MIPS® Architecture for Programmers Volume IV-h: The MCU Application Specific Extension to the MIPS32® Architecture

Document Number: MD00834
Revision 1.03
September 9, 2013

Table of Contents

Chapter 1: About This Book 7
1.1: Typographical Conventions 7
1.1.1: Italic Text 8
1.1.2: Bold Text 8
1.1.3: Courier Text 8
1.2: UNPREDICTABLE and UNDEFINED 8
1.2.1: UNPREDICTABLE 8
1.2.2: UNDEFINED 9
1.2.3: UNSTABLE 9
1.3: Special Symbols in Pseudocode Notation 9
1.4: For More Information 12

Chapter 2: Guide to the Instruction Set 13
2.1: Understanding the Instruction Fields 13
2.1.1: Instruction Fields 14
2.1.2: Instruction Descriptive Name and Mnemonic 15
2.1.3: Format Field 15
2.1.4: Purpose Field 16
2.1.5: Description Field 16
2.1.6: Restrictions Field 16
2.1.7: Operation Field 17
2.1.8: Exceptions Field 17
2.1.9: Programming Notes and Implementation Notes Fields 18
2.2: Operation Section Notation and Functions 18
2.2.1: Instruction Execution Ordering 18
2.2.2: Pseudocode Functions 18
2.2.2.1: Coprocessor General Register Access Functions 18
2.2.2.2: Memory Operation Functions 20
2.2.2.3: Floating Point Functions 23
2.2.2.4: Miscellaneous Functions 26
2.3: Op and Function Subfield Notation 27
2.4: FPU Instructions 27

Chapter 3: The MCU Application-Specific Extension to the MIPS32® and microMIPS32TMArchitecture 29
3.1: Base Architecture Requirements 29
3.2: Software Detection of the ASE 29
3.3: Compliance and Subsetting 29
3.4: Overview of the MCU ASE 29
3.4.1: Interrupt Delivery 29
3.4.2: Interrupt Latency Reduction 29
3.4.2.1: Interrupt Vector Prefetching 30
3.4.2.2: Automated Interrupt Prologue 30
3.4.2.3: Automated Interrupt Epilogue 30
3.4.2.4: Interrupt Chaining 30
3.4.3: I/O Device Programming 30

Chapter 4: The MCU Instruction Set 31
4.1: IRET 31
4.2: ASET 31
4.3: ACLR 31

Chapter 5: The MCU Privileged Resource Architecture 41
5.1: Introduction 41
5.2: The MCU System Coprocessor 41
5.3: Interrupt Delivery 41
5.3.1: Number of Hardware Interrupts 41
5.3.1.1: Changes to Vectored Interrupt Mode 41
5.3.1.2: Changes to External Interrupt Controller Mode 41
5.4: Interrupt Handling 42
5.4.1: Interrupt Vector Prefetching 42
5.4.1.1: Historical Behavior of Pipelines with In-Order Completion 42
5.4.1.2: Historical Behavior of Pipelines with Out-of-Order Completion 42
5.4.1.3: New Feature - Speculative Prefetching 43
5.4.2: Interrupt Automated Prologue (IAP) 44
5.4.2.1: IAP Conditions 44
5.4.2.2: IAP Operation 45
5.4.2.3: Exceptions during IAP 46
5.4.3: Interrupt Automated Epilogue (IAE) 46
5.4.3.1: IAE Conditions 46
5.4.3.2: IAE Operation 46
5.4.3.3: Exceptions during IAE 47
5.4.4: Interrupt Chaining 47
5.4.4.1: Interrupt Chaining Conditions 48
5.5: Modified CP0 Registers 48
5.5.1: CP0 Register Summary 48
5.5.2: Status Register (CP Register 12, Select 0) 48
5.5.3: IntCtl (CP0 Registers 12, Select 1) 55
5.5.4: View_IPL Register (CP0 Register 12, Select 4) 60
5.5.5: SRSMap2 Register (CP0 Register 12, Select 5) 60
5.5.6: Cause Register (CP0 Register 13, Select 0) 61
5.5.7: View_RIPL Register (CP0 Register 13, Select 4) 66
5.5.8: Config Register 3 (CP0 Register 16, Select 3) 67

Appendix A: Revision History 73

List of Figures

Figure 2.1: Example of Instruction Description 14
Figure 2.2: Example of Instruction Fields 15
Figure 2.3: Example of Instruction Descriptive Name and Mnemonic 15
Figure 2.4: Example of Instruction Format 15
Figure 2.5: Example of Instruction Purpose 16
Figure 2.6: Example of Instruction Description 16
Figure 2.7: Example of Instruction Restrictions 17
Figure 2.8: Example of Instruction Operation 17
Figure 2.9: Example of Instruction Exception 17
Figure 2.10: Example of Instruction Programming Notes 18
Figure 2.11: COP_LW Pseudocode Function 19
Figure 2.12: COP_LD Pseudocode Function 19
Figure 2.13: COP_SW Pseudocode Function 19
Figure 2.14: COP_SD Pseudocode Function 20
Figure 2.15: CoprocessorOperation Pseudocode Function 20
Figure 2.16: AddressTranslation Pseudocode Function 20
Figure 2.17: LoadMemory Pseudocode Function 21
Figure 2.18: StoreMemory Pseudocode Function 21
Figure 2.19: Prefetch Pseudocode Function 22
Figure 2.20: SyncOperation Pseudocode Function 23
Figure 2.21: ValueFPR Pseudocode Function 23
Figure 2.22: StoreFPR Pseudocode Function 24
Figure 2.23: CheckFPException Pseudocode Function 25
Figure 2.24: FPConditionCode Pseudocode Function 25
Figure 2.25: SetFPConditionCode Pseudocode Function 25
Figure 2.26: SignalException Pseudocode Function 26
Figure 2.27: SignalDebugBreakpointException Pseudocode Function 26
Figure 2.28: SignalDebugModeBreakpointException Pseudocode Function 26
Figure 2.29: NullifyCurrentInstruction PseudoCode Function 27
Figure 2.30: JumpDelaySlot Pseudocode Function 27
Figure 2.31: PolyMult Pseudocode Function 27
Figure 5-1: Status Register Format 49
Figure 5-2: IntCtl Register Format 55
Figure 5-3: View_IPL Register Format 60
Figure 5-4: SRSMap Register Format 61
Figure 5-5: Cause Register Format 61
Figure 5-6: View_RIPL Register Format 66
Figure 5-7: Config3 Register Format 67

List of Tables

Table 1.1: Symbols Used in Instruction Operation Statements.. 9
Table 2.1: AccessLength Specifications for Loads/Stores 22
Table 5.1: Typical Interrupt Handling Flow in Pipelined Implementation with Out-of-Order Completion 43
Table 5.2: Interrupt Handling Flow with Speculative Prefetching... 44
Table 5.3: MCU Changes to Coprocessor 0 Registers in Numerical Order.. 48
Table 5.4: Status Register Field Descriptions 49
Table 5.5: IntCtl Register Field Descriptions.. 56
Table 5.6: View_IPL Register Field Descriptions 60
Table 5.7: SRSMap Register Field Descriptions.. 61
Table 5.8: Cause Register Field Descriptions... 61
Table 5.9: Cause Register ExcCode Field 65
Table 5.10: View_RIPL Register Field Descriptions 66
Table 5.11: Config3 Register Field Descriptions.. 67

Chapter 1

About This Book

The MIPS® Architecture for Programmers Volume IV-h: The MCU Application Specific Extension to the MIPS32® 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 MIPS32® Architecture
- Volume I-B describes conventions used throughout the document set, and provides an introduction to the microMIPS32™ Architecture
- Volume II-A provides detailed descriptions of each instruction in the MIPS32® instruction set
- Volume II-B provides detailed descriptions of each instruction in the microMIPS32™ 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 MIPS32® 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*, *registers*, that are important from a software perspective (for instance, address bits used by software, and programmable fields and registers), and various *floating point instruction formats*, such as *S*, *D*, and *PS*
- is used for the memory access types, such as *cached* and *uncached*

1.1.2 Bold Text

- represents a term that is being **defined**
- is used for **bits** and **fields** that are important from a hardware perspective (for instance, **register** bits, which are not programmable but accessible only to hardware)
- is used for ranges of numbers; the range is indicated by an ellipsis. For instance, **5..1** indicates numbers 5 through 1
- is used to emphasize **UNPREDICTABLE** and **UNDEFINED** behavior, as defined below.

1.1.3 Courier Text

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

1.2 UNPREDICTABLE and UNDEFINED

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

1.2.1 UNPREDICTABLE

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

UNPREDICTABLE results or operations have several implementation restrictions:

- Implementations of operations generating **UNPREDICTABLE** results must not depend on any data source (memory or internal state) which is inaccessible in the current processor mode
- **UNPREDICTABLE** operations must not read, write, or modify the contents of memory or internal state which is inaccessible in the current processor mode. For example, **UNPREDICTABLE** operations executed in user mode must not access memory or internal state that is only accessible in Kernel Mode or Debug Mode or in another process

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

1.2.2 UNDEFINED

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

UNDEFINED operations or behavior has one implementation restriction:

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

1.2.3 UNSTABLE

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

UNSTABLE values have one implementation restriction:

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

1.3 Special Symbols in Pseudocode Notation

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

Table 1.1 Symbols Used in Instruction Operation Statements

Symbol	Meaning
←	Assignment
=, ≠	Tests for equality and inequality
‖	Bit string concatenation
x^y	A *y*-bit string formed by *y* copies of the single-bit value *x*
b#n	A constant value *n* in base *b*. For instance 10#100 represents the decimal value 100, 2#100 represents the binary value 100 (decimal 4), and 16#100 represents the hexadecimal value 100 (decimal 256). If the "b#" prefix is omitted, the default base is 10.
0bn	A constant value *n* in base *2*. For instance 0b100 represents the binary value 100 (decimal 4).
0xn	A constant value *n* in base *16*. For instance 0x100 represents the hexadecimal value 100 (decimal 256).
$x_{y\ z}$	Selection of bits *y* through *z* of bit string *x*. Little-endian bit notation (rightmost bit is 0) is used. If *y* is less than *z*, this expression is an empty (zero length) bit string.
+, −	2’s complement or floating point arithmetic: addition, subtraction

Table 1.1 Symbols Used in Instruction Operation Statements (Continued)

Symbol	Meaning
*, ×	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*
COC[z]	Coprocessor unit *z* condition signal
Xlat[x]	Translation of the MIPS16e GPR number *x* into the corresponding 32-bit GPR number
BigEndianMem	Endian mode as configured at chip reset (0 →Little-Endian, 1 → Big-Endian). Specifies the endianness of the memory interface (see LoadMemory and StoreMemory pseudocode function descriptions), and the endianness of Kernel and Supervisor mode execution.
BigEndianCPU	The endianness for load and store instructions (0 → Little-Endian, 1 → Big-Endian). In User mode, this endianness may be switched by setting the *RE* bit in the *Status* register. Thus, BigEndianCPU may be computed as (BigEndianMem XOR ReverseEndian).
ReverseEndian	Signal to reverse the endianness of load and store instructions. This feature is available in User mode only, and is implemented by setting the *RE* bit of the *Status* register. Thus, ReverseEndian may be computed as (SR_{RE} and User mode).

Table 1.1 Symbols Used in Instruction Operation Statements (Continued)

<table>
<tr><th>Symbol</th><th>Meaning</th></tr>
<tr><td>LLbit</td><td>Bit of virtual state used to specify operation for instructions that provide atomic read-modify-write. LLbit is set when a linked load occurs and is tested by the conditional store. It is cleared, during other CPU operation, when a store to the location would no longer be atomic. In particular, it is cleared by exception return instructions.</td></tr>
<tr><td>I:,
I+n:,
I-n:</td><td>This occurs as a prefix to Operation description lines and functions as a label. It indicates the instruction time during which the pseudocode appears to "execute." Unless otherwise indicated, all effects of the current instruction appear to occur during the instruction time of the current instruction. No label is equivalent to a time label of I. Sometimes effects of an instruction appear to occur either earlier or later — that is, during the instruction time of another instruction. When this happens, the instruction operation is written in sections labeled with the instruction time, relative to the current instruction I, in which the effect of that pseudocode appears to occur. For example, an instruction may have a result that is not available until after the next instruction. Such an instruction has the portion of the instruction operation description that writes the result register in a section labeled I+1.
The effect of pseudocode statements for the current instruction labelled I+1 appears to occur "at the same time" as the effect of pseudocode statements labeled I for the following instruction. Within one pseudocode sequence, the effects of the statements take place in order. However, between sequences of statements for different instructions that occur "at the same time," there is no defined order. Programs must not depend on a particular order of evaluation between such sections.</td></tr>
<tr><td>PC</td><td>The Program Counter 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 PC during an instruction time. If no value is assigned to PC 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 PC during the instruction time of the instruction in the branch delay slot.
In the MIPS Architecture, the PC value is only visible indirectly, such as when the processor stores the restart address into a GPR on a jump-and-link or branch-and-link instruction, or into a Coprocessor 0 register on an exception. The PC value contains a full 32-bit address all of which are significant during a memory reference.</td></tr>
<tr><td>ISA Mode</td><td>In processors that implement the MIPS16e Application Specific Extension or the microMIPS base architectures, the ISA Mode 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 ISA Mode value is only visible indirectly, such as when the processor stores a combined value of the upper bits of PC and the ISA Mode into a GPR on a jump-and-link or branch-and-link instruction, or into a Coprocessor 0 register on an exception.</td></tr>
<tr><td>PABITS</td><td>The number of physical address bits implemented is represented by the symbol PABITS. As such, if 36 physical address bits were implemented, the size of the physical address space would be $2^{PABITS} = 2^{36}$ bytes.</td></tr>
</table>

Table 1.1 Symbols Used in Instruction Operation Statements (Continued)

Symbol	Meaning
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 MIPSr3) 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 For More Information

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

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

Chapter 2

Guide to the Instruction Set

This chapter provides a detailed guide to understanding the instruction descriptions, which are listed in alphabetical order in the tables at the beginning of the next chapter.

