MIPS32® Architecture For Programmers Volume III: The MIPS32® Privileged Resource Architecture

Document Number: MD00090
Revision 2.62
January 02, 2009

MIPS Technologies, Inc.
1225 Charleston Road
Mountain View, CA 94043-1353

Template: nB1.03, Built with tags: 2B ARCH MIPS32

Contents

Figures

Tables

Chapter 1

About This Book

The MIPS32® Architecture For Programmers Volume III: The MIPS32® Privileged Resource Architecture comes as a multi-volume set.

- Volume I describes conventions used throughout the document set, and provides an introduction to the MIPS32® Architecture

- Volume II provides detailed descriptions of each instruction in the MIPS32® instruction set

- Volume III describes the MIPS32® Privileged Resource Architecture which defines and governs the behavior of the privileged resources included in a MIPS32® processor implementation

- Volume IV-a describes the MIPS16e™ Application-Specific Extension to the MIPS32® Architecture

- Volume IV-b describes the MDMX™ Application-Specific Extension to the MIPS32® Architecture and is not applicable to the MIPS32® document set

- Volume IV-c describes the MIPS-3D® Application-Specific Extension to the MIPS32® Architecture

- Volume IV-d describes the SmartMIPS®Application-Specific Extension to the MIPS32® Architecture

1.1 Typographical Conventions

This section describes the use of *italic*, **bold** and `courier` fonts in this book.

1.1.1 Italic Text

- is used for *emphasis*

- is used for *bits*, *fields*, *registers*, that are important from a software perspective (for instance, address bits used by software, and programmable fields and registers), and various *floating point instruction formats*, such as *S*, *D*, and *PS*

- is used for the memory access types, such as *cached* and *uncached*

1.1.2 Bold Text

- represents a term that is being **defined**

- is used for **bits** and **fields** that are important from a hardware perspective (for instance, **register** bits, which are not programmable but accessible only to hardware)

- is used for ranges of numbers; the range is indicated by an ellipsis. For instance, **5..1** indicates numbers 5 through 1

- is used to emphasize **UNPREDICTABLE** and **UNDEFINED** behavior, as defined below.

1.1.3 Courier Text

`Courier` fixed-width font is used for text that is displayed on the screen, and for examples of code and instruction pseudocode.

1.2 UNPREDICTABLE and UNDEFINED

The terms **UNPREDICTABLE** and **UNDEFINED** are used throughout this book to describe the behavior of the processor in certain cases. **UNDEFINED** behavior or operations can occur only as the result of executing instructions in a privileged mode (i.e., in Kernel Mode or Debug Mode, or with the CP0 usable bit set in the Status register). Unprivileged software can never cause **UNDEFINED** behavior or operations. Conversely, both privileged and unprivileged software can cause **UNPREDICTABLE** results or operations.

1.2.1 UNPREDICTABLE

UNPREDICTABLE results may vary from processor implementation to implementation, instruction to instruction, or as a function of time on the same implementation or instruction. Software can never depend on results that are **UNPREDICTABLE**. **UNPREDICTABLE** operations may cause a result to be generated or not. If a result is generated, it is **UNPREDICTABLE**. **UNPREDICTABLE** operations may cause arbitrary exceptions.

UNPREDICTABLE results or operations have several implementation restrictions:

- Implementations of operations generating **UNPREDICTABLE** results must not depend on any data source (memory or internal state) which is inaccessible in the current processor mode

- **UNPREDICTABLE** operations must not read, write, or modify the contents of memory or internal state which is inaccessible in the current processor mode. For example, **UNPREDICTABLE** operations executed in user mode must not access memory or internal state that is only accessible in Kernel Mode or Debug Mode or in another process

- **UNPREDICTABLE** operations must not halt or hang the processor

1.2.2 UNDEFINED

UNDEFINED operations or behavior may vary from processor implementation to implementation, instruction to instruction, or as a function of time on the same implementation or instruction. **UNDEFINED** operations or behavior may vary from nothing to creating an environment in which execution can no longer continue. **UNDEFINED** operations or behavior may cause data loss.

UNDEFINED operations or behavior has one implementation restriction:

- **UNDEFINED** operations or behavior must not cause the processor to hang (that is, enter a state from which there is no exit other than powering down the processor). The assertion of any of the reset signals must restore the processor to an operational state

1.2.3 UNSTABLE

UNSTABLE results or values may vary as a function of time on the same implementation or instruction. Unlike **UNPREDICTABLE** values, software may depend on the fact that a sampling of an **UNSTABLE** value results in a legal transient value that was correct at some point in time prior to the sampling.

UNSTABLE values have one implementation restriction:

- Implementations of operations generating **UNSTABLE** results must not depend on any data source (memory or internal state) which is inaccessible in the current processor mode

1.3 Special Symbols in Pseudocode Notation

In this book, algorithmic descriptions of an operation are described as pseudocode in a high-level language notation resembling Pascal. Special symbols used in the pseudocode notation are listed in Table 1.1.

Table 1.1 Symbols Used in Instruction Operation Statements

Symbol	Meaning
$\leftarrow$	Assignment
=, ≠	Tests for equality and inequality
$\|\|$	Bit string concatenation
x^y	A *y*-bit string formed by *y* copies of the single-bit value *x*
b#n	A constant value *n* in base *b*. For instance 10#100 represents the decimal value 100, 2#100 represents the binary value 100 (decimal 4), and 16#100 represents the hexadecimal value 100 (decimal 256). If the "b#" prefix is omitted, the default base is 10.
0bn	A constant value *n* in base *2*. For instance 0b100 represents the binary value 100 (decimal 4).
0xn	A constant value *n* in base *16*. For instance 0x100 represents the hexadecimal value 100 (decimal 256).
$x_{y..z}$	Selection of bits *y* through *z* of bit string *x*. Little-endian bit notation (rightmost bit is 0) is used. If *y* is less than *z*, this expression is an empty (zero length) bit string.
+, −	2’s complement or floating point arithmetic: addition, subtraction
*, ×	2’s complement or floating point multiplication (both used for either)
div	2’s complement integer division
mod	2’s complement modulo
/	Floating point division
<	2’s complement less-than comparison
>	2’s complement greater-than comparison
≤	2’s complement less-than or equal comparison
≥	2’s complement greater-than or equal comparison
nor	Bitwise logical NOR
xor	Bitwise logical XOR
and	Bitwise logical AND
or	Bitwise logical OR

Table 1.1 Symbols Used in Instruction Operation Statements (Continued)

Symbol	Meaning
GPRLEN	The length in bits (32 or 64) of the CPU general-purpose registers
GPR[x]	CPU general-purpose register *x*. The content of *GPR[0]* is always zero. In Release 2 of the Architecture, GPR[x] is a short-hand notation for *SGPR[SRSCtl$_{CSS}$, x]*.
SGPR[s,x]	In Release 2 of the Architecture, multiple copies of the CPU general-purpose registers may be implemented. *SGPR[s,x]* refers to GPR set *s*, register *x*.
FPR[x]	Floating Point operand register *x*
FCC[CC]	Floating Point condition code CC. *FCC[0]* has the same value as *COC[1]*.
FPR[x]	Floating Point (Coprocessor unit 1), general register *x*
CPR[z,x,s]	Coprocessor unit *z*, general register *x*, select *s*
CP2CPR[x]	Coprocessor unit 2, general register *x*
CCR[z,x]	Coprocessor unit *z*, control register *x*
CP2CCR[x]	Coprocessor unit 2, control register *x*
COC[z]	Coprocessor unit *z* condition signal
Xlat[x]	Translation of the MIPS16e GPR number *x* into the corresponding 32-bit GPR number
BigEndianMem	Endian mode as configured at chip reset (0 →Little-Endian, 1 → Big-Endian). Specifies the endianness of the memory interface (see LoadMemory and StoreMemory pseudocode function descriptions), and the endianness of Kernel and Supervisor mode execution.
BigEndianCPU	The endianness for load and store instructions (0 → Little-Endian, 1 → Big-Endian). In User mode, this endianness may be switched by setting the *RE* bit in the *Status* register. Thus, BigEndianCPU may be computed as (BigEndianMem XOR ReverseEndian).
ReverseEndian	Signal to reverse the endianness of load and store instructions. This feature is available in User mode only, and is implemented by setting the *RE* bit of the *Status* register. Thus, ReverseEndian may be computed as (SR$_{RE}$ and User mode).
LLbit	Bit of **virtual** state used to specify operation for instructions that provide atomic read-modify-write. *LLbit* is set when a linked load occurs and is tested by the conditional store. It is cleared, during other CPU operation, when a store to the location would no longer be atomic. In particular, it is cleared by exception return instructions.
I:, **I+n**:, **I-n**:	This occurs as a prefix to *Operation* description lines and functions as a label. It indicates the instruction time during which the pseudocode appears to "execute." Unless otherwise indicated, all effects of the current instruction appear to occur during the instruction time of the current instruction. No label is equivalent to a time label of **I**. Sometimes effects of an instruction appear to occur either earlier or later — that is, during the instruction time of another instruction. When this happens, the instruction operation is written in sections labeled with the instruction time, relative to the current instruction **I**, in which the effect of that pseudocode appears to occur. For example, an instruction may have a result that is not available until after the next instruction. Such an instruction has the portion of the instruction operation description that writes the result register in a section labeled **I+1**. The effect of pseudocode statements for the current instruction labelled **I+1** appears to occur "at the same time" as the effect of pseudocode statements labeled **I** for the following instruction. Within one pseudocode sequence, the effects of the statements take place in order. However, between sequences of statements for different instructions that occur "at the same time," there is no defined order. Programs must not depend on a particular order of evaluation between such sections.

Table 1.1 Symbols Used in Instruction Operation Statements (Continued)

<table>
<tr><th>Symbol</th><th>Meaning</th></tr>
<tr><td>PC</td><td>The <i>Program Counter</i> value. During the instruction time of an instruction, this is the address of the instruction word. The address of the instruction that occurs during the next instruction time is determined by assigning a value to <i>PC</i> during an instruction time. If no value is assigned to <i>PC</i> during an instruction time by any pseudocode statement, it is automatically incremented by either 2 (in the case of a 16-bit MIPS16e instruction) or 4 before the next instruction time. A taken branch assigns the target address to the <i>PC</i> during the instruction time of the instruction in the branch delay slot.
In the MIPS Architecture, the PC value is only visible indirectly, such as when the processor stores the restart address into a GPR on a jump-and-link or branch-and-link instruction, or into a Coprocessor 0 register on an exception. The PC value contains a full 32-bit address all of which are significant during a memory reference.</td></tr>
<tr><td>ISA Mode</td><td>In processors that implement the MIPS16e Application Specific Extension, the <i>ISA Mode</i> is a single-bit register that determines in which mode the processor is executing, as follows:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>The processor is executing 32-bit MIPS instructions</td></tr>
<tr><td>1</td><td>The processor is executing MIIPS16e instructions</td></tr>
</table>
In the MIPS Architecture, the ISA Mode value is only visible indirectly, such as when the processor stores a combined value of the upper bits of PC and the ISA Mode into a GPR on a jump-and-link or branch-and-link instruction, or into a Coprocessor 0 register on an exception.</td></tr>
<tr><td>PABITS</td><td>The number of physical address bits implemented is represented by the symbol PABITS. As such, if 36 physical address bits were implemented, the size of the physical address space would be $2^{PABITS} = 2^{36}$ bytes.</td></tr>
<tr><td>FP32RegistersMode</td><td>Indicates whether the FPU has 32-bit or 64-bit floating point registers (FPRs). In MIPS32, the FPU has 32 32-bit FPRs in which 64-bit data types are stored in even-odd pairs of FPRs. In MIPS64, the FPU has 32 64-bit FPRs in which 64-bit data types are stored in any FPR.

In MIPS32 implementations, <b>FP32RegistersMode</b> is always a 0. MIPS64 implementations have a compatibility mode in which the processor references the FPRs as if it were a MIPS32 implementation. In such a case <b>FP32RegisterMode</b> is computed from the FR bit in the <i>Status</i> register. If this bit is a 0, the processor operates as if it had 32 32-bit FPRs. If this bit is a 1, the processor operates with 32 64-bit FPRs.
The value of <b>FP32RegistersMode</b> is computed from the FR bit in the <i>Status</i> register.</td></tr>
<tr><td>InstructionInBranchDelaySlot</td><td>Indicates whether the instruction at the Program Counter address was executed in the delay slot of a branch or jump. This condition reflects the <i>dynamic</i> state of the instruction, not the <i>static</i> state. That is, the value is false if a branch or jump occurs to an instruction whose PC immediately follows a branch or jump, but which is not executed in the delay slot of a branch or jump.</td></tr>
<tr><td>SignalException(exception, argument)</td><td>Causes an exception to be signaled, using the exception parameter as the type of exception and the argument parameter as an exception-specific argument). Control does not return from this pseudocode function—the exception is signaled at the point of the call.</td></tr>
</table>

1.4 For More Information

Various MIPS RISC processor manuals and additional information about MIPS products can be found at the MIPS URL: http://www.mips.com

For comments or questions on the MIPS32® Architecture or this document, send Email to support@mips.com.

Chapter 2

The MIPS32 Privileged Resource Architecture

2.1 Introduction

The MIPS32 Privileged Resource Architecture (PRA) is a set of environments and capabilities on which the Instruction Set Architecture operates. The effects of some components of the PRA are user-visible, for instance, the virtual memory layout. Many other components are visible only to the operating system kernel and to systems programmers. The PRA provides the mechanisms necessary to manage the resources of the CPU: virtual memory, caches, exceptions and user contexts. This chapter describes these mechanisms.

2.2 The MIPS Coprocessor Model

The MIPS ISA provides for up to 4 coprocessors. A coprocessor extends the functionality of the MIPS ISA, while sharing the instruction fetch and execution control logic of the CPU. Some coprocessors, such as the system coprocessor and the floating point unit are standard parts of the ISA, and are specified as such in the architecture documents. Coprocessors are generally optional, with one exception: CP0, the system coprocessor, is required. CP0 is the ISA interface to the Privileged Resource Architecture and provides full control of the processor state and modes.

2.2.1 CP0 - The System Coprocessor

CP0 provides an abstraction of the functions necessary to support an operating system: exception handling, memory management, scheduling, and control of critical resources. The interface to CP0 is through various instructions encoded with the *COP0* opcode, including the ability to move data to and from the CP0 registers, and specific functions that modify CP0 state. The CP0 registers and the interaction with them make up much of the Privileged Resource Architecture.

2.2.2 CP0 Registers

The CP0 registers provide the interface between the ISA and the PRA. The CP0 registers are described in Chapter 8

Chapter 3

MIPS32 Operating Modes

The MIPS32 PRA requires two operating mode: User Mode and Kernel Mode. When operating in User Mode, the programmer has access to the CPU and FPU registers that are provided by the ISA and to a flat, uniform virtual memory address space. When operating in Kernel Mode, the system programmer has access to the full capabilities of the processor, including the ability to change virtual memory mapping, control the system environment, and context switch between processes.

In addition, the MIPS32 PRA supports the implementation of two additional modes: Supervisor Mode and EJTAG Debug Mode. Refer to the EJTAG specification for a description of Debug Mode.

In Release 2 of the Architecture, support was added for 64-bit coprocessors (and, in particular, 64-bit floating point units) with 32-bit CPUs. As such, certain floating point instructions which were previously enabled by 64-bit operations on a MIPS64 processor are now enabled by a new 64-bit floating point operations enabled.

3.1 Debug Mode

For processors that implement EJTAG, the processor is operating in Debug Mode if the DM bit in the CP0 *Debug* register is a one. If the processor is running in Debug Mode, it has full access to all resources that are available to Kernel Mode operation.

3.2 Kernel Mode

The processor is operating in Kernel Mode when the DM bit in the *Debug* register is a zero (if the processor implements Debug Mode), and any of the following three conditions is true:

- The KSU field in the CP0 *Status* register contains 0b00
- The EXL bit in the *Status* register is one
- The ERL bit in the *Status* register is one

The processor enters Kernel Mode at power-up, or as the result of an interrupt, exception, or error. The processor leaves Kernel Mode and enters User Mode or Supervisor Mode when all of the previous three conditions are false, usually as the result of an ERET instruction.

3.3 Supervisor Mode

The processor is operating in Supervisor Mode (if that optional mode is implemented by the processor) when all of the following conditions are true:

- The DM bit in the *Debug* register is a zero (if the processor implements Debug Mode)

- The KSU field in the *Status* register contains 0b01
- The EXL and ERL bits in the *Status* register are both zero

3.4 User Mode

The processor is operating in User Mode when all of the following conditions are true:

- The DM bit in the *Debug* register is a zero (if the processor implements Debug Mode)
- The KSU field in the *Status* register contains 0b10
- The EXL and ERL bits in the *Status* register are both zero

3.5 Other Modes

3.5.1 64-bit Floating Point Operations Enable

Instructions that are implemented by a 64-bit floating point unit are legal under any of the following conditions:

- In an implementation of Release 1 of the Architecture, 64-bit floating point operations are never enabled in a MIPS32 processor.
- If an implementation of Release 2 of the Architecture, 64-bit floating point operations are enabled if the F64 bit in the *FIR* register is a one. The processor must also implement the floating point data type.

3.5.2 64-bit FPR Enable

Access to 64-bit FPRs is controlled by the FR bit in the *Status* register. If the FR bit is one, the FPRs are interpreted as 32 64-bit registers that may contain any data type. If the FR bit is zero, the FPRs are interpreted as 32 32-bit registers, any of which may contain a 32-bit data type (W, S). In this case, 64-bit data types are contained in even-odd pairs of registers.

64-bit FPRs are supported in a MIPS64 processor in Release 1 of the Architecture, or in a 64-bit floating point unit, for both MIPS32 and MIPS64 processors, in Release 2 of the Architecture.

The operation of the processor is **UNPREDICTABLE** under the following conditions:

- The FR bit is a zero, 64-bit operations are enabled, and a floating point instruction is executed whose datatype is L or PS.
- The FR bit is a zero and an odd register is referenced by an instruction whose datatype is 64-bits

3.5.3 Coprocessor 0 Enable

Access to Coprocessor 0 registers are enabled under any of the following conditions:

- The processor is running in Kernel Mode or Debug Mode, as defined above
- The CU0 bit in the *Status* register is one.

Chapter 4

Virtual Memory

4.1 Support in Release 1 and Release 2 of the Architecture

4.1.1 Virtual Memory

In Release 1 of the Architecture, the minimum page size was 4KB, with optional support for pages as large as 256MB. In Release 2 of the Architecture, optional support for 1KB pages was added for use in specific embedded applications that require access to pages smaller than 4KB. Such usage is expected to be in conjunction with a default page size of 4KB and is not intended or suggested to replace the default 4KB page size but, rather, to augment it.

Support for 1KB pages involves the following changes:

- Addition of the *PageGrain* register. This register is also used by the SmartMIPS™ ASE specification, but bits used by Release 2 of the Architecture and the SmartMIPS ASE specification do not overlap.

- Modification of the *EntryHi* register to enable writes to, and use of, bits 12..11 (VPN2X).

- Modification of the *PageMask* register to enable writes to, and use of, bits 12..11 (MaskX).

- Modification of the *EntryLo0* and *EntryLo1* registers to shift the PFN field to the left by 2 bits, when 1KB page support is enabled, to create space for two lower-order physical address bits.

Support for 1KB pages is denoted by the $\text{Config3}_{\text{SP}}$ bit and enabled by the $\text{PageGrain}_{\text{ESP}}$ bit.

4.2 Terminology

4.2.1 Address Space

An *Address Space* is the range of all possible addresses that can be generated. There is one 32-bit Address Space in the MIPS32 Architecture.

4.2.2 Segment and Segment Size

A *Segment* is a defined subset of an Address Space that has self-consistent reference and access behavior. Segments are either 2^{29} or 2^{31} bytes in size, depending on the specific Segment.

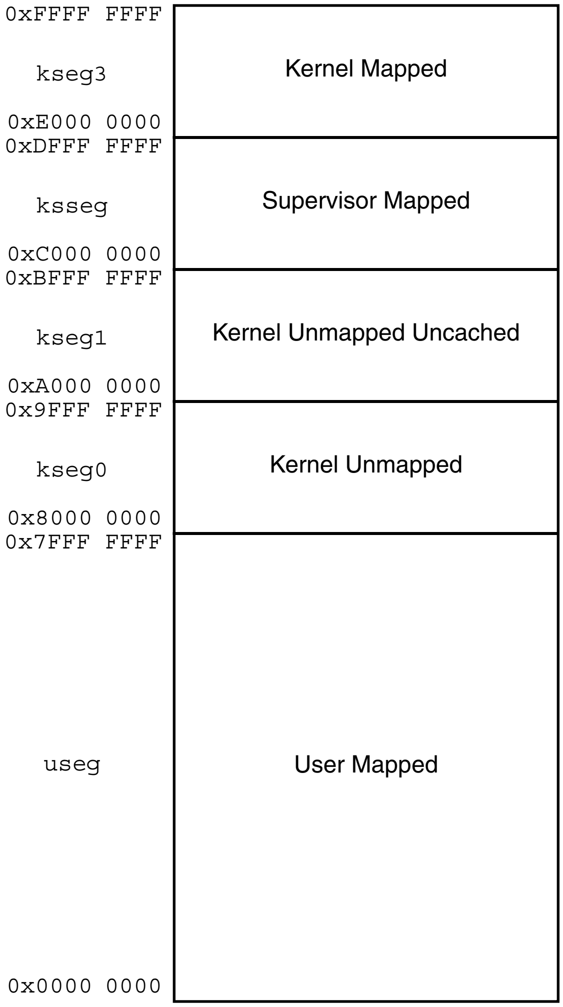
4.2.3 Physical Address Size (PABITS)

The number of physical address bits implemented is represented by the symbol *PABITS*. As such, if 36 physical address bits were implemented, the size of the physical address space would be $2^{PABITS} = 2^{36}$ bytes. The format of the *EntryLo0* and *EntryLo1* registers implicitly limits the physical address size to 2^{36} bytes. Software may determine the value of PABITS by writing all ones to the *EntryLo0* or *EntryLo1* registers and reading the value back. Bits read as “1” from the PFN field allow software to determine the boundary between the PFN and 0 fields to calculate the value of PABITS.

4.3 Virtual Address Spaces

The MIPS32 virtual address space is divided into five segments as shown in Figure 4-1.

Figure 4-1 Virtual Address Space



Each Segment of an Address Space is classified as “Mapped” or “Unmapped”. A “Mapped” address is one that is translated through the TLB or other address translation unit. An “Unmapped” address is one which is not translated through the TLB and which provides a window into the lowest portion of the physical address space, starting at physical address zero, and with a size corresponding to the size of the unmapped Segment.

Additionally, the kseg1 Segment is classified as “Uncached”. References to this Segment bypass all levels of the cache hierarchy and allow direct access to memory without any interference from the caches.

Table 4.1 lists the same information in tabular form. Each Segment of an Address Space is associated with one of the three processor operating modes (User, Supervisor, or Kernel). A Segment that is associated with a particular mode is accessible if the processor is running in that or a more privileged mode. For example, a Segment associated with User Mode is accessible when the processor is running in User, Supervisor, or Kernel Modes. A Segment is not accessible if the processor is running in a less privileged mode than that associated with the Segment. For example, a Segment associated with Supervisor Mode is not accessible when the processor is running in User Mode and such a reference results in an Address Error Exception. The “Reference Legal from Mode(s)” column in Table 4-2 lists the modes from which each Segment may be legally referenced.

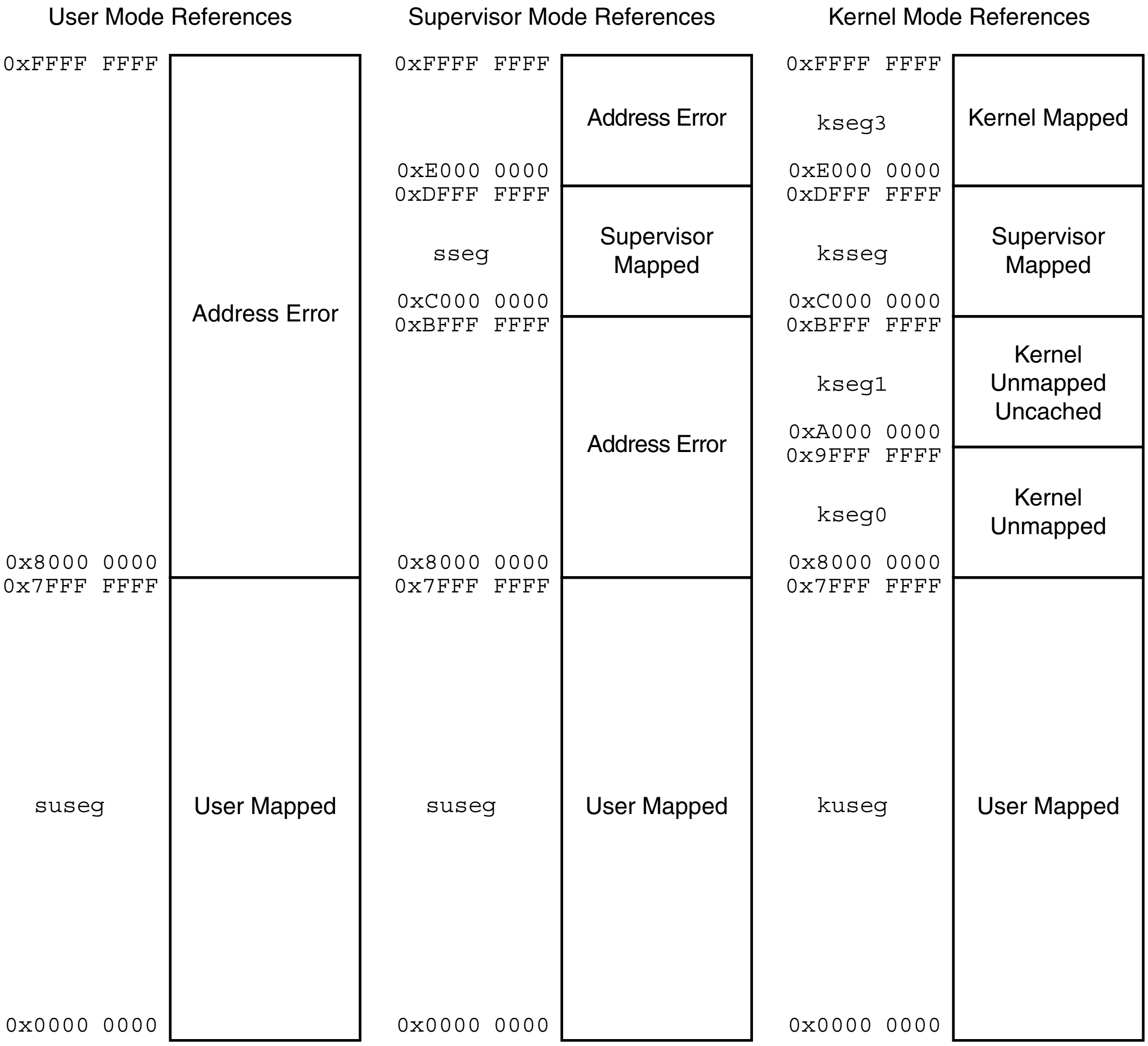
Table 4.1 Virtual Memory Address Spaces

$VA_{31..29}$	Segment Name(s)	Address Range	Associated with Mode	Reference Legal from Mode(s)	Actual Segment Size
0b111	kseg3	0xFFFF FFFF through 0xE000 0000	Kernel	Kernel	2^{29} bytes
0b110	sseg ksseg	0xDFFF FFFF through 0xC000 0000	Supervisor	Supervisor Kernel	2^{29} bytes
0b101	kseg1	0xBFFF FFFF through 0xA000 0000	Kernel	Kernel	2^{29} bytes
0b100	kseg0	0x9FFF FFFF through 0x8000 0000	Kernel	Kernel	2^{29} bytes
0b0xx	useg suseg kuseg	0x7FFF FFFF through 0x0000 0000	User	User Supervisor Kernel	2^{31} bytes

If a Segment has more than one name, each name denotes the mode from which the Segment is referenced. For example, the Segment name “useg” denotes a reference from user mode, while the Segment name “kuseg” denotes a reference to the same Segment from kernel mode.

Figure 4-6 shows the Address Space as seen when the processor is operating in each of the operating modes.

Figure 4-2 References as a Function of Operating Mode



4.4 Compliance

A MIPS32 compliant processor must implement the following Segments:

- useg/kuseg
- kseg0
- kseg1

In addition, a MIPS32 compliant processor using the TLB-based address translation mechanism must also implement the kseg3 Segment.

4.5 Access Control as a Function of Address and Operating Mode

Table 4.2 enumerates the action taken by the processor for each section of the 32-bit Address Space as a function of the operating mode of the processor. The selection of TLB Refill vector and other special-cased behavior is also listed for each reference.

Table 4.2 Address Space Access as a Function of Operating Mode

Virtual Address Range	Segment Name(s)	Action when Referenced from Operating Mode		
		User Mode	Supervisor Mode	Kernel Mode

`0xFFFF FFFF`
through
`0xE000 0000`
	kseg3	Address Error	Address Error	Mapped
				See Section 4.8 for special behavior when Debug$_{DM}$ = 1

`0xDFFF FFFF`
through
`0xC000 0000`
	sseg
	ksseg
		Address Error	Mapped	Mapped

`0xBFFF FFFF`
through
`0xA000 0000`
	kseg1	Address Error	Address Error	Unmapped, Uncached
				See Section 4.6

`0x9FFF FFFF`
through
`0x8000 0000`
	kseg0	Address Error	Address Error	Unmapped
				See Section 4.6

`0x7FFF FFFF`
through
`0x0000 0000`
	useg
	suseg
	kuseg
		Mapped	Mapped	Unmapped if Status$_{ERL}$=1
				See Section 4.7
				Mapped if Status$_{ERL}$=0

4.6 Address Translation and Cacheability & Coherency Attributes for the kseg0 and kseg1 Segments

The kseg0 and kseg1 Unmapped Segments provide a window into the least significant 2^{29} bytes of physical memory, and, as such, are not translated using the TLB or other address translation unit. The cacheability and coherency attribute of the kseg0 Segment is supplied by the K0 field of the CP0 *Config* register. The cacheability and coherency attribute for the kseg1 Segment is always Uncached. Table 4.3 describes how this transformation is done, and the source of the cacheability and coherency attributes for each Segment.

Table 4.3 Address Translation and Cacheability and Coherency Attributes for the kseg0 and kseg1 Segments

Segment Name	Virtual Address Range	Generates Physical Address	Cache Attribute

kseg1	`0xBFFF FFFF`
	through
	`0xA000 0000`
		`0x1FFF FFFF`
		through
		`0x0000 0000`
			Uncached

kseg0	`0x9FFF FFFF`
	through
	`0x8000 0000`
		`0x1FFF FFFF`
		through
		`0x0000 0000`
			From K0 field of *Config* Register

4.7 Address Translation for the kuseg Segment when Status$_{ERL}$ = 1

To provide support for the cache error handler, the kuseg Segment becomes an unmapped, uncached Segment, similar to the kseg1 Segment, if the ERL bit is set in the *Status* register. This allows the cache error exception code to operate uncached using GPR R0 as a base register to save other GPRs before use.

4.8 Special Behavior for the kseg3 Segment when Debug$_{DM}$ = 1

If EJTAG is implemented on the processor, the EJTAG block must treat the virtual address range `0xFF20 0000` through `0xFF3F FFFF`, inclusive, as a special memory-mapped region in Debug Mode. A MIPS32 compliant implementation that also implements EJTAG must:

- explicitly range check the address range as given and not assume that the entire region between `0xFF20 0000` and `0xFFFF FFFF` is included in the special memory-mapped region.
- not enable the special EJTAG mapping for this region in any mode other than in EJTAG Debug mode.

Even in Debug mode, normal memory rules may apply in some cases. Refer to the EJTAG specification for details on this mapping.

4.9 TLB-Based Virtual Address Translation[1]

This section describes the TLB-based virtual address translation mechanism. Note that sufficient TLB entries must be implemented to avoid a TLB exception loop on load and store instructions.

[1] Refer to A.1 “Fixed Mapping MMU” on page 155 and A.2 “Block Address Translation” on page 159 for descriptions of alternative MMU organizations

4.9.1 Address Space Identifiers (ASID)

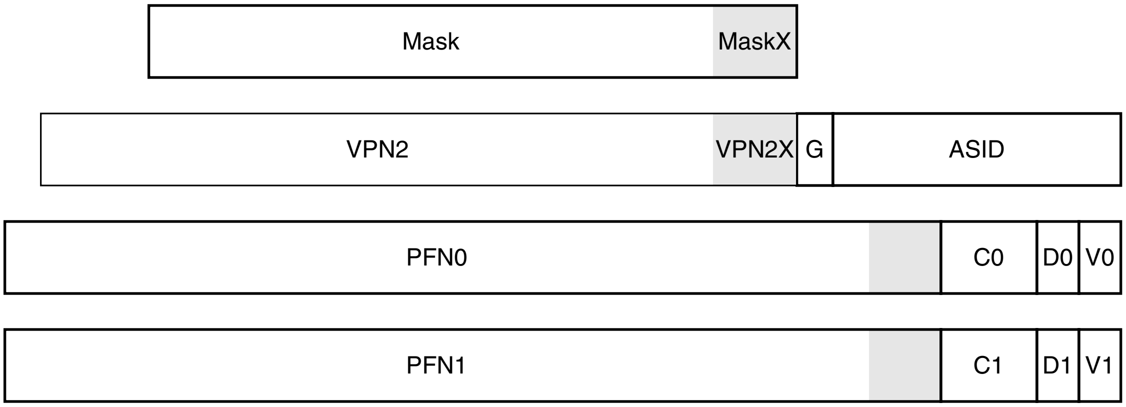
The TLB-based translation mechanism supports Address Space Identifiers to uniquely identify the same virtual address across different processes. The operating system assigns ASIDs to each process and the TLB keeps track of the ASID when doing address translation. In certain circumstances, the operating system may wish to associate the same virtual address with all processes. To address this need, the TLB includes a global (G) bit which over-rides the ASID comparison during translation.

