MIPS® Architecture For Programmers Vol. III: MIPS32®/microMIPS32™ Privileged Resource Architecture

Document Number: MD00090
Revision 5.05
November 14, 2014

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

Contents

Figures

Tables

Chapter 1

About This Book

The MIPS32®/microMIPS32™ Privileged Resource Architecture comes as part of a multi-volume set.

- Volume I-A describes conventions used throughout the document set, and provides an introduction to the MIPS64® Architecture
- Volume I-B describes conventions used throughout the document set, and provides an introduction to the micro™ Architecture
- Volume II-A provides detailed descriptions of each instruction in the MIPS64® instruction set
- Volume II-B provides detailed descriptions of each instruction in the microMIPS64™ instruction set
- Volume III describes the MIPS32® and microMIPS32™ Privileged Resource Architecture which defines and governs the behavior of the privileged resources included in a MIPS® processor implementation
- Volume IV-a describes the MIPS16e™ Application-Specific Extension to the MIPS64® Architecture. Beginning with Release 3 of the Architecture, microMIPS is the preferred solution for smaller code size.
- Volume IV-b describes the MDMX™ Application-Specific Extension to the MIPS64® Architecture and microMIPS64™. It is not applicable to the MIPS32® document set nor the microMIPS32™ document set. With Release 5 of the Architecture, MDMX is deprecated. MDMX and MSA can not be implemented at the same time.
- Volume IV-c describes the MIPS-3D® Application-Specific Extension to the MIPS® Architecture.
- Volume IV-d describes the SmartMIPS®Application-Specific Extension to the MIPS32® Architecture and the microMIPS32™ Architecture .
- Volume IV-e describes the MIPS® DSP Module to the MIPS® Architecture.
- Volume IV-f describes the MIPS® MT Module to the MIPS® Architecture
- Volume IV-h describes the MIPS® MCU Application-Specific Extension to the MIPS® Architecture
- Volume IV-i describes the MIPS® Virtualization Module to the MIPS® Architecture
- Volume IV-j describes the MIPS® SIMD Architecture Module to the MIPS® 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*, and *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* and *D*
- 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 using a high-level language pseudocode 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
←	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.
x.bit[y]	Bit *y* of bitstring *x*. Alternative to the traditional MIPS notation x_y.
x.bits[y..z]	Selection of bits *y* through *z* of bit string *x*. Alternative to the traditional MIPS notation $x_{y..z}$.

Table 1.1 Symbols Used in Instruction Operation Statements (Continued)

Symbol	Meaning
x.byte[y]	Byte *y* of bitstring *x*. Equivalent to the traditional MIPS notation $x_{8*y+7..8*y}$.
x.bytes[y..z]	Selection of bytes *y* through *z* of bit string *x*. Alternative to the traditional MIPS notation $x_{8*y+7..8*z}$.
x.halfword[y] x.word[i] x.doubleword[i]	Similar extraction of particular bitfields (used in e.g., MSA packed SIMD vectors).
x.bit31, x.byte0, etc.	Examples of abbreviated form of x.bit[y], etc. notation, when y is a constant.
x.fieldy	Selection of a named subfield of bitstring *x*, typically a register or instruction encoding. More formally described as "Field y of register x". For example, FIR.D = "the D bit of the Coprocessor 1 Floating-point Implementation Register (FIR)".
+, −	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
not	Bitwise inversion
&&	Logical (non-Bitwise) AND
<<	Logical Shift left (shift in zeros at right-hand-side)
>>	Logical Shift right (shift in zeros at left-hand-side)
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 and subsequent releases, 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*

Table 1.1 Symbols Used in Instruction Operation Statements (Continued)

<table>
<tr><th>Symbol</th><th>Meaning</th></tr>
<tr><td><i>COC[z]</i></td><td>Coprocessor unit <i>z</i> condition signal</td></tr>
<tr><td><i>Xlat[x]</i></td><td>Translation of the MIPS16e GPR number <i>x</i> into the corresponding 32-bit GPR number</td></tr>
<tr><td>BigEndianMem</td><td>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.</td></tr>
<tr><td>BigEndianCPU</td><td>The endianness for load and store instructions (0 → Little-Endian, 1 → Big-Endian). In User mode, this endianness may be switched by setting the <i>RE</i> bit in the <i>Status</i> register. Thus, BigEndianCPU may be computed as (BigEndianMem XOR ReverseEndian).</td></tr>
<tr><td>ReverseEndian</td><td>Signal to reverse the endianness of load and store instructions. This feature is available in User mode only, and is implemented by setting the <i>RE</i> bit of the <i>Status</i> register. Thus, ReverseEndian may be computed as (SR_{RE} and User mode).</td></tr>
<tr><td><i>LLbit</i></td><td>Bit of <b>virtual</b> state used to specify operation for instructions that provide atomic read-modify-write. <i>LLbit</i> 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.</td></tr>
<tr><td><b>I</b>:,
<b>I+n</b>:,
<b>I-n</b>:</td><td>This occurs as a prefix to <i>Operation</i> 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 <b>I</b>. 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 <b>I</b>, 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 <b>I+1</b>.
The effect of pseudocode statements for the current instruction labeled <b>I+1</b> appears to occur “at the same time” as the effect of pseudocode statements labeled <b>I</b> 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.</td></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. he 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 or the microMIPS base architectures, 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 or microMIPS instructions</td></tr>
</table>
In the MIPS Architecture, the <i>ISA Mode</i> value is only visible indirectly, such as when the processor stores a combined value of the upper bits of PC and the <i>ISA Mode</i> into a GPR on a jump-and-link or branch-and-link instruction, or into a Coprocessor 0 register on an exception.</td></tr>
</table>

Table 1.1 Symbols Used in Instruction Operation Statements (Continued)

Symbol	Meaning
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.
FP32RegistersMode	Indicates whether the FPU has 32-bit or 64-bit floating point registers (FPRs). In MIPS32 Release 1, the FPU has 32, 32-bit FPRs, in which 64-bit data types are stored in even-odd pairs of FPRs. In MIPS64, (and optionally in MIPS32 Release2 and Release 3) the FPU has 32 64-bit FPRs in which 64-bit data types are stored in any FPR. In MIPS32 Release 1 implementations, **FP32RegistersMode** 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 **FP32RegisterMode** is computed from the FR bit in the *Status* 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 **FP32RegistersMode** is computed from the FR bit in the *Status* register.
InstructionInBranchDelaySlot	Indicates whether the instruction at the Program Counter address was executed in the delay slot of a branch or jump. This condition reflects the *dynamic* state of the instruction, not the *static* 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.
SignalException(exception, argument)	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.

1.4 Notation for Register Field Accessibility

In this document, the read/write properties of register fields use the notations shown in Table 1.1.

Table 1.2 Read/Write Register 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.

Table 1.2 Read/Write Register Field Notation (Continued)

Read/Write Notation	Hardware Interpretation	Software Interpretation
R0	R0 = reserved, read as zero, ignore writes by software. Hardware ignores software writes to an R0 field. Neither the occurrence of such writes, nor the values written, affects hardware behavior. Hardware always returns 0 to software reads of R0 fields. The Reset State of an R0 field must always be 0. If software performs an mtc0 instruction which writes a non-zero value to an R0 field, the write to the R0 field will be ignored, but permitted writes to other fields in the register will not be affected.	**Architectural Compatibility:** R0 fields are reserved, and may be used for not-yet-defined purposes in future revisions of the architecture. When writing an R0 field, current software should only write either all 0s, or, preferably, write back the same value that was read from the field. Current software should not assume that the value read from R0 fields is zero, because this may not be true on future hardware. Future revisions of the architecture may redefine an R0 field, but must do so in such a way that software which is unaware of the new definition and either writes zeros or writes back the value it has read from the field will continue to work correctly. Writing back the same value that was read is guaranteed to have no unexpected effects on current or future hardware behavior. (Except for non-atomicity of such read-writes.) Writing zeros to an R0 field may not be preferred because in the future this may interfere with the operation of other software which has been updated for the new field definition.
0	**Release 6** Release 6 legacy “0” behaves like R0 - read as zero, nonzero writes ignored. Legacy “0” should not be defined for any new control register fields; R0 should be used instead.	
	HW returns 0 when read. HW ignores writes.	Only zero should be written, or, value read from register.
	pre-Release 6 pre-Release 6 legacy “0” - read as zero, nonzero writes UNDEFINED	
	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.

Table 1.2 Read/Write Register Field Notation (Continued)

Read/Write Notation	Hardware Interpretation	Software Interpretation
R/W0	Like R/W, except that writes of non-zero to a R/W0 field are ignored. E.g. Status.NMI	
	Hardware may set or clear an R/W0 bit. Hardware ignores software writes of nonzero to an R/W0 field. Neither the occurrence of such writes, nor the values written, affects hardware behavior. Software writes of 0 to an R/W0 field may have an effect. Hardware may return 0 or nonzero to software reads of an R/W0 bit. If software performs an mtc0 instruction which writes a non-zero value to an R/W0 field, the write to the R/W0 field will be ignored, but permitted writes to other fields in the register will not be affected.	Software can only clear an R/W0 bit. Software writes 0 to an R/W0 field to clear the field. Software writes nonzero to an R/W0 bit in order to guarantee that the bit is not affected by the write.

1.5 For More Information

MIPS processor manuals and additional information about MIPS products can be found at http://www.imgtec.com.

For comments or questions on the MIPS32® Architecture or this document, send Email to IMGBA-DocFeedback@imgtec.com.

Chapter 2

The MIPS32 and microMIPS32 Privileged Resource Architecture

2.1 Introduction

The MIPS32 and microMIPS32 Privileged Resource Architecture (PRA) is a set of environments and capabilities on which the Instruction Set Architectures operate. 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 9, “Coprocessor 0 Registers” on page 113.

Chapter 3

MIPS32 and microMIPS32 Operating Modes

The MIPS32 and microMIPS32 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 MIPS 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 MIPS32 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. Release 3 (e.g. MIPSr3) introduced the microMIPS instruction set, so all microMIPS processors may implement a 64-bit floating point unit.

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.
- In an implementation of Release 2 (and subsequent releases) 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. Release 3 (e.g., MIPSr3) introduced the microMIPS instruction set. So on all microMIPS processors, 64-bit floating point operations are enabled if the F64 bit in the *FIR* register is a one.

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. 64-bit FPRs are supported for all processors using Architecture releases subsequent to Release 2, including all microMIPS processors. As of Release 5 of the Architecture, if floating point is implemented then *FR*=1 is required. I.e. the 64-bit FPU, with the *FR*=1 64-bit FPU register model, is required. The *FR*=0 32-bit FPU register model continues to be required.

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.

3.5.4 ISA Mode

Release 3 of the Architecture (e.g. MIPSr3™) introduced a second branch of the instruction set family, microMIPS32. Devices can implement both ISA branches (MIPS32 and microMIPS32) or only one branch.

The ISA Mode bit is used to denote which ISA branch to use when decoding instructions. This bit is normally not visible to software. It's value is saved to any GPR that would be used as a jump target address, such as GPR31 when written by a JAL instruction or the source register for a JR instruction.

For processors that implement the MIPS32 ISA, the ISA Mode bit value of zero selects MIPS32. For processors that implement the microMIPS32 ISA, the ISA Mode bit value of one selects microMIPS32. For processors that implement the MIPS16e™ ASE, the ISA Mode bit value of one selects MIPS16e. A processor is not allowed to implement both MIPS16e and microMIPS.

Please read *Volume II-B: Introduction to the microMIPS32 Instruction Set*, Section 5.3, "ISA Mode Switch" for a more in-depth description of ISA mode switching between the ISA branches and the ISA Mode bit.

Chapter 4

Virtual Memory

4.1 Differences between Releases 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 (and subsequent releases), 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 *Config3*$_{SP}$ 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 *Config3*$_{SP}$ bit and enabled by the *PageGrain*$_{ESP}$ bit.

4.1.2 Protection of Virtual Memory Pages

In Release 3 of the Architecture, e.g. MIPSr3, two optional control bits are added to each TLB entry. These bits, *RI (Read Inhibit)* and *XI (Execute Inhibit)*, allows more types of protection to be used for virtual pages - including write-only pages, non-executable pages.

This feature originated in the SmartMIPS ASE but has been modified from the original SmartMIPS definition. For the Release 3 version of this feature, each of the *RI* and *XI* bits can be separately implemented. For the Release 3 version of this feature, new exception codes are used when a TLB access does not obey the *RI*/*XI* bits.

4.1.3 Context Register

In Release 3 of the Architecture, e.g. MIPSr3, the *Context* register is a read/write register containing a address pointer that can point to an arbitrary power-of-two aligned data structure in memory, such as an entry in the page table entry (PTE) array. In Releases 1 & 2, this pointer was defined to reference a fixed-sized 16-byte structure in memory within a linear array containing an entry for each even/odd virtual page pair. The Release 3 version of the *Context* register can be used far more generally.

This feature originated in the SmartMIPS ASE. This feature is optional in the Release 3 version of the base architecture.

4.1.4 Segmentation Control

In Release 3 of the Architecture, e.g. MIPSr3, an optional programmable segmentation feature has been added. This improves the flexibility of the MIPS virtual address space.

With Segmentation Control, address translation begins by matching a virtual address to the region specified in a Segment Configuration. The virtual address space is therefore definable as the set of memory regions specified by Segment Configurations. The behavior and attributes of each region are also specified by Segment Configurations. Six Segment Configurations are defined, fully mapping the virtual address space.

4.1.5 Enhanced Virtual Addressing

In Release 3 of the Architecture, e.g., MIPSr3, an optional Enhanced Virtual Addressing (EVA) feature has been added. EVA is a configuration of Segmentation Control and a set of kernel mode load/store instructions allowing direct access to user-mode memory space from kernel mode. In EVA, Segmentation Control is programmed to define two address ranges, a 3 GB range with mapped-user, mapped-supervisor, and unmapped-kernel access modes and a 1 GB address range with mapped-kernel access mode.

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.

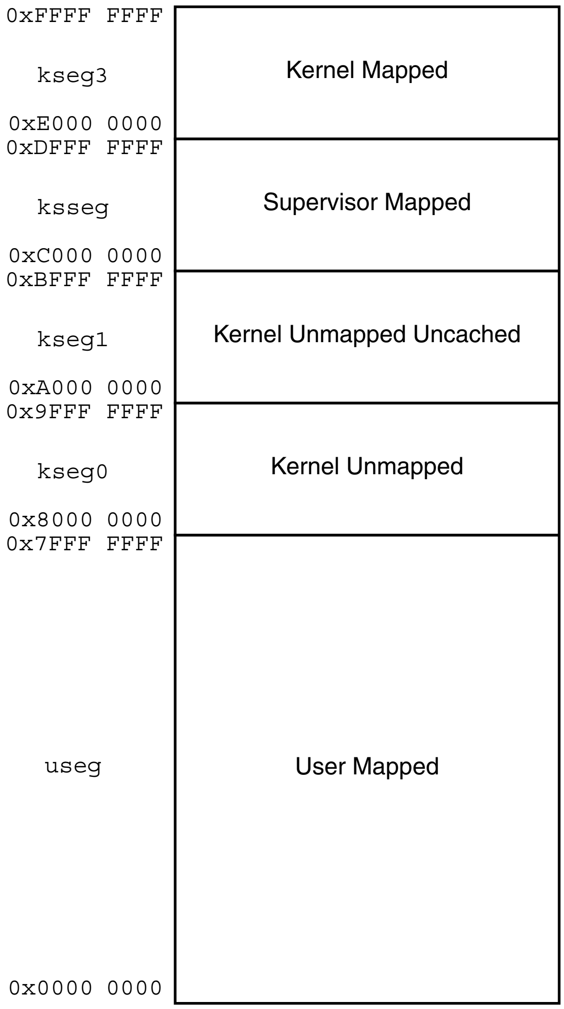
4.2.3 Physical Address Size (PABITS)

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/microMIPS32 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

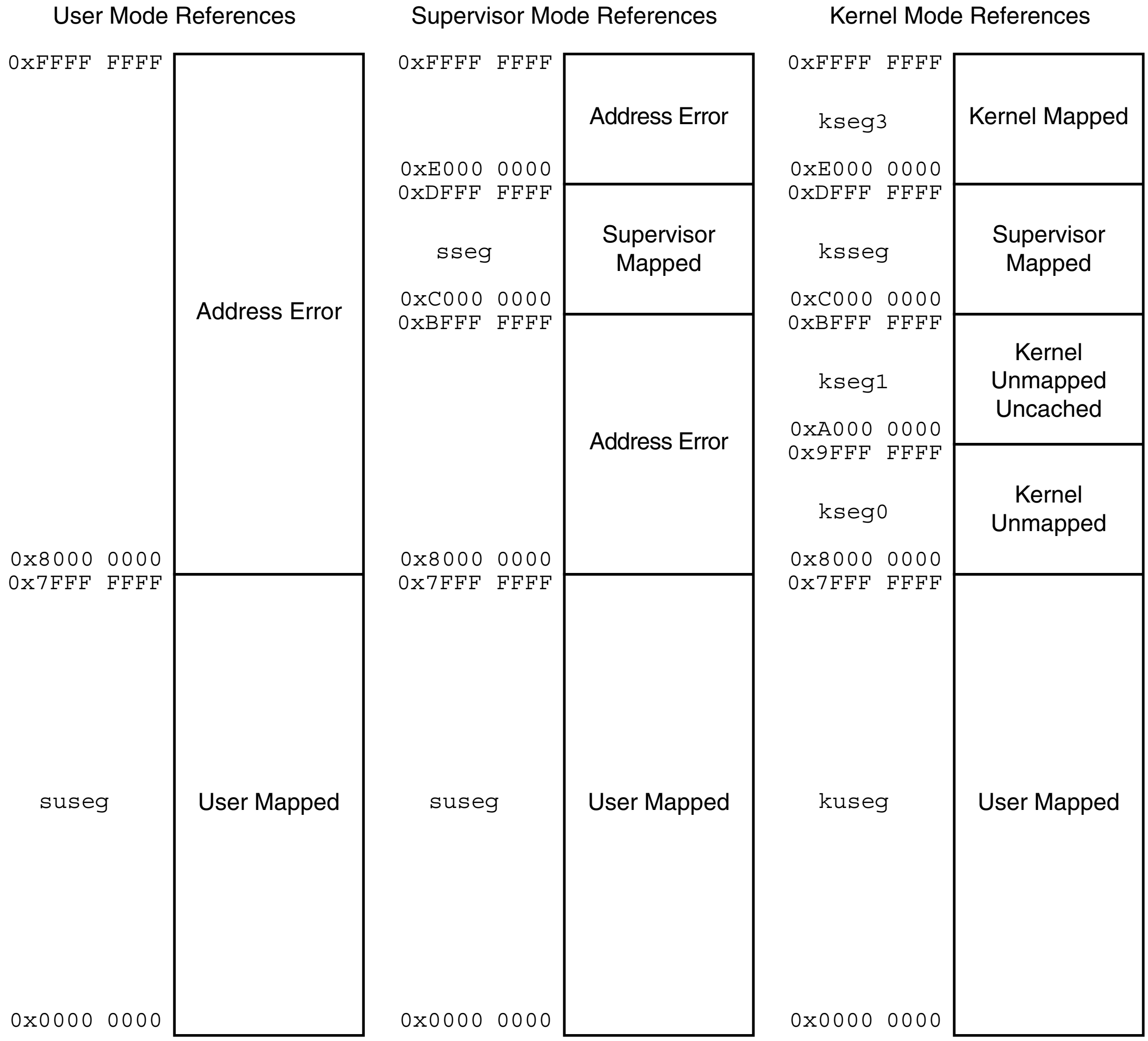
0b100	kseg0	`0x9FFF FFFF` through 0x8000 0000	Kernel	Kernel	2^{29} bytes

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 4.2 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/microMIPS32 compliant processor must implement the following Segments:

- useg/kuseg
- kseg0
- kseg1

In addition, a MIPS32/microMIPS32 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 $\text{Status}_{\text{ERL}}$=1 See Section 4.7 Mapped if $\text{Status}_{\text{ERL}}$=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/microMIPS32 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 267 and A.2 “Block Address Translation” on page 271 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.

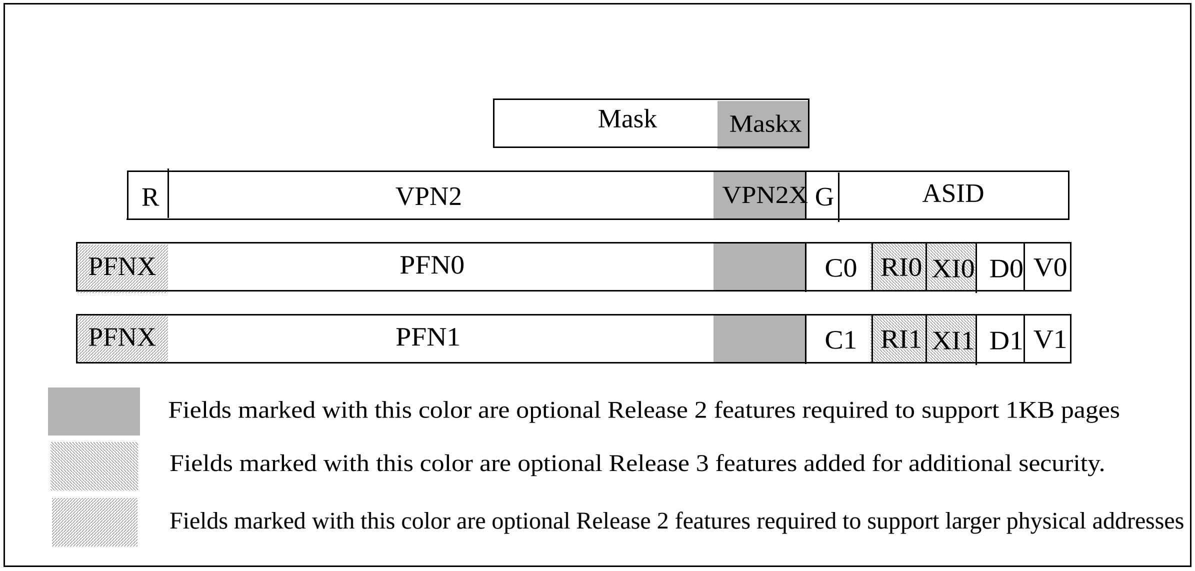
4.9.2 TLB Organization

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 and subsequent releases, 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, optionally read-inhibit and execute-inhibit (RI & XI) bits and a cache coherency field (C), whose valid encodings are given in Table 9.2. 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.

In Revision 3 of the architecture, the RI and XI bits were added to the TLB to enable more secure access of memory pages. These bits (along with the Dirty bit) allow the implementation of read-only, write-only, no-execute access policies for mapped pages.

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

Fields marked with this color are optional Release 3 features added for additional security.

Fields marked with this color are optional Release 2 features required to support larger physical addresses

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 (and subsequent releases), 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 (and subsequent releases), 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. The following example is for processors which do not implement TLB invalidate instructions, i.e. *Config4*$_{IE}$*=0x0*

```
/*
 * 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 107 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
```

The V(alid) bit within the TLB entry represents whether the Page Table Entry held in the TLB entry is valid or not. ThisValid bit does not represent whether the TLB entry has been initialized or not.

The above initialization routine relies on using unmapped addresses to be written to the VPN2 field of the TLB entry to create entries which will never match on mapped addresses. When Segmentation Control is implemented (*Config3*$_{SC}$=1), the virtual address map may be programmed to not have any unmapped address regions. For this reason, the above routine cannot be used when Segmentation Control is implemented. Instead, the TLB invalidate feature must be used. The TLB invalidate feature is discussed in the next paragraph.

Release 3 introduces another optional valid bit which denotes whether the virtual address (the VPN2 field) of the TLB entry has been initialized or not. If the *VPN2* field is marked as invalid, the entry is ignored on address match for memory accesses. This additional valid bit is visible through the EHINV field of the *EntryHi* register. If this bit is implemented (indicated by *Config4*$_{IE}$), then there are 3 ways to initialize a TLB entry: the TLBINV, TLBINVF and TLBWI instructions. This feature is required if Segmentation Control is implemented and is required for FTLB/ VTLB MMUs, optional otherwise.

For Release 3 processors which implement TLB invalidate instructions, the code to initialize the TLB is much simpler. Just write each TLB entry with the *EntryHi*$_{EHINV}$ bit set.

```
/*
 * 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:
 *     Write Each TLB entry with EntryHi.EHINV=1
 *
 * 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 107 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 */
   ori    t0, zero, 0x400
   mtc0   t0, C0_EntryHi             /* Set EHINV bit, Clear VPN2 field */
10:
   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 (and subsequent releases) 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 (and subsequent releases) 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 (and optionally RI and XI 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 (and optionally RI and XI 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. If the RI bit is implemented and is set and the reference was a load, a TLB Invalid (or TLBRI) exception is raised. If the XI bit is implemented and is set and the reference was an instruction fetch, a TLB invalid (or TLBXI) exception is raised.

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 (and subsequent releases) 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 (and subsequent releases) 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 ← $TLB[i]_{D0}$
            if ($Config3_{RXI}$ or $Config3_{SM}$) then
                ri ← $TLB[i]_{RI0}$
                xi ← $TLB[i]_{XI0}$
            endif
        else
            pfn ← $TLB[i]_{PFN1}$
            v ← $TLB[i]_{V1}$
            c ← $TLB[i]_{C1}$
            d ← $TLB[i]_{D1}$
            if ($Config3_{RXI}$ or $Config3_{SM}$) then
                ri ← $TLB[i]_{RI1}$
                xi ← $TLB[i]_{XI1}$
            endif
        endif
        if v = 0 then
            SignalException(TLBInvalid, reftype)
        endif
        if ($Config3_{RXI}$ or $Config3_{SM}$) then
            if (ri = 1) and (reftype = load) then
                if (xi = 0) and (IsPCRelativeLoad(PC))
                    # PC relative loads are allowed where execute is allowed
                else
                    if ($PageGrain_{IEC}$ = 0)
                        SignalException(TLBInvalid, reftype)
                    else
                        SignalException(TLBRI, reftype)
                    endif
                endif
            endif
```

```
         if (xi = 1) and (reftype = fetch) then
            if (PageGrain$_{IEC}$ = 0)
               SignalException(TLBInvalid, reftype)
            else
               SignalException(TLBXI, reftype)
            endif
         endif
      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]$_{D0}$
         if (Config3$_{RXI}$ or Config3$_{SM}$) then
            ri ← TLB[i]$_{RI0}$
            xi ← TLB[i]$_{XI0}$
         endif
      else
         pfn ← TLB[i]$_{PFN1}$
         v ← TLB[i]$_{V1}$
```

```
            c ← $TLB[i]_{C1}$
            d ← $TLB[i]_{D1}$
            if ($Config3_{RXI}$ or $Config3_{SM}$) then
                ri ← $TLB[i]_{RI1}$
                xi ← $TLB[i]_{XI1}$
            endif
        endif
        if v = 0 then
            SignalException(TLBInvalid, reftype)
        endif
        if ($Config3_{RXI}$ or $Config3_{SM}$) then
            if (ri = 1) and (reftype = load) then
                if (xi = 0) and (IsPCRelativeLoad(PC))
                    # PC relative loads are allowed where execute is allowed
                else
                    if ($PageGrain_{IEC}$ = 0)
                        SignalException(TLBInvalid, reftype)
                    else
                        SignalException(TLBRI, reftype)
                    endif
                endif
            endif
            if (xi = 1) and (reftype = fetch) then
                if ($PageGrain_{IEC}$ = 0)
                    SignalException(TLBInvalid, reftype)
                else
                    SignalException(TLBXI, reftype)
                endif
            endif
        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
$PFN_{(PABITS-1)-12..0}$	$PA_{PABITS-1..12}$	Release 1 implementation, or Release 2 (and subsequent releases) implementation without support for 1KB pages

PFN Range	PA Range	Comment
$PFN_{(PABITS-1)-10..0}$	$PA_{PABITS-1..10}$	Release 2 (and subsequent releases) implementation with support for 1KB pages enabled

Table 4.4 Physical Address Generation

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

4.10 Segmentation Control

As an optional alternative to fixed memory segmentation, a programmable segmentation control feature has been added to MIPSr3. This improves the flexibility of the MIPS32 virtual address space.

In the traditional MIPS32 virtual address memory map, the mappability and cacheability attributes of segments are mostly fixed. For example, useg has its mappability attribute fixed while kseg0/1 have their cacheability and mappability attributes fixed. Segmentation Control replaces these fixed attributes with programmable controls for these attributes.

The Segmentation Control system can be used to implement a fully translated flat address space, or used to alter the relative size of cached and uncached windows into the physical address space.

The existence of the unmapped segments in the virtual address map prevents a MIPS CPU from being fully virtualized. Another use of Segmentation Control is to remove the unmapped segments from the virtual address map. Future support for CPU virtualization would require Segmentation Control.

With Segmentation Control, address translation begins by matching a virtual address to the region specified in a Segment Configuration. The virtual address space is therefore definable as the set of memory regions specified by Segment Configurations. The behavior and attributes of each region are also specified by Segment Configurations. Six Segment Configurations are defined, fully mapping the virtual address space.

If Segmentation Control is implemented, the Segment Configurations are always active. Coprocessor 0 registers *SegCtl0, SegCtl1, and SegCtl2* contain six Segment Configurations. *Config5* contains additional control and configuration fields.

The attributes of a Segment Configuration are:

- Access permissions from user, kernel, and supervisor modes
- Enable mapping (address translation) using the MMU specified in $Config_{MT}$
- Physical address when mapping is disabled
- Cache attribute when mapping is disabled
- Force to unmapped, uncached when $Status_{ERL}$=1

Besides the segments controlled by *SegCtl** registers, the reset and BEV exceptions may use another segment which is active only in kernel mode. Please read *Section 4.10.1 "Exception Behavior under Segmentation Control"* for an explanation on how exceptions interact with programmable segmentation.

On reset, Segment Configuration default is implementation specific. A configuration backward compatible with MIPS32 legacy fixed segmentation is defined by Table 9.16

Segment configuration access control modes are specified in *Table 9.15*

Operation of MIPS32Segmentation Control is described below:

```
/* Inputs
 * vAddr  - Virtual Address
 * pLevel - Privilege level - USER, SUPER, KERNEL
 * IorD   - Access type - INSTRUCTION or DATA
 * LorS   - Access type - LOAD or STORE
 *
 * Outputs
 * mapped - segment is mapped
 * pAddr  - physical address (valid when unmapped)
 * CCA    - cache attribute (valid when unmapped)
 *
 * Exceptions: Address Error
 */
subroutine SegmentLookup(vAddr, pLevel, IorD, LorS) :
   Index ← vAddr[31:29]
   pAddr ← vAddr

   case Index
      7:     CFG    ← SegCtl0.CFG0
      6:     CFG    ← SegCtl0.CFG1
      5:     CFG    ← SegCtl1.CFG2
      4:     CFG    ← SegCtl1.CFG3
      3:     CFG    ← SegCtl2.CFG4
      2:     CFG    ← SegCtl2.CFG4
      1:     CFG    ← SegCtl2.CFG5
      0:     CFG    ← SegCtl2.CFG5
   endcase
   AM            ← CFG.AM
   EU            ← CFG.EU
   PA            ← CFG.PA
   C             ← CFG.C

   checkAM(AM,pLevel,IorD,LorS)

   # Special case - Error-Unmapped region when ERL=1
   if (EU = 1) and (Status$_{ERL}$=1) then
      CCA    ← 2          # uncached
      mapped ← 0          # unmapped
   else
      CCA       ← C
      mapped ←isMapped(AM, pLevel,IorD, LorS)
   endif

   # Physical address for unmapped use
   if (mapped = 0) then
      # in a large (1GB) segment, drop the low order bit.
      if (Index < 4) then
         pAddr[35:30] ← PA >> 1
      else
         pAddr[35:29] ← PA
      endif
   else
      (CCA,pAddr) ← TLBLookup(vAddr)
   endif

   return (mapped, pAddr, CCA)
endsub

# Access mode check
subroutine checkAM(AM, pLevel, IorD, LorS)
   case AM
      UK:       seg_err ← (pLevel != KERNEL)
      MK:       seg_err ← (pLevel != KERNEL)
      MSK:      seg_err ← (pLevel = USER)
      MUSK:     seg_err ← 0
      MUSUK:    seg_err ← 0
      USK:      seg_err ← (pLevel = USER)
      UUSK:     seg_err ← 0
      default:  seg_err ← UNDEFINED
   endcase

   if (seg_err != 0) then
      segmentError(IorD, LorS)
   endif
endsub

subroutine isMapped(AM, pLevel,IorD, LorS)
   case AM
      UK:       mapped ← 0
      MK:       mapped ← 1
      MSK:      mapped ← 1
      MUSK:     mapped ← 1
      MUSUK:    mapped ← (pLevel != KERNEL)
      USK:      mapped ← 0
      UUSK:     mapped ← 0
```

```
        default:  mapped ← UNDEFINED
    endcase
    return mapped
endsub

subroutine segmentError(IorD, LorS)
    if (IorD = INSTRUCTION) then
        reftype ← FETCH
    else
        if (LorS = LOAD) then
            reftype ← LOAD
        else
            reftype ← STORE
        endif
    endif
    SignalException(AddrError, reftype)
endsub
```

See Section 9.12 “SegCtl0 (CP0 Register 5, Select 2)”.

The presence of this facility is indicated by the SC field in the *Config3* register. See Section 9.44 “Configuration Register 3 (CP0 Register 16, Select 3)”.

Debug mode behavior is retained in dseg.

4.10.1 Exception Behavior under Segmentation Control

4.10.1.1 Terminology

For this section discussing exception behavior under Segmentation Control, these terms are used:

Legacy Memory map - A MIPS32 Virtual/Physical memory system as described by Section 4.3   on page 28.

Non-Reset Exceptions - exceptions which would use *EBase* for the vector location when $Status_{BEV}$=0

Overlay Segment - A memory segment with these properties:

- Totally managed by hardware, not software programmable.
- Intercepts memory requests before they are dealt with by the rest of the virtual memory system.
- Is active only in specific execution modes.

A pre-existing example of an overlay segment is DSEG which is part of the EJTAG debug architecture and is only active in DebugMode. and $ECR_{ProbeEn}$=1

4.10.1.2 Reset and BEV Vector Base Addresses under Segmentation Control

In the legacy memory map, the Reset/BEV vector base is fixed at virtual address 0xBFC0.0000 and physical address 0x1FC0.0000.

In contrast, Segmentation Control does not define a fixed value for the Reset/BEV vector base virtual address. Instead the virtual addresses and physical addresses for Reset/BEV vector base are considered implementation-specific. In

Segmentation Control, the physical address of Reset/BEV vector does not have to be derived from the virtual address by dropping VA[31:29], other mappings are allowed.

Reset and BEV exceptions - Cacheability and Map-ability

In the legacy memory map, the memory accesses to the Reset/BEV vector region are within KSEG1, which ensures the accesses to this region are always uncached and unmapped.

The architecture requires that the reset and BEV exceptions vector to a memory region which is uncached and unmapped.

Solution 1 - Uncached and Unmapped Segment always available

This architecture requirement can be satisfied if the system can guarantee these conditions:

1. One of the segments always powers up as uncached and unmapped for kernel mode.

2. That segment is always kept as uncached and unmapped for kernel mode.

3. The reset and BEV vectors always reside in the above mentioned segment.

If these conditions are met, then no special support is needed for reset and BEV exceptions.

Solution 2 - Overlay Segments for Reset and BEV exceptions

Not all systems may want to maintain the conditions for Solution 1, since Segmentation Control allows for any of the segments to be programmed with any valid cache-ability and mappability attribute.

To meet the architecture requirement without reserving one segment as uncached and unmapped, overlay segments are introduced in Segmentation Control for reset and exceptions while in kernel mode.

These overlay segments allow the reset/BEV regions to be accessed without accessing the caches and TLB during reset and BEV exceptions. That is, when a reset or BEV exception is taken, the overlay segment handles the memory requests for that vector region and the overlay segment attributes over-rides the cacheability and mappability attributes of the regular segment control register.

If Solution 1 is not implemented, the CPU must implement at least one overlay segment for the Reset/BEV vector location. If there is only one overlay segment for the Reset/BEV vector location, it must deal with memory requests as uncached and unmapped.

Solution 2 - Requirements for Overlay Segments

The starting virtual address, starting physical address and size of this overlay segment are implementation-specific. The overlay segments must be naturally aligned both in the virtual address space as well as the physical address space. The physical address of the overlay segment does not have to be derived from the virtual address of the overlay by dropping VA[31:29], other mappings are allowed.

The overlay segment must be at least 2KB in size. Implementations would likely choose much larger sizes for the overlay segment to access non-volatile memory and potentially other IO devices.

The overlay segment must be accessible while in kernel-mode ($Status_{ERL}$=1 or $Status_{ERL}$=1 or $Status_{KSU}$=kernel).

Solution 2 - Option A - Two Overlay Segments for KSEG0/1 legacy behavior

An implementation may optionally support a second overlay segment for the Reset/BEV vector physical address region. The purpose of two overlay segments is to mimic the cached and uncached views made available through KSEG0 and KSEG1 segments in the legacy memory system. After reset, one overlay segment would be given uncached and unmapped access to these vectors while the other overlay segment would give cached and unmapped access to the vectors.

The two overlay segments must meet these requirements:

- The two overlay segments are of the same size.
- The two overlay segments cannot overlap in the virtual address space.
- The two overlay segments must point to the same physical address space.
- Both overlay segments must treat memory accesses as unmapped.
- The overlay segment in which the BEV/Reset vector location resides must come out of reset treating memory accesses as uncached.
- The cache coherency of each overlay segment can be fixed by hardware or programmable through the legacy register fields in *Config* (see next section).

To mimic the legacy KSEG0/KSEG1 behaviors, one overlay segment would be located within the addresses which belong to $SEGCTL1_{CFG3}$ (virtual addresses equivalent to legacy KSEG0 segment) and the other overlay segment would be located within the addresses which belong to $SEGCTL1_{CFG2}$(virtual addresses equivalent to legacy KSEG1 segment).

Solution 2 - Option B - Overly Segments using legacy Coherency Control Register Fields

Segmentation Control allows the legacy $Config_{K0}$, $Config_{K23}$ and $Config_{KU}$ fields to control cacheability of their respective non-legacy segments coming out of reset. This is in effect when $Config5_K$ =0. If the overlay segment resides in one of these segments, it is optionally allowed for the overlay segment to get its cacheability attribute from the appropriate field (*K0*, *K23*, *KU*) within the *Config* register. If the BEV/Reset vector resides in a overlay segment which is controlled by that *Config* register field, then that register field must be set by hardware to uncached CCA value upon reset.

The use of these register fields allows the boot firmware to be run cached after the caches have been initialized. Code should not be executing within the overlay segment while the cache coherency of the overlay segment would be changing through writing the *Config* register field.

For example, if the Reset/BEV overlay segments resides within the segment controlled by $SEGCTL1_{CFG3}$ (virtual addresses equivalent to legacy KSEG0 segment) and $Config_{K0}$ is enabled coming out of reset, $Config_{K0}$ must be reset to the uncached CCA value. When $Config_{K0}$ is modified, code execution should not be within the $SEGCTL1_{CFG3}$ segment.

NOTE: This use of these legacy coherency fields within the *Config* register is only meant for systems using legacy virtual address maps. For systems using non-legacy virtual address maps, the recommendation is to disable the legacy coherency fields within the *Config* register.

Solution 1 or Solution 2 - Option C - Relocation of non-Reset BEV exception vectors after Reset

There might be transitional devices in which the physical address map was inherited from legacy systems, but the virtual address map to be used is set up by programming the Segmentation Control registers. For such transitional devices, it might be useful to relocate the non-Reset BEV exceptions to an address more appropriate for the non-legacy virtual address map. Such capability is allowed by Segmentation Control.

The $Config5_K$ bit can be used for this purpose. If $Config5_K$ =1, it is allowed to relocate the BEV vector base address for non-reset exceptions.

This feature would be used in this fashion:

1. Device boots up using legacy reset location (e.g. virtual address 0xBFC0.0000)

2. Segmentation Registers are programmed to new non-legacy address map.

3. BEV vector base moved to new location using this capability. Non-Reset BEV exceptions would now use this new location.

For the rest of this section, the following names are used:

- EffectiveBEV_VA - the virtual address of the reset/BEV vector

4.10.1.3 BEV Exceptions under Segmentation Control

As compared to a legacy system, the vector offsets are unchanged while the source of the vector base address is changed.

For Reset/Soft-Reset/NMI, the reset vector is located at virtual address (EffectiveBEV_VA).

If $Status_{BEV}$=1 during other exceptions, the vectors are located at virtual address (EffectiveBEV_VA + 0x200 + offset).

Requirements for Option 2 - Overlay Segments

If there is only one overlay segment for BEV/Reset, then the overlay segment deals with these memory requests as unmapped and uncached. The overlay segment is active in Kernel mode ($Debug_{\mathrm{DM}}$=0 and ($Status_{\mathrm{KSU}}$=Kernel or $Status_{ERL}$=1 or$Status_{EXL}$=1)).

If implemented, the second overlay segment is active at the same time as the first BEV/Reset overlay segment. If there are two overlay segments, the one which contains the reset/BEV vector must use uncached and unmapped behavior coming out of reset. Both overlay segments must use unmapped coherency.

If $Config5_{\mathrm{K}}$ =0 and the overlay resides in a segment that is controlled by one of the $Config_{\mathrm{K0}}$, $Config_{\mathrm{K23}}$ and $Config_{KU}$ register fields, it is allowed for the appropriate *Config* register field to control the cacheability attribute of the overlay segment.

4.10.1.4 Debug Exceptions under Segmentation Control

$ECR_{ProbTrap}$=0

As compared to a legacy system, the vector offset is unchanged while the source of the vector base address is changed.

The debug exception vector is located at (EffectiveBEV_VA + 0x480).

Requirements for Option 2 - Overlay Segments

The sole debug overlay segment is active when $ECR_{ProbeEn}$=1 and $Debug_{DM}$=1. A second overlay segment is not allowed for Debug exceptions.

The overlay segment deals with these memory requests as unmapped.

If $Config5_K$ =0 and the overlay resides in a segment that is controlled by one of the $Config_{K0}$, $Config_{K23}$ and $Config_{KU}$ register fields, it is allowed for the appropriate *Config* register field to control the cacheability attribute of the overlay segment. Otherwise, the overlay segment deals with these memory requests as uncached.

$ECR_{ProbTrap}$=1 and ECR_{En}=1

The debug exception vector is located at virtual address 0xFF20.0200. This virtual address is the same as in the legacy system.

The memory requests to that region are handled by the Debug overlay segment, which covers the Virtual address region of 0xFF20.0000 to 0xFF3F.FFFF. This overlay segment is active when $ECR_{ProbeTrap}$=1 and ECR_{En}=1 and $Debug_{DM}$=1. This DSEG overlay segment takes precedence over the other overlay segments.

4.10.1.5 EBase Exceptions under Segmentation Control

If $Status_{BEV}$=0, then exception vectors are located at virtual address (*Ebase*[31:12] || 0x000 + offset). These virtual addresses are the same as those in the legacy system (except now the upper 2 bits of the *Ebase* register are now also writeable.

The memory requests to that region are handled by the appropriate programmable segment.

Extended Exception Vector Placement (EBase Register)

The *EBase* register is modified to allow exception vectors to be located anywhere in the address space. See Figure 9.39.

4.10.1.6 Cache Error Exceptions under Segmentation Control

The Cache Error Exception operates as defined in the base architecture, with the following additions.

Each Segment Configuration contains an EU bit. When EU=1, the segment becomes uncached and unmapped when $Status_{ERL}$=1. On reset, this bit is set for segments covering the range 0x00000000 to 0x7FFFFFFF, to match kuseg behavior.

On a Cache Error exception, the legacy behavior requires that bit 29 of the exception vector is set true when $Status_{BEV}$=0 and the *EBase* register is present. This places the exception vector in the uncached kseg1 region.

Setting *Config5*$_{CV}$=1 allows this behavior to be overridden - the exception vector is taken directly from the *EBase* register. This feature should be used alongside Segment Configuration EU fields to ensure that code is executed from an uncached region in the event of a Cache Error exception.

The exception vector is computed as follows:

