CPU Exception Processing

5

This chapter describes the CPU exception processing, including an explanation of exception processing, followed by the format and use of each CPU exception register.

The chapter concludes with a description of each exception's cause, together with the manner in which the CPU processes and services these exceptions. For information about Floating-Point Unit exceptions, see Chapter 7.

5.1 How Exception Processing Works

The processor receives exceptions from a number of sources, including translation lookaside buffer (TLB) misses, arithmetic overflows, I/O interrupts, and system calls. When the CPU detects one of these exceptions, the normal sequence of instruction execution is suspended and the processor enters Kernel mode (see Chapter 4 for a description of system operating modes).

The processor then disables interrupts and forces execution of a software exception processor (called a *handler*) located at a fixed address. The handler saves the context of the processor, including the contents of the program counter, the current operating mode (User or Supervisor), and the status of the interrupts (enabled or disabled). This context is saved so it can be restored when the exception has been serviced.

When an exception occurs, the CPU loads the *Exception Program Counter* (*EPC*) register with a location where execution can restart after the exception has been serviced. The restart location in the *EPC* register is the address of the instruction that caused the exception or, if the instruction was executing in a branch delay slot, the address of the branch instruction immediately preceding the delay slot.

The registers described later in the chapter assist in this exception processing by retaining address, cause and status information.

For a description of the exception handling process, see the description of the individual exception contained in this chapter, or the flowcharts at the end of this chapter.

5.2 Exception Processing Registers

This section describes the CP0 registers that are used in exception processing. Table 5-1 lists these registers, along with their number—each register has a unique identification number that is referred to as its *register number*. For instance, the *ECC* register is register number 26. The remaining CP0 registers are used in memory management, as described in Chapter 4.

Software examines the CP0 registers during exception processing to determine the cause of the exception and the state of the CPU at the time the exception occurred. The registers in Table 5-1 are used in exception processing, and are described in the sections that follow.

Table 5-1 CP0 Exception Processing Registers

Register Name	Reg. No.
Context	4
BadVAddr (Bad Virtual Address)	8
Count	9
Compare register	11
Status	12
Cause	13
EPC (Exception Program Counter)	14
WatchLo (Memory Reference Trap Address Low)	18
WatchHi (Memory Reference Trap Address High)	19
XContext	20
ECC	26
CacheErr (Cache Error and Status)	27
ErrorEPC (Error Exception Program Counter)	30

CPU general registers are interlocked and the result of an instruction can normally be used by the next instruction; if the result is not available right away, the processor stalls until it is available. CP0 registers and the TLB are not interlocked, however; there may be some delay before a value written by one instruction is available to following instructions. For more information please see Appendix F.

Context Register (4)

The *Context* register is a read/write register containing the pointer to an entry in the page table entry (PTE) array; this array is an operating system data structure that stores virtual-to-physical address translations. When there is a TLB miss, the CPU loads the TLB with the missing translation from the PTE array. Normally, the operating system uses the *Context* register to address the current page map which resides in the kernel-mapped segment, *kseg3*. The *Context* register duplicates some of the information provided in the *BadVAddr* register, but the information is arranged in a form that is more useful for a software TLB exception handler. Figure 5-1 shows the format of the *Context* register; Table 5-2 describes the *Context* register fields.

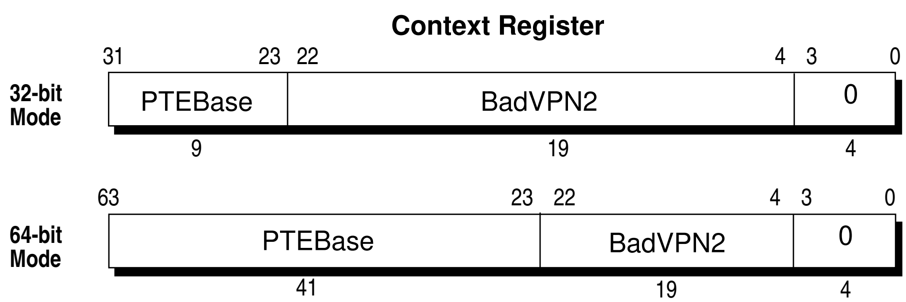


Figure 5-1 Context Register Format

Table 5-2 Context Register Fields

Field	Description
BadVPN2	This field is written by hardware on a miss. It contains the virtual page number (VPN) of the most recent virtual address that did not have a valid translation.
PTEBase	This field is a read/write field for use by the operating system. It is normally written with a value that allows the operating system to use the *Context* register as a pointer into the current PTE array in memory.

The 19-bit *BadVPN2* field contains bits 31:13 of the virtual address that caused the TLB miss; bit 12 is excluded because a single TLB entry maps to an even-odd page pair. For a 4-Kbyte page size, this format can directly address the pair-table of 8-byte PTEs. For other page and PTE sizes, shifting and masking this value produces the appropriate address.

Bad Virtual Address Register (BadVAddr) (8)

The Bad Virtual Address register (*BadVAddr*) is a read-only register that displays the most recent virtual address that caused one of the following exceptions: TLB Invalid, TLB Modified, TLB Refill, Virtual Coherency Data Access, or Virtual Coherency Instruction Fetch.

Figure 5-2 shows the format of the *BadVAddr* register.

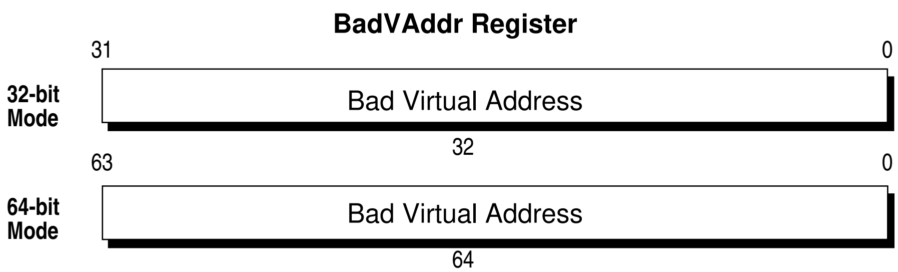


Figure 5-2 BadVAddr Register Format

Note: The *BadVAddr* register does not save any information for bus errors, since bus errors are not addressing errors.

Count Register (9)

The *Count* register acts as a timer, incrementing at a constant rate—half the maximum instruction issue rate—whether or not an instruction is executed, retired, or any forward progress is made through the pipeline.

This register can be read or written. It can be written for diagnostic purposes or system initialization; for example, to synchronize processors.

Figure 5-3 shows the format of the *Count* register.

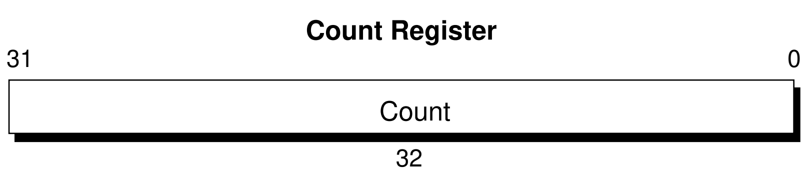


Figure 5-3 Count Register Format

Compare Register (11)

The *Compare* register acts as a timer (see also the *Count* register); it maintains a stable value that does not change on its own.

When the value of the *Count* register equals the value of the *Compare* register, interrupt bit *IP(7)* in the *Cause* register is set. This causes an interrupt as soon as the interrupt is enabled.

Writing a value to the *Compare* register, as a side effect, clears the timer interrupt.

For diagnostic purposes, the *Compare* register is a read/write register. In normal use however, the *Compare* register is write-only. Figure 5-4 shows the format of the *Compare* register.