2.1 Understanding the Instruction Fields

Figure 2.1 shows an example instruction. Following the figure are descriptions of the fields listed below:

- “Instruction Fields” on page 14
- “Instruction Descriptive Name and Mnemonic” on page 15
- “Format Field” on page 15
- “Purpose Field” on page 16
- “Description Field” on page 16
- “Restrictions Field” on page 16
- “Operation Field” on page 17
- “Exceptions Field” on page 17
- “Programming Notes and Implementation Notes Fields” on page 18

Figure 2.1 Example of Instruction Description

Instruction Mnemonic and Descriptive Name →

Example Instruction Name	**EXAMPLE**

EXAMPLE

Instruction encoding constant and variable field names and values →

31–26	25–21	20–16	15–11	10–6	5–0
SPECIAL 000000	0	rt	rd	0 00000	EXAMPLE 000000
6	5	5	5	5	6

Architecture level at which instruction was defined/redefined → **MIPS32**

Assembler format(s) for each definition →

Format: `EXAMPLE fd,rs,rt` **MIPS32**

Short description →

Purpose: Example Instruction Name

To execute an EXAMPLE op.

Symbolic description →

Description: `GPR[rd] ← GPR[r]s exampleop GPR[rt]`

Full description of instruction operation →

This section describes the operation of the instruction in text, tables, and illustrations. It includes information that would be difficult to encode in the Operation section.

Restrictions on instruction and operands →

Restrictions:

This section lists any restrictions for the instruction. This can include values of the instruction encoding fields such as register specifiers, operand values, operand formats, address alignment, instruction scheduling hazards, and type of memory access for addressed locations.

High-level language description of instruction operation →

Operation:

```
/* This section describes the operation of an instruction in */
/* a high-level pseudo-language. It is precise in ways that  */
/* the Description section is not, but is also missing       */
/* information that is hard to express in pseudocode.        */
temp    ← GPR[rs] exampleop GPR[rt]
GPR[rd] ← temp
```

Exceptions that instruction can cause →

Exceptions:

A list of exceptions taken by the instruction

Notes for programmers →

Programming Notes:

Information useful to programmers, but not necessary to describe the operation of the instruction

Notes for implementors →

Implementation Notes:

Like *Programming Notes*, except for processor implementors

2.1.1 Instruction Fields

Fields encoding the instruction word are shown in register form at the top of the instruction description. The following rules are followed:

- The values of constant fields and the *opcode* names are listed in uppercase (SPECIAL and ADD in Figure 2.2). Constant values in a field are shown in binary below the symbolic or hexadecimal value.
- All variable fields are listed with the lowercase names used in the instruction description (*rs*, *rt*, and *rd* in Figure 2.2).
- Fields that contain zeros but are not named are unused fields that are required to be zero (bits 10:6 in Figure 2.2). If such fields are set to non-zero values, the operation of the processor is **UNPREDICTABLE**.

Figure 2.2 Example of Instruction Fields

31 .. 26	25 .. 21	20 .. 16	15 .. 11	10 .. 6	5 .. 0
SPECIAL 000000	rs	rt	rd	0 00000	ADD 100000
6	5	5	5	5	6

2.1.2 Instruction Descriptive Name and Mnemonic

The instruction descriptive name and mnemonic are printed as page headings for each instruction, as shown in Figure 2.3.

Figure 2.3 Example of Instruction Descriptive Name and Mnemonic

Add Word	**ADD**

2.1.3 Format Field

The assembler formats for the instruction and the architecture level at which the instruction was originally defined are given in the *Format* field. If the instruction definition was later extended, the architecture levels at which it was extended and the assembler formats for the extended definition are shown in their order of extension (for an example, see C.cond fmt). The MIPS architecture levels are inclusive; higher architecture levels include all instructions in previous levels. Extensions to instructions are backwards compatible. The original assembler formats are valid for the extended architecture.

Figure 2.4 Example of Instruction Format

Format:	`ADD fd,rs,rt`	**MIPS32**

The assembler format is shown with literal parts of the assembler instruction printed in uppercase characters. The variable parts, the operands, are shown as the lowercase names of the appropriate fields. The architectural level at which the instruction was first defined, for example “MIPS32” is shown at the right side of the page.

There can be more than one assembler format for each architecture level. Floating point operations on formatted data show an assembly format with the actual assembler mnemonic for each valid value of the *fmt* field. For example, the ADD fmt instruction lists both ADD.S and ADD.D.

The assembler format lines sometimes include parenthetical comments to help explain variations in the formats (once again, see C.cond.fmt). These comments are not a part of the assembler format.

2.1.4 Purpose Field

The *Purpose* field gives a short description of the use of the instruction.

Figure 2.5 Example of Instruction Purpose

Purpose: Add Word

To add 32-bit integers. If an overflow occurs, then trap.

2.1.5 Description Field

If a one-line symbolic description of the instruction is feasible, it appears immediately to the right of the *Description* heading. The main purpose is to show how fields in the instruction are used in the arithmetic or logical operation.

Figure 2.6 Example of Instruction Description

Description: `GPR[rd] ← GPR[rs] + GPR[rt]`

The 32-bit word value in GPR *rt* is added to the 32-bit value in GPR *rs* to produce a 32-bit result.

- If the addition results in 32-bit 2's complement arithmetic overflow, the destination register is not modified and an Integer Overflow exception occurs.
- If the addition does not overflow, the 32-bit result is placed into GPR *rd*.

The body of the section is a description of the operation of the instruction in text, tables, and figures. This description complements the high-level language description in the *Operation* section.

This section uses acronyms for register descriptions. "GPR *rt*" is CPU general-purpose register specified by the instruction field *rt*. "FPR *fs*" is the floating point operand register specified by the instruction field *fs*. "CP1 register *fd*" is the coprocessor 1 general register specified by the instruction field *fd*. "*FCSR*" is the floating point *Control / Status* register.

2.1.6 Restrictions Field

The *Restrictions* field documents any possible restrictions that may affect the instruction. Most restrictions fall into one of the following six categories:

- Valid values for instruction fields (for example, see floating point ADD fmt)
- ALIGNMENT requirements for memory addresses (for example, see LW)
- Valid values of operands (for example, see ALNV.PS)
- Valid operand formats (for example, see floating point ADD fmt)

- Order of instructions necessary to guarantee correct execution. These ordering constraints avoid pipeline hazards for which some processors do not have hardware interlocks (for example, see MUL).
- Valid memory access types (for example, see LL/SC)

Figure 2.7 Example of Instruction Restrictions

Restrictions:

None

2.1.7 Operation Field

The *Operation* field describes the operation of the instruction as pseudocode in a high-level language notation resembling Pascal. This formal description complements the *Description* section; it is not complete in itself because many of the restrictions are either difficult to include in the pseudocode or are omitted for legibility.

Figure 2.8 Example of Instruction Operation

Operation:

```
temp ← (GPR[rs]31||GPR[rs]31..0) + (GPR[rt]31||GPR[rt]31..0)
if temp32 ≠ temp31 then
    SignalException(IntegerOverflow)
else
    GPR[rd] ← temp
endif
```

See 2.2 “Operation Section Notation and Functions” on page 18 for more information on the formal notation used here.

2.1.8 Exceptions Field

The *Exceptions* field lists the exceptions that can be caused by *Operation* of the instruction. It omits exceptions that can be caused by the instruction fetch, for instance, TLB Refill, and also omits exceptions that can be caused by asynchronous external events such as an Interrupt. Although a Bus Error exception may be caused by the operation of a load or store instruction, this section does not list Bus Error for load and store instructions because the relationship between load and store instructions and external error indications, like Bus Error, are dependent upon the implementation.

Figure 2.9 Example of Instruction Exception

Exceptions:

Integer Overflow

An instruction may cause implementation-dependent exceptions that are not present in the *Exceptions* section.

2.1.9 Programming Notes and Implementation Notes Fields

The *Notes* sections contain material that is useful for programmers and implementors, respectively, but that is not necessary to describe the instruction and does not belong in the description sections.

Figure 2.10 Example of Instruction Programming Notes

Programming Notes:
ADDU performs the same arithmetic operation but does not trap on overflow.

2.2 Operation Section Notation and Functions

In an instruction description, the *Operation* section uses a high-level language notation to describe the operation performed by each instruction. Special symbols used in the pseudocode are described in the previous chapter. Specific pseudocode functions are described below.

This section presents information about the following topics:

- "Instruction Execution Ordering" on page 18
- "Pseudocode Functions" on page 18

2.2.1 Instruction Execution Ordering

Each of the high-level language statements in the *Operations* section are executed sequentially (except as constrained by conditional and loop constructs).

2.2.2 Pseudocode Functions

There are several functions used in the pseudocode descriptions. These are used either to make the pseudocode more readable, to abstract implementation-specific behavior, or both. These functions are defined in this section, and include the following:

- "Coprocessor General Register Access Functions" on page 18
- "Memory Operation Functions" on page 20
- "Floating Point Functions" on page 23
- "Miscellaneous Functions" on page 26

2.2.2.1 Coprocessor General Register Access Functions

Defined coprocessors, except for CP0, have instructions to exchange words and doublewords between coprocessor general registers and the rest of the system. What a coprocessor does with a word or doubleword supplied to it and how a coprocessor supplies a word or doubleword is defined by the coprocessor itself. This behavior is abstracted into the functions described in this section.

COP_LW

The COP_LW function defines the action taken by coprocessor z when supplied with a word from memory during a load word operation. The action is coprocessor-specific. The typical action would be to store the contents of memword in coprocessor general register *rt*.

Figure 2.11 COP_LW Pseudocode Function

```
COP_LW (z, rt, memword)
    z: The coprocessor unit number
    rt: Coprocessor general register specifier
    memword: A 32-bit word value supplied to the coprocessor

    /* Coprocessor-dependent action */

endfunction COP_LW
```

COP_LD

The COP_LD function defines the action taken by coprocessor z when supplied with a doubleword from memory during a load doubleword operation. The action is coprocessor-specific. The typical action would be to store the contents of memdouble in coprocessor general register *rt*.

Figure 2.12 COP_LD Pseudocode Function

```
COP_LD (z, rt, memdouble)
    z: The coprocessor unit number
    rt: Coprocessor general register specifier
    memdouble:  64-bit doubleword value supplied to the coprocessor.

    /* Coprocessor-dependent action */

endfunction COP_LD
```

COP_SW

The COP_SW function defines the action taken by coprocessor *z* to supply a word of data during a store word operation. The action is coprocessor-specific. The typical action would be to supply the contents of the low-order word in coprocessor general register *rt*.

Figure 2.13 COP_SW Pseudocode Function

```
dataword ← COP_SW (z, rt)
    z: The coprocessor unit number
    rt: Coprocessor general register specifier
    dataword: 32-bit word value

    /* Coprocessor-dependent action */

endfunction COP_SW
```

COP_SD

The COP_SD function defines the action taken by coprocessor *z* to supply a doubleword of data during a store doubleword operation. The action is coprocessor-specific. The typical action would be to supply the contents of the low-order doubleword in coprocessor general register *rt*.

Figure 2.14 COP_SD Pseudocode Function

```
datadouble ← COP_SD (z, rt)
    z:  The coprocessor unit number
    rt: Coprocessor general register specifier
    datadouble: 64-bit doubleword value

    /* Coprocessor-dependent action */

endfunction COP_SD
```

CoprocessorOperation

The CoprocessorOperation function performs the specified Coprocessor operation.

Figure 2.15 CoprocessorOperation Pseudocode Function

```
CoprocessorOperation (z, cop_fun)

    /* z:         Coprocessor unit number */
    /* cop_fun:   Coprocessor function from function field of instruction */

    /* Transmit the cop_fun value to coprocessor z */

endfunction CoprocessorOperation
```

2.2.2.2 Memory Operation Functions

Regardless of byte ordering (big- or little-endian), the address of a halfword, word, or doubleword is the smallest byte address of the bytes that form the object. For big-endian ordering this is the most-significant byte; for a little-endian ordering this is the least-significant byte.

In the *Operation* pseudocode for load and store operations, the following functions summarize the handling of virtual addresses and the access of physical memory. The size of the data item to be loaded or stored is passed in the *AccessLength* field. The valid constant names and values are shown in Table 2.1. The bytes within the addressed unit of memory (word for 32-bit processors or doubleword for 64-bit processors) that are used can be determined directly from the *AccessLength* and the two or three low-order bits of the address.

AddressTranslation

The AddressTranslation function translates a virtual address to a physical address and its cacheability and coherency attribute, describing the mechanism used to resolve the memory reference.

Given the virtual address *vAddr*, and whether the reference is to Instructions or Data (*IorD*), find the corresponding physical address (*pAddr*) and the cacheability and coherency attribute (*CCA*) used to resolve the reference. If the virtual address is in one of the unmapped address spaces, the physical address and *CCA* are determined directly by the virtual address. If the virtual address is in one of the mapped address spaces then the TLB or fixed mapping MMU determines the physical address and access type; if the required translation is not present in the TLB or the desired access is not permitted, the function fails and an exception is taken.

