17. CPU Exceptions

This chapter describes the processor exceptions—a general view of the cause and return of an exception, exception vector locations, and the types of exceptions that are supported, including the cause, processing, and servicing of each exception.

17.1 Causing and Returning from an Exception

When the processor takes an exception, the *EXL* bit in the *Status* register is set to 1, which means the system is in Kernel mode. After saving the appropriate state, the exception handler typically changes the *KSU* bits in the *Status* register to Kernel mode and resets the *EXL* bit back to 0. When restoring the state and restarting, the handler restores the previous value of the *KSU* field and sets the *EXL* bit back to 1.

Returning from an exception also resets the *EXL* bit to 0 (see the ERET instruction in Appendix A).

17.2 Exception Vector Locations

The Cold Reset, Soft Reset, and NMI exceptions are always vectored to the dedicated Cold Reset exception vector at an uncached and unmapped address. Addresses for all other exceptions are a combination of a *vector offset* and a *base address*.

The boot-time vectors (when *BEV* = 1 in the *Status* register) are at uncached and unmapped addresses. During normal operation (when *BEV* = 0) the regular exceptions have vectors in cached address spaces; Cache Error is always at an uncached address so that cache error handling can bypass a suspect cache.

The exception vector assignments for the R10000 processor shown in Table 17-1; the addresses are the same as for the R4400.

Table 17-1 Exception Vector Addresses

BEV	Exception Type	Exception Vector Address	
		32-bit	64-bit
	Cold Reset/Soft Reset/ NMI	`0xBFC00000`	`0xFFFFFFFF BFC00000`
BEV=0	TLB Refill (EXL=0)	`0x80000000`	`0xFFFFFFFF 80000000`
	XTLB Refill (EXL=0)	`0x80000080`	`0xFFFFFFFF 80000080`
	Cache Error	`0xA0000100`	`0xFFFFFFFF A0000100`
	Others	`0x80000180`	`0xFFFFFFFF 80000180`
BEV=1	TLB Refill (EXL=0)	`0xBFC00200`	`0xFFFFFFFF BFC00200`
	XTLB Refill (EXL=0)	`0xBFC00280`	`0xFFFFFFFF BFC00280`
	Cache Error	`0xBFC00300`	`0xFFFFFFFF BFC00300`
	Others	`0xBFC00380`	`0xFFFFFFFF BFC00380`

17.3 TLB Refill Vector Selection

In all present implementations of the MIPS III ISA, there are two TLB refill exception vectors:

- one for references to 32-bit address space (TLB Refill)
- one for references to 64-bit address space (XTLB Refill)

Table 17-2 lists the exception vector addresses.

The TLB refill vector selection is based on the address space of the address (*user*, *supervisor*, or *kernel*) that caused the TLB miss, and the value of the corresponding extended addressing bit in the *Status* register (*UX*, *SX*, or *KX*). The current operating mode of the processor is not important except that it plays a part in specifying in which address space an address resides. The *Context* and *XContext* registers are entirely separate page-table-pointer registers that point to and refill from two separate page tables, however these two registers share *BadVPN2* fields (see Chapter 14 for more information). For all TLB exceptions (Refill, Invalid, TLBL or TLBS), the *BadVPN2* fields of both registers are loaded as they were in the R4400.

In contrast to the R10000, the R4400 processor selects the vector based on the current operating mode of the processor (*user*, *supervisor*, or *kernel*) and the value of the corresponding extended addressing bit in the *Status* register (*UX*, *SX* or *KX*). In addition, the *Context* and *XContext* registers are not implemented as entirely separate registers; the *PTEbase* fields are shared. A miss to a particular address goes through either TLB Refill or XTLB Refill, depending on the source of the reference. There can be only be a single page table unless the refill handlers execute address-deciphering and page table selection in software.

> **NOTE:** Refills for the 0.5 Gbyte supervisor mapped region, *sseg/ksseg*, are controlled by the value of *KX* rather than *SX*. This simplifies control of the processor when supervisor mode is not being used.

Table 17-2 lists the TLB refill vector locations, based on the address that caused the TLB miss and its corresponding mode bit.

