (DAR) is a 32-bit register used by alignment exceptions to identify the address of a memory element.

When an exception occurs, SRR1[0:15] are loaded with exception-specific information and bits SRR1[16:31] are loaded with the corresponding bits of the MSR. The machine state register is shown below.

MSR — Machine State Register

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
RESERVED															ILE
RESET:															
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
EE	PR	FP	ME	FE0	SE	BE	FE1	0	IP	IR	DR	0		RI	LE
RESET:															
0	0	0	U	0	0	0	0	0	*	0	0	0	0	0	0

*Reset value of this bit on value of internal data bus configuration word at reset. Refer to the*System Interface Unit Reference Manual* (SIURM/AD).

Table 6-7 shows the bit definitions for the MSR.

Table 6-7 Machine State Register Bit Settings

Bit(s)	Name	Description
[0:14]	—	Reserved
15	ILE	Exception little-endian mode. When an exception occurs, this bit is copied into MSR[LE] to select the endian mode for the context established by the exception. 0 Processor runs in big-endian mode during exception processing. 1 Processor runs in little-endian mode during exception processing.
16	EE	External interrupt enable 0 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.
17	PR	Privilege level 0 The processor can execute both user- and supervisor-level instructions. 1 The processor can only execute user-level instructions.
18	FP	Floating-point available 0 The processor prevents dispatch of floating-point instructions, including floating-point loads, stores and moves. Floating-point enabled program exceptions can still occur and the FPRs can still be accessed. 1 The processor can execute floating-point instructions, and can take floating-point enabled exception type program exceptions.
19	ME	Machine check enable 0 Machine check exceptions are disabled. 1 Machine check exceptions are enabled.
20	FE0	Floating-point exception mode 0 (See **Table 6-8**.)
21	SE	Single-step trace enable 0 The processor executes instructions normally. 1 The processor generates a single-step trace exception upon the successful execution of the next instruction. When this bit is set, the processor dispatches instructions in strict program order. Successful execution means the instruction caused no other exception. Single-step tracing may not be present on all implementations.
22	BE	Branch trace enable
23	FE1	Floating-point exception mode 1 (See **Table 6-8**.)
24	—	Reserved
25	IP	Exception prefix. The setting of this bit determines the location of the exception vector table. 0 Exceptions are vectored to the physical address 0x0000 0000 plus vector offset. 1 Exceptions are vectored to the physical address 0xFFF0 0000 plus vector offset.
[26:29]	—	Reserved
30	RI	Recoverable exception 0 Exception is not recoverable. 1 Exception is recoverable.
31	LE	Little-endian mode 0 Processor operates in big-endian mode during normal processing. 1 Processor operates in little-endian mode during normal processing.

Table 6-8 Floating-Point Exception Mode Bits

FE[0:1]	Mode
00	Floating-point exceptions disabled
01, 10, 11	Floating-point precise mode

MSR[16:31] are guaranteed to be written to SRR1 when the first instruction of the exception handler is encountered.

6.8.1 Enabling and Disabling Exceptions

When a condition exists that causes an exception to be generated, the processor must determine whether the exception is enabled for that condition.

- Floating-point enabled exceptions (a type of program exception) can be disabled by clearing both MSR[FE0] and MSR[FE1]. If either or both of these bits are set, all floating-point exceptions are taken and cause a program exception. Bits in the FPSCR can enable and disable individual conditions that can generate floating-point exceptions.
- External and decrementer 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.
- System reset and non-maskable external breakpoint exceptions cannot be masked.
- Internal data and instruction breakpoints are specified as maskable or non-maskable by the BRKNOMSK bit in LCTRL2.
- Maskable internal (data and instruction) and external breakpoints are recognized only when MSR[RI] = 1.

6.8.2 Steps for Exception Processing

After determining 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. Loads the machine status save/restore register 0 (SRR0) 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. Loads SRR1[0:15] with information specific to the exception type.
3. Loads SRR1[16:31] with a copy of MSR[16:31].
4. Sets the MSR as described in **Table 6-9**. The new values take effect beginning with the fetching of the first instruction of the exception-handler routine located at the exception vector address.

5. Resumes fetching and executing instructions, using the new MSR value, at a location specific to the exception type. The location is determined by adding the exception's vector (see **Table 6-3**) to the base address determined by MSR[IP]. If IP is cleared, exceptions are vectored beginning at the physical address 0x0000 0000. If IP is set, exceptions are vectored beginning at 0xFFF0 0000. 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).
6. The **lwarx** and **stwx** instructions require special handling if a reservation is still set when an exception occurs.

Table 6-9 shows the MSR bit settings when the processor changes to supervisor mode.

Table 6-9 MSR Setting Due to Excepti on

	MSR Bit										
Exception Type	**EE** 16	**PR** 17	**FP** 18	**ME** 19	**FE0** 20	**SE** 21	**BE** 22	**FE1** 23	**IP** 25	**RI** 30	**LE** 31
Reset	0	0	0	—[1]	0	0	0	0	†[2]	0	0
All others	0	0	0	—	0	0	0	0	—	0	†[3]

NOTES:
1. — indicates that bit is not altered.
2. Depends on value of internal data bus configuration word at reset.
3. Contains value of MSR[ILE] prior to exception.

6.8.3 DAR, DSISR, and BAR Operation

The load/store unit keeps track of all instructions and bus cycles. In case of a bus error, the data address register (DAR) is loaded with the effective address (EA) of the cycle. In case of a multi-cycle instruction, the effective address of the first offending cycle is loaded.

The data storage and interrupt status register (DSISR) identifies the cause of the error in case of an exception caused by a load or a store. The DSISR is loaded with the instruction information as described in **6.11.4 Alignment Exception (0x00600)**.

The breakpoint address register (BAR) indicates the address on which an L-bus breakpoint occurs. For multi-cycle instructions, the BAR contains the address of the first cycle associated with the breakpoint. The BAR has a valid value only when a data breakpoint exception is taken; at all other times its value is boundedly undefined.

Table 6-10 summarizes the values in DAR, BAR, and DSISR following an exception.

Table 6-10 DAR, BAR, and DSISR Values in Exception Processing

Exception Type	DAR Value	DSISR Value	BAR Value
Alignment exception	Data EA	Instruction information	Undefined
L-bus breakpoint exception	Unchanged	Unchanged	Cycle EA
Floating-Point Assist Exception	Unchanged	Unchanged	Undefined
Machine-check exception	Cycle EA	Instruction information	Undefined
Implementation-dependent software emulation exception	Unchanged	Unchanged	Undefined
Floating-point unavailable exception	Unchanged	Unchanged	Undefined
Program exception	Unchanged	Unchanged	Unchanged

6.8.4 Returning from Supervisor Mode

The return from interrupt (**rfi**) instruction performs context synchronization by allowing previously issued instructions to complete before returning to user mode. Execution of the **rfi** instruction ensures the following:

- All previous instructions have been retired.
- Previous instructions complete execution in the context (privilege and protection) under which they were issued.
- The instructions following this instruction execute in the context established by this instruction.

6.9 Process Switching

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

- The **sync** instruction, to resolve any data dependencies between the processes and to synchronize the use of SPRs.
- The **isync** instruction, to ensure that undispatched instructions not in the new process are not used by the new process.
- The **stwcx.** instruction, to clear any outstanding reservations, which ensures that an **lwarx** instruction in the old process is not paired with an **stwcx.** in the new process.

Note that if an exception handler is used to emulate an instruction that is not implemented, the exception handler must report in SRR0 the EA computed by the instruction being emulated and not one used to emulate the instruction.

6.10 Exception Timing

Table 6-11 illustrates the significant events in exception processing.

Table 6-11 Exception Latency

Time	Fetch	Issue	Instruction Complete	Kill Pipeline
A		Faulting instruction issue		
B			Instruction complete and all previous instructions complete	
				Kill pipeline
C	Start fetch handler			
D ≤ B + 3 clocks				
E		1st instruction of handler issued		

At time-point A the excepting instruction issues and begins execution. During the interval A-B previously issued instructions are finishing execution. The interval A-B is equivalent to the time required for all instructions currently in progress to complete, (i.e., the time to serialize the machine).

At time-point B the excepting instruction has reached the head of the history queue, implying that all instructions preceding it in the code stream have finished execution without generating any exception. In addition, the excepting instruction itself has completed execution. At this time the exception is recognized, and exception processing begins. If at this point the instruction had not generated an exception, it would have been retired.

During the interval B-D the machine state is being restored. This can take up to three clock cycles.

At time-point C the processor starts fetching the first instruction of the exception handler.

