MIPS32® Architecture for Programmers Volume IV-d: The SmartMIPS® Application-Specific Extension to the MIPS32® Architecture

Document Number: MD00101
Revision 2.51
July 15, 2008

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

Template: nB1.02, Built with tags: 2B ARCH MIPS32

Table of Contents

Chapter 1: About This Book 7
1.1: Typographical Conventions 7
1.1.1: Italic Text 7
1.1.2: Bold Text 7
1.1.3: Courier Text 8
1.2: UNPREDICTABLE and UNDEFINED 8
1.2.1: UNPREDICTABLE 8
1.2.2: UNDEFINED 8
1.2.3: UNSTABLE 9
1.3: Special Symbols in Pseudocode Notation 9
1.4: For More Information 11

Chapter 2: Guide to the Instruction Set 13
2.1: Understanding the Instruction Fields 13
2.1.1: Instruction Fields 15
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
2.3: Op and Function Subfield Notation 27
2.4: FPU Instructions 27

Chapter 3: The SmartMIPS® Application-Specific Extension to the MIPS32® Architecture 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 SmartMIPS ASE 29
3.4.1: Support for Cryptographic Algorithms in the SmartMIPS ASE 29
3.4.2: Code Density Optimization 30
3.4.3: Other ISA Enhancements 31
3.4.4: Privileged Resource Architecture Enhancements 31
3.5: Instruction Bit Encoding 31

Chapter 4: The SmartMIPS® Cryptographic Feature Set 35
4.1: The Special Register ACX 35
4.2: Change to MADDU Semantics 36
4.3: Change to MULTU Semantics 36
4.4: Possible Changes to other Multiply/Accumulate Semantics 36
4.5: New Instructions 36

4.5.1: MFLHXU 36
4.5.2: MTLHX 36
4.5.3: MADDP 36
4.5.4: MULTP 36
4.5.5: PPERM 37
4.5.6: ROTR 37
4.5.7: ROTRV 37

Chapter 5: Other ISA Elements of the SmartMIPS® ASE 39
5.1: LWXS Instruction 39

Chapter 6: The SmartMIPS® Privileged Resource Architecture 51
6.1: Introduction 51
6.2: Interaction between the SmartMIPS ASE and Release 2 of the MIPS32 Architecture 51
6.3: Overview 51
6.4: Compliance 52
6.5: The SmartMIPS System Coprocessor 52
6.5.1: CP0 Register Summary 52
6.6: Virtual Memory 52
6.6.1: TLB-Based Virtual Address Translation 52
6.6.2: General Exception Processing 55
6.6.3: TLB Refill Exception 56
6.6.4: TLB Invalid Exception 56
6.6.5: TLB Modified Exception 56
6.7: CP0 Registers 57
6.7.1: EntryLo0, EntryLo1 (CP0 Registers 2 and 3, Select 0) 57
6.7.2: Context Register (CP0 Register 4, Select 0) 58
6.7.3: ContextConfig Register (CP0 Register 4, Select 1) 59
6.7.4: PageMask Register (CP0 Register 5, Select 0) 60
6.7.5: PageGrain Register (CP0 Register 5, Select 1) 62
6.7.6: EntryHi Register (CP0 Register 10, Select 0 64
6.7.7: Configuration Register 3 (CP0 Register 16, Select 3) 64

Appendix A: Revision History 65

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 6.1: Contents of a TLB Entry 53
Figure 6.2: SmartMIPS EntryLo0, EntryLo1 Register Format 57
Figure 6.3: SmartMIPS Context Register Format 59
Figure 6.4: SmartMIPS ContextConfig Register Format 60
Figure 6.5: SmartMIPS PageMask Register Format 61
Figure 6.6: SmartMIPS PageGrain Register Format 62
Figure 6.7: SmartMIPS EntryHi Register Format 64

List of Tables

Table 1.1: Symbols Used in Instruction Operation Statements 9
Table 2.1: AccessLength Specifications for Loads/Stores 22
Table 3.1: Symbols Used in the Instruction Encoding Tables 31
Table 3.2: SmartMIPS ASE Encoding of the Opcode Field 32
Table 3.3: SmartMIPS ASE SPECIAL Opcode Encoding of Function Field 32
Table 3.4: SmartMIPS ASE SPECIAL2 Encoding of Function Field 33
Table 3.5: SmartMIPS ASE SRL Encoding of Shift/Rotate 33
Table 3.6: SmartMIPS ASE SRLV Encoding of Shift/Rotate 33
Table 3.7: SmartMIPS ASE MFLO Encoding of MFLO/MFLHXU 33
Table 3.8: SmartMIPS ASE MTLO Encoding of MTLO/MTLHX 33
Table 3.9: SmartMIPS ASE MULTU Encoding of MULTU/MULTP 33
Table 3.10: SmartMIPS ASE MADDU Encoding of MADDU/MADDP/PPERM 34
Table 3.11: SmartMIPS ASE LXS Encoding of LWXS 34
Table 6.1: SmartMIPS Changes to Coprocessor 0 Registers in Numerical Order 52
Table 6.2: Physical Address Generation 55
Table 6.3: TLB Refill Exception State Saved in Addition to the Cause Register 56
Table 6.4: TLB Invalid Exception State Saved in Addition to the Cause Register 56
Table 6.5: TLB Modified Exception State Saved in Addition to the Cause Register 56
Table 6.6: SmartMIPS EntryLo0, EntryLo1 Register Field Descriptions 57
Table 6.7: SmartMIPS Context Register Field Descriptions 59
Table 6.8: SmartMIPS ContextConfig Register Field Descriptions 60
Table 6.9: Recommended ContextConfig Values for SmartMIPS 60
Table 6.10: PageMask Register Field Descriptions 61
Table 6.11: Values for the Mask Field of the PageMask Register 61
Table 6.12: SmartMIPS PageGrain Register Field Descriptions 62
Table 6.13: PageGrain Implementation Subset Behavior 63
Table 6.14: EntryHi Register Field Descriptions 64

Chapter 1

About This Book

The MIPS32® Architecture for Programmers Volume IV-d comes as a multi-volume set.