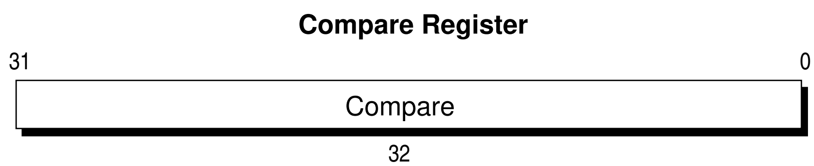


Figure 5-4 Compare Register Format

Status Register (12)

The *Status* register (SR) is a read/write register that contains the operating mode, interrupt enabling, and the diagnostic states of the processor. The following list describes the more important *Status* register fields; Figures 5-5 and 5-6 show the format of the entire register, including descriptions of the fields. Some of the important fields include:

- The 8-bit *Interrupt Mask* (*IM*) field controls the enabling of eight interrupt conditions. Interrupts must be enabled before they can be asserted, and the corresponding bits are set in both the *Interrupt Mask* field of the *Status* register and the *Interrupt Pending* field of the *Cause* register. For more information, refer to the *Interrupt Pending* (*IP*) field of the *Cause* register and Chapter 15, which describes the interrupts.
- The 4-bit *Coprocessor Usability* (*CU*) field controls the usability of 4 possible coprocessors. Regardless of the *CU0* bit setting, CP0 is always usable in Kernel mode.
- The 9-bit *Diagnostic Status* (*DS*) field is used for self-testing, and checks the cache and virtual memory system.
- The *Reverse-Endian (RE)* bit, bit 25, reverses the endianness of the machine. The processor can be configured as either little-endian or big-endian at system reset; reverse-endian selection is used in Kernel and Supervisor modes, and in the User mode when the *RE* bit is 0. Setting the *RE* bit to 1 inverts the User mode endianness.

Status Register Format

Figure 5-5 shows the format of the *Status* register. Table 5-3 describes the *Status* register fields. Figure 5-6 and Table 5-4 provide additional information on the *Diagnostic Status* (*DS*) field. All bits in the *DS* field except *TS* are readable and writable.

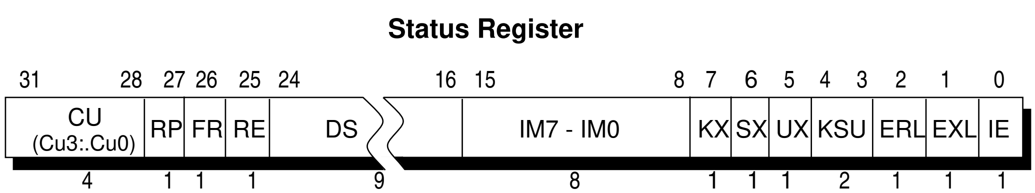


Figure 5-5 Status Register

Table 5-3 Status Register Fields

Field	Description
CU	Controls the usability of each of the four coprocessor unit numbers. CP0 is always usable when in Kernel mode, regardless of the setting of the CU_0 bit. 1 → usable 0 → unusable
RP	Enables reduced-power operation by reducing the internal clock frequency. The clock divisor is programmable at boot time. 0 → full speed 1→ reduced clock
FR	Enables additional floating-point registers 0 → 16 registers 1 → 32 registers
RE	*Reverse-Endian* bit, valid in User mode.
DS	*Diagnostic Status* field (see Figure 5-6).
IM	*Interrupt Mask*: controls the enabling of each of the external, internal, and software interrupts. An interrupt is taken if interrupts are enabled, and the corresponding bits are set in both the *Interrupt Mask* field of the *Status* register and the *Interrupt Pending* field of the *Cause* register. 0 → disabled 1→ enabled
KX	Enables 64-bit addressing in Kernel mode. The extended-addressing TLB refill exception is used for TLB misses on kernel addresses. 0 → 32–bit 1 → 64–bit
SX	Enables 64-bit addressing and operations in Supervisor mode. The extended-addressing TLB refill exception is used for TLB misses on supervisor addresses. 0 → 32–bit 1 → 64–bit

Table 5-3 (cont.) Status Register Fields

Field	Description
UX	Enables 64-bit addressing and operations in User mode. The extended-addressing TLB refill exception is used for TLB misses on user addresses. 0 → 32–bit 1 → 64–bit
KSU	Mode bits 10_2 → User 01_2 → Supervisor 00_2 → Kernel
ERL	Error Level; set by the processor when Reset, Soft Reset, NMI, or Cache Error exception are taken. 0 → normal 1 → error
EXL	Exception Level; set by the processor when any exception other than Reset, Soft Reset, NMI, or Cache Error exception are taken. 0 → normal 1 → exception
IE	Interrupt Enable 0 → disable interrupts 1 → enables interrupts

Diagnostic Status Field

24 23	22	21	20	19	18	17	16
0	BEV	TS	SR	0	CH	CE	DE
2	1	1	1	1	1	1	1

Figure 5-6 Status Register DS Field

Table 5-4 Status Register Diagnostic Status Bits

Bit	Description
BEV	Controls the location of TLB refill and general exception vectors. 0 → normal 1→ bootstrap
TS	1→ Indicates TLB shutdown has occurred (read-only).
SR	1→ Indicates a **Reset*** signal or NMI has caused a Soft Reset exception
CH	Hit (tag match and valid state) or miss indication for last CACHE Hit Invalidate, Hit Write Back Invalidate, Hit Write Back, Hit Set Virtual, or Create Dirty Exclusive for a secondary cache. 0 → miss 1 → hit
CE	Contents of the ECC register set or modify the check bits of the caches when CE = 1; see description of the *ECC* register.
DE	Specifies that cache parity or ECC errors cannot cause exceptions. 0 → parity/ECC remain enabled 1 → disables parity/ECC
0	Reserved. Must be written as zeroes, and returns zeroes when read.

Status Register Modes and Access States

Fields of the *Status* register set the modes and access states described in the sections that follow.

Interrupt Enable: Interrupts are enabled when all of the following conditions are true:

- $IE = 1$
- $EXL = 0$
- $ERL = 0$

If these conditions are met, the settings of the *IM* bits enable the interrupt.

Operating Modes: The following CPU *Status* register bit settings are required for User, Kernel, and Supervisor modes (see Chapter 4 for more information about operating modes).

- The processor is in User mode when $KSU = 10_2$, $EXL = 0$, and $ERL = 0$.
- The processor is in Supervisor mode when $KSU = 01_2$, $EXL = 0$, and $ERL = 0$.
- The processor is in Kernel mode when $KSU = 00_2$, or $EXL = 1$, or $ERL = 1$.

32- and 64-bit Modes: The following CPU *Status* register bit settings select 32- or 64-bit operation for User, Kernel, and Supervisor operating modes. Enabling 64-bit operation permits the execution of 64-bit opcodes and translation of 64-bit addresses. 64-bit operation for User, Kernel and Supervisor modes can be set independently.

- 64-bit addressing for Kernel mode is enabled when $KX = 1$. 64-bit operations are always valid in Kernel mode.
- 64-bit addressing and operations are enabled for Supervisor mode when $SX = 1$.
- 64-bit addressing and operations are enabled for User mode when $UX = 1$.

Kernel Address Space Accesses: Access to the kernel address space is allowed when the processor is in Kernel mode.

Supervisor Address Space Accesses: Access to the supervisor address space is allowed when the processor is in Kernel or Supervisor mode, as described above in the section above titled, Operating Modes.

User Address Space Accesses: Access to the user address space is allowed in any of the three operating modes.

Status Register Reset

The contents of the *Status* register are undefined at reset, except for the following bits in the *Diagnostic Status* field:

- *TS* = 0
- *ERL* and *BEV* = 1

The *SR* bit distinguishes between the Reset exception and the Soft Reset exception (caused either by **Reset*** or Nonmaskable Interrupt [NMI]).

Cause Register (13)

The 32-bit read/write *Cause* register describes the cause of the most recent exception.

Figure 5-7 shows the fields of this register; Table 5-5 describes the *Cause* register fields. A 5-bit exception code (*ExcCode*) indicates one of the causes, as listed in Table 5-6.

All bits in the *Cause* register, with the exception of the *IP(1:0)* bits, are read-only; *IP(1:0)* are used for software interrupts.

Table 5-5 Cause Register Fields

Field	Description
BD	Indicates whether the last exception taken occurred in a branch delay slot. 1 → delay slot 0 → normal
CE	Coprocessor unit number referenced when a Coprocessor Unusable exception is taken.
IP	Indicates an interrupt is pending. 1 → interrupt pending 0 → no interrupt
ExcCode	Exception code field (see Table 5-6)
0	Reserved. Must be written as zeroes, and returns zeroes when read.

