Chapter 5

Exceptions and Interrupts

The interAptiv CPU receives exceptions from a number of sources, including arithmetic overflows, misses in the translation lookaside buffer (TLB), I/O interrupts, and system calls. When the CPU detects an exception, the normal sequence of instruction execution is suspended and the processor enters kernel mode, disables interrupts, loads the *Exception Program Counter* (*EPC*) register with the location where execution can restart after the exception has been serviced, and forces execution of a software exception handler located at a specific address.

The software exception handler saves the context of the processor, including the contents of the program counter, the current operating mode, and the status of the interrupts (enabled or disabled). This context is saved so it can be restored when the exception has been serviced.

Exceptions may be precise or imprecise. Precise exceptions are those for which the *EPC* can be used to identify the instruction that caused the exception. For precise exceptions, 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 (as indicated by the *BD* bit in the *Cause* register), the address of the branch instruction immediately preceding the delay slot. Imprecise exceptions, on the other hand, are those for which no return address can be identified. Bus error exceptions and CP2 exceptions are examples of imprecise exceptions.

This chapter contains the following sections:

5.1 Exception Conditions

When an exception condition occurs, the instruction causing the exception and all those that follow it in the pipeline are cancelled. Accordingly, any stall conditions and any later exception conditions that may have referenced this instruction are inhibited.

When the exception condition is detected on an instruction fetch, the CPU aborts that instruction and all instructions that follow. When the instruction graduates, the exception flag causes it to write various CP0 registers with the excep-

tion state, change the current program counter (PC) to the appropriate exception vector address, and clear the exception bits of earlier pipeline stages.

For most types of exceptions, this implementation allows all preceding instructions to complete execution and prevents all subsequent instructions from completing. Thus, the value in the *EPC* (or *ErrorEPC* for errors or *DEPC* for debug exceptions) is sufficient to restart execution. It also ensures that exceptions are taken in program. An instruction taking an exception may itself be aborted by an instruction further down the pipeline that takes an exception in a later cycle.

Imprecise exceptions are taken after the instruction that caused them has completed and potentially after following instructions have completed.

5.2 Exception Priority

Table 5.1 contains a list and a brief description of all exception conditions, The exceptions are listed in the order of their relative priority, from highest priority (Reset) to lowest priority (Load/store bus error). When several exceptions occur simultaneously, the exception with the highest priority is taken.

Table 5.1 Priority of Exceptions

Exception	Description
Reset	Assertion of SI_Reset signal.
DSS	EJTAG Debug Single Step.
DINT	EJTAG Debug Interrupt. Caused by the assertion of the external *EJ_DINT* input, or by setting the *EjtagBrk* bit in the *ECR* register.
DDBLImpr/DDBSImpr	Debug Data Break Load/Store. Imprecise.
NMI	Asserting edge of *SI_NMI* signal.
Machine Check	TLB write that conflicts with an existing entry.
Interrupt	Assertion of unmasked hardware or software interrupt signal.
Deferred Watch	Deferred Watch (unmasked by K|DM->!(K|DM) transition).
DIB	EJTAG debug hardware instruction break matched.
WATCH	A reference to an address in one of the watch registers (fetch).
AdEL	Fetch address alignment error. Fetch reference to protected address.
TLBL	Fetch TLB miss. Fetch TLB hit to page with V=0.
I-cache Error	Parity error on I-cache access.
IBE	Instruction fetch bus error.
DBp	EJTAG Breakpoint (execution of SDBBP instruction).
Sys	Execution of SYSCALL instruction.
Bp	Execution of BREAK instruction.
CpU	Execution of a coprocessor instruction for a coprocessor that is not enabled.
CEU	Execution of a CorExtend instruction modifying local state when CorExtend is not enabled.
RI	Execution of a Reserved Instruction.
FPE	Floating Point exception.

Table 5.1 Priority of Exceptions *(continued)*

Exception	Description
Ov	Execution of an arithmetic instruction that overflowed.
Tr	Execution of a trap (when trap condition is true).
DDBL / DDBS	EJTAG Data Address Break (address only).
WATCH	A reference to an address in one of the watch registers (data).
AdEL	Load address alignment error. Load reference to protected address.
AdES	Store address alignment error. Store to protected address.
TLBL	Load TLB miss. Load TLB hit to page with V=0
TLBS	Store TLB miss. Store TLB hit to page with V=0.
TLB Mod	Store to TLB page with D=0.
D-cache Error	Cache parity error. Imprecise.
DBE	Load or store bus error. Imprecise.

5.3 Exception Vector Locations

The location of the exception vector in the interAptiv core depends on the operating mode. If the core is in legacy mode, the exception vector location is the same as in previous MIPS processors. However, if the core is in Enhanced Virtual Address (EVA) mode, the exception vector can effectively be placed anywhere within kernel address space.

The *SI_EVAReset* pin, along with the $CONFIG5._K$ bit, determines whether the address mode is set to legacy or EVA at reset.

If the *SI_EVAReset* pin is deasserted at reset, the interAptiv core comes up in legacy mode and hardware takes the following actions:

- The $CONFIG5._K$ bit becomes read-write and is programmed by hardware to a value of 0 to indicate legacy mode. Software can then set this bit to initiate the switch from legacy mode to EVA mode.
- Hardware sets the CP0 memory segmentation registers (*SegCtl0* - *SegCtl2*) for Legacy mode. Note that these registers are new in the interAptiv core and are not used by legacy software. However, they are used by hardware during normal operation, so their default values should not be changed.

If the *SI_EVAReset* pin is asserted at reset, the interAptiv core comes up in EVA mode and hardware takes the following actions:

- The $CONFIG5._K$ bit becomes read-only and is forced to a value of 1 to indicate EVA mode.
- Hardware sets the CP0 memory segmentation registers (*SegCtl0* - *SegCtl2*) for EVA mode.

Another degree of flexibility in the selection of the vector base address, for use when $Status_{BEV}$ equals 1, is provided via a set of input pins, *SI_LegacyUseExceptionBase*, *SI_ExceptionBase*[31:12], and *SI_ExceptionBaseMask*[27:20]. Table 5.3 shows the vector base address when the core is in legacy mode and the *SI_LegacyUseExceptionBase* pin is 0. Table 5.4 shows the vector base addresses when the core is in legacy mode and the *SI_LegacyUseExceptionBase* equals 1. As

can be seen in Table 5.4, when *SI_UseExceptionBase* equals 1, the exception vectors for cases where $Status_{BEV}$ = 0 are not affected.

When the *SI_LegacyUseExceptionBase* pin is 0 and the $Config5._K$ bit is cleared, the device is in legacy mode. In this mode the exception vector location defaults of 0xBFC0_0000 and the *SI_ExceptionBase[31:12]* pins are ignored.

When the *SI_LegacyUseExceptionBase* pin is 1 and the $Config5._K$ bit is cleared, the device is still in legacy mode, but the *SI_ExceptionBase[29:12]* pins are used to indicate the location of the exception vector. Bits 31:30 are forced to a value of 10_2, placing the exception vector somewhere in kseg0/kseg1 space.

If the $Config5._K$ bit is set, the device is in EVA mode. In this case the *SI_LegacyUseExceptionBase* pin is ignored and the *SI_ExceptionBase[31:12]* pins are used to derive the location of the exception vector.

The function of the $Config5._K$ bit and the *SI_LegacyUseExceptionBase* pin is shown in Table 5.2. For more information on EVA mode, refer to the MMU chapter.

Table 5.2 *SI_LegacyUseExceptionBase* Pin and CONFIG5.K Encoding

CONFIG5.K Bit	*SI_LegacyUse-ExceptionBase* Pin	Condition	Action
0	0	Legacy Mode *SI_ExceptionBase[31:12]* pins are not used.	Use default BEV location of 0xBFC0_0000.
0	1	Legacy Mode Use only *SI_ExceptionBase[29:12]* for the BEV base location. Bits 31:30 are forced to a value of 10_2 to put the BEV vector into KSEG0/KSEG1 virtual address space.	The BEV location is determined as follows: *SI_ExceptionBase[31:12]* = 10_2, *SI_ExceptionBase[29:12]* pins, 12'b0 Bits 31:30 are forced to a value of 2'b10 to put the BEV vector into KSEG0/KSEG1 virtual address space.
1	Don't care	EVA Mode Use *SI_ExceptionBase[31:12]* pins.	The *SI_ExceptionBase[31:12]* pins are used directly to derive the BEV location. The *SI_LegacyUseExceptionBase* pin is ignored.

In legacy mode, when the *SI_LegacyUseExceptionBase* pin is 0, the Reset, Soft Reset, NMI, and EJTAG Debug exceptions are vectored to a specific location, as shown in Table 5.3. Addresses for all other exceptions are a combination of a vector offset and a vector base address. In the interAptiv core, software is allowed to specify the vector base address via the *EBase* register for exceptions that occur when $Status_{BEV}$ equals 0.

Table 5.3 Exception Vector Base Addresses — Legacy Mode, SI_LegacyUseExceptionBase = 0

Exception	$Status_{BEV}$	
	0	1
Reset, NMI	`0xBFC0.0000`	
EJTAG Debug (with *ProbEn* = 0, in the EJTAG_Control_register and *DCR.RDVec*=0)	`0xBFC0.0480`	
EJTAG Debug (with *ProbEn* = 0, in the EJTAG_Control_register and *DCR.RDVec*=1)	*DebugVectorAddr*`[31:7] || 2b0000000`	

Table 5.3 Exception Vector Base Addresses — Legacy Mode, SI_LegacyUseExceptionBase = 0 *(continued)*

Exception	$Status_{BEV}$	
	0	1
EJTAG Debug (with *ProbEn* = 1 in the EJTAG_Control_register)	0xFF20.0200	
Cache Error	$EBase_{31:30}$ || 1 || $EBase_{28:12}$ || 0x000 Note that $EBase_{31:30}$ have the fixed value 0b10	0xBFC0.0300
Other	$EBase_{31:12}$ || 0x000 Note that $EBase_{31:30}$ have the fixed value 0b10	0xBFC0.0200
'||' denotes bit string concatenation		

In legacy mode, when the *SI_LegacyUseExceptionBase* pin is 0, the Reset, Soft Reset, NMI, and EJTAG Debug exceptions are vectored to a specific location, as shown in Table 5.4.

Table 5.4 Exception Vector Base Addresses — Legacy Mode, SI_LegacyUseExceptionBase = 1

Exception	$Status_{BEV}$	
	0	1
Reset, NMI	0b10 || *SI_ExceptionBase*[29:12] || 0x000	
EJTAG Debug (with *ProbEn* = 0 in the EJTAG_Control_register and *DCR.RDVec*=0)	0b10 || *SI_ExceptionBase*[29:12] || 0x480	
EJTAG Debug (with *ProbEn* = 0 in the EJTAG_Control_register and *DCR.RDVec*=1)	*DebugVectorAddr*[31:7] || 2b0000000	
EJTAG Debug (with *ProbEn* = 1 in the EJTAG_Control_register)	0xFF20.0200	
Cache Error	$EBase_{31:30}$ || 1 || $EBase_{28:12}$ || 0x000 Note that $EBase_{31:30}$ have the fixed value 0b10. Exception vector resides in kseg1.	0b101 || *SI_ExceptionBase*[28:12] || 0x300 Exception vector resides in kseg1.
Other	$EBase_{31:12}$ || 0x000 Note that $EBase_{31:30}$ have the fixed value 0b10 Exception vector resides in kseg0/kseg1.	0b10 || *SI_ExceptionBase*[29:12] || 0x200 Exception vector resides in kseg0/kseg1.
'||' denotes bit string concatenation		

Table 5.5 shows the offsets from the vector base address as a function of the exception. Note that the IV bit in the *Cause* register causes interrupts to use a dedicated exception vector offset, rather than the general exception vector. Table 5.22 (on page 285) shows the offset from the base address in the case where $Status_{BEV}$ = 0 and $Cause_{IV}$ = 1.