```
if $Status_{BEV}$ = 1 then
    PC ← 0xBFC0 0200 + 0x100
else
    if ArchitectureRevision ≥ 2 then
        if ($Config3_{SC}$=1) and ($Config5_{CV}$=1) then
            /* Use full value of EBase */
            PC ← $EBase_{31..12}$ || 0x100
        else
            /* $EBase_{31..29}$ ignored, resulting PC always in kseg1 */
            PC ← $101_2$ || $EBase_{28..12}$ || 0x100
        endif
    else
        PC ← 0xA000 0000 + 0x100
    endif
endif
```

4.11 Enhanced Virtual Addressing

The addition of Segmentation Control and kernel load/store instructions to the MIPS architecture provide the ability to configure virtual address ranges that exceed prior fixed segmentation limits and to access user address space from kernel mode.

The Enhanced Virtual Addressing (EVA) feature is a configuration of Segmentation Control (refer to Section 4.10 “Segmentation Control”) and a set of kernel mode load/store instructions allowing direct access to user memory from kernel mode. In EVA, Segmentation Control is programmed to define two address ranges, a 3 GB range with mapped-user, mapped-supervisor and unmapped-kernel access modes and a 1 GB address range with mapped-kernel access mode.

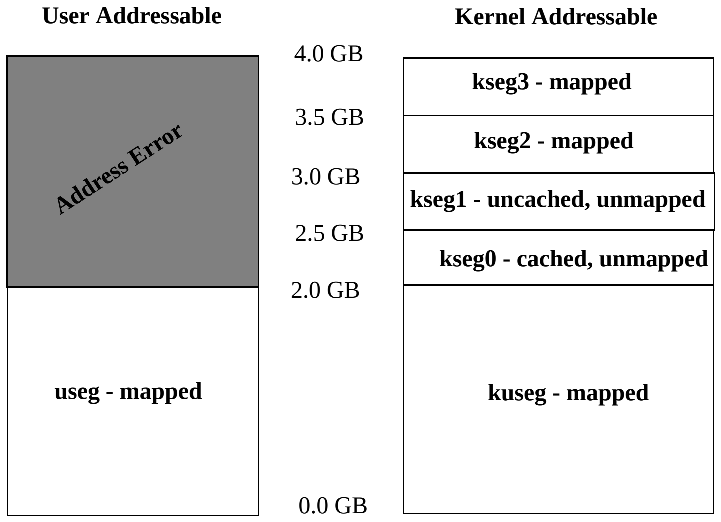
4.11.1 EVA Segmentation Control Configuration

EVA is a 2 section partitioning of the 32-bit virtual address space.

- 3.0GB Mapped User, Mapped Supervisor, Unmapped Kernel
- 1.0GB Mapped Kernel

The legacy fixed segmentation of the 32-bit virtual address space limited user addressable memory to 2.0GB as shown in Figure 4.4.

Figure 4.4 Legacy addressability



Where the EVA programmed segmentation of the 32-bit virtual address space extends user addressable memory to 3.0GB as shown in Figure 4.5.

Figure 4.5 EVA addressability

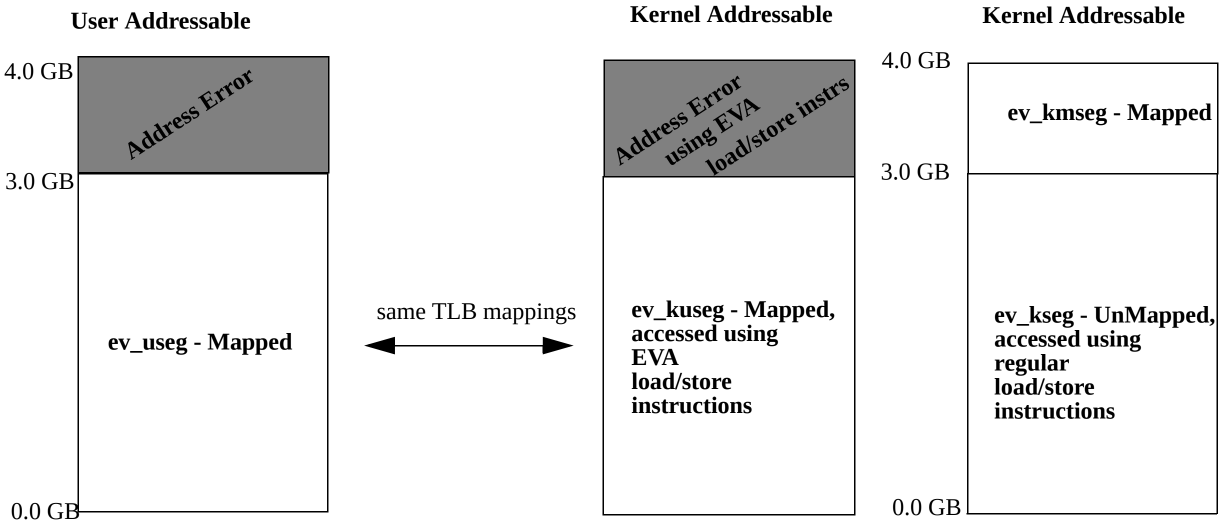
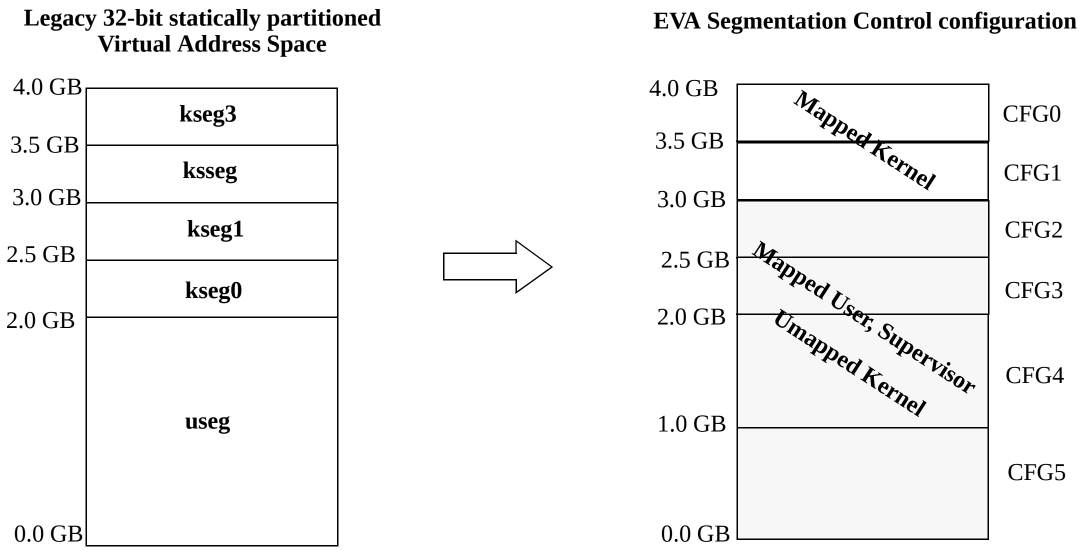

User Addressable
4.0 GB
3.0 GB
0.0 GB
Address Error
ev_useg - Mapped
same TLB mappings
Kernel Addressable
Address Error using EVA load/store instrs
ev_kuseg - Mapped, accessed using EVA load/store instructions
Kernel Addressable
4.0 GB
3.0 GB
0.0 GB
ev_kmseg - Mapped
ev_kseg - UnMapped, accessed using regular load/store instructions

Figure 4.6 shows how the Segmentation Control CFG's remap the legacy fixed partitioning.

Figure 4.6 Legacy to EVA address configuration



To support the EVA configuration, each Segment Configuration field (CFG (defined in “Segmentation Control” on page 42)) must be initialized to define the overall memory map to support a 3GB (mapped user/supervisor, unmapped kernel) memory segment.

To configure Segmentation Control to implement EVA, the *AM*, *PA, C* and *EU* fields of each *CFG* are programmed as follows in the following table.

Table 4.5 Segment Configuration for 3GB EVA

CFG	Description	AM	PA	C	EU
0	1GB Mapped Kernel	MK	0x007	3	0
1		MK	0x006	3	0
2	3GB Mapped User, Supervisor, Unmapped Kernel Region	MUSUK	0x005	3	1
3		MUSUK	0x004	3	1
4		MUSUK	0x002	3	1
5		MUSUK	0x000	3	1

4.11.2 Enhanced Virtual Address (EVA) Instructions

EVA defines a number of new load/store instructions that are used to allow the kernel to access user virtual address space while executing in kernel mode

For example, the kernel can copy data from user address space to kernel physical address space by using these instructions with user virtual addresses. Kernel system-calls from user space can be conveniently changed by replacing normal load/store instructions with these instructions. Switching modes (kernel to user) is an alternative but this is an issue if the same virtual address is being simultaneously used by the kernel. Further, there is a performance penalty in context-switching.

Limitations on use of the EVA load/store instructions are as follows:

- Only usable from Kernel execution mode.
- Only usable on a memory segment configured with a User access mode (AM).
- The address translation selected will be mapped if possible, else unmapped. More simply, a TLB based address translation is preferred.

Refer to Volume II of the MIPS Architectural Reference manual for further information on the EVA Load/Store instructions. The availabilty of these instructions are indicated by the *Config5*$_{EVA}$ register field.

Table 4.6 lists kernel load/store instructions.

Table 4.6 EVA Load/Store Instructions

Instruction Mnemonic	Instruction Name
CACHEE	Perform Cache Operation EVA
LBE	Load Byte EVA
LBUE	Load Byte Unsigned EVA
LHE	Load Halfword EVA
LHUE	Load Halfword Unsigned EVA
LLE	Load-Linked EVA
LWE	Load Word EVA
LWLE	Load Word Left EVA
LWRE	Load Word Right EVA
PREFE	Prefetch EVA
SBE	Store Byte EVA
SCE	Store Conditional EVA
SHE	Store Halfword EVA
SWE	Store Word EVA
SWLE	Store Word Left EVA
SWRE	Store Word Right EVA

Table 4.7 lists the type of address translation (mapped/unmapped) performed by EVA load/store instructions according to Segmentation Control access mode (AM) and processor execution mode (defined by *StatusKSU* = Kernel, Supervisor or User). A Coprocessor 0 unusable exception is thrown if the instruction is executed in other than Kernel mode. An Address Error exception is thrown if the access mode is not allowed.

Table 4.7 Address translation behavior for EVA load/store instructions

AM- Access Mode	Kernel	Supervisor	User
UK	Address Error Excpt	COP0 Unusable Excpt	COP0 Unusable Excpt
MK	Address Error Excpt	COP0 Unusable Excpt	COP0 Unusable Excpt

Table 4.7 Address translation behavior for EVA load/store instructions

AM- Access Mode	Kernel	Supervisor	User
MSK	Address Error Excpt	COP0 Unusable Excpt	COP0 Unusable Excpt
MUSK	mapped	COP0 Unusable Excpt	COP0 Unusable Excpt
MUSUK	mapped	COP0 Unusable Excpt	COP0 Unusable Excpt
USK	Address Error Excpt	COP0 Unusable Excpt	COP0 Unusable Excpt
UUSK	unmapped	COP0 Unusable Excpt	COP0 Unusable Excpt

Table 4.8 lists the type of address translation (mapped/unmapped) performed by ordinary load/store instructions according to Segmentation Control access mode (AM) and processor execution mode (defined by *StatusKSU* = Kernel, Supervisor or User). An Address Error exception is thrown if the access mode is not allowed in the current execution mode.

Table 4.8 Address translation behavior for ordinary load/store instructions

AM - Access Mode	Kernel	Supervisor	User
UK	unmapped	Address Error Excpt	Address Error Excpt
MK	mapped	Address Error Excpt	Address Error Excpt
MSK	mapped	mapped	Address Error Excpt
MUSK	mapped	mapped	mapped
MUSUK	unmapped	mapped	mapped
USK	unmapped	unmapped	Address Error Excpt
UUSK	unmapped	unmapped	unmapped

4.12 Hardware Page Table Walker

Page Table Walking is the process by which a Page Table Entry (PTE) is located in memory. Hardware acceleration for page table walking is an optional feature in the architecture. The mechanism can be used to replace the software handler for the TLB Refill condition. This hardware mechanism is only used for this fast-path handler. This hardware mechanism is not used for the TLB Invalid handler (or slow-path handler).

The MIPS Privileged Resource Architecture (PRA) includes mechanisms intended for rapid handling of TLB exceptions in software. Following a TLB-related exception, the *Context* register can provide the address of a TLB entry - calculated from the faulting virtual address and a Page Table Base address. This mechanism is effective when the OS page table is single level, the TLB entry is 16 bytes in size, and a 4k physical page size is used. Unfortunately, modern operating systems use multi-level page tables, use different page sizes, and store TLB entries in 8, 16 byte and 32-byte forms.

The existence of the Hardware Page Walking feature is denoted when *Config3*$_{PW}$=1.

The Hardware Page Table Walker feature additionally includes enhancements to page table entry format, as follows:

1. Huge Page support in directories (non-leaf levels of the Page Table hierarchy), and Base Page Size for the (Page Table Entry (PTE) levels (leaf levels of the Page Table hierarchy). This is the baseline definition. Inferred size PTEs are supported at non-leaf levels.

2. A reserved field has been added to PTEs. This field is for future extensions.

A Huge Page may logically be specified in two ways:

1. A Huge Page is a region composed of two power-of-4 pages which have adjacent virtual and physical addresses. Since the even page and the odd page are derived from a single directory entry, they will both inherit the same attributes and all but one of the address bits from the single directory entry. The memory region is divided evenly between the even page and the odd page. The physical address held within the directory entry is aligned to 2 x size of the page (which is a power of 4). This is distinct from *EntryLo0* and *EntryLo1* pairs in the Page Table which are only guaranteed to be adjacent in virtual, but not physical address. They may also have differing page attributes. This method is known as **Adjacent Pages** since the *EntryLo0/1* physical addresses are both derived from one entry and have to be adjacent in the physical address space. This is the default method that is supported by this specification. If an implementation chooses to support Huge Pages in the directory levels, then the Adjacent Page method must be implemented.

2. Where a Huge Page is itself a power-of-4 page, it is handled in exactly the same manner as a Base Page in the Page Table. For this case, one directory entry is used for the even page and the adjacent directory entry is used for the odd page. The physical address held within the directory entry is aligned to the size of the page (which is a power of 4). This method is known as **Dual Pages** since each PFN does not have to be adjacent to each other. If an implementation chooses to support Huge Pages in the directory levels, then the Dual Page method is an additional option.

Examples of power-of-4 regions(start with 1KB and multiply by 4 a number of times): 256MB, 1MB, 4MB, 16MB, 64MB, 256MB, 1GB.

Examples of 2x power-of-4 regions (start with 1KB and multiply by 4 a number of times; then multiple by 2) 512MB, 2MB, 8MB, 32MB, 128MB, 512MB, 2GB.

Huge Page Support is optional and is indicated by $PWCtl_{Hugepg}$=1. If an Implementation supports Huge Pages in the directory levels, it must support the Adjacent Page method. The Dual Page method is optional if Huge Pages are supported. The implementation of Dual Page method is indicated by $PWCtl_{DPH}$=1

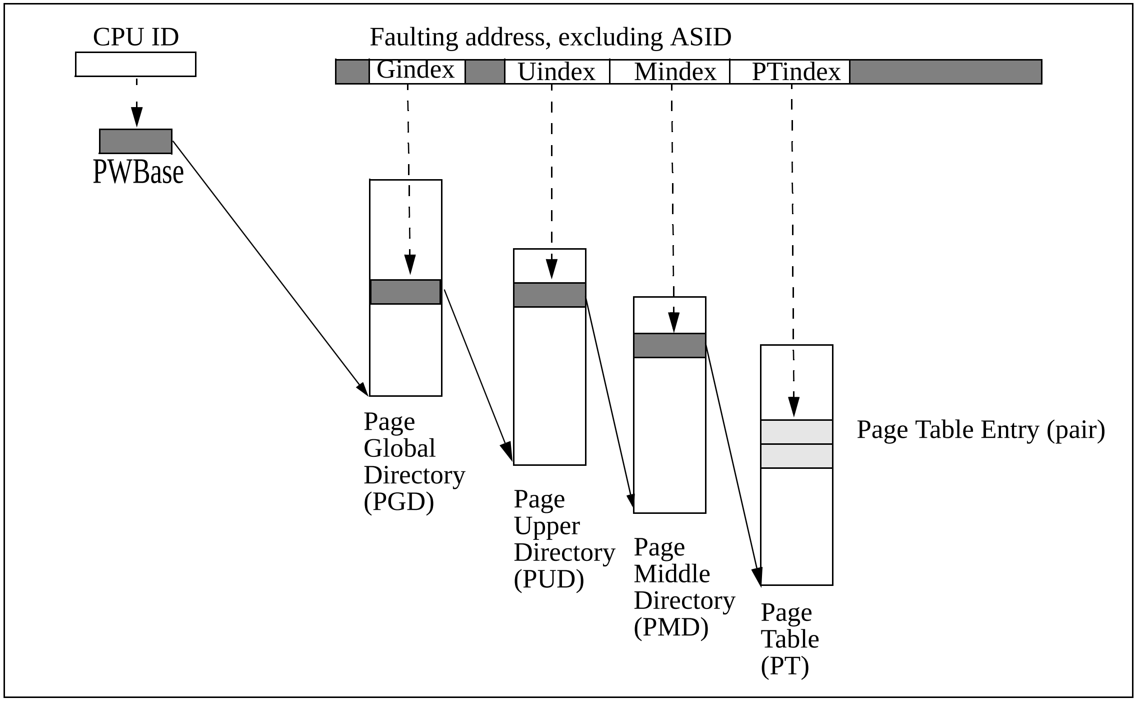
4.12.1 Multi-Level Page Table support

The hardware page table walking system specifies a mechanism for refilling the TLB, independent of the *Context* register. Four additional coprocessor 0 registers are added. The *PWBase* register specifies the per-VPE page table base. The *PWField* and *PWSize* registers specify address generation for up to four levels of page table. The *PWCtl* register controls the behavior of the Page Table Walker. These registers also configure the separation between Page Table Entries (PTEs) in memory and post-load shifting of PTEs.

A multi-level page table system forms a tree structure - the lowest (leaf) elements of which are Page Tables. A Page Table is an array of Page Table Entries. Levels above the Page Tables are known as Directories. A Directory consists of an array of pointers. Each pointer in a Directory is either to another Directory or to a Page Table.

The next figure shows an example of a multi-level page table structure.

Figure 4.7 Page Table Walk Process


CPU ID
Faulting address, excluding ASID
Gindex
Uindex
Mindex
PTindex
PWBase
Page Global Directory (PGD)
Page Upper Directory (PUD)
Page Middle Directory (PMD)
Page Table (PT)
Page Table Entry (pair)

Each executing process is typically associated with a separate page table base pointer (*PWBase*). In a single-threaded, uniprocessor system, only one process is active at once. Where multiple CPUs or VPEs are in use, multiple processes execute simultaneously - thus one page table base pointer is required per CPU or VPE. The term ‘page table base’ refers to the start of a Page Global Directory.

A typical page table structure consists of:

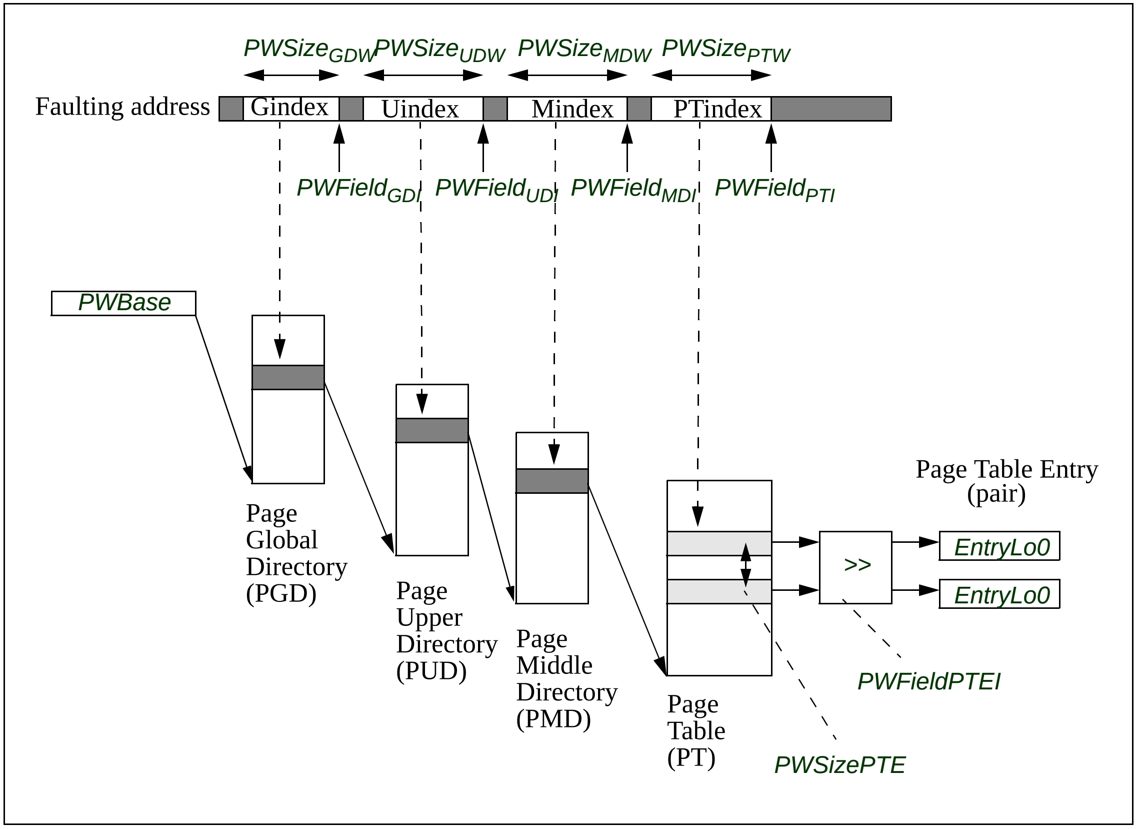
- A per CPU/VPE *PWBase* register, containing the base of the Page Global Directory.
- Page Global Directories, indexed by upper bits from the faulting address, containing pointers to Page Upper Directories.
- Page Upper Directories, indexed by bits from the faulting address, containing pointers to Page Middle Directories.
- Page Middle Directories, indexed by bits from the faulting address, containing pointers to Page Tables.
- Page Tables, indexed by bits from the faulting address, containing Page Table Entry (PTE) pairs.

In some 32-bit systems, the Page Upper Directories and Page Middle Directories are not used. Some systems may wish to exclude certain bits of the faulting address when performing a page table walk. Some systems use bits in the Page Table Entries to store OS-specific flags, which are removed using a shift before writing into EntryLo0/1. Other systems store these flags alongside the PTEs. Some hardware implementations may seek to include more than one page table walker, allowing out-of-order execution to continue despite multiple TLB misses.

The hardware page table walking scheme takes account of all these possibilities.

Figure 4.8 shows the registers and fields used by the page table walking scheme for a four level page table structure.

Figure 4.8 Page Table Walk Process & COP0 Control fields



Hardware page table walking is performed when enabled and a TLB refill condition is detected.

Hardware page table walking is enabled when

// it's globally enabled and

PWCtl$_{PWEn}$=1 and

// There's a page table structure to walk

(*PWSize*$_{GDW}$>0 | *PWSize*$_{UDW}$>0 | *PWSize*$_{MDW}$>0) .

Memory reads during hardware page table walking are performed as if they were kernel-mode load instructions. Addresses contained in the *PWBase* register and in memory-resident directories are virtual addresses.

Physical addresses and cache attributes are obtained from the Segment Configuration system when *Config3*$_{SC}$=1, or from the default MIPS segment system when *Config3*$_{SC}$=0.

The hardware page walk write should treat the multiple-hit case the same as a TLBWR. Assuming that the write by design cannot detect all duplicates, then a preferred implementation is to invalidate the single duplicate and then write the TLB. A Machine Check exception may subsequently be taken on a TLBP or lookup of TLB.

If a synchronous exception condition is detected during the hardware page table walk, the HW walking process is aborted and a TLB Refill exception will be taken. This includes synchronous exceptions such as Address Error, Precise Debug Data Break and other TLB exceptions resulting from accesses to mapped regions.

If an asynchronous exception is detected during the hardware page table walk, the HW walking process is aborted and the asynchronous exception is taken. This includes asynchronous exceptions such as NMI, Cache Error, and Interrupts. It also includes the asynchronous Machine Check exception which results from multiple matching entries being present in the TLB following a TLB write.

Implementations are not required to support hardware page table walk reads from mapped regions of the Virtual Address space. If an implementation does not support reads from mapped regions, an attempted access during a page table walk will cause the process to be aborted, and a TLB Refill exception will be taken.

Pointers within Directories are always treated as 32 bit addresses.

Hardware page table walking is performed as follows:

1. A temporary pointer is loaded with the contents of the *PWBase* register

2. The native pointer size is set to 4 bytes (32 bits).

3. If the Global Directory is disabled by $PWSize_{GDW}$=0, skip to the next step.

    - If Huge Pages are supported, check PTEVld bit to determine if entry is PTE. If PTEVld bit is set, write Huge Page into TLB (details left out for brevity, read pseudo-code at end of this section). Page Walking is complete after Huge Page is written to TLB.

    - Extract $PWSize_{GDW}$ bits from the faulting address, with least-significant bit $PWField_{GDI}$. This is the Global Directory index (Gindex). Logical OR onto the temporary pointer, after multiplying (shifting) by the native pointer size. The result is a pointer to a location within the Global Directory.

    - Perform a memory read from the address in the temporary pointer, of the native pointer size. The returned value is placed into the temporary pointer. If an exception is detected, abort.

4. If the Upper Directory is disabled by $PWSize_{UDW}$=0, skip to the next step.

    - If Huge Pages are supported, check PTEVld bit to determine if entry is PTE. If PTEVld bit is set, write Huge Page into TLB (details left out for brevity, read pseudo-code at end of this section). Page Walking is complete after Huge Page is written to TLB.

    - Extract $PWSize_{UDW}$ bits from the faulting address, with least-significant bit $PWField_{UDI}$. This is the Upper Directory index (Uindex). Logical OR onto the temporary pointer, after multiplying (shifting) by the native pointer size. The result is a pointer to a location within the Upper Directory.

    - Perform a memory read from the address in the temporary pointer, of the native pointer size. The returned value is placed into the temporary pointer. If an exception is detected, abort.

5. If the Middle Directory is disabled by $PWSize_{MDW}$=0, skip to the next step.

    - If Huge Pages are supported, check PTEVld bit to determine if entry is PTE. If PTEVld bit is set, write Huge Page into TLB (details left out for brevity, read pseudo-code at end of this section). Page Walking is complete after Huge Page is written to TLB.

    - Extract $PWSize_{MDW}$ bits from the faulting address, with least-significant bit $PWField_{MDI}$. This is the Middle Directory index (Mindex). Logical OR onto the temporary pointer, after multiplying (shifting) by the native pointer size. The result is a pointer to a location within the Middle Directory.

- Perform a memory read from the address in the temporary pointer, of the native pointer size. The returned value is placed into the temporary pointer. If an exception is detected, abort.

- The temporary pointer now contains the address of the Page Table to be used.

6. Extract *PWSize*$_{PTW}$ bits from the faulting address, with least-significant bit *PWField*$_{PTI}$ This is the Page Table index (PTindex). Multiply (shift) by the native pointer size, then multiply (shift) by the size of the Page Table Entry, specified in *PWSize*$_{PTEW}$.

   - The temporary pointer now contains the address of the first half of the Page Table Entry.

   - Perform a memory read from the address in the temporary pointer, of the native pointer size. The returned value is logically shifted right by *PWField*$_{PTEI}$ bits. This is the first half of the Page Table Entry. If an exception is detected, abort.

7. In the temporary pointer, set the bit located at bit location *PWField*$_{PTEI}$-1.

   - The temporary pointer now contains the address of the second half of the Page Table Entry.

   - Perform a memory read from the address in the temporary pointer, of the native pointer size. The returned value is shifted right by *PWField*$_{PTEI}$ bits. This is the second half of the Page Table Entry. If an exception is detected, abort.

8. Write the two halves of the Page Table Entry into the TLB, using the same semantics as the TLBWR (TLB write random) instruction.

9. Continue with program execution.

Coprocessor 0 registers which are used by software on TLB refill exceptions are unused by the hardware page table walking process. The registers and fields used by software are *BadVAddr*, *EntryHi, PageMask, EntryLo0*, *EntryLo1* and *Context*$_{BadVPN2}$.

4.12.2 PTE and Directory Entry Format

All entries are read from in-memory data structures. There are three types of entries in the baseline definition: Directory Pointer, Huge Page non-leaf PTE of inferred size, and leaf PTE of base size. For options other than baseline, the entry type is a function of the table level and the PTEvld field of an entry. For all but the last level table (leaf level), the PTEvld bit is 0 for directory pointers to the next table and 1 for PTEs. In the leaf table, the entry is alway a PTE and the PTEvld bit is not used by Hardware Walker. The *PWCtl*$_{HugePg}$ register field indicates whether Huge Page non-leaf PTEs are implemented.

All PTEs are shifted right by *PWField*$_{PTEI}$ -2 (shifting in zeros at the most significant bit) and then rotated right by 2 bits before forming the page-walker equivalents of *EntryLo0* and *EntryLo1* values. These operations are used to remove the Software-only bits and placing the RI and XI protection bits in the proper bit location before writing the TLB. If the RI and XI bits are implemented and enabled, the HW Page Walker feature requires the RI bit to be placed right of the G bit in the PTE memory format. Similarly, it is required that the XI bit to be placed right of the RI bit in the PTE memory format.

Note that the bit position of PTEvld is not fixed at 0. It can be programmed by the *PWCtl*$_{Psn}$ field. If non-leaf PTE entries are available, there will already be a bit used by the software TLB handler to distinguish non-leaf PTE entries from directory pointers. Normally, the PTEvld bit is configured to point to that software bit within the PTE.

A possible programming error to avoid is placing the PTEvld bit within the Directory Pointer field, as any of those address bits may be set and thus not appropriate to be used to distinguish between a Directory Pointer or a non-leaf PTE.

The following figures show an example of 4-byte pointers or PTE entries. The 4-byte width is configured by having*PWSIze*$_{PTEW}$=0. In this example, 4bits are used for Software-only flags. The following figures assume a PTE format based on *PWCtl*$_{Psn}$=0, *PWField*$_{PTEI}$=6 and a Base Page Size of 4K for simplicity.

Figure 4.9 4-byte Leaf PTE

31 12	11 9	8	7	6	5	4	3..0	Comment
PFN	C	D	V	G	RI	XI	S/W Use	Page Size=Base

Figure 4.10 4-byte Non-Leaf PTE Options

31 16	15 12	11 9	8	7	6	5	4	3..0	Comment
PFN	Reserved (must be 0)	C	D	V	G	RI	XI	S/W Use	Page Size=HgPgSz PTE format in memory

31 16	15 12	11 9	8	7	6	5	4	3..1	0	
PFN	Reserved (must be 0)	C	D	V	G	RI	XI	Unused by HW	PTEvld=1	Page Size=HgPgSz PTE format interpreted by HW Page Walker; PTEvld configured to be at bit 0

31 12	11 1	0	
Dir Pointer 31:12	0	PTEvld=0	Directory Ptr format interpreted by HW Page Walker; PTEvld configured to be at bit 0

After shifting out the software bits (3..0) (shifting in zeros at the most significant bit) and then rotating *RI* and *XI* fields into bits 31:30, the PTE matches the *EntryLo* register format. In the non-Leaf PTE, 4-bits which are just left of the *C* field are reserved for future features.

Figure 4.11 4-Byte Rotated PTE Formats

Comment	31	30	29 6	5..3	2	1	0	Comment
Leaf PTE	RI	XI	PFN	C	D	V	G	Page Size=Base

	31	30	29 10	9:6	5..3	2	1	0	
Non-leaf PTE	RI	XI	PFN	Reserved (must be 0)	C	D	V	G	Page Size=HgPgSz

The following figures show an example of 8-byte pointers or PTE entries. The 8-byte width is configured by having*PWSize*$_{PTEW}$=1. This example uses 4-bits for Software-only flags. The use of the wider PTE allows for the use of more *PFN* bits to be used for addressing - the 8-byte PTE format is required when more than 32-bits of physical addressing is to be implemented. Both the non-leaf PTE and directory pointer both take 8-bytes of memory space, though only 32-bits are actually used for the memory address. The following figures assume a PTE format based on *PWCtl*$_{Psn}$=0, *PWField*$_{PTEI}$=6 and a Base Page Size of 4k for simplicity.

Figure 4.12 8-byte Leaf PTE

63:36	35 12	11.9	8	7	6	5	4	3..0	Comment
Rsvd	PFN	C	D	V	G	RI	XI	S/W Use	Page Size=Base

Figure 4.13 8-Byte Non-leaf PTE Options

63:36	35 16	15 12	11.9	8	7	6	5	4	3..0	Comment
Rsvd	PFN	Reserved (must be 0)	C	D	V	G	RI	XI	S/W Use	Page Size=HgPgSz PTE format in memory

63:39	35 16	15 12	11.9	8	7	6	5	4	3..1	0	
Rsvd	PFN	Reserved (must be 0)	C	D	V	G	RI	XI	Unused by HW	PTEvld=1	Page Size=HgPgSz PTE format interpreted by HW Page Walker; PTEvld configured to be at bit 0

63 32	31..12	11 1	0	
Rsvd	Directory Ptr	0	PTEvld=0	Directory Pointerformat interpreted by HW Page Walker; PTEvld configured to be at bit 0

After the software bits (3..0) are right shifted away (shifting in zeros at the most significant bit) and the RI and XI fields are rotated to bits 31:30, the PTE matches the *EntryLo* register format. By setting $PWSIze_{PTEW}$=1 to denote 8-byte PTE entries, the shift operation is done on the entire 8 byte PTE, but only the lower 4-bytes are written into the TLB. In the non-Leaf PTE, 4-bits which are just left of the *C* field are reserved for future features.

Figure 4.14 8-Byte Rotated PTE Formats

Comment	31	30	29 6	5..3	2	1	0	Comment
Leaf PTE	RI	XI	PFN	C	D	V	G	Page Size=Base

	31	30	29..10	9..6	5..3	2	1	0	
Non-leaf PTE	RI	XI	PFN	*Rsvd* *(must be 0)*	C	D	V	G	Page Size=HgPgSz

Leaf PTEs always occur in pairs (*EntryLo0* and *EntryLo1*). However, non-leaf PTEs (ones which occur in the upper directories) can occur either in pairs (if Dual Page method is enabled) or occur with just one entry (Adjacent Page method).

For the Adjacent Page method, the single non-leaf PTE represent both *EntryLo0* and *EntryLo1* values. When the walker populates the EntryLo registers for a PTE in a directory, the least significant bit above the page size is 0 for *EntryLo0* and 1 for *EntryLo1*. That is, *EntryLo0* and *EntryLo1* represent adjacent physical pages.

For the Dual Page method, the two PTEs are read from the directory level by the Hardware Page Walker.

For Huge Page handling, the size of the Huge Page is inferred from the directory level in which the Huge Page resides. For the Adjacent Page Method, the size of each individual PTE in *EntryLo0* and *EntryLo1* as synthesized from the single Huge Page is always half the inferred size.

If the inferred page size is 2 x power-of-4, then the Adjacent Page Method is used.

If the inferred page size is a power-of-4, then the Dual Page Method is used (if the Dual Page Method is implemented). If the Dual Page method is implemented (*PWCtl*$_{DPH}$=1), it is implementation-specific whether the PTEVld bit is checked for the second PTE when it is read from memory for writing the second TLB page. The recommended behavior is to check this second PTEVld bit and if it is not set, a Machine Check exception is triggered. The *PageGrain*$_{MCCause}$ register field is used to differentiate between different types of Machine Check exceptions.

If the the inferred Huge Page size is power-of-4, and the Dual Page Methods is not implemented, it is implementation-specific whether a Machine Check is reported.

An example of Huge Page handling follows. It assumes a leaf PTE size of 4KB.

- PMD Huge Page = 2^9 (*PWSize*$_{PTW}$) * 2^12 (*PWField*$_{PTI}$) = 2^21 = 2MB. Each EntryLo0/1 page is 1MB, which is a power-of-4 and use the Adjacent Page method.

- PUD Huge Page = 2^10 (*PWSize*$_{MDW}$) * 2^9 (*PWSize*$_{PTW}$) * 2^12 (*PWField*$_{PTI}$) = 2^31 = 2GB. Each EntryLo0/1 page is 1GB, which is a power-of-4 and would use the Adjacent Page method. Note that the index into PMD has been extended to 10 bits from 9 bits. Each PMD table thus has 1K entries instead of the typical 512 entries.

See also:

- Section 9.15, "PWBase Register (CP0 Register 5, Select 5)" on page 149
- Section 9.16, "PWField Register (CP0 Register 5, Select 6)" on page 149
- Section 9.17, "PWSize Register (CP0 Register 5, Select 7)" on page 152
- Section 9.19, "PWCtl Register (CP0 Register 6, Select 6)" on page 159

4.12.3 Hardware page table walking process

The hardware page table walking process is described in pseudocode as follows:

```
/* Perform hardware page table walk
 *
 * Memory accesses are performed using the KERNEL privilege level.
 * Synchronous exceptions detected on memory accesses cause a silent exit
 * from page table walking, resulting in a TLB  Refill exception.
 *
 * Implementations are not required to support page table walk memory
 * accesses from mapped memory regions. When an unsupported access is
 * attempted, a silent exit is taken, resulting in a TLB  Refill exception.
 *
 * Note that if an exception is caused by AddressTranslation or LoadMemory
 * functions, the exception is not taken, a silent exit is taken,
 * resulting in a TLB  Refill exception.
 *
 * For readability, this pseudo-code does not deal with PTEs of different widths.
 * In reality, implementations will have to deal with the different PTE
 * and directory pointer widths.