Figure 2.16 AddressTranslation Pseudocode Function

```
(pAddr, CCA) ← AddressTranslation (vAddr, IorD, LorS)

    /* pAddr: physical address */
    /* CCA:   Cacheability&Coherency Attribute,the method used to access caches*/
```

```
    /*        and memory and resolve the reference */

    /* vAddr: virtual address */
    /* IorD:  Indicates whether access is for INSTRUCTION or DATA */
    /* LorS:  Indicates whether access is for LOAD or STORE */

    /* See the address translation description for the appropriate MMU */
    /* type in Volume III of this book for the exact translation mechanism */

endfunction AddressTranslation
```

LoadMemory

The LoadMemory function loads a value from memory.

This action uses cache and main memory as specified in both the Cacheability and Coherency Attribute (*CCA*) and the access (*IorD*) to find the contents of *AccessLength* memory bytes, starting at physical location *pAddr*. The data is returned in a fixed-width naturally aligned memory element (*MemElem*). The low-order 2 (or 3) bits of the address and the *AccessLength* indicate which of the bytes within *MemElem* need to be passed to the processor. If the memory access type of the reference is *uncached*, only the referenced bytes are read from memory and marked as valid within the memory element. If the access type is *cached* but the data is not present in cache, an implementation-specific *size* and *alignment* block of memory is read and loaded into the cache to satisfy a load reference. At a minimum, this block is the entire memory element.

Figure 2.17 LoadMemory Pseudocode Function

```
MemElem ← LoadMemory (CCA, AccessLength, pAddr, vAddr, IorD)

    /* MemElem:  Data is returned in a fixed width with a natural alignment. The */
    /*           width is the same size as the CPU general-purpose register, */
    /*           32 or 64 bits, aligned on a 32- or 64-bit boundary, */
    /*           respectively. */
    /* CCA:      Cacheability&CoherencyAttribute=method used to access caches */
    /*           and memory and resolve the reference */

    /* AccessLength: Length, in bytes, of access */
    /* pAddr:    physical address */
    /* vAddr:    virtual address */
    /* IorD:     Indicates whether access is for Instructions or Data */

endfunction LoadMemory
```

StoreMemory

The StoreMemory function stores a value to memory.

The specified data is stored into the physical location *pAddr* using the memory hierarchy (data caches and main memory) as specified by the Cacheability and Coherency Attribute (*CCA*). The *MemElem* contains the data for an aligned, fixed-width memory element (a word for 32-bit processors, a doubleword for 64-bit processors), though only the bytes that are actually stored to memory need be valid. The low-order two (or three) bits of *pAddr* and the *AccessLength* field indicate which of the bytes within the *MemElem* data should be stored; only these bytes in memory will actually be changed.

Figure 2.18 StoreMemory Pseudocode Function

```
StoreMemory (CCA, AccessLength, MemElem, pAddr, vAddr)
```

```
    /* CCA:        Cacheability&Coherency Attribute, the method used to access */
    /*             caches and memory and resolve the reference. */
    /* AccessLength: Length, in bytes, of access */
    /* MemElem:    Data in the width and alignment of a memory element. */
    /*             The width is the same size as the CPU general */
    /*             purpose register, either 4 or 8 bytes, */
    /*             aligned on a 4- or 8-byte boundary. For a */
    /*             partial-memory-element store, only the bytes that will be*/
    /*             stored must be valid.*/
    /* pAddr:      physical address */
    /* vAddr:      virtual address */

endfunction StoreMemory
```

Prefetch

The Prefetch function prefetches data from memory.

Prefetch is an advisory instruction for which an implementation-specific action is taken. The action taken may increase performance but must not change the meaning of the program or alter architecturally visible state.

Figure 2.19 Prefetch Pseudocode Function

```
Prefetch (CCA, pAddr, vAddr, DATA, hint)

    /* CCA:   Cacheability&Coherency Attribute, the method used to access */
    /*        caches and memory and resolve the reference. */
    /* pAddr: physical address */
    /* vAddr: virtual address */
    /* DATA:  Indicates that access is for DATA */
    /* hint:  hint that indicates the possible use of the data */

endfunction Prefetch
```

Table 2.1 lists the data access lengths and their labels for loads and stores.

Table 2.1 AccessLength Specifications for Loads/Stores

AccessLength Name	Value	Meaning
DOUBLEWORD	7	8 bytes (64 bits)
SEPTIBYTE	6	7 bytes (56 bits)
SEXTIBYTE	5	6 bytes (48 bits)
QUINTIBYTE	4	5 bytes (40 bits)
WORD	3	4 bytes (32 bits)
TRIPLEBYTE	2	3 bytes (24 bits)
HALFWORD	1	2 bytes (16 bits)
BYTE	0	1 byte (8 bits)

SyncOperation

The SyncOperation function orders loads and stores to synchronize shared memory.

This action makes the effects of the synchronizable loads and stores indicated by *stype* occur in the same order for all processors.

Figure 2.20 SyncOperation Pseudocode Function

```
SyncOperation(stype)

    /* stype: Type of load/store ordering to perform. */

    /* Perform implementation-dependent operation to complete the */
    /* required synchronization operation */

endfunction SyncOperation
```

2.2.2.3 Floating Point Functions

The pseudocode shown in below specifies how the unformatted contents loaded or moved to CP1 registers are interpreted to form a formatted value. If an FPR contains a value in some format, rather than unformatted contents from a load (uninterpreted), it is valid to interpret the value in that format (but not to interpret it in a different format).

ValueFPR

The ValueFPR function returns a formatted value from the floating point registers.

Figure 2.21 ValueFPR Pseudocode Function

```
value ← ValueFPR(fpr, fmt)

    /* value: The formattted value from the FPR */

    /* fpr:    The FPR number */
    /* fmt:    The format of the data, one of: */
    /*         S, D, W, L, PS, */
    /*         OB, QH, */
    /*         UNINTERPRETED_WORD, */
    /*         UNINTERPRETED_DOUBLEWORD */
    /* The UNINTERPRETED values are used to indicate that the datatype */
    /* is not known as, for example, in SWC1 and SDC1 */

    case fmt of
        S, W, UNINTERPRETED_WORD:
            valueFPR ← FPR[fpr]

        D, UNINTERPRETED_DOUBLEWORD:
            if (FP32RegistersMode = 0)
                if (fpr0 ≠ 0) then
                    valueFPR ← UNPREDICTABLE
                else
                    valueFPR ← FPR[fpr+1]31..0 || FPR[fpr]31..0
                endif
            else
                valueFPR ← FPR[fpr]
            endif

        L, PS:
            if (FP32RegistersMode = 0) then
                valueFPR ← UNPREDICTABLE
```

```
        else
            valueFPR ← FPR[fpr]
        endif

    DEFAULT:
        valueFPR ← UNPREDICTABLE

    endcase
endfunction ValueFPR
```

The pseudocode shown below specifies the way a binary encoding representing a formatted value is stored into CP1 registers by a computational or move operation. This binary representation is visible to store or move-from instructions. Once an FPR receives a value from the StoreFPR(), it is not valid to interpret the value with ValueFPR() in a different format.

StoreFPR

Figure 2.22 StoreFPR Pseudocode Function

```
StoreFPR (fpr, fmt, value)

    /* fpr:   The FPR number */
    /* fmt:   The format of the data, one of: */
    /*        S, D, W, L, PS, */
    /*        OB, QH, */
    /*        UNINTERPRETED_WORD, */
    /*        UNINTERPRETED_DOUBLEWORD */
    /* value: The formattted value to be stored into the FPR */

    /* The UNINTERPRETED values are used to indicate that the datatype */
    /* is not known as, for example, in LWC1 and LDC1 */


    case fmt of
        S, W, UNINTERPRETED_WORD:
            FPR[fpr] ← value

        D, UNINTERPRETED_DOUBLEWORD:
            if (FP32RegistersMode = 0)
                if (fpr0 ≠ 0) then
                    UNPREDICTABLE
                else
                    FPR[fpr]   ← UNPREDICTABLE^32 || value31..0
                    FPR[fpr+1] ← UNPREDICTABLE^32 || value63..32
                endif
            else
                FPR[fpr] ← value
            endif

        L, PS:
            if (FP32RegistersMode = 0) then
                UNPREDICTABLE
            else
                FPR[fpr] ← value
            endif

    endcase
```

```
endfunction StoreFPR
```

The pseudocode shown below checks for an enabled floating point exception and conditionally signals the exception.

CheckFPException

Figure 2.23 CheckFPException Pseudocode Function

```
CheckFPException()

/* A floating point exception is signaled if the E bit of the Cause field is a 1 */
/* (Unimplemented Operations have no enable) or if any bit in the Cause field */
/* and the corresponding bit in the Enable field are both 1 */

    if (   (FCSR17 = 1) or
           ((FCSR16..12 and FCSR11..7) ≠ 0))   ) then
        SignalException(FloatingPointException)
    endif

endfunction CheckFPException
```

FPConditionCode

The FPConditionCode function returns the value of a specific floating point condition code.

Figure 2.24 FPConditionCode Pseudocode Function

```
tf ←FPConditionCode(cc)

    /* tf: The value of the specified condition code */

    /* cc: The Condition code number in the range 0..7 */

    if cc = 0 then
        FPConditionCode ← FCSR23
    else
        FPConditionCode ← FCSR24+cc
    endif

endfunction FPConditionCode
```

SetFPConditionCode

The SetFPConditionCode function writes a new value to a specific floating point condition code.

Figure 2.25 SetFPConditionCode Pseudocode Function

```
SetFPConditionCode(cc, tf)
    if cc = 0 then
        FCSR ← FCSR31..24 || tf || FCSR22..0
    else
        FCSR ← FCSR31..25+cc || tf || FCSR23+cc..0
endif

endfunction SetFPConditionCode
```

2.2.2.4 Miscellaneous Functions

This section lists miscellaneous functions not covered in previous sections.

SignalException

The SignalException function signals an exception condition.

This action results in an exception that aborts the instruction. The instruction operation pseudocode never sees a return from this function call.

Figure 2.26 SignalException Pseudocode Function

```
SignalException(Exception, argument)

    /* Exception:    The exception condition that exists. */
    /* argument:     A exception-dependent argument, if any */

endfunction SignalException
```

SignalDebugBreakpointException

The SignalDebugBreakpointException function signals a condition that causes entry into Debug Mode from non-Debug Mode.

This action results in an exception that aborts the instruction. The instruction operation pseudocode never sees a return from this function call.

Figure 2.27 SignalDebugBreakpointException Pseudocode Function

```
SignalDebugBreakpointException()

endfunction SignalDebugBreakpointException
```

SignalDebugModeBreakpointException

The SignalDebugModeBreakpointException function signals a condition that causes entry into Debug Mode from Debug Mode (i.e., an exception generated while already running in Debug Mode).

This action results in an exception that aborts the instruction. The instruction operation pseudocode never sees a return from this function call.

Figure 2.28 SignalDebugModeBreakpointException Pseudocode Function

```
SignalDebugModeBreakpointException()

endfunction SignalDebugModeBreakpointException
```

NullifyCurrentInstruction

The NullifyCurrentInstruction function nullifies the current instruction.

The instruction is aborted, inhibiting not only the functional effect of the instruction, but also inhibiting all exceptions detected during fetch, decode, or execution of the instruction in question. For branch-likely instructions, nullification kills the instruction in the delay slot of the branch likely instruction.

Figure 2.29 NullifyCurrentInstruction PseudoCode Function

```
NullifyCurrentInstruction()

endfunction NullifyCurrentInstruction
```

JumpDelaySlot

The JumpDelaySlot function is used in the pseudocode for the PC-relative instructions in the MIPS16e ASE. The function returns TRUE if the instruction at *vAddr* is executed in a jump delay slot. A jump delay slot always immediately follows a JR, JAL, JALR, or JALX instruction.

Figure 2.30 JumpDelaySlot Pseudocode Function

```
JumpDelaySlot(vAddr)

    /* vAddr:Virtual address */

endfunction JumpDelaySlot
```

PolyMult

The PolyMult function multiplies two binary polynomial coefficients.

Figure 2.31 PolyMult Pseudocode Function

```
PolyMult(x, y)
    temp ← 0
    for i in 0 .. 31
        if x_i = 1 then
            temp ← temp xor (y_(31-i)..0 || 0^i)
        endif
    endfor

    PolyMult ← temp

endfunction PolyMult
```

2.3 Op and Function Subfield Notation

In some instructions, the instruction subfields *op* and *function* can have constant 5- or 6-bit values. When reference is made to these instructions, uppercase mnemonics are used. For instance, in the floating point ADD instruction, *op*=COP1 and *function*=ADD. In other cases, a single field has both fixed and variable subfields, so the name contains both upper- and lowercase characters.

2.4 FPU Instructions

In the detailed description of each FPU instruction, all variable subfields in an instruction format (such as *fs, ft, immediate*, and so on) are shown in lowercase. The instruction name (such as ADD, SUB, and so on) is shown in uppercase.

For the sake of clarity, an alias is sometimes used for a variable subfield in the formats of specific instructions. For example, *rs=base* in the format for load and store instructions. Such an alias is always lowercase since it refers to a variable subfield.

Bit encodings for mnemonics are given in Volume I, in the chapters describing the CPU, FPU, MDMX, and MIPS16e instructions.

See “Op and Function Subfield Notation” on page 27 for a description of the *op* and *function* subfields.

Chapter 3

The MCU Application-Specific Extension to the MIPS32® and microMIPS32^{TM}Architecture

3.1 Base Architecture Requirements

The MCU® ASE requires at least one of the following base architecture supports:

- **The microMIPS Architecture**: The MCU ASE requires a compliant implementation of the microMIPS Architecture.
- **The MIPS32 Architecture**: The MCU ASE requires a compliant implementation of the MIPS32Architecture.