Table 5.7 combines these three tables into one that contains all possible vector addresses as a function of the state that can affect the vector selection. To avoid complexity in the table, it is assumed that $IntCtl_{VS}$ = 0.

Table 5.5 Exception Vector Offsets

Exception	Vector Offset
TLB Refill, *EXL* = 0	0x000
General Exception	0x180
Interrupt, $Cause_{IV}$ = 1	0x200 (In Release 3 implementations, this is the base of the vectored interrupt table when $Status_{BEV}$ = 0)
Reset, NMI	None (uses reset base address)

In EVA mode, when the *SI_LegacyUseExceptionBase* pin is ignored and the Reset, Soft Reset, NMI, and EJTAG Debug exceptions are vectored to a location determined by the programming of the three Segment Control registers (*SegCtl0* - *SegCtl2*), as shown in Table 5.6.

Table 5.6 Exception Vector Base Addresses — EVA Mode

Exception	$Status_{BEV}$	
	0	1
Reset, NMI	*SI_ExceptionBase*[31:12] || 0x000	
EJTAG Debug (with *ProbEn* = 0 in the EJTAG_Control_register and *DCR.RDVec*=0)	*SI_ExceptionBase*[31:12] || 0x480	
EJTAG Debug (with *ProbEn* = 0 in the EJTAG_Control_register and *DCR.RDVec*=1)	*DebugVectorAddr*[31:7] || 2b0000000	
EJTAG Debug (with *ProbEn* = 1 in the EJTAG_Control_register)	0xFF20.0200	
Cache Error	$EBase_{31:12}$ || 0x000	*SI_ExceptionBase*[31:12] || 0x300 (Forced uncached)
Other	$EBase_{31:12}$ || 0x000	*SI_ExceptionBase*[31:12] || 0x200
'||' denotes bit string concatenation		

Table 5.7 Exception Vectors

Exception	$Config5_K$	SI_LegacyUseExceptionBase	$Status_{BEV}$	$Status_{EXL}$	$Cause_{IV}$	EJTAG ProbEn	Vector ($IntCtl_{VS}$ = 0)
Reset, NMI	0	0	x	x	x	x	0xBFC0.0000
Reset, NMI	0	1	x	x	x	x	0b10 || *SI_ExceptionBase*[29:12] || 0x000
Reset, NMI	1	x	x	x	x	x	*SI_ExceptionBase*[31:12] || 0x000
EJTAG Debug	0	0	x	x	x	0	0xBFC0.0480 (if *DCR.RDVec*=0) *DebugVectorAddr*[31:7] || 2b0000000 (if *DCR.RDVec*=1)
EJTAG Debug	0	1	x	x	x	0	0b10 || *SI_ExceptionBase*[29:12] || 0x480 (if *DCR.RDVec*=0) *DebugVectorAddr*[31:7] || 2b0000000 (if *DCR.RDVec*=1)
EJTAG Debug	1	x	x	x	x	0	*SI_ExceptionBase*[31:12] || 0x480 (if *DCR.RDVec*=0) *DebugVectorAddr*[31:7] || 2b0000000 (if *DCR.RDVec*=1)
EJTAG Debug	x	x	x	x	x	1	0xFF20.0200
TLB Refill	x	x	0	0	x	x	*EBase*[31:12] || 0x000
TLB Refill	x	x	0	1	x	x	*EBase*[31:12] || 0x180
TLB Refill	0	0	1	0	x	x	0xBFC0.0200
TLB Refill	0	1	1	0	x	x	0b10 || *SI_ExceptionBase*[29:12] || 0x200
TLB Refill	1	x	1	0	x	x	*SI_ExceptionBase*[31:12] || 0x200
TLB Refill	0	0	1	1	x	x	0xBFC0.0380
TLB Refill	0	1	1	1	x	x	0b10 || *SI_ExceptionBase*[29:12] || 0x380
TLB Refill	1	x	1	1	x	x	*SI_ExceptionBase*[31:12] || 0x380
Cache Error	0	x	0	x	x	x	*EBase*[31:30] || 0b1 || *EBase*[28:12] || 0x100
Cache Error	1	x	0	x	x	x	0xBFC0.0100
Cache Error	0	0	1	x	x	x	0xBFC0.0300
Cache Error	0	1	1	x	x	x	0b101 || *SI_ExceptionBase*[28:12] || 0x300
Cache Error	1	x	1	x	x	x	*SI_ExceptionBase*[31:12] || 0x300
Interrupt	x	x	0	0	0	x	*EBase*[31:12] || 0x180
Interrupt	x	x	0	0	1	x	*EBase*[31:12] || 0x200
Interrupt	0	0	1	0	0	x	0xBFC0.0380
Interrupt	0	1	1	0	0	x	0b10 || *SI_ExceptionBase*[29:12] || 0x380
Interrupt	1	x	1	0	0	x	*SI_ExceptionBase*[31:12] || 0x380
Interrupt	0	0	1	0	1	x	0xBFC0.0400
Interrupt	0	1	1	0	1	x	0b10 || *SI_ExceptionBase*[29:12] || 0x400
Interrupt	1	x	1	0	1	x	*SI_ExceptionBase*[31:12] || 0x400

Table 5.7 Exception Vectors *(continued)*

Exception	$Config5_K$	SI_LegacyUseExceptionBase	$Status_{BEV}$	$Status_{EXL}$	$Cause_{IV}$	EJTAG ProbEn	Vector ($IntCtl_{VS}$ = 0)
All others	x	x	0	x	x	x	*EBase*[31:12] || 0x180
All others	0	0	1	x	x	x	0xBFC0.0380
All others	0	1	1	x	x	x	0b10 || *SI_ExceptionBase*[29:12] || 0x380
All others	1	x	1	x	x	x	*SI_ExceptionBase*[31:12] || 0x380

'x' denotes don't care,
'||' denotes bit string concatenation

5.4 General Exception Processing

With the exception of Reset, NMI, cache error, and EJTAG Debug exceptions, which have their own special processing as described below, exceptions have the same basic processing flow:

- If the *EXL* bit in the *Status* register is zero, the *EPC* register is loaded with the PC at which execution will be restarted, and the *BD* bit is set appropriately in the *Cause* register. The value loaded into the *EPC* register is dependent on whether the processor implements the MIPS16 ASE, and whether the instruction is in the delay slot of a branch or jump which has delay slots. Table 5.8 shows the value stored in each of the CP0 PC registers, including *EPC*.

  If the *EXL* bit in the *Status* register is set, the *EPC* register is not loaded and the *BD* bit is not changed in the *Cause* register.

Table 5.8 Value Stored in EPC, ErrorEPC, or DEPC on Exception

MIPS16 Implemented?	In Branch/Jump Delay Slot?	Value stored in EPC/ErrorEPC/DEPC
No	No	Address of the instruction
No	Yes	Address of the branch or jump instruction (PC-4)
Yes	No	Upper 31 bits of the address of the instruction, combined with the ISA Mode bit
Yes	Yes	Upper 31 bits of the branch or jump instruction (PC-2 in the MIPS16 ISA Mode and PC-4 in the 32-bit ISA Mode), combined with the ISA Mode bit

- The *CE*, and *ExcCode* fields of the *Cause* registers are loaded with the values appropriate to the exception. The *CE* field is loaded, but not defined, for any exception type other than a coprocessor unusable exception.
- The *EXL* bit is set in the *Status* register.
- The processor begins executing at the exception vector.

The value loaded into *EPC* represents the restart address for the exception and need not be modified by exception handler software in the normal case. Software need not look at the *BD* bit in the *Cause* register unless it wishes to identify the address of the instruction that actually caused the exception.

Note that individual exception types may load additional information into other registers. This is noted in the description of each exception type below.

Operation:

```
/* If Status_EXL is 1, all exceptions go through the general exception vector */
/* and neither the EPC nor Cause_BD are modified */
if Status_EXL = 1 then
   vectorOffset ← 0x180
else
   /* For implementations that include the MIPS16e ASE, calculate potential */
   /* PC adjustment for exceptions in the delay slot */
   if Config1_CA = 0 then
      restartPC ← PC
      branchAdjust ← 4       /* Possible adjustment for delay slot */
   else
      restartPC ← PC_31:1 || ISAMode
      if (ISAMode = 0) or ExtendedMIPS16Instruction
         branchAdjust ← 4    /* Possible adjustment for 32-bit MIPS delay slot */
      else
         branchAdjust ← 2    /* Possible adjustment for MIPS16 delay slot */
      endif
   endif
   if InstructionInBranchDelaySlot then
      EPC ← restartPC - branchAdjust/* PC of branch/jump */
      Cause_BD ← 1
   else
      EPC ← restartPC              /* PC of instruction */
      Cause_BD ← 0
   endif

   /* Compute vector offsets as a function of the type of exception */
   if ExceptionType = TLBRefill then
      vectorOffset ← 0x000
   elseif (ExceptionType = Interrupt) then
      if (Cause_IV = 0) then
         vectorOffset ← 0x180
      else
         if (Status_BEV = 1) or (IntCtl_VS = 0) then
            vectorOffset ← 0x200
         else
            if Config3_VEIC = 1 then
               VecNum ← Cause_RIPL
            else
               VecNum ← VIntPriorityEncoder()
            endif
            vectorOffset ← 0x200 + (VecNum × (IntCtl_VS || 0b00000))
         endif /* if (Status_BEV = 1) or (IntCtl_VS = 0) then */
      endif /* if (Cause_IV = 0) then */
   endif /* elseif (ExceptionType = Interrupt) then */
endif /* if Status_EXL = 1 then */

Cause_CE ← FaultingCoprocessorNumber
Cause_ExcCode ← ExceptionType
```

```
Status_EXL ← 1

if Config1_CA = 1 then
   ISAMode ← 0
endif

/* Calculate the vector base address */
if Status_BEV = 1 then
   vectorBase ← 0xBFC0.0200
else
   if ArchitectureRevision ≥ 2 then
      /* The fixed value of EBase_31:30 forces the base to be in kseg0 or kseg1 */
      vectorBase ← EBase_31:12 || 0x000
   else
      vectorBase ← 0x8000.0000
   endif
endif

/* Exception PC is the sum of vectorBase and vectorOffset */
PC ← vectorBase_31:30 || (vectorBase_29:0 + vectorOffset_29:0)
                         /* No carry between bits 29 and 30 */
```

5.5 Debug Exception Processing

All debug exceptions have the same basic processing flow:

- The *DEPC* register is loaded with the program counter (PC) value at which execution will be restarted and the *DBD* bit is set appropriately in the *Debug* register. The value loaded into the *DEPC* register is the current PC if the instruction is not in the delay slot of a branch, or the PC-4 of the branch if the instruction is in the delay slot of a branch.
- The *DSS*, *DBp*, *DDBL*, *DDBS*, *DIB,* and *DINT* bits in the *Debug* register are updated appropriately, depending on the debug exception type.
- *Halt* and *Doze* bits in the *Debug* register are updated appropriately.
- The *DM* bit in the *Debug* register is set to 1.
- The processor is started at the debug exception vector.