 */
subroutine PageTableWalkRefill(vAddr) :
```

```
if (Config3$_{PW}$ = 0) then
    return(0)  # walker is unimplemented

if (PWCtl$_{PWEn}$=0) then
    return (0)  # walker is disabled

if !(PWSize$_{GDW}$>0|PWSize$_{UDW}$>0|PWSize$_{MDW}$>0) then
    return (0) # no structure to walk

    # Initial values
found  ← 0

encMask ← 0
HugePage ← False
HgPgBDhit ← False
HgPgGDhit ← False
HgPgUDhit ← false
HgPgMDhit ← false

# Native pointer size
NativeShift  ← 2
DSize        ← 32

# Indices computed from faulting address

Gindex    ← (vAddr >> PWField$_{GDI}$) and((1<<PWSize$_{GDW}$)-1)
Uindex    ← (vAddr >> PWField$_{UDI}$) and((1<<PWSize$_{UDW}$)-1)
Mindex    ← (vAddr >> PWField$_{MDI}$) and ((1<<PWSize$_{MDW}$)-1)
PTindex   ← (vAddr >> PWField$_{PTI}$) and((1<<PWSize$_{PTW}$)-1)

# Offsets into tables
Goffset   ← Gindex << NativeShift
Uoffset   ← Uindex << NativeShift
Moffset   ← Mindex << NativeShift
PToffset0 ← (PTindex >> 1) << (NativeShift + PWSize$_{PTEW}$+1)
PToffset1 ← PToffset0 OR (1 << (NativeShift + PWSize$_{PTEW}$))

EntryLo0 ← UNPREDICTABLE
EntryLo1 ← UNPREDICTABLE
Context$_{BadVPN2}$ ← UNPREDICTABLE

# Starting address - Page Table Base
vAddr ← PWBase

# Global Directory
if (PWSize$_{GDW}$ > 0) then
    vAddr         ← vAddr or Goffset
    (pAddr, CCA)  ← AddressTranslation(vAddr, DATA, LOAD, KERNEL)
    t             ← LoadMemory(CCA, DSize, pAddr, vAddr, DATA)

    if (t and (1<<PWCtl$_{Psn}$) && PWCtl$_{Hugpg}$=1) then # PTEvld is set
        HugePage ← true
        HgPgGDHit ← true
        t  ← t >> PWField$_{PTEI}$ - 2 // shift entire PTE
        t  ← ROTRIGHT(t, 2) // 32-bit rotate to place RI/XI bits
        w ← (PWField$_{GDI}$)-1
```

```
      if ( ( PWField$_{GDI}$ and 0x1)=1) // check if index is odd e.g. 2x power of 4
      // generate adjacent page from same PTE for odd TLB page
         lsb ← (1<<w)>> 6
         pw_EntryLo0 ← t and not lsb # lsb=0 even page; note FILL fields are 0
         pw_EntryLo1 ← t or lsb # lsb=1 odd page
      elseif (PWCtl$_{DPH}$ = 1)
      // Dual Pages - figure out whether even or odd page loaded first
         OddPageBit = (1 << PWField$_{GDI}$)
         if (vAddr and OddPageBit)
            pw_EntryLo1 ← t
         else
            pw_EntryLo0 ← t
         endif
      // load second PTE from directory for other TLB page
         vAddr2 ←vAddr xor OddPageBit
         (pAddr2, CCA2) ←AddressTranslation(vAddr2, DATA, LOAD, KERNEL)
         t  ← LoadMemory(CCA2, DSize, pAddr2, vAddr2, DATA)
         t  ← t >> PWField$_{PTEI}$ - 2 // shift entire PTE
         t  ← ROTRIGHT(t, 2) // 32-bit rotate to place RI/XI bits
         if (vAddr and OddPageBit)
            pw_EntryLo0 ← t
         else
            pw_EntryLo1 ← t
         endif
      else
         goto ERROR
      endif
      goto REFILL
   else
      vAddr ← t
   endif
endif

# Upper directory
if (PWSize$_{UDW}$ > 0) then
   vAddr         ← vAddr or Uoffset
   (pAddr, CCA)  ← AddressTranslation(vAddr, DATA, LOAD, KERNEL)
   t             ← LoadMemory(CCA, DSize, pAddr, vAddr, DATA)

   if (t and (1<<PWCtl$_{Psn}$) && PWCtl$_{Hugpg}$=1) then# PTEvld is set
      HugePage ← true
      HgPgUDHit ← true
      t  ← t >> PWField$_{PTEI}$ - 2 // right-shift entire PTE
      t  ← ROTRIGHT(t, 2) // 32-bit rotate to place RI/XI bits
      w ← (PWFIELD$_{UDI}$)-1
      if ( (PWFIELD$_{UDI}$ and 0x1)= 0x1) //check if odd e.g. 2x power of 4
      // generate adjacent page from same PTE for odd TLB page
         lsb ← (1<<w)>> 6 // align PA[12] into EntryLo* register bit 6
         pw_EntryLo0 ← t and not lsb # lsb=0 even page; note FILL fields are 0
         pw_EntryLo1 ← t or lsb # lsb=1 odd page
      elseif (PWCtl$_{DPH}$ = 1)
      // Dual Pages - figure out whether even or odd page loaded first
         OddPageBit = (1 << PWFIELD$_{UDI}$)
         if (vAddr and OddPageBit)
            pw_EntryLo1 ← t
         else
            pw_EntryLo0 ← t
         endif
```

```
        // load second PTE from directory for odd TLB page
            vAddr2 ←vAddr xor OddPageBit
            (pAddr2, CCA2) ←AddressTranslation(vAddr2, DATA, LOAD, KERNEL)
            t  ← LoadMemory(CCA2, DSize, pAddr2, vAddr2, DATA)
            t  ← t >> PWField$_{PTEI}$ - 2 // right-shift entire PTE
            t  ← ROTRIGHT(t, 2) // 32-bit rotate to place RI/XI bits
            if (vAddr and OddPageBit)
                pw_EntryLo0 ← t
            else
                pw_EntryLo1 ← t
            endif
        else
            goto ERROR
        endif
        goto REFILL
    else
        vAddr ← t
    endif
endif

# Middle directory
if (PWSize$_{MDW}$ > 0) then
    vAddr           ← vAddr OR Moffset
    (pAddr, CCA)    ← AddressTranslation(vAddr, DATA, LOAD, KERNEL)
    t               ← LoadMemory(CCA, DSize, pAddr, vAddr, DATA)
    if (t and (1<<PWCtl$_{Psn}$) && PWCtl$_{Hugpg}$=1) then# PTEvld is set
        HugePage ← true
        HgPgMDHit ← true
        t  ← t >> PWField$_{PTEI}$ - 2 // right-shift entire PTE
        t  ← ROTRIGHT(t, 2) // 32-bit rotate to place RI/XI bits
        pw_EntryLo0 ← t # note FILL fields are 0
        w ← (PWField$_{MDI}$)-1
        if ( (PWField$_{MDI}$ and 0x1)= 0x1) // check if odd e.g. 2x power of 4
        // generate adjacent page from same PTE for odd TLB page
        lsb ← (1<<w)>> 6 // align PA[12] into EntryLo* register bit 6
        pw_EntryLo0 ← t and not lsb # lsb=0 even page; note FILL fields are 0
        pw_EntryLo1 ← t or lsb # lsb=1 odd page
        elseif (PWCtl$_{DPH}$ = 1)
        // Dual Pages - figure out whether even or odd page loaded first
            OddPageBit = (1 << PWField$_{MDI}$)
            if (vAddr and OddPageBit)
                pw_EntryLo1 ← t
            else
                pw_EntryLo0 ← t
            endif
        // load second PTE from directory for odd TLB page
            vAddr2 ←vAddr xor (1 << (NativeShift + PWSize$_{PTEW}$)
            (pAddr2, CCA2) ←AddressTranslation(vAddr2, DATA, LOAD, KERNEL)
            t  ← LoadMemory(CCA2, DSize, pAddr2, vAddr2, DATA)
            t  ← t >> PWField$_{PTEI}$ - 2 // right-shift entire PTE
            t  ← ROTRIGHT(t, 2) // 32-bit rotate to place RI/XI bits
            if (vAddr and OddPageBit)
                pw_EntryLo0 ← t
            else
                pw_EntryLo1 ← t
            endif
        else
            goto ERROR
```

```
         endif
         goto REFILL
      else
         vAddr ← t
      endif
   endif

   # Leaf Level Page Table - First half of PTE pair
   vAddr        ← vAddr or PToffset0
   (pAddr, CCA) ← AddressTranslation(vAddr, DATA, LOAD, KERNEL)
   temp0        ← LoadMemory(CCA, DSize, pAddr, vAddr, DATA)

   # Leaf Level Page Table - Second half of PTE pair
   vAddr        ← vAddr or PToffset1
   (pAddr, CCA) ← AddressTranslation(vAddr, DATA, LOAD, KERNEL)
   temp1        ← LoadMemory(CCA, DSize, pAddr, vAddr, DATA)

   # Load Page Table Entry pair into TLB
   temp0        ← temp0 >> $PWField_{PTEI}$ - 2 // right-shift entire PTE
   pw_EntryLo0  ← ROTRIGHT(temp0, 2) // 32-bit rotate to place RI/XI bits

   temp1        ← temp1 >> $PWField_{PTEI}$ - 2 // right-shift entire PTE
   pw_EntryLo1  ← ROTRIGHT(temp1, 2) // 32-bit rotate to place RI/XI bits

REFILL:
   found ← 1
   m ← (1<<$PWField_{PTI}$)-1

   if (HugePage) then
      # Non-power-of-4 page size halved to provide power-of-4 page size.
      # 1st step: Halve page size (1<<(w-1))

      switch ({HgPgBDHit,HgPgGDHit,HgPgUDHit,HgPgMDHit})
         case 1000
            m ← (1<<($PWField_{BDI}$))-1
         case 0100
            m ← (1<<($PWField_{GDI}$))-1
         case 0010
            m ← (1<<($PWField_{UDI}$))-1
         case 0001
            m ← (1<<($PWField_{MDI}$))-1
      end switch
   endif
   # 2nd step: Normalize mask field to 4KB as smallest base (>>12)
   $pw\_PageMask_{Mask}$ ←m>>12


# The hardware page walker inserts a page into the TLB in a manner
# identical to a TLBWR instruction as executed by the software refill handler
   pw_EntryHi = ( vaddr and not 0xfff )| $EntryHi_{ASID}$
   TLBWriteRandom(pw_EntryHi, pw_EntryLo0, pw_EntryLo1, pw_PageMask)
   return(found)
   # If an error/exception condition is detected on a page table
   # walk memory access, this function exits with found=0.
   #
   OnError:
      return(0)
endsub
```

If a page is marked invalid, the hardware refill handler will still fill the page into the TLB. Software can point to invalid PTEs to represent regions that are not mapped. When the Software attempts to use the invalid TLB entry, a TLB invalid exception will be generated.

Chapter 5

Common Device Memory Map

MIPS processors may include memory-mapped IO devices that are packaged as part of the CPU. An example is the Fast Debug Channel, which is a UART-like communication device that uses the EJTAG probe pins to move data to the external world.

The Common Device Memory Map (CDMM) is a region of physical address space that is reserved for mapping IO device configuration registers within a MIPS processor. The CDMM helps aggregate various device mappings into one area, preventing fragmentation of the memory address space. It also enables the use of access control and memory address translation mechanisms for these device registers. The CDMM occupies a maximum of 32KB in the physical address map.

The CMDMM is an optional feature of the architecture. Software detects if CDMM is implemented by reading the *Config3*$_{CDMM}$ register field (bit 3).

Two blocks are defined for the CDMM -

- *CDMMBase* - A new Coprocessor 0 register that sets the base physical address of the CDMM
- CDMM Access Control and Device Register Block - The 32KB CDMM region is divided into smaller 64-byte aligned blocks called ‘Device Register Blocks’ (DRBs). Each block has access control and status information in access control and status registers (ACSRs), followed by IO device registers.

For implementations that have multiple VPEs, the IO devices and their ACSRs are instantiated once per VPE, but the *CDMMBase* register is shared among the VPEs.

Implementations are not required to maintain cache coherence for the CDMM region. For that reason, the memory mapped registers located within this region must be accessed only using uncached memory transactions. Accessing these register using a cacheable CCA may result in **UNPREDICTABLE** behavior.

Each of these blocks are now described in detail.

5.1 CDMMBase Register

The physical base address for the CDMM facility is defined by a coprocessor 0 register called *CDMMBase,* (CP0 register 15, select 2). This address must be aligned to a 32KB boundary.

On a 32-bit core with a TLB-based MMU, this region would most likely be mapped to the lower 512MB of physical memory, allowing kernel-mode unmapped, uncached access via kseg1. User-mode access could be allowed through a TLB mapping using an uncached coherency.

On cores that use a FMT MMU, the region would most likely be mapped to the lower 512MB and made accessible via kernel mode. Alternatively, if user-mode access is allowed, this region could be mapped to correspond to the kuseg physical address segment.

On cores that use a BAT MMU, if only kernel mode access is allowed, the region would be mapped to a physical address region reachable through kseg1 or kseg2/3 (using uncached coherency). If user mode access is allowed, the useg BAT entry must use an uncached coherency.

Please refer to Section 9.39 on page 201 for the description of the *CDMMBase* register.

5.2 CDMM - Access Control and Device Register Blocks

The CDMM is divided into 64-byte aligned segments named ‘Device Register Blocks’ (DRBs), Each device occupies at least one DRB. If a device needs additional address space, it can occupy multiple contiguous 64-byte blocks, eg. multiple DRBs which are adjacent in the physical address map. For each device, device type identification and access control information is located in the DRB allocated for the device with the lowest physical address.

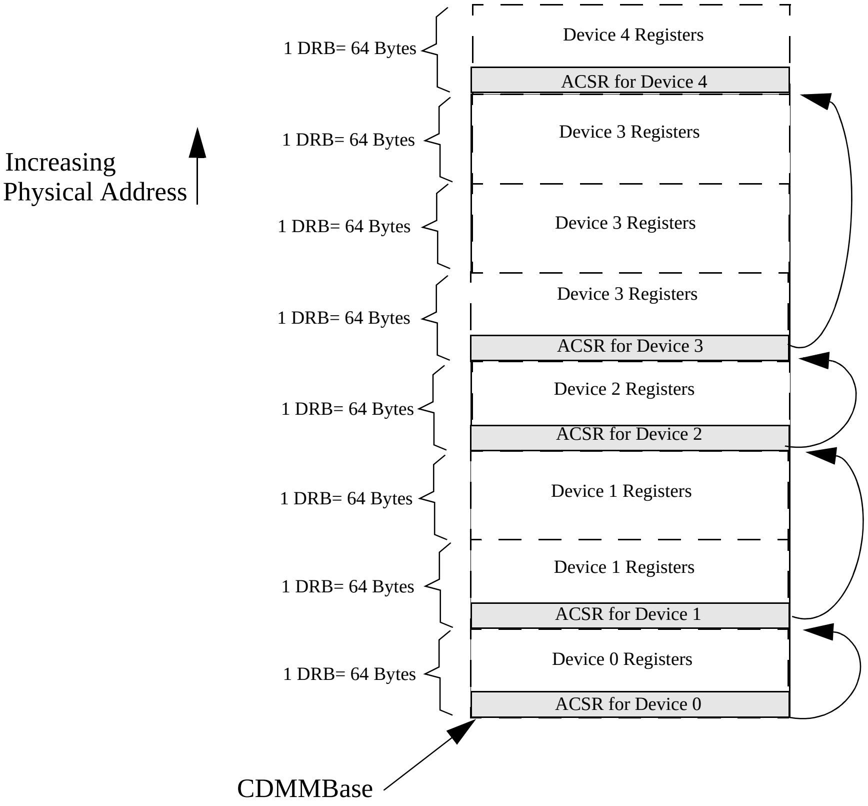
Access control information is specified via ‘Access Control and Status Registers’ (ACSRs) that are found at the start of the DRB allocated for the device with the lowest physical address. The ACSR for a device holds the size of the IO device, and hence also act as a pointer to the start of the next device and its’ ACSR. ACSRs are only accessible in kernel mode. The ACSR is followed by the data/control registers for the IO device. Figure 5.1 shows the organization of the CDMM.

Reading any of the IO device registers in either usermode or supervisor mode when such accesses are not allowed, results in all zeros being returned. Writing any of the IO device registers in either usermode or supervisor mode when such accesses are not allowed, results in the write being ignored and the register not being modified. Reading any of the ACSR registers while not in kernel mode results in all zeros being returned. Writing any of the ACSR registers while not in kernel mode results in the write being ignored and the ACSR not being modified.

Since the ACSR act as a pointer that can only increment, the devices must be allocated in the memory space in a specific manner. The first device must be located at the address pointed by the *CDMMBase* register and any subsequent device is allocated in the next available adjacent DRB.

If the *CI* bit is set in the *CDMMBASE* register, the first DRB of the CDMM (at offset 0x0 from the CDMMBase) is reserved for implementation specific use.

Figure 5.1 Example Organization of the CDMM



5.2.1 Access Control and Status Registers

The first DRB of a device has 8 bytes of access control address space allocated to it. These 8 bytes can be considered to be two 32-bit registers (on a 32-bit or 64-bit core), or a single 64-bit register (on a 64-bit core). In revision 1.00 of the CDMM, only the lower 32-bits hold access control and status information. The control/status register can be accessed in kernel mode only. Reading this register while not in kernel mode results in all zeros being returned. Writing this register while not in kernel mode results in the write being ignored and the register not being modified.

Figure 5.2 has the format of an Access Control and Status register (shown as a 64-bit register), and Table 5.1 describes the register fields.

Figure 5.2 Access Control and Status Register

63 32	31 24	23 22	21 16	15 12	11 4	3	2	1	0
0	DevType	0	DevSize	DevRev	0	Uw	Ur	Sw	Sr

Table 5.1 Access Control and Status Register Field Descriptions

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
DevType	31:24	This field specifies the type of device. A non-zero value indicates the type of device. A zero value indicates the absence of a device.	R	Preset	Required

Table 5.1 Access Control and Status Register Field Descriptions

Fields					
Name	Bits	Description	Read / Write	Reset State	Compliance
DevSize	21:16	This field specifies the number of extra 64-byte blocks allocated to this device. A value of 0 indicates that only one 64-byte block is allocated. This also determines the location of the next device block. A device is limited to 4KB of memory.	R	Preset	Required
DevRev	15:12	This field specifies the revision of device. This field is combined with the DevType field to denote the specific device revision.	R	Preset	Required
Uw	3	This bit indicates if user-mode write access to this device is enabled. A value of 1 indicates that access is enabled. A value of 0 indicates that access is disabled. An attempt to write to the device while in user mode with access disabled is ignored.	R/W	0	Required
Ur	2	This bit indicates if user-mode read access to this device is enabled. A value of 1 indicates that access is enabled. A value of 0 indicates that access is disabled. An attempt to read from the device while in user mode with access disabled is ignored.	R/W	0	Required
Sw	1	This bit indicates if supervisor-mode write access to this device is enabled. A value of 1 indicates that access is enabled. A value of 0 indicates that access is disabled. An attempt to write to the device while in supervisor mode with access disabled is ignored.	R/W	0	Required
Sr	0	This bit indicates if supervisor-mode read access to this device is enabled. A value of 1 indicates that access is enabled. A value of 0 indicates that access is disabled. An attempt to read from the device while in supervisor mode with access disabled is ignored.	R/W	0	Required
0	63:32, 11:4	Reserved for future use. Ignored on write; returns zero on read.	R	0	Required

Chapter 6

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 $Status_{BEV}$ 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.

6.1 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.

6.1.1 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 6.1 shows the current interrupt mode of the processor as a function of the coprocessor 0 register fields that can affect the mode.

Table 6.1 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
0	1	≠0	x	1	External Interrupt Controller

Table 6.1 Interrupt Modes

$Status_{BEV}$	$Cause_{IV}$	$IntCtl_{VS}$	$Config3_{VINT}$	$Config3_{VEIC}$	Interrupt Mode
0	1	≠0	0	0	Not Allowed - $IntCtl_{VS}$ is zero if neither Vectored Interrupt nor External Interrupt Controller mode are implemented.
“x” denotes don’t care					

6.1.1.1 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 $Cause_{IV}$ = 0) or vector offset 0x200 (if $Cause_{IV}$ = 1). This mode is in effect if any of the following conditions are true:

- $Cause_{IV}$ = 0
- $Status_{BEV}$ = 1
- $IntCtl_{VS}$ = 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 6.2.

Table 6.2 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	$Cause_{IP7}$ and $Status_{IM7}$
Hardware Interrupt	HW4	$Cause_{IP6}$ and $Status_{IM6}$
	HW3	$Cause_{IP5}$ and $Status_{IM5}$
	HW2	$Cause_{IP4}$ and $Status_{IM4}$
	HW1	$Cause_{IP3}$ and $Status_{IM3}$
	HW0	$Cause_{IP2}$ and $Status_{IM2}$
Software Interrupt	SW1	$Cause_{IP1}$ and $Status_{IM1}$
	SW0	$Cause_{IP0}$ and $Status_{IM0}$

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

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

/*
 * Each interrupt processing routine processes a specific interrupt, analogous
 * to those reached in VI or EIC interrupt mode. Since each processing routine
 * is dedicated to a particular interrupt line, it has the context to know
 * which line was asserted.  Each processing routine may need to look further
 * to determine the actual source of the interrupt if multiple interrupt requests
 * are ORed together on a single IP line. Once that task is performed, the
 * interrupt may be processed in one of two ways:
 *
 * - Completely at interrupt level (e.g., a 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 StatusIM bits so that “lower” priority interrupts are
 *    also disabled. The NestedInterrupt routine below is an example of this type.
 */

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

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

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

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

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

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

6.1.1.2 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:

- $\mathit{Config3}_{VInt} = 1$
- $\mathit{Config3}_{VEIC} = 0$
- $\mathit{IntCtl}_{IVS} \neq 0$
- $\mathit{Cause}_{IV} = 1$
- $\mathit{Status}_{BEV} = 0$

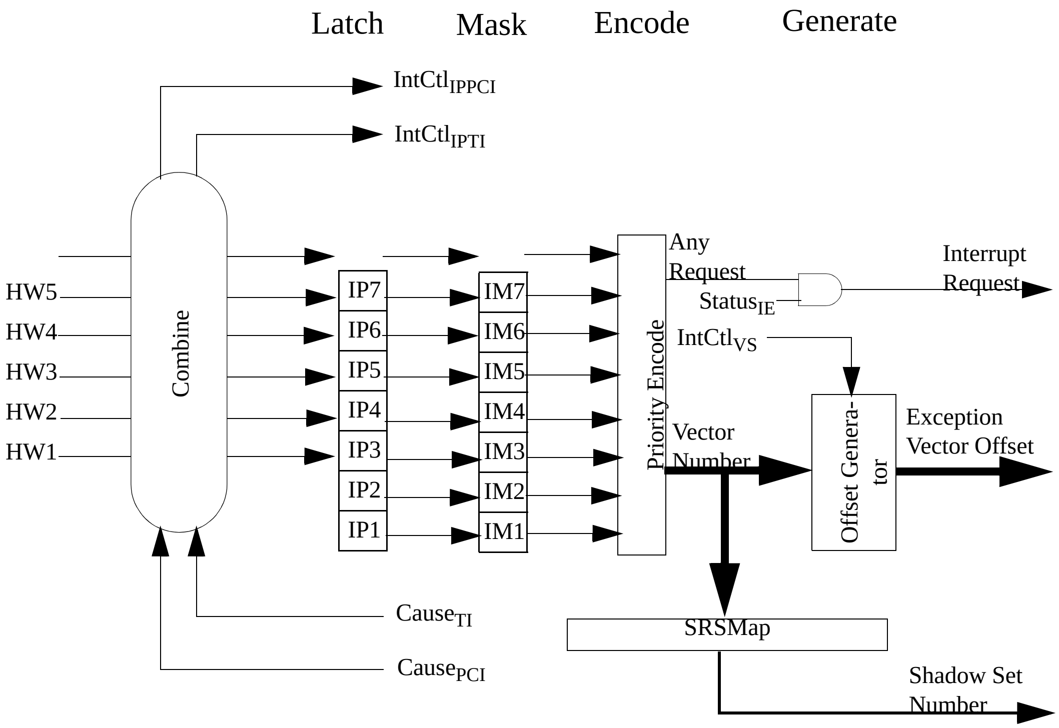
In VI interrupt mode, the six hardware interrupts are interpreted as individual hardware interrupt requests. The timer and performance counter interrupts are combined in an implementation-dependent way with the hardware interrupts (with the interrupt with which they are combined indicated by $IntCtl_{IPTI}$ and $IntCtl_{IPPCI}$, 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 $\mathrm{Status}_{\mathrm{IM}}$ bits. If any of these values is 1, and if interrupts are enabled ($Status_{IE}$ = 1, $Status_{EXL}$ = 0, and $Status_{ERL}$ = 0), an interrupt is signaled and a priority encoder scans the values in the order shown in Table 6.3.

Table 6.3 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	$\mathrm{Cause}_{\mathrm{IP7}}$ and $\mathrm{Status}_{\mathrm{IM7}}$	7
		HW4	$\mathrm{Cause}_{\mathrm{IP6}}$ and $\mathrm{Status}_{\mathrm{IM6}}$	6
		HW3	$\mathrm{Cause}_{\mathrm{IP5}}$ and $\mathrm{Status}_{\mathrm{IM5}}$	5
		HW2	$\mathrm{Cause}_{\mathrm{IP4}}$ and $\mathrm{Status}_{\mathrm{IM4}}$	4
		HW1	$\mathrm{Cause}_{\mathrm{IP3}}$ and $\mathrm{Status}_{\mathrm{IM3}}$	3
		HW0	$\mathrm{Cause}_{\mathrm{IP2}}$ and $\mathrm{Status}_{\mathrm{IM2}}$	2
	Software	SW1	$\mathrm{Cause}_{\mathrm{IP1}}$ and $\mathrm{Status}_{\mathrm{IM1}}$	1
Lowest Priority		SW0	$\mathrm{Cause}_{\mathrm{IP0}}$ and $\mathrm{Status}_{\mathrm{IM0}}$	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 6.1.

Figure 6.1 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 cannot 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 SRSCtlPSS 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 */
```

6.1.1.3 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. There are several implementation options available for the vector offset:

1. The first option is to treat the RIPL value as the vector number for the processor.

2. The second option is to send a separate vector number along with the RIPL to the processor.

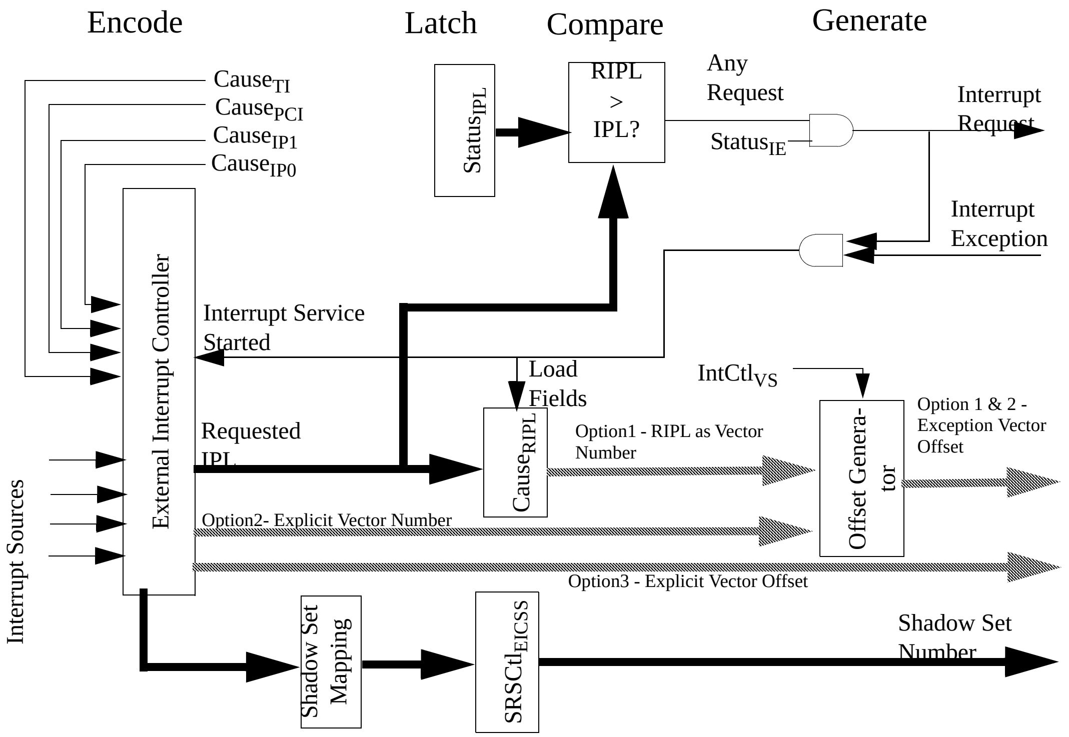
3. A third option is to send an entire vector offset along with the RIPL to the processor.

Status$_{IPL}$ (which overlays *Status*$_{IM7..IM2}$) 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 *Status*$_{IPL}$ to determine if the requested interrupt has higher priority than the current IPL. If RIPL is strictly greater than *Status*$_{IPL}$, and interrupts are enabled (*Status*$_{IE}$ = 1, *Status*$_{EXL}$ = 0, and *Status*$_{ERL}$ = 0) an interrupt request is signaled to the pipeline. When the processor starts the interrupt exception, it loads RIPL into *Cause*$_{RIPL}$ (which overlays *Cause*$_{IP7..IP2}$) and signals the external interrupt controller to notify it that the request is being serviced. Because *Cause*$_{RIPL}$ 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 *IntCtl*$_{VS}$ 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 *Cause*$_{RIPL}$, it also loads the GPR shadow set number into *SRSCtl*$_{EICSS}$, which is copied to *SRSCtl*$_{CSS}$ when the interrupt is serviced.

The operation of EIC interrupt mode is shown pictorially in Figure 6.2.

Figure 6.2 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 cannot 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 SRSCtlPSS 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.
     */
```

6.1.2 Generation of Exception Vector Offsets for Vectored Interrupts

For vectored interrupts (in either VI or EIC interrupt mode - options 1 & 2), 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 6.4 shows the exception vector offset for a representative subset of the vector numbers and values of the *IntCtl*$_{VS}$ field.

Table 6.4 Exception Vector Offsets for Vectored Interrupts

Vector Number	Value of IntCtl$_{VS}$ Field				
	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
• • •					
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:

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

6.1.2.1 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 107. 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 6.5 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 6.5 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	$IP_{1..0}$
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 */
```

6.2 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.

6.2.1 Exception Priority

Table 6.6 lists all possible exceptions, and the relative priority of each, highest to lowest.

Table 6.6 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.	
TLB Execute-Inhibit	An instruction fetch matched a valid TLB entry which had the XI bit set.	
Cache Error - Instruction fetch	A cache error occurred on an instruction fetch.	
Bus Error - Instruction fetch	A bus error occurred on an instruction fetch.	
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

Table 6.6 Priority of Exceptions

Exception	Description	Type
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 Read-Inhibit	A data read access matched a valid TLB entry whose RI bit is set.	
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	

The “Type” column of Table 6.7 describes the type of exception. Table 6.8 explains the characteristics of each exception type.

Table 6.7 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.
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.

6.2.2 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 (and subsequent releases), software is allowed to specify the vector base address via the *EBase* register for exceptions that occur when $Status_{BEV}$ equals 0. Table 6.8 gives the vector base address as a function of the exception and whether the *BEV* bit is set in the *Status* register. Table 6.9 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 (and subsequent releases), Table 6.4 gives the offset from the base address in the case where $Status_{BEV} = 0$ and $Cause_{IV} = 1$. For implementations of Release 1 of the architecture in which $Cause_{IV} = 1$, the vector offset is as if $IntCtl_{VS}$ were 0.

Table 6.10 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 $IntCtl_{VS}$ is 0.

In Release 2 of the Architecture (and subsequent releases), software must guarantee that $EBase_{15..12}$ 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 6.8 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 6.9 Exception Vector Offsets

Exception	Vector Offset
TLB Refill, *EXL* = 0	`0x000`

Table 6.9 Exception Vector Offsets

Exception	Vector Offset
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 6.10 Exception Vectors

Exception	Status$_{BEV}$	Status$_{EXL}$	Cause$_{IV}$	EJTAG ProbTrap	Vector For Release 2 Implementations, assumes that EBase retains its reset state and that IntCtl$_{VS}$ = 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

6.2.3 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 9.39 on page 186). 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 6.11 shows the value stored in each of the CP0 PC registers, including *EPC*. For implementations of Release 2 of the Architecture if *Status*$_{BEV}$ =

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 6.11 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
               if (EIC_option1)
                  VecNum ← Cause$_{RIPL}$
               elseif (EIC_option2)
                  VecNum ← EIC_VecNum_Signal
               endif
               NewShadowSet ← SRSCtl$_{EICSS}$
            else
               VecNum ← VIntPriorityEncoder()
               NewShadowSet ← SRSMap$_{IPL \times 4+3..IPL \times 4}$
            endif
            if (EIC_option3)
               vectorOffset ← EIC_VectorOffset_Signal
            else
               vectorOffset ← 0x200 + (VecNum × (IntCtl$_{VS}$ || 0b00000))
            endif
         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 */
```

6.2.4 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.

6.2.5 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 6.11. 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$_{MaskX}$ ← 0           # 1KB page support implemented
PageGrain$_{ESP}$ ← 0            # 1KB page support implemented
Wired ← 0
HWREna ← 0
EntryHi$_{VPN2X}$ ← 0            # 1KB page support implemented
Status$_{RP}$ ← 0
Status$_{BEV}$ ← 1
Status$_{TS}$ ← 0
```

```
Status$_{SR}$ ← 0
Status$_{NMI}$ ← 0
Status$_{ERL}$ ← 1
IntCtl$_{VS}$ ← 0
SRSCtl$_{HSS}$ ← HighestImplementedShadowSet
SRSCtl$_{ESS}$ ← 0
SRSCtl$_{PSS}$ ← 0
SRSCtl$_{CSS}$ ← 0
SRSMap ← 0
Cause$_{DC}$ ← 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
```

6.2.6 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 6.11.