3.2 Software Detection of the ASE

Software may determine if the MCU ASE is implemented by checking the state of the MCU bit in the *Config3* CP0 register.

3.3 Compliance and Subsetting

There are no instruction subsets of the MCU ASE to the microMIPS/MIPS32 Architecture—all MCU instructions must be implemented.

3.4 Overview of the MCU ASE

The MCU ASE extends the microMIPS32/MIPS32 Architecture with a set of new features designed for the microcontroller market. The MCU ASE contains enhancements in several distinct areas: interrupt delivery, interrupt latency, and I/O peripheral programming.

3.4.1 Interrupt Delivery

The MCU ASE extends the number of hardware interrupt sources from 6 to 8. For legacy and vectored-interrupt mode, this represents 8 external interrupt sources. For EIC mode, the widened IPL and RIPL fields can now represent 256 external interrupt sources.

3.4.2 Interrupt Latency Reduction

The MCU ASE includes a package of extensions to microMIPS/MIPS32 that decrease the latency of the processor's response to a signalled interrupt.

3.4.2.1 Interrupt Vector Prefetching

Normally on MIPS architecture processors, when an interrupt or exception is signalled, execution pipelines must be flushed before the interrupt/exception handler is fetched. This is necessary to avoid mixing the contexts of the interrupted/faulting program and the exception handler. The MCU ASE introduces a hardware mechanism in which the interrupt exception vector is prefetched whenever the interrupt input signals change. The prefetch memory transaction occurs in parallel with the pipeline flush and exception prioritization. This decreases the overall latency of the execution of the interrupt handler's first instruction.

3.4.2.2 Automated Interrupt Prologue

The use of Shadow Register Sets avoids the software steps of having to save general-purpose registers before handling an interrupt.

The MCU ASE adds additional hardware logic that automatically saves some of the COP0 state in the stack and automatically updates some of the COP0 registers in preparation for interrupt handling.

3.4.2.3 Automated Interrupt Epilogue

A mirror to the Automated Prologue, this feature automates the restoration of some of the COP0 registers from the stack and the preparation of some of the COP0 registers for returning to non-exception mode. This feature is implemented within the IRET instruction, which is introduced in this ASE.

3.4.2.4 Interrupt Chaining

An optional feature of the Automated Interrupt Epilogue, this feature allows handling a second interrupt after a primary interrupt is handled, without returning to non-exception mode (and the related pipeline flushes that would normally be necessary).

3.4.3 I/O Device Programming

The ASE includes some instructions that simplify writing the control registers of I/O devices. Specifically, new instructions are made available to avoid read-modify-write hazards, without resorting to busy-wait loops or system calls. Read-modify-write hazards exist when one thread reads a control register, and that thread is interrupted before it modifies the control register.

Chapter 4

The MCU Instruction Set

The MCU ASE includes three new instructions that are particularly useful in microcontroller applications.

4.1 IRET

This instruction can be used as a replacement for the ERET instruction when returning from an interrupt. This instruction implements the Automated Interrupt Epilogue feature, which automates restoring some of the COP0 registers from the stack and updating the C0_Status register in preparation for returning to non-exception mode. This instruction also implements the optional Interrupt Chaining feature, which allows a subsequent interrupt to be handled without returning to non-exception mode.

4.2 ASET

This instruction allows a bit within an uncached I/O control register to be atomically set; that is, the read-modify byte write sequence performed by this instruction cannot be interrupted.

4.3 ACLR

This instruction allows a bit within an uncached I/O control register to be atomically cleared; that is, the read-modify byte write sequence performed by this instruction cannot be interrupted.

31–26	25	24–6	5–0
COP0 010000	C0 1	0 00 0000 0000 0000 0000	IRET 111000
6	1	20	6

Format: `IRET` **MIPS and MCU ASE**

Purpose: Interrupt Return with automated interrupt epilogue handling

Optionally jump directly to another interrupt vector without returning to original return address.

Description:

IRET automates some of the operations that are required when returning from an interrupt handler and can be used in place of the ERET instruction at the end of interrupt handlers. IRET is only appropriate when using Shadow Register Sets and the EIC Interrupt mode. The automated operations of this instruction can be used to reverse the effects of the automated operations of the Auto-Prologue feature.

If the EIC interrupt mode and the Interrupt Chaining feature are used, the IRET instruction can be used to shorten the time between returning from the current interrupt handler and handling the next requested interrupt.

If the Automated Prologue feature is disabled, then IRET behaves exactly like ERET.

If either the $Status_{ERL}$ or $Status_{BEV}$ bits are set, then IRET behaves exactly like ERET.

If Interrupt Chaining is disabled:

> Interrupts are disabled. COP0 *Status*, *SRSCtl*, and *EPC* registers are restored from the stack. GPR 29 is incremented for the stack frame size. IRET then clears execution and instruction hazards, conditionally restores $SRSCtl_{CSS}$ from $SRSCtl_{PSS}$, and returns at the completion of interrupt processing to the interrupted instruction pointed to by the *EPC* register.

If Interrupt Chaining is enabled:

> Interrupts are disabled. COP0 *Status* register is restored from the stack. The priority output of the External Interrupt Controller is compared with the IPL field of the *Status* register.
>
> If $Status_{IPL}$ has a higher priority than or equal to the External Interrupt Controller value:
>
>> COP0 *SRSCtl* and *EPC* registers are restored from the stack. GPR 29 is incremented for the stack frame size. IRET then clears execution and instruction hazards, conditionally restores $SRSCtl_{CSS}$ from $SRSCtl_{PSS}$, and returns to the interrupted instruction pointed to by the *EPC* register at the completion of interrupt processing.
>
> If $Status_{IPL}$ has a lower priority than the External Interrupt Controller value:
>
>> The value of GPR 29 is first saved to a temporary register then GPR 29 is incremented for the stack frame size. The EIC is signalled that the next pending interrupt has been accepted. This signalling will update the $Cause_{RIPL}$ and $SRSCtl_{EICSS}$ fields from the EIC output values. The $SRSCtl_{EICSS}$ field is copied to the $SRSCtl_{CSS}$ field, while the $Cause_{RIPL}$ field is copied to the $Status_{IPL}$ field. The saved temporary register is copied to the GPR 29 of the current SRS. The KSU and EXL fields of the *Status* register are optionally set to zero. No barrier for execution hazards or instruction hazards is created. IRET finishes by jumping to the interrupt vector driven by the EIC.

IRET does not execute the next instruction (i.e., it has no delay slot).

Restrictions:

The operation of the processor is **UNDEFINED** if IRET is executed in the delay slot of a branch or jump instruction.

The operation of the processor is **UNDEFINED** if IRET is executed when either Shadow Register Sets are not enabled, or when the EIC interrupt mode is not enabled.

An IRET placed between an LL and SC instruction will always cause the SC to fail.

The effective addresses used for stack transactions must be naturally-aligned. If either of the two least-significant bits of the address is non-zero, an Address Error exception occurs.

IRET implements a software barrier that resolves all execution and instruction hazards created by Coprocessor 0 state changes (for Release 2 implementations, refer to the SYNCI instruction for additional information on resolving instruction hazards created by writing the instruction stream). The effects of this barrier begin with the instruction fetch and decode of the instruction at the PC to which the IRET returns.

In a Release 2 implementation, IRET does not restore $SRSCtl_{CSS}$ from $SRSCtl_{PSS}$ if $Status_{BEV}$ = 1 or $Status_{ERL}$ = 1, because any exception that sets $Status_{ERL}$ to 1 (Reset, Soft Reset, NMI, or cache error) does not save $SRSCtl_{CSS}$ in $SRSCtl_{PSS}$. If software sets $Status_{ERL}$ to 1, it must be aware of the operation of an IRET that may be subsequently executed.

The stack memory transactions behave as individual LW operations with respect to exception reporting. BadVAddr would report the faulting address for an unaligned access, and the faulting word address for unprivileged access, TLB Refill, and TLB Invalid exceptions. For TLB exceptions, the faulting word address would be reflected in the *Context* and *EntryHi* registers. The *CacheError* register would reflect the faulting word address for Cache Errors.

Operation:

```
if (( IntCtl_APE == 0) | (Status_ERL == 1) | (Status_BEV== 1))
    Act as ERET // read Operation section of ERET description
else
    temp ← 0x4 + GPR[29]
    tempStatus ← LoadStackWord(temp)
    ClearHazards()
    if ( (IntCtl_ICE == 0) |  ((IntCtl_ICE == 1) &
    (tempStatus_IPL ≥ EIC_RIPL)) )
        temp ← 0x8 + GPR[29]
        tempSRSCtl ← LoadStackWord(temp)
        temp ← 0x0 + GPR[29]
        tempEPC ← LoadStackWord(temp)
    endif
    Status ← tempStatus
    if ( (IntCtl_ICE == 0) | ((IntCtl_ICE == 1) &
        (tempStatus_IPL ≥ EIC_RIPL)) )
        GPR[29] ← GPR[29] + DecodedValue(IntCtl_StkDec)
        SRSCtl_PSS ← tempSRSCtl_PSS
        SRSCtl_ESS ← tempSRSCtl_ESS
        EPC ← tempEPC
        temp ← EPC
        Status_EXL ← 0
        if (ArchitectureRevision ≥ 2) and (SRSCtl_HSS > 0) and (Status_BEV = 0) then
            SRSCtl_CSS ← SRSCtl_PSS
        endif
        if IsMicroMIPSImplemented() then
            PC ← temp_31..1 || 0
            ISAMode ← temp_0
        else
            PC ← temp
```

```
            endif
            LLbit ← 0
            Cause_IC ← 0
            ClearHazards()
        else
            Cause_RIPL ← EIC_RIPL
            SRSCtl_EICSS ← EIC_SS
            temp29 ← GPR[29]
            GPR[29] ← GPR[29] + DecodedValue(IntCtl_StkDec)
            Status_IPL ← Cause_RIPL
            SRSCtl_CSS ← SRSCtl_EICSS
            NewShadowSet ← SRSCtl_EICSS
            GPR[29] ← temp29
            if (IntCtl_ClrEXL == 1)
                Status_EXL ← 0
                Status_KSU ← 0
            endif
            LLbit ← 0
            Cause_IC ← 1
            ClearHazards()
            PC ← CalcIntrptAddress()
        endif
    endif


    function LoadStackWord(vaddr)
        if vAddr_1..0 ≠ 0^2 then
            SignalException(AddressError)
        endif
        (pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
        memword ← LoadMemory (CCA, WORD, pAddr, vAddr, DATA)
        LoadStackWord ← memword
    endfunction LoadStackWord


    function CalcIntrptAddress()
        if StatusBEV == 1
            vectorBase ←0xBFC0.0200
        else
            if ( ArchitectureRevision ≥ 2)
                vectorBase ←EBase_31..12 ‖ 0^11)
            else
                vectorBase ←0x8000.0000
            endif
        endif
        if (Cause_IV == 0)
            vectorOffset ← 0x180
        else
            if (Status_BEV = 1) or (IntCtl_VS = 0)
                vectorOffset ← 0x200
            else
                if ( Config3_VEIC == 1 and EIC_Option == 1)
                    VectorNum ← Cause_RIPL
                elseif (Config3_VEIC == 1 and EIC_Option == 2)
                    VectorNum ← EIC_VectorNum
                elseif (Config3_VEIC == 0 )
                    VectorNum ← VIntPriorityEncoder()
```

```
            endif
            if (Config3_VEIC == 1 and EIC_Option == 3)
               vectorOffset ← EIC_VectorOffset
            else
               vectorOffset ← 0x200 + (VectorNum x (IntCtl_VS || 0^5))
            endif
         endif
      endif
      CalcIntrptAddress ← vectorBase | vectorOffset
      if (Config3_ISAOnExec)
         CalcIntrptAddress ← CalcIntrptAddress_31..1 || 1
      endif
   endfunction CalcIntrptAddress
```

Exceptions:

Coprocessor Unusable Exception, TLB Refill, TLB Invalid, Address Error, Watch, Cache Error, Bus Error Exceptions

31 … 26	25 … 21	20 … 16	15	14 … 12	11 … 0
REGIMM 000001	base	ATOMIC 00111	1	Bit	offset
6	5	5	1	3	12

Format: `ASET bit, offset(base)` **MIPS AND MCU ASE**

Purpose: Atomically Set Bit within Byte

Description: `Disable interrupts;temp ← memory[GPR[base] + offset]; temp ← (temp or (1 << bit)) ; memory[GPR[base] + offset] ← temp; Enable Interrupts`

The contents of the byte at the memory location specified by the effective address are fetched. The specified bit within the byte is set to one. The modified byte is stored in memory at the location specified by the effective address. The 12-bit signed *offset* is added to the contents of GPR *base* to form the effective address. The read-modify-write sequence cannot be interrupted.

Transactions with locking semantics occur in some memory interconnects/busses. It is implementation-specific whether this instruction uses such locking transactions.

Restrictions:

The operation of the processor is **UNPREDICTABLE** if an ASET instruction is executed in the delay slot of a branch or jump instruction.