The value loaded into *DEPC* represents the restart address for the debug exception and need not be modified by the debug exception handler software in the usual case. Debug software need not look at the *DBD* bit in the *Debug* register unless it wishes to identify the address of the instruction that actually caused the debug exception.

A unique debug exception is indicated through the *DSS*, *DBp*, *DDBL*, *DDBS*, *DIB* and *DINT* bits (D* bits [5:0]) in the *Debug* register.

No other CP0 registers or fields are changed due to the debug exception, and thus no additional state is saved.

Operation:

```
if InstructionInBranchDelaySlot then
   DEPC ← PC-4
```

```
    Debug_DBD ← 1
else
    DEPC ← PC
    Debug_DBD ← 0
endif
Debug_D* bits at at [5:0] ← DebugExceptionType
Debug_Halt ← HaltStatusAtDebugException
Debug_Doze ← DozeStatusAtDebugException
Debug_DM ← 1
if EJTAGControlRegister_ProbTrap = 1 then
    PC ← 0xFF20_0200
else
    if DebugControlRegister_RDVec = 1 then
        if CacheErr then
            PC ← 2#101 || DebugVectorAddr_28:7 || 2#0000000
        else
            PC ← 2#10 || DebugVectorAddr_29:7 || 2#0000000
    else
        if SI_UseExceptionBase
            if CacheErr then
                PC ← 2#101 || SI_ExceptionBase[28:12] || 0x000
            else
                PC ← 2#10 || SI_ExceptionBase[29:12] || 0x000
        else
            PC ← 0xBFC0_0480
endif
```

The location of the debug exception vector is determined by the *ProbTrap* bit in the *EJTAG Control* register (*ECR*) and the *RDVec* bit in the *Debug Control* register (*DCR*), as shown in Table 5.9.

Table 5.9 Debug Exception Vector Addresses

ProbTrap bit in ECR Register	RDVec bit in DCR Register	Debug Exception Vector Address
0	0	0xBFC0 0480
0	1	$DebugVectorAddr_{31:7}$ || 0000000
1	0	0xFF20 0200 in dmseg
1	1	

The value in the optional drseg register *DebugVectorAddr* (offset 0x00020) is used as the debug exception vector when the *ECR ProbTrap* bit is 0 and when enabled through the optional *RDVec* control bit in the *Debug Control Register* (*DCR*). Bit 0 of *DebugVectorAddr* determines the ISA mode used to execute the handler. Figure 5.1 shows the format of the *DebugVectorAddr* register; Table 5.10 describes the *DebugVectorAddr* register fields.

Figure 5.1 DebugVectorAddr Register Format

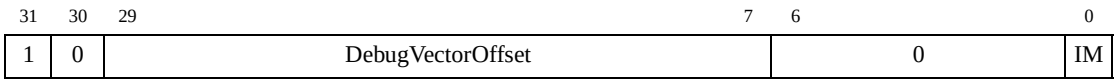


Table 5.10 DebugVectorAddr Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bit(s)			
1	31	Ignored on write; returns one on read.	R	1
DebugVectorOffset	29:7	Programmable Debug Exception Vector Offset	R/W	Preset to 0x7F8009
IM	0	ISA mode to be used for exception handler	R	0
0	30,6:1	Ignored on write; returns zero on read.	R	0

Bits 31:30 of the *DebugVectorAddr* register are fixed with the value 0b10, and the addition of the base address and the exception offset is done inhibiting a carry between bit 29 and bit 30 of the final exception address. The combination of these two restrictions forces the final exception address to be in the kseg0 or kseg1 unmapped virtual address segments. For cache error exceptions, bit 29 is forced to a 1 in the ultimate exception base address, so that this exception always runs in the kseg1 unmapped, uncached virtual address segment.

When MIPS16 is implemented, the power-up state of *IM* is zero. If the implementation does not include MIPS16, the *IM* field is read-only, should be written with zero and will return 0 on a read.

If the TAP is not implemented, the debug exception vector location is as if *ProbTrap*=0.

5.6 Exception Descriptions

The following subsections describe each of the exceptions listed in the same sequence as shown in Table 5.1.

5.6.1 Reset Exception

A reset exception occurs when the *SI_Reset* signal is asserted to the processor. This exception is not maskable. When a Reset exception occurs, the processor performs a full reset initialization, including aborting state machines, establishing critical state, and generally placing the processor in a state in which it can execute instructions from uncached, unmapped address space. On a Reset exception, the state of the processor is not defined, with the following exceptions:

- The *Random* register is initialized to the number of TLB entries - 1.
- The *Wired* register is initialized to zero.
- The *Config* register is initialized with its boot state.
- The *RP*, *BEV*, *TS*, *SR*, *NMI*, and *ERL* fields of the *Status* register are initialized to a specified state.
- The *I*, *R*, and *W* fields of the *WatchLo* register are initialized to 0.
- The *ErrorEPC* register is loaded with PC-4 if the state of the processor indicates that it was executing an instruction in the delay slot of a branch. Otherwise, the *ErrorEPC* register is loaded with PC. Note that this value may or may not be predictable.
- PC is loaded with 0xBFC0_0000.

Cause Register ExcCode Value:

None

Additional State Saved:

None

Entry Vector Used:

Reset (exact vector address depends on mode of operation - Legacy/EVA)

Operation:

```
Random ← TLBEntries - 1
Wired ← 0
Config ← ConfigurationState
Status_RP ← 0
Status_BEV ← 1
Status_TS ← 0
Status_SR ← 0
Status_NMI ← 0
Status_ERL ← 1
WatchLo_I ← 0
WatchLo_R ← 0
WatchLo_W ← 0
if InstructionInBranchDelaySlot then
   ErrorEPC ← PC - 4
else
   ErrorEPC ← PC
endif
PC ← 0xBFC0_0000
```

5.6.2 Debug Single Step Exception

A debug single step exception occurs after the CPU has executed one/two instructions in non-debug mode, when returning to non-debug mode after debug mode. One instruction is allowed to execute when returning to a non-jump/branch instruction, otherwise two instructions are allowed to execute since the jump/branch and the instruction in the delay slot are executed as one step. Debug single step exceptions are enabled by the *SSt* bit in the *Debug* register, and are always disabled for the first one/two instructions after a DERET.

The *DEPC* register points to the instruction on which the debug single step exception occurred, which is also the next instruction to single step or execute when returning from debug mode. So the *DEPC* register will not point to the instruction which has just been single stepped, but rather the following instruction. The *DBD* bit in the *Debug* register is never set for a debug single step exception, since the jump/branch and the instruction in the delay slot is executed in one step.

Exceptions occurring on the instruction(s) executed with debug single step exception enabled are taken even though debug single step was enabled. For a normal exception (other than reset), a debug single step exception is then taken on the first instruction in the normal exception handler. Debug exceptions are unaffected by single step mode, e.g. returning to a SDBBP instruction with debug single step exceptions enabled causes a debug software breakpoint exception, and *DEPC* will point to the SDBBP instruction. However, returning to an instruction (not jump/branch) just before the SDBBP instruction, causes a debug single step exception with *DEPC* pointing to the SDBBP instruction.

To ensure proper functionality of single step, the debug single step exception has priority over all other exceptions, except reset and soft reset.

Debug Register Debug Status Bit Set

DSS

Additional State Saved

None

Entry Vector Used

Debug exception vector

5.6.3 Debug Interrupt Exception

A debug interrupt exception is either caused by the EjtagBrk bit in the *EJTAG Control* register (controlled through the TAP), or caused by the debug interrupt request signal to the CPU.

The debug interrupt exception is an asynchronous debug exception which is taken as soon as possible, but with no specific relation to the executed instructions. The *DEPC* register is set to the instruction where execution should continue after the debug handler is through. The *DBD* bit is set based on whether the interrupted instruction was executing in the delay slot of a branch.

Debug Register Debug Status Bit Set

DINT

Additional State Saved

None

Entry Vector Used

Debug exception vector

5.6.4 Non-Maskable Interrupt (NMI) Exception

A non maskable interrupt exception occurs when the *SI_NMI* signal is asserted to the processor. *SI_NMI* is an edge sensitive signal - only one NMI exception will be taken each time it is asserted. An NMI exception occurs only at instruction boundaries, so it does not cause any reset or other hardware initialization. The state of the cache, memory, and other processor states are consistent and all registers are preserved, with the following exceptions:

- The *BEV*, *TS*, *SR*, *NMI*, and *ERL* fields of the *Status* register are initialized to a specified state.
- The *ErrorEPC* register is loaded with PC-4 if the state of the processor indicates that it was executing an instruction in the delay slot of a branch. Otherwise, the *ErrorEPC* register is loaded with PC.
- PC is loaded with 0xBFC0_0000.

Cause Register ExcCode Value:

None

Additional State Saved:

None

Entry Vector Used:

Reset (exact vector address depends on mode of operation - Legacy/EVA)

Operation:

```
Status_BEV ← 1
Status_TS ← 0
Status_SR ← 0
Status_NMI ← 1
Status_ERL ← 1
if InstructionInBranchDelaySlot then
   ErrorEPC ← PC - 4
else
   ErrorEPC ← PC
endif
PC ← 0xBFC0_0000
```

5.6.5 Machine Check Exception

A machine check exception occurs when the processor detects an internal inconsistency. The following conditions cause a machine check exception:

- A TLBWI instruction to the FTLB and the index and VPN2 are not consistent and the EHINV bit is not set.
- A TLBWI instruction to the FTLB and the PageMask register does not correspond to the FTLB page size setting in bits 12:8 of the *Config4* register (*Config4*$_{FTLB\ Page\ Size}$)
- A TLBP instruction and a duplicate/overlap is detected across the FTLB/VTLB.
- Any TLB lookup and a duplicate/overlap is detected across the FTLB/VTLB.

Cause Register ExcCode Value:

MCheck

Additional State Saved:

None

Entry Vector Used:

General exception vector (offset 0x180)

5.6.6 Interrupt Exception

The interrupt exception occurs when one or more of the six hardware, two software, or timer interrupt requests is enabled by the *Status* register and the interrupt input is asserted. See 5.8 "Interrupts" on page 278 for more details about the processing of interrupts.

Register ExcCode Value:

Int

Additional State Saved:

Table 5.11 Register States an Interrupt Exception

Register State	Value
CauseIP	Indicates the interrupts that are pending.

Entry Vector Used:

See 5.8.2 "Generation of Exception Vector Offsets for Vectored Interrupts" on page 285 for the entry vector used, depending on the interrupt mode the processor is operating in.