By time-point D the state of the machine prior to the issue of the excepting instruction has been restored. During interval D-E, the machine is saving context information in SRR0 and SRR1, disabling interrupts, placing the machine in privileged mode, and may continue the process of fetching the first instructions of the interrupt handler from the vector table.

At time-point E the MSR and instruction pointer of the executing process have been saved and control has been transferred to the exception handler routine.

The interval D-E requires a minimum of one clock cycle. The interval C-E depends on the memory system. This interval is the time it takes to fetch the first instruction of the exception handler. For a full history buffer, it is no less then two clocks.

6.11 Exception Definitions

The following paragraphs describe each type of exception supported by the RCPU.

6.11.1 Reset Exception (0x0100)

The reset exception is a non-maskable, asynchronous exception signaled to the processor by the assertion of the internal reset input signal (RESET). The system interface unit asserts this signal in response to either the assertion of the external $\overline{\text{RESET}}$ pin or an internal reset request, such as from the software watchdog timer. Refer to the *System Interface Unit Reference Manual* (SIURM/AD) for a description of sources within the SIU that can cause the RESET input to the processor to be asserted.

A reset operation should be performed on power-on to appropriately reset the processor. The assertion of RESET causes the reset exception to be taken. The physical address of the handler is 0xFFF0 0100 or 0x0000 0100, depending on the value of the internal data bus configuration word during reset. Refer to the *System Interface Unit Reference Manual* (SIURM/AD) for additional information on the data bus configuration word and system configuration during reset.

Table 6-12 shows the state of the machine just before it fetches the first instruction after reset. Registers not listed are not affected by reset.

Table 6-12 Settings Caused by Reset

Register	Setting
MSR	IP depends on internal data bus configuration word ME is unchanged All other bits are cleared
SRR0	Undefined
SRR1	Undefined
FPECR	0x0000 0000
ICTRL	0x0000 0000
LCTRL1	0x0000 0000
LCTRL2	0x0000 0000
COUNTA[16:31]	0x0000 0000
COUNTB[16:31]	0x0000 0000
DMCR	0x0000 0000
DMMR[2,4,28:31]	Set to one
ICCST	0x0000 0000
ICADR, ICDAT	Undefined

6.11.2 Machine Check Exception (0x00200)

The processor conditionally initiates a machine-check exception after detecting the assertion of the TEA signal, indicating the occurrence of a bus error. The $\overline{\text{TEA}}$ signal can be asserted either externally (by an external device asserting the $\overline{\text{TEA}}$ pin), or internally by the SIU chip-select logic. The processor receives notification of the exception from either the I-bus (if the exception is caused during the instruction phase) or the L-bus (if the exception is caused during the data phase).

Machine check exceptions are unordered. The machine-state exception handler must read the SRR1[RI] bit to determine whether the processor can recover from a machine-check exception. For additional information, refer to **6.5.2 Recovery from Unordered Exceptions**.

A machine-check exception is assumed to be caused by one of the following conditions:

- The accessed address does not exist.
- A data error was detected.
- A storage protection violation was detected by chip-select logic (either on-chip or external).

When a machine-check exception occurs, the processor does one of the following:

- Takes a machine check exception;
- Enters the checkstop state; or
- Enters debug mode.

Which action is taken depends on the value of the MSR[ME] bit, whether or not debug mode was enabled at reset, and (if debug mode is enabled) the values of the CHSTPE (checkstop enable) and MCIE (machine check enable) bits in the debug enable register (DER). **Table 6-13** summarizes the possibilities.

Table 6-13 Machine Check Exception Processor Actions

MSR[ME]	Debug Mode Enable	CHSTPE	MCIE	Action Performed when Exception Detected
0	0	X	X	Enter checkstop state
1	0	X	X	Branch to machine-check exception handler
0	1	0	X	Enter checkstop state
0	1	1	X	Enter debug mode
1	1	X	0	Branch to machine-check exception handler
1	1	X	1	Enter debug mode

6.11.2.1 Machine Check Exception Enabled

A machine check exception is taken when MSR[ME] is set and either debug mode is disabled or DER[MCIE] is cleared. When a machine check exception is taken, registers are updated as shown in **Table 6-14**.

Table 6-14 Register Settings Following a Machine Check Exception

Register	Setting Description	
SRR0	Set to the effective address of the instruction that caused the interrupt.	
SRR1	0 1 [2:15] [16:31]	Cleared Set for instruction-fetch related errors, cleared for load-store related errors Cleared Loaded from MSR[16:31].
MSR	IP ME LE Other bits	No change Cleared to zero Set to value of ILE bit prior to the exception Cleared
DSISR (L-bus case only)	[15:16] 17 [22:31]	Set to bits [29:30] of the instruction if X-form Set to 0b00 if D-form Set to bit 25 of the instruction if X-form Set to bit 5 of the instruction if D-form Set to bits [6:15] of the instruction
DAR (L-bus case only)	Set to the effective address of the data access that caused the exception.	

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

6.11.2.2 Checkstop State

The processor enters the checkstop state when a machine check exception occurs, MSR[ME] equals zero, and either debug mode is disabled or DER[CHSTPE] is cleared. When the processor is in the checkstop state, instruction processing is suspended and generally cannot be restarted without resetting the processor. The contents of all latches (except any associated with the bus clock) are frozen within two cycles upon entering checkstop state so that the state of the processor can be analyzed.

6.11.2.3 Machine-Check Exceptions and Debug Mode

The processor enters debug mode when a machine check exception occurs, debug mode is enabled, and either MSR[ME] = 0 and DER[CHSTPE] = 1, or MSR[ME] = 1 and DER[MCIE] = 1. Refer to **SECTION 8 DEVELOPMENT SUPPORT** for more information.

6.11.3 External Interrupt (0x00500)

The interrupt controller in the on-chip peripheral control unit signals an external interrupt by asserting the $\overline{\text{IRQ}}$ input to the processor. The interrupt may be caused by the assertion of an external $\overline{\text{IRQ}}$ pin, by the periodic interrupt timer, or by an on-chip peripheral. Refer to *System Interface Unit Reference Manual* (SIURM/AD) for more information on the interrupt controller.

The interrupt may be delayed by other higher priority exceptions or if the MSR[EE]

bit is cleared when the exception occurs. MSR[EE] is automatically cleared by hardware to disable external interrupts when any exception is taken.

Upon detecting an external interrupt, the processor assigns it to the instruction at the head of the history buffer (after retiring all instructions that are ready to retire). Refer to **6.4 Implementation of Asynchronous Exceptions** for more information.

The register settings for the external interrupt exception are shown in **Table 6-15**.

Table 6-15 Register Settings Following External Interrupt

Register	Setting Description
SRR0	Set to the effective address of the instruction that the processor would have attempted to execute next if no interrupt conditions were present.
SRR1	[0:15] Cleared [16:31] Loaded from bits [16:31] of the MSR
MSR	IP No change ME No change LE Set to value of ILE bit prior to the exception Other bits Cleared

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

6.11.4 Alignment Exception (0x00600)

The following conditions cause an alignment exception:

- The operand of a floating-point load or store instruction is not word-aligned.
- The operand of a load or store multiple instruction is not word-aligned.
- The operand of **lwarx** or **stwcx.** is not word-aligned.
- The operand of a load or store instruction is not naturally aligned, and MSR[LE] = 1 (little-endian mode).
- The processor attempts to execute a multiple or string instruction when MSR[LE] = 1 (little-endian mode).

Alignment exceptions use the SRR0 and SRR1 to save the machine state and the DSISR to determine the source of the exception.

The register settings for alignment exceptions are shown in **Table 6-16**.

Table 6-16 Register Settings for Alignment Exception

Register	Setting Description
SRR0	Set to the effective address of the instruction that caused the exception.
SRR1	[0:15] Cleared [16:31] Loaded from MSR[16:31]
MSR	IP No change ME No change LE Set to value of ILE bit prior to the exception Other bits Cleared
DSISR	[0:11] Cleared [12:13] Cleared 14 Cleared [15:16] For instructions that use register indirect with index addressing, set to bits [29:30] of the instruction. For instructions that use register indirect with immediate index addressing, cleared. 17 For instructions that use register indirect with index addressing, set to bit 25 of the instruction. For instructions that use register indirect with immediate index addressing, set to bit 5 of the instruction. [18:21] For instructions that use register indirect with index addressing, set to bits [21:24] of the instruction. For instructions that use register indirect with immediate index addressing, set to bits [1:4] of the instruction. [22:26] Set to bits [6:10] (source or destination) of the instruction. [27:31] Set to bits [11:15] of the instruction (**r**A). Set to either bits [11:15] of the instruction or to any register number not in the range of registers loaded by a valid form instruction, for **lmw**, **lswi**, and **lswx** instructions. Otherwise undefined. 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. (If there is no corresponding instruction, no alternative value can be specified.)