4.9.2 TLB Organization

The TLB is a fully-associative structure which is used to translate virtual addresses. Each entry contains two logical components: a comparison section and a physical translation section. The comparison section includes the virtual page number (VPN2 and, in Release 2, VPNX) (actually, the virtual page number/2 since each entry maps two physical pages) of the entry, the ASID, the G(lobal) bit and a recommended mask field which provides the ability to map different page sizes with a single entry. The physical translation section contains a pair of entries, each of which contains the physical page frame number (PFN), a valid (V) bit, a dirty (D) bit, and a cache coherency field (C), whose valid encodings are given in Table 8.8. There are two entries in the translation section for each TLB entry because each TLB entry maps an aligned pair of virtual pages and the pair of physical translation entries corresponds to the even and odd pages of the pair.

Figure 4-3 shows the logical arrangement of a TLB entry, including the optional support added in Release 2 of the Architecture for 1KB page sizes. Light grey fields denote extensions to the right that are required to support 1KB page sizes. This extension is not present in an implementation of Release 1 of the Architecture.

Figure 4-3 Contents of a TLB Entry



▨ Fields marked with this color are optional Release 2 features required to support 1KB pages

The fields of the TLB entry correspond exactly to the fields in the CP0 *PageMask*, *EntryHi*, *EntryLo0* and *EntryLo1* registers. The even page entries in the TLB (e.g., PFN0) come from *EntryLo0*. Similarly, odd page entries come from *EntryLo1*.

4.9.3 TLB Initialization

In many processor implementations, software must initialize the TLB during the power-up process. In processors that detect multiple TLB matches and signal this via a machine check assumption, software must be prepared to handle such an exception or use a TLB initialization algorithm that minimizes or eliminates the possibility of the exception.

In Release 1 of the Architecture, processor implementations could detect and report multiple TLB matches either on a TLB write (TLBWI or TLBWR instructions) or a TLB read (TLB access or TLBR or TLBP instructions). In Release 2 of the Architecture, processor implentations are limited to reporting multiple TLB matches only on TLB write, and this is also true of most implementations of Release 1 of the Architecture.

The following code example shows a TLB initialization routine which, on implementations of Release 2 of the Architecture, eliminates the possibility of reporting a machine check during TLB initialization. This example has equivalent effect on implementations of Release 1 of the Architecture which report multiple TLB exceptions only on a TLB write, and minimizes the probability of such an exception occuring on other implementations.

```
/*
 * InitTLB
 *
 * Initialize the TLB to a power-up state, guaranteeing that all entries
 * are unique and invalid.
 *
 * Arguments:
 *     a0     =   Maximum TLB index (from MMUSize field of C0_Config1)
 *
 * Returns:
 *     No value
 *
 * Restrictions:
 *     This routine must be called in unmapped space
 *
 * Algorithm:
 *     va = kseg0_base;
 *     for (entry = max_TLB_index; entry >= 0, entry--) {
 *        while (TLB_Probe_Hit(va)) {
 *            va += Page_Size;
 *        }
 *        TLB_Write(entry, va, 0, 0, 0);
 *     }
 *
 * Notes:
 *     -   The Hazard macros used in the code below expand to the appropriate
 *         number of SSNOPs in an implementation of Release 1 of the
 *         Architecture, and to an ehb in an implementation of Release 2 of
 *         the Architecture. See , "CP0 Hazards," on page 65 for
 *         more additional information.
 */

InitTLB:
/*
 * Clear PageMask, EntryLo0 and EntryLo1 so that valid bits are off, PFN values
 * are zero, and the default page size is used.
 */
   mtc0   zero, C0_EntryLo0          /* Clear out PFN and valid bits */
   mtc0   zero, C0_EntryLo1
   mtc0   zero, C0_PageMask          /* Clear out mask register *
/* Start with the base address of kseg0 for the VA part of the TLB */
   la     t0, A_K0BASE               /* A_K0BASE == 0x8000.0000 */

/*
 * Write the VA candidate to EntryHi and probe the TLB to see if if is
 * already there. If it is, a write to the TLB may cause a machine
 * check, so just increment the VA candidate by one page and try again.
 */
10:
   mtc0   t0, C0_EntryHi              /* Write VA candidate */
   TLBP_Write_Hazard()                /* Clear EntryHi hazard (ssnop/ehb in R1/2) */
   tlbp                               /* Probe the TLB to check for a match */
   TLBP_Read_Hazard()                 /* Clear Index hazard (ssnop/ehb in R1/2) */
   mfc0   t1, C0_Index                /* Read back flag to check for match */
   bgez   t1, 10b                     /* Branch if about to duplicate an entry */
   addiu  t0, (1<<S_EntryHiVPN2)      /* Add 1 to VPN index in va */

/*
 * A write of the VPN candidate will be unique, so write this entry
 * into the next index, decrement the index, and continue until the
 * index goes negative (thereby writing all TLB entries)
 */
   mtc0   a0, C0_Index                /* Use this as next TLB index */
   TLBW_Write_Hazard()                /* Clear Index hazard (ssnop/ehb in R1/2) */
   tlbwi                              /* Write the TLB entry */
   bne    a0, zero, 10b               /* Branch if more TLB entries to do */
   addiu  a0, -1                      /* Decrement the TLB index

/*
 * Clear Index and EntryHi simply to leave the state constant for all
 * returns
 */
   mtc0   zero, C0_Index
   mtc0   zero, C0_EntryHi
   jr     ra                          /* Return to caller */
   nop
```

4.9.4 Address Translation

Release 2 of the Architecture introduced support for 1KB pages. For clarity in the discussion below, the following terms should be taken in the general sense to include the new Release 2 features:

Term Used Below	Release 2 Substitution	Comment
VPN2	VPN2 || VPN2X	Release 2 implementations that support 1KB pages concatenate the VPN2 and VPN2X fields to form the virtual page number for a 1KB page
Mask	Mask || MaskX	Release 2 implementations that support 1KB pages concatenate the Mask and MaskX fields to form the don’t care mask for 1KB pages

When an address translation is requested, the virtual page number and the current process ASID are presented to the TLB. All entries are checked simultaneously for a match, which occurs when all of the following conditions are true:

- The current process ASID (as obtained from the *EntryHi* register) matches the ASID field in the TLB entry, or the G bit is set in the TLB entry.
- The appropriate bits of the virtual page number match the corresponding bits of the VPN2 field stored within the TLB entry. The “appropriate” number of bits is determined by the Mask fields in each entry by ignoring each bit in the virtual page number and the TLB VPN2 field corresponding to those bits that are set in the Mask fields. This allows each entry of the TLB to support a different page size, as determined by the *PageMask* register at

the time that the TLB entry was written. If the recommended *PageMask* register is not implemented, the TLB operation is as if the PageMask register was written with the encoding for a 4KB page.

If a TLB entry matches the address and ASID presented, the corresponding PFN, C, V, and D bits are read from the translation section of the TLB entry. Which of the two PFN entries is read is a function of the virtual address bit immediately to the right of the section masked with the Mask entry.

The valid and dirty bits determine the final success of the translation. If the valid bit is off, the entry is not valid and a TLB Invalid exception is raised. If the dirty bit is off and the reference was a store, a TLB Modified exception is raised. If there is an address match with a valid entry and no dirty exception, the PFN and the cache coherency bits are appended to the offset-within-page bits of the address to form the final physical address with attributes.

For clarity, the TLB lookup processes have been separated into two sets of pseudo code:

1. One used by an implementation of Release 1 of the Architecture, or an implementation of Release 2 of the Architecture which does not include 1KB page support (as denoted by Config3_{SP}). This instance is called the “4KB TLB Lookup”.

2. One used by an implementation of Release 2 of the Architecture which does include 1KB page support. This instance is called the “1KB TLB Lookup”.

The 4KB TLB Lookup pseudo code is as follows:

```
found ← 0
for i in 0...TLBEntries-1
   if ((TLB[i]$_{VPN2}$ and not (TLB[i]$_{Mask}$)) = (va$_{31..13}$ and not (TLB[i]$_{Mask}$))) and
      (TLB[i]$_{G}$ or (TLB[i]$_{ASID}$ = EntryHi$_{ASID}$)) then
      # EvenOddBit selects between even and odd halves of the TLB as a function of
      # the page size in the matching TLB entry. Not all page sizes need
      # be implemented on all processors, so the case below uses an ‘x’ to
      # denote don’t-care cases. The actual implementation would select
      # the even-odd bit in a way that is compatible with the page sizes
      # actually implemented.
      case TLB[i]$_{Mask}$
         0b0000 0000 0000 0000: EvenOddBit ← 12 /* 4KB page */
         0b0000 0000 0000 0011: EvenOddBit ← 14 /* 16KB page */
         0b0000 0000 0000 11xx: EvenOddBit ← 16 /* 64KB page */
         0b0000 0000 0011 xxxx: EvenOddBit ← 18 /* 256KB page */
         0b0000 0000 11xx xxxx: EvenOddBit ← 20 /* 1MB page */
         0b0000 0011 xxxx xxxx: EvenOddBit ← 22 /* 4MB page */
         0b0000 11xx xxxx xxxx: EvenOddBit ← 24 /* 16MB page */
         0b0011 xxxx xxxx xxxx: EvenOddBit ← 26 /* 64MB page */
         0b11xx xxxx xxxx xxxx: EvenOddBit ← 28 /* 256MB page */
         otherwise:    UNDEFINED
      endcase
      if va$_{EvenOddBit}$ = 0 then
         pfn ← TLB[i]$_{PFN0}$
         v ← TLB[i]$_{V0}$
         c ← TLB[i]$_{C0}$
         d ← TLB[i]$_{D0}$
      else
         pfn ← TLB[i]$_{PFN1}$
         v ← TLB[i]$_{V1}$
         c ← TLB[i]$_{C1}$
         d ← TLB[i]$_{D1}$
      endif
```

```
        if v = 0 then
            SignalException(TLBInvalid, reftype)
        endif
        if (d = 0) and (reftype = store) then
            SignalException(TLBModified)
        endif
        # pfn$_{PABITS-1-12..0}$ corresponds to pa$_{PABITS-1..12}$
        pa ← pfn$_{PABITS-1-12..EvenOddBit-12}$ || va$_{EvenOddBit-1..0}$
        found ← 1
        break
    endif
endfor
if found = 0 then
    SignalException(TLBMiss, reftype)
endif
```

The 1KB TLB Lookup pseudo code is as follows:

```
found ← 0
for i in 0...TLBEntries-1
    if ((TLB[i]$_{VPN2}$ and not (TLB[i]$_{Mask}$)) = (va$_{31..13}$ and not (TLB[i]$_{Mask}$))) and
        (TLB[i]$_{G}$ or (TLB[i]$_{ASID}$ = EntryHi$_{ASID}$)) then
        # EvenOddBit selects between even and odd halves of the TLB as a function of
        # the page size in the matching TLB entry. Not all pages sizes need
        # be implemented on all processors, so the case below uses an ‘x’ to
        # denote don’t-care cases. The actual implementation would select
        # the even-odd bit in a way that is compatible with the page sizes
        # actually implemented.
        case TLB[i]$_{Mask}$
            0b0000 0000 0000 0000 00: EvenOddBit ← 10 /* 1KB page */
            0b0000 0000 0000 0000 11: EvenOddBit ← 12 /* 4KB page */
            0b0000 0000 0000 0011 xx: EvenOddBit ← 14 /* 16KB page */
            0b0000 0000 0000 11xx xx: EvenOddBit ← 16 /* 64KB page */
            0b0000 0000 0011 xxxx xx: EvenOddBit ← 18 /* 256KB page */
            0b0000 0000 11xx xxxx xx: EvenOddBit ← 20 /* 1MB page */
            0b0000 0011 xxxx xxxx xx: EvenOddBit ← 22 /* 4MB page */
            0b0000 11xx xxxx xxxx xx: EvenOddBit ← 24 /* 16MB page */
            0b0011 xxxx xxxx xxxx xx: EvenOddBit ← 26 /* 64MB page */
            0b11xx xxxx xxxx xxxx xx: EvenOddBit ← 28 /* 256MB page */
            otherwise:    UNDEFINED
        endcase
        if va$_{EvenOddBit}$ = 0 then
            pfn ← TLB[i]$_{PFN0}$
            v ← TLB[i]$_{V0}$
            c ← TLB[i]$_{C0}$
            d ← TLB[i]$_{D0}$
        else
            pfn ← TLB[i]$_{PFN1}$
            v ← TLB[i]$_{V1}$
            c ← TLB[i]$_{C1}$
            d ← TLB[i]$_{D1}$
        endif
        if v = 0 then
            SignalException(TLBInvalid, reftype)
        endif
        if (d = 0) and (reftype = store) then
            SignalException(TLBModified)
        endif
```

```
            # $pfn_{PABITS-1-10..0}$ corresponds to $pa_{PABITS-1..10}$
            pa ← $pfn_{PABITS-1-10..EvenOddBit-10}$ || $va_{EvenOddBit-1..0}$
            found ← 1
            break
        endif
    endfor
    if found = 0 then
        SignalException(TLBMiss, reftype)
    endif
```

Table 4.4 demonstrates how the physical address is generated as a function of the page size of the TLB entry that matches the virtual address. The "Even/Odd Select" column of Table 4.4 indicates which virtual address bit is used to select between the even (EntryLo0) or odd (EntryLo1) entry in the matching TLB entry. The "$PA_{(PABITS-1)..0}$ Generated From" columns specify how the physical address is generated from the selected PFN and the offset-in-page bits in the virtual address. In this column, PFN is the physical page number as loaded into the TLB from the *EntryLo0* or *EntryLo1* registers, and has one of two bit ranges:

PFN Range	PA Range	Comment
$PFN_{(\mathit{PABITS}-1)-12..0}$	$PA_{\mathit{PABITS}-1..12}$	Release 1 implementation, or Release 2 implementation without support for 1KB pages
$PFN_{(\mathit{PABITS}-1)-10..0}$	$PA_{\mathit{PABITS}-1..10}$	Release 2 implementation with support for 1KB pages enabled

Table 4.4 Physical Address Generation

		$PA_{(\mathit{PABITS}-1)..0}$ Generated From:	
Page Size	**Even/Odd Select**	**Release 1 or Release 2 with 1KB Page Support Disabled**	**Release 2 with 1KB Page Support Enabled**
1K Bytes	VA_{10}	Not Applicable	$PFN_{(\mathit{PABITS}-1)-10..0} \| VA_{9..0}$
4K Bytes	VA_{12}	$PFN_{(\mathit{PABITS}-1)-12..0} \| VA_{11..0}$	$PFN_{(\mathit{PABITS}-1)-10..2} \| VA_{11..0}$
16K Bytes	VA_{14}	$PFN_{(\mathit{PABITS}-1)-12..2} \| VA_{13..0}$	$PFN_{(\mathit{PABITS}-1)-10..4} \| VA_{13..0}$
64K Bytes	VA_{16}	$PFN_{(\mathit{PABITS}-1)-12..4} \| VA_{15..0}$	$PFN_{(\mathit{PABITS}-1)-10..6} \| VA_{15..0}$
256K Bytes	VA_{18}	$PFN_{(\mathit{PABITS}-1)-12..6} \| VA_{17..0}$	$PFN_{(\mathit{PABITS}-1)-10..8} \| VA_{17..0}$
1M Bytes	VA_{20}	$PFN_{(\mathit{PABITS}-1)-12..8} \| VA_{19..0}$	$PFN_{(\mathit{PABITS}-1)-10..10} \| VA_{19..0}$
4M Bytes	VA_{22}	$PFN_{(\mathit{PABITS}-1)-12..10} \| VA_{21..0}$	$PFN_{(\mathit{PABITS}-1)-10..12} \| VA_{21..0}$
16M Bytes	VA_{24}	$PFN_{(\mathit{PABITS}-1)-12..12} \| VA_{23..0}$	$PFN_{(\mathit{PABITS}-1)-10..14} \| VA_{23..0}$
64MBytes	VA_{26}	$PFN_{(\mathit{PABITS}-1)-12..14} \| VA_{25..0}$	$PFN_{(\mathit{PABITS}-1)-10..16} \| VA_{25..0}$
256MBytes	VA_{28}	$PFN_{(\mathit{PABITS}-1)-12..16} \| VA_{27..0}$	$PFN_{(\mathit{PABITS}-1)-10..18} \| VA_{27..0}$

Chapter 5

Interrupts and Exceptions

Release 2 of the Architecture added the following features related to the processing of Exceptions and Interrupts:

- The addition of the Coprocessor 0 *EBase* register, which allows the exception vector base address to be modified for exceptions that occur when $\text{Status}_{\text{BEV}}$ equals 0. The *EBase* register is required.

- The extension of the Release 1 interrupt control mechanism to include two optional interrupt modes:

  - Vectored Interrupt (VI) mode, in which the various sources of interrupts are prioritized by the processor and each interrupt is vectored directly to a dedicated handler. When combined with GPR shadow registers, introduced in the next chapter, this mode significantly reduces the number of cycles required to process an interrupt.

  - External Interrupt Controller (EIC) mode, in which the definition of the coprocessor 0 register fields associated with interrupts changes to support an external interrupt controller. This can support many more prioritized interrupts, while still providing the ability to vector an interrupt directly to a dedicated handler and take advantage of the GPR shadow registers.

- The ability to stop the *Count* register for highly power-sensitive applications in which the *Count* register is not used, or for reduced power mode. This change is required.

- The addition of the DI and EI instructions which provide the ability to atomically disable or enable interrupts. Both instructions are required.

- The addition of the *TI* and *PCI* bits in the *Cause* register to denote pending timer and performance counter interrupts. This change is required.

- The addition of an execution hazard sequence which can be used to clear hazards introduced when software writes to a coprocessor 0 register which affects the interrupt system state.

5.1 Interrupts

Release 1 of the Architecture included support for two software interrupts, six hardware interrupts, and two special-purpose interrupts: timer and performance counter. The timer and performance counter interrupts were combined with hardware interrupt 5 in an implementation-dependent manner. Interrupts were handled either through the general exception vector (offset 0x180) or the special interrupt vector (0x200), based on the value of Cause_{IV}. Software was required to prioritize interrupts as a function of the Cause_{IP} bits in the interrupt handler prologue.

Release 2 of the Architecture adds an upward-compatible extension to the Release 1 interrupt architecture that supports vectored interrupts. In addition, Release 2 adds a new interrupt mode that supports the use of an external interrupt controller by changing the interrupt architecture.

Although a Non-Maskable Interrupt (NMI) includes “interrupt” in its name, it is more correctly described as an NMI exception because it does not affect, nor is it controlled by the processor interrupt system.

An interrupt is only taken when all of the following are true:

- A specific request for interrupt service is made, as a function of the interrupt mode, described below.
- The *IE* bit in the *Status* register is a one.
- The *DM* bit in the *Debug* register is a zero (for processors implementing EJTAG)
- The *EXL* and *ERL* bits in the *Status* register are both zero.

Logically, the request for interrupt service is ANDed with the *IE* bit of the *Status* register. The final interrupt request is then asserted only if both the *EXL* and *ERL* bits in the *Status* register are zero, and the *DM* bit in the *Debug* register is zero, corresponding to a non-exception, non-error, non-debug processing mode, respectively.

5.1.1 Interrupt Modes

An implementation of Release 1 of the Architecture only implements interrupt compatibility mode.

An implementation of Release 2 of the Architecture may implement up to three interrupt modes:

- Interrupt compatibility mode, which acts identically to that in an implementation of Release 1 of the Architecture. This mode is required.
- Vectored Interrupt (VI) mode, which adds the ability to prioritize and vector interrupts to a handler dedicated to that interrupt, and to assign a GPR shadow set for use during interrupt processing. This mode is optional and its presence is denoted by the VInt bit in the *Config3* register.
- External Interrupt Controller (EIC) mode, which redefines the way in which interrupts are handled to provide full support for an external interrupt controller handling prioritization and vectoring of interrupts. This mode is optional and its presence is denoted by the *VEIC* bit in the *Config3* register.

A compatible implementation of Release 2 of the Architecture must implement interrupt compatibility mode, and may optionally implement one or both vectored interrupt modes. Inclusion of the optional modes may be done selectively in the implementation of the processor, or they may always be implemented and be dynamically enabled based on coprocessor 0 control bits. The reset state of the processor is to interrupt compatibility mode such that an implementation of Release 2 of the Architecture is fully compatible with implementations of Release 1 of the Architecture.

Table 5.1 shows the current interrupt mode of the processor as a function of the coprocessor 0 register fields that can affect the mode.

Table 5.1 Interrupt Modes

$Status_{BEV}$	$Cause_{IV}$	$IntCtl_{VS}$	$Config3_{VINT}$	$Config3_{VEIC}$	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

Table 5.1 Interrupt Modes

$\text{Status}_{\text{BEV}}$	Cause_{IV}	$\text{IntCtl}_{\text{VS}}$	$\text{Config3}_{\text{VINT}}$	$\text{Config3}_{\text{VEIC}}$	Interrupt Mode
0	1	≠0	x	1	External Interrupt Controller
0	1	≠0	0	0	Not Allowed - $\text{IntCtl}_{\text{VS}}$ is zero if neither Vectored Interrupt nor External Interrupt Controller mode are implemented.
"x" denotes don't care					

5.1.1.1 Interrupt Compatibility Mode

This is the only interrupt mode for a Release 1 processor and the default interrupt mode for a Release 2 processor. This mode is entered when a Reset exception occurs. In this mode, interrupts are non-vectored and dispatched though exception vector offset 0x180 (if $\text{Cause}_{\text{IV}} = 0$) or vector offset 0x200 (if $\text{Cause}_{\text{IV}} = 1$). This mode is in effect if any of the following conditions are true:

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

The current interrupt requests are visible via the IP field in the Cause register on any read of the register (not just after an interrupt exception has occurred). Note that an interrupt request may be deasserted between the time the processor starts the interrupt exception and the time that the software interrupt handler runs. The software interrupt handler must be prepared to handle this condition by simply returning from the interrupt via ERET. A request for interrupt service is generated as shown in Table 5.2.

Table 5.2 Request for Interrupt Service in Interrupt Compatibility Mode

Interrupt Type	Interrupt Source	Interrupt Request Calculated From
Hardware Interrupt, Timer Interrupt, or Performance Counter Interrupt	HW5	$\text{Cause}_{\text{IP7}}$ and $\text{Status}_{\text{IM7}}$
Hardware Interrupt	HW4	$\text{Cause}_{\text{IP6}}$ and $\text{Status}_{\text{IM6}}$
	HW3	$\text{Cause}_{\text{IP5}}$ and $\text{Status}_{\text{IM5}}$
	HW2	$\text{Cause}_{\text{IP4}}$ and $\text{Status}_{\text{IM4}}$
	HW1	$\text{Cause}_{\text{IP3}}$ and $\text{Status}_{\text{IM3}}$
	HW0	$\text{Cause}_{\text{IP2}}$ and $\text{Status}_{\text{IM2}}$
Software Interrupt	SW1	$\text{Cause}_{\text{IP1}}$ and $\text{Status}_{\text{IM1}}$
	SW0	$\text{Cause}_{\text{IP0}}$ and $\text{Status}_{\text{IM0}}$

A typical software handler for interrupt compatibility mode might look as follows:

```
/*
 * Assumptions:
 *  - Cause$_{IV}$ = 1 (if it were zero, the interrupt exception would have to
 *                 be isolated from the general exception vector before getting
 *                 here)
 *  - GPRs k0 and k1 are available (no shadow register switches invoked in
 *                                  compatibility mode)
 *  - 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 IntCtl$_{VS}$ */
   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 simply 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 Status$_{IM}$ 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,
     * 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 can not 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 */
```

5.1.1.2 Vectored Interrupt Mode

Vectored Interrupt mode builds on the interrupt compatibility mode by adding a priority encoder to prioritize pending interrupts and to generate a vector with which each interrupt can be directed to a dedicated handler routine. This mode also allows each interrupt to be mapped to a GPR shadow set for use by the interrupt handler. Vectored Interrupt mode is in effect if all of the following conditions are true:

- $\text{Config3}_{\text{VInt}} = 1$

- $\text{Config3}_{\text{VEIC}} = 0$

- $\text{IntCtl}_{\text{VS}} \neq 0$

- Cause_{IV} = 1
- $\text{Status}_{\text{BEV}}$ = 0

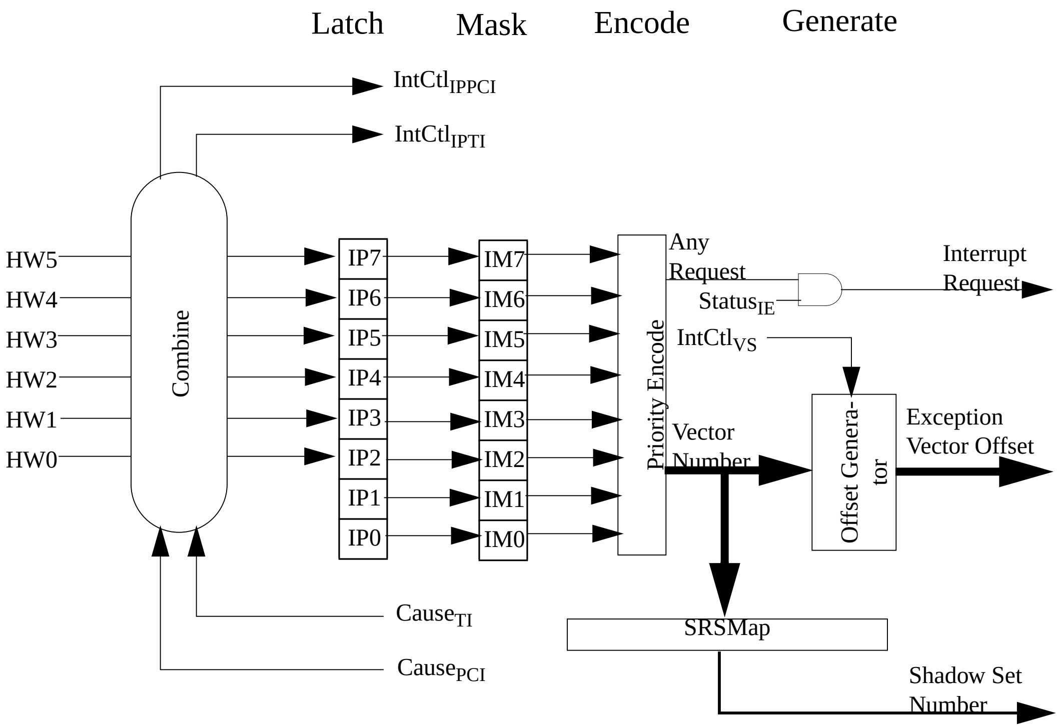
In VI interrupt mode, the six hardware interrupts are interpreted as individual hardware interrupt requests. The timer and performance counter interrupts are combined in an implementation-dependent way with the hardware interrupts (with the interrupt with which they are combined indicated by $\text{IntCtl}_{\text{IPTI}}$ and $\text{IntCtl}_{\text{IPPCI}}$, respectively) to provide the appropriate relative priority of these interrupts with that of the hardware interrupts. The processor interrupt logic ANDs each of the Cause_{IP} bits with the corresponding $\text{Status}_{\text{IM}}$ bits. If any of these values is 1, and if interrupts are enabled ($\text{Status}_{\text{IE}}$ = 1, $\text{Status}_{\text{EXL}}$ = 0, and $\text{Status}_{\text{ERL}}$ = 0), an interrupt is signaled and a priority encoder scans the values in the order shown in Table 5.3.

Table 5.3 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	$\text{Cause}_{\text{IP7}}$ and $\text{Status}_{\text{IM7}}$	7
		HW4	$\text{Cause}_{\text{IP6}}$ and $\text{Status}_{\text{IM6}}$	6
		HW3	$\text{Cause}_{\text{IP5}}$ and $\text{Status}_{\text{IM5}}$	5
		HW2	$\text{Cause}_{\text{IP4}}$ and $\text{Status}_{\text{IM4}}$	4
		HW1	$\text{Cause}_{\text{IP3}}$ and $\text{Status}_{\text{IM3}}$	3
		HW0	$\text{Cause}_{\text{IP2}}$ and $\text{Status}_{\text{IM2}}$	2
	Software	SW1	$\text{Cause}_{\text{IP1}}$ and $\text{Status}_{\text{IM1}}$	1
Lowest Priority		SW0	$\text{Cause}_{\text{IP0}}$ and $\text{Status}_{\text{IM0}}$	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 5-1.

Figure 5-1 Interrupt Generation for Vectored Interrupt Mode



Note that an interrupt request may be deasserted between the time the processor detects the interrupt request and the time that the software interrupt handler runs. The software interrupt handler must be prepared to handle this condition by simply returning from the interrupt via ERET.

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. Unlike the compatibility mode examples, a vectored interrupt handler may take advantage of a dedicated GPR shadow set to avoid saving any registers. As such, the SimpleInterrupt code shown above need not save the GPRs.

A nested interrupt is similar to that shown for compatibility mode, but may also take advantage of running the nested exception routine in the GPR shadow set dedicated to the interrupt or in another shadow set. Such a routine might look as follows:

```
NestedException:
/*
 * Nested exceptions typically require saving the EPC, Status and SRSCtl registers,
 * setting up the appropriate GPR shadow set for the 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 can not cover all nuances of this processing and is intended only
 * to demonstrate the concepts.
 */

   /* Use the current GPR shadow set, 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 */
        mfc0   k0, C0_SRSCtl       /* Save SRSCtl if changing shadow sets */
        sw     k0, SRSCtlSave
        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 */
        /* If switching shadow sets, write new value to SRSCtl_PSS here */
        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 */
        /*
         * If switching shadow sets, clear only KSU above, write target
         * address to EPC, and do execute an eret to clear EXL, switch
         * shadow sets, and jump to routine
         */

        /* 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 */
        lw     k0, SRSCtlSave      /* Get saved SRSCtl */
        mtc0   k1, C0_EPC          /*  and EPC */
        mtc0   k0, C0_SRSCtl       /* Restore shadow sets */
        ehb                        /* Clear hazard */
        eret                       /* Dismiss the interrupt */
```

5.1.1.3 External Interrupt Controller Mode

External Interrupt Controller 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, and performance counter interrupts, and directly supplying to the processor the vector number (and optionally the priority level) 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 interrupt mode, the processor sends the state of the software interrupt requests ($Cause_{IP1..IP0}$), the timer interrupt request ($Cause_{TI}$), and the performance counter interrupt request ($Cause_{PCI}$) to the external interrupt controller, where it prioritizes these interrupts in a system-dependent way with other hardware interrupts. The interrupt controller can be a hard-wired logic block, or it can be configurable based on 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.

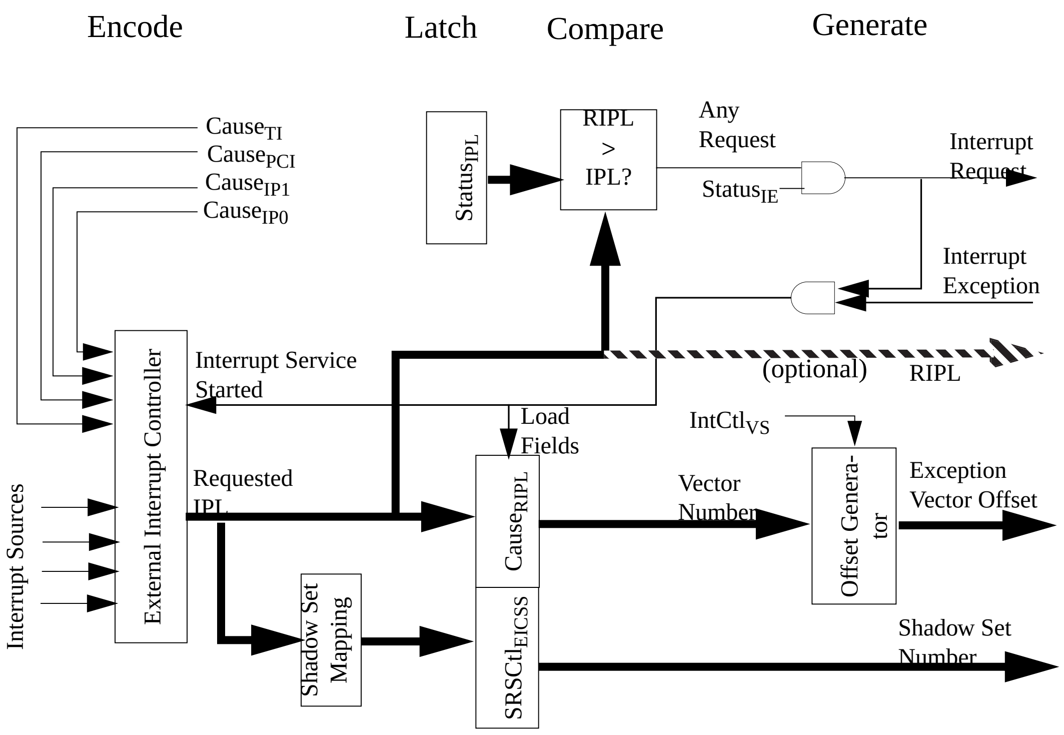
The external interrupt controller prioritizes its interrupt requests and produces the priority level and the vector number of the highest priority interrupt to be serviced. The priority level, called the Requested Interrupt Priority Level (RIPL), is a 6-bit encoded value in the range 0..63, inclusive.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

$\text{Status}_{\text{IPL}}$ (which overlays $\text{Status}_{\text{IM7..IM2}}$) is interpreted as the Interrupt Priority Level (IPL) at which the processor is currently operating (with a value of zero indicating that no interrupt is currently being serviced). When the interrupt controller requests service for an interrupt, the processor compares RIPL with $\text{Status}_{\text{IPL}}$ to determine if the requested interrupt has higher priority than the current IPL. If RIPL is strictly greater than $\text{Status}_{\text{IPL}}$, and interrupts are enabled ($\text{Status}_{\text{IE}}$ = 1, $\text{Status}_{\text{EXL}}$ = 0, and $\text{Status}_{\text{ERL}}$ = 0) an interrupt request is signaled to the pipeline. When the processor starts the interrupt exception, it loads RIPL into $\text{Cause}_{\text{RIPL}}$ (which overlays $\text{Cause}_{\text{IP7..IP2}}$) and signals the external interrupt controller to notify it that the request is being serviced. Because $\text{Cause}_{\text{RIPL}}$ is only loaded by the processor when an interrupt exception is signaled, it is available to software during interrupt processing. The vector number that the EIC passes into the core is combined with the $\text{IntCtl}_{\text{VS}}$ to determine where the interrupt service routines is located. The vector number is not stored in any software visible register. Some implementations may choose to use the RIPL as the vector number, but this is not a requirement.