5.6.7 Debug Instruction Break Exception

A debug instruction break exception occurs when an instruction hardware breakpoint matches an executed instruction. The *DEPC* register and DBD bit in the *Debug* register indicate the instruction that caused the instruction hardware breakpoint to match. This exception can only occur if instruction hardware breakpoints are implemented.

Debug Register Debug Status Bit Set:

DIB

Additional State Saved:

None

Entry Vector Used:

Debug exception vector

5.6.8 Watch Exception — Instruction Fetch or Data Access

The Watch facility provides a software debugging vehicle by initiating a watch exception when an instruction or data reference matches the address information stored in the *WatchHi* and *WatchLo* registers. A Watch exception is taken immediately if the *EXL* and *ERL* bits of the *Status* register are both zero and the *DM* bit of the *Debug* register is also zero. If any of those bits is a one at the time that a watch exception would normally be taken, then the *WP* bit in the *Cause* register is set, and the exception is deferred until all three bits are zero. Software may use the *WP* bit in the *Cause* register to determine if the *EPC* register points at the instruction that caused the watch exception, or if the exception actually occurred while in kernel mode.

The Watch exception can occur on either an instruction fetch or a data access. Watch exceptions that occur on an instruction fetch have a higher priority than watch exceptions that occur on a data access.

Register ExcCode Value:

WATCH

Additional State Saved:

Table 5.12 Register States on Watch Exception

Register State	Value
$Cause_{WP}$	Indicates that the watch exception was deferred until after $Status_{EXL}$, $Status_{ERL}$, and $Debug_{DM}$ were zero. This bit directly causes a watch exception, so software must clear this bit as part of the exception handler to prevent a watch exception loop at the end of the current handler execution.
$WatchHi_{I,R,W}$	Set for the watch channel that matched, and indicates which type of match there was.

Entry Vector Used:

General exception vector (offset 0x180)

5.6.9 Address Error Exception — Instruction Fetch/Data Access

An address error exception occurs on an instruction or data access when an attempt is made to execute one of the following:

- Fetch an instruction, load a word, 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 mode
- Reference to a non-user address space when using the new EVA instructions

Note that in the case of an instruction fetch that is not aligned on a word boundary, PC is updated before the condition is detected. Therefore, both *EPC* and *BadVAddr* point to the unaligned instruction address. In the case of a data access the exception is taken if either an unaligned address or an address that was inaccessible in the current processor mode was referenced by a load or store instruction.

Cause Register ExcCode Value:

ADEL: Reference was a load or an instruction fetch

ADES: Reference was a store

Additional State Saved:

Table 5.13 CP0 Register States on Address Exception Error

Register State	Value
BadVAddr	Failing address
$Context_{VPN2}$	UNPREDICTABLE
$EntryHi_{VPN2}$	UNPREDICTABLE
EntryLo0	UNPREDICTABLE
EntryLo1	UNPREDICTABLE

Entry Vector Used:

General exception vector (offset 0x180)

5.6.10 TLB Refill Exception — Instruction Fetch or Data Access

During an instruction fetch or data access, a TLB refill exception occurs when no TLB entry matches a reference to a mapped address space and the *EXL* bit is 0 in the *Status* register. Note that this is distinct from the case in which an entry matches but has the valid bit off. In that case, a TLB Invalid exception occurs.

Cause Register ExcCode Value:

TLBL: Reference was a load or an instruction fetch

TLBS: Reference was a store

Additional State Saved:

Table 5.14 CP0 Register States on TLB Refill Exception

Register State	Value
BadVAddr	Failing address.
Context	The *BadVPN2* field contains $VA_{31:13}$ of the failing address.
EntryHi	The *VPN2* field contains $VA_{31:13}$ of the failing address; the *ASID* field contains the ASID of the reference that missed.
EntryLo0	UNPREDICTABLE
EntryLo1	UNPREDICTABLE

Entry Vector Used:

TLB refill vector (offset 0x000) if $Status_{EXL}$ = 0 at the time of exception;

General exception vector (offset 0x180) if $Status_{EXL}$ = 1 at the time of exception

5.6.11 TLB Invalid Exception — Instruction Fetch or Data Access

During an instruction fetch or data access, a TLB invalid exception occurs in one of the following cases:

- No TLB entry matches a reference to a mapped address space; and the *EXL* bit is 1 in the *Status* register.

- A TLB entry matches a reference to a mapped address space, but the matched entry has the valid bit off.

Cause Register ExcCode Value:

TLBL: Reference was a load or an instruction fetch

TLBS: Reference was a store

Additional State Saved:

Table 5.15 CP0 Register States on TLB Invalid Exception

Register State	Value
BadVAddr	Failing address
Context	The BadVPN2 field contains $VA_{31:13}$ of the failing address.
EntryHi	The VPN2 field contains $VA_{31:13}$ of the failing address; the ASID field contains the ASID of the reference that missed.
EntryLo0	UNPREDICTABLE
EntryLo1	UNPREDICTABLE

Entry Vector Used:

General exception vector (offset 0x180)

5.6.12 Cache Error Exception

A cache error exception occurs when an instruction or data reference detects a cache tag or data error. This exception is not maskable. Because the error was in a cache, the exception vector is to an unmapped, uncached address. This exception can be imprecise and the *ErrorEPC* may not point to the instruction that saw the error. Additionally, because the caches on the cores within the interAptiv MPS are coherent, cache errors detected on other cores could indicate data corruption for a process on this CPU. An error on another CPU will still cause a Cache Error exception, with the $CacheErr_{EE}$ indicating that the error occurred on another processor.

Instruction cache parity errors are precise and will only be taken if the core is going to execute the instruction that saw the parity error on a read of the instruction cache. If multiple instruction cache errors are detected prior to the exception being taken, the CacheErr register contents will capture the information for the most recent error, which may not correlate to the instruction indicated by ErrorEPC. Error handling code should use the CacheErr contents to process the exception. Upon returning to the instruction indicated by ErrorEPC, that error would be seen again.

For data cache parity or uncorrectable ECC errors, this exception can be imprecise and the ErrorEPC may not point to the instruction that saw the error.

L2 cache errors are considered to be imprecise. An L2 cache error on a data load operation can potentially corrupt the target GPR.

Cause Register ExcCode Value

N/A

Additional State Saved

Table 5.16 CP0 Register States on Cache Error Exception

Register State	Value
CacheErr	Error state
ErrorEPC	Restart PC

Entry Vector Used

Cache error vector (offset 0x100)

5.6.13 Bus Error Exception — Instruction Fetch or Data Access

A bus error exception occurs when an instruction or data access makes a bus request (due to a cache miss or an uncacheable reference) and that request terminates in an error. The bus error exception can occur on either an instruction fetch or a data read. Bus error exceptions cannot be generated on data writes. Bus error exceptions that occur on an instruction fetch have a higher priority than bus error exceptions that occur on a data access.

Instruction errors are precise, will data bus errors can be imprecise. These errors are taken when the ERR code is returned on the *OC_SResp* input.

Cause Register ExcCode Value:

IBE: Error on an instruction reference

DBE: Error on a data reference

Additional State Saved:

None

Entry Vector Used:

General exception vector (offset 0x180)

5.6.14 Debug Software Breakpoint Exception

A debug software breakpoint exception occurs when an SDBBP instruction is executed. The *DEPC* register and DBD bit in the *Debug* register will indicate the SDBBP instruction that caused the debug exception.

Debug Register Debug Status Bit Set:

DBp

Additional State Saved:

None

Entry Vector Used:

Debug exception vector

5.6.15 Execution Exception — System Call

The system call exception is one of the execution exceptions. All of these exceptions have the same priority. A system call exception occurs when a SYSCALL instruction is executed.

Cause Register ExcCode Value:

Sys

Additional State Saved:

None

Entry Vector Used:

General exception vector (offset 0x180)

5.6.16 Execution Exception — Breakpoint

The breakpoint exception is one of the execution exceptions. All of these exceptions have the same priority. A breakpoint exception occurs when a BREAK instruction is executed.

Cause Register ExcCode Value:

Bp

Additional State Saved:

None

Entry Vector Used:

General exception vector (offset 0x180)

5.6.17 Execution Exception — Reserved Instruction

The reserved instruction exception is one of the execution exceptions. All of these exceptions have the same priority. A reserved instruction exception occurs when a reserved or undefined major opcode or function field is executed. This includes Coprocessor 2 instructions which are decoded reserved in the Coprocessor 2.

Cause Register ExcCode Value:

RI

Additional State Saved:

None

Entry Vector Used:

General exception vector (offset 0x180)

5.6.18 Execution Exception — Coprocessor Unusable

The coprocessor unusable exception is one of the execution exceptions. All of these exceptions have the same priority. A coprocessor unusable exception occurs when an attempt is made to execute a coprocessor instruction for one of the following:

- a corresponding coprocessor unit that has not been marked usable by setting its CU bit in the *Status* register
- CP0 instructions, when the unit has not been marked usable, and the processor is executing in user mode

Cause Register ExcCode Value:

CpU

Additional State Saved:

Table 5.17 Register States on Coprocessor Unusable Exception

Register State	Value
$Cause_{CE}$	Unit number of the coprocessor being referenced

Entry Vector Used:

General exception vector (offset 0x180)

5.6.19 Execution Exception — CorExtend Block Unusable

The CorExtend block unusable exception is one of the execution exceptions. All of these exceptions have the same priority. A CEU exception occurs when an attempt is made to execute a CorExtend instruction when the CEE bit in the *Status* register is not set. It is dependent on the implementation of the CorExtend block, but this exception should be taken on any CorExtend instruction that modifies local state within the CorExtend block and can optionally be taken on other CorExtend instructions.

Cause Register ExcCode Value:

CEU

Additional State Saved:

None

Entry Vector Used:

General exception vector (offset 0x180)

5.6.20 Execution Exception — DSP ASE State Disabled

The DSP ASE State Disabled exception an execution exception. It occurs when an attempt is made to execute a DSP ASE instruction when the MX bit in the Status register is not set. This allows an OS to do “lazy” context switching.

Cause Register ExcCode Value:

DSPDis

Additional State Saved:

None

Entry Vector Used:

General exception vector (offset 0x180)

5.6.21 Execution Exception — Floating Point Exception

A floating point exception is initiated by the floating point coprocessor.

Cause Register ExcCode Value:

FPE

Additional State Saved:

Table 5.18 Register States on Floating Point Exception

Register State	Value
FCSR	Indicates the cause of the floating point exception

Entry Vector Used:

General exception vector (offset 0x180)

5.6.22 Execution Exception — Integer Overflow

The integer overflow exception is one of the execution exceptions. All of these exceptions have the same priority. An integer overflow exception occurs when selected integer instructions result in a 2's complement overflow.

Cause Register ExcCode Value:

Ov

Additional State Saved:

None

Entry Vector Used:

General exception vector (offset 0x180)

5.6.23 Execution Exception — Trap

The trap exception is one of the execution exceptions. All of these exceptions have the same priority. A trap exception occurs when a trap instruction results in a TRUE value.