Table 17-2 TLB Refill Vectors

Space	Address Range	Regions	Exception Vector
Kernel	`0xFFFF FFFF E000 0000` to `0xFFFF FFFF FFFF FFFF`	*kseg3*	Refill (KX=0) or XRefill (KX=1)
Supervisor	`0xFFFF FFFF C000 0000` to `0xFFFF FFFF DFFF FFFF`	*sseg*, *ksseg*	Refill (KX=0) or XRefill (KX=1)
Kernel	`0xC000 0000 0000 0000` to `0xC000 0FFE FFFF FFFF`	*xkseg*	XRefill(KX=1)
Supervisor	`0x4000 0000 0000 0000` to `0x4000 0FFF FFFF FFFF`	*xsseg*, *xksseg*	XRefill (SX=1)
User	`0x0000 0000 8000 0000` to `0x0000 0FFF FFFF FFFF`	*xsuseg*, *xuseg*, *xkuseg*	XRefill (UX=1)
User	`0x0000 0000 0000 0000` to `0x0000 0000 7FFF FFFF`	*useg*, *xuseg*, *suseg*, *xsuseg*, *kuseg*, *xkuseg*	Refill (UX=0) or XRefill (UX=1)

Priority of Exceptions

The remainder of this chapter describes exceptions in the order of their priority shown in Table 17-3 (with certain of the exceptions, such as the TLB exceptions and Instruction/Data exceptions, grouped together for convenience). While more than one exception can occur for a single instruction, only the exception with the highest priority is reported. Some exceptions are not caused by the instruction executed at the time, and some exceptions may be deferred. See the individual description of each exception in this chapter for more detail.

Table 17-3 Exception Priority Order

Cold Reset *(highest priority)*
Soft Reset
Nonmaskable Interrupt (NMI)*
Cache error — Instruction cache*
Cache error — Data cache*
Cache error — Secondary cache*
Cache error — System interface*
Address error — Instruction fetch
TLB refill — Instruction fetch
TLB invalid — Instruction fetch
Bus error — Instruction fetch
Integer overflow, Trap, System Call, Breakpoint, Reserved Instruction, Coprocessor Unusable, or Floating-Point Exception
Address error — Data access
TLB refill — Data access
TLB invalid — Data access
TLB modified — Data write
Watch*
Bus error — Data access
Interrupt *(lowest priority)**

* These exceptions are interrupt types, and may be imprecise. Priority may not be followed when considering a specific instruction.

Generally speaking, the exceptions described in the following sections are handled (“processed”) by hardware; these exceptions are then serviced by software.

Cold Reset Exception

Cause

The Cold Reset exception is taken for a power-on or “cold” reset; it occurs when the **SysGnt*** signal is asserted while the **SysReset*** signal is also asserted.† This exception is not maskable.

Processing

The CPU provides a special interrupt vector for this exception:

- location 0xBFC0 0000 in 32-bit mode
- location 0xFFFF FFFF BFC0 0000 in 64-bit mode

The Cold Reset vector resides in unmapped and uncached CPU address space, so the hardware need not initialize the TLB or the cache to process this exception. It also means the processor can fetch and execute instructions while the caches and virtual memory are in an undefined state.

The contents of all registers in the CPU are undefined when this exception occurs, except for the following register fields:

- In the *Status* register, *SR* and *TS* are cleared to 0, and *ERL* and *BEV* are set to 1. All other bits are undefined.
- *Config* register is initialized with the boot mode bits read from the serial input.
- The *Random* register is initialized to the value of its upper bound.
- The *Wired* register is initialized to 0.
- The *EW* bit in the *CacheErr* register is cleared.
- The *ErrorEPC* register gets the PC.
- The *FrameMask* register is set to 0.
- Branch prediction bits are set to 0.
- *Performance Counter* register *Event* field is set to 0.
- All pending cache errors, delayed watch exceptions, and external interrupts are cleared.

Servicing

The Cold Reset exception is serviced by:

- initializing all processor registers, coprocessor registers, caches, and the memory system
- performing diagnostic tests
- bootstrapping the operating system

† If **SysGnt*** remains deasserted (high) while **SysReset*** is asserted, the processor interprets this as a Soft Reset exception.

Soft† Reset Exception

Cause

The Soft Reset exception occurs in response to a Soft Reset (See Chapter 8, the section titled “Soft Reset Sequence”).

A Soft Reset exception is not maskable.

The processor differentiates between a Cold Reset and a Soft Reset as follows:

- A Cold Reset occurs when the **SysGnt*** signal is asserted while the **SysReset*** signal is also asserted.
- A Soft Reset occurs if the **SysGnt*** signal remains negated when a **SysReset*** signal is asserted.