- PC is loaded with `0xBFC0 0000`.

Cause Register ExcCode Value

None

Additional State Saved

None

Entry Vector Used

Reset (0xBFC0 0000)

Operation

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

6.2.7 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 6.11.
- 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
```

6.2.8 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.If the Hardware Page Table Walker feature is implemented and the Directory-level Huge page feature is supported and the Dual Page method is also supported, and if the first accessed PTE entry has PTEVld bit set and the second accessed PTE entry has PTEVld bit clear.

***Cause* Register ExcCode Value**

MCheck (See Table 9.40 on page 189)

Additional State Saved

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

If there are multiple causes for the machine check exception, then the *PageGrain*$_{\text{MCCause}}$ register field is used to distinguish which condition caused the exception.

Entry Vector Used

General exception vector (offset 0x180)

6.2.9 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 9.40 on page 189.

Additional State Saved

Register State	Value
BadVAddr	failing address
$\text{Context}_{\text{VPN2}}$	UNPREDICTABLE
$\text{EntryHi}_{\text{VPN2}}$	UNPREDICTABLE
EntryLo0	UNPREDICTABLE
EntryLo1	UNPREDICTABLE

Entry Vector Used

General exception vector (offset 0x180)

6.2.10 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 9.40 on page 189.

Additional State Saved

Register State	Value
BadVAddr	Failing address
Context	If $\mathit{Config3}_{\mathit{CTXTC}}$ bit is set, then the bits of the *Context* register corresponding to the set bits of the *VirtualIndex* field of the *ContextConfig* register are loaded with the high-order bits of the virtual address that missed. If $\mathit{Config3}_{\mathit{CTXTC}}$ bit is clear, then 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

6.2.11 Execute-Inhibit Exception

An Execute-Inhibit exception occurs when the virtual address of an instruction fetch matches a TLB entry whose *XI* bit is set. This exception type can only occur if the *XI* bit is implemented within the TLB and is enabled, this is denoted by the *PageGrain*$_{XIE}$ bit.

***Cause* Register ExcCode Value**

if *PageGrain*$_{IEC}$ == 0 TLBL

if *PageGrain*$_{IEC}$ == 1 TLBXI

See Table 9.40 on page 189.

Additional State Saved

Register State	Value
BadVAddr	Failing address
Context	If *Config3*$_{CTXTC}$ bit is set, then the bits of the *Context* register corresponding to the set bits of the *VirtualIndex* field of the *ContextConfig* register are loaded with the high-order bits of the virtual address that missed. If *Config3*$_{CTXTC}$ bit is clear, then 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)

6.2.12 Read-Inhibit Exception

An Read-Inhibit exception occurs when the virtual address of a memory load reference matches a TLB entry whose RI bit is set. This exception type can only occur if the RI bit is implemented within the TLB and is enabled, this is denoted by the *PageGrain*$_{RIE}$ bit. MIPS16 PC-relative loads are a special case and are not affected by the RI bit.

***Cause* Register ExcCode Value**

if *PageGrain*$_{IEC}$ == 0 TLBL

if *PageGrain*$_{IEC}$ == 1 TLBRI

See Table 9.40 on page 189.

Additional State Saved

Register State	Value
BadVAddr	Failing address
Context	If *Config3*$_{CTXTC}$ bit is set, then the bits of the *Context* register corresponding to the set bits of the *VirtualIndex* field of the *ContextConfig* register are loaded with the high-order bits of the virtual address that missed. If *Config3*$_{CTXTC}$ bit is clear, then 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)

6.2.13 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).

If the RI and XI bits are implemented within the TLB and the *PageGrain*$_{IEC}$ bit is clear, then this exception also occurs if a valid, matching TLB entry is found with the RI bit set on a memory load reference, or with the XI bit set on an instruction fetch memory reference. MIPS16 PC-relative loads are a special case and are not affected by the RI bit.

***Cause* Register ExcCode Value**

TLBL: Reference was a load or an instruction fetch

TLBS: Reference was a store

See Table 9.39 on page 186.

Additional State Saved

Register State	Value
BadVAddr	Failing address

Register State	Value
Context	If *Config3*$_{CTXTC}$ bit is set, then the bits of the *Context* register corresponding to the set bits of the *VirtualIndex* field of the *ContextConfig* register are loaded with the high-order bits of the virtual address that missed. If *Config3*$_{CTXTC}$ bit is clear, then 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)

6.2.14 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 9.39 on page 186)

Additional State Saved

Register State	Value
BadVAddr	Failing address
Context	If *Config3*$_{CTXTC}$ bit is set, then the bits of the *Context* register corresponding to the set bits of the *VirtualIndex* field of the *ContextConfig* register are loaded with the high-order bits of the virtual address that missed. If *Config3*$_{CTXTC}$ bit is clear, then 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)

6.2.15 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
```

6.2.16 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 9.40 on page 189.

Additional State Saved

None

Entry Vector Used

General exception vector (offset 0x180)

6.2.17 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 9.40 on page 189)

Additional State Saved

None

Entry Vector Used

General exception vector (offset 0x180)

6.2.18 Trap Exception

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

***Cause* Register ExcCode Value**

Tr (See Table 9.40 on page 189)

Additional State Saved

None

Entry Vector Used

General exception vector (offset 0x180)

6.2.19 System Call Exception

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

***Cause* Register ExcCode Value**

Sys (See Table 9.39 on page 186)

Additional State Saved

None

Entry Vector Used

General exception vector (offset 0x180)

6.2.20 Breakpoint Exception

A breakpoint exception occurs when a BREAK instruction is executed.

Cause Register ExcCode Value

Bp (See Table 9.40 on page 189)

Additional State Saved

None

Entry Vector Used

General exception vector (offset 0x180)

6.2.21 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" (Module/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" (Module/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" (Module/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 9.40 on page 189)

Additional State Saved

None

Entry Vector Used

General exception vector (offset 0x180)

6.2.22 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. COP2 instructions include MFC2, DMFC2, CFC2, MFHC2, MTC2, DMTC2, CTC2, MTHC2.

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 9.39 on page 186)

Additional State Saved

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

Entry Vector Used

General exception vector (offset 0x180)

6.2.23 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 9.39 on page 186)

Additional State Saved

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

Entry Vector Used

General exception vector (offset 0x180)

6.2.24 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 9.39 on page 186)

Additional State Saved

Defined by the coprocessor

Entry Vector Used

General exception vector (offset 0x180)

6.2.25 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 9.39 on page 186)

Additional State Saved

Register State	Value
Cause_{WP}	Indicates that the watch exception was deferred until after both *Status*$_{EXL}$ and *Status*$_{ERL}$ 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)

6.2.26 Interrupt Exception

The interrupt exception occurs when an enabled request for interrupt service is made. See Section 6.1 on page 73 for more information.

Register ExcCode Value

Int (See Table 9.40 on page 189)

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 7

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 SRSCtl_{HSS} field, and all shadow sets between 0 and SRSCtl_{HSS}, inclusive must be implemented. If this field is zero, only the normal GPRs are implemented.

7.1 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 SRSCtl_{CSS} is copied to SRSCtl_{PSS}, and SRSCtl_{CSS} is set to the value taken from the appropriate source. On an ERET, the value of SRSCtl_{PSS} is copied back into SRSCtl_{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 $Status_{ERL}$: NMI or cache error.
   - The exception causes entry into EJTAG Debug Mode
   - $Status_{BEV} = 1$
   - $Status_{EXL} = 1$
2. SRSCtl_{CSS} is copied to SRSCtl_{PSS}

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

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

    - The EICSS field of the *SRSCtl* register if the exception is an interrupt, Cause$_{IV}$ = 1, IntCtl$_{VSS}$ ≠ 0, and *Config3$_{VEIC}$* = 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 *Status$_{ERL}$* = 1 or *Status$_{BEV}$* = 1

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

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 (*Status$_{BEV}$* = 1).

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

7.2 Support Instructions

Table 7.1 Instructions Supporting Shadow Sets

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

Chapter 8

CP0 Hazards

8.1 Introduction

Because resources controlled via Coprocessor 0 affect the operation of various pipeline stages of a MIPS32/microMIPS32 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.

8.2 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.

8.2.1 Possible Execution Hazards

Execution hazards are those created by the execution of one instruction, and seen by the execution of another instruction. Table 8.1 lists the possible execution hazards that might exist when there are no hardware interlocks.

Table 8.1 Possible Execution Hazards

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

Table 8.1 Possible 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, TLBWI	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
EI, DI	∅	Interrupted Instruction	Status$_{\text{IE}}$, Status$_{\text{IM}}$
Other Hazards			
LL	∅	MFC0	LLAddr
MTC0	∅	CACHE	PageGrain
CACHE	∅	MFC0	TagLo

Table 8.1 Possible Execution Hazards

Producer	∅	Consumer	Hazard On
MTC0	∅	MFC0	any CoProcessor 0 register

8.2.2 Possible Instruction Hazards

Instruction hazards are those created by the execution of one instruction, and seen by the instruction fetch of another instruction. Table 8.2 lists the possible instruction hazards when there are no hardware interlocks.

Table 8.2 Possible Instruction Hazards

Producer	∅	Consumer	Hazard On
Hazards Related to the TLB			
MTC0	∅	Instruction fetch seeing the new value	$\text{EntryHi}_{\text{ASID}}$, 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	$\text{SRSCtl}_{\text{PSS}}$[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.

8.3 Hazard Clearing Instructions and Events

Table 8.3 lists the instructions designed to eliminate hazards.

Table 8.3 Hazard Clearing Instructions

Mnemonic	Function	Supported Architecture
DERET	Clear both execution and instruction hazards	EJTAG

Table 8.3 Hazard Clearing Instructions

Mnemonic	Function	Supported Architecture
EHB	Clear execution hazard	Release 2 onwards
ERET	Clear both execution and instruction hazards	All
IRET	Clear both execution and instruction hazards when not chaining to another interrupt.	MCU ASE
JALR.HB	Clear both execution and instruction hazards	Release 2 onwards
JR.HB	Clear both execution and instruction hazards	Release 2 onwards
SSNOP	Superscalar No Operation	Release 1 onwards
SYNCI[1]	Synchronize caches after instruction stream write	Release 2 onwards

[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.

8.3.1 MIPS32 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.

8.3.2 microMIPS32 Instruction Encoding

The EHB and SSNOP instructions are encoded using a variant of the NOP encoding. See the EHB and SSNOP instruction description for additional information.

Chapter 9

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.

9.1 Coprocessor 0 Register Summary

Table 9.1 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 9.1 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 9.4 on page 120	Required (TLB MMU); Optional (Others)
0	1	MVPControl	Per-processor register containing global MIPS® MT configuration data	MIPS®MT Module Specification	Required (MIPS MT Module); Optional (Others)
0	2	MVPConf0	Per-processor multi-VPE dynamic configuration information	MIPS®MT Module Specification	Required (MIPS MT Module); Optional (Others)
0	3	MVPConf1	Per-processor multi-VPE dynamic configuration information	MIPS®MT Module Specification	Optional
1	0	Random	Randomly generated index into the TLB array	Section 9.5 on page 121	Required (TLB MMU); Optional (Others)
1	1	VPEControl	Per-VPE register containing relatively volatile thread configuration data	MIPS®MT Module Specification	Required (MIPS MT Module); Optional (Others)
1	2	VPEConf0	Per-VPE multi-thread configuration information	MIPS®MT Module Specification	Required (MIPS MT Module); Optional (Others)
1	3	VPEConf1	Per-VPE multi-thread configuration information	MIPS®MT Module Specification	Optional
1	4	YQMask	Per-VPE register defining which YIELD qualifier bits may be used without generating an exception	MIPS®MT Module Specification	Required (MIPS MT Module); Optional (Others)
1	5	VPESchedule	Per-VPE register to manage scheduling of a VPE within a processor	MIPS®MT Module Specification	Optional
1	6	VPEScheFBack	Per-VPE register to provide scheduling feedback to software	MIPS®MT Module Specification	Optional

Table 9.1 Coprocessor 0 Registers in Numerical Order

Register Number	Sel[1]	Register Name	Function	Reference	Compliance Level
1	7	VPEOpt	Per-VPE register to provide control over optional features, such as cache partitioning control	MIPS®MT Module Specification	Optional
2	0	EntryLo0	Low-order portion of the TLB entry for even-numbered virtual pages	Section 9.6 on page 122	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 Module Specification	Required (MIPS MT Module); Optional (Others)
2	2	TCBind	Per-TC information about TC ID and VPE binding	MIPS®MT Module Specification	Required (MIPS MT Module); Optional (Others)
2	3	TCRestart	Per-TC value of restart instruction address for the associated thread of execution	MIPS®MT Module Specification	Required (MIPS MT Module); Optional (Others)
2	4	TCHalt	Per-TC register controlling Halt state of TC	MIPS®MT Module Specification	Required (MIPS MT Module); Optional (Others)
2	5	TCContext	Per-TC read/write storage for operating system use	MIPS®MT Module Specification	Required (MIPS MT Module); Optional (Others)
2	6	TCSchedule	Per-TC register to manage scheduling of a TC	MIPS®MT Module Specification	Optional
2	7	TCScheFBack	Per-TC register to provide scheduling feedback to software	MIPS®MT Module Specification	Optional
3	0	EntryLo1	Low-order portion of the TLB entry for odd-numbered virtual pages	Section 9.6 on page 122	Required (TLB MMU); Optional (Others)
3	7	TCOpt	Per-TC register to provide control over optional features, such as cache partitioning control	MIPS®MT Module Specification	Optional
4	0	Context	Pointer to page table entry in memory	Section 9.7 on page 131	Required (TLB MMU); Optional (Others)
4	1	ContextConfig	Context register configuration	SmartMIPS ASE Specification and Section 9.8 on page 135	Required (SmartMIPS ASE); Optional (Others)
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 Module, this is a per-TC register.	Section 9.9 on page 137	Recommended (Release 2)
4	3		XContext register configuration in 64-bit implementations		Reserved

Table 9.1 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 9.10 on page 138	Required (TLB MMU); Optional (Others)
5	1	PageGrain	Control for small page support	Section 9.11 on page 140 and SmartMIPS ASE Specification	Required (SmartMIPS ASE); Optional (Release 2)
5	2	SegCtl0	Programmable Control for Segments 0 & 1	Section 9.12 on page 145	Optional
5	3	SegCtl1	Programmable Control for Segments 2 & 3	Section 9.13 on page 145	Optional
5	4	SegCtl2	Programmable Control for Segments 4 & 5	Section 9.14 on page 145	Optional
5	5	PWBase	Page Table Base Address for Hardware Page Walker	Section 9.15 on page 149	Optional
5	6	PWField	Bit indices of pointers for Hardware Page Walker	Section 9.16 on page 149	Optional
5	7	PWSize	Size of pointers for Hardware Page Walker	Section 9.17 on page 152	Optional
6	0	Wired	Controls the number of fixed (“wired”) TLB entries	Section 9.18 on page 156	Required (TLB MMU); Optional (Others)
6	1	SRSConf0	Per-VPE register indicating and optionally controlling shadow register set configuration	MIPS®MT Module Specification	Required (MIPS MT Module); Optional (Others)
6	2	SRSConf1	Per-VPE register indicating and optionally controlling shadow register set configuration	MIPS®MT Module Specification	Optional
6	3	SRSConf2	Per-VPE register indicating and optionally controlling shadow register set configuration	MIPS®MT Module Specification	Optional
6	4	SRSConf3	Per-VPE register indicating and optionally controlling shadow register set configuration	MIPS®MT Module Specification	Optional
6	5	SRSConf4	Per-VPE register indicating and optionally controlling shadow register set configuration	MIPS®MT Module Specification	Optional
6	6	PWCtl	HW Page Walker Control	Section 9.19 on page 159	Optional
7	0	HWREna	Enables access via the RDHWR instruction to selected hardware registers	Section 9.20 on page 162	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 9.21 on page 164	Required
8	1	BadInstr	Reports the instruction which caused the most recent exception.	Section 9.22 on page 165	Optional

Table 9.1 Coprocessor 0 Registers in Numerical Order

Register Number	Sel[1]	Register Name	Function	Reference	Compliance Level
8	2	BadInstrP	Reports the branch instruction if a delay slot caused the most recent exception.	Section 9.23 on page 167	Optional
9	0	Count	Processor cycle count	Section 9.24 on page 168	Required
9	6-7		Available for implementation-dependent user	Section 9.25 on page 168	implementation-dependent
10	0	EntryHi	High-order portion of the TLB entry	Section 9.26 on page 169	Required (TLB MMU); Optional (Others)
10	4	GuestCtl1	GuestID of virtualized Guest	MIPS® VZE Module Specification	Required (MIPS VZE Module ; Optional (Others)
10	5	GuestCtl2	Guest Interrupt Control	MIPS® VZE Module Specification	Required (MIPS VZE Module ; Optional (Others)
10	6	GuestCtl3	Guest Shadow Register Set Control	MIPS® VZE Module Specification	Required (MIPS VZE Module ; Optional (Others)
11	0	Compare	Timer interrupt control	Section 9.27 on page 171	Required
11	4	GuestCtl0Ext	Extension of GuestCtl0	MIPS® VZE Module Specification	Required (MIPS VZE Module ; Optional (Others)
11	6-7		Available for implementation-dependent user	Section 9.28 on page 171	implementation-dependent
12	0	Status	Processor status and control	Section 9.29 on page 172	Required
12	1	IntCtl	Interrupt system status and control	Section 9.30 on page 179	Required (Release 2)
12	2	SRSCtl	Shadow register set status and control	Section 9.31 on page 182	Required (Release 2)
12	3	SRSMap	Shadow set IPL mapping	Section 9.32 on page 185	Required (Release 2 and shadow sets implemented)
12	4	View_IPL	Contiguous view of IM and IPL fields.	MIPS® MCU ASE Specification	Required (MIPS MCU ASE); Optional (Others)
12	5	SRSMap2	Shadow set IPL mapping	MIPS® MCU ASE Specification	Required (MIPS MCU ASE); Optional (Others)
12	6	GuestCtl0	Control of Virtualized Guest OS	MIPS® VZE Module Specification	Required (MIPS VZE Module); Optional (Others)

Table 9.1 Coprocessor 0 Registers in Numerical Order

Register Number	Sel[1]	Register Name	Function	Reference	Compliance Level
12	7	GTOffset	Guest Timer Offset	MIPS® VZE Module Specification	Required (MIPS VZE Module); Optional (Others)
13	0	Cause	Cause of last general exception	Section 9.33 on page 186	Required
13	4	View_RIPL	Contiguous view of IP and RIPL fields.	MIPS® MCU ASE Specification	Required (MIPS MCU ASE); Optional (Others)
13	5	NestedExc	Nested exception Support - EXL, ERL values at current exception	Section 9.34 on page 191	Optional
14	0	EPC	Program counter at last exception	Section 9.35 on page 192	Required
14	2	NestedEPC	Nested exception Support - Program Counter at current exception	Section 9.36 on page 195	Optional
15	0	PRId	Processor identification and revision	Section 9.37 on page 196	Required
15	1	EBase	Exception vector base register	Section 9.38 on page 198	Required (Release 2)
15	2	CDMMBase	Common Device Memory Map Base register	Section 9.39 on page 201	Optional
15	3	CMGCRBase	Coherency Manager Global Control Register Base register	Section 9.40 on page 203	Optional
16	0	Config	Configuration register	Section 9.41 on page 204	Required
16	1	Config1	Configuration register 1	Section 9.42 on page 207	Required
16	2	Config2	Configuration register 2	Section 9.43 on page 211	Optional
16	3	Config3	Configuration register 3	Section 9.44 on page 214	Optional
16	3	Config4	Configuration register 4	Section 9.45 on page 221	Optional
16	4	Config5	Configuration register 5	Section 9.46 on page 227	Optional
16	6-7		Available for implementation-dependent user	Section 9.47 on page 231	implementation-dependent
17	0	LLAddr	Load linked address	Section 9.48 on page 232	Optional
18	0-n	WatchLo	Watchpoint address	Section 9.51 on page 243	Optional
19	0-n	WatchHi	Watchpoint control	Section 9.52 on page 245	Optional
20	0		XContext in 64-bit implementations		Reserved
21	all		Reserved for future extensions.		Reserved

Table 9.1 Coprocessor 0 Registers in Numerical Order

Register Number	Sel[1]	Register Name	Function	Reference	Compliance Level
22	all		Available for implementation-dependent use	Section 9.53 on page 247	implementation-dependent
23	0	Debug	EJTAG Debug register	EJTAG Specification	Optional
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 9.57 on page 251	Recommended
26	0	ErrCtl	Parity/ECC error control and status	Section 9.58 on page 255	Optional
27	0-3	CacheErr	Cache parity error control and status	Section 9.59 on page 256	Optional
28	even selects	TagLo	Low-order portion of cache tag interface	Section 9.60 on page 257	Required (Cache)
28	odd selects	DataLo	Low-order portion of cache data interface	Section 9.61 on page 259	Optional
29	even selects	TagHi	High-order portion of cache tag interface	Section 9.62 on page 260	Required (Cache)
29	odd selects	DataHi	High-order portion of cache data interface	Section 9.63 on page 261	Optional
30	0	ErrorEPC	Program counter at last error	Section 9.64 on page 262	Required
31	0	DESAVE	EJTAG debug exception save register	EJTAG Specification	Optional
31	2-7	KScratch*n*	Scratch Registers for Kernel Mode	Section 9.66 on page 265	Optional; KScratch1 at select 2 and KScratch2 at select 3 are recommended.

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

9.2 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 9.2 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.

9.3 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.

9.4 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 9.1 shows the format of the *Index* register; Table 9.3 describes the *Index* register fields.

Figure 9.1 Index Register Format

<table>
<tr><td>31</td><td></td><td>n</td><td>n-1</td><td></td><td>0</td></tr>
<tr><td>P</td><td colspan="2">0</td><td colspan="3">Index</td></tr>
</table>

Table 9.3 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>

9.5 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 9.2 shows the format of the *Random* register; Table 9.4 describes the *Random* register fields.

Figure 9.2 Random Register Format

31 n	n-1 0
0	Random

Table 9.4 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

9.6 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 *PFN* and *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 to 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 9-3 shows the format of the *EntryLo0* and *EntryLo1* registers; Table 9.5 describes the *EntryLo0* and *EntryLo1* register fields.

For Release 2 of the Architecture, Figure 9-4 shows the format of the *EntryLo0* and *EntryLo1* registers; Table 9.6 describes the *EntryLo0* and *EntryLo1* register fields. Release 2 of the architecture added support for physical address spaces beyond 36 bits in range and support for 1KB pages.

For Release 3 of the Architecture, Figure 9-5 shows the format of the *EntryLo0* and *EntryLo1* registers; Table 9.8 describes the *EntryLo0* and *EntryLo1* register fields. Release 3 of the architecture added support for Read-Inhibit and Execute-Inhibit page protection bits. In Release 5 of the Architecture, *EntryLo0* and *EntryLo1* registers may be optionally extended by 32 bits to support a physical address size greater than 36 bits. A 36-bit PAE is supported in the base architecture; the capability of providing greater than a 36-bit PA in MIPS32 is termed Extended Physical Address (XPA). The practical lower limit of XPA is 40 bits, while the natural upper limit is 59 bits, as determined by the MIPS64 Architecture. The size of XPA within the range of 37 bits and 59 bits is implementation-dependent.

Software can access the 32-bit extension with the new MTHC0 and MFHC0 instructions defined in Release 5.

Software can detect support for XPA and for the *EntryLo0* and *EntryLo1* formats shown in Figure 9-6 by reading *Config3*$_{LPA}$. Software can enable XPA using *PageGrain*$_{ELPA}$.

Figure 9-3 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 9.5 EntryLo0, EntryLo1 Register Field Descriptions in Release 1 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 9.7 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 9.7 for more information.	R/W	Undefined	Required
C	5..3	Cacheability and Coherency Attribute of the page. See Table 9.2 below.	R/W	Undefined	Required
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 9-4 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 9.6 EntryLo0, EntryLo1 Register Field Descriptions in Release 2 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 9.7 for more information.	R	0	Required
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 9.7 for more information.	R/W	Undefined	Required
C	5..3	The definition of this field is unchanged from Release 1. See Table 9.5 above and Table 9.2 below.	R/W	Undefined	Required
D	2	The definition of this field is unchanged from Release 1. See Table 9.5 above.	R/W	Undefined	Required
V	1	The definition of this field is unchanged from Release 1. See Table 9.5 above.	R/W	Undefined	Required
G	0	The definition of this field is unchanged from Release 1. See Table 9.5 above.	R/W	Undefined	Required (TLB MMU)

Table 9.7 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 9.7 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	$36 \geq PABITS > 12$	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	$34 \geq PABITS > 10$	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

Figure 9-5 EntryLo0, EntryLo1 Register Format in Release 3 of the Architecture

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

Table 9.8 EntryLo0, EntryLo1 Register Field Descriptions in Release 3 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 9.7 for more information.	R	0	Required if *RI* and *XI* fields are not implemented.
RI	31	Read Inhibit. If this bit is set in a TLB entry, an attempt, **other than a MIPS16 PC-relative load,** to read data on the virtual page causes a TLB Invalid or a TLBRI exception, even if the *V* (Valid) bit is set. The *RI* bit is writable only if the *RIE* bit of the *PageGrain* register is set. If the *RIE* bit of *PageGrain* is not set, the *RI* bit of *EntryLo0*/*EntryLo1* is set to zero on any write to the register, regardless of the value written. This bit is optional and its existence is denoted by the $Config3_{RXI}$ or $Config3_{SM}$ register fields.	R/W	0	Required by SmartMIPS ASE; Optional otherwise If not implemented, this bit location is part of the *Fill* field.
XI	30	Execute Inhibit. If this bit is set in a TLB entry, an attempt to fetch an instruction or to load MIPS16 PC-relative data from the virtual page causes a TLB Invalid or a TLBXI exception, even if the *V* (Valid) bit is set. The *XI* bit is writable only if the *XIE* bit of the *PageGrain* register is set. If the *XIE* bit of *PageGrain* is not set, the *XI* bit of *EntryLo0*/*EntryLo1* is set to zero on any write to the register, regardless of the value written. This bit is optional and its existence is denoted by the $Config3_{RXI}$ or $Config3_{SM}$ register fields.	R/W	0	Required by SmartMIPS ASE; Optional otherwise If not implemented, this bit location is part of the *Fill* field.

Table 9.8 EntryLo0, EntryLo1 Register Field Descriptions in Release 3 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 9.7 for more information.	R/W	Undefined	Required
C	5..3	The definition of this field is unchanged from Release 1. See Table 9.5 above and Table 9.2 below.	R/W	Undefined	Required
D	2	The definition of this field is unchanged from Release 1. See Table 9.5 above.	R/W	Undefined	Required
V	1	The definition of this field is unchanged from Release 1. See Table 9.5 above.	R/W	Undefined	Required
G	0	The definition of this field is unchanged from Release 1. See Table 9.5 above.	R/W	Undefined	Required (TLB MMU)

Figure 9-6 applies to Table 10, specifically to MIPS32 support for XPA (PA > 36 bits), and it shows the natural upper limit of XPA. If only 40-bit XPA is supported, the most-significant bit of *PFNX* is *EntryLo0[35]* and *EntryLo1[35].*

Figure 9-6 EntryLo0, EntryLo1 Register Format in Release 5

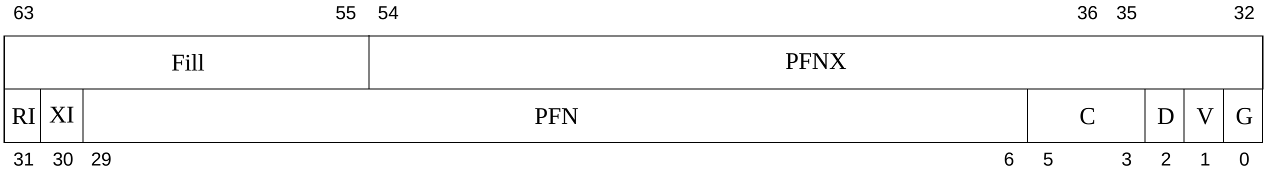


Table 10: EntryLo0, EntryLo1 Register Field Descriptions in Release 5 of the Architecture

Fields					
Name	**Bits**	**Description**	**Read / Write**	**Reset State**	**Compliance**
Fill	63..55	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*.	R	0	Required for XPA; Optional otherwise
PFNX	54..32	Page Frame Number Extension. If the processor is enabled to support XPA ($Config3_{LPA}$ =1 and $PageGrain_{ELPA}$ =1) this field is concatenated with the *PFN* field to form the full page frame number corresponding to the physical address, thereby providing up to 59 bits of physical address. If the processor is enabled to support 1KB pages ($Config3_{SP}$ = 1 and $PageGrain_{ESP}$ =1), the combined *PFNX* || *PFN* fields corresponds to bits *PABITS*-1..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 combined *PFNX* || *PFN* fields corresponds to 0b00 || bits *PABITS*-1..12 of the physical address (the field is unshifted and the upper two bits must be written as zero). The boundaries of this field change as a function of the value of *PABITS*. See Table 9.1 for more information. If support for large physical addresses is not enabled ($Config3_{LPA}$ = 0 or $PageGrain_{ELPA}$ = 0), these bits are ignored on write and return 0 on read, thereby providing full backward compatibility with implementations of Release 1 of the Architecture. To ensure backward compatibility with pre-Release 5 software that does not support XPA, MTC0 is required to zero out the extension bits if $Config5_{MVH}$=1.	R/W	Undefined	Required for XPA; Optional otherwise
RI	31	Read Inhibit. If this bit is set in a TLB entry, an attempt, **other than a MIPS16 PC-relative load,** to read data on the virtual page causes a TLB Invalid or a TLBRI exception, even if the *V* (Valid) bit is set. The *RI* bit is writable only if the *RIE* bit of the *PageGrain* register is set. If the *RIE* bit of *PageGrain* is not set, the *RI* bit of *EntryLo0*/*EntryLo1* is set to zero on any write to the register, regardless of the value written. This bit is optional and its existence is denoted by the $Config3_{RXI}$ or $Config3_{SM}$ register fields. If not implemented, then reads of this field return 0.	R/W	0	Required by SmartMIPS ASE; Optional otherwise

Table 10: EntryLo0, EntryLo1 Register Field Descriptions in Release 5 of the Architecture

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
XI	30	Execute Inhibit. If this bit is set in a TLB entry, an attempt to fetch an instruction or to load MIPS16 PC-relative data from the virtual page causes a TLB Invalid or a TLBXI exception, even if the *V* (Valid) bit is set. The *XI* bit is writable only if the *XIE* bit of the *PageGrain* register is set. If the *XIE* bit of *PageGrain* is not set, the *XI* bit of *EntryLo0*/*EntryLo1* is set to zero on any write to the register, regardless of the value written. This bit is optional and its existence is denoted by the $Config3_{RXI}$ or $Config3_{SM}$ register fields. If not implemented, then reads of this field return 0.	R/W	0	Required by SmartMIPS ASE; Optional otherwise
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*.	R/W	Undefined	Required
C	5..3	The definition of this field is unchanged from Release 1.	R/W	Undefined	Required
D	2	The definition of this field is unchanged from Release 1.	R/W	Undefined	Required
V	1	The definition of this field is unchanged from Release 1.	R/W	Undefined	Required
G	0	The definition of this field is unchanged from Release 1.	R/W	Undefined	Required (TLB MMU)

Table 9.1 shows the movement of the *Fill*, *PFNX*, and *PFN* fields as a function of 1KB page support enabled, and the value of *PABITS*, in Release 5. Note that in implementations of the Architecture, *PABITS* can never be larger than 36 bits and there is no support for 1KB pages, so only the second row of the table applies in Release 1.

.

Table 9.1 EntryLo Field Widths as a Function of PABITS in Release 5

1KB Page Support Enabled?	*PABITS* Value	Corresponding EntryLo Field Bit Ranges			Required Release
		Fill Field	PFNX Field	PFN Field	
No	59 ≥ *PABITS* > 36	63..(55-(59-*PABITS*)) Example: 63..55 if *PABITS* = 59 63..33 if *PABITS* = 37	(54-(59-*PABITS*))..32 Example: 54..32 if *PABITS* = 59 32..32 if *PABITS* = 37 $\text{EntryLo}_{54..32} = \text{PA}_{59..36}$	29..6 $\text{EntryLo}_{29..6} = \text{PA}_{35..12}$	Release 5
	36 ≥ *PABITS* > 12	63..(32-(36-*PABITS*)) Example: 63..32 if *PABITS* = 36 63..32 & 29..7 if *PABITS* = 13	Displaced by the Fill Field	(29-(36-*PABITS*))..6 Example: 29..6 if *PABITS* = 36 6..6 if *PABITS* = 13 $\text{EntryLo}_{29..6} = \text{PA}_{35..12}$	Release 1
Yes	59 ≥ *PABITS* > 34	63..(57-(59-*PABITS*)) Example: 63..57 if *PABITS* = 59 63..33 if *PABITS* = 35	(56-(59-*PABITS*))..32 Example: 56..32 if *PABITS* = 59 33..32 if *PABITS* = 35 $\text{EntryLo}_{56..32} = \text{PA}_{59..34}$	29..6 $\text{EntryLo}_{29..6} = \text{PA}_{33..10}$	Release 5
	34 ≥ *PABITS* > 10	63..(32-(34-*PABITS*)) Example: 63..32 if *PABITS* = 34 63..32 & 29..7 if *PABITS* = 11	Displaced by the Fill Field	(29-(34-*PABITS*))..6 Example: 29..6 if *PABITS* = 34 6..6 if *PABITS* = 11 $\text{EntryLo}_{29..6} = \text{PA}_{33..10}$	Release 2

Programming Note:

In implementations of Release 2 of the Architecture (and subsequent releases), the*PFNX* (Release 5 for MIPS32) and *PFN* fields 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 9.2 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 uses a TLB mapping (either for an instruction fetch or for a load/store instruction) which was created with a C field encoding which is RESERVED for the implementation.

Table 9.2 lists the required and optional encodings for the cacheability and coherency attributes.

Table 9.2 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

9.7 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.

If $Config3_{CTXTC}$ =0 and $Config3_{SM}$ =0 then the *Context* register is organized in such a way that the operating system can directly reference a 16-byte structure in memory that describes the mapping. For PTE structures of other sizes, the content of this register can be used by the TLB refill handler after appropriate shifting and masking.

If $Config3_{CTXTC}$ =0 and $Config3_{SM}$ =0 then 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 9.7 shows the format of the *Context* Register when $Config3_{CTXTC}$ =0 and $Config3_{SM}$ =0; Table 9.3 describes the *Context* register fields $Config3_{CTXTC}$ =0 and $Config3_{SM}$ =0.

Figure 9.7 Context Register Format when $Config3_{CTXTC}$=0 and $Config3_{SM}$=0

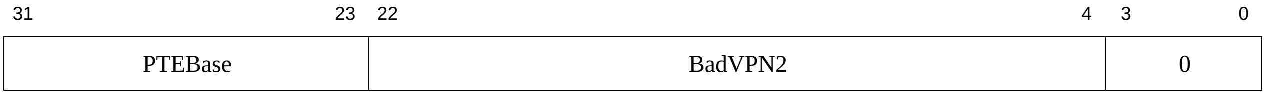


Table 9.3 Context Register Field Descriptions when $Config3_{CTXTC}$=0 and $Config3_{SM}$=0

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

If $Config3_{CTXTC}$ =1 or $Config3_{SM}$ =1 then the pointer implemented by the *Context* register can point to any power-of-two-sized PTE structure within memory. This allows the TLB refill handler to use the pointer without additional shifting and masking steps. Depending on the value in the *ContextConfig* register, it may point to an 8-byte pair of 32-bit PTEs within a single-level page table scheme, or to a first level page directory entry in a two-level lookup scheme.

If $Config3_{CTXTC}$ =1 or $Config3_{SM}$ =1 then the a TLB exception (Refill, Invalid, or Modified) causes bits $VA_{31:31-((X-Y)-1)}$ to be written to a variable range of bits "(X-1):Y" of the *Context* register, where this range corresponds to the contiguous range of set bits in the *ContextConfig* register. Bits 31:X are R/W to software, and are unaffected by the exception. Bits Y-1:0 are unaffected by the exception. If X = 23 and Y = 4, i.e. bits 22:4 are set in *ContextConfig*, the behavior is identical to the standard MIPS32 *Context* register (bits 22:4 are filled with $VA_{31:13}$). Although the fields have been made variable in size and interpretation, the MIPS32 nomenclature is retained. Bits 31:X are referred to as the *PTEBase* field, and bits X-1:Y are referred to as *BadVPN2*.

If $Config3_{SM}$ =1 then Bits Y-1:0 will always read as 0.

The value of the *Context* register is **UNPREDICTABLE** following a modification of the contents of the *ContextConfig* register.

Figure 9.8 shows the format of the *Context* Register when $Config3_{CTXTC}$ =1 or $Config3_{SM}$ =1; Table 9.4 describes the *Context* register fields $Config3_{CTXTC}$ =1 or $Config3_{SM}$ =1.

Figure 9.8 Context Register Format when $Config3_{CTXTC}$=1 or $Config3_{SM}$=1

31 X	X-1 Y	Y-1 0
PTEBase	BadVPN2	0

Table 9.4 Context Register Field Descriptions when $Config3_{CTXTC}$=1 or $Config3_{SM}$=1

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
PTEBase	Variable, 31:X where X in {31..0}. May be null.	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 to an array of data structures in memory corresponding to the address region containing the virtual address which caused the exception.	R/W	Undefined	Required
BadVPN2	Variable, (X-1):Y where X in {32..1} and Y in {31..0}. May be null.	This field is written by hardware on a TLB exception. It contains bits $VA_{31:31-((X-Y)-1)}$ of the virtual address that caused the exception.	R	Undefined	Required

Table 9.4 Context Register Field Descriptions when $\text{Config3}_{\text{CTXTC}}$=1 or $\text{Config3}_{\text{SM}}$=1

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
0	Variable, (Y-1):0 where Y in {31:1}. May be null.	Must be written as zero; returns zero on read.	R or R/W (R/W only allowed for $\text{Config3}_{\text{CTXT}}$=1)	0 (if R) or Undefined (if R/W)	Reserved

9.8 ContextConfig Register (CP0 Register 4, Select 1)

Compliance Level: *Optional.*

The *ContextConfig* register defines the bits of the *Context* register into which the high order bits of the virtual address causing a TLB exception will be written, and how many bits of that virtual address will be extracted. Bits above the selected field of the *Context* register are R/W to software and serve as the *PTEBase* field. Bits below the selected field of the *Context* register will be unaffected by TLB exceptions.