Cause Register ExcCode Value:

Tr

Additional State Saved:

None

Entry Vector Used:

General exception vector (offset 0x180)

5.6.24 Debug Data Break Exception

A debug data break exception occurs when a data hardware breakpoint matches the load/store transaction of an executed load/store instruction. The *DEPC* register and *DBD* bit in the *Debug* register will indicate the load/store instruction that caused the data hardware breakpoint to match. The load/store instruction that caused the debug exception has not completed e.g. not updated the register file, and the instruction can be re-executed after returning from the debug handler.

Debug Register Debug Status Bit Set:

DDBL for a load instruction or *DDBS* for a store instruction

Additional State Saved:

None

Entry Vector Used:

Debug exception vector

5.6.25 TLB Modified Exception — Data Access

During a data access, a TLB modified exception occurs on a store reference to a mapped address if the following condition is true:

- The matching TLB entry is valid, but not dirty.

Cause Register ExcCode Value:

Mod

Additional State Saved:

Table 5.19 Register States on TLB Modified Exception

Register State	Value
BadVAddr	Failing address
Context	The BadVPN2 field contains $VA_{31:13}$ of the failing address.
EntryHi	The VPN2 field contains $VA_{31:13}$ of the failing address; the ASID field contains the ASID of the reference that missed.
EntryLo0	UNPREDICTABLE
EntryLo1	UNPREDICTABLE

Entry Vector Used:

General exception vector (offset 0x180)

5.7 Exception Handling and Servicing Flowcharts

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

- General exceptions
- TLB miss exceptions
- Reset and NMI exceptions
- Debug exceptions

Generally speaking, exceptions are handled by hardware and then serviced by software. Note that unexpected debug exceptions to the debug exception vector at 0xBFC0_0200 may be viewed as a reserved instruction since uncontrolled execution of an SDBBP instruction caused the exception. The DERET instruction must be used at return from the debug exception handler, in order to leave debug mode and return to non-debug mode. The DERET instruction returns to the address in the *DEPC* register.

Figure 5.2 General Exception Handler (HW)

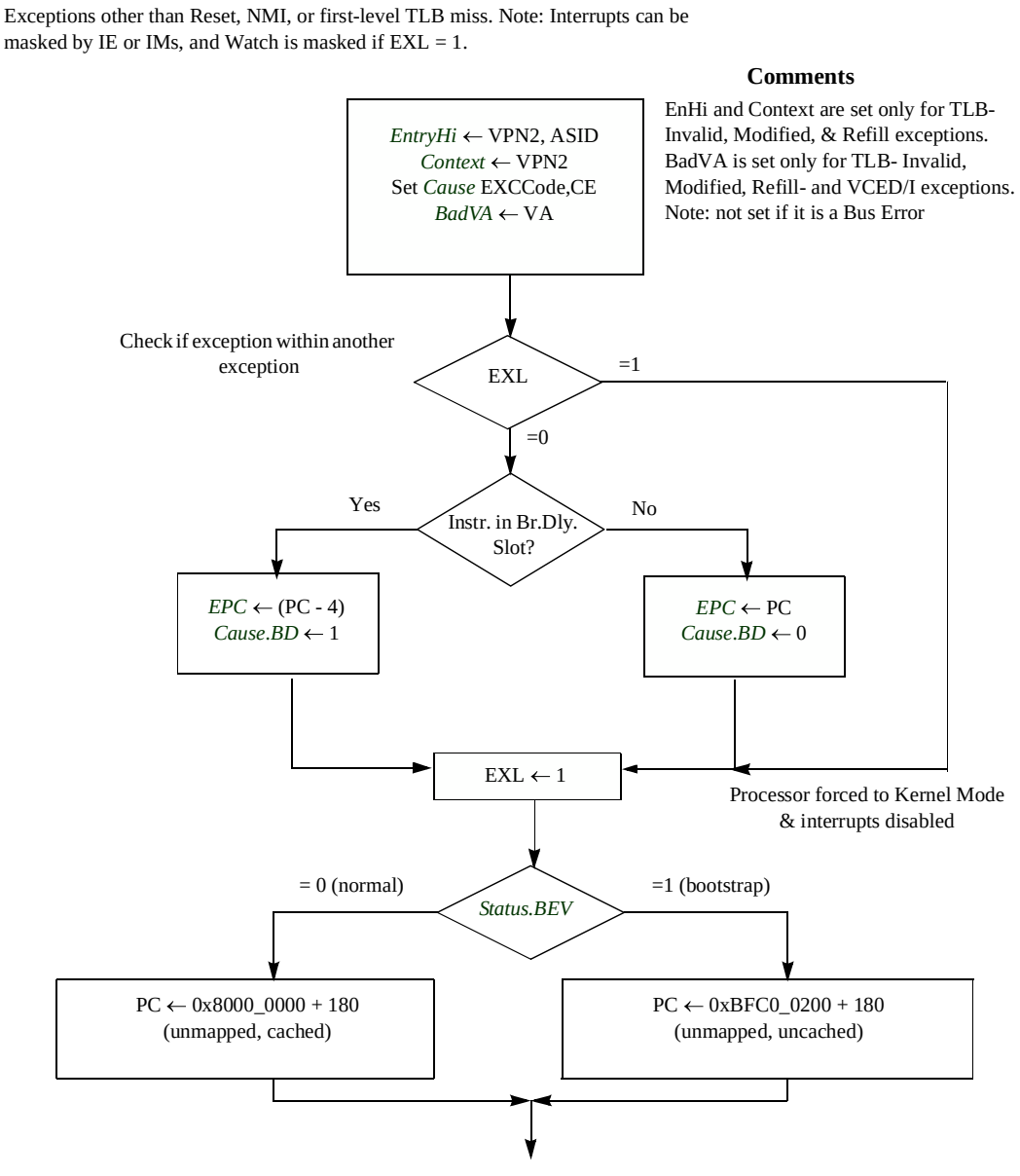


Figure 5.3 General Exception Servicing Guidelines (SW)

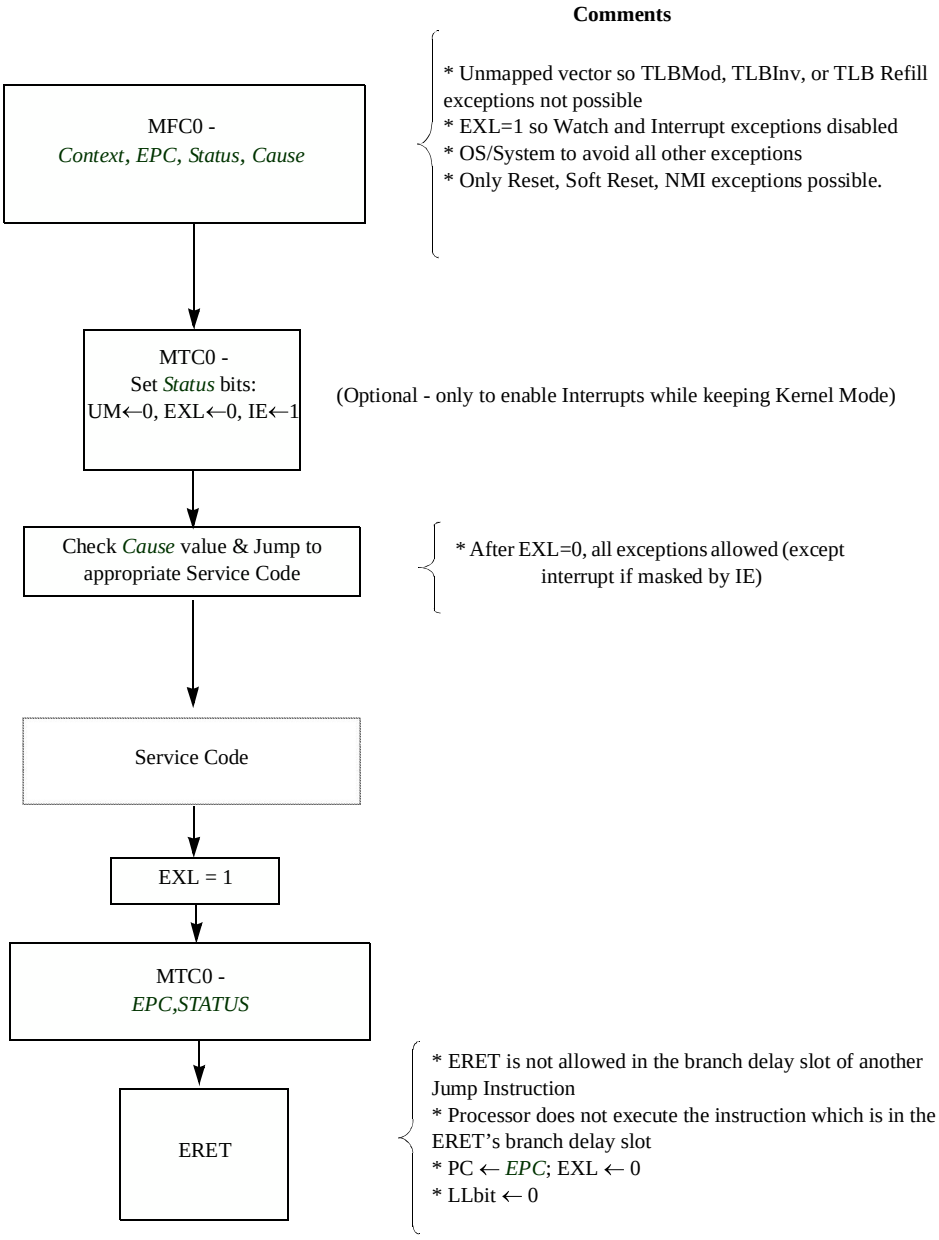


Figure 5.4 TLB Miss Exception Handler (HW)