In EIC interrupt mode, the external interrupt controller is also responsible for supplying the GPR shadow set number to use when servicing the interrupt. As such, the *SRSMap* register is not used in this mode, and the mapping of the vectored interrupt to a GPR shadow set is done by programming (or designing) the interrupt controller to provide the correct GPR shadow set number when an interrupt is requested. When the processor loads an interrupt request into $\text{Cause}_{\text{RIPL}}$, it also loads the GPR shadow set number into $\text{SRSCtl}_{\text{EICSS}}$, which is copied to $\text{SRSCtl}_{\text{CSS}}$ when the interrupt is serviced.

The operation of EIC interrupt mode is shown pictorially in Figure 5-2.

Figure 5-2 Interrupt Generation for External Interrupt Controller Interrupt Mode



A typical software handler for EIC 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. Unlike the compatibility mode examples, an EIC interrupt handler may take advantage of a dedicated GPR shadow set to avoid saving any registers. As such, the SimpleInterrupt code shown above need not save the GPRs.

A nested interrupt is similar to that shown for compatibility mode, but may also take advantage of running the nested exception routine in the GPR shadow set dedicated to the interrupt or in another shadow set. It also need only copy Cause$_{RIPL}$ to Status$_{IPL}$ to prevent lower priority interrupts from interrupting the handler. Such a routine might look as follows:

```
NestedException:
/*
 * Nested exceptions typically require saving the EPC, Status,and SRSCtl registers,
 * setting up the appropriate GPR shadow set for the 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 can not cover all nuances of this processing and is intended only
 * to demonstrate the concepts.
 */

   /* Use the current GPR shadow set, and setup software context */
   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 */
       mfc0   k1, C0_SRSCtl       /* Save SRSCtl if changing shadow sets */
       sw     k1, SRSCtlSave
       /* If switching shadow sets, write new value to SRSCtl$_{PSS}$ here */
       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 */
       /*
        * If switching shadow sets, clear only KSU above, write target
        * address to EPC, and do execute an eret to clear EXL, switch
        * shadow sets, and jump to routine
        */

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

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

5.1.2 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 interrupt 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, and Table 5.4 shows the exception vector offset for a representative subset of the vector numbers and values of the IntCtl$_{VS}$ field.

Table 5.4 Exception Vector Offsets for Vectored Interrupts

	Value of IntCtl$_{VS}$ Field				
Vector Number	**0b00001**	**0b00010**	**0b00100**	**0b01000**	**0b10000**
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
7	0x02E0	0x03C0	0x0580	0x0900	0x1000
• • •					

Table 5.4 Exception Vector Offsets for Vectored Interrupts

Vector Number	Value of $\text{IntCtl}_{\text{VS}}$ Field				
	0b00001	0b00010	0b00100	0b01000	0b10000
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:

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

5.1.2.1 Software Hazards and the Interrupt System

Software writes to certain coprocessor 0 register fields may change the conditions under which an interrupt is taken. This creates a coprocessor 0 (CP0) hazard, as described in the chapter "CP0 Hazards" on page 65. In Release 1 of the Architecture, there was no architecturally-defined method for bounding the number of instructions which would be executed after the instruction which caused the interrupt state change and before the change to the interrupt state was seen. In Release 2 of the Architecture, the EHB instruction was added, and this instruction can be used by software to clear the hazard.

Table 5.5 lists the CP0 register fields which can cause a change to the interrupt state (either enabling interrupts which were previously disabled or disabling interrupts which were previously enabled).

Table 5.5 Interrupt State Changes Made Visible by EHB

Instruction(s)	CP0 Register Written	CP0 Register Field(s) Modified
MTC0	Status	IM, IPL, ERL, EXL, IE
EI, DI	Status	IE
MTC0	Cause	$\text{IP}_{1..0}$
MTC0	PerfCnt Control	IE
MTC0	PerfCnt Counter	Event Count

An EHB, executed after one of these fields is modified by the listed instruction, makes the change to the interrupt state visible no later than the instruction following the EHB.

In the following example, a change to the Cause_{IM} field is made visible by an EHB:

```
mfc0   k0, C0_Status
ins    k0, zero, S_StatusIM4, 1      /* Clear bit 4 of the IM field */
mtc0   k0, C0_Status                 /* Re-write the register */
ehb                                  /* Clear the hazard */
/* Change to the interrupt state is seen no later than this instruction */
```

Similarly, the effects of an DI instruction are made visible by an EHB:

```
di                                   /* Disable interrupts */
```

```
ehb                                    /* Clear the hazard */
/* Change to the interrupt state is seen no later than this instruction */
```

5.2 Exceptions

Normal execution of instructions may be interrupted when an exception occurs. Such events can be generated as a by-product of instruction execution (e.g., an integer overflow caused by an add instruction or a TLB miss caused by a load instruction), or by an event not directly related to instruction execution (e.g., an external interrupt). When an exception occurs, the processor stops processing instructions, saves sufficient state to resume the interrupted instruction stream, enters Kernel Mode, and starts a software exception handler. The saved state and the address of the software exception handler are a function of both the type of exception, and the current state of the processor.

5.2.1 Exception Priority

Table 5.6 lists all possible exceptions, and the relative priority of each, highest to lowest.

Table 5.6 Priority of Exceptions

Exception	Description	Type
Reset	The Cold Reset signal was asserted to the processor	Asynchronous Reset
Soft Reset	The Reset signal was asserted to the processor	
Debug Single Step	An EJTAG Single Step occurred. Prioritized above other exceptions, including asynchronous exceptions, so that one can single-step into interrupt (or other asynchronous) handlers.	Synchronous Debug
Debug Interrupt	An EJTAG interrupt (EjtagBrk or DINT) was asserted.	Asynchronous Debug
Imprecise Debug Data Break	An imprecise EJTAG data break condition was asserted.	
Nonmaskable Interrupt (NMI)	The NMI signal was asserted to the processor.	Asynchronous
Machine Check	An internal inconsistency was detected by the processor.	
Interrupt	An enabled interrupt occurred.	
Deferred Watch	A watch exception, deferred because EXL was one when the exception was detected, was asserted after EXL went to zero.	
Debug Instruction Break	An EJTAG instruction break condition was asserted. Prioritized above instruction fetch exceptions to allow break on illegal instruction addresses.	Synchronous Debug
Watch - Instruction fetch	A watch address match was detected on an instruction fetch. Prioritized above instruction fetch exceptions to allow watch on illegal instruction addresses.	Synchronous
Address Error - Instruction fetch	A non-word-aligned address was loaded into PC.	
TLB Refill - Instruction fetch	A TLB miss occurred on an instruction fetch.	
TLB Invalid - Instruction fetch	The valid bit was zero in the TLB entry mapping the address referenced by an instruction fetch.	
Cache Error - Instruction fetch	A cache error occurred on an instruction fetch.	
Bus Error - Instruction fetch	A bus error occurred on an instruction fetch.	

Table 5.6 Priority of Exceptions

Exception	Description	Type
SDBBP	An EJTAG SDBBP instruction was executed.	Synchronous Debug
Instruction Validity Exceptions	An instruction could not be completed because it was not allowed access to the required resources, or was illegal: Coprocessor Unusable, Reserved Instruction. If both exceptions occur on the same instruction, the Coprocessor Unusable Exception takes priority over the Reserved Instruction Exception.	Synchronous
Execution Exception	An instruction-based exception occurred: Integer overflow, trap, system call, breakpoint, floating point, coprocessor 2 exception.	
Precise Debug Data Break	A precise EJTAG data break on load/store (address match only) or a data break on store (address+data match) condition was asserted. Prioritized above data fetch exceptions to allow break on illegal data addresses.	Synchronous Debug
Watch - Data access	A watch address match was detected on the address referenced by a load or store. Prioritized above data fetch exceptions to allow watch on illegal data addresses.	Synchronous
Address error - Data access	An unaligned address, or an address that was inaccessible in the current processor mode was referenced, by a load or store instruction	
TLB Refill - Data access	A TLB miss occurred on a data access	
TLB Invalid - Data access	The valid bit was zero in the TLB entry mapping the address referenced by a load or store instruction	
TLB Modified - Data access	The dirty bit was zero in the TLB entry mapping the address referenced by a store instruction	
Cache Error - Data access	A cache error occurred on a load or store data reference	Synchronous or Asynchronous
Bus Error - Data access	A bus error occurred on a load or store data reference	
Precise Debug Data Break	A precise EJTAG data break on load (address+data match only) condition was asserted. Prioritized last because all aspects of the data fetch must complete in order to do data match.	Synchronous Debug

The “Type” column of Table 5.7 describes the type of exception. Table 5.8 explains the characteristics of each exception type.

Table 5.7 Exception Type Characteristics

Exception Type	Characteristics
Asynchronous Reset	Denotes a reset-type exception that occurs asynchronously to instruction execution. These exceptions always have the highest priority to guarantee that the processor can always be placed in a runnable state.
Asynchronous Debug	Denotes an EJTAG debug exception that occurs asynchronously to instruction execution. These exceptions have very high priority with respect to other exceptions because of the desire to enter Debug Mode, even in the presence of other exceptions, both asynchronous and synchronous.

Table 5.7 Exception Type Characteristics

Exception Type	Characteristics
Asynchronous	Denotes any other type of exception that occurs asynchronously to instruction execution. These exceptions are shown with higher priority than synchronous exceptions mainly for notational convenience. If one thinks of asynchronous exceptions as occurring between instructions, they are either the lowest priority relative to the previous instruction, or the highest priority relative to the next instruction. The ordering of the table above considers them in the second way.
Synchronous Debug	Denotes an EJTAG debug exception that occurs as a result of instruction execution, and is reported precisely with respect to the instruction that caused the exception. These exceptions are prioritized above other synchronous exceptions to allow entry to Debug Mode, even in the presence of other exceptions.
Synchronous	Denotes any other exception that occurs as a result of instruction execution, and is reported precisely with respect to the instruction that caused the exception. These exceptions tend to be prioritized below other types of exceptions, but there is a relative priority of synchronous exceptions with each other.

5.2.2 Exception Vector Locations

The Reset, Soft Reset, and NMI exceptions are always vectored to location `0xBFC0.0000`. EJTAG Debug exceptions are vectored to location `0xBFC0.0480`, or to location `0xFF20.0200` if the ProbTrap bit is zero or one, respectively, in the EJTAG_Control_register.

Addresses for all other exceptions are a combination of a vector offset and a vector base address. In Release 1 of the architecture, the vector base address was fixed. In Release 2 of the architecture, software is allowed to specify the vector base address via the *EBase* register for exceptions that occur when $\text{Status}_{\text{BEV}}$ equals 0. Table 5.8 gives the vector base address as a function of the exception and whether the *BEV* bit is set in the *Status* register. Table 5.9 gives 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. For implementations of Release 2 of the Architecture, Table 5.4 gives the offset from the base address in the case where $\text{Status}_{\text{BEV}}$ = 0 and Cause_{IV} = 1. For implementations of Release 1 of the architecture in which Cause_{IV} = 1, the vector offset is as if $\text{IntCtl}_{\text{VS}}$ were 0.

Table 5.10 combines these two 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, the vector address value assumes that the *EBase* register, as implemented in Release 2 devices, is not changed from its reset state and that $\text{IntCtl}_{\text{VS}}$ is 0.

In Release 2 of the Architecture, software must guarantee that $\text{EBase}_{15..12}$ contains zeros in all bit positions less than or equal to the most significant bit in the vector offset. This situation can only occur when a vector offset greater than 0xFFF is generated when an interrupt occurs with VI or EIC interrupt mode enabled. The operation of the processor is **UNDEFINED** if this condition is not met.

Table 5.8 Exception Vector Base Addresses

Exception	$Status_{BEV}$	
	0	1
Reset, Soft Reset, NMI	0xBFC0.0000	
EJTAG Debug (with ProbTrap = 0 in the EJTAG_Control_register)	0xBFC0.0480	
EJTAG Debug (with ProbTrap = 1 in the EJTAG_Control_register)	0xFF20.0200	
Cache Error	*For Release 1 of the architecture:* 0xA000.0000 *For Release 2 of the architecture:* $EBase_{31..30}$ || 1 || $EBase_{28..12}$ || 0x000 Note that $EBase_{31..30}$ have the fixed value 0b10	0xBFC0.0200
Other	*For Release 1 of the architecture:* 0x8000.0000 *For Release 2 of the architecture:* $EBase_{31..12}$ || 0x000 Note that $EBase_{31..30}$ have the fixed value 0b10	0xBFC0.0200

Table 5.9 Exception Vector Offsets

Exception	Vector Offset
TLB Refill, EXL = 0	0x000
Cache error	0x100
General Exception	0x180
Interrupt, $Cause_{IV}$ = 1	0x200 (In Release 2 implementations, this is the base of the vectored interrupt table when $Status_{BEV}$ = 0)
Reset, Soft Reset, NMI	None (Uses Reset Base Address)

Table 5.10 Exception Vectors

Exception	$\text{Status}_{\text{BEV}}$	$\text{Status}_{\text{EXL}}$	Cause_{IV}	EJTAG ProbEn	Vector For Release 2 Implementations, assumes that EBase retains its reset state and that $\text{IntCtl}_{\text{VS}}$ = 0
Reset, Soft Reset, NMI	x	x	x	x	`0xBFC0.0000`
EJTAG Debug	x	x	x	0	`0xBFC0.0480`
EJTAG Debug	x	x	x	1	`0xFF20.0200`
TLB Refill	0	0	x	x	`0x8000.0000`
TLB Refill	0	1	x	x	`0x8000.0180`
TLB Refill	1	0	x	x	`0xBFC0.0200`
TLB Refill	1	1	x	x	`0xBFC0.0380`
Cache Error	0	x	x	x	`0xA000.0100`
Cache Error	1	x	x	x	`0xBFC0.0300`
Interrupt	0	0	0	x	`0x8000.0180`
Interrupt	0	0	1	x	`0x8000.0200`
Interrupt	1	0	0	x	`0xBFC0.0380`
Interrupt	1	0	1	x	`0xBFC0.0400`
All others	0	x	x	x	`0x8000.0180`
All others	1	x	x	x	`0xBFC0.0380`
‘x’ denotes don’t care					

5.2.3 General Exception Processing

With the exception of Reset, Soft 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 (see Table 8.26 on page 110). 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 5.11 shows the value stored in each of the CP0 PC registers, including *EPC*. For implementations of Release 2 of the Architecture if $\text{Status}_{\text{BEV}}$ = 0, the *CSS* field in the *SRSCtl* register is copied to the *PSS* field, and the *CSS* value is loaded from the appropriate source.

  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. For implementations of Release 2 of the Architecture, the *SRSCtl* register is not changed.

.

Table 5.11 Value Stored in EPC, ErrorEPC, or DEPC on an 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 is started 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 EPC nor Cause_BD nor SRSCtl are modified */
if Status_EXL = 1 then
    vectorOffset ← 0x180
else
    if InstructionInBranchDelaySlot then
        EPC ← restartPC/* 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 */
    NewShadowSet ← SRSCtl_ESS          /* Assume exception, Release 2 only */
    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 ← $Cause_{RIPL}$
                    NewShadowSet ← $SRSCtl_{EICSS}$
                else
                    VecNum ← VIntPriorityEncoder()
                    NewShadowSet ← $SRSMap_{IPL \times 4+3..IPL \times 4}$
                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 */

    /* Update the shadow set information for an implementation of */
    /* Release 2 of the architecture */
    if (ArchitectureRevision ≥ 2) and ($SRSCtl_{HSS}$ > 0) and ($Status_{BEV}$ = 0) then
        $SRSCtl_{PSS}$ ← $SRSCtl_{CSS}$
        $SRSCtl_{CSS}$ ← NewShadowSet
    endif
endif /* if $Status_{EXL}$ = 1 then */

$Cause_{CE}$ ← FaultingCoprocessorNumber
$Cause_{ExcCode}$ ← ExceptionType
$Status_{EXL}$ ← 1


/* 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. Vector */
/* offsets > 0xFFF (vectored or EIC interrupts only), require */
/* that $EBase_{15..12}$ have zeros in each bit position less than or */
/* equal to the most significant bit position of the vector offset */
PC ← $vectorBase_{31..30}$ ‖ ($vectorBase_{29..0}$ + $vectorOffset_{29..0}$)
                                /* No carry between bits 29 and 30 */
```

5.2.4 EJTAG Debug Exception

An EJTAG Debug Exception occurs when one of a number of EJTAG-related conditions is met. Refer to the EJTAG Specification for details of this exception.

Entry Vector Used

`0xBFC0 0480` if the *ProbTrap* bit is zero in the EJTAG_Control_register; `0xFF20 0200` if the *ProbTrap* bit is one.

5.2.5 Reset Exception

A Reset Exception occurs when the Cold 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, only the following registers have defined state:

- The *Random* register is initialized to the number of TLB entries - 1.
- The *Wired* register is initialized to zero.
- The *Config*, *Config1*, *Config2*, and *Config3* registers are initialized with their boot state.
- The *RP*, *BEV*, *TS*, *SR*, *NMI*, and *ERL* fields of the *Status* register are initialized to a specified state.
- Watch register enables and Performance Counter register interrupt enables are cleared.
- The *ErrorEPC* register is loaded with the restart PC, as described in Table 5.11. Note that this value may or may not be predictable if the Reset Exception was taken as the result of power being applied to the processor because PC may not have a valid value in that case. In some implementations, the value loaded into *ErrorEPC* register may not be predictable on either a Reset or Soft Reset Exception.
- PC is loaded with `0xBFC0 0000`.

Cause Register ExcCode Value

None

Additional State Saved

None

Entry Vector Used

Reset (`0xBFC0 0000`)

Operation

```
Random ← TLBEntries - 1
PageMask$_{\text{MaskX}}$ ← 0            # 1KB page support implemented
PageGrain$_{\text{ESP}}$ ← 0             # 1KB page support implemented
Wired ← 0
HWREna ← 0
EntryHi$_{\text{VPN2X}}$ ← 0             # 1KB page support implemented
Status$_{\text{RP}}$ ← 0
Status$_{\text{BEV}}$ ← 1
Status$_{\text{TS}}$ ← 0
Status$_{\text{SR}}$ ← 0
Status$_{\text{NMI}}$ ← 0
Status$_{\text{ERL}}$ ← 1
IntCtl$_{\text{VS}}$ ← 0
SRSCtl$_{\text{HSS}}$ ← HighestImplementedShadowSet
SRSCtl$_{\text{ESS}}$ ← 0
SRSCtl$_{\text{PSS}}$ ← 0
SRSCtl$_{\text{CSS}}$ ← 0
SRSMap ← 0
Cause$_{\text{DC}}$ ← 0
```

```
EBase$_{ExceptionBase}$ ← 0
Config ← ConfigurationState
Config$_{K0}$ ← 2                   # Suggested - see Config register description
Config1 ← ConfigurationState
Config2 ← ConfigurationState
Config3 ← ConfigurationState
WatchLo[n]$_I$ ← 0                  # For all implemented Watch registers
WatchLo[n]$_R$ ← 0                  # For all implemented Watch registers
WatchLo[n]$_W$ ← 0                  # For all implemented Watch registers
PerfCnt.Control[n]$_{IE}$ ← 0       # For all implemented PerfCnt registers
if InstructionInBranchDelaySlot then
   ErrorEPC ← restartPC # PC of branch/jump
else
   ErrorEPC ← restartPC # PC of instruction
endif
PC ← 0xBFC0 0000
```

5.2.6 Soft Reset Exception

A Soft Reset Exception occurs when the Reset signal is asserted to the processor. This exception is not maskable. When a Soft Reset Exception occurs, the processor performs a subset of the full reset initialization. Although a Soft Reset Exception does not unnecessarily change the state of the processor, it may be forced to do so in order to place the processor in a state in which it can execute instructions from uncached, unmapped address space. Since bus, cache, or other operations may be interrupted, portions of the cache, memory, or other processor state may be inconsistent.

The primary difference between the Reset and Soft Reset Exceptions is in actual use. The Reset Exception is typically used to initialize the processor on power-up, while the Soft Reset Exception is typically used to recover from a non-responsive (hung) processor. The semantic difference is provided to allow boot software to save critical coprocessor 0 or other register state to assist in debugging the potential problem. As such, the processor may reset the same state when either reset signal is asserted, but the interpretation of any state saved by software may be very different.

In addition to any hardware initialization required, the following state is established on a Soft Reset Exception:

- The *RP*, *BEV*, *TS*, *SR*, *NMI*, and *ERL* fields of the *Status* register are initialized to a specified state.
- Watch register enables and Performance Counter register interrupt enables are cleared.
- The *ErrorEPC* register is loaded with the restart PC, as described in Table 5.11.
- PC is loaded with `0xBFC0 0000`.

Cause Register ExcCode Value

None

Additional State Saved

None

Entry Vector Used

Reset (`0xBFC0 0000`)

Operation

```
PageMask$_{MaskX}$ ← 0              # 1KB page support implemented
```

```
PageGrain$_{ESP}$ ← 0              # 1KB page support implemented
EntryHi$_{VPN2X}$ ← 0              # 1KB page support implemented
Config$_{K0}$ ← 2                  # Suggested - see Config register description
Status$_{RP}$ ← 0
Status$_{BEV}$ ← 1
Status$_{TS}$ ← 0
Status$_{SR}$ ← 1
Status$_{NMI}$ ← 0
Status$_{ERL}$ ← 1
WatchLo[n]$_{I}$ ← 0               # For all implemented Watch registers
WatchLo[n]$_{R}$ ← 0               # For all implemented Watch registers
WatchLo[n]$_{W}$ ← 0               # For all implemented Watch registers
PerfCnt.Control[n]$_{IE}$ ← 0      # For all implemented PerfCnt registers
if InstructionInBranchDelaySlot then
    ErrorEPC ← restartPC # PC of branch/jump
else
    ErrorEPC ← restartPC # PC of instruction
endif
PC ← 0xBFC0 0000
```

5.2.7 Non Maskable Interrupt (NMI) Exception

A non maskable interrupt exception occurs when the NMI signal is asserted to the processor.

Although described as an interrupt, it is more correctly described as an exception because it is not maskable. An NMI occurs only at instruction boundaries, so does not do any reset or other hardware initialization. The state of the cache, memory, and other processor state is 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 restart PC, as described in Table 5.11.
- PC is loaded with `0xBFC0 0000`.

Cause Register ExcCode Value

None

Additional State Saved

None

Entry Vector Used

Reset (`0xBFC0 0000`)

Operation

```
Status$_{BEV}$ ← 1
Status$_{TS}$ ← 0
Status$_{SR}$ ← 0
Status$_{NMI}$ ← 1
Status$_{ERL}$ ← 1
if InstructionInBranchDelaySlot then
    ErrorEPC ← restartPC # PC of branch/jump
else
    ErrorEPC ← restartPC # PC of instruction
endif
```

```
PC ← 0xBFC0 0000
```

5.2.8 Machine Check Exception

A machine check exception occurs when the processor detects an internal inconsistency.

The following conditions cause a machine check exception:

- Detection of multiple matching entries in the TLB in a TLB-based MMU.

Cause Register ExcCode Value

MCheck (See Table 8.27 on page 113)

Additional State Saved

Depends on the condition that caused the exception. See the descriptions above.

Entry Vector Used

General exception vector (offset 0x180)

5.2.9 Address Error Exception

An address error exception occurs under the following circumstances:

- An instruction is fetched from an address that is not aligned on a word boundary.
- A load or store word instruction is executed in which the address is not aligned on a word boundary.
- A load or store halfword instruction is executed in which the address is not aligned on a halfword boundary.
- A reference is made to a kernel address space from User Mode or Supervisor Mode.
- A reference is made to a supervisor address space from User Mode.

Note that in the case of an instruction fetch that is not aligned on a word boundary, the PC is updated before the condition is detected. Therefore, both *EPC* and *BadVAddr* point at the unaligned instruction address.

Cause Register ExcCode Value

AdEL: Reference was a load or an instruction fetch

AdES: Reference was a store

See Table 8.27 on page 113.

Additional State Saved

Register State	Value
BadVAddr	failing address
Context$_{VPN2}$	UNPREDICTABLE
EntryHi$_{VPN2}$	UNPREDICTABLE
EntryLo0	UNPREDICTABLE

Register State	Value
EntryLo1	UNPREDICTABLE

Entry Vector Used

General exception vector (offset 0x180)

5.2.10 TLB Refill Exception

A TLB Refill exception occurs in a TLB-based MMU when no TLB entry matches a reference to a mapped address space and the *EXL* bit is zero in the *Status* register. Note that this is distinct from the case in which an entry matches but has the valid bit off, in which case a TLB Invalid exception occurs.

***Cause* Register ExcCode Value**

TLBL: Reference was a load or an instruction fetch

TLBS: Reference was a store

See Table 8.27 on page 113.

Additional State Saved

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

5.2.11 TLB Invalid Exception

A TLB invalid exception occurs when a TLB entry matches a reference to a mapped address space, but the matched entry has the valid bit off.

Note that the condition in which no TLB entry matches a reference to a mapped address space and the *EXL* bit is one in the *Status* register is indistinguishable from a TLB Invalid Exception in the sense that both use the general exception vector and supply an ExcCode value of TLBL or TLBS. The only way to distinguish these two cases is by probing the TLB for a matching entry (using TLBP).

***Cause* Register ExcCode Value**

TLBL: Reference was a load or an instruction fetch

TLBS: Reference was a store

See Table 8.26 on page 110.

Additional State Saved

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)

5.2.12 TLB Modified Exception

A TLB modified exception occurs on a *store* reference to a mapped address when the matching TLB entry is valid, but the entry's *D* bit is zero, indicating that the page is not writable.

Cause Register ExcCode Value

Mod (See Table 8.26 on page 110)

Additional State Saved

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)

5.2.13 Cache Error Exception

A cache error exception occurs when an instruction or data reference detects a cache tag or data error, or a parity or ECC error is detected on the system bus when a cache miss occurs. This exception is not maskable. Because the error was in a cache, the exception vector is to an unmapped, uncached address.

Cause Register ExcCode Value

N/A

Additional State Saved

Register State	Value
CacheErr	Error state
ErrorEPC	Restart PC

Entry Vector Used

Cache error vector (offset 0x100)

Operation

```
CacheErr ← ErrorState
Status_ERL ← 1
if InstructionInBranchDelaySlot then
   ErrorEPC ← restartPC # PC of branch/jump
else
   ErrorEPC ← restartPC # PC of instruction
endif
if Status_BEV = 1 then
   PC ← 0xBFC0 0200 + 0x100
else
   if ArchitectureRevision ≥ 2 then
      /* The fixed value of EBase_31..30 and bit 29 forced to a 1 puts the */
      /* vector in kseg1 */
      PC ← EBase_31..30 || 1 || EBase_28..12 || 0x100
   else
      PC ← 0xA000 0000 + 0x100
   endif
endif
```

5.2.14 Bus Error Exception

A bus error occurs when an instruction, data, or prefetch access makes a bus request (due to a cache miss or an uncacheable reference) and that request is terminated in an error. Note that parity errors detected during bus transactions are reported as cache error exceptions, not bus error exceptions.

Cause Register ExcCode Value

IBE: Error on an instruction reference

DBE: Error on a data reference

See Table 8.27 on page 113.

Additional State Saved

None

Entry Vector Used

General exception vector (offset 0x180)

5.2.15 Integer Overflow Exception

An integer overflow exception occurs when selected integer instructions result in a 2’s complement overflow.

Cause Register ExcCode Value

Ov (See Table 8.27 on page 113)

Additional State Saved

None

Entry Vector Used

General exception vector (offset 0x180)

5.2.16 Trap Exception

A trap exception occurs when a trap instruction results in a TRUE value.

Cause Register ExcCode Value

Tr (See Table 8.27 on page 113)

Additional State Saved

None

Entry Vector Used

General exception vector (offset 0x180)

5.2.17 System Call Exception

A system call exception occurs when a SYSCALL instruction is executed.

Cause Register ExcCode Value

Sys (See Table 8.26 on page 110)

Additional State Saved

None

Entry Vector Used

General exception vector (offset 0x180)

5.2.18 Breakpoint Exception

A breakpoint exception occurs when a BREAK instruction is executed.

Cause Register ExcCode Value

Bp (See Table 8.27 on page 113)

Additional State Saved

None

Entry Vector Used

General exception vector (offset 0x180)

5.2.19 Reserved Instruction Exception

A Reserved Instruction Exception occurs if any of the following conditions is true:

- An instruction was executed that specifies an encoding of the opcode field that is flagged with “*” (reserved), “β” (higher-order ISA), or an unimplemented “ε” (ASE).
- An instruction was executed that specifies a *SPECIAL* opcode encoding of the function field that is flagged with “*” (reserved), or “β” (higher-order ISA).
- An instruction was executed that specifies a *REGIMM* opcode encoding of the rt field that is flagged with “*” (reserved).
- An instruction was executed that specifies an unimplemented *SPECIAL2* opcode encoding of the function field that is flagged with an unimplemented “θ” (partner available), or an unimplemented “σ” (EJTAG).
- An instruction was executed that specifies a *COPz* opcode encoding of the rs field that is flagged with “*” (reserved), “β” (higher-order ISA), or an unimplemented “ε” (ASE), assuming that access to the coprocessor is allowed. If access to the coprocessor is not allowed, a Coprocessor Unusable Exception occurs instead. For the *COP1* opcode, some implementations of previous ISAs reported this case as a Floating Point Exception, setting the Unimplemented Operation bit in the Cause field of the *FCSR* register.
- An instruction was executed that specifies an unimplemented *COP0* opcode encoding of the function field when rs is *CO* that is flagged with “*” (reserved), or an unimplemented “σ” (EJTAG), assuming that access to coprocessor 0 is allowed. If access to the coprocessor is not allowed, a Coprocessor Unusable Exception occurs instead.
- An instruction was executed that specifies a *COP1* opcode encoding of the function field that is flagged with “*” (reserved), “β” (higher-order ISA), or an unimplemented “ε” (ASE), assuming that access to coprocessor 1 is allowed. If access to the coprocessor is not allowed, a Coprocessor Unusable Exception occurs instead. Some implementations of previous ISAs reported this case as a Floating Point Exception, setting the Unimplemented Operation bit in the Cause field of the *FCSR* register.

Cause Register ExcCode Value

RI (See Table 8.27 on page 113)

Additional State Saved

None

Entry Vector Used

General exception vector (offset 0x180)

5.2.20 Coprocessor Unusable Exception

A coprocessor unusable exception occurs if any of the following conditions is true:

- A COP0 or Cache instruction was executed while the processor was running in a mode other than Debug Mode or Kernel Mode, and the *CU0* bit in the *Status* register was a zero
- A COP1, COP1X,LWC1, SWC1, LDC1, SDC1 or MOVCI (Special opcode function field encoding) instruction was executed and the *CU1* bit in the *Status* register was a zero.

- A COP2, LWC2, SWC2, LDC2, or SDC2 instruction was executed, and the *CU2* bit in the *Status* register was a zero.

NOTE: In Release 2 of the MIPS32 Architecture, the use of COP3 as a user-defined coprocessor has been removed. The use of COP3 is reserved for the future extension of the architecture.

Cause Register ExcCode Value

CpU (See Table 8.26 on page 110)

Additional State Saved

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

Entry Vector Used

General exception vector (offset 0x180)

5.2.21 Floating Point Exception

A floating point exception is initiated by the floating point coprocessor to signal a floating point exception.

Register ExcCode Value

FPE (See Table 8.26 on page 110)

Additional State Saved

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

Entry Vector Used

General exception vector (offset 0x180)

5.2.22 Coprocessor 2 Exception

A coprocessor 2 exception is initiated by coprocessor 2 to signal a precise coprocessor 2 exception.

Register ExcCode Value

C2E (See Table 8.26 on page 110)

Additional State Saved

Defined by the coprocessor

Entry Vector Used

General exception vector (offset 0x180)

5.2.23 Watch Exception

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. If either bit is a one at the time that a watch exception would normally be taken, the *WP* bit in the *Cause* register is set, and the exception is deferred until both the *EXL* and *ERL* bits in the *Status* register 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.

If the *EXL* or *ERL* bits are one in the *Status* register and a single instruction generates both a watch exception (which is deferred by the state of the *EXL* and *ERL* bits) and a lower-priority exception, the lower priority exception is taken.

Watch exceptions are never taken if the processor is executing in Debug Mode. Should a watch register match while the processor is in Debug Mode, the exception is inhibited and the *WP* bit is not changed.

It is implementation dependent whether a data watch exception is triggered by a prefetch or cache instruction whose address matches the Watch register address match conditions. A watch triggered by a SC instruction does so even if the store would not complete because the *LL* bit is zero.

Register ExcCode Value

WATCH (See Table 8.26 on page 110)

Additional State Saved

Register State	Value
Cause_{WP}	indicates that the watch exception was deferred until after both $\text{Status}_{\text{EXL}}$ and $\text{Status}_{\text{ERL}}$ 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.

Entry Vector Used

General exception vector (offset 0x180)

5.2.24 Interrupt Exception

The interrupt exception occurs when an enabled request for interrupt service is made. See Section 5.1   on page 31 for more information.

Register ExcCode Value

Int (See Table 8.27 on page 113)

Additional State Saved

Register State	Value
Cause_{IP}	indicates the interrupts that are pending.

Entry Vector Used

General exception vector (offset 0x180) if the *IV* bit in the *Cause* register is zero.

Interrupt vector (offset 0x200) if the *IV* bit in the *Cause* register is one.

Chapter 6

GPR Shadow Registers

The capability in this chapter is targeted at removing the need to save and restore GPRs on entry to high priority interrupts or exceptions, and to provide specified processor modes with the same capability. This is done by introducing multiple copies of the GPRs, called *shadow sets*, and allowing privileged software to associate a shadow set with entry to Kernel Mode via an interrupt vector or exception. The normal GPRs are logically considered shadow set zero.

The number of GPR shadow sets is implementation dependent and may range from one (the normal GPRs) to an architectural maximum of 16. The highest number actually implemented is indicated by the $\text{SRSCtl}_{\text{HSS}}$ field, and all shadow sets between 0 and $\text{SRSCtl}_{\text{HSS}}$, inclusive must be implemented. If this field is zero, only the normal GPRs are implemented.

6.1 Introduction to Shadow Sets

Shadow sets are new copies of the GPRs that can be substituted for the normal GPRs on entry to Kernel Mode via an interrupt or exception. Once a shadow set is bound to a Kernel Mode entry condition, reference to GPRs work exactly as one would expect, but they are redirected to registers that are dedicated to that condition. Privileged software may need to reference all GPRs in the register file, even specific shadow registers that are not visible in the current mode. The RDPGPR and WRPGPR instructions are used for this purpose. The CSS field of the *SRSCtl* register provides the number of the current shadow register set, and the PSS field of the *SRSCtl* register provides the number of the previous shadow register set (that which was current before the last exception or interrupt occurred).