The field to contain the virtual address index is defined by a single block of contiguous non-zero bits within the *ContextConfig* register's *VirtualIndex* field. Any zero bits to the right of the least-significant one bit cause the corresponding *Context* register bits to be unaffected by TLB exceptions. Any zero bits to the left of the most- significant one bit cause the corresponding *Context* register bits to be R/W to software and unaffected by TLB exceptions.

If $Config3_{SM}$ is set, then any zero bits to the right of the least significant one bit causes the corresponding *Context* register bits to be read as zero.

It is permissible to implement a subset of the *ContextConfig* register, in which some number of bits are read-only and set to one or zero as appropriate. Software can determine whether a specific setting is implemented by writing that value into the register and reading back the register value. If the read value matches the original written value exactly, then the setting is supported. It is implementation specific what value is read back when the setting is not implemented except that the read value does not match the original written value. All implementations of the *ContextConfig* register must allow for the emulation of the MIPS32/microMIPS32 fixed *Context* register configuration.

This paragraph describes restrictions on how the *ContextConfig* register may be programmed. The set bits of *ContextConfig* define the *BadVPN2* field within the *Config* register. The *BadVPN2* field cannot contain address bits which are used to index a memory location within the even-odd page pairs used by the JTLB entries. This limits the least significant writeable bit within *ContextConfig* to the bits that represents *BadVPN2* of the smallest implemented page size. For example, if the smallest implemented page size is 4KB, virtual address bit 13 is the least significant bit of the *BadVPN2* field. Another example: if 1KB was the smallest implemented page size then the least significant writeable bit within *ContextConfig* would correspond to virtual address bit 11.

A value of all zeroes means that the full 32 bits of the *Context* register are R/W for software and unaffected by TLB exceptions.

The *ContextConfig* register is optional and its existence is denoted by the $Config3_{CTXTC}$ or $Config3_{SM}$ register fields.

Figure 9.9 shows the formats of the *ContextConfig* Register; Table 9.5 describes the *ContextConfig* register fields.

Figure 9.9 ContextConfig Register Format

31	0
VirtualIndex	

Table 9.5 ContextConfig Register Field Descriptions

Fields					
Name	**Bits**	**Description**	**Read / Write**	**Reset State**	**Compliance**
VirtualIndex	31:0	A mask of 0 to 32 contiguous 1 bits in this field causes the corresponding bits of the *Context* register to be written with the high-order bits of the virtual address causing a TLB exception. Behavior of the processor is **UNDEFINED** if non-contiguous 1 bits are written into the register field.	R/W	0x007ffff0	Required

Table 9.6 describes some useful *ContextConfig* values.

Table 9.6 Recommended ContextConfig Values

Value	Page Table Organization	Page Size	PTE Size	Compliance
0x007ffff0	Single Level	4K	64 bits/page	REQUIRED
0x007ffff8	Single Level	2K	32 bits/page	RECOMMENDED

9.9 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 Module is implemented, the *UserLocal* register is instantiated per TC.

This register only exists if the *Config3ULRI register field is set.*

Figure 9.10 shows the format of the *UserLocal* register; Table 9.7 describes the *UserLocal* register fields.

Figure 9.10 UserLocal Register Format

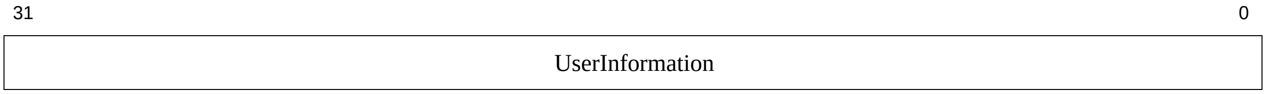


Table 9.7 UserLocal Register Field Descriptions

Fields		Description	Read/ Write	Reset State	Compliance
Name	Bits				
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.

9.10 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 9.9. Figure 9.11 shows the format of the *PageMask* register; Table 9.8 describes the *PageMask* register fields.

Figure 9.11 PageMask Register Format

31 29	28 13	12 11	10 0
0	Mask	MaskX	0

Table 9.8 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 (and subsequent releases), 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 (*Config3*$_{SP}$ = 1 and *PageGrain*$_{ESP}$ = 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 (*Config3*$_{SP}$ = 0 or *PageGrain*$_{ESP}$ = 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 9.9 Values for the Mask and MaskX[1] Fields of the PageMask Register

Page Size	Values for Mask field (lsb of value is located at PageMask$_{13}$)	Values for MaskX[1] field
1 KByte	0x0	0x0
4 KByte	0x0	0x3
16 KByte	0x3	0x3
64 KByte	0xF	0x3
256 KByte	0x3F	0x3
1 MByte	0xFF	0x3
4 MByte	0x3FF	0x3
16 MByte	0xFFF	0x3
64 MByte	0x3FFF	0x3
256 MByte	0xFFFF	0x3

[1] PageMask$_{12..11}$ = PageMask$_{MaskX}$ 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$_{SP}$* = 0 or *PageGrain$_{ESP}$* = 0).

It is implementation-dependent how many of the encodings described in Table 9.9 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 9.9, even if the hardware returns a different value on read. Hardware may depend on this requirement in implementing hardware structures

Config3$_{SP}$ Programming Note:

In implementations of Release 2 (and subsequent releases) 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.

9.11 PageGrain Register (CP0 Register 5, Select 1)

Compliance Level: *Required* for implementations of Release 2 (and subsequent releases) of the Architecture that include TLB-based MMUs and support 1KB pages, the XI/RI TLB protection bits, multiple types of Machine Check exceptions; *Required* for SmartMIPS™ ASE; *Required* for XPA (*Config3*$_{LPA}$=1); *Optional* otherwise.

The *PageGrain* register is a read/write register used for enabling 1KB page support, the XI/RI TLB protection bits, reporting the type of Machine Check exception, and Extended Physical Addressing. The *PageGrain* register is present in both the SmartMIPS™ ASE and in Release 2 (and subsequent releases) of the Architecture. 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. Figure 9-12 shows the format of the *PageGrain* register; Table 9.10 describes the *PageGrain* register fields.

Figure 9-12 PageGrain Register Format

31	30	29	28	27	26	25 13	12 8	7 5	4 0
RIE	XIE	ELPA	ESP	IEC	S32	0	ASE	0	MCCause

Table 9.10 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>RIE</td><td>31</td><td>Read Inhibit Enable.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td><i>RI</i> bit of the <i>EntryLo0</i> and <i>EntryLo1</i> registers is disabled and not writeable by software.</td></tr>
<tr><td>1</td><td><i>RI</i> bit of the <i>EntryLo0</i> and <i>EntryLo1</i> registers is enabled.</td></tr>
</table>
This bit is optional. The existence of this bit is denoted by either the <i>SM</i> or <i>RXI</i> bits in <i>Config3</i>. If this bit is not settable, then the <i>RI</i> bit in the <i>EntryLo*</i> registers is not implemented.</td><td>R/W
or
R</td><td>0</td><td>Required by SmartMIPS ASE; Optional otherwise</td></tr>
</table>

Table 9.10 PageGrain 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>XIE</td><td>30</td><td>Execute Inhibit Enable.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td><i>XI</i> bit of the <i>EntryLo0</i> and <i>EntryLo1</i> registers is disabled and not writeable by software.</td></tr>
<tr><td>1</td><td><i>XI</i> bit of the <i>EntryLo0</i> and <i>EntryLo1</i> registers is enabled.</td></tr>
</table>
This bit is optional. The existence of this bit is denoted by either the <i>SM</i> or <i>RXI</i> bits in the <i>Config3</i> register. If this bit is not settable, the <i>XI</i> bit in the <i>EntryLo*</i> registers is not implemented.</td><td>R/W
or
R</td><td>0</td><td>Required by SmartMIPS ASE; Optional otherwise</td></tr>
<tr><td>ASE</td><td>12..8</td><td>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.</td><td>0</td><td>0</td><td>Required</td></tr>
<tr><td>ELPA</td><td>29</td><td>Enables support for large physical addresses.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Large physical address support is not enabled</td></tr>
<tr><td>1</td><td>Large physical address support is enabled (XPA)</td></tr>
</table>
If this bit is a 1, the following changes occur to Coprocessor 0 registers:
• The <i>PFNX</i> field of the <i>EntryLo0</i> and <i>EntryLo1</i> registers is writable and concatenated with the <i>PFN</i> field to form the full page frame number.
• Access to optional COP0 registers with PA extension, <i>LLAddr, TagLo is defined.</i>
If this bit is a 0 and $Config3_{LPA}$ =1, then writes to above registers or fields are ignored and reads return 0.
<i>ELPA</i> is only writeable in a Release 5 implementation that support XPA i.e., $Config3_{LPA}$ = 1.
For implementations prior to Release 5 of the Architecture, this bit returns zero on read.</td><td>R/W</td><td>0</td><td>Required (Release 5)</td></tr>
</table>

Table 9.10 PageGrain 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>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 <i>PFN</i> 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 <i>MaskX</i> field of the <i>PageMask</i> register is writable and is concatenated to the right of the <i>Mask</i> field to form the “don’t care” mask for the TLB entry.
• The <i>VPN2X</i> 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 $Config3_{SP}$ = 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>IEC</td><td>27</td><td>Enables unique exception codes for the Read-Inhibit and Execute-Inhibit exceptions.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Read-Inhbit and Execute-Inhibit exceptions both use the TLBL exception code.</td></tr>
<tr><td>1</td><td>Read-Inhibit exceptions use the TLBRI exception code.
Execute-Inhibit exceptions use the TLBXI exception code</td></tr>
</table>
For implementations which follow the SmartMIPS ASE, this bit is ignored by the hardware, meaning the Read-Inhibit and Execute-Inhibit exceptions can only use the TLBL exception code.</td><td>R/W</td><td>0</td><td>Required</td></tr>
<tr><td>0</td><td>25..13, 7..5</td><td>Must be written as zero; returns zero on read.</td><td>0</td><td>0</td><td>Reserved</td></tr>
</table>

Table 9.10 PageGrain 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>MCCause</td><td>4..0</td><td>Machine Check Cause . Only valid after a Machine Check Exception.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>No Machine Check Reported</td></tr>
<tr><td>1</td><td>Multiple Hit in TLB(s).</td></tr>
<tr><td>2</td><td>Multiple Hits in TLB(s) for speculative accesses. The value in EPC might not point to the faulting instruction.</td></tr>
<tr><td>3</td><td>For Dual VTLB and FTLB. A page with EntryHi$_{EHINV}$=0 is written into FTLB and PageMask is not set to a pagesize that is supported by the FTLB.</td></tr>
<tr><td>4</td><td>For Dual VTLB and FTLB. A page with EntryHi$_{EHINV}$=0 is written into FTLB but the VPN2 field is not consistent with the TLB set seletected by the Index register.</td></tr>
<tr><td>5</td><td>For Hardware Page Table Walker and Dual Page Mode of Directory Level PTEs - first PTE accessed from memory has PTEVld bit set but second PTE accessed from memory does not have PTEVld bit set.</td></tr>
<tr><td>6</td><td>For Hardware Page Table Walker and derived Huge Page size is power-of-4 but Dual Page mode not implemented.</td></tr>
<tr><td>24-31</td><td>Implementation specific</td></tr>
<tr><td>Others</td><td>Reserved</td></tr>
</table></td><td>R</td><td>0</td><td>Optional if multiple types of Machine Check are supported.; Otherwise not needed.</td></tr>
</table>

Programming Note:

In implementations of Release 2 (and subsequent releases) 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
EntryLo0$_{PFN}$, EntryLo1$_{PFN}$	0
EntryLo0$_{PFNX}$, EntryLo1$_{PFNX}$	0
PageMask$_{MaskX}$	0b11
EntryHi$_{VPN2X}$	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.

9.12 SegCtl0 (CP0 Register 5, Select 2)

9.13 SegCtl1 (CP0 Register 5, Select 3)

9.14 SegCtl2 (CP0 Register 5, Select 4)

Compliance Level: *Required* for programmable memory segmentation; *Optional* otherwise.

The *Segmentation Control* registers allow configuring the memory segmentation system. If implemented, the Segmentation Configurations are always active.

The address space is split into six segments. The behavior of each region is controlled by a Segment Configuration. See Section 4.10 "Segmentation Control".

Segmentation Control allows address-specific behaviors defined by the Privileged Resource Architecture to be modified or disabled.

The *Segmentation Control* registers are instantiated per-VPE in an MT Module processor.

The existence of the *Segmentation Control* registers is denoted by the SC field within the *Config3* register.

The *EntryHi* EHINV TLB invalidate feature is required by Segmentation Control. The legacy software method of representing an invalid TLB entry by using an unmapped address value is not guaranteed to work.

Figure 9.13 shows the format of the *SegCtl0* Register.

Figure 9.13 SegCtl0 Register Format (CP0 Register 5, Select 2)

31 16	15 0
CFG 1	CFG 0

Table 9.11 SegCtl0 Register Field Descriptions

Fields				
Name	**Bits**	**Description**	**Read / Write**	**Reset State**
CFG 1	31..16	Segment Configuration 1, see Table 9.14	R/W	Implementation Dependent
CFG 0	15..0	Segment Configuration 0, see Table 9.14	R/W	

Figure 9.14 shows the format of the *SegCtl1* Register.

Figure 9.14 SegCtl1 Register Format (CP0 Register 5, Select 3)

31 16	15 0
CFG 3	CFG 2

Table 9.12 SegCtl1 Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
CFG 3	31..16	Segment Configuration 3, see Table 9.14	R/W	Implementation Dependent
CFG 2	15..0	Segment Configuration 2, see Table 9.14	R/W	

Figure 9.15 shows the format of the *SegCtl2* Register.

Figure 9.15 SegCtl2 Register Format (CP0 Register 5, Select 4)

31 16	15 0
CFG 5	CFG 4

Table 9.13 SegCtl2 Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
CFG 5	31..16	Segment Configuration 5, see Table 9.14	R/W	Implementation Dependent
CFG 4	15..0	Segment Configuration 4, see Table 9.14	R/W	

Table 9.14 describes the CFG (Segment Configuration) fields defined in all CFG fields of the Segmentation Control registers.

Table 9.14 CFG (Segment Configuration) Field Description

Fields		Description	Read / Write	Compliance
Name	Bits			
PA	15..9	Physical address bits for Segment, for use when unmapped. See Section 4.10 “Segmentation Control”. This field is provisioned to support mapping of up to a 36-bit physical address.	R/W	Required
0	8..7	Reserved.	R0	Required
AM	6..4	Access control mode. See Table 9.15.	R/W	Required
EU	3	Error condition behavior. Segment becomes unmapped and uncached when Status_{ERL}=1.	R/W	Required
C	2..0	Cache coherency attribute, for use when unmapped. As defined by base architecture.	R/W	Required

Table 9.15 describes the access control modes specifiable in the *CFG*$_{AM}$ field.

Table 9.15 Segment Configuration Access Control Modes

Mode		Action when referenced from Operating Mode			Description
		User mode	Supervisor mode	Kernel mode	
UK	000	Address Error	Address Error	Unmapped	Kernel-only unmapped region e.g. kseg0, kseg1
MK	001	Address Error	Address Error	Mapped	Kernel-only mapped region e.g. kseg3
MSK	010	Address Error	Mapped	Mapped	Supervisor and kernel mapped region e.g. ksseg, sseg
MUSK	011	Mapped	Mapped	Mapped	User, supervisor and kernel mapped region e.g. useg, kuseg, suseg
MUSUK	100	Mapped	Mapped	Unmapped	Used to implement a fully-mapped flat address space in user and supervisor modes, with unmapped regions which appear in kernel mode.
USK	101	Address Error	Unmapped	Unmapped	Supervisor and kernel unmapped region e.g. sseg in a fixed mapping TLB.
UUSK	111	Unmapped	Unmapped	Unmapped	Unrestricted unmapped region

Table 9.16 describes a configuration of Segmentation Control equivalent to legacy fixed partitioning. This is a recommended reset configuration for conformance with legacy fixed segmentation.

Table 9.16 Segment Configuration legacy reset state

CFG	Segment	AM	PA	C	EU
0	kseg3	MK	Undefined	Undefined	0
1	ksseg, sseg	MSK	Undefined	Undefined	0
2	kseg1	UK	0x000	2	0
3	kseg0	UK	0x000	3	0
4	kuseg, suseg, useg	MUSK	0x002	Undefined	1
5	kuseg, suseg, useg	MUSK	0x000	Undefined	1

Table 9.17 describes the partitioning of the microMIPS32 Address Space and the virtual address range mapped by each Segment Configuration (CFG).

Table 9.17 Segment Configuration partitioning of MIPS32 address space

CFG	Virtual Address range	Equivalent Segment name(s)

0	`0xFFFF FFFF`
	through
	`0xE000 0000`
		kseg3

1	`0xDFFF FFFF`
	through
	`0xC000 0000`
		ksseg, sseg

Table 9.17 Segment Configuration partitioning of MIPS32 address space

CFG	Virtual Address range	Equivalent Segment name(s)

2	`0xBFFF FFFF`
	through
	`0xA000 0000`
		kseg1

3	`0x9FFF FFFF`
	through
	`0x8000 0000`
		kseg0

4	`0x7FFF FFFF`
	through
	`0x4000 0000`
		kuseg, useg, suseg

5	`0x3FFF FFFF`
	through
	`0x0000 0000`	

9.15 PWBase Register (CP0 Register 5, Select 5)

Compliance Level: *Required* for the hardware page walker feature.

The *PWBase* register contains the Page Table Base virtual address, used as the starting point for hardware page table walking. It is used in combination with the *PWField* and *PWSize* registers.

The *PWBase* register is instantiated per-VPE in an MT Module processor.

The existence of this register is denoted when $Config3_{PW}$=1.

The operation of page table walking is described in Section 4.12 "Hardware Page Table Walker".

Figure 9.16 shows the format of the *PWBase* register; Table 9.18 describes the *PWBase* register fields.

Figure 9.16 PWBase Register Format

31 ... 0
PWBase

Table 9.18 PWBase Register Field Descriptions

Fields					
Name	**Bits**	**Description**	**Read / Write**	**Reset State**	**Compliance**
PWBase	31..0	Page Table Base address pointer	R/W	0	Required

9.16 PWField Register (CP0 Register 5, Select 6)

Compliance Level: *Required* for the hardware page walker feature.

The *PWField* register configures hardware page table walking for TLB refills. It is used in combination with the *PWBase* and *PWSize* registers.

The hardware page walker feature supports multi-level page tables - up to four directory levels plus one page table level. The lowest level of any page table system is an array of Page Table Entries (PTEs). This array is known as a Page Table (PT) and is indexed using bits from the faulting address. A single-level page table system contains only a single Page Table.

A multi-level page table system forms a tree structure - the lowest (leaf) elements of which are Page Table Entries. Levels above the lowest Page Table level are known as Directories. A directory consists of an array of pointers. Each pointer in a directory is either to another directory or to a Page Table.

The Page Table and the Directories are indexed by bits extracted from the faulting address. The *PWBase* register contains the base address of the first Directory or Page Table which will be accessed. The *PWSize* register specifies the number of index bits to be used for each level. The *PWField* register specifies the location of the index fields in the faulting address.

This register only exists if $Config3_{PW}$=1.

The *PWField* register is instantiated per-VPE in an MT Module processor.

If a synchronous exception condition is detected on a read operation during hardware page-table walking, the automated process is aborted and a TLB Refill exception is taken.

Figure 9.17 shows the formats of the *PWField* Register; Table 9.19 describes the *PWField* register fields.

Figure 9.17 PWField Register Format

31 30	29 24	23 18	17 12	11 6	5 0
0	GDI	UDI	MDI	PTI	PTEI

Table 9.19 PWField Register Field Descriptions

Fields					
Name	Bits	Description	Read / Write	Reset State	Compliance
0	31..30	Must be written as zero; returns zero on read.	R0	0	Required
GDI	29..24	Global Directory index. Least significant bit of the index field extracted from the faulting address, which is used to index into the Global Directory. The number of index bits is specified by *PWSize*$_{GDW}$.	R/W	0	Required when *PWSize*$_{GDW}$ is implemented
UDI	23..18	Upper Directory index. Least significant bit of the index field extracted from the faulting address, which is used to index into the Upper Directory. The number of index bits is specified by *PWSize*$_{UDW}$.	R/W	0	Required when *PWSize*$_{UDW}$ is implemented
MDI	17..12	Middle Directory index. Least significant bit of the index field extracted from the faulting address, which is used to index into the Middle Directory. The number of index bits is specified by *PWSize*$_{MDW}$.	R/W	0	Required when *PWSize*$_{MDW}$ is implemented
PTI	11..6	Page Table index. Least significant bit of the index field extracted from the faulting address, which is used to index into the Page Table. The number of index bits is specified by *PWSize*$_{PTW}$.	R/W	0	Required

Table 9.19 PWField Register Field Descriptions (Continued)

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
PTEI	5..0	Page Table Entry shift. Specifies the logical right shift and rotation which will be applied to Page Table Entry values loaded by hardware page table walking. The entire PTE is logically right shifted by *PTEI*-2 bits first. The purpose of this shift is to remove the SW-only bits from what will be written into the TLB entry. Then the two least-significant bits of the shifted value are rotated into position for the RI and XI protection bit locations within the TLB entry. A value of 2 means rotate the right-most 2 bits into the RI/XI bit positions for the TLB entry. A value of 3 means logical shift right by 1 bit the entire PTE and then rotate the right-most 2 bits into the RI/XI positions for the TLB entry. A value of 4 means logical shift right by 2bits the entire PTE and then rotate the right-most 2 bits into the RI/XI positions for the TLB entry. The values of 1 and 0 are RESERVED and should not be used; the operation of the HW Page Walker is **UNPREDICTABLE** for these cases. If the *PTEI* value is larger than the width of EntryLo0, then the value is treated as value-32. For example, a *PTEI* value of 34 is interpreted as PTEI=2. The values of 33 and 32 are RESERVED and should not be used; the operation of the HW Page Walker is **UNPREDICTABLE** for these cases. The set of available non-zero shifts is implementation-dependent. Software can discover the available values by writing this field. If the requested shift value is not available, *PTEI* will contain zero on read. A shift of zero must be implemented.	R/W	0	Required

Note that the *PTEI* field can be incorrectly programmed so that the entire PFN, C, V, G TLB fields are overwritten with zeros by the logical right shift operation. The intention of this facility is to only remove the SW-only bits of the PTE from the value which will be later written into the TLB.

9.17 PWSize Register (CP0 Register 5, Select 7)

Compliance Level: *Required* for the hardware page walk feature.

The *PWSize* register configures hardware page table walking for TLB refills. It is used in combination with the *PWBase* and *PWField* registers.

The operation of page table walking is described in Section 4.12 “Hardware Page Table Walker”.

The hardware page walk feature supports multi-level page tables - up to three directory levels plus one page table level. The lowest level of any page table system is an array of Page Table Entries (PTEs). This array is known as a Page Table (PT) and is indexed using bits from the faulting address. A single-level page table system contains only a single Page Table.

A multi-level page table system forms a tree structure - the lowest (leaf) elements of which are Page Table Entries. Levels above the lowest Page Table level are known as Directories. A directory consists of an array of pointers. Each pointer in a directory is either to another directory or to a Page Table.

The Page Table and the Directories are indexed by bits extracted from the faulting address *BadVAddr*. The *PWBase* register contains the base address of the first Directory or Page Table which will be accessed. The *PWSize* register specifies the number of index bits to be used for each level. The *PWField* register specifies the location of the index fields in *BadVAddr*.

Index values used to access Directories are multiplied by the native pointer size for the refill. For 32-bit addressing, the native pointer size is 32 bits (2 bit left shift). The index value used to access the Page Table is multiplied by the native pointer size. An additional multiplier (left shift value) can be specified using the $PWSize_{PTEW}$ field. This allows space to be allocated in the Page Table structure for software-managed fields.

This register only exists if $Config3_{PW}$=1.

The *PWSize* register is instantiated per-VPE in an MT Module processor.

Figure 9.18 shows the formats of the *PWSize* Register; Table 9.20 describes the *PWSize* register fields.

Figure 9.18 PWSize Register Format

31	30	29 24	23 18	17 12	11 6	5 0
0	PS	GDW	UDW	MDW	PTW	PTEW

Table 9.20 PWSize 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>0</td><td>31</td><td>Must be written as zero; returns zero on read.</td><td>0</td><td>0</td><td>Required</td></tr>
<tr><td>PS</td><td>0</td><td>Pointer Size - this is only used by the 64-bit architectures. For the 32-bit architectures, this bit is fixed to 0.</td><td>R</td><td>0</td><td>Required</td></tr>
<tr><td>GDW</td><td>29..24</td><td>Global Directory index width.
<table>
<tr><th>Value</th><th>Meaning</th></tr>
<tr><td>0</td><td>No read is performed using Global Directory index.</td></tr>
<tr><td>Non-zero</td><td>Number of bits to be extracted from <i>BadVAddr</i> to create an index into the Global Directory. The least significant bit of the field is specified by $PWField_{GDI}$.</td></tr>
</table></td><td>R/W</td><td>0</td><td>Recommended</td></tr>
<tr><td>UDW</td><td>23..18</td><td>Upper Directory index width.
<table>
<tr><th>Value</th><th>Meaning</th></tr>
<tr><td>0</td><td>No read is performed using Upper Directory index.</td></tr>
<tr><td>Non-zero</td><td>Number of bits to be extracted from <i>BadVAddr</i> to create an index into the Upper Directory. The least significant bit of the field is specified by $PWField_{UDI}$.</td></tr>
</table></td><td>R/W</td><td>0</td><td>Recommended</td></tr>
<tr><td>MDW</td><td>17..12</td><td>Middle Directory index width.
<table>
<tr><th>Value</th><th>Meaning</th></tr>
<tr><td>0</td><td>No read is performed using Middle Directory index.</td></tr>
<tr><td>Non-zero</td><td>Number of bits to be extracted from <i>BadVAddr</i> to create an index into the Middle Directory. The least significant bit of the field is specified by $PWField_{MDI}$.</td></tr>
</table></td><td>R/W</td><td>0</td><td>Recommended</td></tr>
</table>

Table 9.20 PWSize 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>PTW</td><td>11..6</td><td>Page Table index width.
<table>
<tr><th>Value</th><th>Meaning</th></tr>
<tr><td>0</td><td>UNPREDICTABLE</td></tr>
<tr><td>Non-zero</td><td>Number of bits to be extracted from <i>BadVAddr</i> to create an index into the Page Table. The least significant bit of the field is specified by $PWField_{PTI}$.</td></tr>
</table></td><td>R/W</td><td>0</td><td>Required</td></tr>
<tr><td>PTEW</td><td>5..0</td><td>Specifies the left shift applied to the Page Table index, in addition to the shift required to account for the native data size of the machine.
The set of available shifts is implementation-dependent. Software can discover the available values by writing this field. If the requested shift value is not available, PTEW will be written as zero. A shift of one must be implemented.</td><td>R/W</td><td>0</td><td>Required</td></tr>
</table>

Table 9.21 describes valid $PWSize_{PS/PTEW}$ and $PWCtl_{HugePg}$ settings.

Table 9.21 PS/PTEW Usage

$PWSize_{PS}$	$PWCtl_{HugePg}$	$PWSize_{PTEW}$	Pointer Addressing	Directory Pointer SIze	Non-Leaf PTE Size	Leaf PTE Size	Suggested Use Case
0	0	0	32 bits	32 bits	N/A	32 bits	32-bit
0	0	1	32 bits	32 bits	N/A	64 bits	32-bit with PA>32bits
0	1	0	32 bits	32 bits	32 bits	32 bits	32-bit with Huge Pages
0	1	1	32 bits	64 bits[1]	64 bits	64 bits	32-bit with Huge Pages & PA>32 bits
N/A	N/A	>1					Not supported

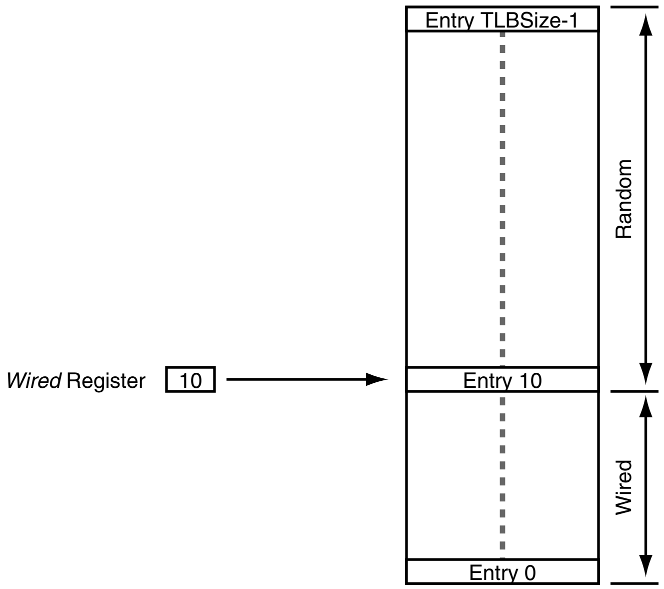
[1] The “Directory Pointer Size” column denotes how many bytes of memory is used for each pointer in the directory levels. If this size is larger than the pointer itself, the pointer uses the least significant bytes.

9.18 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 9.19.

Figure 9.19 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 9.19 shows the format of the *Wired* register; Table 9.22 describes the *Wired* register fields.

Figure 9.20 Wired Register Format

31 … n	n-1 … 0
0	Wired

Table 9.22 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

9.19 PWCtl Register (CP0 Register 6, Select 6)

Compliance Level: *Required* for the hardware page walker feature.

The *PWCtl* register configures hardware page table walking for TLB refills. It is used in combination with the *PWBase, PWField* and *PWSize* registers.

Hardware page table walking is disabled when $PWCtl_{PWEn}$=0.

The hardware page walker feature supports multi-level page tables - up to four directory levels plus one page table level. The lowest level of any page table system is an array of Page Table Entries (PTEs). This array is known as a Page Table (PT) and is indexed using bits from the faulting address. A single-level page table system contains only a single Page Table.

A multi-level page table system forms a tree structure - the lowest (leaf) elements of which are Page Table Entries. Levels above the lowest Page Table level are known as Directories. A directory consists of an array of pointers. Each pointer in a directory is either to another directory or to a Page Table.

The Page Table and the Directories are indexed by bits extracted from the faulting address *BadVAddr.* The *PWBase* register contains the base address of the first Directory or Page Table which will be accessed. The *PWSize* register specifies the number of index bits to be used for each level. The *PWField* register specifies the location of the index fields in *BadVAddr.*

The existence of this register is denoted when $Config3_{PW}$=1.

The *PWField* register is instantiated per-VPE in an MT Module processor.

Figure 9.21 shows the formats of the *PWCtl* Register; Table 9.23 describes the *PWCtl* register fields.

Figure 9.21 PWCtl Register Format

31	30	7	6	5..0
PWEn	Reserved	DPH	HugePg	Psn

Table 9.23 PWCtl Register Field Descriptions

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
PWEn	31	Hardware Page Table walker enable If this bit is set, then the Hardware Page Table is enabled.	R/W	0	Required
-	30..8	Reserved, Must be written as zero; returns zero on read.	R0	0	Required
DPH	7	Dual Page format of Huge Page support. This bit is only used when *HugePg*=1. If *DPH* bit is set, then a Huge Page PTE can represent a power-of-4 memory region or a 2x power-of-4 memory region. For the first case, one PTE is used for even TLB page and the adjacent PTE is used for the odd PTE. For the latter case, the Hardware will synthesize the physical addresses for both the even and odd TLB pages from the single PTE entry. If *DPH* bit is clear, then a Huge Page PTE can only represent a region that is 2 x power-of-4 in size. For this case, the Hardware will synthesize the physical addresses for both the even and odd TLB pages from the single PTE entry.	R or R/W	0	Required
HugePg	6	Huge Page PTE supported in Directory levels. If this bit is set, then Huge Page PTE in non-leaf table (i.e., directory level) is supported.	R or R/W	0	Required
PSn	5:0	Bit position of *PTEvld* in Huge Page PTE. Only used when *HugePg* field is set.	R/W	0	Required

If the implementation supports Huge Pages, then Software enables Huge Pages by setting $PWCtl_{HugePg}$=1. Software can disable Huge Pages by setting $PWCtl_{HugePg}$ = 0. An implementation that does not support Huge Pages is required to hardwire $PWCtl_{HugetPg}$ = 0 read-only. Software can determine Huge Page support by writing 1 to $PWCtl_{HugePg}$, if a following read returns 0, then Huge Page support is not implemented.

The *PWCtlPsn* field is provisioned at 6 bits, allowing a starting bit position for *PTEvld* up to bit 64 in the PTE. An implementation may choose to support a more limited range by hardwiring an implementation defined number of the high order bits of $PWCtl_{Psn}$ to 0. Software can determine the supported range by writing ones to *PWCtlPsn* then reading.

For non-Leaf

Table 9.24 describes how the *HugePg* field is used to denote whether Huge Pages are supported or not.

Table 9.24 HugePg Field and Huge Page configurations

$\text{PWCTL}_{\text{HugePg}}$	Type of Entry		Rsvd Field in Non-leaf entry	Comment
	Non-Leaf	Leaf		
0	Always Pointer $\text{PTE}_{\text{PTEVld}}$ not used	Always PTE $\text{PTE}_{\text{PTEVld}}$ not used	X	No Huge-Page Support
1	$\text{PTE}_{\text{PTEVld}}$=0 means Pointer $\text{PTE}_{\text{PTEVld}}$=1 means Huge Page	Always PTE $\text{PTE}_{\text{PTEVld}}$ not used	Must be 0	Huge-Page Support

Table 9.25 describes how Huge Pages are represented in the Directory Levels.

Table 9.25 Huge Page representation in Directory Levels

$\text{PWCTL}_{\text{DPH}}$	Size of Huge Page		Comment
	Power of 4	non-Power of 4	
0	Not Allowed If encountered, HW Page Walker aborts and TLB Refill exception is taken.	Allowed Even TLB page and Odd TLB page entries both derived from single PTE	Huge-Page region can only be 2x power-of-4
1	Allowed Two PTEs are read from memory by the HW Page Walker to be used for the Even and Odd TLB page entries.	Allowed Even TLB page and Odd TLB page entries both derived from single PTE	Huge-Page region can be any power-of- 2 (either power of 4 or 2x power-of-4)

9.20 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 9.22 shows the format of the *HWREna* Register; Table 9.26 describes the *HWREna* register fields.

Figure 9.22 HWREna Register Format

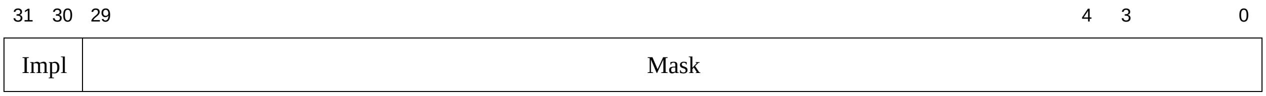


Table 9.26 HWREna Register Field Descriptions

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
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 9.27 lists the RDHWR registers, and register number ‘n’ corresponds to bit ‘n’ in this field.	R/W	0	Required

Table 9.27 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 <i>EBaseCPUNum</i> 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.

9.21 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 9.23 shows the format of the *BadVAddr* register; Table 9.28 describes the *BadVAddr* register fields.

Figure 9.23 BadVAddr Register Format

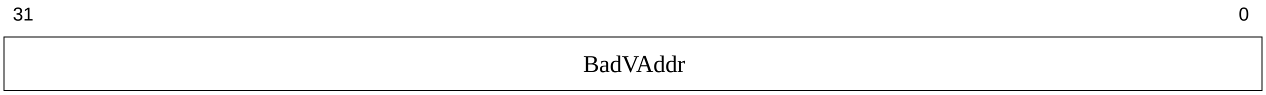


Table 9.28 BadVAddr Register Field Descriptions

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

9.22 BadInstr Register (CP0 Register 8, Select 1)

Compliance Level: *Optional*

The *BadInstr* register is a read-only register that capture the most recent instruction which caused one of the following exceptions:

- Instruction validity

  Coprocessor Unusable, Reserved Instruction

- Execution Exception

  Integer Overflow, Trap, System Call, Breakpoint, Floating Point, Coprocessor 2 exception

- Addressing

  Address Error, TLB Refill, TLB Invalid, TLB Read Inhibit, TLB Execute Inhibit, TLB Modified

The *BadInstr* register is provided to allow acceleration of instruction emulation. The *BadInstr* register is only set by exceptions which are synchronous to an instruction. The *BadInstr* register is not set by Interrupts, NMI, Machine check, Bus Error or Cache Error exceptions. The *BadInstr* register is not set by Watch or EJTAG exceptions.

When a synchronous exception occurs for which there is no valid instruction word (for example TLB Refill - Instruction Fetch), the value stored in *BadInstr* is **UNPREDICTABLE**.

Presence of the *BadInstr* register is indicated by the $Config3_{BI}$ bit. The *BadInstr* register is instantiated per-VPE in an MT Module processor.

Figure 9.24 shows the proposed format of the *BadInstr* register; Table 9.29describes the *BadInstr* register fields.

Figure 9.24 BadInstr Register Format

31 0
BadInstr

Table 9.29 BadInstr Register Field Descriptions

Fields					
Name	Bits	Description	Read / Write	Reset State	Compliance
BadInstr	31:0	Faulting instruction word. Instruction words smaller than 32 bits are placed in bits 15:0, with bits 31:16 containing zero.	R	Undefined	Optional

9.23 BadInstrP Register (CP0 Register 8, Select 2)

Compliance Level: *Optional*

The *BadInstrP* register is used in conjunction with the *BadInstr* register. The *BadInstrP* register contains the prior branch instruction, when the faulting instruction is in a branch delay slot.