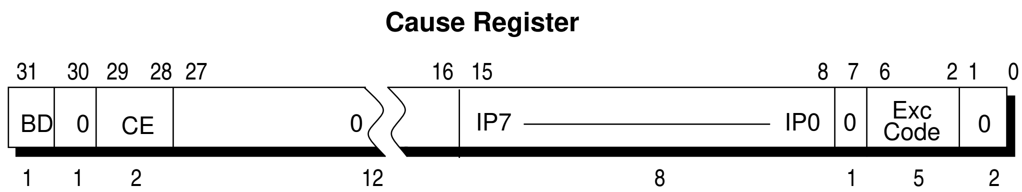


Figure 5-7 Cause Register Format

Table 5-6 Cause Register ExcCode Field

Exception Code Value	Mnemonic	Description
0	Int	Interrupt
1	Mod	TLB modification exception
2	TLBL	TLB exception (load or instruction fetch)
3	TLBS	TLB exception (store)
4	AdEL	Address error exception (load or instruction fetch)
5	AdES	Address error exception (store)
6	IBE	Bus error exception (instruction fetch)
7	DBE	Bus error exception (data reference: load or store)
8	Sys	Syscall exception
9	Bp	Breakpoint exception
10	RI	Reserved instruction exception
11	CpU	Coprocessor Unusable exception
12	Ov	Arithmetic Overflow exception
13	Tr	Trap exception
14	VCEI	Virtual Coherency Exception instruction
15	FPE	Floating-Point exception
16–22	–	Reserved
23	WATCH	Reference to *WatchHi*/*WatchLo* address
24–30	–	Reserved
31	VCED	Virtual Coherency Exception data

Exception Program Counter (EPC) Register (14)

The Exception Program Counter (*EPC*) is a read/write register that contains the address at which processing resumes after an exception has been serviced.

For synchronous exceptions, the *EPC* register contains either:

- the virtual address of the instruction that was the direct cause of the exception, or
- the virtual address of the immediately preceding branch or jump instruction (when the instruction is in a branch delay slot, and the *Branch Delay* bit in the *Cause* register is set).

The processor does not write to the *EPC* register when the *EXL* bit in the *Status* register is set to a 1.

Figure 5-8 shows the format of the *EPC* register.

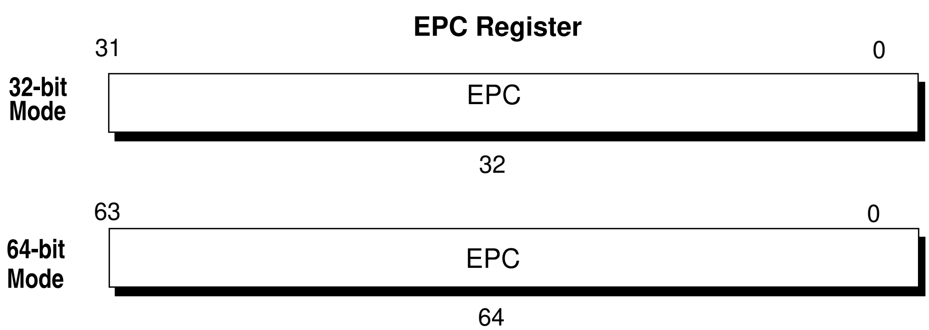


Figure 5-8 EPC Register Format

WatchLo (18) and WatchHi (19) Registers

R4000 processors provide a debugging feature to detect references to a selected physical address; load and store operations to the location specified by the *WatchLo* and *WatchHi* registers cause a Watch exception (described later in this chapter).

Figure 5-9 shows the format of the *WatchLo* and *WatchHi* registers; Table 5-7 describes the *WatchLo* and *WatchHi* register fields.

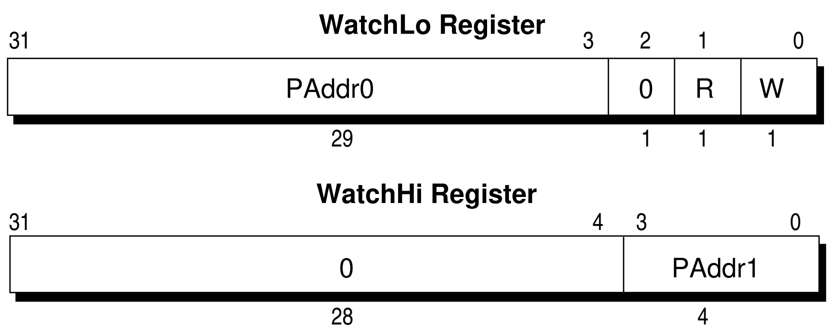


Figure 5-9 WatchLo and WatchHi Register Formats

Table 5-7 WatchHi and WatchLo Register Fields

Field	Description
PAddr1	Bits 35:32 of the physical address
PAddr0	Bits 31:3 of the physical address
R	Trap on load references if set to 1
W	Trap on store references if set to 1
0	Reserved. Must be written as zeroes, and returns zeroes when read.

XContext Register (20)

The read/write *XContext* register contains a pointer to an entry in the page table entry (PTE) array, an operating system data structure that stores virtual-to-physical address translations. When there is a TLB miss, the operating system software loads the TLB with the missing translation from the PTE array. The *XContext* register duplicates some of the information provided in the *BadVAddr* register, and puts it in a form useful for a software TLB exception handler. The *XContext* register is for use with the XTLB refill handler, which loads TLB entries for references to a 64-bit address space, and is included solely for operating system use. The operating system sets the PTE base field in the register, as needed. Normally, the operating system uses the *Context* register to address the current page map, which resides in the kernel-mapped segment *kseg3*. Figure 5-10 shows the format of the *XContext* register; Table 5-8 describes the *XContext* register fields.

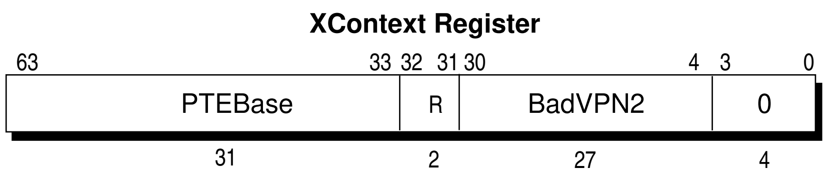


Figure 5-10 XContext Register Format

The 27-bit *BadVPN2* field has bits 39:13 of the virtual address that caused the TLB miss; bit 12 is excluded because a single TLB entry maps to an even-odd page pair. For a 4-Kbyte page size, this format may be used directly to address the pair-table of 8-byte PTEs. For other page and PTE sizes, shifting and masking this value produces the appropriate address.

Table 5-8 XContext Register Fields

Field	Description
BadVPN2	The *Bad Virtual Page Number*/2 field is written by hardware on a miss. It contains the VPN of the most recent invalidly translated virtual address.
R	The *Region* field contains bits 63:62 of the virtual address. 00_2 = user 01_2 = supervisor 11_2 = kernel.
PTEBase	The *Page Table Entry Base* read/write field is normally written with a value that allows the operating system to use the *Context* register as a pointer into the current PTE array in memory.

Error Checking and Correcting (ECC) Register (26)

The 8-bit *Error Checking and Correcting* (*ECC*) register reads or writes either secondary-cache data ECC bits or primary-cache data parity bits for cache initialization, cache diagnostics, or cache error processing. (Tag ECC and parity are loaded from and stored to the *TagLo* register.)

The *ECC* register is loaded by the Index Load Tag CACHE operation. Content of the ECC register is:

- written into the primary data cache on store instructions (instead of the computed parity) when the *CE* bit of the *Status* register is set
- substituted for the computed instruction parity for the CACHE operation Fill
- XORed into the secondary cache computed ECC for the following primary data cache CACHE operations: Index Write Back Invalidate, Hit Write Back, and Hit Write Back Invalidate.

Figure 5-11 shows the format of the *ECC* register; Table 5-9 describes the register fields.

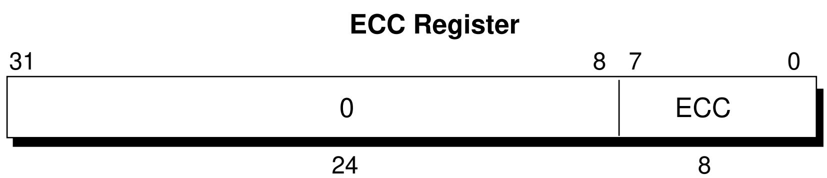


Figure 5-11 ECC Register Format

Table 5-9 ECC Register Fields

Field	Description
ECC	An 8-bit field specifying the ECC bits read from or written to a secondary cache, or the even byte parity bits to be read from or written to a primary cache.
0	Reserved. Must be written as zeroes, and returns zeroes when read.