In R4400 processor, there is no way for software to differentiate between a Soft Reset exception and an NMI exception. In the R10000 processor, a bit labelled *NMI* has been added to the *Status* register to distinguish between these two exceptions. Both Soft Reset and NMI exceptions set the *SR* bit and use the same exception vector. During an NMI exception, the *NMI* bit is set to 1; during a Soft Reset, the *NMI* bit is set to 0.

Processing

When a Soft Reset exception occurs, the *SR* bit of the *Status* register is set, distinguishing this exception from a Cold Reset exception.

When a Soft Reset is detected, the processor initializes minimum processor state. This allows the processor to fetch and execute the instructions of the exception handler, which in turn dumps the current architectural state to external logic. Hardware state that loses architectural state is not initialized unless it is necessary to execute instructions from unmapped uncached space that reads the registers, TLB, and cache contents.

The Soft Reset can begin on an arbitrary cycle boundary and can abort multicycle operations in progress, so it may alter machine state. Hence, caches, memory, or other processor states can be inconsistent: data cache blocks may stay at the refill state and any cached loads/stores to these blocks will hang the processor. Therefore, CacheOps should be used to dump the cache contents.

After the processor state is read out, the processor should be reset with a Cold Reset sequence.

† Soft Reset is also known colloquially as *Warm* Reset.

A Soft Reset exception preserves the contents of all registers, except for:

- *ErrorEPC* register, which contains the PC
- *ERL* bit of the *Status* register, which is set to 1
- *SR* bit of the *Status* register, which is set to 1 on Soft Reset or an NMI; 0 for a Cold Reset
- *BEV* bit of the *Status* register, which is set to 1
- *TS* bit of the *Status* register, which is set to 0
- PC is set to the reset vector 0xFFFF FFFF BFC0 0000
- clears any pending Cache Error exceptions

Servicing

A Soft Reset exception is intended to quickly reinitialize a previously operating processor after a fatal error.

It is not normally possible to continue program execution after returning from this exception, since a **SysReset*** signal can be accepted anytime.

NMI Exception

Cause

The NMI exception is caused by assertion of the **SysNMI*** signal.

An NMI exception is not maskable.

In R4400 processor, there is no way for software to differentiate between a Soft Reset exception and an NMI exception. In the R10000 processor, a bit labelled *NMI* has been added to the *Status* register to distinguish between these two exceptions. Both Soft Reset and NMI exceptions set the *SR* bit and use the same exception vector. During an NMI exception, the *NMI* bit is set to 1; during a Soft Reset, the *NMI* bit is set to 0.

Processing

When an NMI exception occurs, the *SR* bit of the *Status* register is set, distinguishing this exception from a Cold Reset exception.

An exception caused by an NMI is taken at the instruction boundary. It does not abort any state machines, preserving the state of the processor for diagnosis. The *Cause* register remains unchanged and the system jumps to the NMI exception handler (see Table 17-1).

An NMI exception preserves the contents of all registers, except for:

- *ErrorEPC* register, which contains the PC
- *ERL* bit of the *Status* register, which is set to 1
- *SR* bit of the *Status* register, which is set to 1 on Soft Reset or an NMI; 0 for a Cold Reset
- *BEV* bit of the *Status* register, which is set to 1
- *TS* bit of the *Status* register, which is set to 0
- PC is set to the reset vector 0xFFFF FFFF BFC0 0000
- clears any pending Cache Error exceptions

Servicing

The NMI can be used for purposes other than resetting the processor while preserving cache and memory contents. For example, the system might use an NMI to cause an immediate, controlled shutdown when it detects an impending power failure.

It is not normally possible to continue program execution after returning from this exception, since an NMI can occur during another error exception.

Address Error Exception

Cause

The Address Error exception occurs when an attempt is made to execute one of the following:

- reference to an illegal address space
- reference the supervisor address space from User mode
- reference the kernel address space from User or Supervisor mode
- load or store a doubleword that is not aligned on a doubleword boundary
- load, fetch, or store a word that is not aligned on a word boundary
- load or store a halfword that is not aligned on a halfword boundary

This exception is not maskable.

Processing

The common exception vector is used for this exception. The *AdEL* or *AdES* code in the *Cause* register is set, indicating whether the instruction caused the exception with an instruction reference, load operation, or store operation shown by the *EPC* register and *BD* bit in the *Cause* register.