The *BadInstrP* register is updated for these exceptions:

- Instruction validity

  Coprocessor Unusable, Reserved Instruction

- Execution Exception

  Integer Overflow, Trap, System Call, Breakpoint, Floating Point, Coprocessor 2 exception

- Addressing

  Address Error, TLB Refill, TLB Invalid, TLB Read Inhibit, TLB Execute Inhibit, TLB Modified

The *BadInstrP* register is provided to allow acceleration of instruction emulation. The *BadInstrP* register is only set by exceptions which are synchronous to an instruction. The *BadInstrP* register is not set by Interrupts, NMI, Machine check, Bus Error or Cache Error exceptions. The *BadInstr* register is not set by Watch or EJTAG exceptions.

When a synchronous exception occurs and the faulting instruction is not in a branch delay slot, then the value stored in *BadInstrP* is **UNPREDICTABLE**.

Presence of the *BadInstrP* register is indicated by the *Config3*$_{BP}$ bit. The *BadInstrP* register is instantiated per-VPE in an MT Module processor.

Figure 9.25 shows the proposed format of the *BadInstrP* register; Table 9.30describes the *BadInstrP* register fields.

Figure 9.25 BadInstrP Register Format

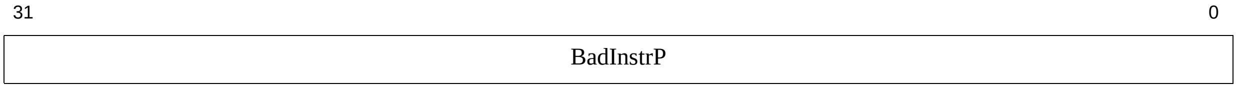


Table 9.30 BadInstrP Register Field Descriptions

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
BadInstrP	31:0	Prior branch instruction. Instruction words smaller than 32 bits are placed in bits 15:0, with bits 31:16 containing zero.	R	Undefined	Optional

9.24 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 9.26 shows the format of the *Count* register; Table 9.31 describes the *Count* register fields.

Figure 9.26 Count Register Format

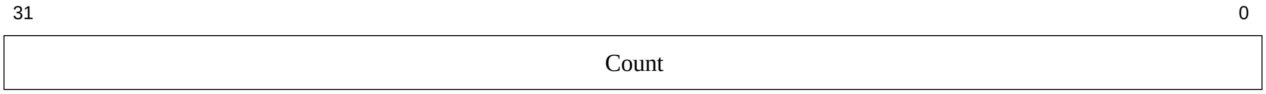


Table 9.31 Count Register Field Descriptions

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

9.25 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.

9.26 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.

In Release 3 of the architecture, the *VPN2* field of the TLB entry can be optionally invalidated. When this is done, the invalidated entry is ignored on address match for memory accesses. One method of invalidating the *VPN2* field is the use of the *EHINV* field with the TLBWI instruction. This field exists if $Config4_{IE}$ is set to a value of 2 or 3. This field is overwritten by a TLBR instruction, so software must save and restore the value of the EHINV field around the use of the TLBR instruction. This is especially important for the subsequent usage of TLBWI instructions.

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 9.27 shows the format of the *EntryHi* register; Table 9.32 describes the *EntryHi* register fields.

Figure 9.27 EntryHi Register Format

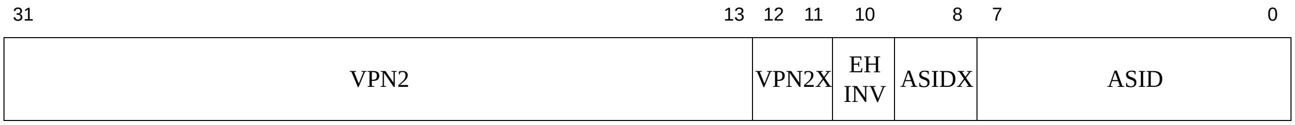


Table 9.32 EntryHi Register Field Descriptions

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

Table 9.32 EntryHi Register Field Descriptions

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
VPN2X	12..11	In Release 2 of the Architecture (and subsequent releases), 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)
EHINV	10	TLB HW Invalidate If $Config4_{IE}$ > 1, and this bit is set, the TLBWI instruction will invalidate the VPN2 field of the selected TLB entry. If $Config4_{IE}$ > 1, a TLBR instruction will update this field withe the VPN2 invalid bit of the read TLB entry.	R/W	0	Optional in release 3. Required for TLBWI invalidate support.
ASIDX	9..8	If $Config4_{AE}$ = 1 then these bits extend the ASID field. If $Config4_{AE}$ = 0 then Must be written as zero; returns zero on read.	If $Config4_{AE}$ = 1 then R/W else 0	If $Config4_{AE}$ = 1 then Undefined else 0	Required
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 (and subsequent releases) 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.

9.27 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 (and subsequent releases) 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 $IntCtl_{IPTI}$ 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 9.28 shows the format of the *Compare* register; Table 9.33 describes the *Compare* register fields.

Figure 9.28 Compare Register Format

31	0
Compare	

Table 9.33 Compare Register Field Descriptions

Fields					
Name	Bits	Description	Read / Write	Reset State	Compliance
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 6.1.2.1 “Software Hazards and the Interrupt System” on page 84.

9.28 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.

9.29 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 and microMIPS32 Operating Modes" on page 23 for a discussion of operating modes, and "Interrupts" on page 73 for a discussion of interrupt modes.

Figure 9.29 shows the format of the *Status* register; Table 9.34 describes the *Status* register fields.

Figure 9.29 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	0	BEV	TS	SR	NMI	ASE	Impl	IM7..IM2	IM1..IM0	0	UM	R0	ERL	EXL	IE
												IPL			KSU				

Table 9.34 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 <i>CU0</i> bit.
In Release 2 (and subsequent releases) 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 <i>CU1</i> enable. <i>CU3</i> is no longer used and is reserved for future use by the Architecture.
If there is no provision for connecting a coprocessor, the corresponding <i>CU</i> 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 9.34 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>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>
In Release 1 of the Architecture, only MIPS64 processors could implement a 64-bit floating point unit. In Release 2 of the Architecture (and subsequent releases), both 32-bit and 64-bit processors can implement a 64-bit floating point unit. As of Release 5 of the Architecture, if floating point is implemented then *FR* = 1 is required. I.e. the 64-bit FPU, with the *FR* = 1 64-bit FPU register model, is required. The *FR* = 0 32-bit FPU register model continues to be required.

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 (and subsequent releases) in which a 64-bit floating point unit is not implemented
Certain combinations of the FR bit and other state or operations can cause **UNPREDICTABLE** behavior. See “64-bit FPR Enable” on page 24 for a discussion of these combinations.
When software changes the value of this bit, the contents of the floating point registers are **UNPREDICTABLE.**</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>
</table>

Table 9.34 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>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 Module 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 Modules; otherwise R/W</td><td>0 if the processor implements neither the MDMX nor the MIPS DSP Modules; otherwise Undefined</td><td>Optional</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 86 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 (and subsequent releases), 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 35 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>
</table>

Table 9.34 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>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>
<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 <b>UNPREDICTABLE</b> 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>ASE</td><td>18</td><td>This bit is reserved for the MCU ASE.
If MCU ASE is not implemented, then this bit must be written as zero; returns zero on read.</td><td>0 if MCU ASE is not implemented</td><td>0 if MCU ASE is not implemented</td><td>Required for MCU ASE; Otherwise 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 73 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 (<i>Config3</i>$_{VEIC}$ = 1), these bits take on a different meaning and are interpreted as the <i>IPL</i> field, described below.</td><td>R/W</td><td>Undefined</td><td>Required</td></tr>
</table>

Table 9.34 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>IPL</td><td>15..10</td><td>Interrupt Priority Level.
In implementations of Release 2 of the Architecture (and subsequent releases) in which EIC interrupt mode is enabled ($Config3_{VEIC}$ = 1), this field is the encoded (0..63) value of the current <i>IPL</i>. 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>
<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 73 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>0</td><td>,23,7:5</td><td>Must be written as zero; returns zero on read</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 and microMIPS32 Operating Modes” on page 23 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 <i>KSU</i> field</td></tr>
</table>
Note: This field overlaps the <i>UM</i> and <i>R0</i> fields, described below.</td><td>R/W</td><td>Undefined</td><td>Required if Supervisor Mode is implemented; Optional otherwise</td></tr>
<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 and microMIPS32 Operating Modes” on page 23 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 <i>KSU</i> field, described above.</td><td>R/W</td><td>Undefined</td><td>Required</td></tr>
</table>

Table 9.34 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>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 <i>KSU</i> 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 <i>ERL</i> is set:
<ul>
<li>The processor is running in kernel mode</li>
<li>Hardware and software interrupts are disabled</li>
<li>The ERET instruction will use the return address held in <i>ErrorEPC</i> instead of EPC</li>
<li>Segment kuseg is treated as an unmapped and uncached region. See “Address Translation for the kuseg Segment when StatusERL = 1” on page 33. This allows main memory to be accessed in the presence of cache errors. The operation of the processor is <b>UNDEFINED</b> if the <i>ERL</i> bit is set while the processor is executing instructions from kuseg.</li>
</ul></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 <i>EXL</i> is set:
<ul>
<li>The processor is running in Kernel Mode</li>
<li>Hardware and software interrupts are disabled.</li>
<li>TLB Refill exceptions use the general exception vector instead of the TLB Refill vector.</li>
<li>EPC, $Cause_{BD}$ and SRSCtl (implementations of Release 2 of the Architecture only) will not be updated if another exception is taken</li>
</ul></td><td>R/W</td><td>Undefined</td><td>Required</td></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 (and subsequent releases), 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 84.

9.30 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 9.30 shows the format of the *IntCtl* register; Table 9.35 describes the *IntCtl* register fields.

.

Figure 9.30 IntCtl Register Format

31 29	28 26	25 23	22 14	13 10	9 5	4 0
IPTI	IPPCI	IPFDC	MCU ASE	0000	VS	0

Table 9.35 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 by hardware or Externally Set</td><td>Required</td></tr>
</table>

Table 9.35 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 by hardware or Externally Set</td><td>Optional (Performance Counters Implemented)</td></tr>
<tr><td>IPFDC</td><td>25..23</td><td>For Interrupt Compatibility and Vectored Interrupt modes, this field specifies the IP number to which the Fast Debug Channel Interrupt request is merged, and allows software to determine whether to consider $Cause_{FDCI}$ 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 EJTAG FDC is not implemented, this field returns zero on read.</td><td>R</td><td>Preset by hardware or Externally Set</td><td>Optional (EJTAG Fast Debug Channel Implemented)</td></tr>
<tr><td>MCU ASE</td><td>22..14</td><td>These bits are reserved for the MicroController ASE.
If that ASE is not implemented, must be written as zero; returns zero on read.</td><td>0</td><td>0</td><td>Reserved</td></tr>
</table>

Table 9.35 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>0</td><td>22..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 **UNDEFINED** 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>

9.31 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 9.31 shows the format of the *SRSCtl* register; Table 9.36 describes the *SRSCtl* register fields.

Figure 9.31 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 9.36 SRSCtl Register Field Descriptions

Fields					
Name	Bits	Description	Read / Write	Reset State	Compliance
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 by hardware	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 80 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 9.36 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 $Status_{BEV}$ = 0. This field is not updated on any exception which sets $Status_{ERL}$ to 1 (i.e., NMI or cache error), an entry into EJTAG Debug mode, or any exception or interrupt that occurs with $Status_{EXL}$ = 1, or $Status_{BEV}$ = 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 9.37 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 $Status_{ERL}$ to 1 (i.e., NMI or cache error), an entry into EJTAG Debug mode, or any exception or interrupt that occurs with $Status_{EXL}$ = 1, or $Status_{BEV}$ = 1. Neither is it updated on an ERET with $Status_{ERL}$ = 1 or $Status_{BEV}$ = 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 9.37 Sources for new $SRSCtl_{CSS}$ on an Exception or Interrupt

Exception Type	Condition	$SRSCtl_{CSS}$ Source	Comment
Exception	All	$SRSCtl_{ESS}$	
Non-Vectored Interrupt	$Cause_{IV}$ = 0	$SRSCtl_{ESS}$	Treat as exception

Table 9.37 Sources for new $\text{SRSCtl}_{\text{CSS}}$ on an Exception or Interrupt

Exception Type	Condition	$\text{SRSCtl}_{\text{CSS}}$ Source	Comment
Vectored Interrupt	$\text{Cause}_{\text{IV}} = 1$ and $\text{Config3}_{\text{VEIC}} = 0$ and $\text{Config3}_{\text{VInt}} = 1$	$\text{SRSMap}_{\texttt{VectNum}\times\texttt{4+3..VectNum}\times\texttt{4}}$	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 109. 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.

9.32 SRSMap Register (CP0 Register 12, Select 3)

Compliance Level: *Required* in Release 2 (and subsequent releases) 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 9.32 shows the format of the *SRSMap* register; Table 9.38 describes the *SRSMap* register fields.

Figure 9.32 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 9.38 SRSMap Register Field Descriptions

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
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

9.33 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 9.33 shows the format of the *Cause* register; Table 9.39 describes the *Cause* register fields.

Figure 9.33 Cause Register Format

31	30	29 28	27	26	25 24	23	22	21	20	17	15 10	9 8	7	6 2	1 0
BD	TI	CE	DC	PCI	ASE	IV	WP	FDCI	000	ASE	IP9..IP2	IP1..IP0	0	Exc Code	0
										ASE	RIPL				

Table 9.39 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 <i>BD</i> 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 <i>IP</i> 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>
<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>
</table>

Table 9.39 Cause 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>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 (and subsequent releases), 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 (<i>Config1</i>$_{PC}$ = 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>ASE</td><td>25:24, 17:16</td><td>These bits are reserved for the MCU ASE.
If MCU ASE is not implemented, these bits return zero on reads and must be written with zeros.</td><td></td><td></td><td>Required for MCU ASE; Otherwise Reserved</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 (and subsequent releases), if the Cause$_{IV}$ is 1 and <i>Status</i>$_{BEV}$ 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 9.39 Cause 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>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 **UNPREDICTABLE** 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>FDCI</td><td>21</td><td>Fast Debug Channel Interrupt. This bit denotes whether a FDC interrupt is pending:<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>No FDC interrupt is pending</td></tr><tr><td>1</td><td>FDC interrupt is pending</td></tr></table></td><td>R</td><td>Undefined</td><td>Required</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 (and subsequent releases) 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>
</table>

Table 9.39 Cause 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>RIPL</td><td>15..10</td><td>Requested Interrupt Priority Level.
In implementations of Release 2 of the Architecture (and subsequent releases) 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>
<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 (and subsequent releases) 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 9.40</td><td>R</td><td>Undefined</td><td>Required</td></tr>
<tr><td>0</td><td>25:24, 20..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 9.40 Cause Register ExcCode Field

Exception Code Value			
Decimal	Hexadecimal	Mnemonic	Description
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 $Debug_{DExcCode}$ field to denote an SDBBP in Debug Mode.
10	0x0a	RI	Reserved instruction exception

Table 9.40 Cause Register ExcCode Field

Exception Code Value			
Decimal	Hexadecimal	Mnemonic	Description
11	0x0b	CpU	Coprocessor Unusable exception
12	0x0c	Ov	Arithmetic Overflow exception
13	0x0d	Tr	Trap exception
14	0x0e	MSAFPE	MSA Floating Point exception
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	0x13	TLBRI	TLB Read-Inhibit exception
20	0x14	TLBXI	TLB Execution-Inhibit exception
21	0x15	MSADis	MSA Disabled exception
22	0x16	MDMX	Previously MDMX Unusable Exception (MDMX ASE). MDMX deprecated with Revision 5.
23	0x17	WATCH	Reference to WatchHi/WatchLo address
24	0x18	MCheck	Machine check
25	0x19	Thread	Thread Allocation, Deallocation, or Scheduling Exceptions (MIPS® MT Module)
26	0x1a	DSPDis	DSP Module State Disabled exception (MIPS® DSP Module)
27	0x1b	GE	Virtualized Guest Exception
28-29	0x1c - 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 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 (and the subsequent releases), 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 84.

9.34 NestedExc (CP0 Register 13, Select 5)

Compliance Level: *Optional*.

The *Nested Exception* (*NestedExc*) register is a read-only register containing the values of *Status*$_{EXL}$ and *Status*$_{ERL}$ prior to acceptance of the current exception.

This register is part of the Nested Fault feature, existence of the register can be determined by reading the *Config5*$_{NFExists}$ bit.

Figure 9.34 shows the format of the *NestedExc* register; Table 9.41 describes the *NestedExc* register fields.

Figure 9.34 NestedExc Register Format

31 3	2	1	0
0	ERL	EXL	0

Table 9.41 NestedExc Register Field Descriptions

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
0	31..3	Reserved, read as 0.	R0	0	Required
ERL	2	Value of *Status*$_{ERL}$ prior to acceptance of current exception. Updated by all exceptions that would set either *Status*$_{EXL}$ or *Status*$_{ERL}$. Not updated by Debug exceptions.	R	Undefined	Required
EXL	1	Value of *Status*$_{EXL}$ prior to acceptance of current exception. Updated by exceptions which would update EPC if *Status*$_{EXL}$ is not set (MCheck, Interrupt, Address Error, all TLB exceptions, Bus Error, CopUnusable, Reserved Instruction, Overflow, Trap, Syscall, FPU, etc.) . For these exception types, this register field is updated regardless of the value of *Status*$_{EXL}$. Not updated by exception types which update *ErrorEPC* - (Reset, Soft Reset, NMI, Cache Error). Not updated by Debug exceptions.	R	Undefined	Required
0	0	Reserved, read as 0.	R0	0	Required

9.35 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 9.35 shows the format of the *EPC* register; Table 9.42 describes the *EPC* register fields.

Figure 9.35 EPC Register Format

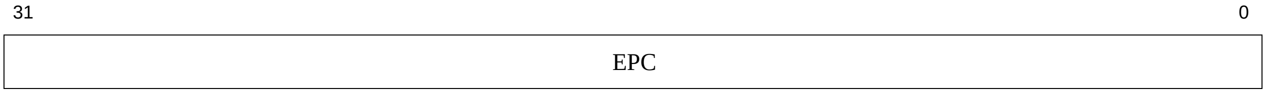


Table 9.42 EPC Register Field Descriptions

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

9.35.1 Special Handling of the EPC Register in Processors that Implement MIPS16e ASE or microMIPS32 Base Architecture

In processors that implement the MIPS16e ASE or microMIPS32 base architecture, 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:

```
$\mathtt{EPC} \leftarrow \mathtt{resumePC}_{31..1} \parallel \mathtt{ISAMode}_0$
```

“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:

```
$\mathtt{PC} \leftarrow \mathtt{EPC}_{31..1} \parallel 0$
$\mathtt{ISAMode} \leftarrow \mathtt{EPC}_0$
```

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.

9.36 Nested Exception Program Counter (CP0 Register 14, Select 2)

Compliance Level: *Optional.*

The *Nested Exception Program Counter* (*NestedEPC*) is a read/write register with the same behavior as the *EPC* register except that:

- The *NestedEPC* register ignores the value of *Status*$_{EXL}$ and is therefore updated on the occurance of any exception, including nested exceptions.

- The *NestedEPC* register is not used by the ERET/DERET/IRET instructions. Software is required to copy the value of the *NestedEPC* register to the *EPC* register if it is desired to return to the address stored in *NestedEPC*.

This register is part of the Nested Fault feature, existence of the register can be determined by reading the *Config5*$_{NFExists}$ bit.

Figure 9.36 shows the format of the *NestedEPC* register; Table 9.43 describes the *NestedEPC* register fields.

Figure 9.36 NestedEPC Register Format

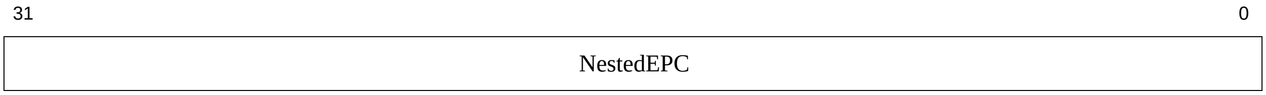


Table 9.43 NestedEPC Register Field Descriptions

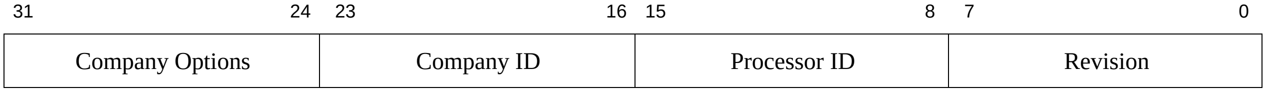
Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
NestedEPC	31..0	Nested Exception Program Counter Updated by exceptions which would update EPC if *Status*$_{EXL}$ is not set (MCheck, Interrupt, Address Error, all TLB exceptions, Bus Error, CopUnusable, Reserved Instruction, Overflow, Trap, Syscall, FPU, etc.) . For these exception types, this register field is updated regardless of the value of *Status*$_{EXL}$. Not updated by exception types which update *ErrorEPC* - (Reset, Soft Reset, NMI, Cache Error). Not updated by Debug exceptions.	R/W	Undefined	Required

9.37 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 9.37 shows the format of the *PRId* register; Table 9.44 describes the *PRId* register fields.

Figure 9.37 PRId Register Format



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

Table 9.44 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 by hardware</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/microMIPS32 or MIPS64/microMIPS64 processor from one implementing an earlier MIPS ISA by checking this field for zero. If it is non-zero the processor implements the MIPS32/microMIPS32 or MIPS64/microMIPS64 Architecture.
Company IDs are assigned by MIPS Technologies when a MIPS32/microMIPS32 or MIPS64/microMIPS64 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/microMIPS32 or MIPS64/microMIPS64 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 by hardware</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 by hardware</td><td>Required</td></tr>
<tr><td>Revision</td><td>7..0</td><td>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.</td><td>R</td><td>Preset by hardware</td><td>Optional</td></tr>
</table>

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 support@mips.com, reporting the specific case.

9.38 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 *Status*$_{BEV}$ 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 *Status*$_{BEV}$ is 0. The exception vector base address comes from the fixed defaults (see 6.2.2 "Exception Vector Locations" on page 86) when *Status*$_{BEV}$ 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.

If the write-gate bit is not implemented, 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.

The operation of the *EBase* register can be optionally extended to allow the upper bits of the Exception Base field to be written. This allows exception vectors to be placed anywhere in the address space. To ensure backward compatibility with MIPS32, the write-gate bit must be set before the upper bits can be changed. For the write-gate case, the full set of bits 31..12 are used to compute the vector location. Software can detect the existence of the write-gate by writing one to that bit position and checking if the bit was set.

The addition of the base address and the exception offset is performed inhibiting a carry between bits 29 and 30 of the final exception address.

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

Figure 9.38 shows the format of the *EBase* register if the write-gate is not implemented. ; Table 9.45 describes the *EBase* register fields.

Figure 9.38 EBase Register Format

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

Table 9.45 EBase Register Field Descriptions

Fields					
Name	Bits	Description	Read / Write	Reset State	Compliance
1	31	This bit is ignored on write and returns one on read.	R	1	Required

Table 9.45 EBase Register Field Descriptions

Fields					
Name	Bits	Description	Read / Write	Reset State	Compliance
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 *Status*$_{BEV}$ is zero.	R/W	0	Required
0	11..10	Must be written as zero; returns zero on read.	0	0	Reserved
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 by hardware or Externally Set	Required

Figure 9.39 shows the format of the *EBase* register if the write-gate is implemented. Table 9.46 describes the *EBase* register fields.

Figure 9.39 EBase Register Format

31 12	11	10	9 0
Exception Base	WG	0	CPUNum

Table 9.46 EBase Register Field Descriptions

Fields					
Name	Bits	Description	Read / Write	Reset State	Compliance
Exception Base	31..12	This field specifies the base address of the exception vectors when *Status*$_{BEV}$ is zero. Bits 31..30 can be written only when WG is set. When WG is zero, these bits are unchanged on write.	R/W	0x80000	Required
WG	11	Write gate. Bits 31..30 are unchanged on writes to EBase when WG=0 in the value being written. The WG bit must be set true in the written value to change the values of bits 31..30 .	R/W	0	Required
0	10	Must be written as zero; returns zero on read.	R0	0	Reserved
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 $EBase_{15..12}$ 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 9.47 shows the conditions under which each *EBase* bit must be set to zero. *VN* represents the interrupt vector number as described in Table 6.4 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 9.47 Conditions Under Which EBase15..12 Must Be Zero

	Interrupt Vector Spacing in Bytes ($IntCtl_{VS}$[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 9.35 on page 179

9.39 CDMMBase Register (CP0 Register 15, Select 2)

Compliance Level: *Optional*.

The 36-bit physical base address for the Common Device Memory Map facility is defined by this register. This register only exists if *Config3*$_{CDMM}$ is set to one.

For devices that implement multiple VPEs, access to this register is controlled by the *VPEConf0*$_{MVP}$ register field. If the *MVP* bit is cleared, a read to this register returns all zeros and a write to this register is ignored.

Figure 9.40 has the format of the *CDMMBase* register, and Table 9.48 describes the register fields.

Figure 9.40 CDMMBase Register

31 — 11	10	9	8 — 0
CDMM_UPPER_ADDR	EN	CI	CDMMSize

Table 9.48 CDMMBase 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>CDMM_UPPER_ADDR</td><td>31:11</td><td>Bits 35:15 of the base physical address of the memory mapped registers.

The number of implemented physical address bits is implementation specific. For the unimplemented address bits - writes are ignored, returns zero on read.</td><td>R/W</td><td>Undefined</td><td>Required</td></tr>
<tr><td>EN</td><td>10</td><td>Enables the CDMM region.
If this bit is cleared, memory requests to this address region go to regular system memory. If this bit is set, memory requests to this region go to the CDMM logic
<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>CDMM Region is disabled.</td></tr><tr><td>1</td><td>CDMM Region is enabled.</td></tr></table></td><td>R/W</td><td>0</td><td>Required</td></tr>
<tr><td>CI</td><td>9</td><td>If set to 1, this indicates that the first 64-byte Device Register Block of the CDMM is reserved for additional registers which manage CDMM region behavior and are not IO device registers.</td><td>R</td><td>Preset</td><td>Optional</td></tr>
<tr><td>CDMMSize</td><td>8:0</td><td>This field represents the number of 64-byte Device Register Blocks are instantiated in the core.
<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>1 DRB</td></tr><tr><td>1</td><td>2 DRBs</td></tr><tr><td>2</td><td>3 DRBs</td></tr><tr><td>...</td><td>...</td></tr><tr><td>511</td><td>512 DRBs</td></tr></table></td><td>R</td><td>Preset</td><td>Required</td></tr>
</table>

9.40 CMGCRBase Register (CP0 Register 15, Select 3)

Compliance Level: *Optional*.

The 36-bit physical base address for the memory-mapped Coherency Manager Global Configuration Register space is reflected by this register. This register only exists if *Config3*$_{CMGCR}$ is set to one.

On devices that implement the MIPS MT Module, this register is instantiated once per processor.

Figure 9.41 has the format of the *CMGCRBase* register, and Table 9.49 describes the register fields.

Figure 9.41 CMGCRBase Register

31 11	10 0
CMGCR_BASE_ADDR	0

Table 9.49 CMGCRBase Register Field Descriptions

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
CMGCR_BASE_ADDR	31:11	Bits 35:15 of the base physical address of the memory-mapped Coherency Manager GCR registers. This register field reflects the value of the GCR_BASE field within the memory-mapped Coherency Manager GCR Base Register. The number of implemented physical address bits is implementation specific. For the unimplemented address bits - writes are ignored, returns zero on read.	R	Preset by hardware (IP Configuration Value)	Required
0	10:0	Must be written as zero; returns zero on read	0	0	Reserved

9.41 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 9.42 shows the format of the *Config* register; Table 9.50 describes the *Config* register fields.

Figure 9.42 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 9.50 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 267 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 267 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 by harddware or Externally Set</td><td>Required</td></tr>
</table>

Table 9.50 Config 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>AT</td><td>14:13</td><td>Architecture Type implemented by the processor.
For Release 3, encoding values of 0-2, denotes address and register width (32-bit or 64-bit).
The implemented instruction sets (MIPS32/64 and/or microMIPS32/64) are denoted by the ISA register field of <i>Config3.</i>
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>MIPS32 or microMIPS32</td></tr>
<tr><td>1</td><td>MIPS64 or microMIPS64 with access only to 32-bit compatibility segments</td></tr>
<tr><td>2</td><td>MIPS64or microMIPS64 with access to all address segments</td></tr>
<tr><td>3</td><td>Reserved</td></tr>
</table></td><td>R</td><td>Preset by hardware</td><td>Required</td></tr>
<tr><td>AR</td><td>12:10</td><td>MIPS32 Architecture revision level.
microMIPS32 Architecture revision level is denoted by the MMAR field of <i>Config3.</i> If <i>Config3</i> register is not implemented then microMIPS is not implemented.
If the <i>ISA</i> field of <i>Config3</i> is one, then MIPS32 is not implemented and this field is not used.
<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 or Release 3/MIPSr3 or Release 5
All features introduced in Release 3 and Release 5 are optional and detectable through <i>Config3</i> or other register fields.</td></tr>
<tr><td>2-7</td><td>Reserved</td></tr>
</table></td><td>R</td><td>Preset by hardware</td><td>Required</td></tr>
</table>

Table 9.50 Config 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>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 (See “TLB Organization” on page 34)</td></tr>
<tr><td>2</td><td>BAT (See “Block Address Translation” on page 271)</td></tr>
<tr><td>3</td><td>Fixed Mapping (See “Fixed Mapping MMU” on page 267)</td></tr>
<tr><td>4</td><td>Dual VTLB and FTLB (See “Dual Variable-Page-Size and Fixed-Page-Size TLBs” on page 273)</td></tr>
</table></td><td>R</td><td>Preset by hardware</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 by hardware</td><td>Required</td></tr>
<tr><td>K0</td><td>2:0</td><td>Kseg0 cacheability and coherency attribute. See Table 9.2 on page 130 for the encoding of this field.</td><td>R/W</td><td>Undefined</td><td>Required</td></tr>
</table>

9.42 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 I-Cache and D-Cache 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 9.43 shows the format of the *Config1* register; Table 9.51 describes the *Config1* register fields.

Figure 9.43 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 9.51 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 by hardware</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 in 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 by hardware</td><td>Required</td></tr>
<tr><td>IS</td><td>24:22</td><td>I=cache 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>32</td></tr>
</table></td><td>R</td><td>Preset by hardware</td><td>Required</td></tr>
</table>

Table 9.51 Config1 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>IL</td><td>21:19</td><td>I-cache line size:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>No I-Cache 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 by hardware</td><td>Required</td></tr>
<tr><td>IA</td><td>18:16</td><td>I-cache 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 by hardware</td><td>Required</td></tr>
<tr><td>DS</td><td>15:13</td><td>D-cache 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>32</td></tr>
</table></td><td>R</td><td>Preset by hardware</td><td>Required</td></tr>
</table>

Table 9.51 Config1 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>DL</td><td>12:10</td><td>D-cache line size:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>No D-Cache 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 by hardware</td><td>Required</td></tr>
<tr><td>DA</td><td>9:7</td><td>D-cache 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 by hardware</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>R</td><td>Preset by hardware</td><td>Required</td></tr>
<tr><td>MD</td><td>5</td><td>Used to denote MDMX ASE implemented on a MIPS64/microMIPS64 processor. Not used on a MIPS32/microMIPS32 processor.

This bit indicates not only that the processor contains support for MDMX, but that such a processing element is attached.
MDMX is deprecated in Release 5 and cannot be implemented when the MSA Module is implemented.</td><td>R</td><td>0</td><td>Required</td></tr>
</table>

Table 9.51 Config1 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>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 by hardware</td><td>Required</td></tr>
<tr><td>WR</td><td>3</td><td><i>Watch</i> 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><i>Watch</i> registers implemented</td></tr></table></td><td>R</td><td>Preset by hardware</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 by hardware</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 by hardware</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 by hardware</td><td>Required</td></tr>
</tbody>
</table>

9.43 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 9.44 shows the format of the *Config2* register; Table 9.52 describes the *Config2* register fields.

Figure 9.44 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 9.52 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 by hardware</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 by hardware</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 by hardware</td><td>Required</td></tr>
</table>

Table 9.52 Config2 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>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 by hardware</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 by hardware</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 by hardware</td><td>Optional</td></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 by hardware</td><td>Required</td></tr>
</table>

Table 9.52 Config2 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>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 by hardware</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 by hardware</td><td>Required</td></tr>
</table>

9.44 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.

Release 5 adds $Config3_{LPA}$ to allow software to determine the presence of XPA (>36-bit PA support).

Figure 9-45 shows the format of the *Config3* register; Table 9.53 describes the *Config3* register fields.

Figure 9-45 Config3 Register Format

31	30	29	28	27	26	25	24	23	22 21	20 19 18	17	16	15 14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
M	0	CMGCR	MSAP	BP	BI	SC	PW	VZ	IPLW	MMAR	MuCon	ISA On Exc	ISA	ULRI	RXI	DSP2P	DSPP	CTXTC	ITL	LPA	VEIC	VInt	SP	CDMM	MT	SM	TL

Table 9.53 Config3 Register Field Descriptions

Fields		Description	Read/Write	Reset State	Compliance
Name	Bits				
M	31	This bit is reserved to indicate that a *Config4* register is present. If the *Config4* register is not implemented, this bit should read as a 0. If the *Config4* register is implemented, this bit should read as a 1.	R	Preset by hardware	Required
0	30	Must be written as zero; returns zeros on read.	0	0	Reserved
CMGCR	29	Coherency Manager memory-mapped Global Configuration Register Space is implemented. **Encoding** / **Meaning** 0 / CM GCR space is not implemented 1 / CM GCR space is implemented	R	Preset by hardware	Required for Coherent Multiple -Core implementations that use the Coherency Manager.
MSAP	28	MIPS SIMD Architecture (MSA) is implemented. **Encoding** / **Meaning** 0 / MSA Module not implemented 1 / MSA Module is implemented	R	Preset by hardware	Required

Table 9.53 Config3 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>BP</td><td>27</td><td><i>BadInstrP</i> register implemented. This bit indicates whether the faulting prior branch instruction word register is present.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td><i>BadInstrP</i> register not implemented</td></tr>
<tr><td>1</td><td><i>BadInstrP</i> register implemented</td></tr>
</table></td><td>R</td><td>Preset by hardware</td><td>Required</td></tr>
<tr><td>BI</td><td>26</td><td><i>BadInstr</i> register implemented. This bit indicates whether the faulting instruction word register is present.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td><i>BadInstr</i> register not implemented</td></tr>
<tr><td>1</td><td><i>BadInstr</i> register implemented</td></tr>
</table></td><td>R</td><td>Preset by hardware</td><td>Required</td></tr>
<tr><td>SC</td><td>25</td><td>Segment Control implemented. This bit indicates whether the Segment Control registers <i>SegCtl0</i>, <i>SegCtl1</i> and <i>SegCtl2</i> are present.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Segment Control not implemented</td></tr>
<tr><td>1</td><td>Segment Control is implemented</td></tr>
</table></td><td>R</td><td>Preset by hardware</td><td>Required</td></tr>
<tr><td>PW</td><td>24</td><td>HardWare Page Table Walk implemented. This bit indicates whether the Page Table Walking registers <i>PWBase</i>, <i>PWField</i> and <i>PWSize</i> are present.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Page Table Walking not implemented</td></tr>
<tr><td>1</td><td>Page Table Walking is implemented</td></tr>
</table></td><td>R</td><td>Preset by hardware</td><td>Required</td></tr>
<tr><td>VZ</td><td>23</td><td>Virtualization Module implemented. This bit indicates whether the Virtualization Module is implemented.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Virtualization Module not implemented</td></tr>
<tr><td>1</td><td>Virtualization Module is implemented</td></tr>
</table></td><td>R</td><td>Preset by hardware</td><td>Required</td></tr>
</table>

Table 9.53 Config3 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>IPLW</td><td>22:21</td><td>Width of $Status_{IPL}$ and $Cause_{RIPL}$ fields:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td><i>IPL</i> and <i>RIPL</i> fields are 6-bits in width.</td></tr>
<tr><td>1</td><td><i>IPL</i> and <i>RIPL</i> fields are 8-bits in width.</td></tr>
<tr><td>Others</td><td>Reserved.</td></tr>
</table>
If the <i>IPL</i> field is 8-bits in width, bits 18 and 16 of <i>Status</i> are used as the most-significant bit and second most-significant bit, respectively, of that field.
If the <i>RIPL</i> field is 8-bits in width, bits 17 and 16 of <i>Cause</i> are used as the most-significant bit and second most-significant bit, respectively, of that field.</td><td>R</td><td>Preset by hardware</td><td>Required if MCU ASE is implemented</td></tr>
<tr><td>MMAR</td><td>20:18</td><td>microMIPS32 Architecture revision level.
MIPS32 Architecture revision level is denoted by the <i>AR</i> field of <i>Config.</i>
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Release3/MIPSr3</td></tr>
<tr><td>1-7</td><td>Reserved</td></tr>
</table>
If the <i>ISA</i> field of <i>Config3</i> is zero, microMIPS32 is not implemented and this field is not used.</td><td>R</td><td>Preset by hardware</td><td>Required if microMIPS is implemented</td></tr>
<tr><td>MCU</td><td>17</td><td>MIPS® MCU ASE is implemented.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>MCU ASE is not implemented.</td></tr>
<tr><td>1</td><td>MCU ASE is implemented</td></tr>
</table></td><td>R</td><td>Preset by hardware</td><td>Required if MCU ASE is implemented</td></tr>
<tr><td>ISAOnExc</td><td>16</td><td>Reflects the Instruction Set Architecture used after vectoring to an exception. Affects all exceptions whose offsets are relative to <i>EBase</i>.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>MIPS32is used on entrance to an exception vector.</td></tr>
<tr><td>1</td><td>microMIPS is used on entrance to an exception vector.</td></tr>
</table></td><td>RW if both instruction sets are implemented; Preset if only microMIPS is implemented.</td><td>Undefined</td><td>Required if microMIPS is implemented</td></tr>
</table>

Table 9.53 Config3 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>ISA</td><td>15:14</td><td>Indicates Instruction Set Availability.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Only MIPS32 Instruction Set is implemented.</td></tr>
<tr><td>1</td><td>Only microMIPS32 is implemented.</td></tr>
<tr><td>2</td><td>Both MIPS32and microMIPS32 ISAs are implemented. MIPS32 ISA used when coming out of reset.</td></tr>
<tr><td>3</td><td>Both MIPS32and microMIPS32 ISAs are implemented. microMIPS32 ISA used when coming out of reset.</td></tr>
</table></td><td>R</td><td>Preset by hardware</td><td>Required if microMIPS is implemented</td></tr>
<tr><td>ULRI</td><td>13</td><td>*UserLocal* register implemented. This bit indicates whether the *UserLocal* Coprocessor 0 register is implemented.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>*UserLocal* register is not implemented</td></tr>
<tr><td>1</td><td>*UserLocal* register is implemented</td></tr>
</table></td><td>R</td><td>Preset by hardware</td><td>Required</td></tr>
<tr><td>RXI</td><td>12</td><td>Indicates whether the *RIE* and *XIE* bits exist within the *PageGrain* register.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>The *RIE* and *XIE* bits are not implemented within the *PageGrain* register.</td></tr>
<tr><td>1</td><td>The *RIE* and *XIE* bits are implemented within the *PageGrain* register.</td></tr>
</table></td><td>R</td><td>Preset by hardware</td><td>Required</td></tr>
<tr><td>DSP2P</td><td>11</td><td>MIPS® DSP Module Revision 2 implemented. This bit indicates whether Revision 2 of the MIPS DSP Module is implemented.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Revision 2 of the MIPS DSP Module is not implemented</td></tr>
<tr><td>1</td><td>Revision 2 of the MIPS DSP Module is implemented</td></tr>
</table></td><td>R</td><td>Preset by hardware</td><td>Required</td></tr>
</table>

Table 9.53 Config3 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>DSPP</td><td>10</td><td>MIPS® DSP Module implemented. This bit indicates whether the MIPS DSP Module is implemented.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>MIPS DSP Module is not implemented</td></tr>
<tr><td>1</td><td>MIPS DSP Module is implemented</td></tr>
</table></td><td>R</td><td>Preset by hardware</td><td>Required</td></tr>
<tr><td>CTXTC</td><td>9</td><td><em>ContextConfig</em> registers is implemented and the width of the BadVPN2 field within the <em>Config</em> register register depends on the contents of the <em>ContextConfig</em> register.
.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td><em>ContextConfig</em> is not implemented.</td></tr>
<tr><td>1</td><td><em>ContextConfig</em> is implemented and is used for the $Config_{BadVPN2}$ field.</td></tr>
</table></td><td>R</td><td>Preset by hardware</td><td>Required</td></tr>
<tr><td>ITL</td><td>8</td><td>MIPS® IFlowtrace™ 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 by hardware</td><td>Required (Release 2.1 Only)</td></tr>
<tr><td>LPA</td><td>7</td><td>Large Physical Address support is implemented, and the <em>PageGrain</em> register existsThe following Coprocessor 0 fields and associated control are present if this bit is a 1:
• Modifications to <em>EntryLo0</em> and <em>EntryLo1</em> to support physical addresses larger than 36 bits i.e., the XPA feature of Release 5.
• Modifications to other optional COP0 registers with PA: <em>LLAddr, TagLo.</em>
• <em>PageGrain</em>
• $Config5_{MVH}$
The following instructions or modified behavior are required:
• MTHC0, MFHC0 for access to extensions.
• MTC0 modification to zero-writeable extensions.
For implementations prior to Release 5 of the Architecture, this bit returns zero on read.</td><td>R</td><td>Preset by hardware</td><td>Required (Release 5)</td></tr>
</table>

Table 9.53 Config3 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>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 by hardware</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 by hardware</td><td>Required (Release 2 Only)</td></tr>
<tr><td>SP</td><td>4</td><td>Small (1KByte) page support is implemented, and the *PageGrain* register exists
<table>
<tr><td>Encoding</td><td>Meaning</td></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 by hardware</td><td>Required (Release 2 Only)</td></tr>
<tr><td>CDMM</td><td>3</td><td>Common Device Memory Map implemented. This bit indicates whether the CDMM is implemented.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>CDMM is not implemented</td></tr>
<tr><td>1</td><td>CDMM is implemented</td></tr>
</table></td><td>R</td><td>Preset by hardware</td><td>Required</td></tr>
</table>

Table 9.53 Config3 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>MT</td><td>2</td><td>MIPS® MT Module implemented. This bit indicates whether the MIPS MT Module is implemented.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>MIPS MT Module is not implemented</td></tr>
<tr><td>1</td><td>MIPS MT Module is implemented</td></tr>
</table></td><td>R</td><td>Preset by hardware</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 by hardware</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 by hardware</td><td>Required</td></tr>
</table>

9.45 Configuration Register 4 (CP0 Register 16, Select 4)

Compliance Level: *Required* if any optional feature described by this register is implemented: Release 2 of the Architecture; *Optional* otherwise.