When an alignment exception is taken, instruction execution resumes at offset 0x00600 from the physical base address indicated by MSR[IP].

6.11.4.1 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, it needs 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

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 6-17 shows how the DSISR bits identify the instruction that caused the exception.

Table 6-17 DSISR[15:21] Settings

DSISR[15:21]	Instruction
00 0 0000	**lwarx**, **lwz**, reserved[1]
00 0 0010	**stw**
00 0 0100	**lhz**
00 0 0101	**lha**
00 0 0110	**sth**
00 0 0111	**lmw**
00 0 1000	**lfs**
00 0 1001	**lfd**
00 0 1010	**stfs**
00 0 1011	**stfd**
00 1 0000	**lwzu**
00 1 0010	**stwu**
00 1 0100	**lhzu**
00 1 0101	**lhau**
00 1 0110	**sthu**
00 1 0111	**stmw**
00 1 1000	**lfsu**
00 1 1001	**lfdu**
00 1 1010	**stfsu**
00 1 1011	**stfdu**
01 0 1000	**lswx**
01 0 1001	**lswi**
01 0 1010	**stswx**

Table 6-17 DSISR[15:21] Settings (Continued)

DSISR[15:21]	Instruction
01 0 1011	**stswi**
01 1 0101	**lwaux**
10 0 0010	**stwcx.**
10 0 1000	**lwbrx**
10 0 1010	**stwbrx**
10 0 1100	**lhbrx**
10 0 1110	**sthbrx**
11 0 0000	**lwzx**
11 0 0010	**stwx**
11 0 0100	**lhzx**
11 0 0101	**lhax**
11 0 0110	**sthx**
11 0 1000	**lfsx**
11 0 1001	**lfdx**
11 0 1010	**stfsx**
11 0 1011	**stfdx**
11 1 0000	**lwzux**
11 1 0010	**stwux**
11 1 0100	**lhzux**
11 1 0101	**lhaux**
11 1 0110	**sthux**
11 1 1000	**lfsux**
11 1 1001	**lfdux**
11 1 1010	**stfsux**
11 1 1011	**stfdux**

NOTES:

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

6.11.5 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 floating-point enabled exception — A system floating-point enabled exception is generated when the following condition is met as a result of a move to FPSCR instruction, move to MSR (**mtmsr**) instruction, or return from interrupt (**rfi**) instruction:

  (MSR[FE0] | MSR[FE1]) & FPSCR[FEX] = 1.

  Notice that in the RCPU implementation of the PowerPC architecture, a program interrupt is not generated by a floating-point arithmetic instruction that results in the condition shown above; a floating-point assist exception is generated instead.
- Privileged instruction — A privileged instruction type program exception is generated by any of the following conditions:
  — The execution of a privileged instruction (**mfmsr**, **mtmsr**, or **rfi**) is attempted and the processor is operating at the user privilege level (MSR[PR] = 1).
  — The execution of **mtspr** or **mfspr** where SPR0 = 1 in the instruction encoding (indicating a supervisor-access register) and MSR[PR] = 1 (indicating the processor is operating at the user privilege level), provided the SPR instruction field encoding represents either:
    - a valid internal-to-the-processor special-purpose register; or
    - an external-to-the-processor special-purpose register (either valid or invalid).
  — Refer to **7.5 Implementation of Special-Purpose Registers** for a discussion of internal- and external-to-the-processor SPRs.
- Trap — A trap type program exception is generated when any of the conditions specified in a trap instruction is met. Trap instructions are described in **SECTION 4 ADDRESSING MODES AND INSTRUCTION SET SUMMARY**.

Notice that, in contrast to some other PowerPC processors, the RCPU generates a software emulation exception, rather than a program exception, when an attempt is made to execute any unimplemented instruction. This includes all illegal instructions and optional instructions not implemented in the RCPU.

The register settings are shown in **Table 6-18**.

Table 6-18 Register Settings Following Program Exception

Register	Setting Description
SRR0	Contains the effective address of the excepting instruction
SRR1	[0:10] Cleared 11 Set for a floating-point enabled program exception; otherwise cleared. 12 Cleared. 13 Set for a privileged instruction program exception; otherwise cleared. 14 Set for a trap program exception; otherwise cleared. 15 Cleared if SRR0 contains the address of the instruction causing the exception, and set if SRR0 contains the address of a subsequent instruction. [16:31] Loaded from MSR[16:31]. Note that only one of bits 11, 13, and 14 can be set.
MSR	IP No change ME No change LE Set to value of ILE bit prior to the exception Other bits Cleared to zero

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

6.11.6 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, and move instructions), and the floating-point available bit in the MSR is disabled, (MSR[FP] = 0).

The register settings for floating-point unavailable exceptions are shown in **Table 6-19**.

Table 6-19 Register Settings Following a Floating-Point Unavailable Exception

Register	Setting Description
SRR0	Set to the effective address of the instruction that caused the exception.
SRR1	[0:15] Cleared [16:31] Loaded from MSR[16:31]
MSR	IP No change ME No change LE Set to value of ILE bit prior to the exception Other bits Cleared

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

6.11.7 Decrementer Exception (0x00900)

A decrementer exception occurs when no higher priority exception exists, the decrementer register has completed decrementing, and MSR[EE] = 1. The decrementer exception request is canceled when the exception is handled. The decrementer register counts down, causing an exception (unless masked) when passing through zero. The decrementer implementation meets the following requirements:

- 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 zero to one, an 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 zero to one, an interrupt request is signaled.

The register settings for the decrementer exception are shown in **Table 6-20**.

Table 6-20 Register Settings Following a Decrementer Exception

Register	Setting Description
SRR0	Set to the effective address of the instruction that the processor would have attempted to execute next if no exception conditions were present.
SRR1	[0:15] Cleared [16:31] Loaded from MSR[16:31]
MSR	IP No change ME No change LE Set to value of ILE bit prior to the exception Other bits Cleared to zero

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

6.11.8 System Call Exception (0x00C00)

A system call exception occurs when a system call instruction is executed. The effective address of the instruction following the **sc** instruction is placed into SRR0. MSR[16:31] are placed into SRR1[16:31], and SRR1[0:15] are set to undefined values. Then a system call exception is generated.

The system call instruction is context synchronizing. That is, when a system call exception occurs, instruction dispatch is halted and the following synchronization is performed:

1. The exception mechanism waits for all instructions in execution to complete to a point where they report all exceptions they will cause.
2. The processor ensures that all instructions in execution complete in the con-

text in which they began execution.
3. Instructions dispatched after the exception is processed are fetched and executed in the context established by the exception mechanism.

Register settings are shown in **Table 6-22**.

Table 6-21 Register Settings Following a System Call Exception

Register	Setting Description	
SRR0	Set to the effective address of the instruction following the System Call instruction	
SRR1	[0:15] [16:31]	Undefined Loaded from MSR[16:31]
MSR	IP ME LE Other bits	No change No change Set to value of ILE bit prior to the exception Cleared to zero

When a system call exception is taken, instruction execution resumes at offset 0x00C00 from the physical base address indicated by MSR[IP].

6.11.9 Trace Exception (0x00D00)

A trace exception occurs if MSR[SE] = 1 and any instruction other than **rfi** is successfully completed, or if MSR[BE] = 1 and a branch is completed. Notice that the trace exception does not occur after an instruction that causes an exception.

A monitor or debugger software needs to change the vectors of other possible exception addresses in order to single-step such instructions. If this is not desirable, other debugging features can be used. Refer to **SECTION 8 DEVELOPMENT SUPPORT** for more information.

Register settings are shown in **Table 6-22**.

Table 6-22 Register Settings Following a Trace Exception

Register	Setting Description	
SRR0	Set to the effective address of the instruction following the executed instruction	
SRR1	[0:15] [16:31]	Cleared to zero Loaded from MSR[16:31]
MSR	IP ME LE Other bits	No change No change Set to value of ILE bit prior to the exception Cleared to zero

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

6.11.10 Floating-Point Assist Exception (0x00E00)

A floating point assist exception occurs in the following cases:

- When the following conditions are true:
  — A floating-point enabled exception condition is detected;
  — the corresponding floating-point enable bit in the FPSCR (floating point status and control register) is set (exception enabled); and
  — MSR[FE0] | MSR[FE1] = 1
  These conditions are summarized in the following equation:

  (MSR[FE0] | MSR[FE1]) &FPSCR[FEX] = 1

  Note that when ((MSR[FE0] | MSR[FE1]) &FPSCR[FEX]) is set as a result of move to FPSCR, move to MSR or **rfi**, a program exception is generated, rather than a floating-point assist exception.
- When a tiny result is detected and the floating point underflow exception is disabled (FPSCR[UE] = 0)
- In some cases when at least one of the source operands is denormalized (refer to **6.11.10.2 Floating-Point Assist for Denormalized Operands**)

The register settings for floating-point assist exceptions are shown in **Table 6-22**. In addition, floating-point enabled exceptions affect the FPSCR as shown in **Table 6-26**.