When this exception occurs, the *BadVAddr* register retains the virtual address that was not properly aligned or that referenced protected address space. The contents of the *VPN* field of the *Context*, *XContext*, and *EntryHi* registers are undefined, as are the contents of the *EntryLo* register.

The *EPC* register contains the address of the instruction that caused the exception, unless this instruction is in a branch delay slot. If it is in a branch delay slot, the *EPC* register contains the address of the preceding branch instruction and the *BD* bit of the *Cause* register is set as indication.

Servicing

The process executing at the time is handed a UNIX SIGSEGV (segmentation violation) signal. This error is usually fatal to the process incurring the exception.

TLB Exceptions

Three types of TLB exceptions can occur:

- TLB Refill occurs when there is no TLB entry that matches an attempted reference to a mapped address space.
- TLB Invalid occurs when a virtual address reference matches a TLB entry that is marked invalid.
- TLB Modified occurs when a store operation virtual address reference to memory matches a TLB entry which is marked valid but is not dirty (the entry is not writable).

The following three sections describe these TLB exceptions.

NOTE: TLB Refill vector selection is also described earlier in this chapter, in the section titled, TLB Refill Vector Selection.

TLB Refill Exception

Cause

The TLB refill exception occurs when there is no TLB entry to match a reference to a mapped address space. This exception is not maskable.

Processing

There are two special exception vectors for this exception; one for references to 32-bit address spaces, and one for references to 64-bit address spaces. The *UX, SX,* and *KX* bits of the *Status* register determine whether the user, supervisor or kernel address spaces referenced are 32-bit or 64-bit spaces; the TLB refill vector is selected based upon the address space of the address causing the TLB miss (user, supervisor, or kernel mode address space), together with the value of the corresponding extended addressing bit in the *Status* register (*UX*, *SX*, or *KX*). The current operating mode of the processor is not important except that it plays a part in specifying in which space an address resides. An address is in *user* space if it is in *useg*, *suseg*, *kuseg*, *xuseg*, *xsuseg*, or *xkuseg* (see the description of virtual address spaces in Chapter 16). An address is in *supervisor* space if it is in *sseg*, *ksseg*, *xsseg* or *xksseg*, and an address is in *kernel* space if it is in either *kseg3* or *xkseg*. *Kseg0*, *kseg1*, and kernel physical spaces (*xkphys*) are kernel spaces but are not mapped.

All references use these vectors when the *EXL* bit is set to 0 in the *Status* register. This exception sets the *TLBL* or *TLBS* code in the *ExcCode* field of the *Cause* register. This code indicates whether the instruction, as shown by the *EPC* register and the *BD* bit in the *Cause* register, caused the miss by an instruction reference, load operation, or store operation.

When this exception occurs, the *BadVAddr*, *Context*, *XContext* and *EntryHi* registers hold the virtual address that failed address translation. The *EntryHi* register also contains the ASID from which the translation fault occurred. The *Random* register normally contains a valid location in which to place the replacement TLB entry. The contents of the *EntryLo* register are undefined. The *EPC* register contains the address of the instruction that caused the exception, unless this instruction is in a branch delay slot, in which case the *EPC* register contains the address of the preceding branch instruction and the *BD* bit of the *Cause* register is set.

Servicing

To service this exception, the contents of the *Context* or *XContext* register are used as a virtual address to fetch memory locations containing the physical page frame and access control bits for a pair of TLB entries. The two entries are placed into the E*ntryLo0/EntryLo1* register; the *EntryHi* and *EntryLo* registers are written into the TLB.

It is possible that the virtual address used to obtain the physical address and access control information is on a page that is not resident in the TLB. This condition is processed by allowing a TLB refill exception in the TLB refill handler. This second exception goes to the common exception vector because the *EXL* bit of the *Status* register is set.

TLB Invalid Exception

Cause

The TLB invalid exception occurs when a virtual address reference matches a TLB entry that is marked invalid (TLB valid bit cleared). This exception is not maskable.

Processing

The common exception vector is used for this exception. The *TLBL* or *TLBS* code in the *ExcCode* field of the *Cause* register is set. This indicates whether the instruction, as shown by the *EPC* register and *BD* bit in the *Cause* register, caused the miss by an instruction reference, load operation, or store operation.