Cache Error (CacheErr) Register (27)

The 32-bit read-only *CacheErr* register processes ECC errors in the secondary cache and parity errors in the primary cache. Parity errors cannot be corrected.

All single- and double-bit ECC errors in the secondary cache tag and data are detected; single-bit errors in the cache tag are automatically corrected. Single-bit ECC errors in the secondary cache data are not automatically corrected.

The *CacheErr* register holds cache index and status bits that indicate the source and nature of the error; it is loaded when a Cache Error exception is asserted.

Figure 5-12 shows the format of the *CacheErr* register and Table 5-10 describes the *CacheErr* register fields.

CacheErr Register

31	30	29	28	27	26	25	24	23	22	21 … 2	2 … 0
ER	EC	ED	ET	ES	EE	EB	EI	EW	0	SIdx	PIDx
1	1	1	1	1	1	1	1	1	1	19	3

Figure 5-12 CacheErr Register Format

Table 5-10 CacheErr Register Fields

Field	Description
ER	Type of reference 0 → instruction 1 → data
EC	Cache level of the error 0 → primary 1 → secondary
ED	Indicates if a data field error occurred 0 → no error 1 → error
ET	Indicates if a tag field error occurred 0 → no error 1 → error

Table 5-10 (cont.) CacheErr Register Fields

Field	Description
ES	Indicates the error occurred while accessing primary or secondary cache in response to an external request. 0 → internal reference 1 → external reference
EE	This bit is set if the error occurred on the SysAD bus.
EB	This bit is set if a data error occurred in addition to the instruction error (indicated by the remainder of the bits). If so, this requires flushing the data cache after fixing the instruction error.
EI	This bit is set on a secondary data cache ECC error while refilling the primary cache on a store miss. The ECC handler must first do an Index Store Tag to invalidate the incorrect data from the primary data cache.
EW	This bit is only available on the R4400 processor. It is set on an multiprocessor cache error when the *CacheErr* register is already holding the values of a previous cache error. This bit could be set by the processor from the time the *CacheErr* register is loaded due to an error until the time that an ERET instruction is executed. Once the *EW* bit is set, it can only be cleared by a reset. The following errors set the *EW* bit: • Secondary cache tag errors arising from an external request (multibit errors only) • Secondary cache data errors arising from an external update • Primary cache tag errors arising from an external request
SIdx	Bits pAddr(21:3) of the reference that encountered the error (which is not necessarily the same as the address of the doubleword in error, but is sufficient to locate that doubleword in the secondary cache).
PIdx	Bits vAddr(14:12) of the doubleword in error (used with *SIdx* to construct a virtual index for the primary caches).
0	Reserved. Must be written as zeroes, and returns zeroes when read.

Error Exception Program Counter (Error EPC) Register (30)

The *ErrorEPC* register is similar to the *EPC* register, except that *ErrorEPC* is used on ECC and parity error exceptions. It is also used to store the program counter (PC) on Reset, Soft Reset, and nonmaskable interrupt (NMI) exceptions.

The read/write *ErrorEPC* register contains the virtual address at which instruction processing can resume after servicing an error. This address can be:

- the virtual address of the instruction that caused the exception
- the virtual address of the immediately preceding branch or jump instruction, when this address is in a branch delay slot.

There is no branch delay slot indication for the *ErrorEPC* register.

Figure 5-13 shows the format of the *ErrorEPC* register.

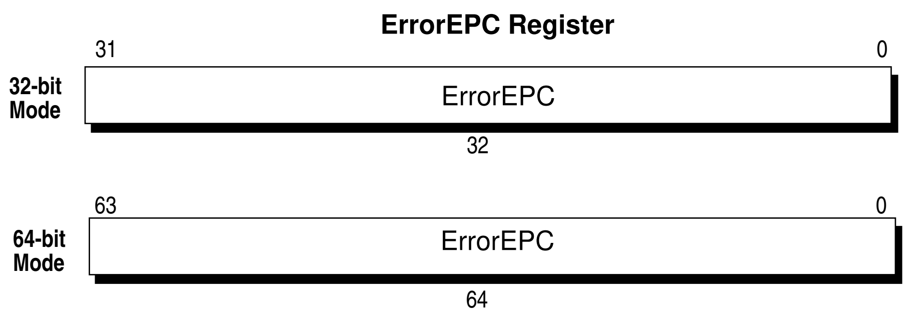


Figure 5-13 ErrorEPC Register Format

5.3 Processor Exceptions

This section describes the processor exceptions—it describes the cause of each exception, its processing by the hardware, and servicing by a handler (software). The types of exception, with exception processing operations, are described in the next section.

Exception Types

This section gives sample exception handler operations for the following exception types:

- reset
- soft reset
- nonmaskable interrupt (NMI)
- cache error
- remaining processor exceptions

When the *EXL* bit in the *Status* register is 0, either User, Supervisor, or Kernel operating mode is specified by the *KSU* bits in the *Status* register. When the *EXL* bit is a 1, the processor is in Kernel mode.

When the processor takes an exception, the *EXL* bit is set to 1, which means the system is in Kernel mode. After saving the appropriate state, the exception handler typically changes *KSU* 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).

In the following sections, sample hardware processes for various exceptions are shown, together with the servicing required by the handler (software).

Reset Exception Process

Figure 5-14 shows the Reset exception process.

```
T:  undefined
    Random ← TLBENTRIES–1
    Wired ← 0
    Config ← CM || EC || EP || SB || SS || SW || EW || SC || SM || BE || EM || EB || 0 || IC
                || DC || undefined^6
    ErrorEPC ← RestartPC /* If the instruction is in a branch delay slot, RestartPC  */
                              /* holds the value of PC-4, otherwise RestartPC = PC */
    If R4400 then
              CacheErr ← undefined^8  || 0 || undefined^23  /* Set EW bit to 0 */
    endif
    SR ← SR_31:23 || 1 || 0 || 0 || SR_19:3 || 1 || SR_1:0
    PC ← 0xFFFF FFFF BFC0 0000
```

Figure 5-14 Reset Exception Processing

Cache Error Exception Process

Figure 5-15 shows the Cache Error exception process.

```
T:  ErrorEPC ← RestartPC  /* If the instruction is in a branch delay slot, RestartPC  */
                              /* holds the value of PC-4, otherwise RestartPC = PC */
    if R4000 then
              CacheErr ← ER || EC || ED || ET || ES || EE || EB || EI || 0^2 || SIdx || PIdx
    else      /* R4400 */
              CacheErr ← ER || EC || ED || ET || ES || EE || EB || EI || EW || 0 || SIdx || PIdx
    endif
    SR ← SR_31:3 || 1 ||SR_1:0
    if SR_22 = 1 then
       PC ← 0xFFFF FFFF BFC0 0200 + 0x100
    else
       PC ← 0xFFFF FFFF A000 0000 + 0x100
    endif
```

Figure 5-15 Cache Error Exception Processing

Soft Reset and NMI Exception Process

Figure 5-16 shows the Soft Reset and NMI exception process.

```
T:  ErrorEPC ← RestartPC   /* If the instruction is in a branch delay slot, RestartPC */
                           /* holds the value of PC-4, otherwise RestartPC = PC */
    SR ← SR31:23 || 1 || 0 || 1 || SR19:3 || 1 || SR1:0
    If R4400 then
              CacheErr ← CacheErr31:24 || 0 || CacheErr22:0
    endif
    PC ← 0xFFFF FFFF BFC0 0000
```

Figure 5-16 Soft Reset and NMI Exception Processing

General Exception Process

Figure 5-17 shows the process used for exceptions other than Reset, Soft Reset, NMI, and Cache Error.

```
T:  if SR1 = 0 then        /* if not EXL */
              EPC ← RestartPC          /* If the instruction is in a branch delay slot, */
                                       /* RestartPC holds the value of PC-4, */
                                       /* otherwise RestartPC = PC */
              Cause ← BD || 0 || CE || 0^12 || Cause15:8 || 0 || ExcCode || 0^2
              if TLBrefill then          vector ← 0x000
              elseif XTLBrefill then     vector ← 0x080
              else   /* not a miss */    vector ← 0x180
    else
              Cause ← Cause31 || 0 || CE || 0^12 || Cause15:8 || 0 || ExcCode || 0^2
              vector ← 0x180
    endif
    SR ← SR31:2 || 1 || SR0   /* EXL */
    if SR22 = 1 then
       PC ← 0xFFFF FFFF BFC0 0200 + vector
    else
       PC ← 0xFFFF FFFF 8000 0000 + vector
    endif
```

Figure 5-17 General Exception Processing (Except Reset, Soft Reset, NMI, and Cache Error)

Exception Vector Locations

The Reset, Soft Reset, and NMI exceptions are always vectored to the dedicated 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.