Table 6-23 Register Settings Following a Floating-Point Assist Exception

Register	Setting Description	
SRR0	Set to the effective address of the instruction that caused the exception	
SRR1	[0:15] [16:31]	Cleared to zero Loaded from bits [16:31] of the MSR
MSR	IP ME LE Other bits	No change No change Set to value of ILE bit prior to the exception Cleared to zero

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

6.11.10.1 Floating-Point Software Envelope

The floating-point assist software envelope is an exception handler for floating-point assist exceptions. Use of this exception handler guarantees that results of floating-point operations are in compliance with IEEE standards.

In most cases, floating-point operations are implemented in hardware in the RCPU. For cases in which the hardware needs software assistance, the software envelope emulates the instruction using a special synchronized ignore exceptions (SIE) hardware mode. This mode is useful only for emulating an instruction executed in the floating-point unit, not an instruction executed by the load/store unit. SIE mode is described in **6.11.10.3 Synchronized Ignore Exceptions (SIE) Mode**.

Execution of floating-point instructions is illustrated in **Figure 6-2**. This process shows the execution of all floating-point instructions except the floating-point move to FPSCR type instructions.

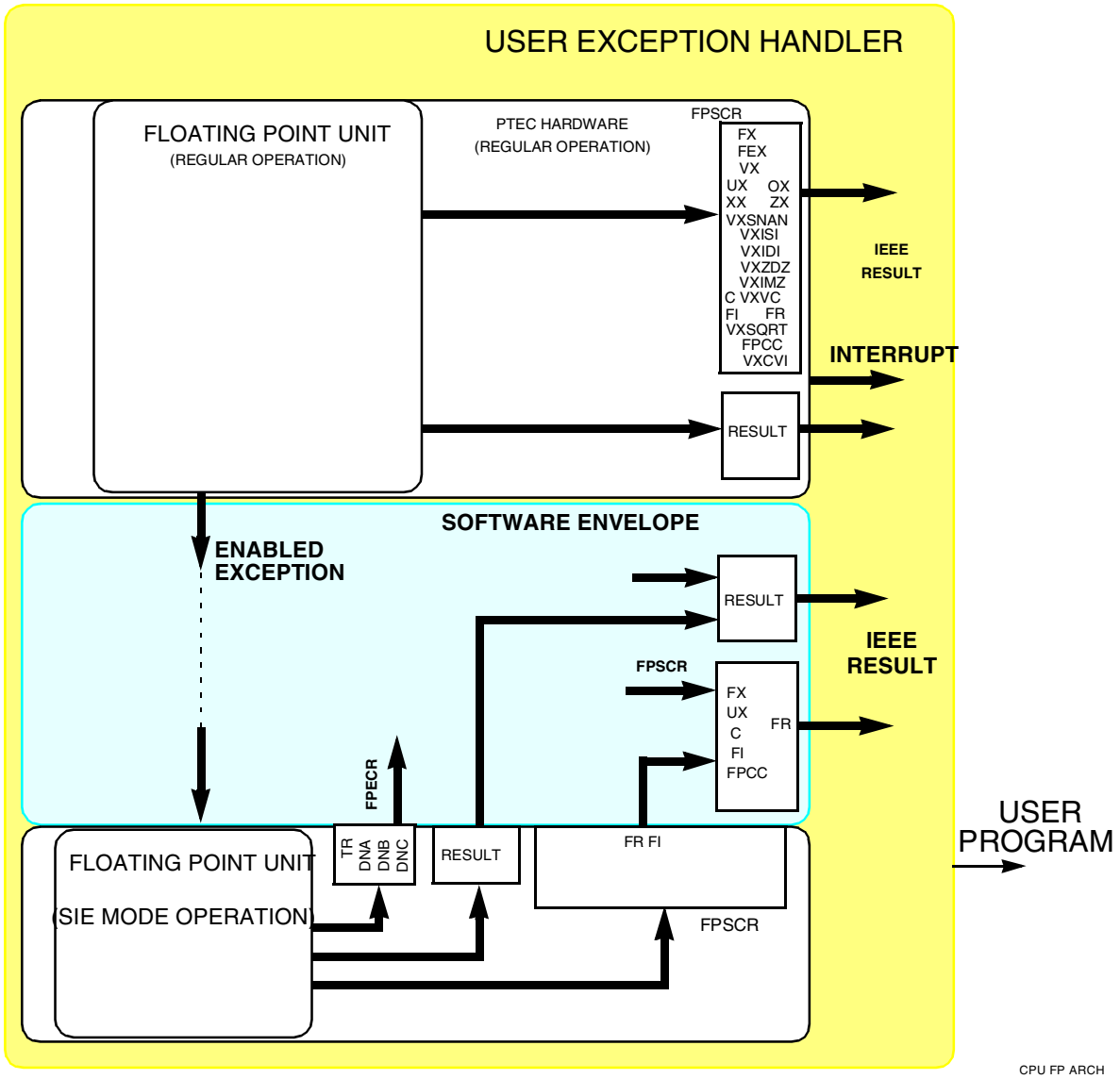

Figure 6-2 RCPU Floating-Point Architecture

6.11.10.2 Floating-Point Assist for Denormalized Operands

When a denormalized operand is detected there are some cases in which the processor needs the assistance of the software to perform the operation. In these cases the software envelope is invoked. **Table 6-24** summarizes the hardware/software partitioning in handling denormalized operands in the input stage of the execution units. The ranges referred to in the table are defined in **Figure 6-3**.

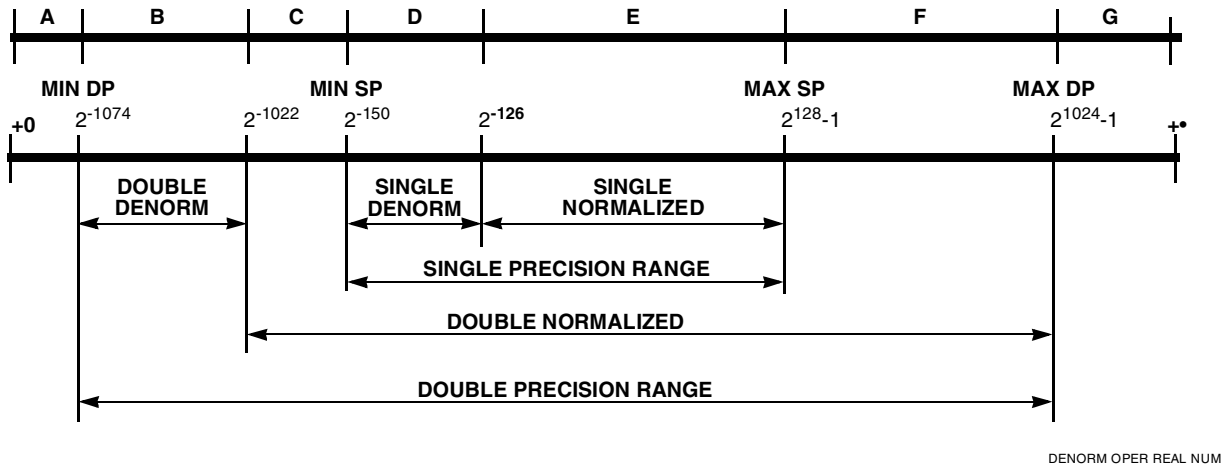

Figure 6-3 Real Numbers Axis for Denormalized Operands

Table 6-24 Software/Hardware Partitioning in Operands Treatment

Instruction	Range B	Range C	Range D	Range E	Range F
Load single	NA	NA	Floating-Point Assist	Hardware	NA
Load double	Hardware	Hardware	Hardware	Hardware	Hardware
Store single	Programming error[1]	Programming error[1]	Floating-Point Assist	Hardware[2]	Programming error
Store double	Hardware	Hardware	Hardware	Hardware	Hardware
FP arithmetic & Multiply-add single	Programming error[1]	Programming error[1]	Hardware[2]	Hardware[2]	Programming error[1]
FP arithmetic & Multiply-add double	Floating-Point Assist	Hardware	Hardware	Hardware	Hardware
Round to single	Floating-Point Assist	Hardware[3]	Hardware[3]	Hardware	Hardware
FP compare, FP move, convert to integer & FPSCR instr.	Hardware	Hardware	Hardware	Hardware	Hardware

NOTES:

1. The results in all cases of programming errors are boundedly undefined.
2. When used by a single precision instruction, generates correct result only if bits [35:63] of the operand equal zero, otherwise it is a programming error.
3. Since the result is tiny, a floating-point assist exception is taken at the end of instruction execution.