When this exception occurs, the *BadVAddr*, *Context*, *XContext* and *EntryHi* registers contain the virtual address that failed address translation. The *EntryHi* register also contains the ASID from which the translation fault occurred. The *Random* register normally contains a valid location in which to put the replacement TLB entry. The contents of the *EntryLo* registers are undefined.

The *EPC* register contains the address of the instruction that caused the exception unless this instruction is in a branch delay slot, in which case the *EPC* register contains the address of the preceding branch instruction and the *BD* bit of the *Cause* register is set.

Servicing

A TLB entry is typically marked invalid when one of the following is true:

- a virtual address does not exist
- the virtual address exists, but is not in main memory (a page fault)
- a trap is desired on any reference to the page (for example, to maintain a reference bit)

After servicing the cause of a TLB Invalid exception, the TLB entry is located with TLBP (TLB Probe), and replaced by an entry with that entry's *Valid* bit set.

TLB Modified Exception

Cause

The TLB modified exception occurs when a store operation virtual address reference to memory matches a TLB entry that is marked valid but is not dirty and therefore is not writable. This exception is not maskable.

Processing

The common exception vector is used for this exception, and the *Mod* code in the *Cause* register is set.

When this exception occurs, the *BadVAddr*, *Context*, *XContext* and *EntryHi* registers contain the virtual address that failed address translation. The *EntryHi* register also contains the ASID from which the translation fault occurred. The contents of the *EntryLo* register are undefined.

The *EPC* register contains the address of the instruction that caused the exception unless that instruction is in a branch delay slot, in which case the *EPC* register contains the address of the preceding branch instruction and the *BD* bit of the *Cause* register is set.

Servicing

The kernel uses the failed virtual address or virtual page number to identify the corresponding access control information. The page identified may or may not permit write accesses; if writes are not permitted, a write protection violation occurs.

If write accesses are permitted, the page frame is marked dirty/writable by the kernel in its own data structures. The TLBP instruction places the index of the TLB entry that must be altered into the *Index* register. The *EntryLo* register is loaded with a word containing the physical page frame and access control bits (with the *D* bit set), and the *EntryHi* and *EntryLo* registers are written into the TLB.

Cache Error Exception

The Cache Error exception is described in Chapter 9, *Cache Error Exception*.

Virtual Coherency Exception

The Virtual Coherency exception is not implemented in the R10000 processor, since the virtual coherency condition is handled in hardware.

In the R4400 processor, a Virtual Coherency exception occurs when a primary cache miss hits in the secondary cache but **VA[14:12]** are not the same as the *PIdx* field of the secondary cache tag, and the cache algorithm specifies that the page is cached. When such a situation is detected in the R10000 processor, the primary cache lines at the old virtual index are invalidated and the *PIdx* field of the secondary cache is written with the new virtual index.

Bus Error Exception

Cause

A Bus Error exception occurs when a processor block read, upgrade, or double/single/partial-word read request receives an external ERR completion response, or a processor double/single/partial-word read request receives an external ACK completion response where the associated external double/single/partial-word data response contains an uncorrectable error. This exception is not maskable.

Processing

The common interrupt vector is used for a Bus Error exception. The *IBE* or *DBE* code in the *ExcCode* field of the *Cause* register is set, signifying whether the instruction (as indicated by the *EPC* register and *BD* bit in the *Cause* register) caused the exception by an instruction reference, load operation, or store operation.

The *EPC* register contains the address of the instruction that caused the exception, unless it is in a branch delay slot, in which case the *EPC* register contains the address of the preceding branch instruction and the *BD* bit of the *Cause* register is set.

Servicing

The physical address at which the fault occurred can be computed from information available in the CP0 registers.

- If the *IBE* code in the *Cause* register is set (indicating an instruction fetch reference), the instruction that caused the exception is located at the virtual address contained in the *EPC* register (or 4+ the contents of the *EPC* register if the *BD* bit of the *Cause* register is set).
- If the *DBE* code is set (indicating a load or store reference), the instruction that caused the exception is located at the virtual address contained in the *EPC* register (or 4+ the contents of the *EPC* register if the *BD* bit of the *Cause* register is set).

The virtual address of the load and store reference can then be obtained by interpreting the instruction. The physical address can be obtained by using the TLBP instruction and reading the *EntryLo* registers to compute the physical page number. The process executing at the time of this exception is handed a UNIX SIGBUS (bus error) signal, which is usually fatal.