If the processor is operating in VI interrupt mode, binding of a vectored interrupt to a shadow set is done by writing to the *SRSMap* register. If the processor is operating in EIC interrupt mode, the binding of the interrupt to a specific shadow set is provided by the external interrupt controller, and is configured in an implementation-dependent way. Binding of an exception or non-vectored interrupt to a shadow set is done by writing to the ESS field of the *SRSCtl* register. When an exception or interrupt occurs, the value of $\text{SRSCtl}_{\text{CSS}}$ is copied to $\text{SRSCtl}_{\text{PSS}}$, and $\text{SRSCtl}_{\text{CSS}}$ is set to the value taken from the appropriate source. On an ERET, the value of $\text{SRSCtl}_{\text{PSS}}$ is copied back into $\text{SRSCtl}_{\text{CSS}}$ to restore the shadow set of the mode to which control returns. More precisely, the rules for updating the fields in the *SRSCtl* register on an interrupt or exception are as follows:

1. No field in the *SRSCtl* register is updated if any of the following conditions are true. In this case, steps 2 and 3 are skipped.

   - The exception is one that sets $\text{Status}_{\text{ERL}}$: NMI or cache error.

   - The exception causes entry into EJTAG Debug Mode

   - $\text{Status}_{\text{BEV}} = 1$

   - $\text{Status}_{\text{EXL}} = 1$

2. $\text{SRSCtl}_{\text{CSS}}$ is copied to $\text{SRSCtl}_{\text{PSS}}$

3. $\text{SRSCtl}_{\text{CSS}}$ is updated from one of the following sources:

   - The appropriate field of the *SRSMap* register, based on IPL, if the exception is an interrupt, $\text{Cause}_{\text{IV}} = 1$, $\text{IntCtl}_{\text{VSS}} \neq 0$, $\text{Config3}_{\text{VEIC}} = 0$, and $\text{Config3}_{\text{VInt}} = 1$. These are the conditions for a vectored interrupt.

   - The EICSS field of the *SRSCtl* register if the exception is an interrupt, $\text{Cause}_{\text{IV}} = 1$, $\text{IntCtl}_{\text{VSS}} \neq 0$, and $\text{Config3}_{\text{VEIC}} = 1$. These are the conditions for a vectored EIC interrupt.

   - The ESS field of the *SRSCtl* register in any other case. This is the condition for a non-interrupt exception, or a non-vectored interrupt.

Similarly, the rules for updating the fields in the SRSCtl register at the end of an exception or interrupt are as follows:

1. No field in the *SRSCtl* register is updated if any of the following conditions is true. In this case, step 2 is skipped.

   - A DERET is executed

   - An ERET is executed with $\text{Status}_{\text{ERL}} = 1$ or $\text{Status}_{\text{BEV}} = 1$

2. $\text{SRSCtl}_{\text{PSS}}$ is copied to $\text{SRSCtl}_{\text{CSS}}$

These rules have the effect of preserving the *SRSCtl* register in any case of a nested exception or one which occurs before the processor has been fully initialize ($\text{Status}_{\text{BEV}} = 1$).

Privileged software may switch the current shadow set by writing a new value into $\text{SRSCtl}_{\text{PSS}}$, loading EPC with a target address, and doing an ERET.

6.2 Support Instructions

Table 6.1 Instructions Supporting Shadow Sets

Mnemonic	Function	MIPS64 Only?
RDPGPR	Read GPR From Previous Shadow Set	No
WRPGPR	Write GPR to Shadow Set	No

Chapter 7

CP0 Hazards

7.1 Introduction

Because resources controlled via Coprocessor 0 affect the operation of various pipeline stages of a MIPS32 processor, manipulation of these resources may produce results that are not detectable by subsequent instructions for some number of execution cycles. When no hardware interlock exists between one instruction that causes an effect that is visible to a second instruction, a *CP0 hazard* exists.

In Release 1 of the MIPS32® Architecture, CP0 hazards were relegated to implementation-dependent cycle-based solutions, primarily based on the SSNOP instruction. Since that time, it has become clear that this is an insufficient and error-prone practice that must be addressed with a firm compact between hardware and software. As such, new instructions have been added to Release 2 of the architecture which act as explicit barriers that eliminate hazards. To the extent that it was possible to do so, the new instructions have been added in such a way that they are backward-compatible with existing MIPS processors.

7.2 Types of Hazards

In privileged software, there are two different types of hazards: execution hazards and instruction hazards. Both are defined below.

Implementations using Release 1 of the architecture should refer to their Implementation documentation for the required instruction “spacing” that is required to eliminate these hazards.

Note that, for superscalar MIPS implementations, the number of instructions issued per cycle may be greater than one, and thus that the duration of the hazard in instructions may be greater than the duration in cycles. It is for this reason that MIPS32 Release 1 defines the SSNOP instruction to convert instruction issues to cycles in a superscalar design.

7.2.1 Execution Hazards

Execution hazards are those created by the execution of one instruction, and seen by the execution of another instruction. Table 7.1 lists execution hazards.

Table 7.1 Execution Hazards

Producer	→	Consumer	Hazard On
Hazards Related to the TLB			
MTC0	→	TLBR, TLBWI, TLBWR	EntryHi

Table 7.1 Execution Hazards

Producer	→	Consumer	Hazard On
MTC0	→	TLBWI, TLBWR	EntryLo0, EntryLo1, Index, PageMask, PageGrain
MTCO	→	TLBWR	Wired
MTC0	→	TLBP, Load or Store Instruction	EntryHi$_{\text{ASID}}$
MTC0	→	Load/store affected by new state	EntryHi$_{\text{ASID}}$, WatchHi, WatchLo, Config
TLBP	→	MFC0	Index
TLBR	→	MFC0	EntryHi, EntryLo0, EntryLo1, PageMask
TLBWI, TLBWR	→	TLBP, TLBR, Load/store using new TLB entry	TLB entry
Hazards Related to Exceptions or Interrupts			
MTC0	→	Coprocessor instruction execution depends on the new value of Status$_{\text{CU}}$	Status$_{\text{CU}}$
MTC0	→	ERET	DEPC, EPC, ErrorEPC, Status
MTC0	→	Interrupted Instruction	Cause$_{\text{IP}}$, Cause$_{\text{IV}}$ Compare, Count, PerfCnt Control$_{\text{IE}}$, PerfCnt Counter, Status$_{\text{IE}}$, Status$_{\text{IM}}$ EBase SRSCtl SRSMap

Table 7.1 Execution Hazards

Producer	→	Consumer	Hazard On
EI, DI	→	Interrupted Instruction	Status_{IE}, Status_{IM}
Other Hazards			
LL	→	MFC0	LLAddr
MTC0	→	CACHE	PageGrain

7.2.2 Instruction Hazards

Instruction hazards are those created by the execution of one instruction, and seen by the instruction fetch of another instruction. Table 7.2 lists instruction hazards.

Table 7.2 Instruction Hazards

Producer	→	Consumer	Hazard On
Hazards Related to the TLB			
MTC0	→	Instruction fetch seeing the new value	EntryHi_{ASID}, WatchHi, WatchLo Config
MTC0	→	Instruction fetch seeing the new value (including a change to ERL followed by an instruction fetch from the useg segment)	Status
TLBWI, TLBWR	→	Instruction fetch using new TLB entry	TLB entry
Hazards Related to Writing the Instruction Stream or Modifying an Instruction Cache Entry			
Instruction stream writes	→	Instruction fetch seeing the new instruction stream	Cache entries
CACHE	→	Instruction fetch seeing the new instruction stream	Cache entries
Other Hazards			
MTC0	→	RDPGPR WRPGPR	SRSCtl_{PSS}[1]

[1] This is not precisely a hazard on the instruction fetch. Rather it is a hazard on a modification to the previous GPR context field, followed by a previous-context reference to the GPRs. It is considered an instruction hazard rather than an execution hazard because some implementation may require that the previous GPR context be established early in the pipeline, and execution hazards are not meant to cover this case.

7.3 Hazard Clearing Instructions and Events

Table 7.3 lists the instructions designed to eliminate hazards.

Table 7.3 Hazard Clearing Instructions

Mnemonic	Function
DERET	Clear both execution and instruction hazards
EHB	Clear execution hazard
ERET	Clear both execution and instruction hazards
JALR.HB	Clear both execution and instruction hazards
JR.HB	Clear both execution and instruction hazards
SSNOP	Superscalar No Operation
SYNCI[1]	Synchronize caches after instruction stream write

[1] SYNCI synchronizes caches after an instruction stream write, and before execution of that instruction stream. As such, it is not precisely a coprocessor 0 hazard, but is included here for completeness.

DERET, ERET, and SSNOP are available in Release 1 of the Architecture; EHB, JALR.HB, JR.HB, and SYNCI were added in Release 2 of the Architecture. In both Release 1 and Release 2 of the Architecture, DERET and ERET clear both execution and instruction hazards and they are the only timing-independent instructions which will do this in both releases of the architecture.

Even though DERET and ERET clear hazards between the execution of the instruction and the target instruction stream, an execution hazard may still be created between a write of the *DEPC*, *EPC*, *ErrorEPC*, or *Status* registers and the DERET or ERET instruction.

In addition, an exception or interrupt also clears both execution and instruction hazards between the instruction that created the hazard and the first instruction of the exception or interrupt handler. Said another way, no hazards remain visible by the first instruction of an exception or interrupt handler.

7.3.1 Instruction Encoding

The EHB instruction is encoded using a variant of the NOP/SSNOP encoding. This encoding was chosen for compatibility with the Release 1 SSNOP instruction, such that existing software may be modified to be compatible with both Release 1 and Release 2 implementations. See the EHB instruction description for additional information.

The JALR.HB and JR.HB instructions are encoding using bit 10 of the *hint* field of the JALR and JR instructions. These encodings were chosen for compatibility with existing MIPS implementations, including many which pre-date the MIPS32 architecture. Because a pipeline flush clears hazards on most early implementations, the JALR.HB or JR.HB instructions can be included in existing software for backward and forward compatibility. See the JALR.HB and JR.HB instructions for additional information.

The SYNCI instruction is encoded using a new encoding of the REGIMM opcode. This encoding was chosen because it causes a Reserved Instruction exception on all Release 1 implementations. As such, kernel software running on processors that don't implement Release 2 can emulate the function using the CACHE instruction.

Chapter 8

Coprocessor 0 Registers

The Coprocessor 0 (CP0) registers provide the interface between the ISA and the PRA. Each register is discussed below, with the registers presented in numerical order, first by register number, then by select field number.

8.1 Coprocessor 0 Register Summary

Table 8.1 lists the CP0 registers in numerical order. The individual registers are described later in this document. If the compliance level is qualified (e.g., "*Required* (TLB MMU)"), it applies only if the qualifying condition is true. The Sel column indicates the value to be used in the field of the same name in the MFC0 and MTC0 instructions.

Table 8.1 Coprocessor 0 Registers in Numerical Order

Register Number	Sel[1]	Register Name	Function	Reference	Compliance Level
0	0	Index	Index into the TLB array	Section 8.4 on page 76	Required (TLB MMU); Optional (Others)
0	1	MVPControl	Per-processor register containing global MIPS® MT configuration data	MIPS®MT ASE Specification	Required (MIPS MT ASE); Optional (Others)
0	2	MVPConf0	Per-processor multi-VPE dynamic configuration information	MIPS®MT ASE Specification	Required (MIPS MT ASE); Optional (Others)
0	3	MVPConf1	Per-processor multi-VPE dynamic configuration information	MIPS®MT ASE Specification	Optional
1	0	Random	Randomly generated index into the TLB array	Section 8.5 on page 77	Required (TLB MMU); Optional (Others)
1	1	VPEControl	Per-VPE register containing relatively volatile thread configuration data	MIPS®MT ASE Specification	Required (MIPS MT ASE); Optional (Others)
1	2	VPEConf0	Per-VPE multi-thread configuration information	MIPS®MT ASE Specification	Required (MIPS MT ASE); Optional (Others)
1	3	VPEConf1	Per-VPE multi-thread configuration information	MIPS®MT ASE Specification	Optional
1	4	YQMask	Per-VPE register defining which YIELD qualifier bits may be used without generating an exception	MIPS®MT ASE Specification	Required (MIPS MT ASE); Optional (Others)

Table 8.1 Coprocessor 0 Registers in Numerical Order

Register Number	Sel[1]	Register Name	Function	Reference	Compliance Level
1	5	VPESchedule	Per-VPE register to manage scheduling of a VPE within a processor	MIPS®MT ASE Specification	Optional
1	6	VPEScheFBack	Per-VPE register to provide scheduling feedback to software	MIPS®MT ASE Specification	Optional
1	7	VPEOpt	Per-VPE register to provide control over optional features, such as cache partitioning control	MIPS®MT ASE Specification	Optional
2	0	EntryLo0	Low-order portion of the TLB entry for even-numbered virtual pages	Section 8.6 on page 78	Required (TLB MMU); Optional (Others)
2	1	TCStatus	Per-TC status information, including copies of thread-specific bits of *Status* and *EntryHi* registers.	MIPS®MT ASE Specification	Required (MIPS MT ASE); Optional (Others)
2	2	TCBind	Per-TC information about TC ID and VPE binding	MIPS®MT ASE Specification	Required (MIPS MT ASE); Optional (Others)
2	3	TCRestart	Per-TC value of restart instruction address for the associated thread of execution	MIPS®MT ASE Specification	Required (MIPS MT ASE); Optional (Others)
2	4	TCHalt	Per-TC register controlling Halt state of TC	MIPS®MT ASE Specification	Required (MIPS MT ASE); Optional (Others)
2	5	TCContext	Per-TC read/write storage for operating system use	MIPS®MT ASE Specification	Required (MIPS MT ASE); Optional (Others)
2	6	TCSchedule	Per-TC register to manage scheduling of a TC	MIPS®MT ASE Specification	Optional
2	7	TCScheFBack	Per-TC register to provide scheduling feedback to software	MIPS®MT ASE Specification	Optional
3	0	EntryLo1	Low-order portion of the TLB entry for odd-numbered virtual pages	Section 8.6 on page 78	Required (TLB MMU); Optional (Others)
4	0	Context	Pointer to page table entry in memory	Section 8.7 on page 82	Required (TLB MMU); Optional (Others)
4	1	ContextConfig	Context and XContext register configuration	SmartMIPS ASE Specification	Required (SmartMIPS ASE Only)
4	2	UserLocal	User information that can be written by privileged software and read via RDHWR register 29. If the processor implements the MIPS® MT ASE, this is a per-TC register.	Section 8.8 on page 83	Recommended (Release 2)

Table 8.1 Coprocessor 0 Registers in Numerical Order

Register Number	Sel[1]	Register Name	Function	Reference	Compliance Level
5	0	PageMask	Control for variable page size in TLB entries	Section 8.9 on page 84	Required (TLB MMU); Optional (Others)
5	1	PageGrain	Control for small page support	Section 8.10 on page 86 and SmartMIPS ASE Specification	Required (SmartMIPS ASE); Optional (Release 2)
6	0	Wired	Controls the number of fixed (“wired”) TLB entries	Section 8.11 on page 88	Required (TLB MMU); Optional (Others)
6	1	SRSConf0	Per-VPE register indicating and optionally controlling shadow register set configuration	MIPS®MT ASE Specification	Required (MIPS MT ASE); Optional (Others)
6	2	SRSConf1	Per-VPE register indicating and optionally controlling shadow register set configuration	MIPS®MT ASE Specification	Optional
6	3	SRSConf2	Per-VPE register indicating and optionally controlling shadow register set configuration	MIPS®MT ASE Specification	Optional
6	4	SRSConf3	Per-VPE register indicating and optionally controlling shadow register set configuration	MIPS®MT ASE Specification	Optional
6	5	SRSConf4	Per-VPE register indicating and optionally controlling shadow register set configuration	MIPS®MT ASE Specification	Optional
7	0	HWREna	Enables access via the RDHWR instruction to selected hardware registers	Section 8.12 on page 90	Required (Release 2)
7	1-7		Reserved for future extensions		Reserved
8	0	BadVAddr	Reports the address for the most recent address-related exception	Section 8.13 on page 92	Required
9	0	Count	Processor cycle count	Section 8.14 on page 93	Required
9	6-7		Available for implementation dependent user	Section 8.15 on page 93	Implementation Dependent
10	0	EntryHi	High-order portion of the TLB entry	Section 8.16 on page 94	Required (TLB MMU); Optional (Others)
11	0	Compare	Timer interrupt control	Section 8.17 on page 96	Required
11	6-7		Available for implementation dependent user	Section 8.18 on page 96	Implementation Dependent

Table 8.1 Coprocessor 0 Registers in Numerical Order

Register Number	Sel[1]	Register Name	Function	Reference	Compliance Level
12	0	Status	Processor status and control	Section 8.19 on page 97	Required
12	1	IntCtl	Interrupt system status and control	Section 8.20 on page 104	Required (Release 2)
12	2	SRSCtl	Shadow register set status and control	Section 8.21 on page 106	Required (Release 2)
12	3	SRSMap	Shadow set IPL mapping	Section 8.22 on page 109	Required (Release 2 and shadow sets implemented)
13	0	Cause	Cause of last general exception	Section 8.23 on page 110	Required
14	0	EPC	Program counter at last exception	Section 8.24 on page 115	Required
15	0	PRId	Processor identification and revision	Section 8.25 on page 117	Required
15	1	EBase	Exception vector base register	Section 8.26 on page 119	Required (Release 2)
16	0	Config	Configuration register	Section 8.27 on page 121	Required
16	1	Config1	Configuration register 1	Section 8.28 on page 123	Required
16	2	Config2	Configuration register 2	Section 8.29 on page 127	Optional
16	3	Config3	Configuration register 3	Section 8.30 on page 130	Optional
16	6-7		Available for implementation dependent user	Section 8.31 on page 133	Implementation Dependent
17	0	LLAddr	Load linked address	Section 8.32 on page 134	Optional
18	0-n	WatchLo	Watchpoint address	Section 8.33 on page 135	Optional
19	0-n	WatchHi	Watchpoint control	Section 8.34 on page 137	Optional
20	0		XContext in 64-bit implementations		Reserved
21	all		Reserved for future extensions		Reserved
22	all		Available for implementation dependent use	Section 8.35 on page 139	Implementation Dependent
23	0	Debug	EJTAG Debug register	EJTAG Specification	Optional

Table 8.1 Coprocessor 0 Registers in Numerical Order

Register Number	Sel[1]	Register Name	Function	Reference	Compliance Level
23	1	TraceControl	PDtrace control register	PDtrace Specification	Optional
23	2	TraceControl2	PDtrace control register	PDtrace Specification	Optional
23	3	UserTraceData1	PDtrace control register	PDtrace Specification	Optional
23	4	TraceIBPC	PDtrace control register	PDtrace Specification	Optional
23	5	TraceDBPC	PDtrace control register	PDtrace Specification	Optional
23	6	Debug2	EJTAG Debug2 register	EJTAG Specification	Optional
24	0	DEPC	Program counter at last EJTAG debug exception	EJTAG Specification	Optional
24	2	TraceContol3	PDtrace control register	PDtrace Specification	Optional
24	3	UserTraceData2	PDtrace control register	PDtrace Specification	Optional
25	0-n	PerfCnt	Performance counter interface	Section 8.38 on page 142	Recommended
26	0	ErrCtl	Parity/ECC error control and status	Section 8.39 on page 146	Optional
27	0-3	CacheErr	Cache parity error control and status	Section 8.40 on page 147	Optional
28	even selects	TagLo	Low-order portion of cache tag interface	Section 8.41 on page 148	Required (Cache)
28	odd selects	DataLo	Low-order portion of cache data interface	Section 8.42 on page 149	Optional
29	even selects	TagHi	High-order portion of cache tag interface	Section 8.43 on page 150	Required (Cache)
29	odd selects	DataHi	High-order portion of cache data interface	Section 8.44 on page 151	Optional
30	0	ErrorEPC	Program counter at last error	Section 8.45 on page 152	Required
31	0	DESAVE	EJTAG debug exception save register	EJTAG Specification	Optional

1. Any select (Sel) value not explicitly noted as available for implementation-dependent use is reserved for future use by the Architecture.

8.2 Notation

For each register described below, field descriptions include the read/write properties of the field, and the reset state of the field. For the read/write properties of the field, the following notation is used:

Table 8.2 Read/Write Bit Field Notation

Read/Write Notation	Hardware Interpretation	Software Interpretation
R/W	A field in which all bits are readable and writable by software and, potentially, by hardware. Hardware updates of this field are visible by software read. Software updates of this field are visible by hardware read. If the Reset State of this field is “Undefined”, either software or hardware must initialize the value before the first read will return a predictable value. This should not be confused with the formal definition of **UNDEFINED** behavior.	
R	A field which is either static or is updated only by hardware. If the Reset State of this field is either “0”, “Preset”, or “Externally Set”, hardware initializes this field to zero or to the appropriate state, respectively, on powerup. The term “Preset” is used to suggest that the processor establishes the appropriate state, whereas the term “Externally Set” is used to suggest that the state is established via an external source (e.g., personality pins or initialization bit stream). These terms are suggestions only, and are not intended to act as a requirement on the implementation. If the Reset State of this field is “Undefined”, hardware updates this field only under those conditions specified in the description of the field.	A field to which the value written by software is ignored by hardware. Software may write any value to this field without affecting hardware behavior. Software reads of this field return the last value updated by hardware. If the Reset State of this field is “Undefined”, software reads of this field result in an **UNPREDICTABLE** value except after a hardware update done under the conditions specified in the description of the field.
0	A field which hardware does not update, and for which hardware can assume a zero value.	A field to which the value written by software must be zero. Software writes of non-zero values to this field may result in **UNDEFINED** behavior of the hardware. Software reads of this field return zero as long as all previous software writes are zero. If the Reset State of this field is “Undefined”, software must write this field with zero before it is guaranteed to read as zero.

8.3 Writing CPU Registers

With certain restrictions, software may assume that it can validly write the value read from a coprocessor 0 register back to that register without having unintended side effects. This rule means that software can read a register, modify one field, and write the value back to the register without having to consider the impact of writes to other fields. Processor designers should take this into consideration when using coprocessor 0 register fields that are reserved for implementations and make sure that the use of these bits is consistent with software assumptions.

The most significant exception to this rule is a situation in which the processor modifies the register between the software read and write, such as might occur if an exception or interrupt occurs between the read and write. Software must guarantee that such an event does not occur.

8.4 Index Register (CP0 Register 0, Select 0)

Compliance Level: *Required* for TLB-based MMUs; *Optional* otherwise.

The *Index* register is a 32-bit read/write register which contains the index used to access the TLB for TLBP, TLBR, and TLBWI instructions. The width of the index field is implementation-dependent as a function of the number of TLB entries that are implemented. The minimum value for TLB-based MMUs is Ceiling(Log2(TLBEntries)). For example, six bits are required for a TLB with 48 entries).

The operation of the processor is **UNDEFINED** if a value greater than or equal to the number of TLB entries is written to the *Index* register.

Figure 8-1 shows the format of the *Index* register; Table 8.3 describes the *Index* register fields.

Figure 8-1 Index Register Format

31	n	n-1 0
P	0	Index

Table 8.3 Index Register Field Descriptions

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read/ Write</th><th rowspan="2">Reset State</th><th rowspan="2">Compliance</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>P</td><td>31</td><td>Probe Failure. Hardware writes this bit during execution of the TLBP instruction to indicate whether a TLB match occurred:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>A match occurred, and the <i>Index</i> field contains the index of the matching entry</td></tr>
<tr><td>1</td><td>No match occurred and the Index field is <b>UNPREDICTABLE</b></td></tr>
</table></td><td>R</td><td>Undefined</td><td>Required</td></tr>
<tr><td>0</td><td>30..n</td><td>Must be written as zero; returns zero on read.</td><td>0</td><td>0</td><td>Reserved</td></tr>
<tr><td>Index</td><td>n-1..0</td><td>TLB index. Software writes this field to provide the index to the TLB entry referenced by the TLBR and TLBWI instructions.
Hardware writes this field with the index of the matching TLB entry during execution of the TLBP instruction. If the TLBP fails to find a match, the contents of this field are <b>UNPREDICTABLE</b>.</td><td>R/W</td><td>Undefined</td><td>Required</td></tr>
</table>

8.5 Random Register (CP0 Register 1, Select 0)

Compliance Level: *Required* for TLB-based MMUs; *Optional* otherwise.

The *Random* register is a read-only register whose value is used to index the TLB during a TLBWR instruction. The width of the Random field is calculated in the same manner as that described for the *Index* register above.

The value of the register varies between an upper and lower bound as follow:

- A lower bound is set by the number of TLB entries reserved for exclusive use by the operating system (the contents of the *Wired* register). The entry indexed by the *Wired* register is the first entry available to be written by a TLB Write Random operation.
- An upper bound is set by the total number of TLB entries minus 1.

Within the required constraints of the upper and lower bounds, the manner in which the processor selects values for the *Random* register is implementation-dependent.

The processor initializes the *Random* register to the upper bound on a Reset Exception, and when the *Wired* register is written.

Figure 8-2 shows the format of the *Random* register; Table 8.4 describes the *Random* register fields.

Figure 8-2 Random Register Format

31 n	n-1 0
0	Random

Table 8.4 Random Register Field Descriptions

Fields		Description	Read/ Write	Reset State	Compliance
Name	Bits				
0	31..n	Must be written as zero; returns zero on read.	0	0	Reserved
Random	n-1..0	TLB Random Index	R	TLB Entries - 1	Required

8.6 EntryLo0, EntryLo1 (CP0 Registers 2 and 3, Select 0)

Compliance Level: *EntryLo0* is *Required* for a TLB-based MMU; *Optional* otherwise.

Compliance Level: *EntryLo1* is *Required* for a TLB-based MMU; *Optional* otherwise.

The pair of *EntryLo* registers act as the interface between the TLB and the TLBP, TLBR, TLBWI, and TLBWR instructions. *EntryLo0* holds the entries for even pages and *EntryLo1* holds the entries for odd pages.

Software may determine the value of *PABITS* by writing all ones to the *EntryLo0* or *EntryLo1* registers and reading the value back. Bits read as “1” from the PFN field allow software to determine the boundary between the PFNand Fill fields to calculate the value of *PABITS*.

The contents of the *EntryLo0* and *EntryLo1* registers are not defined after an address error exception and some fields may be modified by hardware during the address error exception sequence. Software writes of the *EntryHi* register (via MTC0) do not cause the implicit update of address-related fields in the *BadVAddr* or *Context* registers.

For Release 1 of the Architecture, Figure 8-3 shows the format of the *EntryLo0* and *EntryLo1* registers; Table 8.5 describes the *EntryLo0* and *EntryLo1* register fields. For Release 2 of the Architecture, Figure 8-4 shows the format of the *EntryLo0* and *EntryLo1* registers; Table 8.6 describes the *EntryLo0* and *EntryLo1* register fields.

Figure 8-3 EntryLo0, EntryLo1 Register Format in Release 1 of the Architecture

31 30	29 6	5 3	2	1	0
Fill	PFN	C	D	V	G

Table 8.5 EntryLo0, EntryLo1 Register Field Descriptions in Release 1 of the Architecture

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
Fill	31..30	These bits are ignored on write and return zero on read. The boundaries of this field change as a function of the value of *PABITS*. See Table 8.7 for more information.	R	0	Required
PFN	29..6	Page Frame Number. Corresponds to bits *PABITS*-1..12 of the physical address, where *PABITS* is the width of the physical address in bits. The boundaries of this field change as a function of the value of *PABITS*. See Table 8.7 for more information.	R/W	Undefined	Required
C	5..3	Cacheability and Coherency Attribute of the page. See Table 8.8 below.	R/W	Undefined	Required

Table 8.5 EntryLo0, EntryLo1 Register Field Descriptions in Release 1 of the Architecture

Fields					
Name	**Bits**	**Description**	**Read / Write**	**Reset State**	**Compliance**
D	2	“Dirty” bit, indicating that the page is writable. If this bit is a one, stores to the page are permitted. If this bit is a zero, stores to the page cause a TLB Modified exception. Kernel software may use this bit to implement paging algorithms that require knowing which pages have been written. If this bit is always zero when a page is initially mapped, the TLB Modified exception that results on any store to the page can be used to update kernel data structures that indicate that the page was actually written.	R/W	Undefined	Required
V	1	Valid bit, indicating that the TLB entry, and thus the virtual page mapping are valid. If this bit is a one, accesses to the page are permitted. If this bit is a zero, accesses to the page cause a TLB Invalid exception.	R/W	Undefined	Required
G	0	Global bit. On a TLB write, the logical AND of the G bits from both *EntryLo0* and *EntryLo1* becomes the G bit in the TLB entry. If the TLB entry G bit is a one, ASID comparisons are ignored during TLB matches. On a read from a TLB entry, the G bits of both *EntryLo0* and *EntryLo1* reflect the state of the TLB G bit.	R/W	Undefined	Required (TLB MMU)

Figure 8-4 EntryLo0, EntryLo1 Register Format in Release 2 of the Architecture

31 30	29 6	5 3	2	1	0
Fill	PFN	C	D	V	G

Table 8.6 EntryLo0, EntryLo1 Register Field Descriptions in Release 2 of the Architecture

Fields					
Name	**Bits**	**Description**	**Read / Write**	**Reset State**	**Compliance**
Fill	31..30	These bits are ignored on write and return zero on read. The boundaries of this field change as a function of the value of *PABITS*. See Table 8.7 for more information.	R	0	Required

Table 8.6 EntryLo0, EntryLo1 Register Field Descriptions in Release 2 of the Architecture

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
PFN	29..6	Page Frame Number. This field contains the physical page number corresponding to the virtual page. If the processor is enabled to support 1KB pages (Config3$_{SP}$ = 1 and PageGrain$_{ESP}$ = 1), the PFN field corresponds to bits 33..10 of the physical address (the field is shifted left by 2 bits relative to the Release 1 definition to make room for PA$_{11..10}$). If the processor is not enabled to support 1KB pages (Config3$_{SP}$ = 0 or PageGrain$_{ESP}$ = 0), the PFN field corresponds to bits 35..12 of the physical address. The boundaries of this field change as a function of the value of *PABITS*. See Table 8.7 for more information.	R/W	Undefined	Required
C	5..3	The definition of this field is unchanged from Release 1. See Table 8.5 above and Table 8.8 below.	R/W	Undefined	Required
D	2	The definition of this field is unchanged from Release 1. See Table 8.5 above.	R/W	Undefined	Required
V	1	The definition of this field is unchanged from Release 1. See Table 8.5 above.	R/W	Undefined	Required
G	0	The definition of this field is unchanged from Release 1. See Table 8.5 above.	R/W	Undefined	Required (TLB MMU)

Table 8.7 shows the movement of the Fill and PFN fields as a function of 1KB page support enabled, and the value of *PABITS*. Note that in implementations of Release 1 of the Architecture, there is no support for 1KB pages, so only the first row of the table applies to Release 1.

Table 8.7 EntryLo Field Widths as a Function of PABITS

1KB Page Support Enabled?	*PABITS* Value	Corresponding EntryLo Field Bit Ranges		Release 2 Required?
		Fill Field	PFN Field	
No	36 ≥ *PABITS* > 12	31..(30-(36-*PABITS*)) Example: 31..30 if *PABITS* = 36 31..7 if *PABITS* = 13	(29-(36-*PABITS*))..6 Example: 29..6 if *PABITS* = 36 6..6 if *PABITS* = 13 EntryLo$_{29..6}$ = PA$_{35..12}$	No
Yes	34 ≥ *PABITS* > 10	31..(30-(34-*PABITS*)) Example: 31..30 if *PABITS* = 34 31..7 if *PABITS* = 11	(29-(34-*PABITS*))..6 Example: 29..6 if *PABITS* = 34 6..6 if *PABITS* = 11 EntryLo$_{29..6}$ = PA$_{33..10}$	Yes

Programming Note:

In implementations of Release 2 of the Architecture, the PFN field of both the *EntryLo0* and *EntryLo1* registers must be written with zero and the TLB must be flushed before each instance in which the value of the *PageGrain* register is changed. This operation must be carried out while running in an unmapped address space. The operation of the processor is **UNDEFINED** if this sequence is not done.

Table 8.8 lists the encoding of the C field of the *EntryLo0* and *EntryLo1* registers and the K0 field of the *Config* register. An implementation may choose to implement a subset of the cache coherency attributes shown, but must implement at least encodings 2 and 3 such that software can always depend on these encodings working appropriately. In other cases, the operation of the processor is **UNDEFINED** if software specifies an unimplemented encoding.

Table 8.8 lists the required and optional encodings for the cacheability and coherency attributes.

Table 8.8 Cacheability and Coherency Attributes

C(5:3) Value	Cacheability and Coherency Attributes With Historical Usage	Compliance
0	Available for implementation dependent use	Optional
1	Available for implementation dependent use	Optional
2	Uncached	Required
3	Cacheable	Required
4	Available for implementation dependent use	Optional
5	Available for implementation dependent use	Optional
6	Available for implementation dependent use	Optional
7	Available for implementation dependent use	Optional

8.7 Context Register (CP0 Register 4, Select 0)

Compliance Level: *Required* for TLB-based MMUs; *Optional* otherwise.

The *Context* register is a read/write register containing a pointer to an entry in the page table entry (PTE) array. This array is an operating system data structure that stores virtual-to-physical translations. During a TLB miss, the operating system loads the TLB with the missing translation from the PTE array. The *Context* register duplicates some of the information provided in the *BadVAddr* register, but is organized in such a way that the operating system can directly reference a 16-byte structure in memory that describes the mapping.

A TLB exception (TLB Refill, TLB Invalid, or TLB Modified) causes bits $VA_{31..13}$ of the virtual address to be written into the *BadVPN2* field of the *Context* register. The *PTEBase* field is written and used by the operating system.

The *BadVPN2* field of the *Context* register is not defined after an address error exception and this field may be modified by hardware during the address error exception sequence.

Figure 8-5 shows the format of the *Context* Register; Table 8.9 describes the *Context* register fields.