6.11.10.3 Synchronized Ignore Exceptions (SIE) Mode

The software envelope uses SIE mode to emulate instructions executed by the floating-point unit. (This mode is not used to emulate floating-point instructions executed by the load/store unit.) In SIE mode the floating-point unit does the following:

- Re-executes the instruction (without generating a floating-point assist exception a second time)
- Generates default results in hardware
- Updates the FPSCR
- Updates the floating-point exceptions cause register (FPECR).

The FPECR is a special-purpose register used by the software envelope. It contains four status bits indicating whether the result of the operation is tiny and whether any of three source operands are denormalized. In addition, it contains one control bit to enable or disable SIE mode. This register must not be accessed by user code. Refer to **6.11.10.4 Floating-Point Exception Cause Register** for more information.

If as a result of the operation performed in SIE mode, ((MSR[FE0] | MSR[FEf1]) & FPSCR[FEX]) is set, a program exception is taken. It is the responsibility of the software envelope to make sure that when executing an instruction in SIE mode ((MSR[FE0] | MSR[FE1]) = 0).

Except when the result is tiny or when denormalized operands are detected, the results generated by the hardware in SIE mode are practically all that is needed in order to complete the operation according to the IEEE standard. Therefore, in most cases after executing the instruction in SIE mode all that is needed by the software is to issue **rfi.** Upon execution of the **rfi,** the hardware restores the previous value of the MSR, as it was saved in SRR1. If as a result ((MSR[FE0] | MSR[FE1]) & FPSCR[FEX]) is set, a program exception is generated.

When the result is tiny and the floating-point underflow exception is disabled (FPSCR[UE] = 0), the hardware in SIE mode delivers the same result as when the exception is enabled (FPSCR[UE] = 1), (i.e., rounded mantissa with exponent adjusted by adding 192 for single precision or 1536 for double precision). This intermediate result simplifies the task of the emulation routine that finishes the instruction execution and delivers the correct IEEE result. In this case the software envelope is responsible for updating the floating-point underflow exception bit (FPSCR[UX]) as well.

When at least one of the source operands is denormalized and the hardware can not complete the operation, the destination register value is unchanged. In this case, the software emulation routine must execute the instruction in software, deliver the result to the destination register, and update the FPSCR.

6.11.10.4 Floating-Point Exception Cause Register

The FPECR is a special-purpose register used by the software envelope. It con-

tains four status bits indicating whether the result of the operation is tiny and whether any of three source operands are denormalized. In addition, it contains one control bit to enable or disable SIE mode. This register must not be accessed by user code.

FPECR — Floating-Point Exception Cause Register

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
SIE	RESERVED														
RESET:															
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
RESERVED												DNC	DNB	DNA	TR
RESET:															
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

A listing of FPECR bit settings is shown in **Table 6-26**.

Table 6-25 FPECR Bit Settings

Bit(s)	Name	Description
0	SIE	SIE mode control bit 0 Disable SIE mode 1 Enable SIE mode
[1:27]	—	Reserved
28	DNC	Source operand C denormalized status bit 0 Source operand C is not denormalized 1 Source operand C is denormalized
29	DNB	Source operand B denormalized status bit 0 Source operand B is not denormalized 1 Source operand B is denormalized
30	DNA	Source operand A denormalized status bit 0 Source operand A is not denormalized 1 Source operand A is denormalized
31	TR	Floating-point tiny result 0 Floating-point result is not tiny 1 Floating-point result is tiny

NOTE

Software must insert a **sync** instruction before reading the FPECR.

6.11.10.5 Floating-Point Enabled Exceptions

Floating-point exceptions are signaled by condition bits set in the floating-point status and control register (FPSCR). They can cause the system floating-point enabled exception error handler to be invoked. All floating-point exceptions are handled precisely. The FPSCR is shown below.

FPSCR — Floating-Point Status and Control Register

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
FX	FEX	VX	OX	UX	ZX	XX	VXS-NAN	VXISI	VXIDI	VXZDZ	VXIMZ	VXVC	FR	FI	FPRF0
RESET:															
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
FPRF[16:19]				0	VX-SOFT	VX-SQRT	VXCVI	VE	OE	UE	ZE	XE	NI	RN	
RESET:															
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

A listing of FPSCR bit settings is shown in **Table 6-26**.

Table 6-26 FPSCR Bit Settings

Bit(s)	Name	Description
0	FX	Floating-point exception summary (FX). Every floating-point instruction implicitly sets FPSCR[FX] if that instruction causes any of the floating-point exception bits in the FPSCR to change from zero to one. The **mcrfs** instruction implicitly clears FPSCR[FX] if the FPSCR field containing FPSCR[FX] is copied. The **mtfsf**, **mtfsfi**, **mtfsb0**, and **mtfsb1** instructions can set or clear FPSCR[FX] explicitly. This is a sticky bit.
1	FEX	Floating-point enabled exception summary (FEX). This bit signals the occurrence of any of the enabled exception conditions. It is the logical OR of all the floating-point exception bits masked with their respective enable bits. The **mcrfs** instruction implicitly clears FPSCR[FEX] if the result of the logical OR described above becomes zero. The **mtfsf**, **mtfsfi**, **mtfsb0**, and **mtfsb1** instructions cannot set or clear FPSCR[FEX] explicitly. This is not a sticky bit.
2	VX	Floating-point invalid operation exception summary (VX). This bit signals the occurrence of any invalid operation exception. It is the logical OR of all of the invalid operation exceptions. The **mcrfs** instruction implicitly clears FPSCR[VX] if the result of the logical OR described above becomes zero. The **mtfsf**, **mtfsfi**, **mtfsb0**, and **mtfsb1** instructions cannot set or clear FPSCR[VX] explicitly. This is not a sticky bit.
3	OX	Floating-point overflow exception (OX). This is a sticky bit. See **6.11.10.8 Overflow Exception Condition**.
4	UX	Floating-point underflow exception (UX). This is a sticky bit. See **6.11.10.9 Underflow Exception Condition**.

Table 6-26 FPSCR Bit Settings (Continued)

Bit(s)	Name	Description
5	ZX	Floating-point zero divide exception (ZX). This is a sticky bit. See **6.11.10.7 Zero Divide Exception Condition**.
6	XX	Floating-point inexact exception (XX). This is a sticky bit. See **6.11.10.10 Inexact Exception Condition**.
7	VXSNAN	Floating-point invalid operation exception for SNaN (VXSNAN). This is a sticky bit. See **6.11.10.6 Invalid Operation Exception Conditions**.
8	VXISI	Floating-point invalid operation exception for ×-× (VXISI). This is a sticky bit. See **6.11.10.6 Invalid Operation Exception Conditions**.
9	VXIDI	Floating-point invalid operation exception for ×/× (VXIDI). This is a sticky bit. See **6.11.10.6 Invalid Operation Exception Conditions**.
10	VXZDZ	Floating-point invalid operation exception for 0/0 (VXZDZ). This is a sticky bit. See **6.11.10.6 Invalid Operation Exception Conditions**.
11	VXIMZ	Floating-point invalid operation exception for ×*0 (VXIMZ). This is a sticky bit. See **6.11.10.6 Invalid Operation Exception Conditions**.
12	VXVC	Floating-point invalid operation exception for invalid compare (VXVC). This is a sticky bit. See **6.11.10.6 Invalid Operation Exception Conditions**.
13	FR	Floating-point fraction rounded (FR). The last floating-point instruction that potentially rounded the intermediate result incremented the fraction. (See **3.3.11 Rounding**.) This bit is not sticky.
14	FI	Floating-point fraction inexact (FI). The last floating-point instruction that potentially rounded the intermediate result produced an inexact fraction or a disabled exponent overflow. (See **3.3.11 Rounding**.) This bit is not sticky.
[15:19]	FPRF	Floating-point result flags (FPRF). This field is based on the value placed into the target register even if that value is undefined. Refer to **Table 6-27** for specific bit settings. 15 Floating-point result class descriptor (C). Floating-point instructions other than the compare instructions may set this bit with the FPCC bits, to indicate the class of the result. [16:19] Floating-point condition code (FPCC). Floating-point compare instructions always set one of the FPCC bits to one and the other three FPCC bits to zero. Other floating-point instructions may set the FPCC bits with the C bit, to indicate the class of the result. Note that in this case the high-order three bits of the FPCC retain their relational significance indicating that the value is less than, greater than, or equal to zero. 16 Floating-point less than or negative (FL or <) 17 Floating-point greater than or positive (FG or >) 18 Floating-point equal or zero (FE or =) 19 Floating-point unordered or NaN (FU or?)
20	—	Reserved
21	VXSOFT	Floating-point invalid operation exception for software request (VXSOFT). This bit can be altered only by the **mcrfs**, **mtfsfi**, **mtfsf**, **mtfsb0**, or **mtfsb1** instructions. The purpose of VXSOFT is to allow software to cause an invalid operation condition for a condition that is not necessarily associated with the execution of a floating-point instruction. For example, it might be set by a program that computes a square root if the source operand is negative. This is a sticky bit. See **6.11.10.6 Invalid Operation Exception Conditions**.
22	VXSQRT	Floating-point invalid operation exception for invalid square root (VXSQRT). This is a sticky bit. This guarantees that software can simulate **fsqrt** and **frsqrte**, and to provide a consistent interface to handle exceptions caused by square-root operations. See**6.11.10.6 Invalid Operation Exception Conditions**.