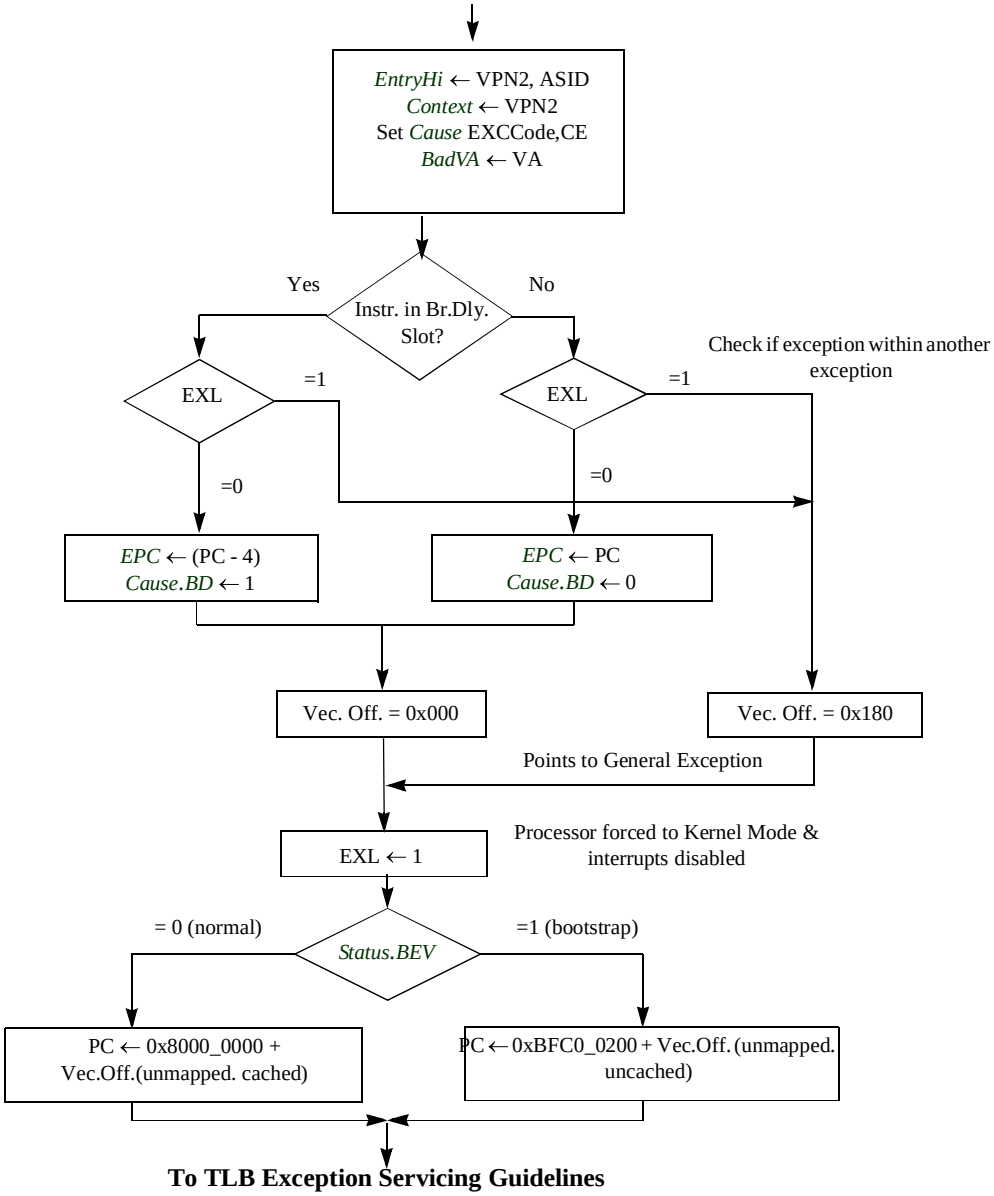


Figure 5.5 TLB Exception Servicing Guidelines (SW)

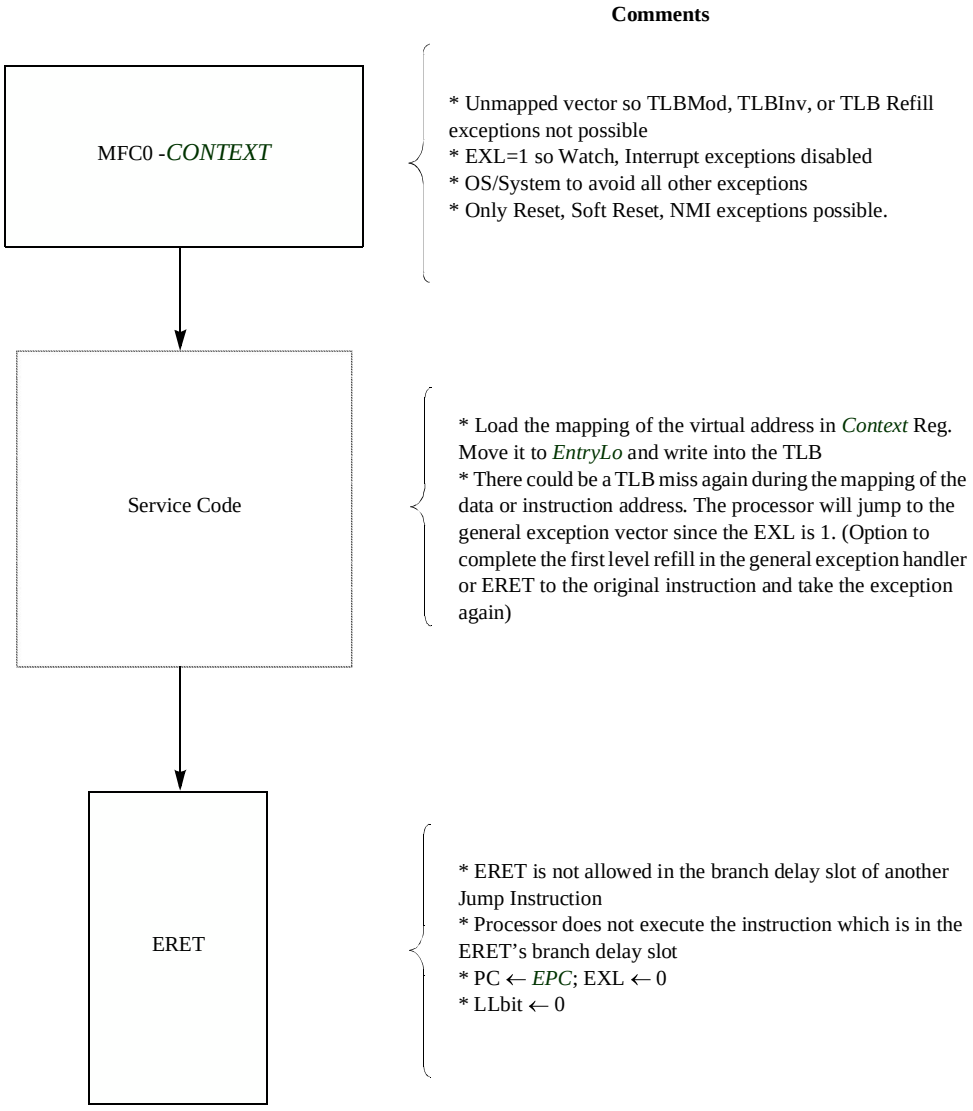
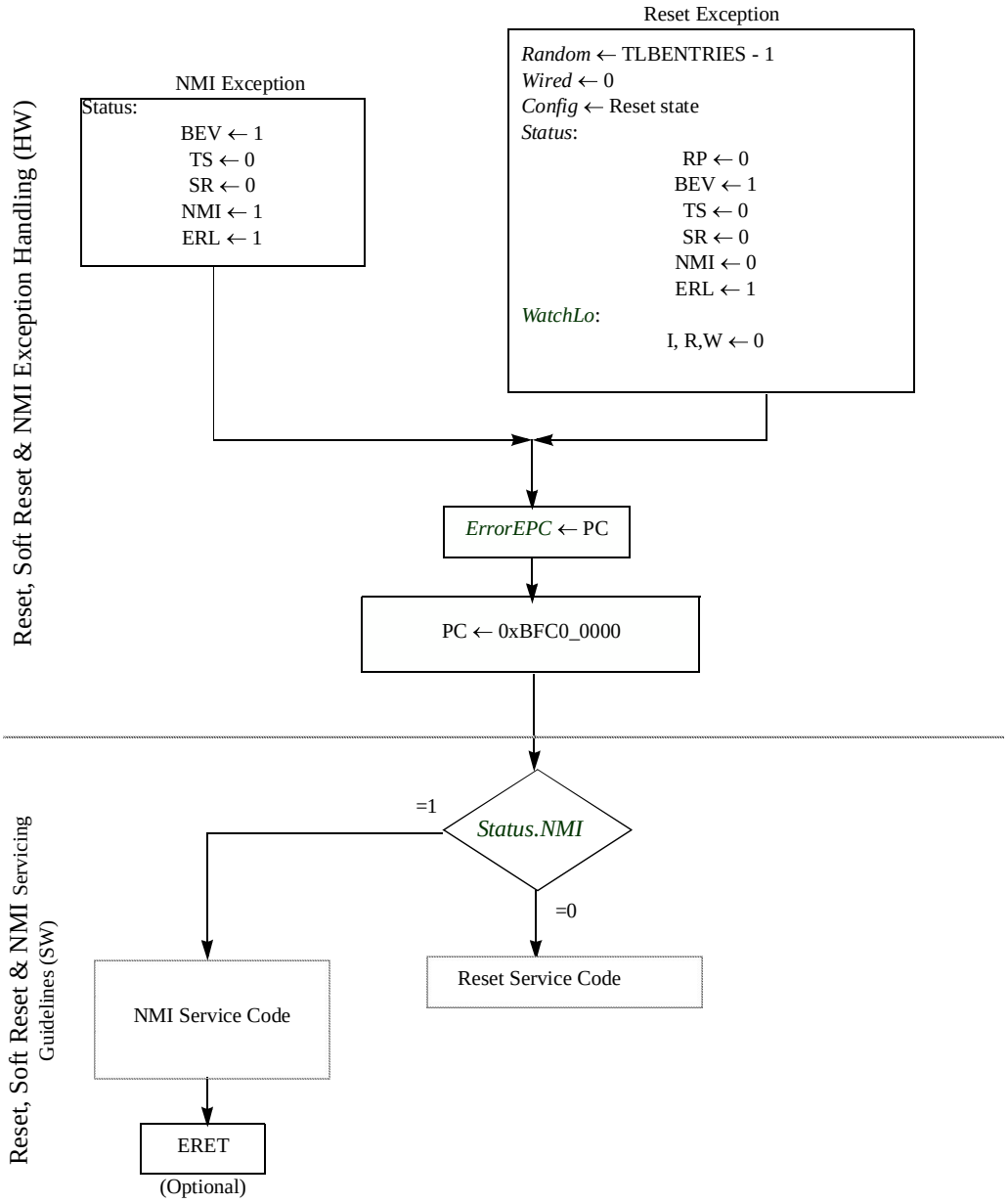


Figure 5.6 Reset and NMI Exception Handling and Servicing Guidelines



5.8 Interrupts

In the MIPS32® Release 1 architecture, support for exceptions included two software interrupts, six hardware interrupts, and a special-purpose timer interrupt. The timer interrupt was provided external to the CPU and was typically combined with hardware interrupt 5 in a system-dependent manner. Interrupts were handled either through the general exception vector (offset 0x180) or the special interrupt vector (0x200), based on the value of *CauseIV*. Software was required to prioritize interrupts as a function of the $Cause_{IV}$ bits in the interrupt handler prologue.

Release 2 of the Architecture, implemented by the interAptiv CPU, adds a number of upward-compatible extensions to the Release 1 interrupt architecture, including support for vectored interrupts and the implementation of a new interrupt mode that permits the use of an external interrupt controller.

Additionally, internal performance counters have been added to the interAptiv CPU. These counters can be configured to count various events within the CPU. When the MSB of the counter is set, it can trigger a performance counter interrupt. This interrupt, like the timer interrupt, is an output from the CPU that can be brought back into the CPU's interrupt pins in a system-dependent manner.

The Fast Debug Channel feature in EJTAG provides a low overhead means for sending data between CPU software and the EJTAG probe. It includes a pair of FIFOs for transmit and receive data. Software can define FIFO thresholds for generating an interrupt. The fast debug channel interrupt is also routed similarly to the timer and performance counter interrupts. The interrupt status is made available on an output pin and can be brought back into the CPU's interrupt pins.