Table 5-11 shows the 64-bit-mode vector base address for all exceptions; the 32-bit mode address is the low-order 32 bits (for instance, the base address for NMI in 32-bit mode is 0xBFC0 0000).

Table 5-12 shows the vector offset added to the base address to create the exception address.

Table 5-11 Exception Vector Base Addresses

Exception	BEV	
	0	**1**
Cache Error	0xFFFF FFFF A000 0000	0xFFFF FFFF BFC0 0200
Others	0xFFFF FFFF 8000 0000	0xFFFF FFFF BFC0 0200
Reset, NMI, Soft Reset	0xFFFF FFFF BFC0 0000	

Table 5-12 Exception Vector Offsets

Exception	R4000 Processor Vector Offset
TLB refill, EXL = 0	0x000
XTLB refill, EXL = 0 (X = 64-bit TLB)	0x080
Cache Error	0x100
Others	0x180
Reset, Soft Reset, NMI	none

Priority of Exceptions

The remainder of this chapter describes exceptions in the order of their priority shown in Table 5-13 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.

Table 5-13 Exception Priority Order

Reset *(highest priority)*
Soft Reset caused by **Reset*** signal
Nonmaskable Interrupt (NMI) (Soft Reset exception caused by NMI)
Address error — Instruction fetch
TLB refill — Instruction fetch
TLB invalid — Instruction fetch
Cache error — Instruction fetch
Virtual Coherency — 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
Cache error — Data access
Watch
Virtual Coherency — Data access
Bus error — Data access
Interrupt *(lowest priority)*

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

Reset Exception

Cause

The Reset exception occurs when the **ColdReset***† signal is asserted and then deasserted. 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 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 (see Figure 5-14).
- 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 (R4400 only).

Reset exception processing is shown in Figure 5-14.

Servicing

The Reset exception is serviced by:

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

† In the following sections—indeed, throughout this book—a signal followed by an asterisk, such as **Reset***, is low active.

Soft Reset Exception

Cause

The Soft Reset exception occurs in response to either the **Reset*** input signal or a Nonmaskable Interrupt (NMI)†.

The NMI is caused either by an assertion of the **NMI*** signal or an external write to the *Int*[6]* bit of the *Interrupt* register.

This exception is not maskable.

Processing

Regardless of the cause, when this exception occurs the *SR* bit of the *Status* register is set, distinguishing this exception from a Reset exception.

The processor does not indicate any distinction between an exception caused by the **Reset*** signal or the **NMI*** signal.

- An exception caused by an **NMI** can only be taken if the processor is processing instructions; it is taken at the instruction boundary. It does not abort any state machines, preserving the state of the processor for diagnosis.
- An exception caused by assertion of **Reset*** performs a subset of the full reset initialization. After a processor is completely initialized by a Reset exception (caused by **ColdReset*** or **Power-On**), **Reset*** can be asserted on the processor in any state, even if the processor is no longer processing instructions. In this situation the processor does not read or set processor configuration parameters. It does, however, initialize all other processor state that requires *hardware* initialization (for instance, the state machines and registers), in order that the CPU can fetch and execute the Reset exception handler located in uncached and unmapped space. Although no other processor state is unnecessarily changed, a soft reset sequence may be forced to alter some state since the exception can be invoked arbitrarily on a cycle boundary, and abort any multicycle operation in progress. Since bus, cache, or other operations may be interrupted, portions of the cache, memory, or other processor state may be inconsistent.

† In this book, a Soft Reset exception caused by assertion of the **Reset*** signal is referred to as a "soft reset" or "warm reset." A Soft Reset exception caused by a nonmaskable interrupt (NMI) is referred to as a "nonmaskable interrupt exception."

In both the **Reset*** and NMI cases the processor jumps to the Reset exception vector located in unmapped and uncached address space, so that the cache and TLB contents need not be initialized to service this exception. Typically, the Reset exception vector is located in PROM, and system memory does not need to be initialized to handle the exception.

As previously noted, state machines interrupted by **Reset*** may cause some register contents to be inconsistent with the other processor state. Otherwise, on an exception caused by **Reset*** or NMI the contents of all registers are preserved, except for:

- *EW* bit in the *CacheErr* register, which is reset to 0 (R4400 only)
- *ErrorEPC* register, which contains the restart PC
- *ERL* bit of the *Status* register, which is set to 1
- *SR* bit of the *Status* register, which is set to 1
- *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

Soft reset exception processing is shown in Figure 5-16.

Servicing

The exception initiated by **Reset*** is intended to quickly reinitialize a previously operating processor after a fatal error such as a Master/Checker mismatch. 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.

The exceptions due to **Reset*** and NMI appear identical to software; both exceptions jump to the Reset exception vector and have the *Status* register *SR* bit set. Unless external hardware provides a way to distinguish between the two, they are serviced by saving the current user-visible processor state for diagnostic purposes and reinitializing as for the Reset exception. It is not normally possible to continue program execution after returning from this exception, since a **Reset*** signal can be accepted anytime and an NMI can occur in the midst of another error exception.

Address Error Exception

Cause

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

- 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
- reference the kernel address space from User or Supervisor mode
- reference the supervisor address space from User mode

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* 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.

Address Error exception processing is shown in Figure 5-17.

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.

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. 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.

TLB Refill exception processing is shown in Figure 5-17.

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* 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.

TLB Invalid exception processing is shown in Figure 5-17.

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.

TLB Modified exception processing is shown in Figure 5-17.

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

Cause

The Cache Error exception occurs when either a secondary cache ECC error, primary cache parity error, or SysAD bus parity/ECC error condition occurs and error detection is enabled. This exception is not maskable, but error detection can be disabled if either *ERL* or *DE* = 1 in the *Status* register.

Processing

The processor sets the *ERL* bit in the *Status* register, saves the exception restart address in the *ErrorEPC* register, records information about the error in the *CacheErr* register, and then transfers to a special vector that is always in uncached space (Tables 5-11 and 5-12). No other registers are changed. Cache Error exception processing is shown in Figure 5-15.

Servicing

Unlike other exception conditions, cache errors cannot be avoided while operating at exception level, so Cache Error exceptions must be handled from exception level. Any general register used by the handler must be saved before use and restored before return; this includes the registers available to regular exception handlers without save/restore. When *ERL*=1 in the *Status* register, the user address region becomes a 2^{31}-byte uncached space mapped directly to physical addresses, allowing the Cache Error handler to save registers to memory without using a register to construct the address. The handler can save and restore registers using operating system-reserved locations in low physical memory by using *R0* as the base register for load and store instructions. All errors should be logged. To correct single-bit ECC errors in the secondary cache, the system uses the CACHE instruction. Execution then resumes through an ERET instruction. To correct cache parity errors and non-single-bit ECC errors in unmodified cache blocks, the system uses the CACHE instruction to invalidate the cache block, overwrites the old data through a cache miss, and resumes execution with an ERET. Other errors are not correctable and are likely to be fatal to the current process. The exception handler cannot be interrupted by another Cache Error exception because error detection is disabled while *ERL* = 1, so the handler should avoid actions which might cause an unnoticed cache error. The R4400 (but not R4000) implements the *EW* bit in the *CacheErr* register to record a nonrecoverable error occurring while *ERL* = 1.

Virtual Coherency Exception

Cause

A Virtual Coherency exception occurs when all of the following conditions are true:

- a primary cache miss hits in the secondary cache
- bits 14:12 of the virtual address were not equal to the corresponding bits of the *PIdx* field of the secondary cache tag
- the cache algorithm for the page (from the *C* field in the TLB) specifies that the page is cached

This exception is not maskable.

Processing

The common exception vector is used for this exception.