Integer Overflow Exception

Cause

An Integer Overflow exception occurs when an ADD, ADDI, SUB, DADD, DADDI or DSUB instruction results in a 2's complement overflow. This exception is not maskable.

Processing

The common exception vector is used for this exception, and the *OV* code in the *Cause* register is set.

The *EPC* register contains the address of the instruction that caused the exception unless the instruction is in a branch delay slot, in which case the *EPC* register contains the address of the preceding branch instruction and the *BD* bit of the *Cause* register is set.

Servicing

The process executing at the time of the exception is handed a UNIX SIGFPE/ FPE_INTOVF_TRAP (floating-point exception/integer overflow) signal. This error is usually fatal to the current process.

Trap Exception

Cause

The Trap exception occurs when a TGE, TGEU, TLT, TLTU, TEQ, TNE, TGEI, TGEUI, TLTI, TLTUI, TEQI, or TNEI instruction results in a TRUE condition. This exception is not maskable.

Processing

The common exception vector is used for this exception, and the *Tr* code in the *Cause* register is set.

The *EPC* register contains the address of the instruction causing the exception unless the instruction is in a branch delay slot, in which case the *EPC* register contains the address of the preceding branch instruction and the *BD* bit of the *Cause* register is set.

Servicing

The process executing at the time of a Trap exception is handed a UNIX SIGFPE/ FPE_INTOVF_TRAP (floating-point exception/integer overflow) signal. This error is usually fatal.

System Call Exception

Cause

A System Call exception occurs during an attempt to execute the SYSCALL instruction. This exception is not maskable.

Processing

The common exception vector is used for this exception, and the *Sys* code in the *Cause* register is set.

The *EPC* register contains the address of the SYSCALL instruction unless it is in a branch delay slot, in which case the *EPC* register contains the address of the preceding branch instruction.

If the SYSCALL instruction is in a branch delay slot, the *BD* bit of the *Status* register is set; otherwise this bit is cleared.

Servicing

When the System Call exception occurs, control is transferred to the applicable system routine. Additional distinctions can be made by analyzing the *Code* field of the SYSCALL instruction (bits 25:6), and loading the contents of the instruction whose address the EPC register contains.

To resume execution, the *EPC* register must be altered so that the SYSCALL instruction does not re-execute; this is accomplished by adding a value of 4 to the *EPC* register (*EPC* register + 4) before returning.

If a SYSCALL instruction is in a branch delay slot, a more complicated algorithm, beyond the scope of this description, may be required.

Breakpoint Exception

Cause

A Breakpoint exception occurs when an attempt is made to execute the BREAK instruction. This exception is not maskable.

Processing

The common exception vector is used for this exception, and the *BP* code in the *Cause* register is set.

The *EPC* register contains the address of the BREAK instruction unless it is in a branch delay slot, in which case the *EPC* register contains the address of the preceding branch instruction.

If the BREAK instruction is in a branch delay slot, the *BD* bit of the *Status* register is set, otherwise the bit is cleared.

Servicing

When the Breakpoint exception occurs, control is transferred to the applicable system routine. Additional distinctions can be made by analyzing the *Code* field of the BREAK instruction (bits 25:6), and loading the contents of the instruction whose address the EPC register contains. A value of 4 must be added to the contents of the EPC register (EPC register + 4) to locate the instruction if it resides in a branch delay slot.

To resume execution, the EPC register must be altered so that the BREAK instruction does not re-execute; this is accomplished by adding a value of 4 to the EPC register (EPC register + 4) before returning.

If a BREAK instruction is in a branch delay slot, interpretation of the branch instruction is required to resume execution.

Reserved Instruction Exception

Cause

The Reserved Instruction exception occurs when one of the following conditions occurs:

- an attempt is made to execute an instruction with an undefined major opcode (bits 31:26)
- an attempt is made to execute a SPECIAL instruction with an undefined minor opcode (bits 5:0)
- an attempt is made to execute a REGIMM instruction with an undefined minor opcode (bits 20:16)
- an attempt is made to execute 64-bit operations in 32-bit mode when in User or Supervisor modes
- an attempt is made to execute a COP1X when the MIPS IV ISA is not enabled

64-bit operations are always valid in Kernel mode regardless of the value of the *KX* bit in the *Status* register.

This exception is not maskable.

Processing

The common exception vector is used for this exception, and the *RI* code in the *Cause* register is set.

The *EPC* register contains the address of the reserved instruction unless it is in a branch delay slot, in which case the *EPC* register contains the address of the preceding branch instruction.