Figure 8-5 Context Register Format

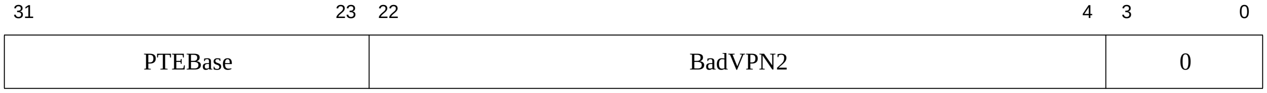


Table 8.9 Context Register Field Descriptions

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
PTEBase	31..23	This field is for use by the operating system and 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.	R/W	Undefined	Required
BadVPN2	22..4	This field is written by hardware on a TLB exception. It contains bits $VA_{31..13}$ of the virtual address that caused the exception.	R	Undefined	Required
0	3..0	Must be written as zero; returns zero on read.	0	0	Reserved

8.8 UserLocal Register (CP0 Register 4, Select 2)

Compliance Level: *Recommended*.

The *UserLocal* register is a read-write register that is not interpreted by the hardware and conditionally readable via the RDHWR instruction.

If the MIPS® MT ASE is implemented, the *UserLocal* register is instantiated per TC.

Figure 8-6 shows the format of the *UserLocal* register; Table 8.10 describes the *UserLocal* register fields.

Figure 8-6 UserLocal Register Format

31	0
UserInformation	

Table 8.10 UserLocal Register Field Descriptions

Fields					
Name	**Bits**	**Description**	**Read/ Write**	**Reset State**	**Compliance**
UserInformation	31..0	This field contains software information that is not interpreted by the hardware.	R/W	Undefined	Required

Programming Notes

Privileged software may write this register with arbitrary information and make it accessable to unprivileged software via register 29 (ULR) of the RDHWR instruction. To do so, bit 29 of the *HWREna* register must be set to a 1 to enable unprivileged access to the register. In some operating environments, the *UserLocal* register contains a pointer to a thread-specific storage block that is obtained via the RDHWR register.

8.9 PageMask Register (CP0 Register 5, Select 0)

Compliance Level: *Required* for TLB-based MMUs; *Optional* otherwise.

The *PageMask* register is a read/write register used for reading from and writing to the TLB. It holds a comparison mask that sets the variable page size for each TLB entry, as shown in Table 8.12. Figure 8-7 shows the format of the *PageMask* register; Table 8.11 describes the *PageMask* register fields.

Figure 8-7 PageMask Register Format

31 29	28 13	12 11	0
0	Mask	MaskX	0

Table 8.11 PageMask Register Field Descriptions

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
Mask	28..13	The Mask field is a bit mask in which a “1” bit indicates that the corresponding bit of the virtual address should not participate in the TLB match.	R/W	Undefined	Required
MaskX	12..11	In Release 2 of the Architecture, the MaskX field is an extension to the Mask field to support 1KB pages with definition and action analogous to that of the Mask field, defined above. If 1KB pages are enabled ($\text{Config3}_{\text{SP}}$ = 1 and $\text{PageGrain}_{\text{ESP}}$ = 1), these bits are writable and readable, and their values are copied to and from the TLB entry on a TLB write or read, respectively. If 1KB pages are not enabled ($\text{Config3}_{\text{SP}}$ = 0 or $\text{PageGrain}_{\text{ESP}}$ = 0), these bits are not writable, return zero on read, and the effect on the TLB entry on a write is as if they were written with the value 0b11. In Release 1 of the Architecture, these bits must be written as zero, return zero on read, and have no effect on the virtual address translation.	R/W	0 (See Description)	Required (Release 2)
0	31..29, 10..0	Ignored on write; returns zero on read.	R	0	Required

Table 8.12 Values for the Mask and MaskX[1] Fields of the PageMask Register

Page Size	Bit																	
	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12[1]	11[1]
1 KByte	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4 KBytes	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1
16 KBytes	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1
64 KBytes	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1
256 KBytes	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
1 MByte	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1
4 MByte	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1
16 MByte	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1
64 MByte	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
256 MByte	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

[1] PageMask$_{12..11}$ = PageMask$_{\text{MaskX}}$ exists only on implementations of Release 2 of the architecture and are treated as if they had the value 0b11 if 1K pages are not enabled (Config3$_{\text{SP}}$ = 0 or PageGrain$_{\text{ESP}}$ = 0).

It is implementation dependent how many of the encodings described in Table 8.12 are implemented. All processors must implement the 4KB page size. If a particular page size encoding is not implemented by a processor, a read of the *PageMask* register must return zeros in all bits that correspond to encodings that are not implemented, thereby potentially returning a value different than that written by software.

Software may determine which page sizes are supported by writing all ones to the *PageMask* register, then reading the value back. If a pair of bits reads back as ones, the processor implements that page size. The operation of the processor is **UNDEFINED** if software loads the *Mask* field with a value other than one of those listed in Table 8.12, even if the hardware returns a different value on read. Hardware may depend on this requirement in implementing hardware structures

Programming Note:

In implementations of Release 2 of the Architecture, the *MaskX* field of the *PageMask* register must be written with 0b11 and the TLB must be flushed before each instance in which the value of the *PageGrain* register is changed. This operation must be carried out while running in an unmapped address space. The operation of the processor is **UNDEFINED** if this sequence is not done.

8.10 PageGrain Register (CP0 Register 5, Select 1)

Compliance Level: *Required* for implementations of Release 2 of the Architecture that include TLB-based MMUs and support 1KB pages; *Optional* otherwise.

The *PageGrain* register is a read/write register used for enabling 1KB page support. The *PageGrain* register is present in both the SmartMIPS™ ASE, and in Release 2 of the Architecture, although there are no bits in common between the two uses of this register. As such, the description below only describes the fields relevant to Release 2 of the Architecture. In implementations of both Release 2 of the Architecture and the SmartMIPS™ ASE, the ASE definitions take precedence and none of the Release 2 fields described below are present. Figure 8-8 shows the format of the *PageGrain* register; Table 8.13 describes the *PageGrain* register fields.

Figure 8-8 PageGrain Register Format

31 30	29	28	27 13	12 8	7 0
ASE	ELPA	ESP	0	ASE	0

Table 8.13 PageGrain Register Field Descriptions

Fields					
Name	Bits	Description	Read / Write	Reset State	Compliance
ASE	31..30, 12..8	These fields are control features of the SmartMIPS™ ASE and are not used in implementations of Release 2 of the Architecture unless such an implementation also implements the SmartMIPS™ ASE.	0	0	Required
ELPA	29	Used to enable support for large physical addresses in MIPS64 processors; not used by MIPS32 processors. This bit is ignored on write and returns zero on read.	R	0	Required

Table 8.13 PageGrain Register Field Descriptions

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read / Write</th><th rowspan="2">Reset State</th><th rowspan="2">Compliance</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>ESP</td><td>28</td><td>Enables support for 1KB pages.
<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>1KB page support is not enabled</td></tr><tr><td>1</td><td>1KB page support is enabled</td></tr></table>If this bit is a 1, the following changes occur to coprocessor 0 registers:
• The PFN field of the <i>EntryLo0</i> and <i>EntryLo1</i> registers holds the physical address down to bit 10 (the field is shifted left by 2 bits from the Release 1 definition)
• The MaskX field of the <i>PageMask</i> register is writable and is concatenated to the right of the Mask field to form the “don’t care” mask for the TLB entry.
• The VPN2X field of the <i>EntryHi</i> register is writable and bits 12..11 of the virtual address.
• The virtual address translation algorithm is modified to reflect the smaller page size.
If $\text{Config3}_{\text{SP}}$ = 0, 1KB pages are not implemented, and this bit is ignored on write and returns zero on read.</td><td>R/W</td><td>0</td><td>Required</td></tr>
<tr><td>0</td><td>27..13, 7..0</td><td>Must be written as zero; returns zero on read.</td><td>0</td><td>0</td><td>Reserved</td></tr>
</table>

Programming Note:

In implementations of Release 2 of the Architecture, the following fields must be written with the specified values, and the TLB must be flushed before each instance in which the value of the *PageGrain* register is changed. This operation must be carried out while running in an unmapped address space. The operation of the processor is **UNDEFINED** if this sequence is not done.

Field	Required Value
$\text{EntryLo0}_{\text{PFN}}$, $\text{EntryLo1}_{\text{PFN}}$	0
$\text{EntryLo0}_{\text{PFNX}}$, $\text{EntryLo1}_{\text{PFNX}}$	0
$\text{PageMask}_{\text{MaskX}}$	0b11
$\text{EntryHi}_{\text{VPN2X}}$	0

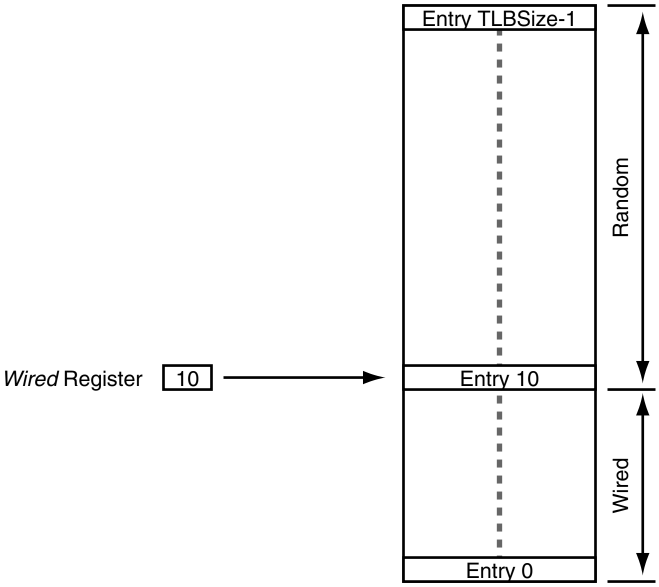
Note also that if *PageGrain* is changed, a hazard may be created between the instruction that writes *PageGrain* and a subsequent CACHE instruction. This hazard must be cleared using the EHB instruction.

8.11 Wired Register (CP0 Register 6, Select 0)

Compliance Level: *Required* for TLB-based MMUs; *Optional* otherwise.

The *Wired* register is a read/write register that specifies the boundary between the wired and random entries in the TLB as shown in Figure 8-9.

Figure 8-9 Wired And Random Entries In The TLB



The width of the *Wired* field is calculated in the same manner as that described for the *Index* register. *Wired* entries are fixed, non-replaceable entries which are not overwritten by a TLBWR instruction.*Wired* entries can be overwritten by a TLBWI instruction.

The *Wired* register is set to zero by a Reset Exception. Writing the *Wired* register causes the *Random* register to reset to its upper bound.

The operation of the processor is **UNDEFINED** if a value greater than or equal to the number of TLB entries is written to the *Wired* register.

Figure 8-9 shows the format of the *Wired* register; Table 8.14 describes the *Wired* register fields.

Figure 8-10 Wired Register Format

31 n	n-1 0
0	Wired

Table 8.14 Wired Register Field Descriptions

Fields					
Name	Bits	Description	Read/ Write	Reset State	Compliance
0	31..n	Must be written as zero; returns zero on read.	0	0	Reserved
Wired	n-1..0	TLB wired boundary	R/W	0	Required

8.12 HWREna Register (CP0 Register 7, Select 0)

Compliance Level: *Required* (Release 2).

The *HWREna* register contains a bit mask that determines which hardware registers are accessible via the RDHWR instruction when that instruction is executed in a mode in which coprocessor 0 is not enabled.

Figure 8-11 shows the format of the *HWREna* Register; Table 8.15 describes the *HWREna* register fields.

Figure 8-11 HWREna Register Format

31 30	29 4 3 0
Impl	Mask

Table 8.15 HWREna Register Field Descriptions

Fields					
Name	Bits	Description	Read / Write	Reset State	Compliance
31..30	Impl	These bits enable access to the implementation-dependent hardware registers 31 and 30. If a register is not implemented, the corresponding bit returns a zero and is ignored on write. If a register is implemented, access to that register is enabled if the corresponding bit in this field is a 1 and disabled if the corresponding bit is a 0.	R/W	0	Optional - Reserved for Implementations
Mask	29..0	Each bit in this field enables access by the RDHWR instruction to a particular hardware register (which may not be an actual register). If RDHWR register ‘n’ is not implemented, bit ‘n’ of this field returns a zero and is ignored on a write. If RDHWR register ‘n’ is implemented, access to the register is enabled if bit ‘n’ in this field is a 1 and disabled if bit ‘n’ of this field is a 0. See the RDHWR instruction for a list of valid hardware registers. Table 8.16 lists the RDHWR registers, and register number ‘n’ corresponds to bit ‘n’ in this field.	R/W	0	Required

Table 8.16 RDHWR Register Numbers

<table>
<tr><th>Register Number</th><th>Mnemonic</th><th>Description</th><th>Compliance</th></tr>
<tr><td>0</td><td>CPUNum</td><td>Number of the CPU on which the program is currently running. This register provides read access to the coprocessor 0 $EBase_{CPUNum}$ field.</td><td>Required</td></tr>
<tr><td>1</td><td>SYNCI_Step</td><td>Address step size to be used with the SYNCI instruction. See that instruction’s description for the use of this value. In the typical implementation, this value should be zero if there are no caches in the system which must be synchronize (either because there are no caches, or because the instruction cache tracks writes to the data cache). In other cases, the return value should be the smallest line size of the caches that must be synchronize.</td><td>Required</td></tr>
<tr><td>2</td><td>CC</td><td>High-resolution cycle counter. This register provides read access to the coprocessor 0 <i>Count</i> Register.</td><td>Required</td></tr>
<tr><td>3</td><td>CCRes</td><td>Resolution of the CC register. This value denotes the number of cycles between update of the register. For example:
<table>
<tr><th>CCRes Value</th><th>Meaning</th></tr>
<tr><td>1</td><td>CC register increments every CPU cycle</td></tr>
<tr><td>2</td><td>CC register increments every second CPU cycle</td></tr>
<tr><td>3</td><td>CC register increments every third CPU cycle</td></tr>
<tr><td colspan="2">etc.</td></tr>
</table>
</td><td>Required</td></tr>
<tr><td>4-28</td><td></td><td>These registers numbers are reserved for future architecture use. Access results in a Reserved Instruction Exception.</td><td>Reserved</td></tr>
<tr><td>29</td><td>ULR</td><td>User Local Register. This register provides read access to the coprocessor 0 <i>UserLocal</i> register, if it is implemented. In some operating environments, the <i>UserLocal</i> register is a pointer to a thread-specific storage block.</td><td>Required if the <i>UserLocal</i> register is implemented</td></tr>
<tr><td>30-31</td><td></td><td>These register numbers are reserved for implementation-dependent use. If they are not implemented, access results in a Reserved Instruction Exception.</td><td>Optional</td></tr>
</table>

Using the *HWREna* register, privileged software may select which of the hardware registers are accessible via the RDHWR instruction. In doing so, a register may be virtualized at the cost of handling a Reserved Instruction Exception, interpreting the instruction, and returning the virtualized value. For example, if it is not desirable to provide direct access to the *Count* register, access to that register may be individually disabled and the return value can be virtualized by the operating system.

Software may determine which registers are implemented by writing all ones to the *HWREna* register, then reading the value back. If a bit reads back as a one, the processor implements that hardware register.

8.13 BadVAddr Register (CP0 Register 8, Select 0)

Compliance Level: *Required*.

The *BadVAddr* register is a read-only register that captures the most recent virtual address that caused one of the following exceptions:

- Address error (AdEL or AdES)
- TLB Refill
- TLB Invalid (TLBL, TLBS)
- TLB Modified

The *BadVAddr* register does not capture address information for cache or bus errors, or for Watch exceptions, since none is an addressing error.

Figure 8-12 shows the format of the *BadVAddr* register; Table 8.17 describes the *BadVAddr* register fields.

Figure 8-12 BadVAddr Register Format

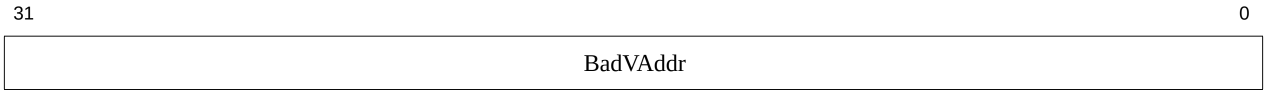


Table 8.17 BadVAddr Register Field Descriptions

Fields		Description	Read/Write	Reset State	Compliance
Name	Bits				
BadVAddr	31..0	Bad virtual address	R	Undefined	Required

8.14 Count Register (CP0 Register 9, Select 0)

Compliance Level: *Required*.

The *Count* register acts as a timer, incrementing at a constant rate, whether or not an instruction is executed, retired, or any forward progress is made through the pipeline. The rate at which the counter increments is implementation dependent, and is a function of the pipeline clock of the processor, not the issue width of the processor.

The *Count* register can be written for functional or diagnostic purposes, including at reset or to synchronize processors.

The Count register can also be read via RDHWR register 2.

Figure 8-13 shows the format of the *Count* register; Table 8.18 describes the *Count* register fields.

Figure 8-13 Count Register Format

31	0
Count	

Table 8.18 Count Register Field Descriptions

Fields		Description	Read/Write	Reset State	Compliance
Name	Bits				
Count	31..0	Interval counter	R/W	Undefined	Required

8.15 Reserved for Implementations (CP0 Register 9, Selects 6 and 7)

Compliance Level: *Implementation Dependent*.

CP0 register 9, Selects 6 and 7 are reserved for implementation dependent use and are not defined by the architecture.

8.16 EntryHi Register (CP0 Register 10, Select 0)

Compliance Level: *Required* for TLB-based MMU; *Optional* otherwise.

The *EntryHi* register contains the virtual address match information used for TLB read, write, and access operations.

A TLB exception (TLB Refill, TLB Invalid, or TLB Modified) causes bits VA$_{31..13}$ of the virtual address to be written into the *VPN2* field of the *EntryHi* register. An implementation of Release 2 of the Architecture which supports 1KB pages also writes VA$_{12..11}$ into the *VPN2X* field of the *EntryHi* register. A TLBR instruction writes the *EntryHi* register with the corresponding fields from the selected TLB entry. The *ASID* field is written by software with the current address space identifier value and is used during the TLB comparison process to determine TLB match.

Because the *ASID* field is overwritten by a TLBR instruction, software must save and restore the value of *ASID* around use of the TLBR. This is especially important in TLB Invalid and TLB Modified exceptions, and in other memory management software.

The *VPNX2* and *VPN2* fields of the *EntryHi* register are not defined after an address error exception and these fields may be modified by hardware during the address error exception sequence.Software writes of the *EntryHi* register (via MTC0) do not cause the implicit write of address-related fields in the *BadVAddr* or *Context* registers.

Figure 8-14 shows the format of the *EntryHi* register; Table 8.19 describes the *EntryHi* register fields.

Figure 8-14 EntryHi Register Format

31 13	12 11	10 8	7 0
VPN2	VPN2X	0	ASID

Table 8.19 EntryHi Register Field Descriptions

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
VPN2	31..13	VA$_{31..13}$ of the virtual address (virtual page number / 2). This field is written by hardware on a TLB exception or on a TLB read, and is written by software before a TLB write.	R/W	Undefined	Required
VPN2X	12..11	In Release 2 of the Architecture, the VPN2X field is an extension to the VPN2 field to support 1KB pages. These bits are not writable by either hardware or software unless Config3$_{SP}$ = 1 and PageGrain$_{ESP}$ = 1. If enabled for write, this field contains VA$_{12..11}$ of the virtual address and is written by hardware on a TLB exception or on a TLB read, and is by software before a TLB write. If writes are not enabled, and in implementations of Release 1 of the Architecture, this field must be written with zero and returns zeros on read.	R/W	0	Required (Release 2 and 1KB Page Support)
0	10..8	Must be written as zero; returns zero on read.	0	0	Reserved

Table 8.19 EntryHi Register Field Descriptions

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
ASID	7..0	Address space identifier. This field is written by hardware on a TLB read and by software to establish the current ASID value for TLB write and against which TLB references match each entry's TLB ASID field.	R/W	Undefined	Required (TLB MMU)

Programming Note:

In implementations of Release 2 of the Architecture, the VPN2X field of the *EntryHi* register must be written with zero and the TLB must be flushed before each instance in which the value of the *PageGrain* register is changed. This operation must be carried out while running in an unmapped address space. The operation of the processor is **UNDEFINED** if this sequence is not done.

8.17 Compare Register (CP0 Register 11, Select 0)

Compliance Level: *Required*.

The *Compare* register acts in conjunction with the *Count* register to implement a timer and timer interrupt function. The *Compare* register maintains a stable value and does not change on its own.

When the value of the *Count* register equals the value of the *Compare* register, an interrupt request is made. In Release 1 of the architecture, this request is combined in an implementation-dependent way with hardware interrupt 5 to set interrupt bit IP(7) in the *Cause* register. In Release 2 of the Architecture, the presence of the interrupt is visible to software via the Cause_{TI} bit and is combined in an implementation-dependent way with a hardware or software interrupt. For Vectored Interrupt Mode, the interrupt is at the level specified by the $\text{IntCtl}_{\text{IPTI}}$ field.

For diagnostic purposes, the *Compare* register is a read/write register. In normal use however, the *Compare* register is write-only. Writing a value to the *Compare* register, as a side effect, clears the timer interrupt. Figure 8-15 shows the format of the *Compare* register; Table 8.20 describes the *Compare* register fields.

Figure 8-15 Compare Register Format

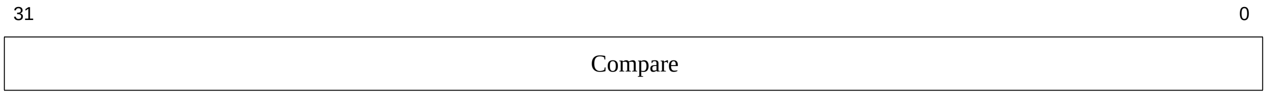


Table 8.20 Compare Register Field Descriptions

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
Compare	31..0	Interval count compare value	R/W	Undefined	Required

Programming Note:

In Release 2 of the Architecture, the EHB instruction can be used to make interrupt state changes visible when the *Compare* register is written. See 5.1.2.1 “Software Hazards and the Interrupt System” on page 42.

8.18 Reserved for Implementations (CP0 Register 11, Selects 6 and 7)

Compliance Level: *Implementation Dependent*.

CP0 register 11, Selects 6 and 7 are reserved for implementation dependent use and are not defined by the architecture.

8.19 Status Register (CP Register 12, Select 0)

Compliance Level: *Required*.

The *Status* register is a read/write register that contains the operating mode, interrupt enabling, and the diagnostic states of the processor. Fields of this register combine to create operating modes for the processor. Refer to "MIPS32 Operating Modes" on page 17 for a discussion of operating modes, and "Interrupts" on page 31 for a discussion of interrupt modes.

Figure 8-16 shows the format of the *Status* register; Table 8.21 describes the *Status* register fields.

Figure 8-16 Status Register Format

31 28	27	26	25	24	23	22	21	20	19	18	17 16	15 10	9 8	7	6	5	4	3	2	1	0
CU3..CU0	RP	FR	RE	MX	PX	BEV	TS	SR	NMI	0	Impl	IM7..IM2	IM1..IM0	KX	SX	UX	UM	R0	ERL	EXL	IE
												IPL					KSU				

Table 8.21 Status Register Field Descriptions

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read / Write</th><th rowspan="2">Reset State</th><th rowspan="2">Compliance</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>CU (CU3.. CU0)</td><td>31..28</td><td>Controls access to coprocessors 3, 2, 1, and 0, respectively:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Access not allowed</td></tr>
<tr><td>1</td><td>Access allowed</td></tr>
</table>
Coprocessor 0 is always usable when the processor is running in Kernel Mode or Debug Mode, independent of the state of the CU_0 bit.
In Release 2 of the Architecture, and for 64-bit implementations of Release 1 of the Architecture, execution of all floating point instructions, including those encoded with the COP1X opcode, is controlled by the CU1 enable. CU3 is no longer used and is reserved for future use by the Architecture.
If there is no provision for connecting a coprocessor, the corresponding CU bit must be ignored on write and read as zero.</td><td>R/W</td><td>Undefined</td><td>Required for all implemented coprocessors</td></tr>
<tr><td>RP</td><td>27</td><td>Enables reduced power mode on some implementations. The specific operation of this bit is implementation dependent.
If this bit is not implemented, it must be ignored on write and read as zero. If this bit is implemented, the reset state must be zero so that the processor starts at full performance.</td><td>R/W</td><td>0</td><td>Optional</td></tr>
</table>

Table 8.21 Status Register Field Descriptions (Continued)

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read / Write</th><th rowspan="2">Reset State</th><th rowspan="2">Compliance</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>FR</td><td>26</td><td>In Release 1 of the Architecture, only MIPS64 processors could implement a 64-bit floating point unit. In Release 2 of the Architecture, both MIPS32 and MIPS64 processors can implement a 64-bit floating point unit. This bit is used to control the floating point register mode for 64-bit floating point units:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Floating point registers can contain any 32-bit datatype. 64-bit datatypes are stored in even-odd pairs of registers.</td></tr>
<tr><td>1</td><td>Floating point registers can contain any datatype</td></tr>
</table>
This bit must be ignored on write and read as zero under the following conditions:
• No floating point unit is implemented
• In a MIPS32 implementation of Release 1 of the Architecture
• In an implementation of Release 2 of the Architecture in which a 64-bit floating point unit is not implemented
Certain combinations of the FR bit and other state or operations can cause <b>UNPREDICTABLE</b> behavior. See “64-bit FPR Enable” on page 18 for a discussion of these combinations.</td><td>R/W</td><td>Undefined</td><td>Required</td></tr>
<tr><td>RE</td><td>25</td><td>Used to enable reverse-endian memory references while the processor is running in user mode:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>User mode uses configured endianness</td></tr>
<tr><td>1</td><td>User mode uses reversed endianness</td></tr>
</table>
Neither Debug Mode nor Kernel Mode nor Supervisor Mode references are affected by the state of this bit.
If this bit is not implemented, it must be ignored on write and read as zero.</td><td>R/W</td><td>Undefined</td><td>Optional</td></tr>
<tr><td>MX</td><td>24</td><td>Enables access to MDMX™ and MIPS® DSP resources on processors implementing one of these ASEs. If neither the MDMX nor the MIPS DSP ASE is implemented, this bit must be ignored on write and read as zero.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Access not allowed</td></tr>
<tr><td>1</td><td>Access allowed</td></tr>
</table></td><td>R if the processor implements neither the MDMX nor the MIPS DSP ASEs; otherwise R/W</td><td>0 if the processor implements neither the MDMX nor the MIPS DSP ASEs; otherwise Undefined</td><td>Optional</td></tr>
</table>

Table 8.21 Status Register Field Descriptions (Continued)

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read / Write</th><th rowspan="2">Reset State</th><th rowspan="2">Compliance</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>PX</td><td>23</td><td>Enables access to 64-bit operations on MIPS64 processors. Not used by MIPS32 processors. This bit must be ignored on write and read as zero.</td><td>R</td><td>0</td><td>Required</td></tr>
<tr><td>BEV</td><td>22</td><td>Controls the location of exception vectors:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Normal</td></tr>
<tr><td>1</td><td>Bootstrap</td></tr>
</table>
See “Exception Vector Locations” on page 45 for details.</td><td>R/W</td><td>1</td><td>Required</td></tr>
<tr><td>TS[1]</td><td>21</td><td>Indicates that the TLB has detected a match on multiple entries. It is implementation dependent whether this detection occurs at all, on a write to the TLB, or an access to the TLB. In Release 2 of the Architecture, multiple TLB matches may only be reported on a TLB write. When such a detection occurs, the processor initiates a machine check exception and sets this bit. It is implementation dependent whether this condition can be corrected by software. If the condition can be corrected, this bit should be cleared by software before resuming normal operation.
See “TLB Initialization” on page 25 for a discussion of software TLB initialization used to avoid a machine check exception during processor initialization.
If this bit is not implemented, it must be ignored on write and read as zero.
Software should not write a 1 to this bit when its value is a 0, thereby causing a 0-to-1 transition. If such a transition is caused by software, it is <b>UNPREDICTABLE</b> whether hardware ignores the write, accepts the write with no side effects, or accepts the write and initiates a machine check exception.</td><td>R/W</td><td>0</td><td>Required if the processor detects and reports a match on multiple TLB entries</td></tr>
<tr><td>SR</td><td>20</td><td>Indicates that the entry through the reset exception vector was due to a Soft Reset:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Not Soft Reset (NMI or Reset)</td></tr>
<tr><td>1</td><td>Soft Reset</td></tr>
</table>
If this bit is not implemented, it must be ignored on write and read as zero.
Software should not write a 1 to this bit when its value is a 0, thereby causing a 0-to-1 transition. If such a transition is caused by software, it is <b>UNPREDICTABLE</b> whether hardware ignores or accepts the write.</td><td>R/W</td><td>1 for Soft Reset; 0 otherwise</td><td>Required if Soft Reset is implemented</td></tr>
</table>

Table 8.21 Status Register Field Descriptions (Continued)

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read / Write</th><th rowspan="2">Reset State</th><th rowspan="2">Compliance</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>NMI</td><td>19</td><td>Indicates that the entry through the reset exception vector was due to an NMI exception:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Not NMI (Soft Reset or Reset)</td></tr>
<tr><td>1</td><td>NMI</td></tr>
</table>
If this bit is not implemented, it must be ignored on write and read as zero.
Software should not write a 1 to this bit when its value is a 0, thereby causing a 0-to-1 transition. If such a transition is caused by software, it is **UNPREDICTABLE** whether hardware ignores or accepts the write.</td><td>R/W</td><td>1 for NMI;
0 otherwise</td><td>Required if NMI is implemented</td></tr>
<tr><td>0</td><td>18</td><td>Must be written as zero; returns zero on read.</td><td>0</td><td>0</td><td>Reserved</td></tr>
<tr><td>Impl</td><td>17..16</td><td>These bits are implementation dependent and are not defined by the architecture. If they are not implemented, they must be ignored on write and read as zero.</td><td></td><td>Undefined</td><td>Optional</td></tr>
<tr><td>IM7..IM2</td><td>15..10</td><td>Interrupt Mask: Controls the enabling of each of the hardware interrupts. Refer to “Interrupts” on page 31 for a complete discussion of enabled interrupts.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Interrupt request disabled</td></tr>
<tr><td>1</td><td>Interrupt request enabled</td></tr>
</table>
In implementations of Release 2 of the Architecture in which EIC interrupt mode is enabled (Config3$_{VEIC}$ = 1), these bits take on a different meaning and are interpreted as the IPL field, described below.</td><td>R/W</td><td>Undefined</td><td>Required</td></tr>
<tr><td>IPL</td><td>15..10</td><td>Interrupt Priority Level.
In implementations of Release 2 of the Architecture in which EIC interrupt mode is enabled (Config3$_{VEIC}$ = 1), this field is the encoded (0..63) value of the current IPL. An interrupt will be signaled only if the requested IPL is higher than this value.
If EIC interrupt mode is not enabled (Config3$_{VEIC}$ = 0), these bits take on a different meaning and are interpreted as the IM7..IM2 bits, described above.</td><td>R/W</td><td>Undefined</td><td>Optional (Release 2 and EIC interrupt mode only)</td></tr>
</table>

Table 8.21 Status Register Field Descriptions (Continued)

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read / Write</th><th rowspan="2">Reset State</th><th rowspan="2">Compliance</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>IM1..IM0</td><td>9..8</td><td>Interrupt Mask: Controls the enabling of each of the software interrupts. Refer to “Interrupts” on page 31 for a complete discussion of enabled interrupts.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Interrupt request disabled</td></tr>
<tr><td>1</td><td>Interrupt request enabled</td></tr>
</table>
In implementations of Release 2 of the Architecture in which EIC interrupt mode is enabled (Config3$_{VEIC}$ = 1), these bits are writable, but have no effect on the interrupt system.</td><td>R/W</td><td>Undefined</td><td>Required</td></tr>
<tr><td>KX</td><td>7</td><td>Enables access to 64-bit kernel address space on 64-bit MIPS processors. Not used by MIPS32 processors. This bit must be ignored on write and read as zero.</td><td>R</td><td>0</td><td>Reserved</td></tr>
<tr><td>SX</td><td>6</td><td>Enables access to 64-bit supervisor address space on 64-bit MIPS processors. Not used by MIPS32 processors. This bit must be ignored on write and read as zero.</td><td>R</td><td>0</td><td>Reserved</td></tr>
<tr><td>UX</td><td>5</td><td>Enables access to 64-bit user address space on 64-bit MIPS processors Not used by MIPS32 processors. This bit must be ignored on write and read as zero.</td><td>R</td><td>0</td><td>Reserved</td></tr>
<tr><td>KSU</td><td>4..3</td><td>If Supervisor Mode is implemented, the encoding of this field denotes the base operating mode of the processor. See “MIPS32 Operating Modes” on page 17 for a full discussion of operating modes. The encoding of this field is:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0b00</td><td>Base mode is Kernel Mode</td></tr>
<tr><td>0b01</td><td>Base mode is Supervisor Mode</td></tr>
<tr><td>0b10</td><td>Base mode is User Mode</td></tr>
<tr><td>0b11</td><td>Reserved. The operation of the processor is <b>UNDEFINED</b> if this value is written to the KSU field</td></tr>
</table>
Note: This field overlaps the UM and R0 fields, described below.</td><td>R/W</td><td>Undefined</td><td>Required if Supervisor Mode is implemented; Optional otherwise</td></tr>
</table>

Table 8.21 Status Register Field Descriptions (Continued)

<table>
<thead>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read / Write</th><th rowspan="2">Reset State</th><th rowspan="2">Compliance</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
</thead>
<tbody>
<tr><td>UM</td><td>4</td><td>If Supervisor Mode is not implemented, this bit denotes the base operating mode of the processor. See “MIPS32 Operating Modes” on page 17 for a full discussion of operating modes. The encoding of this bit is:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Base mode is Kernel Mode</td></tr>
<tr><td>1</td><td>Base mode is User Mode</td></tr>
</table>
Note: This bit overlaps the KSU field, described above.</td><td>R/W</td><td>Undefined</td><td>Required</td></tr>
<tr><td>R0</td><td>3</td><td>If Supervisor Mode is not implemented, this bit is reserved. This bit must be ignored on write and read as zero.
Note: This bit overlaps the KSU field, described above.</td><td>R</td><td>0</td><td>Reserved</td></tr>
<tr><td>ERL</td><td>2</td><td>Error Level; Set by the processor when a Reset, Soft Reset, NMI or Cache Error exception are taken.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Normal level</td></tr>
<tr><td>1</td><td>Error level</td></tr>
</table>
When ERL is set:
• The processor is running in kernel mode
• Hardware and software interrupts are disabled
• The ERET instruction will use the return address held in ErrorEPC instead of EPC
• Segment kuseg is treated as an unmapped and uncached region. See “Address Translation for the kuseg Segment when StatusERL = 1” on page 24. This allows main memory to be accessed in the presence of cache errors. The operation of the processor is <b>UNDEFINED</b> if the ERL bit is set while the processor is executing instructions from kuseg.</td><td>R/W</td><td>1</td><td>Required</td></tr>
<tr><td>EXL</td><td>1</td><td>Exception Level; Set by the processor when any exception other than Reset, Soft Reset, NMI or Cache Error exception are taken.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Normal level</td></tr>
<tr><td>1</td><td>Exception level</td></tr>
</table>
When EXL is set:
• The processor is running in Kernel Mode
• Hardware and software interrupts are disabled.
• TLB Refill exceptions use the general exception vector instead of the TLB Refill vector.
• EPC, Cause_{BD} and SRSCtl (implementations of Release 2 of the Architecture only) will not be updated if another exception is taken</td><td>R/W</td><td>Undefined</td><td>Required</td></tr>
</tbody>
</table>

Table 8.21 Status Register Field Descriptions (Continued)

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read / Write</th><th rowspan="2">Reset State</th><th rowspan="2">Compliance</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>IE</td><td>0</td><td>Interrupt Enable: Acts as the master enable for software and hardware interrupts:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Interrupts are disabled</td></tr>
<tr><td>1</td><td>Interrupts are enabled</td></tr>
</table>
In Release 2 of the Architecture, this bit may be modified separately via the DI and EI instructions.</td><td>R/W</td><td>Undefined</td><td>Required</td></tr>
</table>

[1] The TS bit originally indicated a “TLB Shutdown” condition in which circuits detected multiple TLB matches and shutdown the TLB to prevent physical damage. In newer designs, multiple TLB matches do not cause physical damage to the TLB structure, so the TS bit retains its name, but is simply an indicator to the machine check exception handler that multiple TLB matches were detected and reported by the processor.