The *VCEI* or *VCED* code in the *Cause* register is set for instruction and data cache misses respectively.

The *BadVAddr* register holds the virtual address that caused the exception.

Virtual Coherency exception processing is shown in Figure 5-17.

Servicing

Using the appropriate CACHE instruction(s), the primary cache line at both the previous and the new virtual index should be invalidated† (and written back, if necessary), and the *PIDx* field of the secondary cache should be written with the new virtual index. Once completed, the program continues.

Software can avoid the cost of this exception by using consistent virtual primary cache indexes to access the same physical data.

† When a cache miss occurs, the processor refills the primary cache line at the present virtual index before taking an exception.

Bus Error Exception

Cause

A Bus Error exception is raised by board-level circuitry for events such as bus time-out, backplane bus parity errors, and invalid physical memory addresses or access types. This exception is not maskable.

A Bus Error exception occurs either when the **SysCmd(5)** bit indicates the data is erroneous (see Chapter 12) or the **IvdErr*** signal is asserted (Chapter 12). This can only occur when a cache miss refill, uncached reference, or an unbuffered write occurs synchronously; a Bus Error exception resulting from a buffered write transaction must be reported using the general interrupt mechanism.

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. Bus Error processing is shown in Figure 5-17.

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 virtual address is contained in the *EPC* register.
- 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* register 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.

Integer Overflow exception processing is shown in Figure 5-17.

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.

† See Appendix A for a description of these instructions.

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.

Trap exception processing is shown in Figure 5-17.

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.

† See Appendix A for a description of these instructions.

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.

System Call exception processing is shown in Figure 5-17.

Servicing

When this exception occurs, control is transferred to the applicable system routine.

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.

Breakpoint exception processing is shown in Figure 5-17.

Servicing

When the Breakpoint exception occurs, control is transferred to the applicable system routine. Additional distinctions can be made by analyzing the unused bits 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

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.

Reserved Instruction exception processing is shown in Figure 5-17.

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 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.

Coprocessor Unusable exception processing is shown in Figure 5-17.

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.

Floating-Point exception processing is shown in Figure 5-17.

Servicing

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

For an unimplemented instruction exception, the kernel should emulate the instruction; for other exceptions, the kernel should pass the exception to the user program that caused the exception.

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 the *EXL* bit is set in the *Status* register, and Watch is only maskable by setting the *EXL* bit in the *Status* register.

Processing

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

Watch exception processing is shown in Figure 5-17.

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, 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.

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 *Int-Mask* 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.

Interrupt exception processing is shown in Figure 5-17.

Servicing

If the interrupt is caused by one of the two software-generated exceptions (*SW1* or *SW0*), the interrupt condition is cleared by setting the corresponding *Cause* register bit to 0.

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

5.4 Exception Handling and Servicing Flowcharts

The remainder of this chapter contains flowcharts for the following exceptions and guidelines for their handlers:

- general exceptions and their exception handler
- TLB/XTLB miss exception and their exception handler
- cache error exception and its handler
- reset, soft reset and NMI exceptions, and a guideline to their handler.

Generally speaking, the exceptions are handled by hardware (HW); the exceptions are then serviced by software (SW).

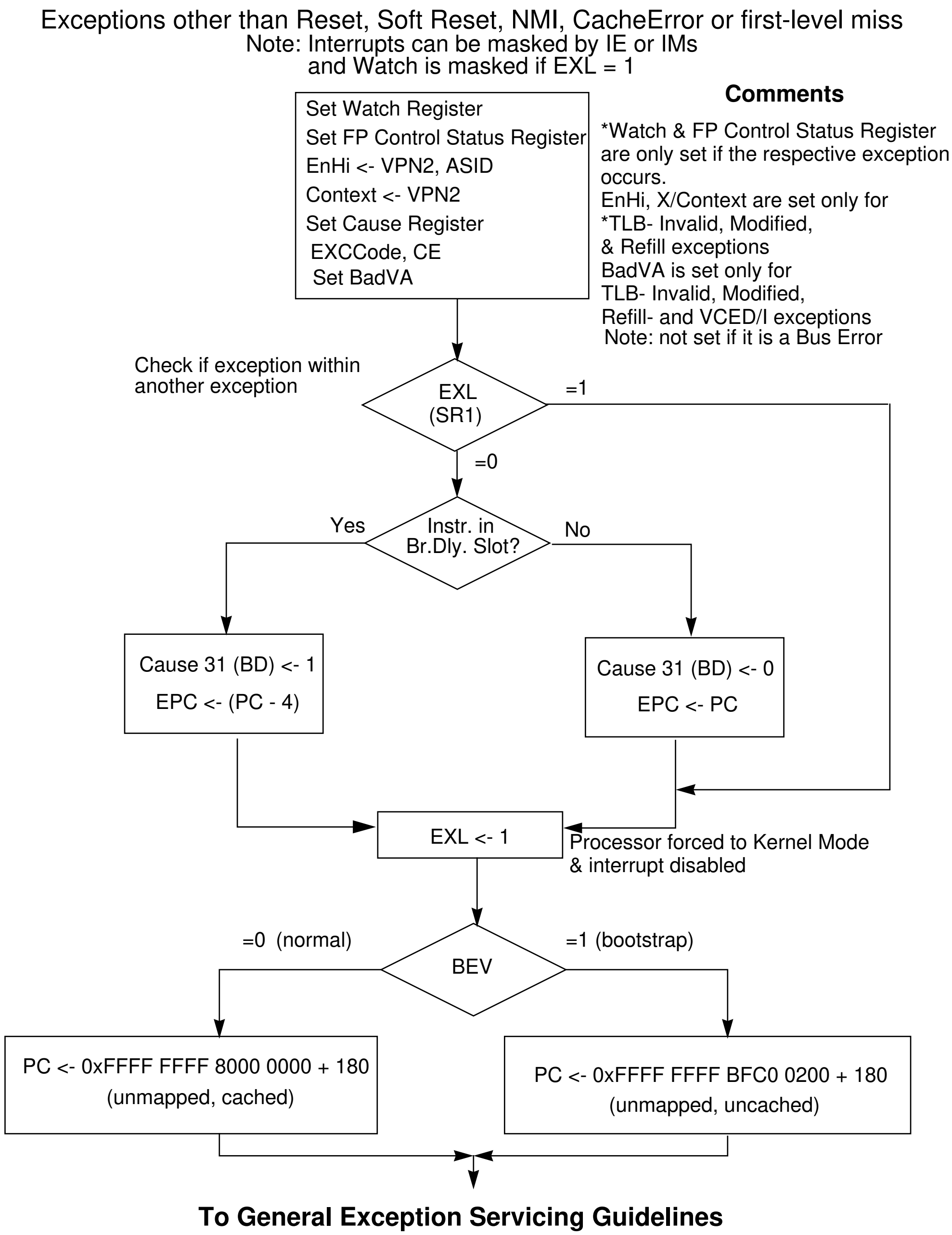


Figure 5-18 General Exception Handler (HW)