Servicing

No instructions in the MIPS ISA are currently interpreted. The process executing at the time of this exception is handed a UNIX SIGILL/ILL_RESOP_FAULT (illegal instruction/reserved operand fault) signal. This error is usually fatal.

Coprocessor Unusable Exception

Cause

The Coprocessor Unusable exception occurs when an attempt is made to execute a coprocessor instruction for either:

- a corresponding coprocessor unit (CP1 or CP2) that has not been marked usable, or
- CP0 instructions, when the unit has not been marked usable and the process executes in either User or Supervisor mode.

This exception is not maskable.

Processing

The common exception vector is used for this exception, and the *CpU* code in the *Cause* register is set. The contents of the *Coprocessor Usage Error* field of the coprocessor *Control* register indicate which of the four coprocessors was referenced. The *EPC* register contains the address of the unusable coprocessor instruction unless it is in a branch delay slot, in which case the *EPC* register contains the address of the preceding branch instruction.

Servicing

The coprocessor unit to which an attempted reference was made is identified by the Coprocessor Usage Error field, which results in one of the following situations:

- If the process is entitled access to the coprocessor, the coprocessor is marked usable and the corresponding user state is restored to the coprocessor.
- If the process is entitled access to the coprocessor, but the coprocessor does not exist or has failed, interpretation of the coprocessor instruction is possible.
- If the *BD* bit is set in the *Cause* register, the branch instruction must be interpreted; then the coprocessor instruction can be emulated and execution resumed with the *EPC* register advanced past the coprocessor instruction.
- If the process is not entitled access to the coprocessor, the process executing at the time is handed a UNIX SIGILL/ILL_PRIVIN_FAULT (illegal instruction/privileged instruction fault) signal. This error is usually fatal.

Floating-Point Exception

Cause

The Floating-Point exception is used by the floating-point coprocessor. This exception is not maskable.

Processing

The common exception vector is used for this exception, and the *FPE* code in the *Cause* register is set.

The contents of the *Floating-Point Control/Status* register indicate the cause of this exception.

Servicing

This exception is cleared by clearing the appropriate bit in the *Floating-Point Control/Status* register.

Watch Exception

Cause

A Watch exception occurs when a load or store instruction references the physical address specified in the *WatchLo*/*WatchHi* System Control Coprocessor (CP0) registers. The *WatchLo* register specifies whether a load or store initiated this exception.

The CACHE instruction never causes a Watch exception.

The Watch exception is postponed if either the *EXL* or *ERL* bit is set in the *Status* register. If either bit is set, the instruction referencing the *WatchLo*/*WatchHi* address is executed and the exception is delayed until the delay condition is cleared; that is, until *ERL* and *EXL* both are cleared (set to 0). The *EPC* contains the address of the next unexecuted instruction.

A delayed Watch exception is cleared by system reset or by writing a value to the *WatchLo* register.†

Watch is maskable by setting the *EXL* or *ERL* bits in the *Status* register.

Processing

The common exception vector is used for this exception, and the *Watch* code in the *Cause* register is set.

Servicing

The Watch exception is a debugging aid; typically the exception handler transfers control to a debugger, allowing the user to examine the situation.

To continue program execution, the Watch exception must be disabled to execute the faulting instruction. The Watch exception must then be reenabled. The faulting instruction can be executed either by interpretation or by setting breakpoints.

† An MTC0 to the *WatchLo* register clears a delayed Watch exception.

Interrupt Exception

Cause

The Interrupt exception occurs when one of the eight interrupt conditions is asserted. The significance of these interrupts is dependent upon the specific system implementation.

Each of the eight interrupts can be masked by clearing the corresponding bit in the *Interrupt-Mask (IM)* field of the *Status* register, and all of the eight interrupts can be masked at once by clearing the *IE* bit of the *Status* register.

Processing

The common exception vector is used for this exception, and the *Int* code in the *Cause* register is set.

The *IP* field of the *Cause* register indicates current interrupt requests. It is possible that more than one of the bits can be simultaneously set (or even *no* bits may be set) if the interrupt is asserted and then deasserted before this register is read.