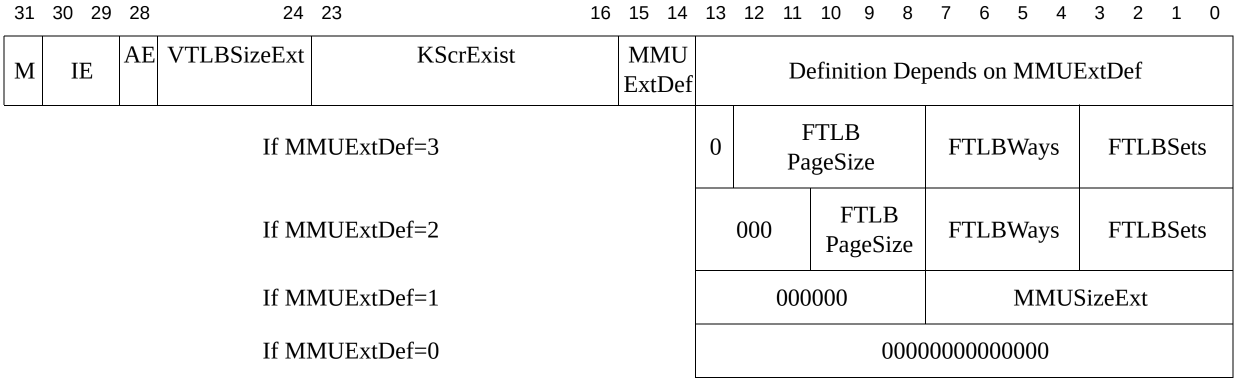
The *Config4* register encodes additional capabilities.

The number of page-pair entries within the FTLB = decode(FTLBSets) * decode(FTLBWays).

The number of page-pair entries accessible in the VTLB is defined by concatenating $\mathit{Config4}_{VTLBSizeExt}$ and $\mathit{Config1}_{MMUSize}$. Modifying VTLB size can be used to allow software to reserve high index slots in the VTLB.

Figure 9.46 shows the format of the *Config4* register; Table 9.54 describes the *Config4* register fields.

Figure 9.46 Config4 Register Format



31	30 29	28	27 24	23 16	15 14	13 12 11 10 9 8 7 6 5 4 3 2 1 0
M	IE	AE	VTLBSizeExt	KScrExist	MMU ExtDef	Definition Depends on MMUExtDef

If MMUExtDef=3	0	FTLB PageSize	FTLBWays	FTLBSets
If MMUExtDef=2	000	FTLB PageSize	FTLBWays	FTLBSets
If MMUExtDef=1	000000	MMUSizeExt		
If MMUExtDef=0	00000000000000			

Table 9.54 Config4 Register Field Descriptions

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
M	31	This bit is reserved to indicate that a *Config5* register is present. If the *Config5* register is not implemented, this bit should read as a 0. If the *Config5* register is implemented, this bit should read as a 1.	R	Preset by hardware	Required

Table 9.54 Config4 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>30:29</td><td>TLB invalidate instruction support/configuration.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>00</td><td>TLBINV, TLBINVF, $EntryHi_{EHINV}$ not supported by hardware</td></tr>
<tr><td>01</td><td>Reserved.</td></tr>
<tr><td>10</td><td>TLBINV, TLBINVF supported.
$EntryHi_{EHINV}$ supported. Refer to Volume II for the full description of these instructions.
TLBINV* instructions operate on one TLB entry.</td></tr>
<tr><td>11</td><td>TLBINV, TLBINVF supported.
$EntryHi_{EHINV}$ supported. Refer to Volume II for the full description of these instructions.
TLBINV* instructions operate on entire MMU.</td></tr>
</table></td><td>R</td><td>Preset by hardware</td><td>Required for TLBINV, TLBINVF, $\mathrm{EntryHi}_{\mathrm{EHINV}}$
These features must be implemented if Segmentation Control is implemented.
These features are recommended for FTLB/VTLB MMUs.</td></tr>
<tr><td>AE</td><td>28</td><td>If this bit is set, then $EntryHI_{ASID}$ is extended to 10 bits.</td><td>R</td><td>Preset by hardware</td><td>Required</td></tr>
<tr><td>VTLBSizeExt</td><td>27:24</td><td>If $Config4_{MMUExt}$=3 then this field is concatenated to the left of the most-significant bit of the $Config1_{MMUSize}$ field to indicate the size of the VTLB.</td><td>R</td><td>Preset by hardware</td><td>Required if MMUExtDef=3</td></tr>
<tr><td>KScr Exist</td><td>23:16</td><td>Indicates how many scratch registers are available to kernel-mode software within COP0 Register 31.
Each bit represents a select for Coproecessor0 Register 31. Bit 16 represents Select 0, Bit 23 represents Select 7.
If the bit is set, the associated scratch register is implemented and available for kernel-mode software.
Scratch registers meant for other purposes are not represented in this field. For example, if EJTAG is implemented, Bit 16 is preset to zero even though *DESAVE* register is implemented at Select 0. Select 1 is reserved for future debug purposes and should not be used as a kernel scratch register, so bit 17 is preset to zero.</td><td>R</td><td>Preset by hardware</td><td>Required if Kernel Scratch Registers are available</td></tr>
</table>

Table 9.54 Config4 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>MMU Ext Def</td><td>15:14</td><td>MMU Extension Definition.
Defines how *Config4*[13:0] is to be interpreted.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Reserved.
Config4[12:0] - Must be written as zeros, returns zeros on read.</td></tr>
<tr><td>1</td><td>Config4[7:0] used as MMUSizeExt.</td></tr>
<tr><td>2</td><td>Config4[3:0] used as FTLBSets.
Config4[7:4] used as FTLBWays.
Config4[10:8] used as FTLBPageSize.</td></tr>
<tr><td>3</td><td>FTLB and VTLB supported.
Config4[3:0] used as FTLBSets.
Config4[7:4] used as FTLBWays.
Config4[12:8] used as FTLBPageSize.
Config4[27:24] used as VTLBSizeExt.</td></tr>
</table></td><td>R</td><td>Preset by hardware</td><td>Required</td></tr>
<tr><td>FTLB Page Size</td><td>10:8</td><td>Indicates the Page Size of the FTLB Array Entries.
<table>
<tr><th>Encoding</th><th>Page Size</th></tr>
<tr><td>0</td><td>1 KB</td></tr>
<tr><td>1</td><td>4 KB</td></tr>
<tr><td>2</td><td>16 KB</td></tr>
<tr><td>3</td><td>64KB</td></tr>
<tr><td>4</td><td>256 KB</td></tr>
<tr><td>5</td><td>1 GB</td></tr>
<tr><td>6</td><td>4 GB</td></tr>
<tr><td>7</td><td>Reserved</td></tr>
</table>
Implementations are allowed to implement any subset of these sizes, even a subset of only one pagesize. Software can detect if a FTLB page size is implemented by writing the desired size into this register field. If the size is implemented, the register field is updated to the desired encoding. If the size is not implemented, the register field value is not changed.

The FTLB must be flushed of any valid entries before this register field value is changed by software. The FTLB behavior is **UNDEFINED** if there are valid FTLB entries which were not all programmed using a common page size.</td><td>RW if multiple FTLB pagesizes are implle-mented

R if only one FTLB page size is imple-mented.</td><td>Preset by hardware, chosen value is implementation specific</td><td>Required if MMUExt-Def=2</td></tr>
</table>

Table 9.54 Config4 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>FTLB Page Size</td><td>12:8</td><td>Indicates the Page Size of the FTLB Array Entries.
<table>
<tr><th>Encoding</th><th>Page Size</th></tr>
<tr><td>0</td><td>1 KB</td></tr>
<tr><td>1</td><td>4 KB</td></tr>
<tr><td>2</td><td>16 KB</td></tr>
<tr><td>3</td><td>64KB</td></tr>
<tr><td>4</td><td>256 KB</td></tr>
<tr><td>5</td><td>1 MB</td></tr>
<tr><td>6</td><td>4 MB</td></tr>
<tr><td>7</td><td>16 MB</td></tr>
<tr><td>8</td><td>64 MB</td></tr>
<tr><td>9</td><td>256 MB</td></tr>
<tr><td>10</td><td>1 GB</td></tr>
<tr><td>11</td><td>4 GB</td></tr>
<tr><td>12</td><td>16 GB</td></tr>
<tr><td>13</td><td>64 GB</td></tr>
<tr><td>14</td><td>256 GB</td></tr>
<tr><td>15</td><td>1 TB</td></tr>
<tr><td>16</td><td>4 TB</td></tr>
<tr><td>17</td><td>16 TB</td></tr>
<tr><td>18</td><td>64 TB</td></tr>
<tr><td>19</td><td>256 TB</td></tr>
</table>

Implementations are allowed to implement any subset of these sizes, even a subset of only one page size. Software can detect if an FTLB page size is implemented by writing the desired size into this register field. If the size is implemented, the register field is updated to the desired encoding. If the size is not implemented, the register field value is not changed.

The FTLB must be flushed of any valid entries before this register field value is changed by software. The FTLB behavior is <b>UNDEFINED</b> if there are valid FTLB entries which were not all programmed using a common page size.</td><td>R/W if multiple FTLB pagesizes are implemented

R if only one FTLB page size is implemented.</td><td>Preset by hardware, chosen value is implementation specific</td><td>Required if MMUExtDef=3</td></tr>
</table>

Table 9.54 Config4 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>FTLB Ways</td><td>7:4</td><td>Indicates the Set Associativity of the FTLB Array.
<table>
<tr><th>Encoding</th><th>Associativity</th></tr>
<tr><td>0</td><td>2</td></tr>
<tr><td>1</td><td>3</td></tr>
<tr><td>2</td><td>4</td></tr>
<tr><td>3</td><td>5</td></tr>
<tr><td>4</td><td>6</td></tr>
<tr><td>5</td><td>7</td></tr>
<tr><td>6</td><td>8</td></tr>
<tr><td>7-15</td><td>Reserved</td></tr>
</table></td><td>R</td><td>Preset by hardware</td><td>Required if MMUExt-Def=2</td></tr>
<tr><td>FTLB Sets</td><td>3:0</td><td>Indicates the number of Sets per Way within the FTLB Array.
<table>
<tr><th>Encoding</th><th>Sets per Way</th></tr>
<tr><td>0</td><td>1</td></tr>
<tr><td>1</td><td>2</td></tr>
<tr><td>2</td><td>4</td></tr>
<tr><td>3</td><td>8</td></tr>
<tr><td>4</td><td>16</td></tr>
<tr><td>5</td><td>32</td></tr>
<tr><td>6</td><td>64</td></tr>
<tr><td>7</td><td>128</td></tr>
<tr><td>8</td><td>256</td></tr>
<tr><td>9</td><td>512</td></tr>
<tr><td>10</td><td>1024</td></tr>
<tr><td>11</td><td>2048</td></tr>
<tr><td>12</td><td>4096</td></tr>
<tr><td>13</td><td>8192</td></tr>
<tr><td>14</td><td>16384</td></tr>
<tr><td>15</td><td>32768</td></tr>
</table></td><td>R</td><td>Preset by hardware</td><td>Required if MMUExt-Def=2</td></tr>
<tr><td>MMU Size Ext</td><td>7:0</td><td>If $\mathit{Config4}_{MMUExt}$=1 then this field is an extension of $\mathit{Config1}_{MMUSize-1}$ field.
This field is concatenated to the left of the most-significant bit to the *MMUSize-1* field to indicate the size of the TLB-1.</td><td>R</td><td>Preset by hardware</td><td>Required if MMUExt-Def=1</td></tr>
</table>

9.46 Configuration Register 5 (CP0 Register 16, Select 5)

Compliance Level: *Required* if any optional feature described by this register is implemented: Release 3 of the Architecture; *Optional* otherwise.

The *Config5* register encodes additional capabilities:

- Cache Error exception vector control.
- Segmentation Control legacy compatability.
- Existence of EVA instructions (LBK, LBUK, LHK, LHUK, LWK, SBK, SHK, SWK).
- Existence of the User Mode FP Register mode-changing facility (*UFR*).
- Existence of the Nested Fault feature (*NestedExc*, *NestedEPC*).
- Existence of COP0 *MAAR* and *MAARI* (*MRP*).
- Support for additional LL/SC instruction handling capabilities (*LLB*).
- Existence of MTHC0 and MFHC0 instructions.

Figure 9.47 shows the format of the *Config5* register; Table 9.56 describes the *Config5* register fields.

Figure 9.47 Config5 Register Format

31	30	29	28	27	26 6	5	4	3	2	1	0
M	K	CV	EVA	MSAEn	0	MVH	LLB	MRP	UFR	0	NFExists

Table 9.55 Config5 Register Field Descriptions

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
M	31	This bit is reserved to indicate that as yet undefined configuration registers are present. With the current architectural definition, this bit should always read as a 0.	R	Preset by hardware	Required

Table 9.55 Config5 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>K</td><td>30</td><td>Enable/disable $Config_{K0}$, $Config_{Ku}$, $Config_{K23}$ Cache Coherency Attribute control if Segmentation Control is implemented.[1]
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>$Config_{K0}$, $Config_{Ku}$, $Config_{K23}$ enabled.</td></tr>
<tr><td>1</td><td>$Configk0$, $Config_{Ku}$, $Config_{K23}$ disabled.</td></tr>
</table></td><td>R/W</td><td>0</td><td>Required for Segmentation Control. (Refer to 4.1.4 on page 28)</td></tr>
<tr><td>CV</td><td>29</td><td>Cache Error Exception Vector control. Disables logic forcing use of kseg1 region in the event of a Cache Error exception when $Status_{BEV}$=0.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>On Cache Error exception, vector address bits 31..29 forced to place vector in kseg1.</td></tr>
<tr><td>1</td><td>On Cache Error exception, vector address uses full *EBase* value for bits 31..29.</td></tr>
</table></td><td>R/W</td><td>0</td><td>Required for Segmentation Control. (Refer to 4.1.4 on page 28)</td></tr>
<tr><td>EVA</td><td>28</td><td>Enhanced Virtual Addressing instructions implemented</td><td>R</td><td>Preset by hardware</td><td>Optional</td></tr>
<tr><td>MSAEn</td><td>27</td><td>MIPS SIMD Architecture (MSA) Enable.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>MSA instructions and registers are disabled. Executing a MSA instruction causes a MSA Disabled exception.</td></tr>
<tr><td>1</td><td>MSA instructions and registers are enabled.</td></tr>
</table></td><td>R/W</td><td>0</td><td>Required if MSA Module is implemented.</td></tr>
<tr><td>0</td><td>26:5</td><td>Returns zeros on read.</td><td>R0</td><td>0</td><td>Reserved</td></tr>
</table>

Table 9.55 Config5 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>MVH</td><td>5</td><td>Move To/From High COP0 (MTHC0/MFHC0) instructions are implemented.
Currently these instructions are only required for Extended Physical Addressing (XPA).
.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>MTHC0 and MFHC0 are not supported. COP0 extensions do not exist.</td></tr>
<tr><td>1</td><td>MTHC0 and MFHC0 are supported. Extensions to 32-bit COP0 registers exist.</td></tr>
</table></td><td>R</td><td>Preset by hardware</td><td>Required for XPA (Release 5)</td></tr>
<tr><td>LLB</td><td>4</td><td>Load-Linked Bit software support is present.

Features enabled by $\mathit{Config5}_{LLB}$ =1 are recommended if Virtualization is supported, i.e., $\mathit{Config3}_{VZ}$=1.
.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>No new support added. Hardware is pre-Release 5 LL/SC compatible.</td></tr>
<tr><td>1</td><td>Additional support exists:
• ERETNC instruction added.
• COP0 LLAddr_{LLB} is mandatory.
• *LLbit* is software accessible through *LLAddr*[0].
• SC instruction behaviour is modified.</td></tr>
</table></td><td>R</td><td>Preset by hardware</td><td>Required if LLB support implemented (Release 5)</td></tr>
<tr><td>MRP</td><td>3</td><td>COP0 Memory Accessibility Attribute Registers, *MAAR* and *MAARI*, are present.
.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>*MAAR* and *MAARI* not present.</td></tr>
<tr><td>1</td><td>*MAAR* and *MAARI* present.
Software may program these registers to apply additional attributes to fetch/load/store access to memory/IO address ranges</td></tr>
</table></td><td>R</td><td>Preset by hardware</td><td>Required if MAAR(I) implemented (Release 5)</td></tr>
</table>

Table 9.55 Config5 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>UFR</td><td>2</td><td>This feature allows user-mode access to $Status_{FR}$ using CTC1 and CFC1 instructions.
.
<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>User-mode FR instructions not allowed.</td></tr><tr><td>1</td><td>User-mode FR instructions allowed.</td></tr></table></td><td>R/W if FIR_{UFRP} =1 else 0[2]</td><td>0</td><td>Required in (Release 5)</td></tr>
<tr><td>NF Exists</td><td>0</td><td>Indicates that the Nested Fault feature exists.
The Nested Fault feature allows recognition of faulting behavior within an exception handler.</td><td>R</td><td>Preset</td><td>Required if the Nested Fault feature exists.</td></tr>
</table>

[1] Note on $Config5_K$, Segment CCA determination: Table 9.56 below shows which field determines the CCA of a segment when $Config5_K$=0 or $Config5_K$=1, on implementations with/without a TLB, when the region is accessed unmapped.

[2] $Config5_{UFR}$ is R/W if an FPU is present, and if the User-mode FR changing feature is present, i.e. if FIR_{UFRP} is set. Otherwise $Config5_{UFR}$ is 0.

Table 9.56 $SegCtl0_K$ Segment CCA Determination

Segment	$Config5_K$=0	$Config5_K$=0	$Config5_K$=1
	No TLB	With TLB	
0	$Config_{K23}$	Undefined[1]	$SegCtl0_{C0}$
1	$Config_{K23}$	Undefined[1]	$SegCtl0_{C1}$
2	$SegCtl1_{C2}$	$SegCtl1_{C2}$	$SegCtl1_{C2}$
3	$Config_{K0}$	$Config_{K0}$	$SegCtl1_{C3}$
4	$Config_{KU}$	Undefined[1]	$SegCtl2_{C4}$
5	$Config_{KU}$	Undefined[1]	$SegCtl2_{C5}$

[1] Note: Reset state of these regions is mapped on implementations containing a TLB. Software must set $Config5_K$=1 if it is programming any of these segments to be used as unmapped on an implementation containing a TLB.

9.47 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.

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

Compliance Level: *Optional* prior to Release 5. *Required* in Release 5 if *Config5*$_{LLB}$=1.

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

If XPA, a Release5 feature that permits a PA size larger than 36 bits, is supported, is extended to support up to a 59-bit PA, as specified in the MIPS64 LLAddr instruction definition. The number of additional bits supported is a function of the physical address size. Any high-order bits greater than bit 31 of this register are accessed with MTHC0 and MFHC0 instructions.

Release 5 also provides software with the ability to read and clear the LLbit, which is set when an LL instruction is executed. The presence of LLB in *LLAddr* in Release 5 can be detected by software through *Config5*$_{LLB}$.

Figure 9-48 shows the format of the *LLAddr* register and Table 9.57 describes the *LLAddr* register fields for pre-Release 5 implementations.

Figure 9-49 shows the format of the *LLAddr* register; Table 10 describes the *LLAddr* register fields.

Figure 9-48 LLAddr Register Format (pre Release 5)

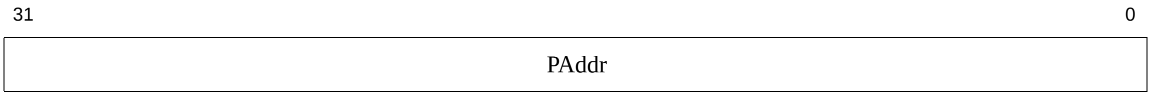


Table 9.57 LLAddr Register Field Descriptions (pre Release 5)

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
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

Figure 9-49 LLAddr Register Format (Release 5)

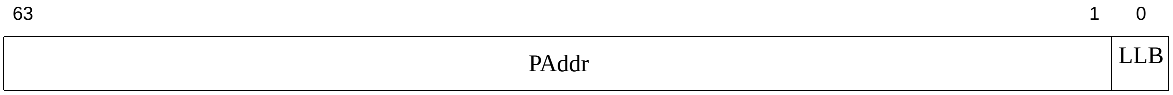


Table 10: LLAddr Register Field Descriptions (Release 5)

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
PAddr	63..1	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. *LLAddr*[1] is always aligned to PA[5], which implies that *PAddr* is always 32-byte aligned. In Release 5 implementations that do not support XPA (*Config3*$_{LPA}$ = 0), this field represents up to 36 bits of PA. *LLAddr* is then equivalent to a 32-bit register with *LLAddr*[31] equal to PA[35]. If *Config3*$_{LPA}$ = 1, then up to a 59-bit PA can be supported with *LLAddr*[54] = PA[59]. The number of physical address bits is implementation-specific. For the unimplemented address bits, writes are ignored and reads return zero.	R	Undefined	Optional
LLB	0	LLbit. *LLB* is set when the LL instruction is executed. The SC instructions and other hardware events may clear *LLB*. This field allows the LLbit to be software accessible. *LLB* can be cleared by software but cannot be set.	R/W	0	Required if *Config5*$_{LLB}$=1 (Release 5)

9.49 Memory Accessibility Attribute Register (CP0 Register 17, Select 1)

Compliance Level: *Optional*

The *MAAR* register is a read/write register included in Release 5 of the architecture that defines the accessibility attributes of physical address regions. In particular, *MAAR* defines whether a instruction fetch or data load can speculatively access such a region within the physical address bounds specified by the *MAAR.*

If the *MAAR* function yields a valid attribute, it will only override any equivalent attribute determined through other means, if it provides a more conservative outcome. For example, if the MMU yields a cacheable CCA, but *MAAR* yields a speculate attribute set to 0, then the access should not speculate as determined by the *MAAR* result. Similarly, if the MMU yields an uncacheable CCA, but *MAAR* yields a speculate attribute set to 1, then the access should not speculate.

In Release 5, the CCA of an access now defines speculation, along with MAAR. An access with a cacheable CCA is allowed to speculate. An access with uncacheable CCA on the other hand is not allowed to speculate unless the uncacheable CCA=7 (UCA) is used. The final speculative attribute is a combination of the CCA and MAAR as described above.

The address range specified by a *MAAR* may be used to specify an attribute for any region of the address space, whether memory (DRAM) or memory-mapped I/O.

MAAR is impacted by Extended Physical Addressing (XPA), a Release 5 feature, if included. If XPA is supported, then *MAAR* must be extended by additional physical address bits. To maintain atomicity of the write to an extended *MAAR*, two valid bits, VL and VH are required. The use of both bits is conditional on $PageGrain_{ELPA}$. While a write to the upper half of the extended MAAR could precede the write to the lower half to maintain atomicity, the required property of MTC0 to zero out extended PA bits prevents software from using this method.

It is recommended that Release 5 implementations of the architecture include the *MAAR* feature to allow architectural instead of implementation-dependent definition of speculation.

The Release 5 specification of *MAAR* requires that *MAAR* registers be paired, i.e., one specifies an upper bounds of the address range, and the other the lower bound. Future extensions to this specification may allow the flexibility of not pairing registers to allow fewer registers to be implemented with contiguous address ranges but different attribute types.

MAAR must be implemented in conjunction with *MAARI (MAAR Index*, CP0 Register 17, Sel 2). *MAARI* must be initialized with the appropriate *MAARI* register number before the *MAAR* is accessed with an MTC0 or MFC0. An EHB instruction is required to be placed between the write to *MAARI* and subsequent execution of MTC0 or MFC0 that specifies *MAAR*.

The presence of *MAAR* can be detected by software through $Config5_{MRP}$.

Figure 9.50 shows the format of the *MAAR* register; Table 9.1 describes the *MAAR* register fields.

Operation:

The pseudo-code shows a 3-pair MAAR implementation to determine speculation. It is recommended that implementations follow this description to enable portable software. As described, software must set the logical valid to 1 of each register in the pair to enable a MAAR pair. It may however, clear any one logical valid of the pair to invalidate the whole MAAR pair. Once both logical valids are set to 1, hardware factors in the speculate attribute of only the upper MAAR register with even index. The logical valid is determined as described in the pseudo-code below.

```
speculate$_{CCA}$ ← 0 // default is not to speculate
// Modify speculate attribute as per CCA of memory access (Release 5)
// Release 5: cached CCA and UCA speculates
if ((CCA == “cached”) or (CCA == “uncached-accelerated(UCA)”))
          speculate$_{CCA}$ ← 1
endif

// Now factor in MAAR
MAARmatch ← 0
speculate$_{MAAR}$ ← 1
// Example of 40-bit PA is 64KB aligned
PA_Align ← PA[39:16]
for (i=0; i<6; i=i+2) // assume 3 pairs
    // Factor in XPA (Extended Physical Addressing)
    MAAR[i]$_{V}$ = MAAR[i]$_{VL}$ and (MAAR[i]$_{VH}$ or not PageGrain$_{ELPA}$)
    MAAR[i+1]$_{V}$ = MAAR[i+1]$_{VL}$ and (MAAR[i+1]$_{VH}$ or not PageGrain$_{ELPA}$)
    if (MAAR[i]$_{V}$ and MAAR[i+1]$_{V}$) // both logical valids must be set to 1
        if ((MAAR[i][35:12] >= PA_Align) && // upper bound
                (MAAR[i+1][35:12] <= PA_Align))// lower bound
            speculate$_{MAAR}$ ← speculate$_{MAAR}$ and MAAR[i]$_{S}$
            MAARmatch ← 1
        endif
    endif
endfor

// if no MAAR is valid, or no MAAR match occurs, then speculate$_{MAAR}$ ← 0
speculate ← speculate$_{MAAR}$ and speculate$_{CCA}$ and MAARmatch
```

Programming Notes :

The purpose of *MAAR* is to control speculation on load or fetch access to memory and IO address. A load is considered speculative if it accesses memory prior to its being the oldest instruction to retire. A fetch typically always speculates on access to memory, while never speculating to IO. For implementations that support load or fetch speculation, support and initialization of *MAAR* is a requirement.

MAAR as defined has the following properties :

- If all *MAAR* instances are invalid, then no speculation is allowed. This allows the *MAAR* initialization to occur at any point of time without the risk of execution speculative (bad path) loads or fetches from issuing to IO addresses, with the tradeoff possibly being lower performance.
- If any *MAAR* region enables speculation, then accesses to physical addresses outside this *MAAR* region must be non-speculative, unless the physical address of the access matches against a *MAAR* region with speculation enabled. This access can then speculate.
- *MAAR* overlap is allowed : This allows non-speculative *MAAR* region to overlap a speculative *MAAR* region. For e.g., with this property, a non-speculative region can be overlayed on a speculative DRAM region with the use of just two MAAR pairs.

For software to enable a speculative region out of reset, it should first initialize *MAAR[*31:0] and then *MAAR*[63:32], assuming XPA is supported and to be enabled.

Software must follow the described method for *reprogramming* the state of a *MAAR* pair. The example assumes XPA is supported.

- Disable the *MAAR* pair by clearing $MAAR_{VL}$ and $MAAR_{VH}$. Accesses to the *MAAR* region become non-speculative.
- Program $PageGrain_{ELPA}$ as needed.
- Set $MAAR_{VL}$ along with other fields in *MAAR*[31:0]
- Initialize *MAAR*[63:32] if XPA is enabled.

Figure 9.50 MAAR Register Format

63		55	32
VH	0	ADDR	

31	16	15	12	2	1	0
ADDR			0		S	VL

Table 9.1 MAAR 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>VH</td><td>63</td><td>Valid, High 32-bits
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>MAAR[63:32] is not valid and should not modify behavior of any instruction fetch or data load.</td></tr>
<tr><td>1</td><td>MAAR[63:32] is valid and may modify behavior of any instruction fetch or data load that falls within the range of addresses specified by the MAAR register pair.</td></tr>
</table>
If XPA is supported and enabled, then both VL and VH must be factored in determining whether a MAAR register is valid :
$MAAR_V = MAAR_{VL}$ and $(MAAR_{VH}$ or not $PageGrain_{ELPA})$
If either valid bit (as calculated above) of the MAAR register pair is set to 0, then the pair is assumed invalid and thus will not modify behavior of a memory access. Software may thus clear one valid in one register of the MAAR pair to invalidate the MAAR comparison.</td><td>R/W</td><td>0</td><td>Required if XPA supported, otherwise Reserved</td></tr>
</table>

Table 9.1 MAAR 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>0</td><td>62:56</td><td>Reserved. Writes are ignored, read as 0.</td><td>R</td><td>0</td><td>Required</td></tr>
<tr><td>ADDR</td><td>55:12</td><td>Address bounds.
ADDR must always specify a physical address.
MAAR regions are at least 64KB aligned, and thus the least significant bit of ADDR is equal to PA[16].
If the register specifies the upper bound then any sourced address must be less than or equal to ADDR.
If the register specifies the lower bound then any sourced address must be greater than or equal to ADDR.
See MAAR Index (CP0 Register 17, Select 2) for method to determine which register is upper of lower in a pair.

MAAR[12] = PA[16]. This allows a 32-bit MAAR to specify 36-bits of PA, where MAAR[31] = PA[35].
If XPA is included, then ADDR may be extended to a maximum of 59 physical address bits. Unused PA bits should be treated as reserved. For this purpose, the MAAR register must be extended by upto an additional 32-bits, accessible by MTHC0 and MFHC0 which are defined in Release 5. An implementation that does not support XPA is limited to a 32-bit MAAR register.</td><td>R/W</td><td>Undefined</td><td>Required</td></tr>
<tr><td>0</td><td>15:2</td><td>Reserved. Writes are ignored, read as 0.</td><td>R</td><td>0</td><td>Required</td></tr>
<tr><td>S</td><td>1</td><td>Speculate.
If an access is qualified as non-speculative, it must be oldest in machine before being allowed to access memory or memory-mapped regions.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Instruction fetch or data load that matches MAAR register pair address range is <i>never</i> allowed to speculatively access address range.</td></tr>
<tr><td>1</td><td>Instruction fetch or data load that matches MAAR register pair address range <i>may</i> be allowed to speculate.</td></tr>
</table>
MAAR regions are allowed to overlap. The cumulative speculative attribute for overlapping regions is determined by ANDing individual valid MAAR pair speculation attributes.</td><td>R/W</td><td>Undefined</td><td>Required</td></tr>
</table>

Table 9.1 MAAR 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>VL</td><td>0</td><td>Valid, Low 32-bits
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>MAAR[31:0] is not valid and should not modify behavior of any instruction fetch or data load.</td></tr>
<tr><td>1</td><td>MAAR[31:0] is valid and may modify behavior of any instruction fetch or data load that falls within the range of addresses specified by the MAAR register pair.</td></tr>
</table>
If XPA is supported and enabled, then both VL and VH must be factored in determining whether a MAAR register is valid :
$MAAR_V = MAAR_{VL} \text{ and } (MAAR_{VH} \text{ or not } PageGrain_{ELPA})$

If either valid bit (as calculated above) of the MAAR register pair is set to 0, then the pair is assumed invalid and thus will not modify behavior of a memory access. Software may thus clear one valid bit in one register of the MAAR pair to invalidate the MAAR comparison.</td><td>R/W</td><td>0</td><td>Required</td></tr>
</table>

Table 9.2 shows how the valid attribute for a *MAAR* pair is determined from the cumulative individual *MAAR* register valids.

Table 9.2 Valid Determination for MAAR Pair

$MAAR[i]_V$ where i is even	$MAAR[i+1]_V$	Result
0	0	Result is invalid
0	1	Result is invalid
1	0	Result is invalid
1	1	Result is valid

Table 9.3 shows how the speculate attribute for a *MAAR* pair is determined by the cumulative individual speculate attributes.

Table 9.3 Speculate Determination for MAAR Pair

$MAAR[i]_S$ where i is even	$MAAR[i+1]_S$	Result
1	0/1	Valid access may speculate
0	0/1	Valid access may never speculate

9.50 Memory Accessibility Attribute Register Index (CP0 Register 17, Select 2)

Compliance Level: *Optional*

MAAR Index is used in conjunction with an implementation that supports *MAAR* registers (CP0 Register 17, Select 1). Multiple *MAAR* registers may be implemented - *MAAR Index* is used to specify a *MAAR* register number that may be accessed by software with an MTC0 or MFC0 instruction.

MAAR Index is always required if *MAAR* (CP0 Register 17, Select 1) is supported. This is because *MAAR*s are paired in Release 5, and thus there is always more than one *MAAR* register.

Prior to access by MTC0 or MFC0, software must set $MAARI_{INDEX}$ to the appropriate value.

Figure 9.51 shows the format of the *MAAR Index* register; Table 9.4 describes the *MAAR Index* register fields.

The presence of *MAARI* can be detected by software through $Config5_{MRP}$

Figure 9.51 MAAR Index Register Format

63 6	5 2 1 0
0	INDEX

Table 9.4 MAAR Index Register Field Descriptions

Fields		Description	Read/ Write	Reset State	Compliance
Name	Bits				
0	31:6	Reserved. Writes are ignored, read as 0.	R	0	Required

Table 9.4 MAAR 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>INDEX</td><td>5:0</td><td>MAAR index
The number of MAAR registers is greater than 1. INDEX specifies the MAAR register to access.
MAAR registers are paired. The least-significant bit of INDEX is encoded as follows to indicate which register of the pair is being accessed.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>This register specifies the upper address bound of the MAAR register pair.</td></tr>
<tr><td>1</td><td>This register specifies the lower address bound of the MAAR register pair.</td></tr>
</table>
The number of MAAR registers included is implementation dependent but must be an even number in Release 5. Software may write all ones to Index to determine the maximum value supported. Other than the all ones, if the value written is not supported, then Index is unchanged from previous value. The register range is always contiguous and starts at value 0.</td><td>R/W</td><td>Undefined</td><td>Required</td></tr>
</table>

9.51 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 and subsequent releases only) instruction whose address matches the *Watch* register address match conditions.For microMIPS implementations, it is implementation-dependent whether a match occurs if the second half-word overlaps a watched address and the first half-word does not overlap with the watched address.

Figure 9.52 shows the format of the *WatchLo* register; Table 9.5 describes the *WatchLo* register fields.