Operation:

```
vAddr ← sign_extend(offset) + GPR[base]
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, STORE)
pAddr ← pAddr_{PSIZE-1..2} || (pAddr_{1..0} xor ReverseEndian^2)
TempIE ← Status_{IE}
Status_{IE} ← 0
memword ← LoadMemory (CCA, BYTE, pAddr, vAddr, DATA)
byte ← vAddr_{1..0} xor BigEndianCPU^2
temp ← memword_{7+8*byte..8*byte}
temp ← temp or ( 1 || 0^{bit})
dataword ← temp || 0^{8*byte}
StoreMemory (CCA, BYTE, dataword, pAddr, vAddr, DATA)
Status_{IE} ← TempIE
```

Exceptions:

TLB Refill, TLB Invalid, TLB Modified, Address Error, Watch

Programming Notes:

Upon a TLB miss, a TLBS exception is signalled in the ExcCode field of the *Cause* register. For address error, a ADES exception is signalled in the ExcCode field of the *Cause* register. For other data-stream related exceptions such as Debug Data Break exceptions and Watch exceptions, it is implementation-specific whether this instruction is treated as a load or as a store.

31–26	25–21	20–16	15	14–12	11–0
REGIMM 000001	base	ATOMIC 00111	0	Bit	offset
6	5	5	1	3	12

Format: `ACLR bit, offset(base)` **MIPS and MCU ASE**

Purpose: Atomically Clear Bit within Byte

Description: `Disable interrupts; temp ← memory[GPR[base] + offset]; temp ← (temp and ~(1 << bit)) ; memory[GPR[base] + offset] ← temp; Enable Interrupts`

The contents of the byte at the memory location specified by the effective address are fetched. The specified bit within the byte is cleared to zero. The modified byte is stored in memory at the location specified by the effective address. The 12-bit signed *offset* is added to the contents of GPR *base* to form the effective address. The read-modify-write sequence cannot be interrupted.

Transactions with locking semantics occur in some memory interconnects/busses. It is implementation-specific whether this instruction uses such locking transactions.

Restrictions:

The operation of the processor is **UNPREDICTABLE** if an ACLR instruction is executed in the delay slot of a branch or jump instruction.

Operation:

```
vAddr ← sign_extend(offset) + GPR[base]
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, STORE)
pAddr ← pAddr_{PSIZE-1..2} || (pAddr_{1..0} xor ReverseEndian^2)
TempIE ← Status_{IE}
Status_{IE} ← 0
memword ← LoadMemory (CCA, BYTE, pAddr, vAddr, DATA)
byte ← vAddr_{1..0} xor BigEndianCPU^2
temp ← memword_{7+8*byte..8*byte}
temp ← temp and (( 1 || 0^{bit}) xor 0xFF))
dataword ← temp || 0^{8*byte}
StoreMemory (CCA, BYTE, dataword, pAddr, vAddr, DATA)
Status_{IE} ← TempIE
```

Exceptions:

TLB Refill, TLB Invalid, TLB Modified, Address Error, Watch

Programming Notes:

Upon a TLB miss, a TLBS exception is signalled in the ExcCode field of the *Cause* register. For address error, a ADES exception is signalled in the ExcCode field of the *Cause* register. For other data-stream related exceptions such as Debug Data Break exceptions and Watch exceptions, it is implementation-specific whether this instruction is treated as a load or as a store.

Chapter 5

The MCU Privileged Resource Architecture

5.1 Introduction

The MIPS32 Privileged Resource Architecture (PRA) defines a set of environments and capabilities on which the Instruction Set Architecture operates. This includes definitions of the programming interface and operation of the system coprocessor, CP0. MCU defines extensions to the MIPS32 PRA that are desirable in a microcontroller environment. This document describes these extensions. It is not intended to be a stand-alone PRA specification and must be read in the context of the MIPS32 Architecture specification.

5.2 The MCU System Coprocessor

The MCU system coprocessor interface and functionality is identical to MIPS32. except as defined below.

5.3 Interrupt Delivery

5.3.1 Number of Hardware Interrupts

The MCU ASE increases the number of Hardware Interrupts to 8. To accommodate this, the privileged architecture has the following changes:

- Bits 18 and 16 of the *Status* Register are used to extend the IM/IPL fields.
- Bits 17 and 16 of the *Cause* Register are used to extend the IP/RIPL fields. $Cause_{17}$ corresponds to $Status_{18}$, and $Cause_{16}$ corresponds to $Status_{16}$.
- An additional COP0 register (*SRSMAP2*), located at CP0 Register 12, Select 5, is used to map the Shadow Register Set for the two new Vector Numbers available in Vectored Interrupt Mode.

5.3.1.1 Changes to Vectored Interrupt Mode

The highest priority interrupt source is now represented by $Cause_{17}$ and $Status_{18}$. The Shadow Register Set for this interrupt source is specified by the SSV9 field in *SRSMAP2* (bits 7:4).

The second highest priority interrupt source is now represented by $Cause_{16}$ and $Status_{16}$. The Shadow Register Set for this interrupt source is specified by the SSV8 field in *SRSMAP2* (bits 3:0).

5.3.1.2 Changes to External Interrupt Controller Mode

The $Status_{IPL}$ and $Cause_{RIPL}$ fields are now 8 bits in width, which allows these fields to represent 256 external interrupt sources.

5.4 Interrupt Handling

5.4.1 Interrupt Vector Prefetching

5.4.1.1 Historical Behavior of Pipelines with In-Order Completion

Even on a processor that completes instructions in program order, traditionally there is some latency from when the interrupt is recognized by the pipeline and when the first instruction of the interrupt handler is executed. Because interrupts must be reported on a valid instruction, the interrupt is normally recognized by the pipeline in one of the later pipeline stages. Subsequent instructions in the pipeline would be annulled for the context switch to exception mode. The instruction fetch for the interrupt handler could be started after the interrupt is recognized by the pipeline as the highest priority exception, but the annulled instructions would still have to drain from the pipeline.

Typical Interrupt Handling Flow in Pipelined Implementation with In-Order Completion

Time	Interrupt Pins	Pipeline Control Logic	Instruction Fetch Logic	Exception Logic
Earlier		Executing Thread A	Fetching along Thread A	
	Interrupt Pin Asserted			
				Interrupt recognized, exception signalled to pipeline
		Stop issuing new instructions, annul subsequent instructions	Previous fetch discarded	
				Interrupt recognized as highest priority exception
			Fetch interrupt vector	
		Annulled instructions drained from pipeline		
		Pipeline restart		
Later		Execute interrupt handler		

5.4.1.2 Historical Behavior of Pipelines with Out-of-Order Completion

Historically many MIPS architecture implementations would flush the pipeline before processing any exception, especially in implementations with non-blocking caches. This was done to avoid mixing context from the interrupted process and the exception handler. This allows the exception handler to immediately save registers onto the stack without the fear of missing pending register updates from yet to be completed instructions.

Table 5.1 Typical Interrupt Handling Flow in Pipelined Implementation with Out-of-Order Completion

Time	Interrupt Pins	Pipeline Control Logic	Instruction Fetch Logic	Exception Logic
Earlier		Executing Thread A	Fetching along Thread A	
	Interrupt Pin Asserted			
				Interrupt Recognized, Exception signalled to pipeline
		Stop Issuing new instructions, Annul subsequent instructions, Wait for previous instructions to complete	Previous fetch squashed	
			Idle	
		Annulled subsequent instructions drained from pipeline	Idle	
			Idle	
		All previous instructions completed	Idle	
			Idle	Interrupt Recognized as highest priority exception.
		Pipeline restart	Fetch Interrupt Vector	
Later		Execute Interrupt Handler		

If the instructions at the exception vector were executed before all of the instructions of the interrupted process were completed, the possibility of imprecise exceptions would be introduced.

An exception is imprecise when *EPC/ErrorEPC/DEPC* does not point to the instruction that caused the exception. For example, if a load instruction misses in all of the caches for the requested data, and the cache hierarchy is non-blocking, execution may proceed pass the load. An interrupt may be recognized and accepted on an instruction subsequent to the load. While the interrupt handler is being executed, the response of the load returns and the response signals a Bus Error. In that case, a nested exception would occur, but the EPC for the bus error would not hold the address of the faulting load instruction. If the EXL bit is set at the time the Bus Error exception is recognized, the EPC would not be updated: for this case, the EPC would point to an instruction within the interrupt handler. A similar case can occur for late-arriving Floating-Point exceptions. In order to avoid these situations, some implementations flush the pipeline and wait until all outstanding instructions are completed before proceeding with the exception handler.

5.4.1.3 New Feature - Speculative Prefetching

This new feature allows for the fetching of the interrupt vector address when any interrupt is signalled to the processor core. The fetching is done before the pipeline has been flushed and even before the exception priority logic has determined if the interrupt is the highest priority exception that should be serviced. The purpose of this feature is to allow the memory transaction to occur in parallel with the pipeline flush and exception prioritization.

Table 5.2 Interrupt Handling Flow with Speculative Prefetching

Time	Interrupt Pins	Pipeline Control Logic	Instruction Fetch Logic	Exception Logic
Earlier		Executing Thread A	Fetching along Thread A	
	Interrupt Pin Asserted			
				Interrupt Recognized, Exception signalled to pipeline
		Stop Issuing new instructions, Wait for previous instructions to complete	Previous fetch squashed	
			Prefetch Interrupt Vector	
			Hold results from prefetch	
		All previous instructions completed		
				Interrupt recognized as highest priority exception
		Pipeline restart	If Interrupt not highest priority exception, squash prefetch and fetch correct exception vector	
Later		Execute Interrupt Handler		

This feature is supported for all 3 interrupt modes: Release 1 Interrupt compatibility mode, Vectored Interrupt Mode, and External Interrupt Controller/EIC mode. This feature is enabled by the *IntCtl*.PF bit.

Strictly speaking, this feature is not architecturally visible (that is, visible to software). However, to maintain the same precise exception model that has been traditionally used, the prefetched instructions must be treated as speculative. This means that any exception that might occur for the interrupt vector address prefetch—BusError, Parity Error, non-Correctable ECC—must be held until all of the instructions of the interrupted process have completed and the program counter has advanced to point to the interrupt vector address. A similar case occurs when the interrupt vector address is prefetched, but the exception priority logic subsequently decides that another higher priority exception (not an Interrupt) is to be serviced first. This other exception would use a different vector address, and the prefetch memory transaction must be dropped.

5.4.2 Interrupt Automated Prologue (IAP)

The use of Shadow Register Sets already decreases the overhead of saving usermode state before executing an interrupt service routine. The Interrupt Automated Prologue (IAP) feature automates some of the software steps which would be needed to save COP0 state before executing an interrupt service routine. Decreased latency to executing the first useful instruction of an interrupt service routine can be achieved by executing some of the steps using parallel hardware instead of serial execution of instructions.

5.4.2.1 IAP Conditions

This feature is only available when:

- Shadow Register Sets are implemented (*SRSCtl*$_{HSS}$!= 0)
- External Interrupt Controller Mode is enabled (*Config3*$_{VEIC}$=1, *IntCtl*$_{VS}$!= 0, *Cause*$_{IV}$=1, and *Status*$_{BEV}$=0)
- *IntCtl*$_{APE}$=1

This feature only takes effect when an interrupt is signalled to the processor core and the exception priority logic has resolved the interrupt to be the highest priority exception to be handled. If an exception other than an interrupt is signalled, this feature does not take effect.

5.4.2.2 IAP Operation

IAP Operation with one stack pointer.

These are the steps that are automated by this feature:

1. If (*IntCtl*$_{UseKStk}$ is zero) or (*IntCtl*$_{UseKStk}$ is one and interrupted instruction was executing in kernel mode) , then TempStackPointer is updated with the value from GPR 29 of the Previous Shadow Register Set. Else, go to Step A) (in the next section).
2. TempStackPointer is decremented by the value specified by the *IntCtl*$_{StkDec}$ register field.
3. The value in COP0 *EPC* register is stored to external memory using virtual address [TempStackPointer] + 0x0
4. The value in COP0 *Status* register is stored to external memory using virtual address [TempStackPointer]+0x4.
5. The value in COP0 *SRSCtl* register is stored to external memory using virtual address [TempStackPointer]+0x8.
6. GPR 29 of the Current Shadow Register Set is written with the value of TempStackPointer.
7. *Status*$_{IPL}$ register field is updated with the value in *Cause*$_{RIPL}$.
8. If *IntCtl*$_{ClrEXL}$ is set, then KSU, ERL and EXL fields of the *Status* register are cleared to zero.

TempStackPointer is an internal register within the processor and is not visible to software. It is used so that the modification of GPR 29 does not happen until there is no longer any possibility of memory exceptions occurring during IAP. This allows the TLB handler to be used without modification for a TLB exception that happens during IAP.

IAP Operation with multiple stack pointers.

The previous sequence is for simple software environments where there is only one stack. In more complicated environments with both user-mode and kernel-mode stacks, the *IntCtl*$_{UseKStk}$ control bit can be used to select another stack pointer for the interrupt handling. In this case, GPR 29 of the Shadow Register Set 1 is always used to hold the kernel stack pointer. GPR 29 of Shadow Register Set 1 has been pre-initialized to hold the appropriate kernel stack pointer value. The following steps illustrate how IAP works when the pre-initialized stack pointer is used (*IntCtl*$_{UseKStk}$ is one).

A) If (*IntCtl*$_{UseKStk}$ is one) and (interrupted instruction was not executing in kernel mode) then TempStackPointer = GPR 29 of Shadow Register 1 else TempStackPointer = GPR 29 of Shadow Register Set used at the time of the interrupted instruction.

B) Go to Step 2 (in previous section).

For Step A, if the interrupted instruction was already in kernel mode, then it would have been using the a stack pointer value that was previously derived from the kernel stack pointer held in GPR 29 of Shadow Register 1.