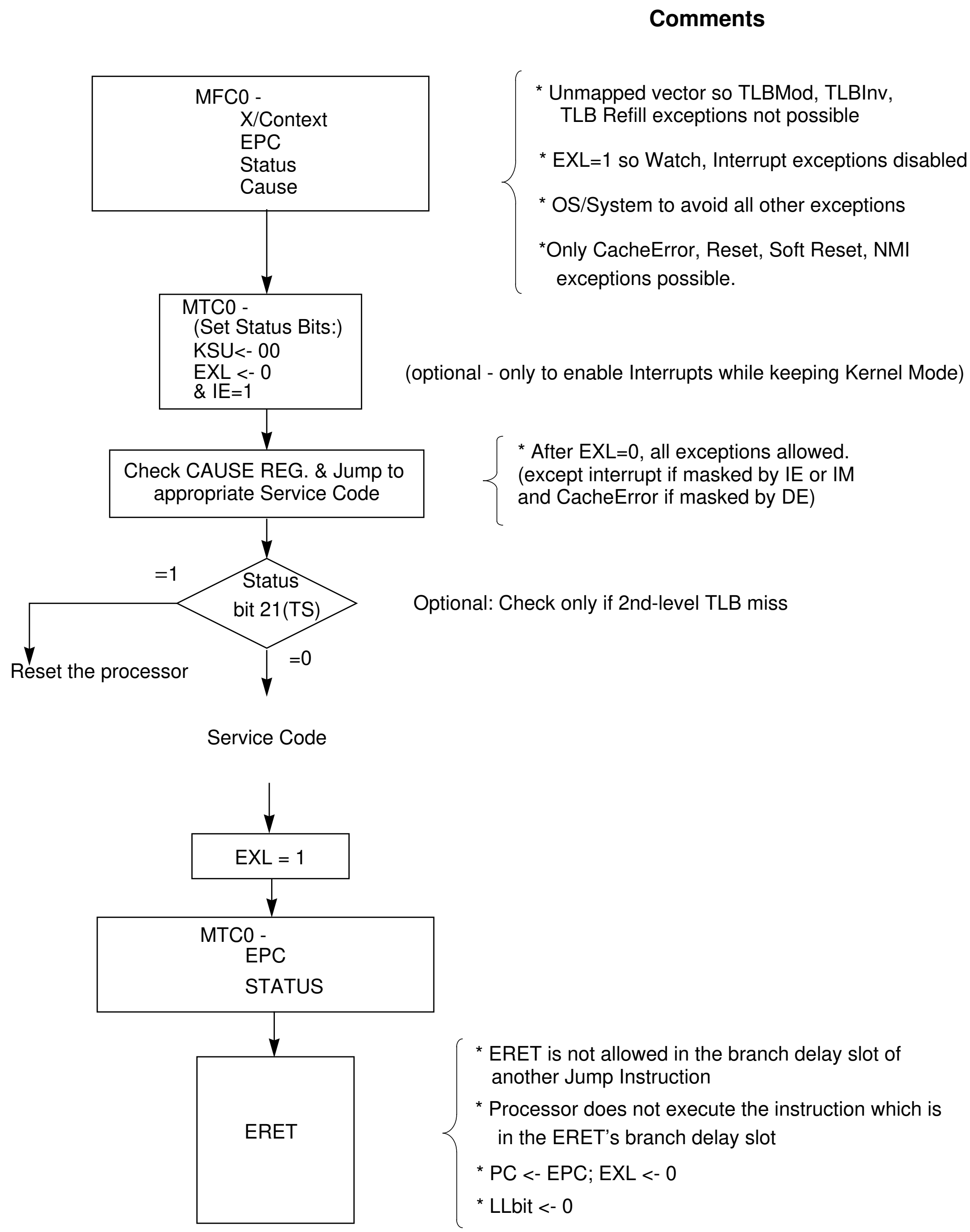


Figure 5-19 General Exception Servicing Guidelines (SW)

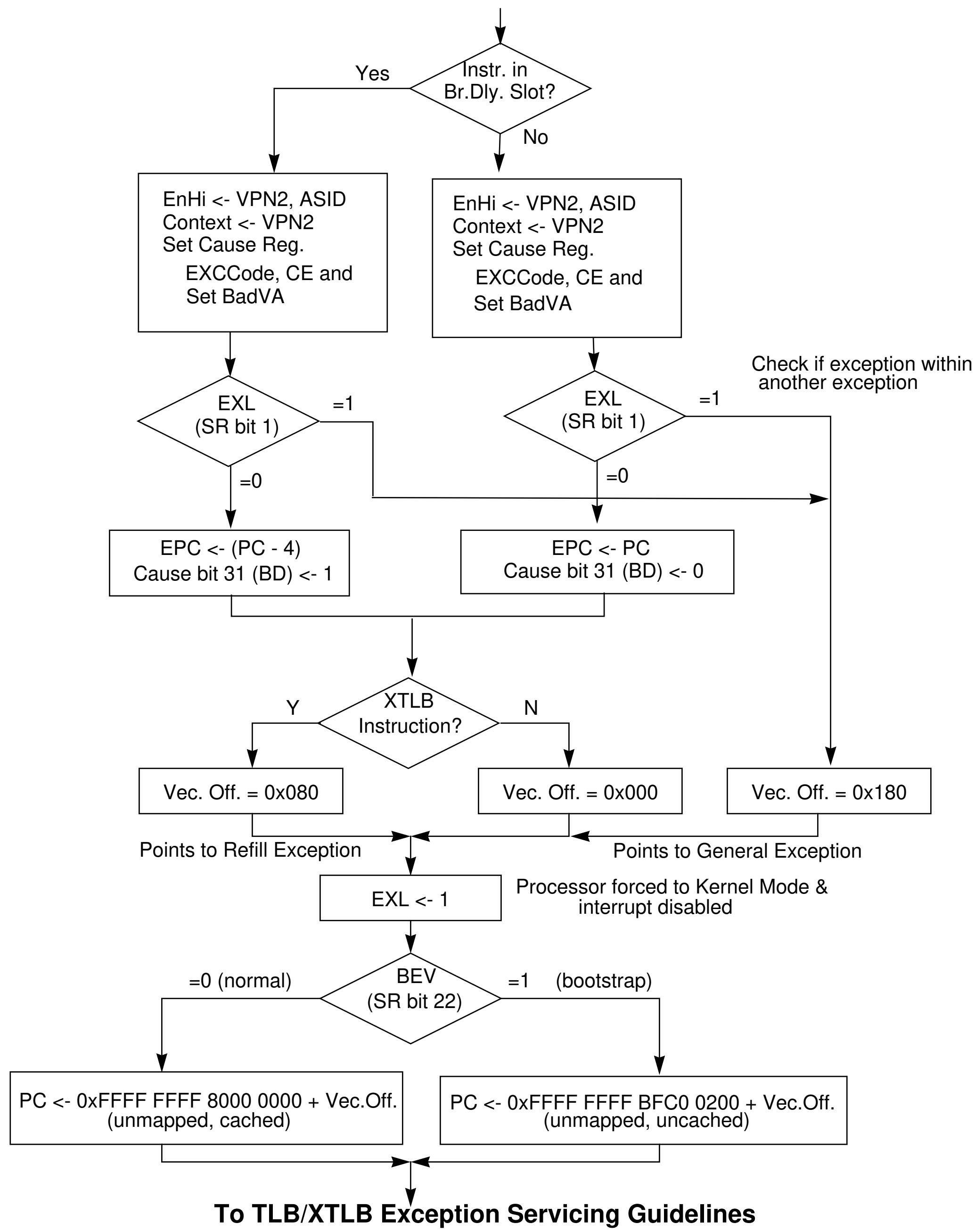
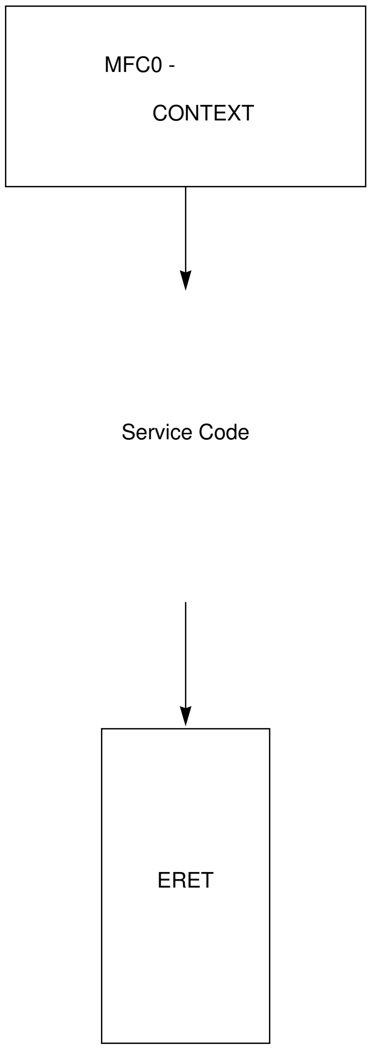


Figure 5-20 TLB/XTLB Miss Exception Handler (HW)



Comments

* Unmapped vector so TLBMod, TLBInv,
TLB Refill or VCEP exceptions
not possible

* EXL=1 so Watch, Interrupt exceptions disabled

* OS/System to avoid all other exceptions

*Only CacheError, Reset, Soft Reset, NMI
exceptions possible.