Programming Note:

In Release 2 of the Architecture, the EHB instruction can be used to make interrupt state changes visible when the *IM*, *IPL*, *ERL*, *EXL*, or *IE* fields of the *Status* register are written. See “Software Hazards and the Interrupt System” on page 42.

8.20 IntCtl Register (CP0 Register 12, Select 1)

Compliance Level: *Required* (Release 2).

The *IntCtl* register controls the expanded interrupt capability added in Release 2 of the Architecture, including vectored interrupts and support for an external interrupt controller. This register does not exist in implementations of Release 1 of the Architecture.

Figure 8-17 shows the format of the *IntCtl* register; Table 8.22 describes the *IntCtl* register fields.

Figure 8-17 IntCtl Register Format

31 29	28 26	25 10	9 5	4 0
IPTI	IPPCI	0 00 0000 0000 0000 00	VS	0

Table 8.22 IntCtl Register Field Descriptions

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read / Write</th><th rowspan="2">Reset State</th><th rowspan="2">Compliance</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>IPTI</td><td>31..29</td><td>For Interrupt Compatibility and Vectored Interrupt modes, this field specifies the IP number to which the Timer Interrupt request is merged, and allows software to determine whether to consider Cause$_{TI}$ for a potential interrupt.
<table>
<tr><th>Encoding</th><th>IP bit</th><th>Hardware Interrupt Source</th></tr>
<tr><td>2</td><td>2</td><td>HW0</td></tr>
<tr><td>3</td><td>3</td><td>HW1</td></tr>
<tr><td>4</td><td>4</td><td>HW2</td></tr>
<tr><td>5</td><td>5</td><td>HW3</td></tr>
<tr><td>6</td><td>6</td><td>HW4</td></tr>
<tr><td>7</td><td>7</td><td>HW5</td></tr>
</table>
The value of this field is <b>UNPREDICTABLE</b> if External Interrupt Controller Mode is both implemented and enabled. The external interrupt controller is expected to provide this information for that interrupt mode.</td><td>R</td><td>Preset or Externally Set</td><td>Required</td></tr>
</table>

Table 8.22 IntCtl Register Field Descriptions (Continued)

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read / Write</th><th rowspan="2">Reset State</th><th rowspan="2">Compliance</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>IPPCI</td><td>28..26</td><td>For Interrupt Compatibility and Vectored Interrupt modes, this field specifies the IP number to which the Performance Counter Interrupt request is merged, and allows software to determine whether to consider Cause$_{PCI}$ for a potential interrupt.
<table>
<tr><th>Encoding</th><th>IP bit</th><th>Hardware Interrupt Source</th></tr>
<tr><td>2</td><td>2</td><td>HW0</td></tr>
<tr><td>3</td><td>3</td><td>HW1</td></tr>
<tr><td>4</td><td>4</td><td>HW2</td></tr>
<tr><td>5</td><td>5</td><td>HW3</td></tr>
<tr><td>6</td><td>6</td><td>HW4</td></tr>
<tr><td>7</td><td>7</td><td>HW5</td></tr>
</table>
The value of this field is <b>UNPREDICTABLE</b> if External Interrupt Controller Mode is both implemented and enabled. The external interrupt controller is expected to provide this information for that interrupt mode.
If performance counters are not implemented (Config1$_{PC}$ = 0), this field returns zero on read.</td><td>R</td><td>Preset or Externally Set</td><td>Optional (Performance Counters Implemented)</td></tr>
<tr><td>0</td><td>25..10</td><td>Must be written as zero; returns zero on read.</td><td>0</td><td>0</td><td>Reserved</td></tr>
<tr><td>VS</td><td>9..5</td><td>Vector Spacing. If vectored interrupts are implemented (as denoted by Config3$_{VInt}$ or Config3$_{VEIC}$), this field specifies the spacing between vectored interrupts.
<table>
<tr><th rowspan="2">Encoding</th><th colspan="2">Spacing Between Vectors</th></tr>
<tr><th>(hex)</th><th>(decimal)</th></tr>
<tr><td>0x00</td><td>0x000</td><td>0</td></tr>
<tr><td>0x01</td><td>0x020</td><td>32</td></tr>
<tr><td>0x02</td><td>0x040</td><td>64</td></tr>
<tr><td>0x04</td><td>0x080</td><td>128</td></tr>
<tr><td>0x08</td><td>0x100</td><td>256</td></tr>
<tr><td>0x10</td><td>0x200</td><td>512</td></tr>
</table>
All other values are reserved. The operation of the processor is <b>UNDEFINED</b> if a reserved value is written to this field.
If neither EIC interrupt mode nor VI mode are implemented (Config3$_{VEIC}$ = 0 and Config3$_{VINT}$ = 0), this field is ignored on write and reads as zero.</td><td>R/W</td><td>0</td><td>Optional</td></tr>
<tr><td>0</td><td>4..0</td><td>Must be written as zero; returns zero on read.</td><td>0</td><td>0</td><td>Reserved</td></tr>
</table>

8.21 SRSCtl Register (CP0 Register 12, Select 2)

Compliance Level: *Required* (Release 2).

The *SRSCtl* register controls the operation of GPR shadow sets in the processor. This register does not exist in implementations of the architecture prior to Release 2.

Figure 8-18 shows the format of the *SRSCtl* register; Table 8.23 describes the *SRSCtl* register fields.

Figure 8-18 SRSCtl Register Format

31 30	29 26	25 22	21 18	17 16	15 12	11 10	9 6	5 4	3 0
0 00	HSS	0 00 00	EICSS	0 00	ESS	0 00	PSS	0 00	CSS

Table 8.23 SRSCtl Register Field Descriptions

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
0	31..30	Must be written as zeros; returns zero on read.	0	0	Reserved
HSS	29..26	Highest Shadow Set. This field contains the highest shadow set number that is implemented by this processor. A value of zero in this field indicates that only the normal GPRs are implemented. A non-zero value in this field indicates that the implemented shadow sets are numbered 0..n, where n is the value of the field. The value in this field also represents the highest value that can be written to the *ESS*, *EICSS*, *PSS*, and *CSS* fields of this register, or to any of the fields of the *SRSMap* register. The operation of the processor is **UNDEFINED** if a value larger than the one in this field is written to any of these other values.	R	Preset	Required
0	25..22	Must be written as zeros; returns zero on read.	0	0	Reserved
EICSS	21..18	EIC interrupt mode shadow set. If Config3$_{VEIC}$ is 1 (EIC interrupt mode is enabled), this field is loaded from the external interrupt controller for each interrupt request and is used in place of the *SRSMap* register to select the current shadow set for the interrupt. See "External Interrupt Controller Mode" on page 38 for a discussion of EIC interrupt mode. If Config3$_{VEIC}$ is 0, this field must be written as zero, and returns zero on read.	R	Undefined	Required (EIC interrupt mode only)
0	17..16	Must be written as zeros; returns zero on read.	0	0	Reserved

Table 8.23 SRSCtl Register Field Descriptions (Continued)

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
ESS	15..12	Exception Shadow Set. This field specifies the shadow set to use on entry to Kernel Mode caused by any exception other than a vectored interrupt. The operation of the processor is **UNDEFINED** if software writes a value into this field that is greater than the value in the HSS field.	R/W	0	Required
0	11..10	Must be written as zeros; returns zero on read.	0	0	Reserved
PSS	9..6	Previous Shadow Set. If GPR shadow registers are implemented, and with the exclusions noted in the next paragraph, this field is copied from the CSS field when an exception or interrupt occurs. An ERET instruction copies this value back into the CSS field if $\text{Status}_{\text{BEV}}$ = 0. This field is not updated on any exception which sets $\text{Status}_{\text{ERL}}$ to 1 (i.e., NMI or cache error), an entry into EJTAG Debug mode, or any exception or interrupt that occurs with $\text{Status}_{\text{EXL}}$ = 1, or $\text{Status}_{\text{BEV}}$ = 1. The operation of the processor is **UNDEFINED** if software writes a value into this field that is greater than the value in the HSS field.	R/W	0	Required
0	5..4	Must be written as zeros; returns zero on read.	0	0	Reserved
CSS	3..0	Current Shadow Set. If GPR shadow registers are implemented, this field is the number of the current GPR set. With the exclusions noted in the next paragraph, this field is updated with a new value on any interrupt or exception, and restored from the *PSS* field on an ERET. Table 8.24 describes the various sources from which the *CSS* field is updated on an exception or interrupt. This field is not updated on any exception which sets $\text{Status}_{\text{ERL}}$ to 1 (i.e., NMI or cache error), an entry into EJTAG Debug mode, or any exception or interrupt that occurs with $\text{Status}_{\text{EXL}}$ = 1, or $\text{Status}_{\text{BEV}}$ = 1. Neither is it updated on an ERET with $\text{Status}_{\text{ERL}}$ = 1 or $\text{Status}_{\text{BEV}}$ = 1. The value of *CSS* can be changed directly by software only by writing the *PSS* field and executing an ERET instruction.	R	0	Required

Table 8.24 Sources for new $\text{SRSCtl}_{\text{CSS}}$ on an Exception or Interrupt

Exception Type	Condition	$\text{SRSCtl}_{\text{CSS}}$ Source	Comment
Exception	All	$\text{SRSCtl}_{\text{ESS}}$	

Table 8.24 Sources for new $\text{SRSCtl}_{\text{CSS}}$ on an Exception or Interrupt

Exception Type	Condition	$\text{SRSCtl}_{\text{CSS}}$ Source	Comment
Non-Vectored Interrupt	$\text{Cause}_{\text{IV}} = 0$	$\text{SRSCtl}_{\text{ESS}}$	Treat as exception
Vectored Interrupt	$\text{Cause}_{\text{IV}} = 1$ and $\text{Config3}_{\text{VEIC}} = 0$ and $\text{Config3}_{\text{VInt}} = 1$	$\text{SRSMap}_{\text{VectNum}\times4+3..\text{VectNum}\times4}$	Source is internal map register
Vectored EIC Interrupt	$\text{Cause}_{\text{IV}} = 1$ and $\text{Config3}_{\text{VEIC}} = 1$	$\text{SRSCtl}_{\text{EICSS}}$	Source is external interrupt controller.

Programming Note:

A software change to the PSS field creates an instruction hazard between the write of the *SRSCtl* register and the use of a RDPGPR or WRPGPR instruction. This hazard must be cleared with a JR.HB or JALR.HB instruction as described in “Hazard Clearing Instructions and Events” on page 68. A hardware change to the PSS field as the result of interrupt or exception entry is automatically cleared for the execution of the first instruction in the interrupt or exception handler.

8.22 SRSMap Register (CP0 Register 12, Select 3)

Compliance Level: *Required* in Release 2 of the Architecture if Additional Shadow Sets and Vectored Interrupt Mode are Implemented

The *SRSMap* register contains 8 4-bit fields that provide the mapping from an vector number to the shadow set number to use when servicing such an interrupt. The values from this register are not used for a non-interrupt exception, or a non-vectored interrupt (Cause_{IV} = 0 or IntCtl_{VS} = 0). In such cases, the shadow set number comes from SRSCtl_{ESS}.

If SRSCtl_{HSS} is zero, the results of a software read or write of this register are **UNPREDICTABLE**.

The operation of the processor is **UNDEFINED** if a value is written to any field in this register that is greater than the value of SRSCtl_{HSS}.

The *SRSMap* register contains the shadow register set numbers for vector numbers 7..0. The same shadow set number can be established for multiple interrupt vectors, creating a many-to-one mapping from a vector to a single shadow register set number.

Figure 8-19 shows the format of the *SRSMap* register; Table 8.25 describes the *SRSMap* register fields.

Figure 8-19 SRSMap Register Format

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
SSV7	SSV6	SSV5	SSV4	SSV3	SSV2	SSV1	SSV0

Table 8.25 SRSMap Register Field Descriptions

Fields					
Name	Bits	Description	Read / Write	Reset State	Compliance
SSV7	31..28	Shadow register set number for Vector Number 7	R/W	0	Required
SSV6	27..24	Shadow register set number for Vector Number 6	R/W	0	Required
SSV5	23..20	Shadow register set number for Vector Number 5	R/W	0	Required
SSV4	19..16	Shadow register set number for Vector Number 4	R/W	0	Required
SSV3	15..12	Shadow register set number for Vector Number 3	R/W	0	Required
SSV2	11..8	Shadow register set number for Vector Number 2	R/W	0	Required
SSV1	7..4	Shadow register set number for Vector Number 1	R/W	0	Required
SSV0	3..0	Shadow register set number for Vector Number 0	R/W	0	Required

8.23 Cause Register (CP0 Register 13, Select 0)

Compliance Level: *Required*.

The *Cause* register primarily describes the cause of the most recent exception. In addition, fields also control software interrupt requests and the vector through which interrupts are dispatched. With the exception of the $IP_{1..0}$, DC, IV, and WP fields, all fields in the *Cause* register are read-only. Release 2 of the Architecture added optional support for an External Interrupt Controller (EIC) interrupt mode, in which $IP_{7..2}$ are interpreted as the Requested Interrupt Priority Level (RIPL).

Figure 8-20 shows the format of the *Cause* register; Table 8.26 describes the *Cause* register fields.

Figure 8-20 Cause Register Format

31	30	29 28	27	26	25 24	23	22	21 16	15 10	9 8	7	6 2	1 0
BD	TI	CE	DC	PCI	0	IV	WP	0	IP7..IP2	IP1..IP0	0	Exc Code	0
									RIPL				

Table 8.26 Cause Register Field Descriptions

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read / Write</th><th rowspan="2">Reset State</th><th rowspan="2">Compliance</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>BD</td><td>31</td><td>Indicates whether the last exception taken occurred in a branch delay slot:<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Not in delay slot</td></tr><tr><td>1</td><td>In delay slot</td></tr></table>The processor updates BD only if $Status_{EXL}$ was zero when the exception occurred.</td><td>R</td><td>Undefined</td><td>Required</td></tr>
<tr><td>TI</td><td>30</td><td>Timer Interrupt. In an implementation of Release 2 of the Architecture, this bit denotes whether a timer interrupt is pending (analogous to the IP bits for other interrupt types):<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>No timer interrupt is pending</td></tr><tr><td>1</td><td>Timer interrupt is pending</td></tr></table>In an implementation of Release 1 of the Architecture, this bit must be written as zero and returns zero on read.</td><td>R</td><td>Undefined</td><td>Required
(Release 2)</td></tr>
</table>

Table 8.26 Cause Register Field Descriptions

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read / Write</th><th rowspan="2">Reset State</th><th rowspan="2">Compliance</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>CE</td><td>29..28</td><td>Coprocessor unit number referenced when a Coprocessor Unusable exception is taken. This field is loaded by hardware on every exception, but is <b>UNPREDICTABLE</b> for all exceptions except for Coprocessor Unusable.</td><td>R</td><td>Undefined</td><td>Required</td></tr>
<tr><td>DC</td><td>27</td><td>Disable <i>Count</i> register. In some power-sensitive applications, the <i>Count</i> register is not used but may still be the source of some noticeable power dissipation. This bit allows the <i>Count</i> register to be stopped in such situations.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Enable counting of <i>Count</i> register</td></tr>
<tr><td>1</td><td>Disable counting of <i>Count</i> register</td></tr>
</table>
In an implementation of Release 1 of the Architecture, this bit must be written as zero, and returns zero on read.</td><td>R/W</td><td>0</td><td>Required (Release 2)</td></tr>
<tr><td>PCI</td><td>26</td><td>Performance Counter Interrupt. In an implementation of Release 2 of the Architecture, this bit denotes whether a performance counter interrupt is pending (analogous to the IP bits for other interrupt types):
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>No performance counter interrupt is pending</td></tr>
<tr><td>1</td><td>Performance counter interrupt is pending</td></tr>
</table>
In an implementation of Release 1 of the Architecture, or if performance counters are not implemented ($Config1_{PC}$ = 0), this bit must be written as zero and returns zero on read.</td><td>R</td><td>Undefined</td><td>Required (Release 2 and performance counters implemented)</td></tr>
<tr><td>IV</td><td>23</td><td>Indicates whether an interrupt exception uses the general exception vector or a special interrupt vector:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Use the general exception vector (0x180)</td></tr>
<tr><td>1</td><td>Use the special interrupt vector (0x200)</td></tr>
</table>
In implementations of Release 2 of the architecture, if the $Cause_{IV}$ is 1 and $Status_{BEV}$ is 0, the special interrupt vector represents the base of the vectored interrupt table.</td><td>R/W</td><td>Undefined</td><td>Required</td></tr>
</table>

Table 8.26 Cause Register Field Descriptions

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read / Write</th><th rowspan="2">Reset State</th><th rowspan="2">Compliance</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>WP</td><td>22</td><td>Indicates that a watch exception was deferred because Status$_{EXL}$ or Status$_{ERL}$ were a one at the time the watch exception was detected. This bit both indicates that the watch exception was deferred, and causes the exception to be initiated once Status$_{EXL}$ and Status$_{ERL}$ are both zero. As such, software must clear this bit as part of the watch exception handler to prevent a watch exception loop.
Software should not write a 1 to this bit when its value is a 0, thereby causing a 0-to-1 transition. If such a transition is caused by software, it is <b>UNPREDICTABLE</b> whether hardware ignores the write, accepts the write with no side effects, or accepts the write and initiates a watch exception once Status$_{EXL}$ and Status$_{ERL}$ are both zero.
If watch registers are not implemented, this bit must be ignored on write and read as zero.</td><td>R/W</td><td>Undefined</td><td>Required if watch registers are implemented</td></tr>
<tr><td>IP7..IP2</td><td>15..10</td><td>Indicates an interrupt is pending:
<table>
<tr><th>Bit</th><th>Name</th><th>Meaning</th></tr>
<tr><td>15</td><td>IP7</td><td>Hardware interrupt 5</td></tr>
<tr><td>14</td><td>IP6</td><td>Hardware interrupt 4</td></tr>
<tr><td>13</td><td>IP5</td><td>Hardware interrupt 3</td></tr>
<tr><td>12</td><td>IP4</td><td>Hardware interrupt 2</td></tr>
<tr><td>11</td><td>IP3</td><td>Hardware interrupt 1</td></tr>
<tr><td>10</td><td>IP2</td><td>Hardware interrupt 0</td></tr>
</table>
In implementations of Release 1 of the Architecture, timer and performance counter interrupts are combined in an implementation-dependent way with hardware interrupt 5.
In implementations of Release 2 of the Architecture in which EIC interrupt mode is not enabled (Config3$_{VEIC}$ = 0), timer and performance counter interrupts are combined in an implementation-dependent way with any hardware interrupt. If EIC interrupt mode is enabled (Config3$_{VEIC}$ = 1), these bits take on a different meaning and are interpreted as the RIPL field, described below.</td><td>R</td><td>Undefined</td><td>Required</td></tr>
<tr><td>RIPL</td><td>15..10</td><td>Requested Interrupt Priority Level.
In implementations of Release 2 of the Architecture in which EIC interrupt mode is enabled (Config3$_{VEIC}$ = 1), this field is the encoded (0..63) value of the requested interrupt. A value of zero indicates that no interrupt is requested.
If EIC interrupt mode is not enabled (Config3$_{VEIC}$ = 0), these bits take on a different meaning and are interpreted as the IP7..IP2 bits, described above.</td><td>R</td><td>Undefined</td><td>Optional (Release 2 and EIC interrupt mode only)</td></tr>
</table>

Table 8.26 Cause Register Field Descriptions

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read / Write</th><th rowspan="2">Reset State</th><th rowspan="2">Compliance</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>IP1..IP0</td><td>9..8</td><td>Controls the request for software interrupts:
<table>
<tr><th>Bit</th><th>Name</th><th>Meaning</th></tr>
<tr><td>9</td><td>IP1</td><td>Request software interrupt 1</td></tr>
<tr><td>8</td><td>IP0</td><td>Request software interrupt 0</td></tr>
</table>
An implementation of Release 2 of the Architecture which also implements EIC interrupt mode exports these bits to the external interrupt controller for prioritization with other interrupt sources.</td><td>R/W</td><td>Undefined</td><td>Required</td></tr>
<tr><td>ExcCode</td><td>6..2</td><td>Exception code - see Table 8.27</td><td>R</td><td>Undefined</td><td>Required</td></tr>
<tr><td>0</td><td>25..24, 21..16, 7, 1..0</td><td>Must be written as zero; returns zero on read.</td><td>0</td><td>0</td><td>Reserved</td></tr>
</table>

Table 8.27 Cause Register ExcCode Field

Exception Code Value		Mnemonic	Description
Decimal	Hexadecimal		
0	0x00	Int	Interrupt
1	0x01	Mod	TLB modification exception
2	0x02	TLBL	TLB exception (load or instruction fetch)
3	0x03	TLBS	TLB exception (store)
4	0x04	AdEL	Address error exception (load or instruction fetch)
5	0x05	AdES	Address error exception (store)
6	0x06	IBE	Bus error exception (instruction fetch)
7	0x07	DBE	Bus error exception (data reference: load or store)
8	0x08	Sys	Syscall exception
9	0x09	Bp	Breakpoint exception. If EJTAG is implemented and an SDBBP instruction is executed while the processor is running in EJTAG Debug Mode, this value is written to the Debug$_{\text{DExcCode}}$ field to denote an SDBBP in Debug Mode.
10	0x0a	RI	Reserved instruction exception
11	0x0b	CpU	Coprocessor Unusable exception
12	0x0c	Ov	Arithmetic Overflow exception
13	0x0d	Tr	Trap exception

Table 8.27 Cause Register ExcCode Field

Exception Code Value		Mnemonic	Description
Decimal	Hexadecimal		
14	0x0e	-	Reserved
15	0x0f	FPE	Floating point exception
16-17	0x10-0x11	-	Available for implementation dependent use
18	0x12	C2E	Reserved for precise Coprocessor 2 exceptions
19-21	0x13-0x15	-	Reserved
22	0x16	MDMX	MDMX Unusable Exception (MDMX ASE)
23	0x17	WATCH	Reference to WatchHi/WatchLo address
24	0x18	MCheck	Machine check
25	0x19	Thread	Thread Allocation, Deallocation, or Scheduling Exceptions (MIPS® MT ASE)
26-29	0x20-0x1d	-	Reserved
30	0x1e	CacheErr	Cache error. In normal mode, a cache error exception has a dedicated vector and the Cause register is not updated. If EJTAG is implemented and a cache error occurs while in Debug Mode, this code is written to the $\text{Debug}_{\text{DExcCode}}$ field to indicate that re-entry to Debug Mode was caused by a cache error.
31	0x1f	-	Reserved

Programming Note:

In Release 2 of the Architecture, the EHB instruction can be used to make interrupt state changes visible when the $IP_{1..0}$ field of the *Cause* register is written. See “Software Hazards and the Interrupt System” on page 42.

8.24 Exception Program Counter (CP0 Register 14, Select 0)

Compliance Level: *Required*.

The *Exception Program Counter* (*EPC*) is a read/write register that contains the address at which processing resumes after an exception has been serviced. All bits of the *EPC* register are significant and must be writable.

Unless the *EXL* bit in the *Status* register is already a 1, the processor writes the *EPC* register when an exception occurs.

- For synchronous (precise) exceptions, *EPC* 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 exception causing instruction is in a branch delay slot, and the *Branch Delay* bit in the *Cause* register is set.
- For asynchronous (imprecise) exceptions, *EPC* contains the address of the instruction at which to resume execution.

The processor reads the *EPC* register as the result of execution of the ERET instruction.

Software may write the *EPC* register to change the processor resume address and read the *EPC* register to determine at what address the processor will resume.

Figure 8-21 shows the format of the *EPC* register; Table 8.28 describes the *EPC* register fields.

Figure 8-21 EPC Register Format

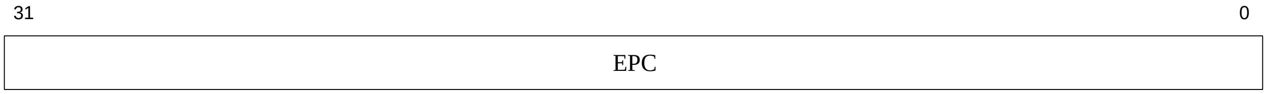


Table 8.28 EPC Register Field Descriptions

Fields					
Name	**Bits**	**Description**	**Read / Write**	**Reset State**	**Compliance**
EPC	31..0	Exception Program Counter	R/W	Undefined	Required

8.24.1 Special Handling of the EPC Register in Processors That Implement the MIPS16e ASE

In processors that implement the MIPS16e ASE, the *EPC* register requires special handling.

When the processor writes the *EPC* register, it combines the address at which processing resumes with the value of the *ISA Mode* register:

```
EPC ← resumePC$_{31..1}$ || ISAMode$_0$
```

“resumePC” is the address at which processing resumes, as described above.

When the processor reads the *EPC* register, it distributes the bits to the *PC* and *ISAMode* registers:

```
PC ← $EPC_{31..1}$ ‖ 0
ISAMode ← $EPC_0$
```

Software reads of the *EPC* register simply return to a GPR the last value written with no interpretation. Software writes to the *EPC* register store a new value which is interpreted by the processor as described above.

8.25 Processor Identification (CP0 Register 15, Select 0)

Compliance Level: *Required*.

The *Processor Identification* (*PRId*) register is a 32 bit read-only register that contains information identifying the manufacturer, manufacturer options, processor identification and revision level of the processor. Figure 8-22 shows the format of the *PRId* register; Table 8.29 describes the *PRId* register fields.

Figure 8-22 PRId Register Format

31 24	23 16	15 8	7 0
Company Options	Company ID	Processor ID	Revision

Table 8.29 PRId Register Field Descriptions

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read / Write</th><th rowspan="2">Reset State</th><th rowspan="2">Compliance</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>Company Options</td><td>31..24</td><td>Available to the designer or manufacturer of the processor for company-dependent options. The value in this field is not specified by the architecture. If this field is not implemented, it must read as zero.</td><td>R</td><td>Preset</td><td>Optional</td></tr>
<tr><td>Company ID</td><td>23..16</td><td>Identifies the company that designed or manufactured the processor.
Software can distinguish a MIPS32 or MIPS64 processor from one implementing an earlier MIPS ISA by checking this field for zero. If it is non-zero the processor implements the MIPS32 or MIPS64 Architecture.
Company IDs are assigned by MIPS Technologies when a MIPS32 or MIPS64 license is acquired. The encodings in this field are:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Not a MIPS32 or MIPS64 processor</td></tr>
<tr><td>1</td><td>MIPS Technologies, Inc.</td></tr>
<tr><td>2-255</td><td>Contact MIPS Technologies, Inc. for the list of Company ID assignments</td></tr>
</table></td><td>R</td><td>Preset</td><td>Required</td></tr>
<tr><td>Processor ID</td><td>15..8</td><td>Identifies the type of processor. This field allows software to distinguish between various processor implementations within a single company, and is qualified by the CompanyID field, described above. The combination of the CompanyID and ProcessorID fields creates a unique number assigned to each processor implementation.</td><td>R</td><td>Preset</td><td>Required</td></tr>
</table>

Table 8.29 PRId Register Field Descriptions

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
Revision	7..0	Specifies the revision number of the processor. This field allows software to distinguish between one revision and another of the same processor type. If this field is not implemented, it must read as zero.	R	Preset	Optional

Software should not use the fields of this register to infer configuration information about the processor. Rather, the configuration registers should be used to determine the capabilities of the processor. Programmers who identify cases in which the configuration registers are not sufficient, requiring them to revert to check on the *PRId* register value, should send email to architecture@mips.com, reporting the specific case.

8.26 EBase Register (CP0 Register 15, Select 1)

Compliance Level: *Required* (Release 2).

The *EBase* register is a read/write register containing the base address of the exception vectors used when $\text{Status}_{\text{BEV}}$ equals 0, and a read-only CPU number value that may be used by software to distinguish different processors in a multi-processor system.

The *EBase* register provides the ability for software to identify the specific processor within a multi-processor system, and allows the exception vectors for each processor to be different, especially in systems composed of heterogeneous processors. Bits 31..12 of the *EBase* register are concatenated with zeros to form the base of the exception vectors when $\text{Status}_{\text{BEV}}$ is 0. The exception vector base address comes from the fixed defaults (see 5.2.2 "Exception Vector Locations" on page 45) when $\text{Status}_{\text{BEV}}$ is 1, or for any EJTAG Debug exception. The reset state of bits 31..12 of the *EBase* register initialize the exception base register to `0x8000.0000`, providing backward compatibility with Release 1 implementations.

Bits 31..30 of the *EBase* 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.

If the value of the exception base register is to be changed, this must be done with $\text{Status}_{\text{BEV}}$ equal 1. The operation of the processor is **UNDEFINED** if the Exception Base field is written with a different value when $\text{Status}_{\text{BEV}}$ is 0.

Figure 8-23 shows the format of the *EBase* register; Table 8.30 describes the *EBase* register fields.

Figure 8-23 EBase Register Format

31	30	29 12	11 10	9 0
1	0	Exception Base	0 0	CPUNum

Table 8.30 EBase Register Field Descriptions

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
1	31	This bit is ignored on write and returns one on read.	R	1	Required
0	30	This bit is ignored on write and returns zero on read.	R	0	Required
Exception Base	29..12	In conjunction with bits 31..30, this field specifies the base address of the exception vectors when $\text{Status}_{\text{BEV}}$ is zero.	R/W	0	Required
0	11..10	Must be written as zero; returns zero on read.	0	0	Reserved

Table 8.30 EBase Register Field Descriptions

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
CPUNum	9..0	This field specifies the number of the CPU in a multi-processor system and can be used by software to distinguish a particular processor from the others. The value in this field is set by inputs to the processor hardware when the processor is implemented in the system environment. In a single processor system, this value should be set to zero. This field can also be read via RDHWR register 0	R	Preset or Externally Set	Required

Programming Note:

Software must set $\text{EBase}_{15..12}$ to zero in all bit positions less than or equal to the most significant bit in the vector offset. This situation can only occur when a vector offset greater than 0xFFF is generated when an interrupt occurs with *VI* or *EIC* interrupt mode enabled. The operation of the processor is **UNDEFINED** if this condition is not met. Table 8.31 shows the conditions under which each *EBase* bit must be set to zero. *VN* represents the interrupt vector number as described in Table 5.4 and the bit must be set to zero if any of the relationships in the row are true. No *EBase* bits must be set to zero if the interrupt vector spacing is 32 (or zero) bytes.

Table 8.31 Conditions Under Which EBase15..12 Must Be Zero

	Interrupt Vector Spacing in Bytes ($\text{IntCtl}_{\text{VS}}$[1])				
EBase bit	32	64	128	256	512
15	None	None	None	None	VN ≥ 63
14		None	None	VN ≥ 62	VN ≥ 31
13		None	VN ≥ 60	VN ≥ 30	VN ≥ 15
12		VN ≥ 56	VN ≥ 28	VN ≥ 14	VN ≥ 7

[1] See Table 8.22 on page 104

8.27 Configuration Register (CP0 Register 16, Select 0)

Compliance Level: *Required*.

The *Config* register specifies various configuration and capabilities information. Most of the fields in the *Config* register are initialized by hardware during the Reset Exception process, or are constant. Three fields, *K23*, *KU*, and *K0*, must be initialized by software in the reset exception handler.

Figure 8-24 shows the format of the *Config* register; Table 8.32 describes the *Config* register fields.