5.4.2.3 Exceptions during IAP

The memory store operations which occur during Auto-Prologue may result in Address Error, TLB refill, TLB invalid, TLB modify, Cache Error, Bus Error exceptions. If such memory exceptions occur during Auto-Prologue:

- The $Cause_{\text{ExcCode}}$ register field reports the exception type
- $Cause_{\text{AP}}$ register bit is set
- *EPC* is unchanged; points to the instruction which was originally interrupted.
- All of the other exception reporting COP0 registers (*BadVaddr, EntryHi, EntryLo*, Context, CacheError*) are updated as appropriate for the exception type. These registers reflect the effective word address which caused the exception, e.g., as if an individual SW instruction had caused the exception.
- If the memory store operation uses a mapped address and there is no matching address in the TLB, the TLB refill exception handler (offset 0x0) is used. The other TLB related exceptions (invalid, modify) use the general exception handler (offset 0x180).
- The Shadow Register Set designated by the $SRSCtl_{\text{ESS}}$ register field is used for the memory exception.
- The memory exception handler returns to the original code PC location, which is held in *C0_EPC*.
- Since the interrupt is still asserted, the interrupt is signalled again and IAP is repeated. This time, it completes as the faulting condition had previously been fixed.

The IAP feature will run to completion unless one of these memory exceptions takes place. The IAP feature is not interruptable, that is, IAP is atomic from the point of view of another pending interrupt.

5.4.3 Interrupt Automated Epilogue (IAE)

This feature is the mirror of Interrupt Automated Prologue. In preparation for returning to non-exception mode, this feature automates restoring COP0 *Status, SRSCtl* and *EPC* registers from the stack.

5.4.3.1 IAE Conditions

This feature is made available through the IRET instruction. The IRET instruction should only be used when:

- Shadow Register Sets are implemented ($SRSCtl_{\text{HSS}}$!= 0)
- External Interrupt Controller Mode is enabled ($Config3_{\text{VEIC}}$=1, $IntCtl_{\text{VS}}$!= 0, $Cause_{\text{IV}}$=1 $Status_{\text{BEV}}$=0).

The IRET instruction is meant to reverse the effects of the Interrupt Automated Prologue feature. So the IRET instruction should only be used when the COP0 registers are saved onto the stack in the manner specified by the IAP feature.

5.4.3.2 IAE Operation

Refer to the IRET instruction description.

5.4.3.3 Exceptions during IAE

The memory store operations which occur during Auto-Epilogue may result in Address Error, TLB refill, TLB invalid, TLB modify, Cache Error, Bus Error exceptions. If such memory exceptions occur during Auto-Epilogue:

- The $Cause_{ExcCode}$ register field reports the exception type.
- *EPC* is updated to the IRET instruction location.
- All of the other exception-reporting COP0 registers (*BadVaddr, EntryHi, EntryLo*, Context, CacheError*) are updated as appropriate for the exception type. These registers reflect the effective word address which caused the exception, e.g., as if an individual LW instruction caused the exception.
- If the memory store operation uses a mapped address and there is no matching address in the TLB, the TLB refill exception handler (offset 0x0) is used. The other TLB related exceptions (invalid, modify) use the general exception handler (offset 0x180).
- The Shadow Register Set designated by the $SRSCtl_{ESS}$ register field is used for the memory exception.
- The memory exception handler returns to the IRET instruction, which is held in *C0_EPC*.
- The IRET instruction now completes since the faulting condition was previously fixed. The IRET returns to the original code PC location, which is un-wound from the stack.

The IRET instruction will run to completion unless one of these memory exceptions takes place. The IRET instruction is not interruptable, that is, IRET is atomic from the point of view of another pending interrupt.

5.4.4 Interrupt Chaining

This feature reduces the number of cycles needed to respond to a subsequent higher priority interrupt when the processor is returning from exception mode and has disabled interrupts.

Normally, software has to disable interrupts during the critical section when restoring registers from a stack when finishing handling an exception. During that time, another interrupt could be signalled. The new interrupt is ignored until the ERET instruction clears the EXL bit and has started execution at the return address pointed by *EPC*. During this time, the pipeline is flushed to complete the exception handling. When the subsequent interrupt is finally recognized by the exception logic, a second pipeline flush is necessary as the processor was about start executing the instructions at the return address.

The Interrupt Chaining feature avoids these pipeline flushes by allowing the EIC unit to update its interrupts signals sent to the processor core before the IRET instruction completes. If these signals represent an interrupt which is higher priority than the current priority (in $Status_{IPL}$), the IRET instruction will update the COP0 registers as if just entering exception mode. The IRET instruction will then jump directly to the new interrupt vector - **avoiding** these steps:

1. Flushing the pipeline in return to non-exception mode
2. Clearing the $Status_{EXL}$ bit
3. Returning to the *EPC* address
4. Flushing the pipeline a second time to enter exception mode.

5.4.4.1 Interrupt Chaining Conditions

This feature is made available through the IRET instruction. Interrupt Chaining is only available when:

- Shadow Register Sets are implemented ($SRSCtl_{HSS}$!= 0)
- External Interrupt Controller Mode is enabled ($Config3_{VEIC}$=1, $IntCtl_{VS}$!= 0, $Cause_{IV}$=1 $Status_{BEV}$=0)
- $IntCtl_{ICE}$ = 1

5.5 Modified CP0 Registers

The CP0 registers provide the interface between the ISA and the PRA. Those CP0 registers that are extended or redefined for the MCU ASE relative to the MIPS32 Architecture reference are discussed below, with the registers presented in numerical order, first by register number, then by select field number.

5.5.1 CP0 Register Summary

Table 5.3 lists the CP0 registers affected by the MCU specification in numerical order. The individual registers are described later in this document. Otherwise the definition reverts to the MISP32 specification. The *Sel* column indicates the value to be used in the field of the same name in the MFC0 and MTC0 instructions.

Table 5.3 MCU Changes to Coprocessor 0 Registers in Numerical Order

Register Number	Sel	Register Name	Modification	Reference	Compliance Level
12	0	*Status*	IM/IPL field extended by 2 bits	Section 5.5.2	Required for MCU ASE
12	1	*IntCtl*	PF, ICE, StkDec, ClrEXL, APE, UseKStk fields added	Section 5.5.3	Required for MCU ASE
12	4	*View_IPL*	New Register	Section 5.5.4	Required for MCU ASE
12	5	*SRSMAP2*	New Register	Section 5.5.5	Required for MCU ASE
13	0	*Cause*	IC, AP fields added. IP/RIPL field extended by 2 bits.	Section 5.5.6	Required for MCU ASE
13	4	*View_RIPL*	New Register	Section 5.5.7	Required for MCU ASE
16	3	*Config3*	IPLW, MCU fields added.	Section 5.5.8	Required for MCU ASE

5.5.2 Status Register (CP Register 12, Select 0)

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.

Figure 5-1 shows the format of the *Status* register; Table 5.4 describes the *Status* register fields.

Figure 5-1 Status Register Format

31 28	27	26	25	24	23	22	21	20	19	18	17 16	15 10	9 8	7	6	5	4	3	2	1	0
CU3..CU0	RP	FR	RE	MX	PX	BEV	TS	SR	NMI	IM9	Impl	IM8..IM2	IM1..IM0	KX	SX	UX	UM	R0	ERL	EXL	IE
										IPL		IPL					KSU				

Table 5.4 Status Register Field Descriptions

Fields: Name	Fields: Bits	Description	Read / Write	Reset State	Compliance
CU (CU3.. CU0)	31..28	Controls access to coprocessors 3, 2, 1, and 0, respectively: **Encoding** / **Meaning** 0 — Access not allowed 1 — Access allowed Coprocessor 0 is always usable when the processor is running in Kernel Mode or Debug Mode, independent of the state of the CU_0 bit. In Release 2 of the Architecture, and for 64-bit implementations of Release 1 of the Architecture, execution of all floating point instructions, including those encoded with the COP1X opcode, is controlled by the CU1 enable. CU3 is no longer used and is reserved for future use by the Architecture. If there is no provision for connecting a coprocessor, the corresponding CU bit must be ignored on writes and return zero on reads.	R/W	Undefined	Required for all implemented coprocessors
RP	27	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 writes and return zero on reads. If this bit is implemented, the reset state must be zero so that the processor starts at full performance.	R/W	0	Optional

Table 5.4 Status Register Field Descriptions (Continued)

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read / Write</th><th rowspan="2">Reset State</th><th rowspan="2">Compliance</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>FR</td><td>26</td><td>In Release 1 of the Architecture, only MIPS64 processors could implement a 64-bit floating point unit. In Release 2 of the Architecture, both MIPS32 and MIPS64 processors can implement a 64-bit floating point unit. This bit is used to control the floating point register mode for 64-bit floating point units:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Floating point registers can contain any 32-bit data type. 64-bit data types are stored in even-odd pairs of registers.</td></tr>
<tr><td>1</td><td>Floating point registers can contain any datatype</td></tr>
</table>
This bit must be ignored on writes and return zero on reads under the following conditions:
• No floating point unit is implemented
• In a MIPS32 implementation of Release 1 of the Architecture
• In an implementation of Release 2 of the Architecture in which a 64-bit floating point unit is not implemented
Certain combinations of the FR bit and other state or operations can cause UNPREDICTABLE behavior.</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 writes and return zero on reads.</td><td>R/W</td><td>Undefined</td><td>Optional</td></tr>
<tr><td>MX</td><td>24</td><td>Enables access to MDMX and MIPS DSP resources on processors implementing one of these ASEs. If neither the MDMX nor the MIPS DSP ASE is implemented, this bit must be ignored on writes and return zero on reads.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Access not allowed</td></tr>
<tr><td>1</td><td>Access allowed</td></tr>
</table></td><td>R if the processor implements neither the MDMX nor the MIPS DSP ASEs; otherwise R/W</td><td>0 if the processor implements neither the MDMX nor the MIPS DSP ASEs; otherwise Undefined</td><td>Optional</td></tr>
<tr><td>PX</td><td>23</td><td>Enables access to 64-bit operations on MIPS64 processors. Not used by MIPS32 processors. This bit must be ignored on writes and return zero on reads.</td><td>R</td><td>0</td><td>Required</td></tr>
</table>

Table 5.4 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>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 80 for details.</td><td>R/W</td><td>1</td><td>Required</td></tr>
<tr><td>TS[1]</td><td>21</td><td>Indicates that the TLB has detected a match on multiple entries. It is implementation-dependent whether this detection occurs at all, on a write to the TLB, or an access to the TLB. In Release 2 of the Architecture, multiple TLB matches may only be reported on a TLB write. When such a detection occurs, the processor initiates a machine check exception and sets this bit. It is implementation-dependent whether this condition can be corrected by software. If the condition can be corrected, this bit should be cleared by software before resuming normal operation.
See “TLB Initialization” on page 44 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 writes and return zero on reads.
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 machine check exception.</td><td>R/W</td><td>0</td><td>Required if the processor detects and reports a match on multiple TLB entries</td></tr>
<tr><td>SR</td><td>20</td><td>Indicates that the entry through the reset exception vector was due to a Soft Reset:
<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Not Soft Reset (NMI or Reset)</td></tr><tr><td>1</td><td>Soft Reset</td></tr></table>If this bit is not implemented, it must be ignored on writes and return zero on reads.
Software should not write a 1 to this bit when its value is a 0, thereby causing a 0-to-1 transition. If such a transition is caused by software, it is UNPREDICTABLE whether hardware ignores or accepts the write.</td><td>R/W</td><td>1 for Soft Reset; 0 otherwise</td><td>Required if Soft Reset is implemented</td></tr>
</table>

Table 5.4 Status Register Field Descriptions (Continued)

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read / Write</th><th rowspan="2">Reset State</th><th rowspan="2">Compliance</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>NMI</td><td>19</td><td>Indicates that the entry through the reset exception vector was due to an NMI exception:
<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Not NMI (Soft Reset or Reset)</td></tr><tr><td>1</td><td>NMI</td></tr></table>If this bit is not implemented, it must be ignored on writes and return zero on reads.
Software should not write a 1 to this bit when its value is a 0, thereby causing a 0-to-1 transition. If such a transition is caused by software, it is UNPREDICTABLE whether hardware ignores or accepts the write.</td><td>R/W</td><td>1 for NMI; 0 otherwise</td><td>Required if NMI is implemented</td></tr>
<tr><td>0</td><td>18</td><td>Must be written as zero; returns zero on read.</td><td>0</td><td>0</td><td>Reserved</td></tr>
<tr><td>Impl</td><td>17</td><td>These bits are implementation-dependent and are not defined by the architecture. If they are not implemented, they must be ignored on writes and return zero on reads.</td><td></td><td>Undefined</td><td>Optional</td></tr>
<tr><td>IM9..IM2</td><td>18, 16..10</td><td>Interrupt Mask: Controls the enabling of each of the hardware interrupts. Refer to “Interrupts” on page 65 for a complete discussion of enabled interrupts.
<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Interrupt request disabled</td></tr><tr><td>1</td><td>Interrupt request enabled</td></tr></table>In implementations of Release 2 of the Architecture in which EIC interrupt mode is enabled (Config3$_{VEIC}$ = 1), these bits take on a different meaning and are interpreted as the IPL field, described below.</td><td>R/W</td><td>Undefined for IM7:IM2

0 for IM9:IM8</td><td>Required</td></tr>
<tr><td>IPL</td><td>18, 16..10</td><td>Interrupt Priority Level.
In implementations of Release 2 of the Architecture in which EIC interrupt mode is enabled (Config3$_{VEIC}$ = 1), this field is the encoded (0..63) value of the current IPL. An interrupt will be signaled only if the requested IPL is higher than this value.
If EIC interrupt mode is not enabled (Config3$_{VEIC}$ = 0), these bits take on a different meaning and are interpreted as the IM7..IM2 bits, described above.</td><td>R/W</td><td>Undefined for IPL15:IPL10