Table 6-26 FPSCR Bit Settings (Continued)

Bit(s)	Name	Description
23	VXCVI	Floating-point invalid operation exception for invalid integer convert (VXCVI). This is a sticky bit. See **6.11.10.6 Invalid Operation Exception Conditions**.
24	VE	Floating-point invalid operation exception enable (VE). See **6.11.10.6 Invalid Operation Exception Conditions**.
25	OE	Floating-point overflow exception enable (OE). See **6.11.10.8 Overflow Exception Condition**.
26	UE	Floating-point underflow exception enable (UE). This bit should not be used to determine whether denormalization should be performed on floating-point stores. See **6.11.10.9 Underflow Exception Condition**.
27	ZE	Floating-point zero divide exception enable (ZE). See **6.11.10.7 Zero Divide Exception Condition**.
28	XE	Floating-point inexact exception enable (XE). See **6.11.10.10 Inexact Exception Condition**.
29	NI	Non-IEEE mode bit. See **3.4.3 Non-IEEE Operation**.
[30:31]	RN	Floating-point rounding control (RN). See **3.3.11 Rounding**. 00 Round to nearest 01 Round toward zero 10 Round toward +infinity 11 Round toward –infinity

Table 6-27 illustrates the floating-point result flags that correspond to FPSCR[15:19].

Table 6-27 Floating-Point Result Flags in FPSCR

Result Flags FPSCR[15:19] C<>=?	Result Value Class
10001	Quiet NaN
01001	–Infinity
01000	–Normalized number
11000	–Denormalized number
10010	–Zero
00010	+ Zero
10100	+ Denormalized number
00100	+Normalized number
00101	+Infinity

The following conditions cause floating-point assist exceptions when the corresponding enable bit in the FPSCR is set and the FE field in the MSR has a nonzero value (enabling floating-point exceptions). These conditions may occur during execution of floating-point arithmetic instructions. The corresponding status bits in the

FPSCR are indicated in parentheses.

- Invalid floating-point operation exception condition (VX)
  — SNaN condition (VXSNAN)
  — Infinity–infinity condition (VXISI)
  — Infinity/infinity condition (VXIDI)
  — Zero/zero condition (VXZDZ)
  — Infinity*zero condition (VXIMZ)
  — Illegal compare condition (VXVC)
  These exception conditions are described in **6.11.10.6 Invalid Operation Exception Conditions**.
- Software request condition (VXSOFT). These exception conditions are described in **6.11.10.6 Invalid Operation Exception Conditions**.
- Illegal integer convert condition (VXCVI). These exception conditions are described in **6.11.10.6 Invalid Operation Exception Conditions**.
- Zero divide exception condition (ZX). These exception conditions are described in **6.11.10.7 Zero Divide Exception Condition**.
- Overflow Exception Condition (OX). These exception conditions are described in **6.11.10.8 Overflow Exception Condition**.
- Underflow Exception Condition (UX). These exception conditions are described in **6.11.10.9 Underflow Exception Condition**.
- Inexact Exception Condition (XX). These exception conditions are described in **6.11.10.10 Inexact Exception Condition**.

Each floating-point exception condition and each category of illegal floating-point operation exception condition have a corresponding exception bit in the FPSCR. In addition, each floating-point exception has a corresponding enable bit in the FPSCR. The exception bit indicates the occurrence of the corresponding condition. If a floating-point exception occurs, the corresponding enable bit governs the result produced by the instruction and, in conjunction with bits FE0 and FE1, whether and how the system floating-point enabled exception error handler is invoked. (The “enabling” specified by the enable bit is of invoking the system error handler, not of permitting the exception condition to occur. The occurrence of an exception condition depends only on the instruction and its inputs, not on the setting of any control bits.)

The floating-point exception summary bit (FX) in the FPSCR is set when any of the exception condition bits transitions from a zero to a one or when explicitly set by software. The floating-point enabled exception summary bit (FEX) in the FPSCR is set when any of the exception condition bits is set and the exception is enabled (enable bit is one).

A single instruction may set more than one exception condition bit in the following cases:

- The inexact exception condition bit may be set with overflow exception condition.
- The inexact exception condition bit may be set with underflow exception condition.
- The illegal floating-point operation exception condition bit (SNaN) may be set

with illegal floating-point operation exception condition (×*0) for multiply-add instructions.
- The illegal operation exception condition bit (SNaN) may be set with illegal floating-point operation exception condition (illegal compare) for compare ordered instructions.
- The illegal floating-point operation exception condition bit (SNaN) may be set with illegal floating-point operation exception condition (illegal integer convert) for convert to integer instructions.

When an exception occurs, the instruction execution may be suppressed or a result may be delivered, depending on the exception condition.

Instruction execution is suppressed for the following kinds of exception conditions, so that there is no possibility that one of the operands is lost:

- Enabled illegal floating-point operation
- Enabled zero divide

For the remaining kinds of exception conditions, a result is generated and written to the destination specified by the instruction causing the exception. The result may be a different value for the enabled and disabled conditions for some of these exception conditions. The kinds of exception conditions that deliver a result are the following:

- Disabled illegal floating-point operation
- Disabled zero divide
- Disabled overflow
- Disabled underflow
- Disabled inexact
- Enabled overflow
- Enabled underflow
- Enabled inexact

Subsequent sections define each of the floating-point exception conditions and specify the action taken when they are detected.

The IEEE standard specifies the handling of exception conditions in terms of traps and trap handlers. In the PowerPC architecture, setting an FPSCR exception enable bit causes generation of the result value specified in the IEEE standard for the trap enabled case — the expectation is that the exception is detected by software, which will revise the result. An FPSCR exception enable bit of zero causes generation of the default result value specified for the trap disabled (or no trap occurs or trap is not implemented) case — the expectation is that the exception will not be detected by software, which will simply use the default result. The result to be delivered in each case for each exception is described in the following sections.

The IEEE default behavior when an exception occurs, which is to generate a default value and not to notify software, is obtained by clearing all FPSCR exception enable bits and using ignore exceptions mode (see **Table 6-8**). In this case the system floating-point assist error handler is not invoked, even if floating-point excep-

tions occur. If necessary, software can inspect the FPSCR exception bits to determine whether exceptions have occurred.

If the program exception handler notifies software that a given exception condition has occurred, the corresponding FPSCR exception enable bit must be set and a mode other than ignore exceptions mode must be used. In this case the system floating-point assist error handler is invoked if an enabled floating-point exception condition occurs.

Whether the system floating-point enabled exception error handler is invoked if an enabled floating-point exception occurs is controlled by MSR bits FE0 and FE1 as shown in **Table 6-8**. (The system floating-point enabled exception error handler is never invoked because of a disabled floating-point exception.)

Table 6-28 Floating-Point Exception Mode Bits

FE[0:1]	Mode
00	Ignore exceptions mode — Floating-point exceptions do not cause the floating-point assist error handler to be invoked.
01, 10, 11	Floating-point precise mode — The system floating-point assist error handler is invoked precisely at the instruction that caused the enabled exception.

Whenever the system floating-point enabled exception error handler is invoked, the processor ensures that all instructions logically residing before the excepting instruction have completed, and no instruction after that instruction has been executed.