Figure 8-24 Config Register Format

31	30 28	27 25	24 16	15	14 13	12 10	9 7	6 4	3	2 0
M	K23	KU	Impl	BE	AT	AR	MT	0	VI	K0

Table 8.32 Config Register Field Descriptions

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read / Write</th><th rowspan="2">Reset State</th><th rowspan="2">Compliance</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>M</td><td>31</td><td>Denotes that the <i>Config1</i> register is implemented at a select field value of 1.</td><td>R</td><td>1</td><td>Required</td></tr>
<tr><td>K23</td><td>30:28</td><td>For processors that implement a Fixed Mapping MMU, this field specifies the kseg2 and kseg3 cacheability and coherency attribute. For processors that do not implement a Fixed Mapping MMU, this field reads as zero and is ignored on write.
See “Alternative MMU Organizations” on page 155 for a description of the Fixed Mapping MMU organization.</td><td>R/W</td><td>Undefined for processors with a Fixed Mapping MMU; 0 otherwise</td><td>Optional</td></tr>
<tr><td>KU</td><td>27:25</td><td>For processors that implement a Fixed Mapping MMU, this field specifies the kuseg cacheability and coherency attribute. For processors that do not implement a Fixed Mapping MMU, this field reads as zero and is ignored on write.
See “Alternative MMU Organizations” on page 155 for a description of the Fixed Mapping MMU organization.</td><td>R/W</td><td>Undefined for processors with a Fixed Mapping MMU; 0 otherwise</td><td>Optional</td></tr>
<tr><td>Impl</td><td>24:16</td><td>This field is reserved for implementations. Refer to the processor specification for the format and definition of this field</td><td></td><td>Undefined</td><td>Optional</td></tr>
<tr><td>BE</td><td>15</td><td>Indicates the endian mode in which the processor is running:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Little endian</td></tr>
<tr><td>1</td><td>Big endian</td></tr>
</table></td><td>R</td><td>Preset or Externally Set</td><td>Required</td></tr>
</table>

Table 8.32 Config Register Field Descriptions

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read / Write</th><th rowspan="2">Reset State</th><th rowspan="2">Compliance</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>AT</td><td>14:13</td><td>Architecture type implemented by the processor:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>MIPS32</td></tr>
<tr><td>1</td><td>MIPS64 with access only to 32-bit compatibility segments</td></tr>
<tr><td>2</td><td>MIPS64 with access to all address segments</td></tr>
<tr><td>3</td><td>Reserved</td></tr>
</table></td><td>R</td><td>Preset</td><td>Required</td></tr>
<tr><td>AR</td><td>12:10</td><td>Architecture revision level:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Release 1</td></tr>
<tr><td>1</td><td>Release 2</td></tr>
<tr><td>2-7</td><td>Reserved</td></tr>
</table></td><td>R</td><td>Preset</td><td>Required</td></tr>
<tr><td>MT</td><td>9:7</td><td>MMU Type:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>None</td></tr>
<tr><td>1</td><td>Standard TLB</td></tr>
<tr><td>2</td><td>Standard BAT (see “Block Address Translation” on page 159)</td></tr>
<tr><td>3</td><td>Standard fixed mapping (see “Fixed Mapping MMU” on page 155)</td></tr>
<tr><td>4-7</td><td>Reserved</td></tr>
</table></td><td>R</td><td>Preset</td><td>Required</td></tr>
<tr><td>0</td><td>6:4</td><td>Must be written as zero; returns zero on read.</td><td>0</td><td>0</td><td>Reserved</td></tr>
<tr><td>VI</td><td>3</td><td>Virtual instruction cache (using both virtual indexing and virtual tags):
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Instruction Cache is not virtual</td></tr>
<tr><td>1</td><td>Instruction Cache is virtual</td></tr>
</table></td><td>R</td><td>Preset</td><td>Required</td></tr>
<tr><td>K0</td><td>2:0</td><td>Kseg0 cacheability and coherency attribute. See Table 8.8 on page 81 for the encoding of this field.</td><td>R/W</td><td>Undefined</td><td>Required</td></tr>
</table>

8.28 Configuration Register 1 (CP0 Register 16, Select 1)

Compliance Level: *Required*.

The *Config1* register is an adjunct to the *Config* register and encodes additional capabilities information. All fields in the *Config1* register are read-only.

The Icache and Dcache configuration parameters include encodings for the number of sets per way, the line size, and the associativity. The total cache size for a cache is therefore:

```
Cache Size = Associativity * Line Size * Sets Per Way
```

If the line size is zero, there is no cache implemented.

Figure 8-25 shows the format of the *Config1* register; Table 8.33 describes the *Config1* register fields.

Figure 8-25 Config1 Register Format

31	30 25	24 22	21 19	18 16	15 13	12 10	9 7	6	5	4	3	2	1	0
M	MMU Size - 1	IS	IL	IA	DS	DL	DA	C2	MD	PC	WR	CA	EP	FP

Table 8.33 Config1 Register Field Descriptions

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read / Write</th><th rowspan="2">Reset State</th><th rowspan="2">Compliance</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>M</td><td>31</td><td>This bit is reserved to indicate that a <i>Config2</i> register is present. If the <i>Config2</i> register is not implemented, this bit should read as a 0. If the <i>Config2</i> register is implemented, this bit should read as a 1.</td><td>R</td><td>Preset</td><td>Required</td></tr>
<tr><td>MMU Size - 1</td><td>30..25</td><td>Number of entries in the TLB minus one. The values 0 through 63 is this field correspond to 1 to 64 TLB entries. The value zero is implied by Config$_{MT}$ having a value of ‘none’.</td><td>R</td><td>Preset</td><td>Required</td></tr>
<tr><td>IS</td><td>24:22</td><td>Icache sets per way:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>64</td></tr>
<tr><td>1</td><td>128</td></tr>
<tr><td>2</td><td>256</td></tr>
<tr><td>3</td><td>512</td></tr>
<tr><td>4</td><td>1024</td></tr>
<tr><td>5</td><td>2048</td></tr>
<tr><td>6</td><td>4096</td></tr>
<tr><td>7</td><td>Reserved</td></tr>
</table>
</td><td>R</td><td>Preset</td><td>Required</td></tr>
</table>

Table 8.33 Config1 Register Field Descriptions

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read / Write</th><th rowspan="2">Reset State</th><th rowspan="2">Compliance</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>IL</td><td>21:19</td><td>Icache line size:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>No Icache present</td></tr>
<tr><td>1</td><td>4 bytes</td></tr>
<tr><td>2</td><td>8 bytes</td></tr>
<tr><td>3</td><td>16 bytes</td></tr>
<tr><td>4</td><td>32 bytes</td></tr>
<tr><td>5</td><td>64 bytes</td></tr>
<tr><td>6</td><td>128 bytes</td></tr>
<tr><td>7</td><td>Reserved</td></tr>
</table></td><td>R</td><td>Preset</td><td>Required</td></tr>
<tr><td>IA</td><td>18:16</td><td>Icache associativity:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Direct mapped</td></tr>
<tr><td>1</td><td>2-way</td></tr>
<tr><td>2</td><td>3-way</td></tr>
<tr><td>3</td><td>4-way</td></tr>
<tr><td>4</td><td>5-way</td></tr>
<tr><td>5</td><td>6-way</td></tr>
<tr><td>6</td><td>7-way</td></tr>
<tr><td>7</td><td>8-way</td></tr>
</table></td><td>R</td><td>Preset</td><td>Required</td></tr>
<tr><td>DS</td><td>15:13</td><td>Dcache sets per way:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>64</td></tr>
<tr><td>1</td><td>128</td></tr>
<tr><td>2</td><td>256</td></tr>
<tr><td>3</td><td>512</td></tr>
<tr><td>4</td><td>1024</td></tr>
<tr><td>5</td><td>2048</td></tr>
<tr><td>6</td><td>4096</td></tr>
<tr><td>7</td><td>Reserved</td></tr>
</table></td><td>R</td><td>Preset</td><td>Required</td></tr>
</table>

Table 8.33 Config1 Register Field Descriptions

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read / Write</th><th rowspan="2">Reset State</th><th rowspan="2">Compliance</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>DL</td><td>12:10</td><td>Dcache line size:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>No Dcache present</td></tr>
<tr><td>1</td><td>4 bytes</td></tr>
<tr><td>2</td><td>8 bytes</td></tr>
<tr><td>3</td><td>16 bytes</td></tr>
<tr><td>4</td><td>32 bytes</td></tr>
<tr><td>5</td><td>64 bytes</td></tr>
<tr><td>6</td><td>128 bytes</td></tr>
<tr><td>7</td><td>Reserved</td></tr>
</table></td><td>R</td><td>Preset</td><td>Required</td></tr>
<tr><td>DA</td><td>9:7</td><td>Dcache associativity:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Direct mapped</td></tr>
<tr><td>1</td><td>2-way</td></tr>
<tr><td>2</td><td>3-way</td></tr>
<tr><td>3</td><td>4-way</td></tr>
<tr><td>4</td><td>5-way</td></tr>
<tr><td>5</td><td>6-way</td></tr>
<tr><td>6</td><td>7-way</td></tr>
<tr><td>7</td><td>8-way</td></tr>
</table></td><td>R</td><td>Preset</td><td>Required</td></tr>
<tr><td>C2</td><td>6</td><td>Coprocessor 2 implemented:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>No coprocessor 2 implemented</td></tr>
<tr><td>1</td><td>Coprocessor 2 implements</td></tr>
</table>
This bit indicates not only that the processor contains support for Coprocessor 2, but that such a coprocessor is attached.</td><td></td><td></td><td></td></tr>
<tr><td>MD</td><td>5</td><td>Used to denote MDMX ASE implemented on a MIPS64 processor. Not used on a MIPS32 processor.
This bit indicates not only that the processor contains support for MDMX, but that such a processing element is attached.</td><td>R</td><td>0</td><td>Required</td></tr>
</table>

Table 8.33 Config1 Register Field Descriptions

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read / Write</th><th rowspan="2">Reset State</th><th rowspan="2">Compliance</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>PC</td><td>4</td><td>Performance Counter registers implemented:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>No performance counter registers implemented</td></tr>
<tr><td>1</td><td>Performance counter registers implemented</td></tr>
</table></td><td>R</td><td>Preset</td><td>Required</td></tr>
<tr><td>WR</td><td>3</td><td>Watch registers implemented:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>No watch registers implemented</td></tr>
<tr><td>1</td><td>Watch registers implemented</td></tr>
</table></td><td>R</td><td>Preset</td><td>Required</td></tr>
<tr><td>CA</td><td>2</td><td>Code compression (MIPS16e) implemented:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>MIPS16e not implemented</td></tr>
<tr><td>1</td><td>MIPS16e implemented</td></tr>
</table></td><td>R</td><td>Preset</td><td>Required</td></tr>
<tr><td>EP</td><td>1</td><td>EJTAG implemented:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>No EJTAG implemented</td></tr>
<tr><td>1</td><td>EJTAG implemented</td></tr>
</table></td><td>R</td><td>Preset</td><td>Required</td></tr>
<tr><td>FP</td><td>0</td><td>FPU implemented:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>No FPU implemented</td></tr>
<tr><td>1</td><td>FPU implemented</td></tr>
</table>
This bit indicates not only that the processor contains support for a floating point unit, but that such a unit is attached.
If an FPU is implemented, the capabilities of the FPU can be read from the capability bits in the <i>FIR</i> CP1 register.</td><td>R</td><td>Preset</td><td>Required</td></tr>
</table>

8.29 Configuration Register 2 (CP0 Register 16, Select 2)

Compliance Level: *Required* if a level 2 or level 3 cache is implemented, or if the *Config3* register is required; *Optional* otherwise.

The *Config2* register encodes level 2 and level 3 cache configurations.

Figure 8-26 shows the format of the *Config2* register; Table 8.34 describes the *Config2* register fields.

Figure 8-26 Config2 Register Format

31	30 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
M	TU	TS	TL	TA	SU	SS	SL	SA

Table 8.34 Config2 Register Field Descriptions

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read / Write</th><th rowspan="2">Reset State</th><th rowspan="2">Compliance</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>M</td><td>31</td><td>This bit is reserved to indicate that a <i>Config3</i> register is present. If the <i>Config3</i> register is not implemented, this bit should read as a 0. If the <i>Config3</i> register is implemented, this bit should read as a 1.</td><td>R</td><td>Preset</td><td>Required</td></tr>
<tr><td>TU</td><td>30:28</td><td>Implementation-specific tertiary cache control or status bits. If this field is not implemented it should read as zero and be ignored on write.</td><td>R/W</td><td>Preset</td><td>Optional</td></tr>
<tr><td>TS</td><td>27:24</td><td>Tertiary cache sets per way:
<table>
<tr><th>Encoding</th><th>Sets Per Way</th></tr>
<tr><td>0</td><td>64</td></tr>
<tr><td>1</td><td>128</td></tr>
<tr><td>2</td><td>256</td></tr>
<tr><td>3</td><td>512</td></tr>
<tr><td>4</td><td>1024</td></tr>
<tr><td>5</td><td>2048</td></tr>
<tr><td>6</td><td>4096</td></tr>
<tr><td>7</td><td>8192</td></tr>
<tr><td>8-15</td><td>Reserved</td></tr>
</table></td><td>R</td><td>Preset</td><td>Required</td></tr>
</table>

Table 8.34 Config2 Register Field Descriptions

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read / Write</th><th rowspan="2">Reset State</th><th rowspan="2">Compliance</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>TL</td><td>23:20</td><td>Tertiary cache line size:
<table>
<tr><th>Encoding</th><th>Line Size</th></tr>
<tr><td>0</td><td>No cache present</td></tr>
<tr><td>1</td><td>4</td></tr>
<tr><td>2</td><td>8</td></tr>
<tr><td>3</td><td>16</td></tr>
<tr><td>4</td><td>32</td></tr>
<tr><td>5</td><td>64</td></tr>
<tr><td>6</td><td>128</td></tr>
<tr><td>7</td><td>256</td></tr>
<tr><td>8-15</td><td>Reserved</td></tr>
</table></td><td>R</td><td>Preset</td><td>Required</td></tr>
<tr><td>TA</td><td>19:16</td><td>Tertiary cache associativity:
<table>
<tr><th>Encoding</th><th>Associativity</th></tr>
<tr><td>0</td><td>Direct Mapped</td></tr>
<tr><td>1</td><td>2</td></tr>
<tr><td>2</td><td>3</td></tr>
<tr><td>3</td><td>4</td></tr>
<tr><td>4</td><td>5</td></tr>
<tr><td>5</td><td>6</td></tr>
<tr><td>6</td><td>7</td></tr>
<tr><td>7</td><td>8</td></tr>
<tr><td>8-15</td><td>Reserved</td></tr>
</table></td><td>R</td><td>Preset</td><td>Required</td></tr>
<tr><td>SU</td><td>15:12</td><td>Implementation-specific secondary cache control or status bits. If this field is not implemented it should read as zero and be ignored on write.</td><td>R/W</td><td>Preset</td><td>Optional</td></tr>
</table>

Table 8.34 Config2 Register Field Descriptions

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read / Write</th><th rowspan="2">Reset State</th><th rowspan="2">Compliance</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>SS</td><td>11:8</td><td>Secondary cache sets per way:
<table>
<tr><th>Encoding</th><th>Sets Per Way</th></tr>
<tr><td>0</td><td>64</td></tr>
<tr><td>1</td><td>128</td></tr>
<tr><td>2</td><td>256</td></tr>
<tr><td>3</td><td>512</td></tr>
<tr><td>4</td><td>1024</td></tr>
<tr><td>5</td><td>2048</td></tr>
<tr><td>6</td><td>4096</td></tr>
<tr><td>7</td><td>8192</td></tr>
<tr><td>8-15</td><td>Reserved</td></tr>
</table></td><td>R</td><td>Preset</td><td>Required</td></tr>
<tr><td>SL</td><td>7:4</td><td>Secondary cache line size:
<table>
<tr><th>Encoding</th><th>Line Size</th></tr>
<tr><td>0</td><td>No cache present</td></tr>
<tr><td>1</td><td>4</td></tr>
<tr><td>2</td><td>8</td></tr>
<tr><td>3</td><td>16</td></tr>
<tr><td>4</td><td>32</td></tr>
<tr><td>5</td><td>64</td></tr>
<tr><td>6</td><td>128</td></tr>
<tr><td>7</td><td>256</td></tr>
<tr><td>8-15</td><td>Reserved</td></tr>
</table></td><td>R</td><td>Preset</td><td>Required</td></tr>
<tr><td>SA</td><td>3:0</td><td>Secondary cache associativity:
<table>
<tr><th>Encoding</th><th>Associativity</th></tr>
<tr><td>0</td><td>Direct Mapped</td></tr>
<tr><td>1</td><td>2</td></tr>
<tr><td>2</td><td>3</td></tr>
<tr><td>3</td><td>4</td></tr>
<tr><td>4</td><td>5</td></tr>
<tr><td>5</td><td>6</td></tr>
<tr><td>6</td><td>7</td></tr>
<tr><td>7</td><td>8</td></tr>
<tr><td>8-15</td><td>Reserved</td></tr>
</table></td><td>R</td><td>Preset</td><td>Required</td></tr>
</table>

8.30 Configuration Register 3 (CP0 Register 16, Select 3)

Compliance Level: *Required* if any optional feature described by this register is implemented: Release 2 of the Architecture, the SmartMIPS™ ASE, or trace logic; *Optional* otherwise.

The *Config3* register encodes additional capabilities. All fields in the *Config3* register are read-only.

Figure 8-27 shows the format of the *Config3* register; Table 8.35 describes the *Config3* register fields.

Figure 8-27 Config3 Register Format

31	30 14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
M	0 000 0000 0000 0000 00	ULRI	0	DSP2P	DSPP	0	ITL	LPA	VEIC	VInt	SP	0	MT	SM	TL

Table 8.35 Config3 Register Field Descriptions

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read / Write</th><th rowspan="2">Reset State</th><th rowspan="2">Compliance</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>M</td><td>31</td><td>This bit is reserved to indicate that a <i>Config4</i> register is present. With the current architectural definition, this bit should always read as a 0.</td><td>R</td><td>Preset</td><td>Required</td></tr>
<tr><td>0</td><td>30:14, 12, 9, 3</td><td>Must be written as zeros; returns zeros on read</td><td>0</td><td>0</td><td>Reserved</td></tr>
<tr><td>ULRI</td><td>13</td><td><i>UserLocal</i> register implemented. This bit indicates whether the <i>UserLocal</i> coprocessor 0 register is implemented.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td><i>UserLocal</i> register is not implemented</td></tr>
<tr><td>1</td><td><i>UserLocal</i> register is implemented</td></tr>
</table></td><td>R</td><td>Preset</td><td>Required</td></tr>
<tr><td>DSP2P</td><td>11</td><td>MIPS® DSP ASE Revision 2 implemented. This bit indicates whether Revision 2 of the MIPS DSP ASE is implemented.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Revision 2 of the MIPS DSP ASE is not implemented</td></tr>
<tr><td>1</td><td>Revision 2 of the MIPS DSP ASE is implemented</td></tr>
</table></td><td>R</td><td>Preset</td><td>Required</td></tr>
</table>

Table 8.35 Config3 Register Field Descriptions

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read / Write</th><th rowspan="2">Reset State</th><th rowspan="2">Compliance</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>DSPP</td><td>10</td><td>MIPS® DSP ASE implemented. This bit indicates whether the MIPS DSP ASE is implemented.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>MIPS DSP ASE is not implemented</td></tr>
<tr><td>1</td><td>MIPS DSP ASE is implemented</td></tr>
</table></td><td>R</td><td>Preset</td><td>Required</td></tr>
<tr><td>ITL</td><td>8</td><td>MIPS® IFlowTrace^{TM} mechanism implemented. This bit indicates whether the MIPS IFlowTrace is implemented.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>MIPS IFlowTrace is not implemented</td></tr>
<tr><td>1</td><td>MIPS IFlowTrace is implemented</td></tr>
</table></td><td>R</td><td>Preset</td><td>Required (Release 2.1 Only)</td></tr>
<tr><td>LPA</td><td>7</td><td>Denotes the presence of support for large physical addresses on MIPS64 processors. Not used by MIPS32 processors and returns zero on read.
For implementations of Release 1 of the Architecture, this bit returns zero on read.</td><td>R</td><td>Preset</td><td>Required (Release 2 Only)</td></tr>
<tr><td>VEIC</td><td>6</td><td>Support for an external interrupt controller is implemented.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Support for EIC interrupt mode is not implemented</td></tr>
<tr><td>1</td><td>Support for EIC interrupt mode is implemented</td></tr>
</table>
For implementations of Release 1 of the Architecture, this bit returns zero on read.
This bit indicates not only that the processor contains support for an external interrupt controller, but that such a controller is attached.</td><td>R</td><td>Preset</td><td>Required (Release 2 Only)</td></tr>
<tr><td>VInt</td><td>5</td><td>Vectored interrupts implemented. This bit indicates whether vectored interrupts are implemented.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Vector interrupts are not implemented</td></tr>
<tr><td>1</td><td>Vectored interrupts are implemented</td></tr>
</table>
For implementations of Release 1 of the Architecture, this bit returns zero on read.</td><td>R</td><td>Preset</td><td>Required (Release 2 Only)</td></tr>
</table>

Table 8.35 Config3 Register Field Descriptions

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read / Write</th><th rowspan="2">Reset State</th><th rowspan="2">Compliance</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>SP</td><td>4</td><td>Small (1KByte) page support is implemented, and the *PageGrain* register exists
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Small page support is not implemented</td></tr>
<tr><td>1</td><td>Small page support is implemented</td></tr>
</table>
For implementations of Release 1 of the Architecture, this bit returns zero on read.</td><td>R</td><td>Preset</td><td>Required (Release 2 Only)</td></tr>
<tr><td>MT</td><td>2</td><td>MIPS® MT ASE implemented. This bit indicates whether the MIPS MT ASE is implemented.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>MIPS MT ASE is not implemented</td></tr>
<tr><td>1</td><td>MIPS MT ASE is implemented</td></tr>
</table></td><td>R</td><td>Preset</td><td>Required</td></tr>
<tr><td>SM</td><td>1</td><td>SmartMIPS™ ASE implemented. This bit indicates whether the SmartMIPS ASE is implemented.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>SmartMIPS ASE is not implemented</td></tr>
<tr><td>1</td><td>SmartMIPS ASE is implemented</td></tr>
</table></td><td>R</td><td>Preset</td><td>Required</td></tr>
<tr><td>TL</td><td>0</td><td>Trace Logic implemented. This bit indicates whether PC or data trace is implemented.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Trace logic is not implemented</td></tr>
<tr><td>1</td><td>Trace logic is implemented</td></tr>
</table></td><td>R</td><td>Preset</td><td>Required</td></tr>
</table>

8.31 Reserved for Implementations (CP0 Register 16, Selects 6 and 7)

Compliance Level: *Implementation Dependent.*

CP0 register 16, Selects 6 and 7 are reserved for implementation dependent use and is not defined by the architecture. In order to use CP0 register 16, Selects 6 and 7, it is not necessary to implement CP0 register 16, Selects 2 through 5 only to set the M bit in each of these registers. That is, if the *Config2* and *Config3* registers are not needed for the implementation, they need not be implemented just to provide the M bits.

The architecture only defines the use of the M bits for presence detection of Selects 1 to 5.

8.32 Load Linked Address (CP0 Register 17, Select 0)

Compliance Level: *Optional*.

The *LLAddr* register contains relevant bits of the physical address read by the most recent Load Linked instruction. This register is implementation dependent and for diagnostic purposes only and serves no function during normal operation.

Figure 8-28 shows the format of the *LLAddr* register; Table 8.36 describes the *LLAddr* register fields.

Figure 8-28 LLAddr Register Format

31	0
PAddr	

Table 8.36 LLAddr Register Field Descriptions

Fields					
Name	**Bits**	**Description**	**Read / Write**	**Reset State**	**Compliance**
PAddr	31..0	This field encodes the physical address read by the most recent Load Linked instruction. The format of this register is implementation dependent, and an implementation may implement as many of the bits or format the address in any way that it finds convenient.	R	Undefined	Optional

8.33 WatchLo Register (CP0 Register 18)

Compliance Level: *Optional*.

The *WatchLo* and *WatchHi* registers together provide the interface to a watchpoint debug facility which initiates a watch exception if an instruction or data access matches the address specified in the registers. As such, they duplicate some functions of the EJTAG debug solution. Watch exceptions are taken only if the *EXL* and *ERL* bits are zero in the *Status* register. If either bit is a one, the *WP* bit is set in the *Cause* register, and the watch exception is deferred until both the *EXL* and *ERL* bits are zero.

An implementation may provide zero or more pairs of *WatchLo* and *WatchHi* registers, referencing them via the select field of the MTC0/MFC0 instructions, and each pair of Watch registers may be dedicated to a particular type of reference (e.g., instruction or data). Software may determine if at least one pair of *WatchLo* and *WatchHi* registers are implemented via the *WR* bit of the *Config1* register. See the discussion of the *M* bit in the *WatchHi* register description below.

The *WatchLo* register specifies the base virtual address and the type of reference (instruction fetch, load, store) to match. If a particular Watch register only supports a subset of the reference types, the unimplemented enables must be ignored on write and return zero on read. Software may determine which enables are supported by a particular Watch register pair by setting all three enables bits and reading them back to see which ones were actually set.

It is implementation dependent whether a data watch is triggered by a prefetch, CACHE, or SYNCI (Release 2 only) instruction whose address matches the Watch register address match conditions.

Figure 8-29 shows the format of the *WatchLo* register; Table 8.37 describes the *WatchLo* register fields.

Figure 8-29 WatchLo Register Format

31 3	2	1	0
VAddr	I	R	W

Table 8.37 WatchLo Register Field Descriptions

Fields					
Name	Bits	Description	Read / Write	Reset State	Compliance
VAddr	31..3	This field specifies the virtual address to match. Note that this is a doubleword address, since bits [2:0] are used to control the type of match.	R/W	Undefined	Required
I	2	If this bit is one, watch exceptions are enabled for instruction fetches that match the address and are actually issued by the processor (speculative instructions never cause Watch exceptions). If this bit is not implemented, writes to it must be ignored, and reads must return zero.	R/W	0	Optional

Table 8.37 WatchLo Register Field Descriptions

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
R	1	If this bit is one, watch exceptions are enabled for loads that match the address. For the purposes of the MIPS16e PC-relative load instructions, the PC-relative reference is considered to be a data, rather than an instruction reference. That is, the watchpoint is triggered only if this bit is a 1. If this bit is not implemented, writes to it must be ignored, and reads must return zero.	R/W	0	Optional
W	0	If this bit is one, watch exceptions are enabled for stores that match the address. If this bit is not implemented, writes to it must be ignored, and reads must return zero.	R/W	0	Optional

8.34 WatchHi Register (CP0 Register 19)

Compliance Level: *Optional*.

The *WatchLo* and *WatchHi* registers together provide the interface to a watchpoint debug facility which initiates a watch exception if an instruction or data access matches the address specified in the registers. As such, they duplicate some functions of the EJTAG debug solution. Watch exceptions are taken only if the *EXL* and *ERL* bits are zero in the *Status* register. If either bit is a one, the *WP* bit is set in the *Cause* register, and the watch exception is deferred until both the *EXL* and *ERL* bits are zero.

An implementation may provide zero or more pairs of *WatchLo* and *WatchHi* registers, referencing them via the select field of the MTC0/MFC0 instructions, and each pair of Watch registers may be dedicated to a particular type of reference (e.g., instruction or data). Software may determine if at least one pair of *WatchLo* and *WatchHi* registers are implemented via the *WR* bit of the *Config1* register. If the *M* bit is one in the *WatchHi* register reference with a select field of '*n*', another *WatchHi*/*WatchLo* pair is implemented with a select field of '*n+1*'.

The *WatchHi* register contains information that qualifies the virtual address specified in the *WatchLo* register: an *ASID*, a *G*(lobal) bit, an optional address mask, and three bits (*I*, *R*, and *W*) which denote the condition that caused the watch register to match. If the *G* bit is one, any virtual address reference that matches the specified address will cause a watch exception. If the *G* bit is a zero, only those virtual address references for which the *ASID* value in the *WatchHi* register matches the *ASID* value in the *EntryHi* register cause a watch exception. The optional mask field provides address masking to qualify the address specified in *WatchLo*.

The *I*, *R*, and *W* bits are set by the processor when the corresponding watch register condition is satisfied and indicate which watch register pair (if more than one is implemented) and which condition matched. When set by the processor, each of these bits remain set until cleared by software. All three bits are "write one to clear", such that software must write a one to the bit in order to clear its value. The typical way to do this is to write the value read from the *WatchHi* register back to *WatchHi*. In doing so, only those bits which were set when the register was read are cleared when the register is written back.

Figure 8-30 shows the format of the *WatchHi* register; Table 8.38 describes the *WatchHi* register fields.

Figure 8-30 WatchHi Register Format

31	30	29 24	23 16	15 12	11 3	2	1	0
M	G	0	ASID	0	Mask	I	R	W

Table 8.38 WatchHi Register Field Descriptions

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
M	31	If this bit is one, another pair of *WatchHi*/*WatchLo* registers is implemented at a MTC0 or MFC0 select field value of '*n+1*'	R	Preset	Required

Table 8.38 WatchHi Register Field Descriptions

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
G	30	If this bit is one, any address that matches that specified in the *WatchLo* register will cause a watch exception. If this bit is zero, the *ASID* field of the *WatchHi* register must match the *ASID* field of the *EntryHi* register to cause a watch exception.	R/W	Undefined	Required
ASID	23..16	*ASID* value which is required to match that in the *EntryHi* register if the *G* bit is zero in the *WatchHi* register.	R/W	Undefined	Required
Mask	11..3	Optional bit mask that qualifies the address in the *WatchLo* register. If this field is implemented, any bit in this field that is a one inhibits the corresponding address bit from participating in the address match. If this field is not implemented, writes to it must be ignored, and reads must return zero. Software may determine how many mask bits are implemented by writing ones the this field and then reading back the result.	R/W	Undefined	Optional
I	2	This bit is set by hardware when an instruction fetch condition matches the values in this watch register pair. When set, the bit remains set until cleared by software, which is accomplished by writing a 1 to the bit.	W1C	Undefined	Required (Release 2)
R	1	This bit is set by hardware when a load condition matches the values in this watch register pair. When set, the bit remains set until cleared by software, which is accomplished by writing a 1 to the bit.	W1C	Undefined	Required (Release 2)
W	0	This bit is set by hardware when a store condition matches the values in this watch register pair. When set, the bit remains set until cleared by software, which is accomplished by writing a 1 to the bit.	W1C	Undefined	Required (Release 2)
0	29..24, 15..12	Must be written as zero; returns zero on read.	0	0	Reserved

8.35 Reserved for Implementations (CP0 Register 22, all Select values)

Compliance Level: *Implementation Dependent*.

CP0 register 22 is reserved for implementation dependent use and is not defined by the architecture.

8.36 Debug Register (CP0 Register 23)

Compliance Level: *Optional*.

The *Debug* register is part of the EJTAG specification. Refer to that specification for the format and description of this register.

8.37 DEPC Register (CP0 Register 24)

Compliance Level: *Optional*.

The *DEPC* register is a read-write register that contains the address at which processing resumes after a debug exception has been serviced. It is part of the EJTAG specification and the reader is referred there for the format and description of the register. All bits of the *DEPC* register are significant and must be writable.

When a debug exception occurs, the processor writes the *DEPC* register with,

- 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 exception causing instruction is in a branch delay slot, and the *Branch Delay* bit in the *Cause* register is set.

The processor reads the *DEPC* register as the result of execution of the DERET instruction.

Software may write the *DEPC* register to change the processor resume address and read the *DEPC* register to determine at what address the processor will resume.

8.37.1 Special Handling of the DEPC Register in Processors That Implement the MIPS16e ASE

In processors that implement the MIPS16e ASE, the *DEPC* register requires special handling.

When the processor writes the *DEPC* register, it combines the address at which processing resumes with the value of the *ISA Mode* register:

```
$\text{DEPC} \leftarrow \text{resumePC}_{31..1} \,\|\, \text{ISAMode}_0$
```

“resumePC” is the address at which processing resumes, as described above.

When the processor reads the *DEPC* register, it distributes the bits to the *PC* and *ISA Mode* registers:

```
$\text{PC} \leftarrow \text{DEPC}_{31..1} \,\|\, 0$
$\text{ISAMode} \leftarrow \text{DEPC}_0$
```

Software reads of the *DEPC* register simply return to a GPR the last value written with no interpretation. Software writes to the *DEPC* register store a new value which is interpreted by the processor as described above.

8.38 Performance Counter Register (CP0 Register 25)

Compliance Level: *Recommended*.

The MIPS32 Architecture supports implementation dependent performance counters that provide the capability to count events or cycles for use in performance analysis. If performance counters are implemented, each performance counter consists of a pair of registers: a 32-bit control register and a 32-bit counter register. To provide additional capability, multiple performance counters may be implemented.

Performance counters can be configured to count implementation dependent events or cycles under a specified set of conditions that are determined by the control register for the performance counter. The counter register increments once for each enabled event. When the most significant bit of the counter register is a one (the counter overflows), the performance counter optionally requests an interrupt. In implementations of Release 1 of the Architecture, this interrupt is combined in a implementation-dependent way with hardware interrupt 5. In Release 2 of the Architecture, pending interrupts from all performance counters are ORed together to become the *PCI* bit in the *Cause* register, and are prioritized as appropriate to the interrupt mode of the processor. Counting continues after a counter register overflow whether or not an interrupt is requested or taken.

Each performance counter is mapped into even-odd select values of the *PerfCnt* register: Even selects access the control register and odd selects access the counter register. Table 8.39 shows an example of two performance counters and how they map into the select values of the *PerfCnt* register.

Table 8.39 Example Performance Counter Usage of the PerfCnt CP0 Register

Performance Counter	PerfCnt Register Select Value	PerfCnt Register Usage
0	PerfCnt, Select 0	Control Register 0
	PerfCnt, Select 1	Counter Register 0
1	PerfCnt, Select 2	Control Register 1
	PerfCnt, Select 3	Counter Register 1

More or less than two performance counters are also possible, extending the select field in the obvious way to obtain the desired number of performance counters. Software may determine if at least one pair of Performance Counter Control and Counter registers is implemented via the *PC* bit in the *Config1* register. If the *M* bit is one in the Performance Counter Control register referenced via a select field of ‘*n*’, another pair of Performance Counter Control and Counter registers is implemented at the select values of ‘*n*+2’ and ‘*n*+3’.

The Control Register associated with each performance counter controls the behavior of the performance counter. Figure 8-31 shows the format of the Performance Counter Control Register; Table 8.40 describes the Performance Counter Control Register fields.