5.8.1 Interrupt Modes

The interAptiv CPU includes support for three interrupt modes, as defined by Release 3 of the Architecture:

- Interrupt Compatibility mode, in which the behavior of the interAptiv is identical to the behavior of an implementation of Release 1 of the Architecture.
- Vectored Interrupt (VI) mode adds the ability to prioritize and vector interrupts to a handler dedicated to that interrupt. The presence of this mode is denoted by the *VInt* bit in the *Config3* register. Although this mode is architecturally optional, it is always present on the interAptiv CPU, so the *VInt* bit will always read as a 1.
- External Interrupt Controller (EIC) mode, which redefines the way interrupts are handled to provide full support for an external interrupt controller that handles prioritization and vectoring of interrupts. As with VI mode, this mode is architecturally optional. The presence of this mode is denoted by the *VEIC* bit in the *Config3* register. On the interAptiv CPU, the *VEIC* bit is set externally by the static input, *SI_EICPresent*, to allow system logic to indicate the presence of an external interrupt controller.

Following reset, the interAptiv processor defaults to Compatibility mode, which is fully compatible with all implementations of Release 1 of the Architecture.

Table 5.20 shows the current interrupt mode of the processor as a function of the Coprocessor 0 register fields that can affect the mode.

Table 5.20 Interrupt Modes

StatusBEV	*CauseIV*	*IntCtlVS*	*Config3VINT*	*Config3VEIC*	**Interrupt Mode**
1	x	x	x	x	Compatibility
x	0	x	x	x	Compatibility
x	x	=0	x	x	Compatibility
0	1	≠0	1	0	Vectored Interrupt
0	1	≠0	x	1	External Interrupt Controller
0	1	≠0	0	0	Cannot occur because *IntCtl* $_{VS}$ cannot be non-zero if neither Vectored Interrupt nor External Interrupt Controller mode is implemented.
“x” denotes don’t care					

5.8.1.1 Interrupt Compatibility Mode

This is the default interrupt mode for the processor and is entered when a Reset exception occurs. In this mode, interrupts are non-vectored and dispatched though exception vector offset 0x180 (if *Cause*$_{IV}$ = 0) or vector offset 0x200 (if *Cause* $_{IV}$ = 1). This mode is in effect when any of the following conditions are true:

- *Cause*$_{IV}$ = 0
- *Status*$_{BEV}$ = 1
- *IntCtl*$_{VS}$ = 0, which is the case if vectored interrupts are not implemented or have been disabled.

Here is a typical software handler for compatibility mode:

```
/*
 * Assumptions:
 *  - CauseIV = 1 (if it were zero, the interrupt exception would have to
 *                 be isolated from the general exception vector before arriving
 *                 here)
 *  - GPRs k0 and k1 are available
 *  - The software priority is IP7:IP0 (HW5:HW0, SW1:SW0)
 *
 * Location: Offset 0x200 from exception base
 */

IVexception:
   mfc0   k0, C0_Cause        /* Read Cause register for IP bits */
   mfc0   k1, C0_Status       /* and Status register for IM bits */
   andi   k0, k0, M_CauseIM   /* Keep only IP bits from Cause */
   and    k0, k0, k1          /* and mask with IM bits */
   beq    k0, zero, Dismiss   /* no bits set - spurious interrupt */
   clz    k0, k0              /* Find first bit set, IP7:IP0; k0 = 16:23 */
   xori   k0, k0, 0x17        /* 16:23 => 7:0 */
   sll    k0, k0, VS          /* Shift to emulate software IntCtlVS */
```

```
    la     k1, VectorBase      /* Get base of 8 interrupt vectors */
    addu   k0, k0, k1          /* Compute target from base and offset */
    jr     k0                  /* Jump to specific exception routine */
    nop

/*
 * Each interrupt processing routine processes a specific interrupt, analogous
 * to those reached in VI or EIC interrupt mode. Since each processing routine
 * is dedicated to a particular interrupt line, it has the context to know
 * which line was asserted.  Each processing routine may need to look further
 * to determine the actual source of the interrupt if multiple interrupt requests
 * are ORed together on a single IP line. Once that task is performed, the
 * interrupt may be processed in one of two ways:
 *
 * - Completely at interrupt level (e.g., a simple UART interrupt). The
 *   SimpleInterrupt routine below is an example of this type.
 * - By saving sufficient state and re-enabling other interrupts. In this
 *   case the software model determines which interrupts are disabled during
 *   the processing of this interrupt. Typically, this is either the single
 *   StatusIM bit that corresponds to the interrupt being processed, or some
 *   collection of other StatusIM bits so that "lower" priority interrupts are
 *    also disabled. The NestedInterrupt routine below is an example of this type.
 */

SimpleInterrupt:
/*
 * Process the device interrupt here and clear the interupt request
 * at the device. In order to do this, some registers may need to be
 * saved and restored. The coprocessor 0 state is such that an ERET
 * will simply return to the interrupted code.
 */
    eret                       /* Return to interrupted code */

NestedException:
/*
 * Nested exceptions typically require saving the EPC and Status registers,
 * saving any GPRs that may be modified by the nested exception routine, disabling
 * the appropriate IM bits in Status to prevent an interrupt loop, putting
 * the processor in kernel mode, and re-enabling interrupts. The sample code
 * below cannot cover all nuances of this processing and is intended only
 * to demonstrate the concepts.
 */

    /* Save GPRs here, and setup software context */
    mfc0   k0, C0_EPC          /* Get restart address */
    sw     k0, EPCSave         /* Save in memory */
    mfc0   k0, C0_Status       /* Get Status value */
    sw     k0, StatusSave      /* Save in memory */
    li     k1, ~IMbitsToClear  /* Get IM bits to clear for this interrupt */
                               /*   this must include at least the IM bit */
                               /*   for the current interrupt, and may include */
                               /*   others */
    and    k0, k0, k1             /* Clear bits in copy of Status */
    ins    k0, zero, S_StatusEXL, (W_StatusKSU+W_StatusERL+W_StatusEXL)
                                  /* Clear KSU, ERL, EXL bits in k0 */
    mtc0   k0, C0_Status          /* Modify mask, switch to kernel mode, */
                                  /*   re-enable interrupts */
```

```
    /*
     * Process interrupt here, including clearing device interrupt.
     * In some environments this may be done with a thread running in
     * kernel or user mode. Such an environment is well beyond the scope of
     * this example.
     */

/*
 * To complete interrupt processing, the saved values must be restored
 * and the original interrupted code restarted.
 */

    di                          /* Disable interrupts - may not be required */
    lw     k0, StatusSave       /* Get saved Status (including EXL set) */
    lw     k1, EPCSave          /*   and EPC */
    mtc0   k0, C0_Status        /* Restore the original value */
    mtc0   k1, C0_EPC           /*  and EPC */
    /* Restore GPRs and software state */
    eret                        /* Dismiss the interrupt */
```

5.8.1.2 Vectored Interrupt Mode

In Vectored Interrupt (VI) mode, a priority encoder prioritizes pending interrupts and generates a vector which can be used to direct each interrupt to a dedicated handler routine. VI mode is in effect when all the following conditions are true:

- $Config3_{VInt} = 1$
- $Config3_{VEIC} = 0$
- $IntCtl_{VS} \neq 0$
- $Cause_{IV} = 1$
- $Status_{BEV} = 0$

In VI interrupt mode, the six hardware interrupts are interpreted as individual hardware interrupt requests. The timer, performance counter, and fast debug channel interrupts are combined in a system-dependent way (external to the CPU) with the hardware interrupts (the interrupt with which they are combined is indicated by the $IntCtl_{IPTI/IPCI/IPFDCI}$ fields) to provide the appropriate relative priority of the those interrupts with that of the hardware interrupts. The processor interrupt logic ANDs each of the $Cause_{IP}$ bits with the corresponding $Status_{IM}$ bits. If any of these values is 1,

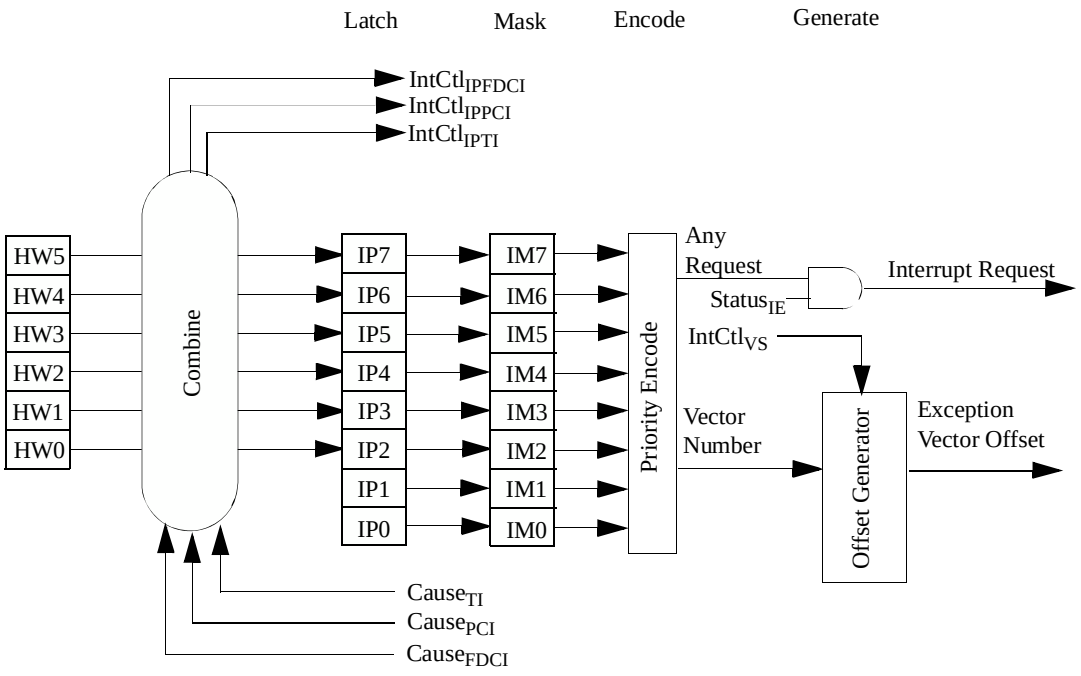
and if interrupts are enabled ($Status_{IE}$ = 1, $Status_{EXL}$ = 0, and $Status_{ERL}$ = 0), an interrupt is signaled and a priority encoder scans the values in the order shown in Table 5.21.

Table 5.21 Relative Interrupt Priority for Vectored Interrupt Mode

Relative Priority	Interrupt Type	Interrupt Source	Interrupt Request Calculated From	Vector Number Generated by Priority Encoder
Highest Priority	Hardware	HW5	IP7 and IM7	7
		HW4	IP6 and IM6	6
		HW3	IP5 and IM5	5
		HW2	IP4 and IM4	4
		HW1	IP3 and IM3	3
		HW0	IP2 and IM2	2
	Software	SW1	IP1 and IM1	1
Lowest Priority		SW0	IP0 and IM0	0

The priority order places a relative priority on each hardware interrupt and places the software interrupts at a priority lower than all hardware interrupts. When the priority encoder finds the highest priority pending interrupt, it outputs an encoded vector number that is used in the calculation of the handler for that interrupt, as described below. This is shown pictorially in Figure 5.7.