Figure 9.52 WatchLo Register Format

31 3	2	1	0
VAddr	I	R	W

Table 9.5 WatchLo Register Field Descriptions

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
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 9.5 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

9.52 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*) that 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 9.53 shows the format of the *WatchHi* register; Table 9.6 describes the *WatchHi* register fields.

Figure 9.53 WatchHi Register Format

31	30	29 28	27 26	25 24	23 16	15 12	11 3	2	1	0
M	G	WM	0	EAS	ASID	0	Mask	I	R	W

Table 9.6 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 an MTC0 or MFC0 select field value of '*n+1*'	R	Preset	Required

Table 9.6 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
WM	29:28	Reserved for Virtualization Module.	0	0	Reserved
EAS	25:24	If *Config4*$_{AE}$ = 1 then these bits extend the *ASID* field of this register. If *Config4*$_{AE}$ = 0 then Must be written as zero; returns zero on read.	If *Config4*$_{AE}$ = 1 then R/W else 0	If *Config4*$_{AE}$ = 1 then Undefined else 0	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	27..26, 15..12	Must be written as zero; returns zero on read.	0	0	Reserved

9.53 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.

9.54 Debug Register (CP0 Register 23, Select 0)

Compliance Level: *Optional*.

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

9.55 Debug2 Register (CP0 Register 23, Select 6)

Compliance Level: *Optional*.

The *Debug2* register is part of the EJTAG specification. Refer to that specification for the format and description of this register.

9.56 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.

9.56.1 Special Handling of the DEPC Register in Processors That Implement the MIPS16e ASE or microMIPS32 Base Architecture

In processors that implement the MIPS16e ASE or the microMIPS32 base architecture, 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:

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

“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:

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

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.

9.57 Performance Counter Register (CP0 Register 25)

Compliance Level: *Recommended*.

The 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 9.7 shows an example of two performance counters and how they map into the select values of the *PerfCnt* register.

Table 9.7 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 9.54 shows the format of the Performance Counter Control Register; Table 9.8 describes the Performance Counter Control Register fields.

Figure 9.54 Performance Counter Control Register Format

31	30	29 25	24 23	22 16	15	14 11	10 5	4	3	2	1	0
M	0	Impl	EC	0	PC TD	EventExt	Event	IE	U	S	K	EXL

Table 9.8 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 an MTC0 or MFC0 select field value of ‘<i>n</i>+2’ and ‘<i>n</i>+3’.</td><td>R</td><td>Preset by hardware</td><td>Required</td></tr>
<tr><td>0</td><td>30</td><td>Reserved for MIPS64/microMIPS64 processor. Unused on a MIPS32/microMIPS32 processor.</td><td>R</td><td>Preset by hardware</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>EC</td><td>24..23</td><td>Resarved for Virtualization Module.</td><td>0</td><td>0</td><td>Reserved</td></tr>
<tr><td>0</td><td>22..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 9.8 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 24 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 23 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 23, this bit enables event counting only when the <i>EXL</i> and <i>ERL</i> 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 9.8 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 <i>EXL</i> bit in the <i>Status</i> register is one and the <i>ERL</i> 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 <i>EXL</i> = 1, <i>ERL</i> = 0</td></tr>
<tr><td>1</td><td>Enable event counting while <i>EXL</i> = 1, <i>ERL</i> = 0</td></tr>
</table>
Counting is never enabled when the <i>ERL</i> bit in the <i>Status</i> register or the <i>DM</i> 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 9.55 shows the format of the Performance Counter Counter Register; Table 9.9 describes the Performance Counter Counter Register fields.

Figure 9.55 Performance Counter Counter Register Format

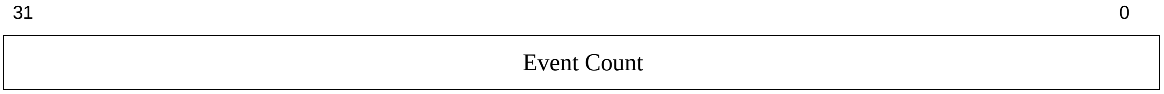


Table 9.9 Performance Counter Counter Register Field Descriptions

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 6.1.2.1 “Software Hazards and the Interrupt System” on page 84.

9.58 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.

9.59 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.

9.60 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 core specification for the format of this register and a description of the register fields. However, in all implementations. 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 power-up.If there is support for XPA (PA > 36 bits), the *PTagLo* field is extended to support up to a 59-bit PA, as specified in the MIPS64 definition. The number of additional bits supported is a function of the implemented physical address size. XPA is a Release 5 feature.

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.

Figure 9-56 Example TagLo Register Format

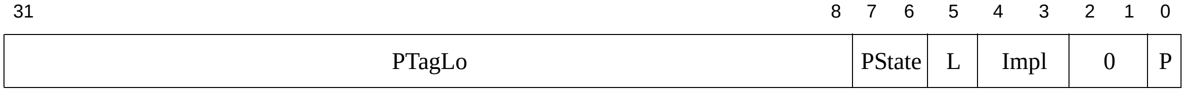


Table 10: Example TagLo Register Field Descriptions

Fields		Description	Read/ Write	Reset State	Compliance
Name	Bits				
PTagLo	31..8	Specifies the upper address bits of the cache tag. Refer to the processor-specific description for the detailed definition. With a page size of 4Kbytes, the field as shown can contain a physical address of up to 36 bits.	R/W	Undefined	Optional
PState	7:6	Specifies the state bits for the cache tag. Refer to the processor-specific description for the detailed definition.	R/W	Undefined	Optional
L	5	Specifies the lock bit for the cache tag. Refer to the processor-specific description for the detailed definition.	R/W	Undefined	Optional
Impl	4:3	This field is reserved for implementations.		Undefined	Optional
0	2:1	Must be written as zero; returns zero on read.	0	0	Reserved

Table 10: Example TagLo Register Field Descriptions (Continued)

Fields		Description	Read/ Write	Reset State	Compliance
Name	Bits				
P	0	Specifies the parity bit for the cache tag. Refer to the processor-specific description for the detailed definition.	R/W	Undefined	Optional

9.61 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.

9.62 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.

9.63 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.

9.64 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 9.57 shows the format of the *ErrorEPC* register; Table 9.1 describes the *ErrorEPC* register fields.

Figure 9.57 ErrorEPC Register Format

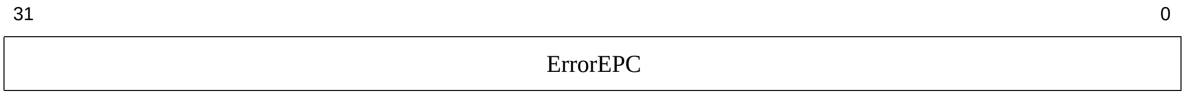


Table 9.1 ErrorEPC Register Field Descriptions

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

9.64.1 Special Handling of the ErrorEPC Register in Processors That Implement the MIPS16e ASE or microMIPS32 Base Architecture

In processors that implement the MIPS16e ASE or microMIPS32 base architecture, 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:

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

“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.

9.65 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.

The *DESAVE* register is meant to be used solely while in Debug Mode. If kernel mode software uses this register, it would conflict with debugging kernel mode software. For that reason, it is strongly recommended that kernel mode software not use this register. If the *KScratch** registers are implemented, kernel software can use those registers.

9.66 KScratch*n* Registers (CP0 Register 31, Selects 2 to 7)

Compliance Level: *Optional, KScratch1 and KScratch2 at selects 2, 3 are recommended.*

The *KScratchn* registers are read/write registers available for scratch pad storage by kernel mode software. These registers are 32bits in width for 32-bit processors and 64bits for 64-bit processors.

The existence of these registers is indicated by the *KScrExist* field within the *Config4* register. The *KScrExist* field specifies which of the selects are populated with a kernel scratch register.

Debug Mode software should not use these registers, instead debug software should use the *DESAVE* register. If EJTAG is implemented, select 0 should not be used for a *KScratch* register. Select 1 is being reserved for future debug use and should not be used for a *KScratch* register.

Figure 9.58 KScratch*n* Register Format

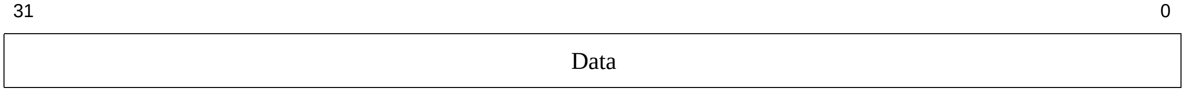


Table 9.2 KScratch*n* Register Field Descriptions

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
Data	31:0	Scratch pad data saved by kernel software.	R/W	Undefined	Optional

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/microMIPS32 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	
kseg1	0xA000 0000 through 0xBFFF FFFF	0x0000 0000 through 0x0x1FFF FFFF	

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

Segment Name	Virtual Address	Generates Physical Address	
		Status$_{ERL}$ = 0	Status$_{ERL}$ = 1

sseg
ksseg
kseg2
	`0xC000 0000`
	through
	`0xDFFF FFFF`
		`0xC000 0000`
		through
		`0xDFFF FFFF`	

kseg3	`0xE000 0000`
	through
	`0xFFFF FFFF`
		`0xE000 0000`
		through
		`0xFFFF FFFF`	

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

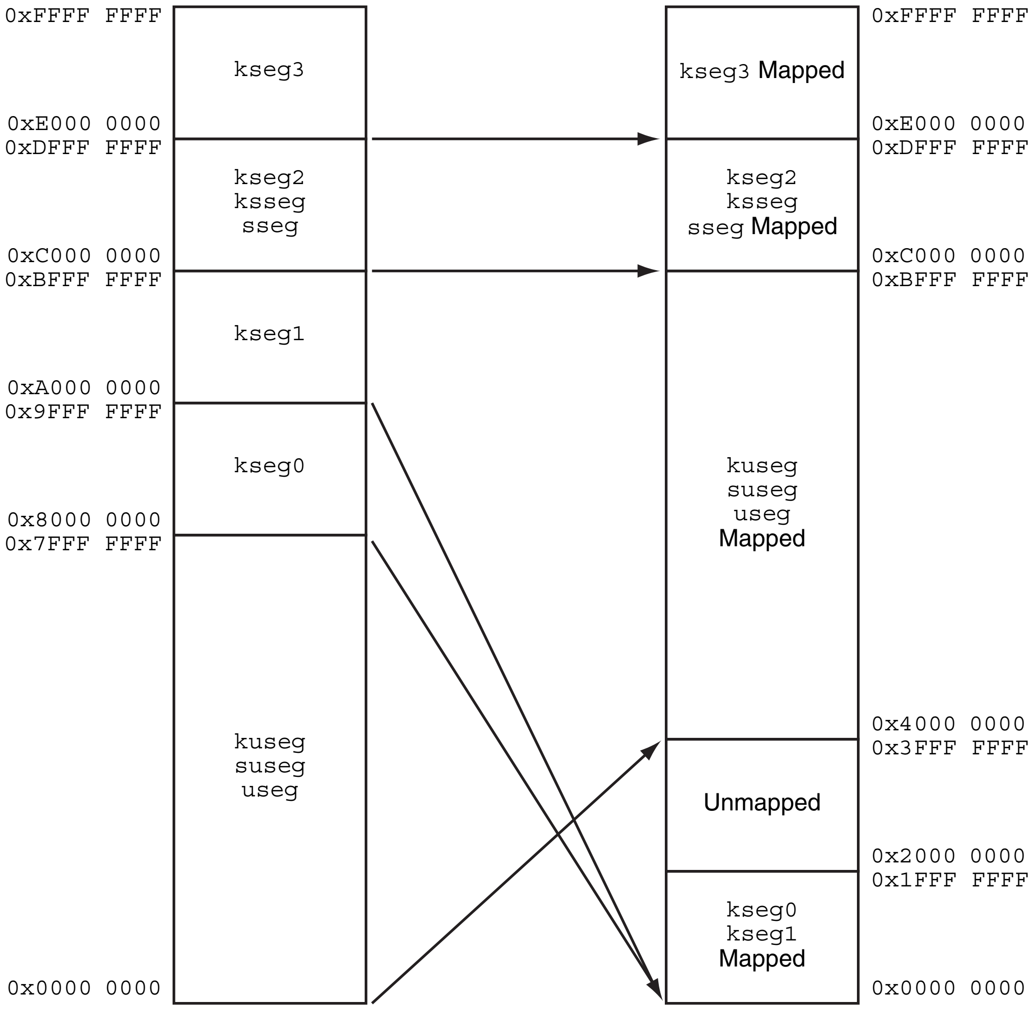
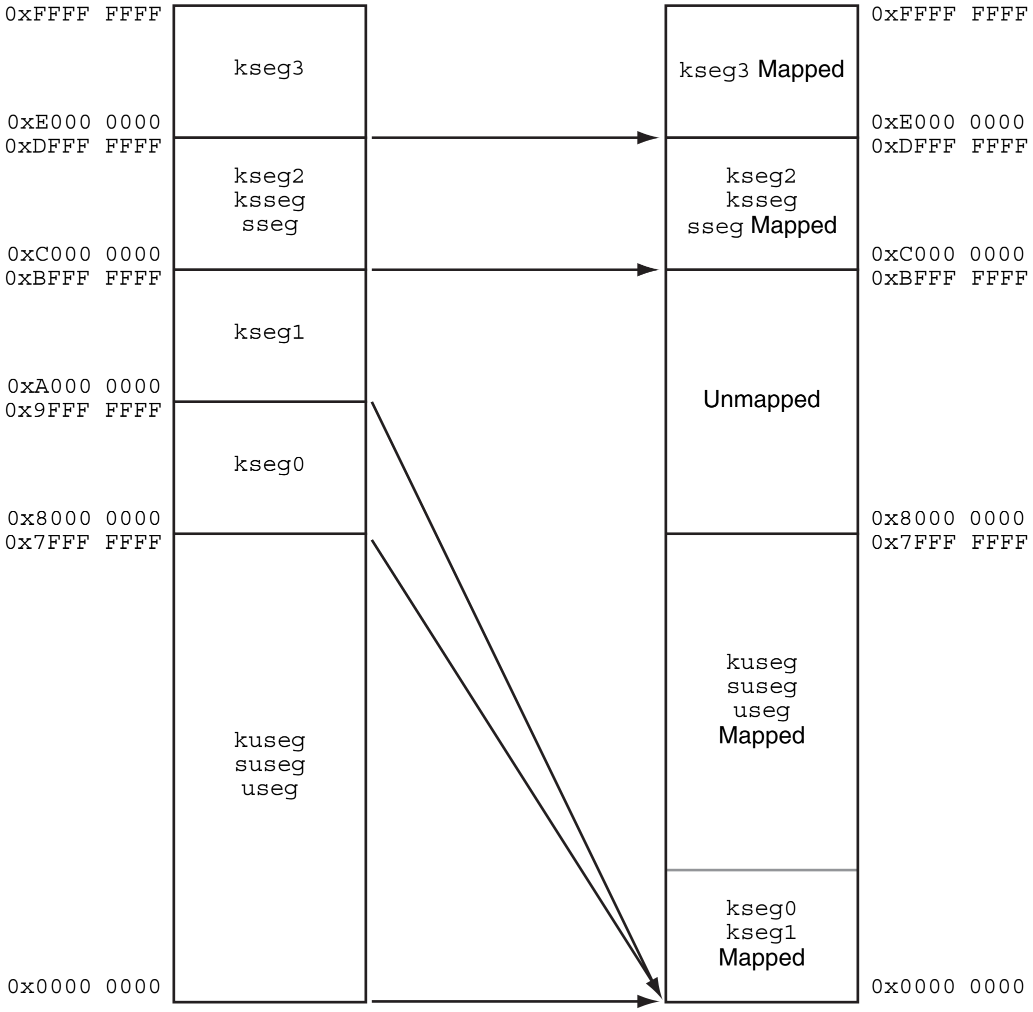


Figure A.2 Memory Mapping when ERL = 1



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 9.2 on page 130 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 9.2 on page 130 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

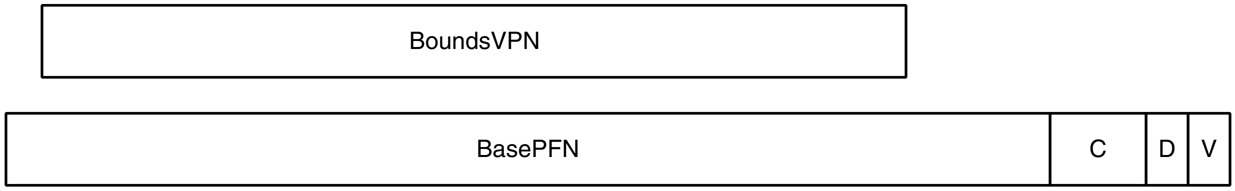
This section describes the architecture for a block address translation (BAT) mechanism that reuses much of the hardware and software interface that exists for a TLB-Based virtual address translation mechanism. This mechanism has the following features:

- 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.

A.3 Dual Variable-Page-Size and Fixed-Page-Size TLBs

Most MIPS CPU cores implement a fully associative Joint TLB. Unfortunately, such fully-associative structures can be slow, can require a large amount of logic components to implement and can dissipate a lot of power. The number of entries for a fully associative array that can be practically implemented is not large.

In high performance systems, it is desirable to minimize the frequency of TLB misses. In small and low-cost systems, it is desirable to keep the implementation costs of a TLB to a minimum. This section describes an optional alternative MMU configuration which decreases the implementation costs of a small TLB as well as allows for a TLB that can map a very large number of pages to be reasonably implemented.

A.3.1 MMU Organization

This alternative MMU configuration uses two TLB structures.

1. This first TLB is called the Fixed-Page-Size TLB or the FTLB.
   - At any one time, all entries within the FTLB use a shared, common page size.
   - The FTLB is not fully-associative, but rather set associative.
   - The number of ways per set is implementation specific.
   - The number of sets is implementation specific.
   - The common page size is also implementation specific.
   - The common page size is allowed to be software configurable. The choice of the common page size is done once for the entire FTLB, not on a per-entry basis. This configuration by software can only be done after a full flush/initialization of the FTLB, before there are any valid entries within the FTLB. Implementations are also allowed to support only one page size for the FTLB - in that case, the FTLB page size is fixed by hardware and not software configurable.

- The EHINV TLB invalidate feature is required for FTLB implementation. The legacy method of using reserved address values to represent invalid TLB entries is not guaranteed to work where the implementation can limit what addresses are allowable at a specific TLB index.

2. The second TLB is called the Variable-Page-Size TLB or the VTLB.
   - The choice of page size is done on a per-entry basis. That is, one VTLB entry can use a page size that is different from the size used by another VTLB entry.
   - The VTLB is fully-associative.
   - The number of entries is implementation specific.
   - The set of allowable page sizes for VTLB entries is implementation specific.

Just as for the JTLB, both the FTLB and VTLB are shared between the instruction stream and the data stream. For address translation, the virtual address is presented to both the FTLB and VTLB in parallel. Entries in both structures are accessed in parallel to search for the physical address.

The use of two TLB structures has these benefits:

- The implementation costs of building a set-associative TLB with many entries can be much less than that of implementing a large fully-associative TLB.
- The existence of a VTLB retains the capability of using large pages to map large sections of physical memory without consuming a large number of entries in the FTLB.

Random replacement of pages in the MMU happens mainly in the FTLB. In most operating systems, on-demand paging only uses one page size so the FTLB is sufficient for this purpose. Some of the address bits of the specified virtual address are used to index into the FTLB as appropriate for the chosen FTLB array size. The method of choosing which FTLB way to modify is implementation specific.

The VTLB is very similar to the JTLB. The *C0_PageMask* register is used to program the page size used for a particular VTLB entry.

The configuration of the FTLB is reflected in the FTLB fields within the new *Config4* register. The size of the VTLB is reflected in the $Config1_{MMUSize-1}$ field. The presence of the dual FTLB and VTLB is denoted by the value of 0x4 in $Config_{MT}$ register field. These registers are described in "Changes to the COP0 Registers" on page 277.

Most implementations would choose to build a VTLB with a smaller number of entries and a FTLB with a larger number of entries. This combination allows for many on-demand fixed-sized pages as well as for a small number of large address blocks to be simultaneously mapped by the MMU.

A.3.2 Programming Interface

The software programming interface used for the fully-associative JTLB is maintained as much as possible to decrease the amount of software porting.

Also for that purpose, each entry in the FTLB as well as the VTLB use one tag (VPN2) to map two physical pages (PFN), just as in the JTLB. The entries in either array are accessed through the *C0_EntryHi* and *C0_EntryLo0/1* registers.

Entries in either array (FTLB or VTLB) can be accessed with the TLBWI and TLBWR instructions.

The *PageMask* register is used to set the page size for the VTLB entries. This register is also used to choose which array (FTLB or VTLB) to write for the TLBWR instruction.

For the rest of this section, the following parameters are used:

3. FPageSize - the page size used by the FTLB entries

4. FSetSize - Number of entries in one way of the FTLB.

5. FWays - Number of ways within a set of the FTLB.

6. VIndex - Number of entries in the VTLB.

For the *C0_Index*, the *C0_Wired* registers, the TLBP, TLBR and TLBWI instructions; the VTLB occupies indices 0 to VIndex-1. The FTLB occupies indices VIndex to VIndex + (FSetSize * FWays)-1.

The TLBP instruction produces a value which can be used by the TLBWI instruction without modification by software. When referring to the FTLB, the value is the concatenation of the selected FTLB way and set, and incremented by the size of the VTLB. For example, {selected FTLB Way, selected FTLB Set} + VIndex.

If *C0_PageMask* is set to the page size used by the FTLB, the TLBWR instruction modifies entries within the FTLB or the VTLB. It is implementation specific whether the VTLB will be modified for this case.

How the FTLB set-associative array is indexed is implementation specific. In any indexing scheme, the least significant address bit that can be used for indexing is $\log_2$(FPageSize)+1. The number of index bits needed to select the correct set within the FTLB array is $\log_2$(FSetSize).

Since the FTLB array can be modified through the TLBWI instruction, it is possible for software to choose an inappropriate FTLB index value for the specified virtual address. In this case, it is implementation specific whether a Machine Check exception is generated for the TLBWI instruction.

The EHINV TLB entry invalidate feature is required for a FTLB. Since it is implementation defined as to whether a particular FTLB index value can be used for a specific virtual address, the legacy method of representing an invalid TLB entry by using a predefined address value is not guaranteed to work.

The method of choosing which FTLB way to modify is implementation specific.

If *C0_PageMask* is not set to the page size used by the FTLB, the TLBWR instruction modifies entries within the VTLB. The VTLB entry to be written is specified by the $\log_2$(VIndex) least significant bits of the *C0_Random* register value.

For both the TLBWR and TLBWI instruction, it is implementation specific whether both (FTLB and VTLB) arrays are checked for duplicate or overlapping entries and whether a Machine Check exception is generated for these cases.

A.3.2.1 Example with chosen FTLB and VTLB sizes

As an example, let’s assume an implementation chooses these values:

1. FPageSize - 4KB used by the FTLB entries

2. FSetSize - 128 in one way of the FTLB.

3. FWays - 4 ways within a set of the FTLB. (The FTLB has (128 sets x 4 ways/set) 512 entries, capable of mapping (512 entries x 2 pages/entry x 4KB/page) 4MB of address space simultaneously.

4. VIndex - 8 entries in the VTLB.

For the *C0_Index*, the *C0_Wired* registers, the TLBP, TLBR and TLBWI instructions; the VTLB occupies indices 0 to 7. The FTLB occupies indices 8 to 519.

The FTLB entries have a VPN2 field which starts at virtual address bit 12.

The least significant virtual address bit that can be used for FTLB indexing is virtual address 13. To index the FTLB set-associative array, 7 index bits are needed.

In this simple example, the design uses contiguous virtual address bits directly for indexing the FTLB (it does not create a hash for the FTLB indexing). The FTLB set-associative array is indexed using virtual address bits 19:13. The TLBWR instruction uses these address bits held in *C0_EntryHi.*

In this simple example, the design uses a cycle counter of 2 bits for way selection within the FTLB.

The *Random* register field within *C0_Random* is 3 bits wide to select the entry within the VTLB.

A.3.3 Changes to the TLB Instructions

TLBP

Both the VTLB and the FTLB are probed in parallel for the specified virtual address.

If the address hits in the VTLB, *C0_Index* specifies the entry within the VTLB (a value within 0 to VIndex-1).

If the address hits in the FTLB, *C0_Index* specifies the entry within the FTLB (a value within VIndex to VIndex+(FSetSize * FWays)-1). Which bits are used to encode the selected FTLB set as opposed to which bits are used to encode the selected FTLB way is implementation specific, but must match what is expected by the TLBWI instruction implementation. *C0_PageMask* reflects the page size used by the FTLB.

TLBR

Either a VTLB entry or a FTLB entry is read depending on the specified index in *C0_Index*.

Index values of 0 to VIndex-1 access the VTLB. Index values VIndex to VIndex+(FSetSize * FWays)-1 access the FTLB.

TLBWI

Either the VTLB or FTLB entry is written depending on the specified index in *C0_Index*.

Index values of 0 to VIndex-1 access the VTLB. Index values VIndex to VIndex+(FSetSize * FWays)-1 access the FTLB.

It is implementation specific if the hardware checks the *VPN2* field of *C0_EntryHi* is appropriate for the specified set within the FTLB. The implementation may generate a machine-check exception if the *VPN2* field is not appropriate for the specified set.

It is implementation-specific if the hardware checks both arrays (FTLB and VTLB) for valid duplicate or overlapping entries and if the hardware signals a Machine Check exception for these cases.

TLBWR

Either the VTLB or FTLB entry is written depending on the specified page size in C0_PageMask.

If *C0_PageMask* is set to any page size other than that used by the FTLB, the TLBWR instruction modifies a VTLB entry. The VTLB entry is specified by the Random register field within *C0_Random*.

If *C0_PageMask* is set to the page size used by the FTLB, the TLBWR modifies either a FTLB entry or a VLTB entry. It is implementation specific which array is modified. The FTLB set-associative array is indexed in an implementation-specific manner.

The method of selecting which FTLB way to modify is implementation specific.

It is implementation specific if the hardware checks both arrays (FTLB and VTLB) for valid duplicate or overlapping entries and if the hardware signals a Machine Check exception for these cases.

A.3.4 Changes to the COP0 Registers

C0_Config4 (CP0 Register 16, Select 4)

A new register introduced to reflect the FTLB configuration. $Config4_{MMUExtDef}$ register field must be set to a value of 2 or 3 to reflect that the Dual VTLB and FTLB configuration is implemented. If either *Config4* is not implemented or the $Config4_{MMUExtDef}$ field is not fixed to 2 or 3, the Dual VTLB/FTLB configuration is not implemented.

If $Config4_{MMUExtDef}$ is fixed to a value of 2 or 3, the *FTLBPageSize*, *FTLBWays* and *FTLBSets* fields reflect the FTLB configuration. Please refer to "Configuration Register 4 (CP0 Register 16, Select 4)" on page 221 for more detail on this register.

C0_Config1 (CP0 Register 16, Select 1)

If $Config4_{MMUExtDef}$ is fixed to a value of 2 or 3, the *MMUSize-1* register field is redefined to reflect only the size of the VTLB.

C0_Config (CP0 Register 16, Select 0)

If $Config_{MT}$ is fixed to a value of 4, the implemented MMU Type is the dual FTLB and VTLB configuration.

C0_Index (CP0 Register 0, Select 0)

If $Config4_{MMUExtDef}$ is fixed to a value of 2 or 3, the register is redefined in this way:

> The value held in the Index field can refer to either an entry in the FTLB or the VTLB. Index values of 0 to VIndex-1 access the VTLB. Index values VIndex to VIndex+(FSetSize * FWays)-1 access the FTLB. Which bits in the register field which encode the FTLB set as opposed to which bits encode the FTLB way is implementation specific, but must match what is expected by the TLBWI instruction implementation.

C0_Random (CP0 Register 1, Select 0)

If *Config4*$_{MMUExtDef}$ is fixed to a value of 2 or 3, the register is redefined in this way:

> If the value in *C0_PageMask* is not set to the page-size used by the FTLB, and a TLBWR instruction is executed, a VTLB entry is modified. The Random register field is used to select the VTLB entry which is modified.
>
> If the value in *C0_PageMask* is set to the page-size used by the FTLB, and a TLBWR instruction is executed, a FTLB entry or a VTLB entry is modified. It is implementation specific whether the *C0_RANDOM* register is used to select the FTLB entry.
>
> The upper bound of the *Random* register field value is VIndex.

C0_Wired (CP0 Register 6, Select 0)

If *Config4*$_{MMUExtDef}$ is fixed to a value of 2 or 3, the *Wired* register field can only hold a value of VIndex-1 or less. That is, only VTLB entries can be wired down.

C0_PageMask (CP0 Register 5, Select 0)

If *Config4*$_{MMUExtDef}$ is fixed to a value of 2 or 3, the register is redefined in this way:

> The *Mask* and *MaskX* field values determine whether the VTLB or the FTLB is modified by a TLBWR instruction.
>
> The *Mask* and *MaskX* register fields do not affect the TLB address match operation for FTLB entries. The page size used by the FTLB entries are specified by the *Config4*$_{FPageSize}$ register field.
>
> The software writeable bits in the *Mask* and *MaskX* fields reflect what page sizes are available in the VTLB. These fields do not reflect the page sizes which are available in the FTLB.

A.3.5 Software Compatibility

One of the main software visible changes introduced by this alternative MMU are the values reported in the *C0_Index* register. Previously, it was just a simple linear index. For this alternative MMU configuration, the value reflects both a selected way as well as a selected set when a FTLB entry is specified.

Fortunately, this Index value isn’t frequently generated by software nor read by software. Instead, the contents of the *C0_Index* register is generated by hardware upon a TLBP instruction. Software then just issues the TLBWI instruction once the *C0_EnLo** registers have been appropriately modified.

Another software visible change is that the *MMUSize-1* field no longer reports the entire MMU size. For TLB initialization and TLB flushing, the contents of *Config1*$_{MMUSize-1}$, *Config4*$_{FTLBWays}$ and *Config4*$_{FTLBSets}$ register fields must all be read to calculate the entire number of TLB entries that must be initialized. TLB initialization and flushing are the only times software needs to generate an Index value to write into the *C0_Index* register.

Only the VTLB entries may be wired down. This limitation is due to using some of the *EntryHi* VPN2 bits to index the FTLB array.

If a page using the FTLB page-size is to be wired down, that page must be programmed into the VTLB using the TLBWI instruction, as the TLBWR instruction would only access the FTLB in that situation and could not access any wired-down TLB entry. The TLBWI instruction is normally used for wired-down pages, so this restriction should not affect existing software.

The EHINV TLB entry invalidate feature is required for a FTLB. Since it is implementation-defined as to whether a particular FTLB index value can be used for a specific virtual address, the legacy method of representing an invalid TLB entry by using a predefined address value is not guaranteed to work.

Appendix B

Revision History

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 *Status*$_{BEV}$ = 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 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 *Cause*$_{PCI}$ 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 Cause$_{IP1..0}$ going to the EIC, rather than Status$_{IP1..0}$ • 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 SRSCtl$_{PSS}$ 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 29 bits of kuseg becomes uncached and unmapped.
2.62	January 2,009	• CCRes is accessed through $3 not $4 - *HWENA* register affected. • PCTD bit added to *C0_PerfCtl*.
2.70	January 22, 2009	MIPS Technologies-only release for internal review: • Added CP0 Reg 31, Select 2 & 3 as kernel scratch registers. • Added VTLB/FTLB optional MMU configuration to Appendix A and *Config4* register for these new MMU configurations • Added CDMM chapter, *CDMMBase* COP0 Register, CDMM bit in *C0_Config3*, FDCI bit in *C0_Cause* register and IPFDC field in *IntCtl* register.
2.71	January 28, 2009	MIPS Technologies-only release for internal review: • EIC mode - revision 2.70, was actually missing the new option of EIC driving an explicit vector offset (not using VectorNumbers). • Clarified the text and diagrams for the 3 EIC options - RIPL=VectorNum, Explicit VectorNum; Explicit VectorOffset.
2.72	April 20, 2009	MIPS Technologies-only release for internal review: • Table was incorrectly saying $\text{ECR}_{\text{ProbEn}}$ selected debug exception Vector. Changed to $\text{ECR}_{\text{ProbTrap}}$. • Added MIPS Technologies traditional meanings for CCA values. • Added list of COP2 instruction to COPUnusable Exception description. • Added statement that only uncached access is allowed to CDMM region. • Updated Exception Handling Operation pseudo-code for EIC Option_3 (EIC sends entire vector). •
2.73	April 22, 2009	MIPS Technologies-only release for internal review: • Fixed comments for ASE.
2.74	June 03, 2009	MIPS Technologies-only release for internal review: • Added CDMM Enable Bit in *CDMMBase* COP0 register • Reserved CCA values can be used to init TLB; just can’t be used for mapping.
2.75	June 12, 2009	MIPS Technologies-only release for internal review: • CDMMBase_Upper_Address Field doesn’t have a fixed reset value. • Added DSP State Disabled Exception to *C0_Cause* Exception Type table.
2.80	July 20, 2009	• FTLB and VTLB MMU configuration denoted by 0x4 in $Config_{MT}$ • Added TLBP -> TLBWI hazard • Added KScrExist field in *Config4*.

Revision	Date	Description
2.81	September 22, 2009	MIPS Technologies-only release for internal review: • ContextConfig Register description added. • Context Register description updated for SmartMIPS behavior. • EntryLo* register descriptions updated for RI & XI bits. • TLB description and pseudo-code updated for RI & XI bits. • PageMask register updated for RIE and XIE bits. • Config3 register updated for CTXTC and RXI bits. • Reserve MCU ASE bits in C0_Cause and C0_Status. • Clean up description for KScratch registers - selects 2&3 are recommended, but additional scratch registers are allowed.
2.82	January 19, 2010	MIPS Technologies-only release for internal review: • Added Debug2 register.
3.00	March 8, 2010	• RI/XI feature moved from SmartMIPS ASE. • microMIPS features added • MCU ASE features added. • XI and RI exceptions can be programmed to use their own exception codes instead of using TLBL code. • XI and RI can be independently implemented as XIE and RIE bits are allowed to be Read-Only. • TCOpt Register added to C0 Register list. • Added encoding (0x7) for 32 sets for one cache way.
3.05	July 07, 2010	• CMGCRBase register added. • Lower bits of C0_Context register allowed to be write-able if Config3.CTXTC=1 and Config3.SM=0.
3.10	July 27, 2010	• Explain the limits of the BadVPN2 field within Context register and the relationships with the writeable bits within ContextConfig register.
3.11	April 24, 2011	MIPS Technologies-only release for internal review: • FPR registers are UNPREDICTABLE after change of Status.FR bit. • 1004K did not support CCA=0 • Config4 - KScratch Registers, mention that select 1 is reserved for future debugger use. • Context Register - the bit subscripts describing which VA bits go into the BadVPN2 field was incorrect for the case when the ContextConfig register is used. The correct VA bits are 31:31-((X-Y)-1) for MIPS32.
3.12	April 28, 2011	• Changes for 64-bit architectures, no changes for 32-bit architectures.
3.13	November 10, 2011	MIPS Technologies-only release for internal review: • Nested Exception handling support. Config5 register added.
3.14	February 17, 2012	MIPS Technologies-only release for internal review: • Segmentation Control, EVA scheme added: a) Adds SegCfg0, SegCfg1, SegCfg2 registers b) SegCtl - Modifies EBase, Config3. • TLB Invalidate feature.
3.50	September 20, 2012	• Added BadInstr & BadInstrP registers. • Added extended ASID field in EntryHi and WatchHi. • Added Hardware Page Table Walking Feature
3.51	October 2, 2012	MIPS Technologies-only release for internal review: • Hardware Page Table Walker - previous description wasn't fully correct. PTEVld bit is only used for Directory PTE entries as leaf PTE entries are always loaded from memory. • Added TLB init routine for SegmentationControl/EVA.

Revision	Date	Description
3.52	November 12, 2012	• SegCtl Overlay segment(s) are available in kernelmode. Re-iterate that. • FTLB/VTLB - if PageMask set to FTLB size, allowed to modify VTLB. • implementation-dependent whether *Watch* Registers match on 2nd half of microMIPS instruction. • Hardware Page Table Walker - give example of 4-byte PTE. • Hardware Page Table Walker - added option so Directory PTE entries can represent power-of-4 memory region, using Dual Page Method. • Optional PageGrain.MCCause field to record different types of Machine Check Exceptions.
5.00	December 14, 2012	• R5 changes - include MSA and Virtualization registers and control bits in Register table. • R5 changes - include MSA and Virtualization exceptions in Cause exception types. • R5 changes - MT and DSP ASEs -> Modules • R5 changes - MDMX now deprecated. • “Preset” -> “Preset by hardware”
5.01	December 16, 2012	• No technical content change: • Update cover logos • Update copyright text
5.02	April 2012	• R5 changes: FR=1 64-bit FPU register model required is required, if floating point is supported. Section 3.5.2 64-bit FPR Enable. Table 9.41 Status Register Field Descriptions, FR (floating point register mode) bit. • R5 extension: Table 9.57 Config Register Field Descriptions, AR bit (Architecture revision level). AR=1 indicates Release 2 or Release 3 or Release 5. Like Release 3, all features introduced in Release 5 are optional. • Correction: Table 9.59 BPG, Big Pages feature, not supported in MIPS32, only in MIPS64
5.03	September 9, 2013	• Update document template to reflect new ownership by Imagination Technologies.
5.04	September 29, 2013	MAAR initial version • Add MAAR, MAARI and Config5.MRP • Table 1.1 typo. Speculate=1 should not contain comment about oldest in machine. Meaningful to Speculate=0. Moved outside sub-table. • Added a condition to sw write of MAARI.Index - write of all 1s returns the largest value supported.
5.04	January 15, 2013	XPA initial Version. • Add extended EntryLo0/1, LLAddr, TagLo, CDMMBase, CMGCRBase • PageGrain.ELPA, Config3.LPA, Config5.MVH • Remove comment about SW having to initialize the extension bits (of EntryLo,TagLo) if PageGrain.ELPA=0. HW had been asked to reset to 0, but the current POR solution is for mtc0 to 0 out the extension bits that are writeable. HW is responsible for zeroing out read-only bits on operation that updates the bits. • Remove CDMMBase and CMGCRBase from list of COP0 registers requiring extensions. The two registers support upto 36b PA which is sufficient for their purpose. Less testing. • Add a config bit, Config5.MVH, for mth/mfhc0. Since mth/mfhc0 may be used independently of XPA in the future, it is easier for software to query one bit instead of multiple. Further Config3.LPA=1 on 64-bit HW need not mean mthc0/mfhc0 are implemented.
5.05	November 14, 2014	Updated MAAR and MAARI register descriptions.