Figure 8-31 Performance Counter Control Register Format

31	30	29 25	24 16	15	14 11	10 5	4	3	2	1	0
M	W	Impl	0	PCTD	EventExt	Event	IE	U	S	K	EXL

Table 8.40 Performance Counter Control Register Field Descriptions

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read / Write</th><th rowspan="2">Reset State</th><th rowspan="2">Compliance</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>M</td><td>31</td><td>If this bit is a one, another pair of Performance Counter Control and Counter registers is implemented at a MTC0 or MFC0 select field value of ‘<i>n</i>+2’ and ‘<i>n</i>+3’.</td><td>R</td><td>Preset</td><td>Required</td></tr>
<tr><td>W</td><td>30</td><td>Denotes that the corresponding Counter register is 64 bits wide on a MIPS64 processor. Unused on a MIPS32 processor.</td><td>R</td><td>Preset</td><td>Required</td></tr>
<tr><td>Impl</td><td>29:25</td><td>This field is implementation dependent and is not specified by the architecture.
If not used by the implementation, must be written as zero; returns zero on read.</td><td></td><td>Undefined
0 if not used by the implementation</td><td>Optional</td></tr>
<tr><td>0</td><td>24..16</td><td>Must be written as zero; returns zero on read</td><td>0</td><td>0</td><td>Reserved</td></tr>
<tr><td>PCTD</td><td>15</td><td>Performance Counter Trace Disable.
The PDTrace facility (revision 6.00 and higher) has the ability to trace Performance Counter in its output. This bit is used to disable the specified performance counter from being traced when performance counter trace is enabled and a performance counter trace event is triggered.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Tracing is enabled for this counter.</td></tr>
<tr><td>1</td><td>Tracing is disabled for this counter.</td></tr>
</table></td><td>RW</td><td>0</td><td>Required if PDTrace Performance Counter Tracing feature is implemented.</td></tr>
<tr><td>EventExt</td><td>14..11</td><td>In some implementations which support more than the the 64 encodings possible in the 6-bit Event field, the EventExt field acts as an extension to the Event field. In such instances the event selection is the concatentation of the two fields, i.e., EventExt|Event.
The actual field width is implementation dependent. Any bits that are not implemented read as zero and are ignored on write.</td><td>RW</td><td>Undefined</td><td>Optional</td></tr>
<tr><td>Event</td><td>10..5</td><td>Selects the event to be counted by the corresponding Counter Register. The list of events is implementation dependent, but typical events include cycles, instructions, memory reference instructions, branch instructions, cache and TLB misses, etc.
Implementations that support multiple performance counters allow ratios of events, e.g., cache miss ratios if cache miss and memory references are selected as the events in two counters</td><td>R/W</td><td>Undefined</td><td>Required</td></tr>
</table>

Table 8.40 Performance Counter Control Register Field Descriptions

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read / Write</th><th rowspan="2">Reset State</th><th rowspan="2">Compliance</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>IE</td><td>4</td><td>Interrupt Enable. Enables the interrupt request when the corresponding counter overflows (the most significant bit of the counter is one. This is bit 31 for a 32-bit wide counter or bit 63 of a 64-bit wide counter, denoted by the W bit in this register).
Note that this bit simply enables the interrupt request. The actual interrupt is still gated by the normal interrupt masks and enable in the <i>Status</i> register.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Performance counter interrupt disabled</td></tr>
<tr><td>1</td><td>Performance counter interrupt enabled</td></tr>
</table></td><td>R/W</td><td>0</td><td>Required</td></tr>
<tr><td>U</td><td>3</td><td>Enables event counting in User Mode. Refer to Section 3.4 “User Mode” on page 18 for the conditions under which the processor is operating in User Mode.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Disable event counting in User Mode</td></tr>
<tr><td>1</td><td>Enable event counting in User Mode</td></tr>
</table></td><td>R/W</td><td>Undefined</td><td>Required</td></tr>
<tr><td>S</td><td>2</td><td>Enables event counting in Supervisor Mode (for those processors that implement Supervisor Mode). Refer to Section 3.3 “Supervisor Mode” on page 17 for the conditions under which the processor is operating in Supervisor mode.
If the processor does not implement Supervisor Mode, this bit must be ignored on write and return zero on read.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Disable event counting in Supervisor Mode</td></tr>
<tr><td>1</td><td>Enable event counting in Supervisor Mode</td></tr>
</table></td><td>R/W</td><td>Undefined</td><td>Required</td></tr>
<tr><td>K</td><td>1</td><td>Enables event counting in Kernel Mode. Unlike the usual definition of Kernel Mode as described in Section 3.2 “Kernel Mode” on page 17, this bit enables event counting only when the EXL and ERL bits in the <i>Status</i> register are zero.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Disable event counting in Kernel Mode</td></tr>
<tr><td>1</td><td>Enable event counting in Kernel Mode</td></tr>
</table></td><td>R/W</td><td>Undefined</td><td>Required</td></tr>
</table>

Table 8.40 Performance Counter Control Register Field Descriptions

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read / Write</th><th rowspan="2">Reset State</th><th rowspan="2">Compliance</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>EXL</td><td>0</td><td>Enables event counting when the EXL bit in the <i>Status</i> register is one and the ERL bit in the <i>Status</i> register is zero.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Disable event counting while EXL = 1, ERL = 0</td></tr>
<tr><td>1</td><td>Enable event counting while EXL = 1, ERL = 0</td></tr>
</table>
Counting is never enabled when the ERL bit in the <i>Status</i> register or the DM bit in the <i>Debug</i> register is one.</td><td>R/W</td><td>Undefined</td><td>Required</td></tr>
</table>

The Counter Register associated with each performance counter increments once for each enabled event. Figure 8-32 shows the format of the Performance Counter Counter Register; Table 8.41 describes the Performance Counter Counter Register fields.

Figure 8-32 Performance Counter Counter Register Format

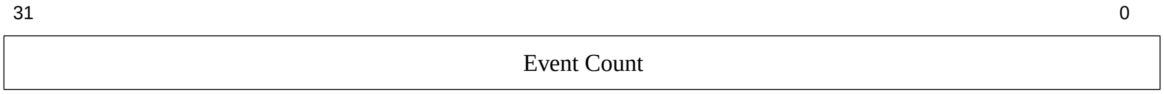


Table 8.41 Performance Counter Counter Register Field Descriptions

Fields		Description	Read/ Write	Reset State	Compliance
Name	Bits				
Event Count	31..0	Increments once for each event that is enabled by the corresponding Control Register. When the most significant bit is one, a pending interrupt request is ORed with those from other performance counters and indicated by the PCI bit in the *Cause* register.	R/W	Undefined	Required

Programming Note:

In Release 2 of the Architecture, the EHB instruction can be used to make interrupt state changes visible when the IE field of the Control register or the Event Count Field of the Counter register are written. See sECTION 5.1.2.1 “Software Hazards and the Interrupt System” on page 42.

8.39 ErrCtl Register (CP0 Register 26, Select 0)

Compliance Level: *Optional*.

The *ErrCtl* register provides an implementation dependent diagnostic interface with the error detection mechanisms implemented by the processor. This register has been used in previous implementations to read and write parity or ECC information to and from the primary or secondary cache data arrays in conjunction with specific encodings of the Cache instruction or other implementation-dependent method. The exact format of the *ErrCtl* register is implementation dependent and not specified by the architecture. Refer to the processor specification for the format of this register and a description of the fields.

8.40 CacheErr Register (CP0 Register 27, Select 0)

Compliance Level: *Optional.*

The *CacheErr* register provides an interface with the cache error detection logic that may be implemented by a processor.

The exact format of the *CacheErr* register is implementation dependent and not specified by the architecture. Refer to the processor specification for the format of this register and a description of the fields.

8.41 TagLo Register (CP0 Register 28, Select 0, 2)

Compliance Level: *Required* if a cache is implemented; *Optional* otherwise.

The *TagLo* and *TagHi* registers are read/write registers that act as the interface to the cache tag array. The Index Store Tag and Index Load Tag operations of the CACHE instruction use the *TagLo* and *TagHi* registers as the source or sink of tag information, respectively.

The exact format of the *TagLo* and *TagHi* registers is implementation dependent. Refer to the processor specification for the format of this register and a description of the fields.

However, software must be able to write zeros into the *TagLo* and *TagHi* registers and then use the Index Store Tag cache operation to initialize the cache tags to a valid state at powerup.

It is implementation dependent whether there is a single *TagLo* register that acts as the interface to all caches, or a dedicated *TagLo* register for each cache. If multiple *TagLo* registers are implemented, they occupy the even select values for this register encoding, with select 0 addressing the instruction cache and select 2 addressing the data cache. Whether individual *TagLo* registers are implemented or not for each cache, processors must accept a write of zero to select 0 and select 2 of *TagLo* as part of the software process of initializing the cache tags at powerup.

8.42 DataLo Register (CP0 Register 28, Select 1, 3)

Compliance Level: *Optional*.

The *DataLo* and *DataHi* registers are registers that act as the interface to the cache data array and are intended for diagnostic operation only. The Index Load Tag operation of the CACHE instruction reads the corresponding data values into the *DataLo* and *DataHi* registers.

The exact format and operation of the *DataLo* and *DataHi* registers is implementation dependent. Refer to the processor specification for the format of this register and a description of the fields.

It is implementation dependent whether there is a single *DataLo* register that acts as the interface to all caches, or a dedicated *DataLo* register for each cache. If multiple *DataLo* registers are implemented, they occupy the odd select values for this register encoding, with select 1 addressing the instruction cache and select 3 addressing the data cache.

8.43 TagHi Register (CP0 Register 29, Select 0, 2)

Compliance Level: *Required* if a cache is implemented; *Optional* otherwise.

The *TagLo* and *TagHi* registers are read/write registers that act as the interface to the cache tag array. The Index Store Tag and Index Load Tag operations of the CACHE instruction use the *TagLo* and *TagHi* registers as the source or sink of tag information, respectively.

The exact format of the *TagLo* and *TagHi* registers is implementation dependent. Refer to the processor specification for the format of this register and a description of the fields. However, software must be able to write zeros into the *TagLo* and *TagHi* registers and the use the Index Store Tag cache operation to initialize the cache tags to a valid state at powerup.

It is implementation dependent whether there is a single *TagHi* register that acts as the interface to all caches, or a dedicated *TagHi* register for each cache. If multiple *TagHi* registers are implemented, they occupy the even select values for this register encoding, with select 0 addressing the instruction cache and select 2 addressing the data cache. Whether individual *TagHi* registers are implemented or not for each cache, processors must accept a write of zero to select 0 and select 2 of *TagHi* as part of the software process of initializing the cache tags at powerup.

8.44 DataHi Register (CP0 Register 29, Select 1, 3)

Compliance Level: *Optional*.

The *DataLo* and *DataHi* registers are registers that act as the interface to the cache data array and are intended for diagnostic operation only. The Index Load Tag operation of the CACHE instruction reads the corresponding data values into the *DataLo* and *DataHi* registers.

The exact format and operation of the *DataLo* and *DataHi* registers is implementation dependent. Refer to the processor specification for the format of this register and a description of the fields.

8.45 ErrorEPC (CP0 Register 30, Select 0)

Compliance Level: Required.

The *ErrorEPC* register is a read-write register, similar to the *EPC* register, at which processing resumes after a Reset, Soft Reset, Nonmaskable Interrupt (NMI) or Cache Error exceptions (collectively referred to as error exceptions). Unlike the *EPC* register, there is no corresponding branch delay slot indication for the *ErrorEPC* register. All bits of the *ErrorEPC* register are significant and must be writable.

When an error exception occurs, the processor writes the *ErrorEPC* register with:

- 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 error causing instruction is in a branch delay slot.

The processor reads the *ErrorEPC* register as the result of execution of the ERET instruction.

Software may write the *ErrorEPC* register to change the processor resume address and read the *ErrorEPC* register to determine at what address the processor will resume

Figure 8-33 shows the format of the *ErrorEPC* register; Table 8.42 describes the *ErrorEPC* register fields.

Figure 8-33 ErrorEPC Register Format

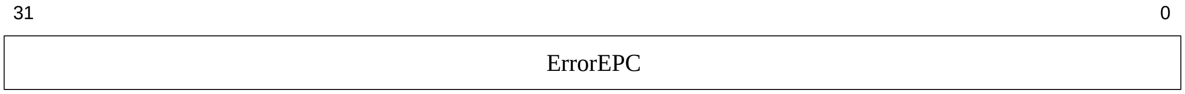


Table 8.42 ErrorEPC Register Field Descriptions

Fields					
Name	Bits	Description	Read / Write	Reset State	Compliance
ErrorEPC	31..0	Error Exception Program Counter	R/W	Undefined	Required

8.45.1 Special Handling of the ErrorEPC Register in Processors That Implement the MIPS16e ASE

In processors that implement the MIPS16e ASE, the *ErrorEPC* register requires special handling.

When the processor writes the *ErrorEPC* register, it combines the address at which processing resumes with the value of the *ISA Mode* register:

$$\texttt{ErrorEPC} \leftarrow \texttt{resumePC}_{31..1} \parallel \texttt{ISAMode}_{0}$$

“resumePC” is the address at which processing resumes, as described above.

When the processor reads the *ErrorEPC* register, it distributes the bits to the *PC* and *ISAMode* registers:

```
PC ← $ErrorEPC_{31..1}$ ‖ 0
ISAMode ← $ErrorEPC_0$
```

Software reads of the *ErrorEPC* register simply return to a GPR the last value written with no interpretation. Software writes to the *ErrorEPC* register store a new value which is interpreted by the processor as described above.

8.46 DESAVE Register (CP0 Register 31)

Compliance Level: *Optional*.

The *DESAVE* register is part of the EJTAG specification. Refer to that specification for the format and description of this register.

Appendix A

Alternative MMU Organizations

The main body of this specification describes the TLB-based MMU organization. This appendix describes other potential MMU organizations.

A.1 Fixed Mapping MMU

As an alternative to the full TLB-based MMU, the MIPS32 Architecture supports a lightweight memory management mechanism with fixed virtual-to-physical address translation, and no memory protection beyond what is provided by the address error checks required of all MMUs. This may be useful for those applications which do not require the capabilities of a full TLB-based MMU.

A.1.1 Fixed Address Translation

Address translation using the Fixed Mapping MMU is done as follows:

- Kseg0 and Kseg1 addresses are translated in an identical manner to the TLB-based MMU: they both map to the low 512MB of physical memory.

- Useg/Suseg/Kuseg addresses are mapped by adding 1GB to the virtual address when the ERL bit is zero in the Status register, and are mapped using an identity mapping when the ERL bit is one in the Status register.

- Sseg/Ksseg/Kseg2/Kseg3 addresses are mapped using an identity mapping.

Supervisor Mode is not supported with a Fixed Mapping MMU.

Table A.1 lists all mappings from virtual to physical addresses. Note that address error checking is still done before the translation process. Therefore, an attempt to reference kseg0 from User Mode still results in an address error exception, just as it does with a TLB-based MMU.

Table A.1 Physical Address Generation from Virtual Addresses

Segment Name	Virtual Address	Generates Physical Address	
		Status$_{ERL}$ = 0	Status$_{ERL}$ = 1
useg suseg kuseg	0x0000 0000 through 0x7FFF FFFF	0x4000 0000 through 0xBFFF FFFF	0x0000 0000 through 0x7FFF FFFF
kseg0	0x8000 0000 through 0x9FFF FFFF	0x0000 0000 through 0x1FFF FFFF	

Table A.1 Physical Address Generation from Virtual Addresses (Continued)

<table>
<tr><th rowspan="2">Segment Name</th><th rowspan="2">Virtual Address</th><th colspan="2">Generates Physical Address</th></tr>
<tr><th>$\text{Status}_{\text{ERL}} = 0$</th><th>$\text{Status}_{\text{ERL}} = 1$</th></tr>
<tr><td>kseg1</td><td>0xA000 0000
through
0xBFFF FFFF</td><td colspan="2">0x0000 0000
through
0x0x1FFF FFFF</td></tr>
<tr><td>sseg
ksseg
kseg2</td><td>0xC000 0000
through
0xDFFF FFFF</td><td colspan="2">0xC000 0000
through
0xDFFF FFFF</td></tr>
<tr><td>kseg3</td><td>0xE000 0000
through
0xFFFF FFFF</td><td colspan="2">0xE000 0000
through
0xFFFF FFFF</td></tr>
</table>

Note that this mapping means that physical addresses `0x2000 0000` through `0x3FFF FFFF` are inaccessible when the ERL bit is off in the *Status* register, and physical addresses `0x8000 0000` through `0xBFFF FFFF` are inaccessible when the ERL bit is on in the *Status* register.

Figure A-1 shows the memory mapping when the ERL bit in the *Status* register is zero; Figure A-2 shows the memory mapping when the ERL bit is one.

Figure A-1 Memory Mapping when ERL = 0

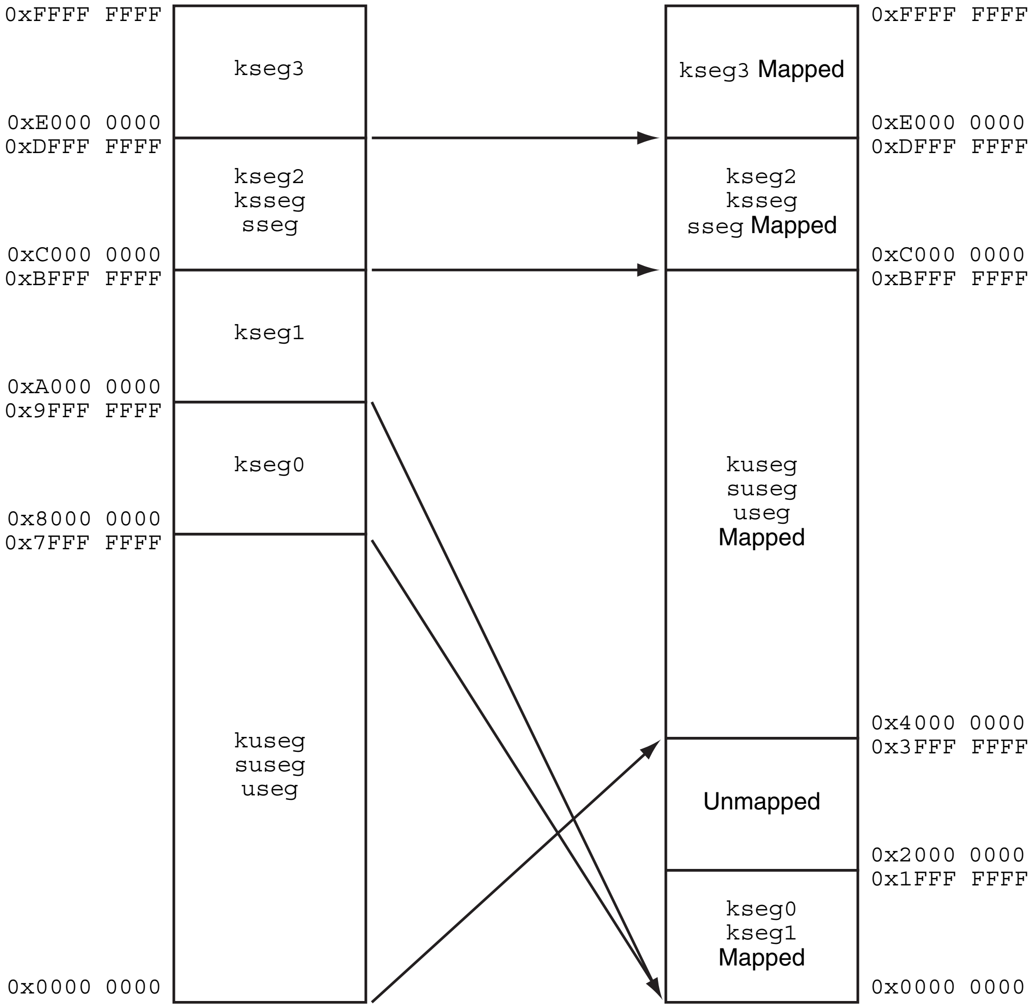
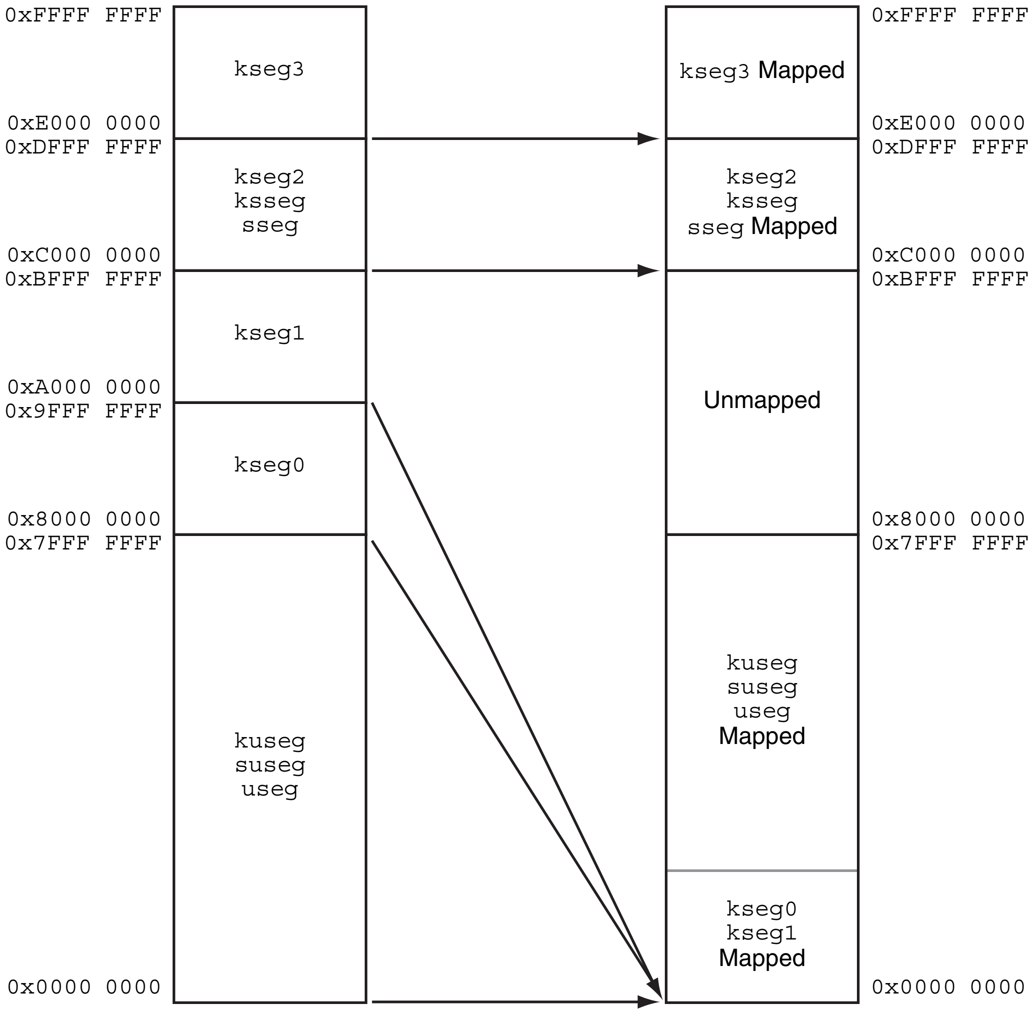


Figure A-2 Memory Mapping when ERL = 1



A.1.2 Cacheability Attributes

Because the TLB provided the cacheability attributes for the kuseg, kseg2, and kseg3 segments, some mechanism is required to replace this capability when the fixed mapping MMU is used. Two additional fields are added to the *Config* register whose encoding is identical to that of the K0 field. These additions are the K23 and KU fields which control the cacheability of the kseg2/kseg3 and the kuseg segments, respectively. Note that when the ERL bit is on in the *Status* register, kuseg data references are always treated as uncacheable references, independent of the value of the KU field. The operation of the processor is **UNDEFINED** if the ERL bit is set while the processor is executing instructions from kuseg.

The cacheability attributes for kseg0 and kseg1 are provided in the same manner as for a TLB-based MMU: the cacheability attribute for kseg0 comes from the K0 field of *Config*, and references to kseg1 are always uncached.

Figure A-3 shows the format of the additions to the *Config* register; Table A.2 describes the new *Config* register fields.

Figure A-3 Config Register Additions

31	30 28	27 25	24 16	15	14 13	12 10	9 7	6 4	3	2 0
M	K23	KU	0	BE	AT	AR	MT	0	VI	K0

Table A.2 Config Register Field Descriptions

Fields		Description	Read/ Write	Reset State	Compliance
Name	Bits				
K23	30:28	Kseg2/Kseg3 cacheability and coherency attribute. See Table 8.8 on page 81 for the encoding of this field.	R/W	Undefined	Required
KU	27:25	Kuseg cacheability and coherency attribute when $Status_{ERL}$ is zero. See Table 8.8 on page 81 for the encoding of this field.	R/W	Undefined	Required

A.1.3 Changes to the CP0 Register Interface

Relative to the TLB-based address translation mechanism, the following changes are necessary to the CP0 register interface:

- The Index, Random, EntryLo0, EntryLo1, Context, PageMask, Wired, and EntryHi registers are no longer required and may be removed. The effects of a read or write to these registers are **UNDEFINED**.

- The TLBWR, TLBWI, TLBP, and TLBR instructions are no longer required and must cause a Reserved Instruction Exception.

A.2 Block Address Translation

This section describes the architecture for a block address translation (BAT) mechanism that reuses much of the hardware and software interface that exists for a TLB-Based virtual address translation mechanism. This mechanism has the following features:

- It preserves as much as possible of the TLB-Based interface, both in hardware and software.

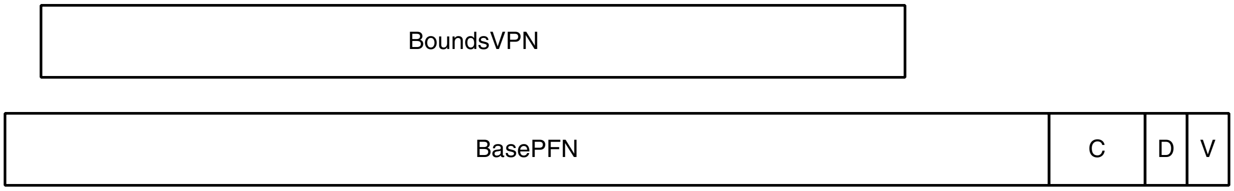
- It provides independent base-and-bounds checking and relocation for instruction references and data references.

- It provides optional support for base-and-bounds relocation of kseg2 and kseg3 virtual address regions.

A.2.1 BAT Organization

The BAT is an indexed structure which is used to translate virtual addresses. It contains pairs of instruction/data entries which provide the base-and-bounds checking and relocation for instruction references and data references, respectively. Each entry contains a page-aligned bounds virtual page number, a base page frame number (whose width is implementation dependent), a cache coherence field (C), a dirty (D) bit, and a valid (V) bit. Figure A-4 shows the logical arrangement of a BAT entry.

Figure A-4 Contents of a BAT Entry



The BAT is indexed by the reference type and the address region to be checked as shown in Table A.3.

Table A.3 BAT Entry Assignments

Entry Index	Reference Type	Address Region
0	Instruction	useg/kuseg
1	Data	
2	Instruction	kseg2 (or kseg2 and kseg3)
3	Data	
4	Instruction	kseg3
5	Data	

Entries 0 and 1 are required. Entries 2, 3, 4 and 5 are optional and may be implemented as necessary to address the needs of the particular implementation. If entries for kseg2 and kseg3 are not implemented, it is implementation-dependent how, if at all, these address regions are translated. One alternative is to combine the mapping for kseg2 and kseg3 into a single pair of instruction/data entries. Software may determine how many BAT entries are implemented by looking at the MMU Size field of the *Config1* register.

A.2.2 Address Translation

When a virtual address translation is requested, the BAT entry that is appropriate to the reference type and address region is read. If the virtual address is greater than the selected bounds address, or if the valid bit is off in the entry, a TLB Invalid exception of the appropriate reference type is initiated. If the reference is a store and the D bit is off in the entry, a TLB Modified exception is initiated. Otherwise, the base PFN from the selected entry, shifted to align with bit 12, is added to the virtual address to form the physical address. The BAT process can be described as follows:

```
i ← SelectIndex (reftype, va)
bounds ← $BAT[i]_{BoundsVPN}$ || $1^{12}$
pfn ← $BAT[i]_{BasePFN}$
c ← $BAT[i]_C$
d ← $BAT[i]_D$
v ← $BAT[i]_V$
if (va > bounds) or (v = 0) then
    InitiateTLBInvalidException(reftype)
endif
if (d = 0) and (reftype = store) then
    InitiateTLBModifiedException()
```

```
endif
pa ← va + (pfn || 0^12)
```

Making all addresses out-of-bounds can only be done by clearing the valid bit in the BAT entry. Setting the bounds value to zero leaves the first virtual page mapped.

A.2.3 Changes to the CP0 Register Interface

Relative to the TLB-based address translation mechanism, the following changes are necessary to the CP0 register interface:

- The *Index* register is used to index the BAT entry to be read or written by the TLBWI and TLBR instructions.
- The *EntryHi* register is the interface to the BoundsVPN field in the BAT entry.
- The *EntryLo0* register is the interface to the BasePFN and C, D, and V fields of the BAT entry. The register has the same format as for a TLB-based MMU.
- The *Random*, *EntryLo1*, *Context*, *PageMask*, and *Wired* registers are eliminated. The effects of a read or write to these registers is **UNDEFINED**.
- The TLBP and TLBWR instructions are unnecessary. The TLBWI and TLBR instructions reference the BAT entry whose index is contained in the *Index* register. The effects of executing a TLBP or TLBWR are **UNDEFINED**, but processors should signal a Reserved Instruction Exception.

Appendix B

Revision History

In the left hand page margins of this document you may find vertical change bars to note the location of significant changes to this document since its last release. Significant changes are defined as those which you should take note of as you use the MIPS IP. Changes to correct grammar, spelling errors or similar may or may not be noted with change bars. Change bars will be removed for changes which are more than one revision old.

Please note: Limitations on the authoring tools make it difficult to place change bars on changes to figures. Change bars on figure titles are used to denote a potential change in the figure itself.

Revision	Date	Description
0.92	January 20, 2001	Internal review copy of reorganized and updated architecture documentation.
0.95	March 12, 2001	Clean up document for external review release
1.00	August 29, 2002	Update based on review feedback: • Change ProbEn to ProbeTrap in the EJTAG Debug entry vector location discussion. • Add cache error and EJTAG Debug exceptions to the list of exceptions that do not go through the general exception processing mechanism. • Fix incorrect branch offset adjustment in general exception processing pseudo code to deal with extended MIPS16e instructions. • Add $\text{Config}_{\text{VI}}$ to denote an instruction cache with both virtual indexing and virtual tags. • Correct XContext register description to note that both BadVPN2 and R fields are UNPREDICTABLE after an address error exception. • Note that Supervisor Mode is not supported with a Fixed Mapping MMU. • Define TagLo bits 4..3 as implementation dependent. • Describe the intended usage model differences between Reset and Soft Reset Exceptions. • Correct the minimum number of TLB entries to be 3, not 2, and show an example of the need for 3. • Modify the description of PageMask and the TLB lookup process to acknowledge the fact that not all implementations may support all page sizes.
1.90	September 1, 2002	Update the specification with the changes introduced in Release 2 of the Architecture. Changes in this revision include: • The following new Coprocessor 0 registers were added: EBase, HWREna, IntCtl, PageGrain, SRSCtl, SRSMap. • The following Coprocessor 0 registers were modified: Cause, Config, Config2, Config3, EntryHi, EntryLo0, EntryLo1, PageMask, PerfCnt, Status, WatchHi, WatchLo. • The descriptions of Virtual memory, exceptions, and hazards have been updated to reflect the changes in Release 2. • A chapter on GPR shadow regsiters has been added. • The chapter on CP0 hazards has been completely rewriten to reflect the Release 2 changes.

Revision	Date	Description
2.00	June 9, 2003	Complete the update to include Release 2 changes. These include: • Make bits 12..11 of the PageMask register power up zero and be gated by 1K page enable. This eliminates the problem of having these bits set to 0b11 on a Release 2 chip in which kernel software has not enabled 1K page support. • Correct the address of the cache error vector when the BEV bit is 1. It should be 0xBFC0.0300,. not 0xBFC0.0200. • Correct the introduction to shadow registers to note that the SRSCtl register is not updated at the end of an exception in which $\text{Status}_{\text{BEV}}$ = 1. • Clarify that a MIPS16e PC-relative load reference is a data reference for the purposes of the Watch registers. • Add note about a hardware interrupt being deasserted between the time that the processor detects the interrupt request and the time that the software interrupt handler runs. Software must be prepared for this case and simply dismiss the interrupt via an ERET. • Add restriction that software must set $\text{EBase}_{15..12}$ to zero in all bit positions less than or equal to the most significant bit in the vector offset. This is only required in certain combinations of vector number and vector spacing when using VI or EIC Interrupt modes. • Add suggested software TLB init routine which reduced the probability of triggering a machine check.
2.50	July 1, 2005	Changes in this revision: • Correct the encoding table description for the $\text{Cause}_{\text{PCI}}$ bit to indicate that the bit controlls the performance counter, not the timer interrupt. • Correct the figure Interrupt Generation for External Interrupt Controller Interrupt Mode to show $\text{Cause}_{\text{IP1..0}}$ going to the EIC, rather than $\text{Status}_{\text{IP1..0}}$ • Update all files to FrameMaker 7.1. • Update reset exception list to reflect missing Release 2 reset requirements. • Define bits 31..30 in the *HWREna* register as access enables for the implementation-dependent hardware registers 31 and 30. • Add definition for Coprocessor 0 Enable to Operating Modes chapter. • Add K23 and KU fields to main Config register definition as a pointer to the Fixed Mapping MMU appendix. • Add specific note about the need to implement all shadow sets between 0 and HSS - no holes are allowed. • Change the hazard from a software write to the $\text{SRSCtl}_{\text{PSS}}$ field and a RDPGPR and WRPGPR and instruction hazard vs. an execution hazard. • Correct the pseudo-code in the cache error exception description to reflect the Release 2 change that introduced EBase. • Document that EHB clears instruction state change hazards for writes to interrupt-related fields in the *Status*, *Cause*, *Compare*, and *PerfCnt* registers. • Note that implementation-dependent bits in the *Status* and *Config* registers should be defined in such a way that standard boot software will run, and that software which preserves the value of the field when writing the registers will also run correctly. • With Release 2 of the Architecture the FR bit in the *Status* register should be a R/W bit, not a R bit. • Improve the organization of the CP0 hazards table, and document that DERET, ERET, and exceptions and interrupts clear all hazards before the instruction fetch at the target instruction. • Add list of MIPS® MT CP0 registers and MIPS MT and MIPS® DSP present bits in the *Config3* register.

Revision	Date	Description
2.60	Jun 25, 2008	Changes in this revision: • Add the *UserLocal* register and access to it via the RDHWR instruction. • Operating Modes - footnote about ksseg/sseg • COP3 no longer usable for customer extensions • EIC Mode allows VectorNum != RIPL • CP0Regs Table - added missing EJTAG & PDTrace Registers • *C0_DataLo/Hi* are actually R/W • Hazards table - added a bunch of missing ones • Various typos fixed.
2.61	August 01, 2008	• In the *Status* register description, the ERL behavior description was incorrect in saying only 29bits of kuseg becomes uncached&unmapped.
2.62	January 2, 2009	• *C0_HWREna* register - CCRes is accessed with register number 3, not 4. • Added *C0_PerfCnt.PCTD* control bit.