Figure 5.7 Interrupt Generation for Vectored Interrupt Mode



A typical software handler for Vectored Interrupt mode bypasses the entire sequence of code following the `IVexception` label shown for the compatibility mode handler above. Instead, the hardware performs the prioritization, dispatching directly to the interrupt processing routine.

A nested interrupt is similar to that shown for compatibility mode. Such a routine might look as follows:

```
NestedException:
/*
* Nested exceptions typically require saving the EPC and Status registers,
* disabling the appropriate IM bits in Status to prevent an interrupt loop,
* putting the processor in kernel mode, and re-enabling interrupts. The sample
* code below cannot cover all nuances of this processing and is intended only
* to demonstrate the concepts.
*/
   mfc0   k0, C0_EPC          /* Get restart address */
   sw     k0, EPCSave         /* Save in memory */
   mfc0   k0, C0_Status       /* Get Status value */
   sw     k0, StatusSave      /* Save in memory */
   li     k1, ~IMbitsToClear  /* Get IM bits to clear for this interrupt */
                              /*   this must include at least the IM bit */
                              /*   for the current interrupt, and may include */
                              /*   others */
   and    k0, k0, k1             /* Clear bits in copy of Status */
   ins    k0, zero, S_StatusEXL, (W_StatusKSU+W_StatusERL+W_StatusEXL)
                                 /* Clear KSU, ERL, EXL bits in k0 */
   mtc0   k0, C0_Status          /* Modify mask, switch to kernel mode, */
                                 /*   re-enable interrupts */

   /* Process interrupt here, including clearing device interrupt */

/*
 * To complete interrupt processing, the saved values must be restored
 * and the original interrupted code restarted.
 */

   di                         /* Disable interrupts - may not be required */
   lw     k0, StatusSave      /* Get saved Status (including EXL set) */
   lw     k1, EPCSave         /*   and EPC */
   mtc0   k0, C0_Status       /* Restore the original value */
   mtc0   k1, C0_EPC          /*  and EPC */
   ehb                        /* Clear hazard */
   eret                       /* Dismiss the interrupt */
```

5.8.1.3 External Interrupt Controller Mode

External Interrupt Controller (EIC) mode redefines the way that the processor interrupt logic is configured to provide support for an external interrupt controller. The interrupt controller is responsible for prioritizing all interrupts, including hardware, software, timer, fast debug channel, and performance counter interrupts, and directly supplying to the processor the vector number of the highest priority interrupt.

EIC interrupt mode is in effect if all of the following conditions are true:

- $Config3_{VEIC} = 1$
- $IntCtl_{VS} \neq 0$
- $Cause_{IV} = 1$
- $Status_{BEV} = 0$

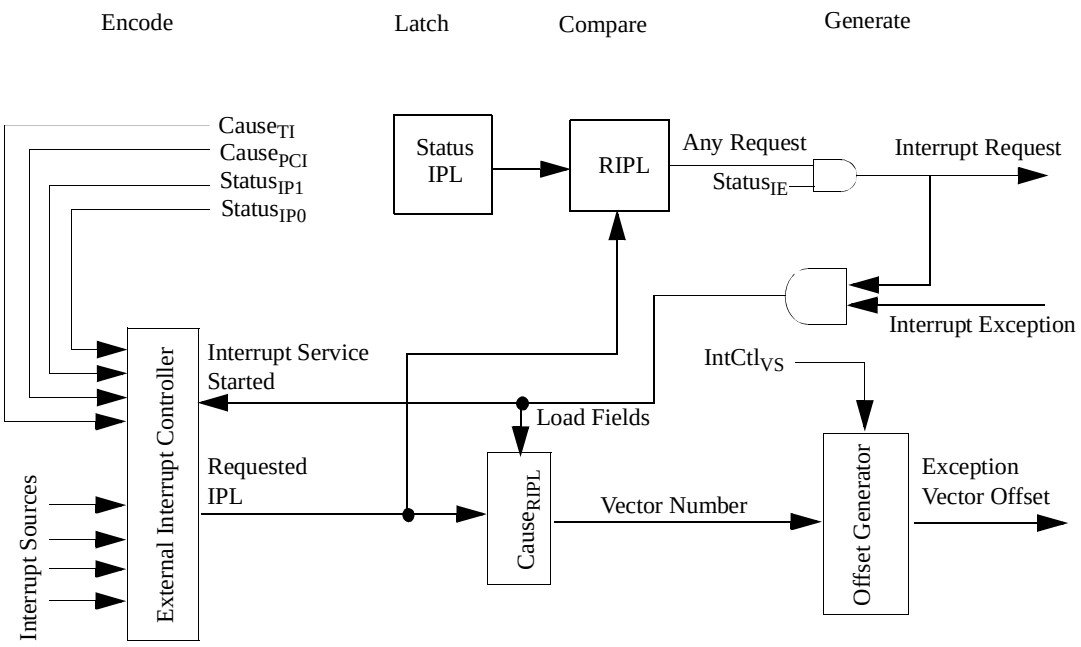
In EIC mode, the processor sends the state of the software interrupt requests ($Cause_{IP1:IP0}$) and the timer, performance counter, and fast debug channel interrupt requests ($Cause_{TI/PCI/FDCI}$) to the external interrupt controller, which prioritizes these interrupts in a system-dependent way with other hardware interrupts. The interrupt controller can be a hardwired logic block, or it can be configurable by control and status registers. This allows the interrupt controller to be more specific or more general as a function of the system environment and needs.

The external interrupt controller prioritizes its interrupt requests and produces the vector number of the highest priority interrupt to be serviced. The vector number, called the Requested Interrupt Priority Level (RIPL), is a 6-bit encoded value in the range 0:63, inclusive. The values 1:63 represent the lowest (1) to highest (63) RIPL for the interrupt to be serviced. A value of 0 indicates that no interrupt requests are pending. The interrupt controller inputs this value on the 6 hardware interrupt lines, which are treated as an encoded value in EIC mode.

$Status_{IPL}$ (which overlays $Status_{IM7:IM2}$) is interpreted as the Interrupt Priority Level (IPL) at which the processor is currently operating (a value of zero indicates that no interrupt is currently being serviced). When the interrupt controller requests service for an interrupt, the processor compares RIPL with $Status_{IPL}$ to determine if the requested interrupt has a higher priority than the current IPL. If RIPL is strictly greater than $Status_{IPL}$, and interrupts are enabled ($Status_{IE}$ = 1, $Status_{EXL}$ = 0, and $Status_{ERL}$ = 0), an interrupt request is signaled to the pipeline. When the processor starts the interrupt exception, it loads RIPL into $Cause_{RIPL}$ (which overlays $Cause_{IP7:IP2}$) and signals the external interrupt controller to notify it that the request is being serviced. The interrupt exception uses the value of $Cause_{RIPL}$ as the vector number. Because $Cause_{RIPL}$ is only loaded by the processor when an interrupt exception is signaled, it is available to software during interrupt processing.

The operation of EIC interrupt mode is shown in Figure 5.8.

Figure 5.8 Interrupt Generation for External Interrupt Controller Interrupt Mode



A typical software handler for EIC mode bypasses the entire sequence of code following the `IV exception` label shown for the compatibility-mode handler above. Instead, the hardware performs the prioritization, dispatching directly to the interrupt processing routine.

A nested interrupt is similar to that shown for compatibility mod. It also need only copy *Cause*$_{RIPL}$ to *Status*$_{IPL}$ to prevent lower priority interrupts from interrupting the handler. Here is an example of such a routine:

```
NestedException:
/*
* Nested exceptions typically require saving the EPC and Status registers,
* disabling the appropriate IM bits in Status to prevent an interrupt loop,
* putting the processor in kernel mode, and re-enabling interrupts.
* The sample code below can not cover all nuances of this processing and is
* intended only to demonstrate the concepts.
*/

   mfc0   k1, C0_Cause        /* Read Cause to get RIPL value */
   mfc0   k0, C0_EPC          /* Get restart address */
   srl    k1, k1, S_CauseRIPL /* Right justify RIPL field */
   sw     k0, EPCSave         /* Save in memory */
   mfc0   k0, C0_Status       /* Get Status value */
   sw     k0, StatusSave      /* Save in memory */
   ins    k0, k1, S_StatusIPL, 6 /* Set IPL to RIPL in copy of Status */
   ins    k0, zero, S_StatusEXL, (W_StatusKSU+W_StatusERL+W_StatusEXL)
                              /* Clear KSU, ERL, EXL bits in k0 */
   mtc0   k0, C0_Status       /* Modify IPL, switch to kernel mode, */
                              /*   re-enable interrupts */

   /* Process interrupt here, including clearing device interrupt */

/*
 * The interrupt completion code is identical to that shown for VI mode above.
 */
```

5.8.2 Generation of Exception Vector Offsets for Vectored Interrupts

For vectored interrupts (in either VI or EIC interrupt mode), a vector number is produced by the interrupt control logic. This number is combined with *IntCtl*$_{VS}$ to create the interrupt offset, which is added to 0x200 to create the exception vector offset. For VI mode, the vector number is in the range 0:7, inclusive. For EIC interrupt mode, the vector number is in the range 1:63, inclusive (0 being the encoding for “no interrupt”). The *IntCtl*$_{VS}$ field specifies the spacing between vector locations. If this value is zero (the default reset state), the vector spacing is zero and the processor reverts to Interrupt Compatibility mode. A non-zero value enables vectored interrupts. Table 5.22 shows the exception vector offset for a representative subset of the vector numbers and values of the *IntCtl*$_{VS}$ field.

Table 5.22 Exception Vector Offsets for Vectored Interrupts

Vector Number	Value of IntCtl$_{VS}$ Field				
	00001_2	00010_2	00100_2	01000_2	10000_2
0	0x0200	0x0200	0x0200	0x0200	0x0200
1	0x0220	0x0240	0x0280	0x0300	0x0400
2	0x0240	0x0280	0x0300	0x0400	0x0600
3	0x0260	0x02C0	0x0380	0x0500	0x0800
4	0x0280	0x0300	0x0400	0x0600	0x0A00
5	0x02A0	0x0340	0x0480	0x0700	0x0C00
6	0x02C0	0x0380	0x0500	0x0800	0x0E00

Table 5.22 Exception Vector Offsets for Vectored Interrupts*(continued)*

Vector Number	Value of $IntCtl_{VS}$ Field				
	00001_2	00010_2	00100_2	01000_2	10000_2
7	0x02E0	0x03C0	0x0580	0x0900	0x1000
	• • •				
61	0x09A0	0x1140	0x2080	0x3F00	0x7C00
62	0x09C0	0x1180	0x2100	0x4000	0x7E00
63	0x09E0	0x11C0	0x2180	0x4100	0x8000

The general equation for the exception vector offset for a vectored interrupt is:

$vectorOffset \leftarrow 0x200 + (vectorNumber \times (IntCtl_{VS} \parallel 0b00000))$

5.8.3 Global Interrupt Controller

The Global Interrupt Controller (GIC) handles the routing and masking of local interrupts, such as the timer, performance counter, fast debug channel interrupts, inter-processor interrupts, and external interrupts. This block can be configured to support various numbers of external interrupts and to support any of the CPU interrupt modes.

An interactive GUI is available to simplify the setup of desired event-routing through the GIC. The tool outputs a C-language function covering all required programming registers of the GIC.