If exceptions are ignored, an FPSCR instruction can be used to force any exceptions caused by instructions initiated before the FPSCR instruction to be recorded in the FPSCR. A **sync** instruction can also be used to force exceptions, but is likely to degrade performance more than an FPSCR instruction.

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

- If the IEEE default results are acceptable to the application, FE0 and FE1 should be cleared (ignore exceptions mode). All FPSCR exception enable bits should be cleared.
- For even faster operation, non-IEEE can be selected by setting the NI bit in the FPSCR. To ensure that the software envelope is never invoked, select non-IEEE mode, disable all floating-point exceptions, and avoid using denormalized numbers as input to floating-point calculations. Refer to **3.4.3 Non-IEEE Operation** and **3.4.4 Working Without the Software Envelope** for more information.
- Ignore exceptions mode should not, in general, be used when any FPSCR exception enable bits are set.

- Precise mode may degrade performance in some implementations, perhaps substantially, and therefore should be used only for debugging and other specialized applications.

6.11.10.6 Invalid Operation Exception Conditions

An invalid operation exception occurs when an operand is invalid for the specified operation. The invalid operations are as follows:

- Any operation except load, store, move, select, or **mtfsf** on a signaling NaN (SNaN)
- For add or subtract operations, magnitude subtraction of infinities (×-×)
- Division of infinity by infinity (×/×)
- Division of zero by zero (0/0)
- Multiplication of infinity by zero (×*0)
- Ordered comparison involving a NaN (invalid compare)
- Square root or reciprocal square root of a negative, non-zero number (invalid square root)
- Integer convert involving a number that is too large to be represented in the format, an infinity, or a NaN (invalid integer convert)

FPSCR[VXSOFT] allows software to cause an invalid operation exception for a condition that is not necessarily associated with the execution of a floating-point instruction. For example, it might be set by a program that computes a square root if the source operand is negative. This facilitates the emulation of PowerPC instructions not implemented in the RCPU.

When an invalid-operation exception occurs, the action to be taken depends on the setting of the invalid operation exception enable bit of the FPSCR. When invalid operation exception is enabled (FPSCR[VE] = 1) and invalid operation occurs or software explicitly requests the exception, the following actions are taken:

- The following status bits are set in the FPSCR:
  - VXSNAN (if SNaN)
  - VXISI (if ×-×)
  - VXIDI (if ×/×)
  - VXZDZ (if 0/0)
  - VXIMZ (if ×*0)
  - VXVC (if invalid comparison)
  - VXSOFT (if software request)
  - VXCVI (if invalid integer convert)
- If the operation is an arithmetic or convert-to-integer operation,
  - the target FPR is unchanged
  - FPSCR[FR] and FPSCR[FI] are cleared
  - FPSCR[FPRF] is unchanged
- If the operation is a compare,
  - the FR, FI, and C bits in the FPSCR are unchanged
  - FPSCR[FPCC] is set to reflect unordered
- If software explicitly requests the exception, FPSCR[FR FI FPRF] are as set by the **mtfsfi**, **mtfsf**, or **mtfsb1** instruction

When invalid operation exception condition is disabled (FPSCRVE = 0) and invalid operation occurs or software explicitly requests the exception, the following actions are taken:

- The same status bits are set in the FPSCR as when the exception is enabled.
- If the operation is an arithmetic operation,
  — the target FPR is set to a quiet NaN
  — FPSCR[FR] and FPSCR[FI] are cleared
  — FPSCR[FPRF] is set to indicate the class of the result (quiet NaN)
- If the operation is a convert to 32-bit integer operation, the target FPR is set as follows:
  — FRT[0:31] = undefined
  — FRT[32:63] = most negative 32-bit integer
  — FPSCR[FR] and FPSCR[FI] are cleared
  — FPSCR[FPRF] is undefined
- If the operation is a convert to 64-bit integer operation, the target FPR is set as follows:
  — FRT[0:63] = most negative 64-bit integer
  — FPSCR[FR] and FPSCR[FI] are cleared
  — FPSCR[FPRF] is undefined
- If the operation is a compare,
  — The FR, FI, and C bits in the FPSCR are unchanged
  — FPSCR[FPCC] is set to reflect unordered
- If software explicitly requests the exception, the FR, FI and FPRF fields in the FPSCR are as set by the **mtfsfi**, **mtfsf**, or **mtfsb1** instruction.

6.11.10.7 Zero Divide Exception Condition

A zero divide exception condition occurs when a divide instruction is executed with a zero divisor value and a finite, non-zero dividend value.

When a zero divide exception occurs, the action to be taken depends on the setting of the zero divide exception condition enable bit of the FPSCR. When the zero divide exception condition is enabled (FPSCR[ZE] = 1) and a zero divide condition occurs, the following actions are taken:

- Zero divide exception condition bit is set: FPSCR[ZX] = 1
- The target FPR is unchanged
- FPSCR[FR] and FPSCR[FI] are cleared
- FPSCR[FPRF] is unchanged

When zero divide exception condition is disabled (FPSCR[ZE] = 0) and zero divide occurs, the following actions are taken:

- Zero divide exception condition bit is set: FPSCR[ZX] = 1
- The target FPR is set to a ±infinity, where the sign is determined by the XOR of the signs of the operands
- FPSCR[FR] and FPSCR[FI] are cleared
- FPSCR[FPRF] is set to indicate the class and sign of the result (±infinity)

6.11.10.8 Overflow Exception Condition

Overflow occurs when the magnitude of what would have been the rounded result if the exponent range were unbounded exceeds that of the largest finite number of the specified result precision.

The action to be taken depends on the setting of the overflow exception condition enable bit of the FPSCR. When the overflow exception condition is enabled (FPSCR[OE] = 1) and an exponent overflow condition occurs, the following actions are taken:

- Overflow exception condition bit is set: FPSCR[OX] = 1.
- For double-precision arithmetic instructions, the exponent of the normalized intermediate result is adjusted by subtracting 1536.
- For single-precision arithmetic instructions and the floating round to single-precision instruction, the exponent of the normalized intermediate result is adjusted by subtracting 192.
- The adjusted rounded result is placed into the target FPR.
- FPSCR[FPRF] is set to indicate the class and sign of the result (±normal number).

When the overflow exception condition is disabled (FPSCR[OE] = 0) and an overflow condition occurs, the following actions are taken:

- Overflow exception condition bit is set: FPSCR[OX] = 1
- Inexact exception condition bit is set: FPSCR[XX] = 1
- The result is determined by the rounding mode (FPSCR[RN]) and the sign of the intermediate result as follows:
  - — Round to nearest
    Store ± infinity, where the sign is the sign of the intermediate result
  - — Round toward zero
    Store the format's largest finite number with the sign of the intermediate result
  - — Round toward +infinity
    For negative overflows, store the format's most negative finite number; for positive overflows, store +infinity
  - — Round toward –infinity
    For negative overflows, store –infinity; for positive overflows, store the format's largest finite number
- The result is placed into the target FPR
- FPSCR[FR FI] are cleared
- FPSCR[FPRF] is set to indicate the class and sign of the result (±infinity or ±normal number)

6.11.10.9 Underflow Exception Condition

The underflow exception condition is defined separately for the enabled and disabled states:

- Enabled — Underflow occurs when the intermediate result is tiny.
- Disabled — Underflow occurs when the intermediate result is tiny and there is loss of accuracy.

A tiny result is detected before rounding, when a non-zero result value computed as though the exponent range were unbounded would be less in magnitude than the smallest normalized number.

If the intermediate result is tiny and the underflow exception condition enable bit is cleared (FPSCR[UE] = 0), the intermediate result is denormalized.

Loss of accuracy is detected when the delivered result value differs from what would have been computed were both the exponent range and precision unbounded.

When an underflow exception occurs, the action to be taken depends on the setting of the underflow exception condition enable bit of the FPSCR.

When the underflow exception condition is enabled (FPSCR[UE] = 1) and an exponent underflow condition occurs, the following actions are taken:

- Underflow exception condition bit is set: FPSCR[UX] = 1.
- For double-precision arithmetic and conversion instructions, the exponent of the normalized intermediate result is adjusted by adding 1536.
- For single-precision arithmetic instructions and the floating round to single-precision instruction, the exponent of the normalized intermediate result is adjusted by adding 192.
- The adjusted rounded result is placed into the target FPR.
- FPSCR[FPRF] is set to indicate the class and sign of the result (±normalized number).

The FR and FI bits in the FPSCR allow the system floating-point enabled exception error handler, when invoked because of an underflow exception condition, to simulate a trap disabled environment. That is, the FR and FI bits allow the system floating-point enabled exception error handler to unround the result, thus allowing the result to be denormalized.