* Load the mapping of the virtual address in Context Reg.
Move it to ENLO and Write into the TLB

* There could be a TLB miss again during the mapping
of the data or instruction address. The processor will
jump to the general exception vector since the EXL is 1.
(Option to complete the first level refill in the general
exception handler or ERET to the original instruction
and take the exception again)

* ERET is not allowed in the branch delay slot of
another Jump Instruction

* Processor does not execute the instruction which is
in the ERET's branch delay slot

* PC <- EPC; EXL <- 0

* LLbit <- 0

Figure 5-21 TLB/XTLB Exception Servicing Guidelines (SW)

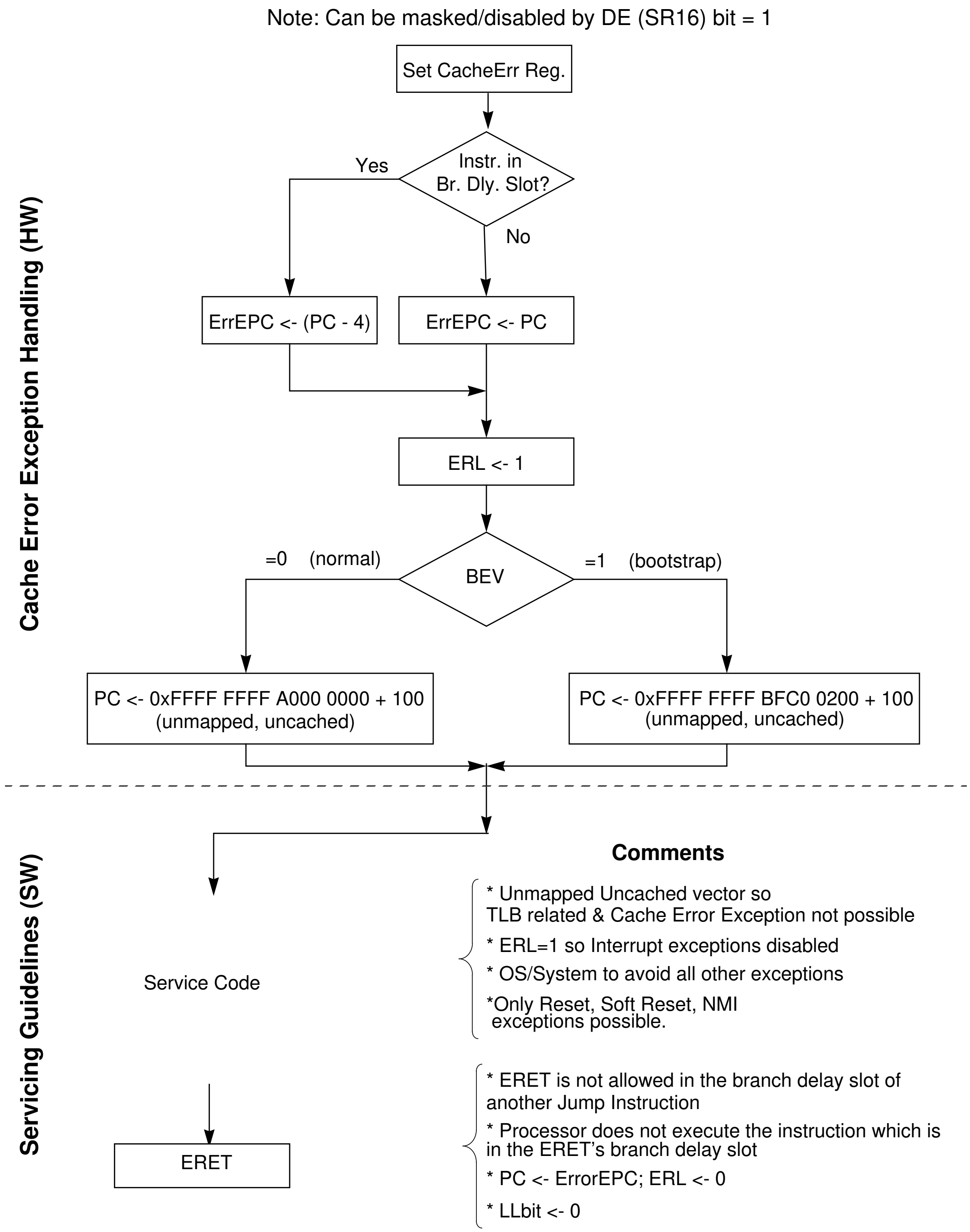


Figure 5-22 Cache Error Exception Handling (HW) and Servicing Guidelines (SW)

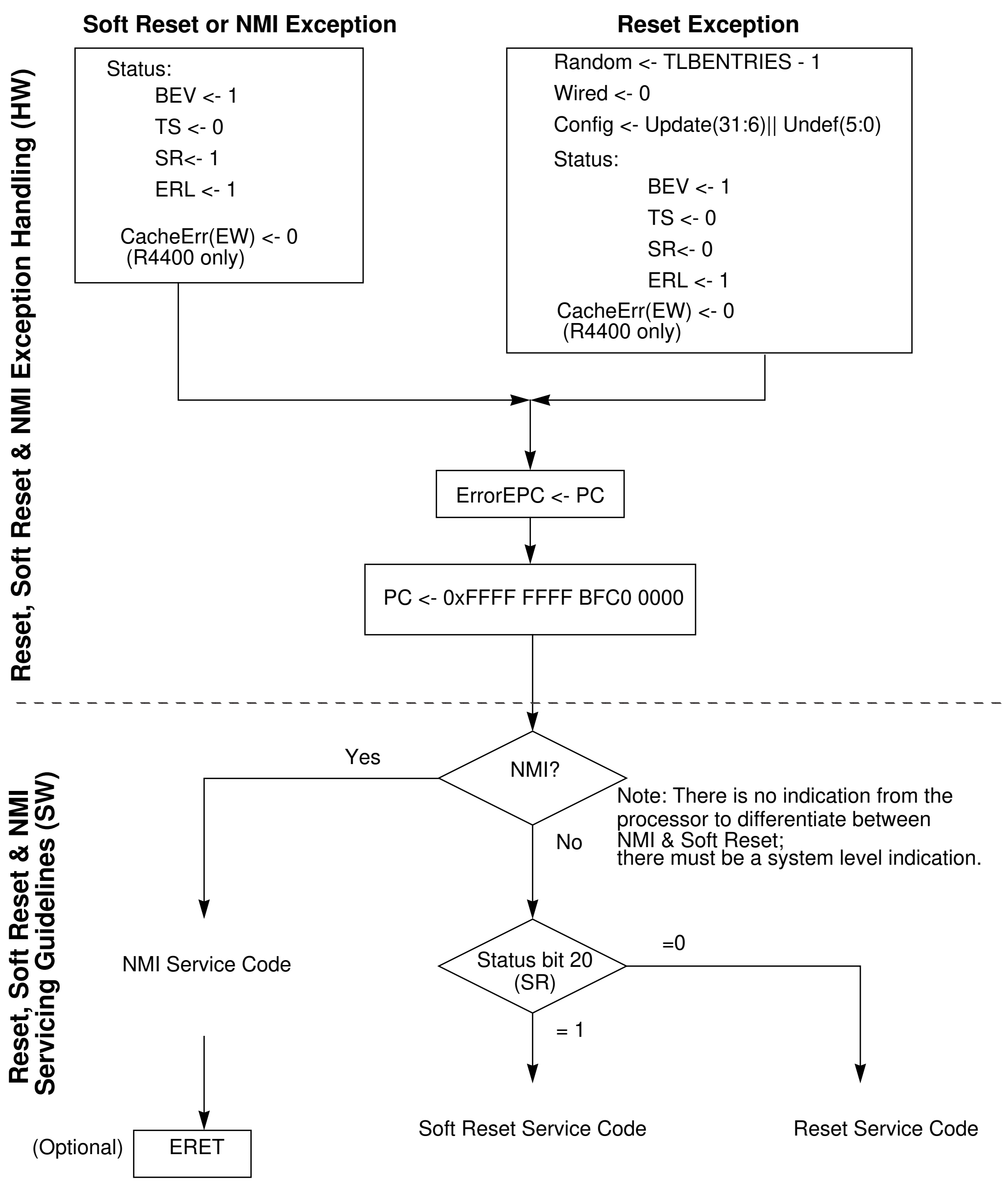


Figure 5-23 Reset, Soft Reset & NMI Exception Handling (HW) and Servicing Guidelines (SW)