0 for IPL18:IPL17</td><td>Optional (Release 2 and EIC interrupt mode only)</td></tr>
</table>

Table 5.4 Status Register Field Descriptions (Continued)

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read / Write</th><th rowspan="2">Reset State</th><th rowspan="2">Compliance</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>IM1..IM0</td><td>9..8</td><td>Interrupt Mask: Controls the enabling of each of the software interrupts. Refer to “Interrupts” on page 65 for a complete discussion of enabled interrupts.
<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Interrupt request disabled</td></tr><tr><td>1</td><td>Interrupt request enabled</td></tr></table>In implementations of Release 2 of the Architecture in which EIC interrupt mode is enabled ($Config3_{VEIC}$ = 1), these bits are writable, but have no effect on the interrupt system.</td><td>R/W</td><td>Undefined</td><td>Required</td></tr>
<tr><td>KX</td><td>7</td><td>Enables access to 64-bit kernel address space on 64-bit MIPS processors. Not used by MIPS32 processors. This bit must be ignored on writes and return zero on reads.</td><td>R</td><td>0</td><td>Reserved</td></tr>
<tr><td>SX</td><td>6</td><td>Enables access to 64-bit supervisor address space on 64-bit MIPS processors. Not used by MIPS32 processors. This bit must be ignored on writes and return zero on reads.</td><td>R</td><td>0</td><td>Reserved</td></tr>
<tr><td>UX</td><td>5</td><td>Enables access to 64-bit user address space on 64-bit MIPS processors Not used by MIPS32 processors. This bit must be ignored on writes and return zero on reads.</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 “MIPS3264 and microMIPS3264 Operating Modes” on page 19 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 UNDEFINED if this value is written to the KSU field</td></tr></table>Note: This field overlaps the UM and R0 fields, described below.</td><td>R/W</td><td>Undefined</td><td>Required if Supervisor Mode is implemented; Optional otherwise</td></tr>
</table>

Table 5.4 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>UM</td><td>4</td><td>If Supervisor Mode is not implemented, this bit denotes the base operating mode of the processor. See “MIPS3264 and microMIPS3264 Operating Modes” on page 19 for a full discussion of operating modes. The encoding of this bit is:<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Base mode is Kernel Mode</td></tr><tr><td>1</td><td>Base mode is User Mode</td></tr></table>Note: This bit overlaps the KSU field, described above.</td><td>R/W</td><td>Undefined</td><td>Required</td></tr>
<tr><td>R0</td><td>3</td><td>If Supervisor Mode is not implemented, this bit is reserved. This bit must be ignored on writes and return zero on reads.
Note: This bit overlaps the KSU field, described above.</td><td>R</td><td>0</td><td>Reserved</td></tr>
<tr><td>ERL</td><td>2</td><td>Error Level; Set by the processor when a Reset, Soft Reset, NMI or Cache Error exception are taken.<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Normal level</td></tr><tr><td>1</td><td>Error level</td></tr></table>When ERL is set:
• The processor is running in kernel mode
• Hardware and software interrupts are disabled
• The ERET instruction will use the return address held in ErrorEPC instead of EPC
• Segment kuseg is treated as an unmapped and uncached region. See “Address Translation for the kuseg Segment when $Status_{ERL}$ = 1” on page 41. This allows main memory to be accessed in the presence of cache errors. The operation of the processor is UNDEFINED if the ERL bit is set while the processor is executing instructions from kuseg.</td><td>R/W</td><td>1</td><td>Required</td></tr>
<tr><td>EXL</td><td>1</td><td>Exception Level; Set by the processor when any exception other than Reset, Soft Reset, NMI or Cache Error exception are taken.<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Normal level</td></tr><tr><td>1</td><td>Exception level</td></tr></table>When EXL is set:
• The processor is running in Kernel Mode
• Hardware and software interrupts are disabled.
• TLB Refill exceptions use the general exception vector instead of the TLB Refill vector.
• EPC, $Cause_{BD}$ and SRSCtl (implementations of Release 2 of the Architecture only) will not be updated if another exception is taken</td><td>R/W</td><td>Undefined</td><td>Required</td></tr>
</table>

Table 5.4 Status Register Field Descriptions (Continued)

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

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

Programming Note:

In Release 2 of the Architecture, the EHB instruction can be used to make interrupt state changes visible when the *IM*, *IPL*, *ERL*, *EXL*, or *IE* fields of the *Status* register are written.

5.5.3 IntCtl (CP0 Registers 12, Select 1)

Figure 5-2 shows the format of the *IntCtl* register; Table 5.5 describes the *IntCtl* register fields.

Figure 5-2 IntCtl Register Format

31 29	28 26	25 23	22	21	20 16	15	14	13	12 10	9 5	4 0
IPTI	IPPCI	IPFDC	PF	ICE	StkDec	Clr-EXL	APE	Use KStk	000	VS	0

Table 5.5 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 UNPREDICTABLE 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>
<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 UNPREDICTABLE 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>
</table>

Table 5.5 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>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$_{FDC}$ 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 UNPREDICTABLE 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>PF</td><td>22</td><td>Enables Vector Prefetching Feature.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Vector Prefetching disabled</td></tr>
<tr><td>1</td><td>Vector Prefetching enabled</td></tr>
</table></td><td>RW</td><td>0</td><td>Required if MCU ASE is implemented</td></tr>
<tr><td>ICE</td><td>21</td><td>For IRET instruction. Enables Interrupt Chaining.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Interrupt Chaining disabled</td></tr>
<tr><td>1</td><td>Interrupt Chaining enabled</td></tr>
</table></td><td>RW</td><td>0</td><td>Required if MCU ASE is implemented</td></tr>
</table>

Table 5.5 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>StkDec</td><td>20..16</td><td>For Auto-Prologue feature. This is the number of 4-byte words that is decremented from the value of GPR29
<table>
<tr><th>Encoding</th><th>Decrement Amount in words</th><th>Decrement Amount in bytes</th></tr>
<tr><td>0-3</td><td>3</td><td>12</td></tr>
<tr><td>Others</td><td>As encoded, e.g. 0x5 means 5 words</td><td>4 * encoded value e.g. 0x5 means 20 bytes</td></tr>
</table></td><td>RW</td><td>0x3</td><td>Required if MCU ASE is implemented</td></tr>
<tr><td>ClrEXL</td><td>15</td><td>For Auto-Prologue feature and IRET instruction.
If set, during Auto-Prologue and IRET interrupt chaining, the KSU/ERL/EXL fields are cleared.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Fields are not cleared by these operations</td></tr>
<tr><td>1</td><td>Fields are cleared by these operations</td></tr>
</table></td><td>RW</td><td>0</td><td>Required if MCU ASE is implemented</td></tr>
<tr><td>APE</td><td>14</td><td>Enables Auto-Prologue feature.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Auto-Prologue disabled</td></tr>
<tr><td>1</td><td>Auto-Prologue enabled</td></tr>
</table></td><td>RW</td><td>0</td><td>Required if MCU ASE is implemented</td></tr>
</table>

Table 5.5 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>UseKStk</td><td>13</td><td>Chooses which Stack to use during Interrupt Automated Prologue.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Copy $29 of the Previous SRS to the Current SRS at the beginning of IAP.

This is used for Bare-Iron environments with only one stack.</td></tr>
<tr><td>1</td><td>Use $29 of the Current SRS at the beginning of IAP.

This is used for environments where there are separate User-mode and Kernel mode stacks. In this case, $29 of the SRS used during IAP must be pre-initialized by software to hold the Kernel mode stack pointer.</td></tr>
</table></td><td>RW</td><td>0</td><td>Required if MCU ASE is implemented</td></tr>
<tr><td>0</td><td>13..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>

5.5.4 View_IPL Register (CP0 Register 12, Select 4)

Figure 5-3 View_IPL Register Format

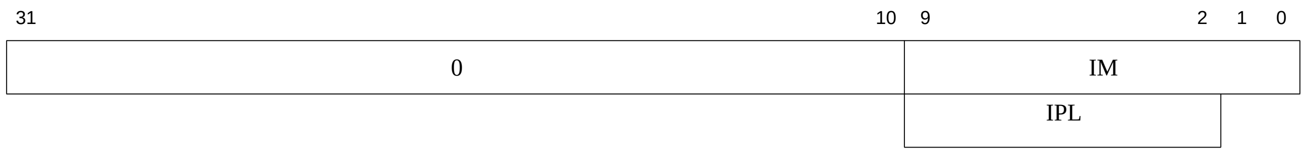

Table 5.6 View_IPL Register Field Descriptions

Fields		Description	Read / Write	Reset State	Compliance
Name	**Bits**				
IM	9:0	Interrupt Mask. If EIC interrupt mode is not enabled, controls which interrupts are enabled.	R/W	Undefined for IM7:IM2 0 for IM9:IM8	Required
IPL	9..2	Interrupt Priority Level. If EIC interrupt mode is enabled, this field is the encoded value of the current IPL.	R/W	Undefined	Required
0	31..10,1..0	Must be written as zero; returns zero on read.	0	0	Reserved

This register gives read and write access to the IM or IPL field that is also available in the *Status* Register. The use of this register allows the Interrupt Mask or the Priority Level to be read/written without extracting/inserting that bit field from/to the *Status* Register.

The IPL field might be located in non-contiguous bits within the *Status* Register. All of the IPL bits are presented as a contiguous field within this register.

5.5.5 SRSMap2 Register (CP0 Register 12, Select 5)

The *SRSMap2* register contains 2 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 *SRSMap2* register contains the shadow register set numbers for vector numbers 9..8. 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 5-4 shows the format of the *SRSMap2* register; Table 5.7 describes the *SRSMap2* register fields.

Figure 5-4 SRSMap Register Format

31 … 8	7 … 4	3 … 0
0	SSV9	SSV8

Table 5.7 SRSMap Register Field Descriptions

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
0	31..8	Must be written as zero; returns zero on read.	R	0	RESERVED
SSV9	7..4	Shadow register set number for Vector Number 9	R/W	0	Required
SSV8	3..0	Shadow register set number for Vector Number 8	R/W	0	Required

5.5.6 Cause Register (CP0 Register 13, Select 0)

Compliance Level: *Context* register modifications are *Required* for a MCU MMU.

Figure 5-5 shows the format of the *Cause* register; Table 5.8 describes the *Cause* register fields.

Figure 5-5 Cause Register Format

31	30	29 28	27	26	25	24	23	22	21	20 … 18	17 16 15 … 10	9 8	7	6 … 2	1 0
BD	TI	CE	DC	PCI	IC	AP	IV	WP	FDCI	0	IP9..IP2	IP1..IP0	0	Exc Code	0
											RIPL				

Table 5.8 Cause Register Field Descriptions

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
BD	31	Indicates whether the last exception taken occurred in a branch delay slot: **Encoding** / **Meaning** 0 — Not in delay slot 1 — In delay slot The processor updates BD only if $Status_{EXL}$ was zero when the exception occurred.	R	Undefined	Required

Table 5.8 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>TI</td><td>30</td><td>Timer Interrupt. In an implementation of Release 2 of the Architecture, this bit denotes whether a timer interrupt is pending (analogous to the IP bits for other interrupt types):
<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>No timer interrupt is pending</td></tr><tr><td>1</td><td>Timer interrupt is pending</td></tr></table>In an implementation of Release 1 of the Architecture, this bit must be written as zero and returns zero on read.</td><td>R</td><td>Undefined</td><td>Required (Release 2)</td></tr>
<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 UNPREDICTABLE for all exceptions except for Coprocessor Unusable.</td><td>R</td><td>Undefined</td><td>Required</td></tr>
<tr><td>DC</td><td>27</td><td>Disable Count register. In some power-sensitive applications, the Count register is not used but may still be the source of some noticeable power dissipation. This bit allows the Count register to be stopped in such situations.
<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Enable counting of Count register</td></tr><tr><td>1</td><td>Disable counting of Count register</td></tr></table>In an implementation of Release 1 of the Architecture, this bit must be written as zero, and returns zero on read.</td><td>R/W</td><td>0</td><td>Required (Release 2)</td></tr>
<tr><td>PCI</td><td>26</td><td>Performance Counter Interrupt. In an implementation of Release 2 of the Architecture, this bit denotes whether a performance counter interrupt is pending (analogous to the IP bits for other interrupt types):
<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>No performance counter interrupt is pending</td></tr><tr><td>1</td><td>Performance counter interrupt is pending</td></tr></table>In an implementation of Release 1 of the Architecture, or if performance counters are not implemented ($Config1_{PC}$ = 0), this bit must be written as zero and returns zero on read.</td><td>R</td><td>Undefined</td><td>Required (Release 2 and performance counters implemented)</td></tr>
</table>

Table 5.8 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>IC</td><td>25</td><td>Indicates if Interrupt Chaining occurred on the last IRET instruction.<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Interrupt Chaining did not happen on last IRET</td></tr><tr><td>1</td><td>Interrupt Chaining occurred during last IRET</td></tr></table></td><td>R</td><td>Undefined</td><td>Required if MCU ASE is implemented</td></tr>
<tr><td>AP</td><td>24</td><td>Indicates whether an exception occurred during Interrupt Auto-Prologue.<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Exception did not occur during Auto-Prologue operation.</td></tr><tr><td>1</td><td>Exception occurred during Auto-Prologue operation.</td></tr></table></td><td>R</td><td>Undefined</td><td>Required if MCU ASE is implemented</td></tr>
<tr><td>IV</td><td>23</td><td>Indicates whether an interrupt exception uses the general exception vector or a special interrupt vector:<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Use the general exception vector (0x180)</td></tr><tr><td>1</td><td>Use the special interrupt vector (0x200)</td></tr></table>In implementations of Release 2 of the architecture, if the $Cause_{IV}$ is 1 and $Status_{BEV}$ is 0, the special interrupt vector represents the base of the vectored interrupt table.</td><td>R/W</td><td>Undefined</td><td>Required</td></tr>
<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 writes and return zero on reads.</td><td>R/W</td><td>Undefined</td><td>Required if watch registers are implemented</td></tr>
</table>