On Cold Reset, an R4400 processor can be configured with *IP[7]* either as a sixth external interrupt, or as an internal interrupt set when the *Count* register equals the *Compare* register. There is no such option on the R10000 processor; *IP[7]* is always an internal interrupt that is set when one of the following occurs:

- the *Count* register is equal to the *Compare* register
- either one of the two performance counters overflows

Software needs to poll each source to determine the cause of the interrupt (which could come from more than one source at a time). For instance, writing a value to the *Compare* register clears the timer interrupt but it may not clear *IP[7]* if one of the performance counters is simultaneously overflowing. Performance counter interrupts can be disabled individually without affecting the timer interrupt, but there is no way to disable the timer interrupt without disabling the performance counter interrupt.

Servicing

If the interrupt is caused by one of the two software-generated exceptions (described in Chapter 6, the section titled "Software Interrupts"), the interrupt condition is cleared by setting the corresponding *Cause* register bit, *IP[1:0],* to 0. Software interrupts are imprecise. Once the software interrupt is enabled, program execution may continue for several instructions before the exception is taken. Timer interrupts are cleared by writing to the *Compare* register. The Performance Counter interrupt is cleared by writing a 0 to bit 31, the overflow bit, of the counter.

Cold Reset and Soft Reset exceptions clear all the outstanding external interrupt requests, *IP[2]* to *IP[6]*.

If the interrupt is hardware-generated, the interrupt condition is cleared by correcting the condition causing the interrupt pin to be asserted.

17.4 MIPSIV Instructions

The system must either be in Kernel or Supervisor mode, or have set the *XX* bit of the *Status* register to a 1 in order to use the MIPS IV instruction set. In User mode, if *XX* is a 0 and an attempt is made to execute MIPS IV instructions, an exception will be taken. The type of exception that will be taken depends upon the type of instruction whose execution was attempted; a list is given in Table 17-4. Note that operating with MIPS IV instructions does not require that MIPS III instruction set or 64-bit addressing is enabled.

MIPS IV instructions that use or modify the floating-point registers (CP1 state) are also affected by the *CU1* bit of the CP0 Status register. If *CU1* is not set, a Coprocessor Unusable exception may be signaled.

The Reserved Instruction (RI), Coprocessor Unusable (CU), and Unimplemented Operation (UO) exceptions for MIPS IV instructions are listed in the Table 17-4 below.

Table 17-4 MIPS IV Instruction Exceptions

Exceptions	Instructions	CU1	MIPS4
RI	CPU (undefined)	-	-
RI	MOVN,Z	-	0
RI	MOVT,F	-	0
CU		0	1
RI	PREF	-	0
CU	COP1 (all instructions)	0	-
UO	(undefined)	1	-
RI	BC (cc>0)	1	0
UO	C (cc>0)	1	0
UO	MOVN,Z,T,F	1	0
UO	RECIP, RSQRT	1	0
RI	COP1X (all instructions)	-	0
CU	(all instructions)	0	1
RI	(undefined)	1	1

17.5 COP0 Instructions

Execution of an RFE instruction causes a Reserved Instruction exception in the R10000 processor.

The execution of undefined COP0 functions is undefined in the R10000 processor.

17.6 COP1 Instructions

The R10000 and R4400 processors do not generate the same exceptions for undefined COP1 instructions. In the R4400 processor, undefined opcodes or formats in the *sub* field take an Unimplemented Operation exceptions. In the R10000 processor, undefined opcodes (bits 25:24 are 0 or 1) take Reserved Instruction exceptions and undefined formats (bits 25:24 are 2 or 3) take Unimplemented Operation exceptions.

In MIPS II on an R4400 processor, the execution of DMTC1, DMFC1, and L format take Unimplemented Operation exceptions. In MIPS II on the R10000 processor, the execution of DMTC1 and DMFC1 take Reserved Instruction exceptions

The attempted execution of the L format takes an Unimplemented Operation exception when the MIPS III mode is not enabled.

A CTC1 instruction that sets both *Cause* and *Enable* bits also forces an immediate floating-point exception; the *EPC* register points to the offending CTC1 instruction.

17.7 COP2 Instructions

If the *CU2* bit of the CP0 *Status* register is not set during an attempted execution of such Coprocessor 2 instructions as COP2, LWC2, SWC2, LDC2, and SDC2, the system takes a Coprocessor Unusable exception.

In the R4400 processor, if the *CU2* bit is set, COP2 instructions are handled as NOPs; the operations of Coprocessor 2 load/store instructions are undefined. In the R10000 processor, an execution of a Coprocessor 2 instruction takes a Reserved Instruction exception when *CU2* bit is set.