When the underflow exception condition is disabled (FPSCR[UE] = 0) and an underflow condition occurs, the following actions are taken:

- Underflow exception condition enable bit is set: FPSCR[UX] = 1
- The rounded result is placed into the target FPR
- FPSCR[FPRF] is set to indicate the class and sign of the result (±denormalized number or ±zero)

6.11.10.10 Inexact Exception Condition

The inexact exception condition occurs when one of two conditions occur during rounding:

- The rounded result differs from the intermediate result assuming the intermediate result exponent range and precision to be unbounded.
- The rounded result overflows and overflow exception condition is disabled.

When the inexact exception condition occurs, regardless of the setting of the inexact exception condition enable bit of the FPSCR, the following actions are taken:

- Inexact exception condition enable bit in the FPSCR is set: FPSCR[XX] = 1.
- The rounded or overflowed result is placed into the target FPR.
- FPSCR[FPRF] is set to indicate the class and sign of the result.

6.11.11 Software Emulation Exception (0x01000)

An implementation-dependent software emulation exception occurs in the following cases:

- An attempt is made to execute an instruction that is not implemented in the RCPU. This includes all illegal and optional instructions. Since an RCPU-based MCU does not contain a data cache, segment registers, or a translation lookaside buffer, the following optional PowerPC instructions cause the RCPU to generate a software emulation exception:
  — Data cache instructions (**dcbt**, **dcbtst**, **dcbz**, **dcbst**, **dcbf**, **dcbi**)
  — Instructions to access segment registers (**mtsr**, **mfsr**, **mtsrin**, **mfsrin**)
  — Instructions to manage translation lookaside buffers (**tlbei**, **tlbiex**, **tlbsync**, **tlbie**, **tlbia**)
- An attempt is made to execute an **mtspr** or **mfspr** instruction that specifies an unimplemented internal-to-the-processor SPR. (This exception is taken regardless of the value of the SPR0 bit of the instruction. That is, if the SPR0 bit of the instruction equals one, indicating a privileged register, and the processor is operating in user mode, this exception is taken rather than a program exception.)

  Refer to **7.5 Implementation of Special-Purpose Registers** for an explanation of internal- and external-to-the-processor SPRs.
- An attempt is made to execute a **mtspr** or **mfspr** instruction that specifies an unimplemented external-to-the-processor register, and either SPR0 = 0 or MSR[PR] = 0 (no program exception condition).

Register settings after a software emulation exception is taken are shown in **Table 6-22**.

Table 6-29 Register Settings Following a Software Emulation Exception

Register	Setting Description	
SRR0	Set to the effective address of the instruction that caused the exception	
SRR1	[0:15] [16:31]	Cleared to zero Loaded from MSR[16:31]
MSR	IP ME LE Other bits	No change No change Set to value of ILE bit prior to the exception Cleared

When a software emulation exception is taken, execution resumes at offset 0x01000 from the base address indicated by MSR[IP].

6.11.12 Data Breakpoint Exception (0x01C00)

An implementation-dependent data (L-bus) breakpoint occurs when an internal breakpoint match occurs on the load/store bus.

The processor can be programmed to recognize a data breakpoint at all times (non-masked mode), or only when the MSR[RI] bit is set (masked mode). When operating in non-masked mode, the processor enters a non-restartable state if it recognizes an internal breakpoint when MSR[RI] is cleared.

In order to enable the user to use the breakpoint features without adding restrictions on the software, the address of the load/store cycle that generated the data breakpoint is not stored in the DAR (data address register), as with other exceptions that occur during loads or stores. Instead, the address of the load/store cycle that generated the breakpoint is stored in an implementation dependent register called the breakpoint address register (BAR).

Register settings after a data breakpoint exception is taken are shown in **Table 6-22**.

Table 6-30 Register Settings Following Data Breakpoint Exception

Register	Setting Description
SRR0	Set to the effective address of the instruction following the instruction that caused the exception
SRR1	[0:15] Cleared to zero [16:31] Loaded from bits MSR[16:31] If development port request is asserted at reset, the value of SRR1 is undefined.
MSR	IP No change ME No change LE Set to value of ILE bit prior to the exception Other bits Cleared
BAR	Set to the effective address of the data access as computed by the instruction that caused the exception.
DSISR, DAR	No change

When a data breakpoint exception is taken, execution resumes at offset 0x01C00 from the base address indicated by MSR[IP].

Refer to **SECTION 8 DEVELOPMENT SUPPORT** for additional information on data breakpoints.

6.11.13 Instruction Breakpoint Exception (0x01D00)

An implementation-dependent instruction (I-bus) breakpoint occurs when an internal breakpoint match occurs on the instruction bus.

The processor can be programmed to recognize a data breakpoint at all times (non-masked mode), or only when the MSR[RI] bit is set (masked mode). When operating in non-masked mode, the processor enters a non-restartable state if it recognizes an internal breakpoint when MSR[RI] is cleared.

Register settings after an instruction breakpoint exception is taken are shown in **Table 6-22**.

Table 6-31 Register Settings Following an Instruction Breakpoint Exception

Register	Setting Description
SRR0	Set to the effective address of the instruction that caused the exception
SRR1	[0:15] Cleared to zero [16:31] Loaded fromMSR[16:31] If development port request is asserted at reset, the value of SRR1 is undefined.

Table 6-31 Register Settings Following an Instruction Breakpoint Exception

Register	Setting Description	
MSR	IP	No change
	ME	No change
	LE	Set to value of ILE bit prior to the exception
	Other bits	Cleared

When an instruction breakpoint exception is taken, execution resumes at offset 0x01D00 from the base address indicated by MSR[IP].

Refer to **SECTION 8 DEVELOPMENT SUPPORT** for more information on instruction breakpoint exceptions.

6.11.14 Maskable External Breakpoint Exception (0x01E00)

An implementation-dependent maskable external breakpoint can be generated by any of the peripherals of the system, including those found on the L-bus, I-bus, IMB2 and external bus, and also by an external development system. Peripherals found on the external bus use the serial interface of the development port to assert the external breakpoint. Breakpoints are generated by the development port from the associated bits of the trap enable control register.

Maskable external breakpoint exceptions are asynchronous and ordered. The processor does not take the exception if the RI (recoverable exception) bit in the MSR is cleared. Refer to **SECTION 8 DEVELOPMENT SUPPORT** for more information.

Table 6-32 Register Settings Following a Maskable External Breakpoint Exception

Register	Setting Description	
SRR0	Set to the effective address of the instruction that caused the exception	
SRR1	[0:15]	Cleared to zero
	[16:31]	Loaded fromMSR[16:31]
	If development port request is asserted at reset, the value of SRR1 is undefined.	
MSR	IP	No change
	ME	No change
	LE	Set to value of ILE bit prior to the exception
	Other bits	Cleared to zero

When a maskable external breakpoint exception is taken, execution resumes at offset 0x01E00 from the base address indicated by MSR[IP].

6.11.15 Non-Maskable External Breakpoint Exception (0x01F00)

An implementation-dependent non-maskable external breakpoint exception is generated by the development port from the associated bits of the trap enable mode serial communications.

This exception is asynchronous and unordered. The exception is not referenced to any particular instruction. The processor stops instruction execution and either begins exception processing or enters debug mode as soon as possible after detecting the breakpoint exception.

The non-maskable external breakpoint exception causes the processor to stop without regard to the state of the MSR[RI] bit. If the processor is in a non-recoverable state when the exception occurs, the state of the SRR0, SRR1, DAR, and DSISR registers may have been overwritten. In this case, it is not possible to restart the processor since the restarting address and MSR context are saved in the SRR0 and SRR1 registers.

This exception allows the user to stop the processor in cases in which it would otherwise not stop, but with the penalty that the processor may not be restartable. The value of the MSR[RI] bit, as saved in the SRR1 register, indicates whether the processor stopped in a recoverable state or not.

Table 6-33 Register Settings Following a Non-Maskable External Breakpoint Exception

Register	Setting Description
SRR0	Set to the effective address of the instruction that would have been executed next if no exception had occurred. If the development port request is asserted at reset, the value of SRR0 is undefined.
SRR1	[0:15] Cleared to zero [16:31] Loaded from bits [16:31] of the MSR If development port request is asserted at reset, the value of SRR1 is undefined.
MSR	IP No change ME No change LE Set to value of ILE bit prior to the exception Other bits Cleared to zero

When a non-maskable external breakpoint exception is taken, execution resumes at offset 0x01000 from the base address indicated by MSR[IP].