Table 5.8 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>FDCI</td><td>21</td><td>Fast Debug Channel Interrupt. This bit denotes whether a FDC 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 Fast Debug Channel interrupt is pending</td></tr>
<tr><td>1</td><td>Fast Debug Channel interrupt is pending</td></tr>
</table></td><td>R</td><td>Undefined</td><td>Required if EJTAG Fast Debug Channel is implemented.</td></tr>
<tr><td>IP9..IP2</td><td>17..10</td><td>Indicates an interrupt is pending:
<table>
<tr><th>Bit</th><th>Name</th><th>Meaning</th></tr>
<tr><td>17</td><td>IP9</td><td>Hardware Interrupt 7</td></tr>
<tr><td>16</td><td>IP8</td><td>Hardware Interrupt 6</td></tr>
<tr><td>15</td><td>IP7</td><td>Hardware interrupt 5</td></tr>
<tr><td>14</td><td>IP6</td><td>Hardware interrupt 4</td></tr>
<tr><td>13</td><td>IP5</td><td>Hardware interrupt 3</td></tr>
<tr><td>12</td><td>IP4</td><td>Hardware interrupt 2</td></tr>
<tr><td>11</td><td>IP3</td><td>Hardware interrupt 1</td></tr>
<tr><td>10</td><td>IP2</td><td>Hardware interrupt 0</td></tr>
</table>
In implementations of Release 1 of the Architecture, timer and performance counter interrupts are combined in an implementation-dependent way with hardware interrupt 5. In implementations of Release 2 of the Architecture in which EIC interrupt mode is not enabled ($Config3_{VEIC}$ = 0), timer and performance counter interrupts are combined in an implementation-dependent way with any hardware interrupt. If EIC interrupt mode is enabled ($Config3_{VEIC}$ = 1), these bits take on a different meaning and are interpreted as the RIPL field, described below.</td><td>R</td><td>Undefined for IP7:IP2

0 for IP9:IP8</td><td>Required</td></tr>
<tr><td>RIPL</td><td>17..10</td><td>Requested Interrupt Priority Level.
In implementations of Release 2 of the Architecture in which EIC interrupt mode is enabled ($Config3_{VEIC}$ = 1), this field is the encoded (0..255) 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 for bits 15:10

0 for bits 17:16</td><td>Optional (Release 2 and EIC interrupt mode only)</td></tr>
</table>

Table 5.8 Cause Register Field Descriptions

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read / Write</th><th rowspan="2">Reset State</th><th rowspan="2">Compliance</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>IP1..IP0</td><td>9..8</td><td>Controls the request for software interrupts:<table><tr><th>Bit</th><th>Name</th><th>Meaning</th></tr><tr><td>9</td><td>IP1</td><td>Request software interrupt 1</td></tr><tr><td>8</td><td>IP0</td><td>Request software interrupt 0</td></tr></table>An implementation of Release 2 of the Architecture which also implements EIC interrupt mode exports these bits to the external interrupt controller for prioritization with other interrupt sources.</td><td>R/W</td><td>Undefined</td><td>Required</td></tr>
<tr><td>ExcCode</td><td>6..2</td><td>Exception code - see Table 5.9.</td><td>R</td><td>Undefined</td><td>Required</td></tr>
<tr><td>0</td><td>20..18, 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 5.9 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 $Debug_{\text{DExcCode}}$ field to denote an SDBBP in Debug Mode.
10	0x0a	RI	Reserved instruction exception
11	0x0b	CpU	Coprocessor Unusable exception
12	0x0c	Ov	Arithmetic Overflow exception
13	0x0d	Tr	Trap exception
14	0x0e	-	Reserved
15	0x0f	FPE	Floating point exception
16-17	0x10-0x11	-	Available for implementation-dependent use
18	0x12	C2E	Reserved for precise Coprocessor 2 exceptions

Table 5.9 Cause Register ExcCode Field

Exception Code Value			
Decimal	**Hexadecimal**	**Mnemonic**	**Description**
19-21	0x13-0x15	-	Reserved
22	0x16	MDMX	MDMX Unusable Exception (MDMX ASE)
23	0x17	WATCH	Reference to WatchHi/WatchLo address
24	0x18	MCheck	Machine check
25	0x19	Thread	Thread Allocation, Deallocation, or Scheduling Exceptions (MIPS® MT ASE)
26-29	0x20-0x1d	-	Reserved
30	0x1e	CacheErr	Cache error. In normal mode, a cache error exception has a dedicated vector and the *Cause* register is not updated. If EJTAG is implemented and a cache error occurs while in Debug Mode, this code is written to the $Debug_{DExcCode}$ field to indicate that re-entry to Debug Mode was caused by a cache error.
31	0x1f	-	Reserved

Programming Note:

In Release 2 of the Architecture, the EHB instruction can be used to make interrupt state changes visible when the $IP_{1..0}$ field of the *Cause* register is written.

5.5.7 View_RIPL Register (CP0 Register 13, Select 4)

Figure 5-6 View_RIPL Register Format

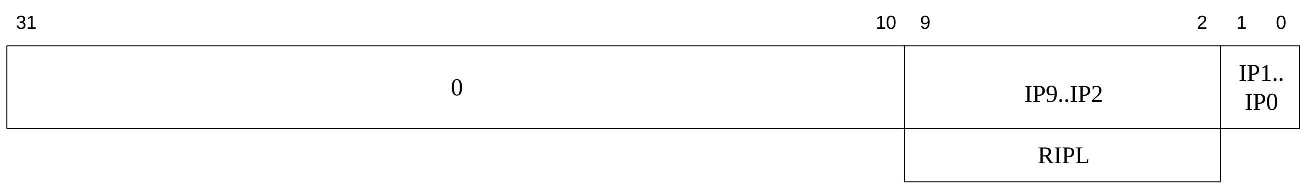

Table 5.10 View_RIPL Register Field Descriptions

Fields					
Name	**Bits**	**Description**	**Read / Write**	**Reset State**	**Compliance**
IP1..IP0	1:0	SW Interrupt Pending. If EIC interrupt mode is not enabled, controls which SW interrupts are pending.	R/W	Undefined	Required
IP9..IP2	9:2	HW Interrupt Pending. If EIC interrupt mode is not enabled, indicates which HW interrupts are pending.	R	Undefined for IP7:IP2 0 for IP9:IP8	Required

Table 5.10 View_RIPL Register Field Descriptions

Fields					
Name	**Bits**	**Description**	**Read / Write**	**Reset State**	**Compliance**
RIPL	9..2	Interrupt Priority Level. If EIC interrupt mode is enabled, this field indicates the Requested Priority Level of the pending interrupt.	R	Undefined	Required
0	31..10,1..0	Must be written as zero; returns zero on read.	0	0	Reserved

This register gives read access to the IP or RIPL field that is also available in the *Cause* Register. The use of this register allows the Interrupt Pending or the Requested Priority Level to be read without extracting that bit field from the *Cause* Register.

5.5.8 Config Register 3 (CP0 Register 16, Select 3)

Compliance Level: *Required* for a MCU MMU.

Figure 5-7 shows the format of the *Config3* register; Table 5.11 describes the *Config3* register fields.

Figure 5-7 Config3 Register Format

31	30 24 23	22 21	20 18	17	16	15 14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
M	0 00000000	IPLW	MMAR	MuCon	ISAOnExc	ISA	ULRI	0	DSP2P	DSPP	0	ITL	LPA	VEIC	VInt	SP	CDMM	MT	SM	TL

Table 5.11 Config3 Register Field Descriptions

Fields					
Name	**Bits**	**Description**	**Read / Write**	**Reset State**	**Compliance**
M	31	This bit is reserved to indicate that a *Config4* register is present. With the current architectural definition, this bit should always read as a 0.	R	Preset by hardware	Required
0	30:23,. 12, 9	Must be written as zeros; returns zeros on read	0	0	Reserved

Table 5.11 Config3 Register Field Descriptions

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read / Write</th><th rowspan="2">Reset State</th><th rowspan="2">Compliance</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>IPLW</td><td>22:21</td><td>Width of the $Status_{IPL}$ and $Cause_{RIPL}$ fields:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>IPL and RIPL fields are 6-bits in width.</td></tr>
<tr><td>1</td><td>IPL and RIPL fields are 8-bits in width.</td></tr>
<tr><td>Others</td><td>Reserved.</td></tr>
</table>
If the IPL field is 8-bits in width, bits 18 and 16 of Status are used as the most significant bit and second most significant bit, respectively, of that field.

If the RIPL field is 8-bits in width, bits 17 and 16 of Cause 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>microMIPS Architecture revision level:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Release 1</td></tr>
<tr><td>1-7</td><td>Reserved</td></tr>
</table></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 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>ISAOn-Exc</td><td>16</td><td>Reflects the Instruction Set Architecture used when vectoring to an exception. Affects exceptions whose vectors are offsets from EBASE.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>MIPS32ISA is used on entrance to an exception vector.</td></tr>
<tr><td>1</td><td>microMIPS ISA is used on entrance to an exception vector.</td></tr>
</table></td><td>RW</td><td>Preset by hardware, driven by signal external to CPU core</td><td>Required if both microMIPS and MIPS32are implemented</td></tr>
</table>

Table 5.11 Config3 Register Field Descriptions

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read / Write</th><th rowspan="2">Reset State</th><th rowspan="2">Compliance</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>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 is implemented.</td></tr>
<tr><td>1</td><td>Only microMIPS is implemented.</td></tr>
<tr><td>2</td><td>Both MIPS32and MicroMIPS ISAs are implemented. MIPS32 ISA used when coming out of reset.</td></tr>
<tr><td>3</td><td>Both MIPS32 and MicroMIPS ISAs are implemented. MicroMIPS ISA used when coming out of reset.</td></tr>
</table></td><td>R</td><td>Preset by hardware, driven by signal external to CPU core</td><td>Required if both microMIPS and MIPS32are 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>DSP2P</td><td>11</td><td>MIPS® DSP ASE Revision 2 implemented. This bit indicates whether Revision 2 of the MIPS DSP ASE is implemented.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Revision 2 of the MIPS DSP ASE is not implemented</td></tr>
<tr><td>1</td><td>Revision 2 of the MIPS DSP ASE is implemented</td></tr>
</table></td><td>R</td><td>Preset by hardware</td><td>Required</td></tr>
<tr><td>DSPP</td><td>10</td><td>MIPS® DSP ASE implemented. This bit indicates whether the MIPS DSP ASE is implemented.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>MIPS DSP ASE is not implemented</td></tr>
<tr><td>1</td><td>MIPS DSP ASE is implemented</td></tr>
</table></td><td>R</td><td>Preset 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>
</table>

Table 5.11 Config3 Register Field Descriptions

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read / Write</th><th rowspan="2">Reset State</th><th rowspan="2">Compliance</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>LPA</td><td>7</td><td>Denotes the presence of support for large physical addresses on MIPS64 processors. Not used by MIPS32 processors and returns zero on read.
For implementations of Release 1 of the Architecture, this bit returns zero on read.</td><td>R</td><td>Preset by hardware</td><td>Required (Release 2 Only)</td></tr>
<tr><td>VEIC</td><td>6</td><td>Support for an external interrupt controller is implemented.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Support for EIC interrupt mode is not implemented</td></tr>
<tr><td>1</td><td>Support for EIC interrupt mode is implemented</td></tr>
</table>
For implementations of Release 1 of the Architecture, this bit returns zero on read.
This bit indicates not only that the processor contains support for an external interrupt controller, but that such a controller is attached.</td><td>R</td><td>Preset 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><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Small page support is not implemented</td></tr>
<tr><td>1</td><td>Small page support is implemented</td></tr>
</table>
For implementations of Release 1 of the Architecture, this bit returns zero on read.</td><td>R</td><td>Preset 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 5.11 Config3 Register Field Descriptions

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read / Write</th><th rowspan="2">Reset State</th><th rowspan="2">Compliance</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>MT</td><td>2</td><td>MIPS MT ASE implemented. This bit indicates whether the MIPS MT ASE is implemented.<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>MIPS MT ASE is not implemented</td></tr><tr><td>1</td><td>MIPS MT ASE is implemented</td></tr></table></td><td>R</td><td>Preset 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>

Revision History

.

Version	Date	Comments
0.80	December 1, 2009	• Cleanup for external distribution - make Title more sensible.
0.81	January 15, 2010	• Re-phased the conditions for UseKStk=0/1 conditions in IAP section. • Clean-up of IRET description • 1. IRET always clears LLBit • 2. IRET acts as if EXL is always clear for its memory TLB exceptions. • 3. IRET only modifies the SW write-able fields of the SRSCtl register. • 4. IRET checks ISAMode bit when chaining is done.
1.00	March 20, 2010	• Item 4 was incorrect in 0.81 revision, IRET should check $Config3_{ISAOnDebug}$ • Clear Change-bars • For M14K* GA release.
1.01	March 21,2011	• AFP version - change security classification
1.02	December 16, 2012	• Update Cover logos • Update copyright text. • About this Book chapter updated for R5 (MT, DSP, VZ, MSA modules)
1.03	September 9, 2013	• Update Cover logos and copyright text