- Volume I describes conventions used throughout the document set, and provides an introduction to the MIPS32® Architecture
- Volume II provides detailed descriptions of each instruction in the MIPS32® instruction set
- Volume III describes the MIPS32® Privileged Resource Architecture which defines and governs the behavior of the privileged resources included in a MIPS32® processor implementation
- Volume IV-a describes the MIPS16e™ Application-Specific Extension to the MIPS32® Architecture
- Volume IV-b describes the MDMX™ Application-Specific Extension to the MIPS32® Architecture and is not applicable to the MIPS32® document set
- Volume IV-c describes the MIPS-3D® Application-Specific Extension to the MIPS32® Architecture
- Volume IV-d describes the SmartMIPS®Application-Specific Extension to the MIPS32® Architecture

1.1 Typographical Conventions

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

1.1.1 Italic Text

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

1.1.2 Bold Text

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

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

1.1.3 Courier Text

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

1.2 UNPREDICTABLE and UNDEFINED

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

1.2.1 UNPREDICTABLE

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

UNPREDICTABLE results or operations have several implementation restrictions:

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

1.2.2 UNDEFINED

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

UNDEFINED operations or behavior has one implementation restriction:

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

1.2.3 UNSTABLE

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

UNSTABLE values have one implementation restriction:

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

1.3 Special Symbols in Pseudocode Notation

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

Table 1.1 Symbols Used in Instruction Operation Statements

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

Table 1.1 Symbols Used in Instruction Operation Statements (Continued)

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

Table 1.1 Symbols Used in Instruction Operation Statements (Continued)

Symbol	Meaning
PC	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.
ISA Mode	In processors that implement the MIPS16e Application Specific Extension, the *ISA Mode* is a single-bit register that determines in which mode the processor is executing, as follows: **Encoding** / **Meaning** 0 — The processor is executing 32-bit MIPS instructions 1 — The processor is executing MIIPS16e instructions 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.
PABITS	The number of physical address bits implemented is represented by the symbol PABITS. As such, if 36 physical address bits were implemented, the size of the physical address space would be $2^{PABITS} = 2^{36}$ bytes.
FP32RegistersMode	Indicates whether the FPU has 32-bit or 64-bit floating point registers (FPRs). In MIPS32, the FPU has 32 32-bit FPRs in which 64-bit data types are stored in even-odd pairs of FPRs. In MIPS64, the FPU has 32 64-bit FPRs in which 64-bit data types are stored in any FPR. In MIPS32 implementations, **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

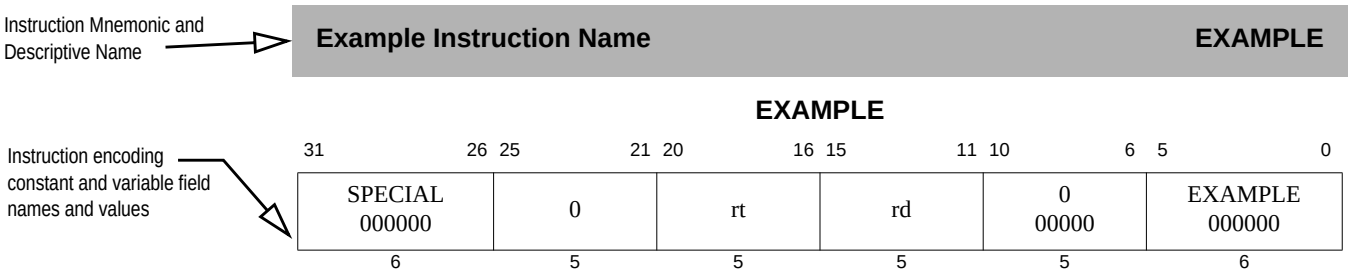
This chapter provides a detailed guide to understanding the instruction descriptions, which are listed in alphabetical order in the tables at the beginning of the next chapter.

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 15
- "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



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

Assembler format(s) for each definition

Format: `EXAMPLE fd,rs,rt`

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:

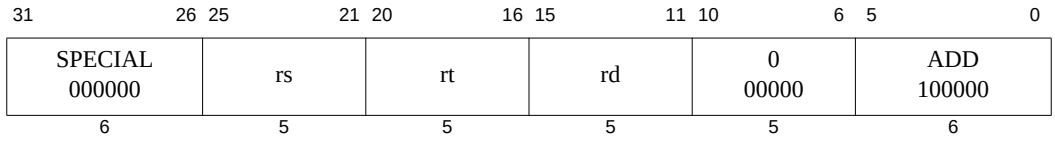
Like *Programming Notes*, except for processor implementors

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



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)
   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 SmartMIPS® Application-Specific Extension to the MIPS32® Architecture

3.1 Base Architecture Requirements

The SmartMIPS® ASE requires the following base architecture support:

- **The MIPS32 Architecture**: The SmartMIPS ASE requires a compliant implementation of the MIPS32 Architecture.

3.2 Software Detection of the ASE

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

3.3 Compliance and Subsetting

There are no instruction subsets of the SmartMIPS ASE to the MIPS32 Architecture – all SmartMIPS instructions must be implemented.

3.4 Overview of the SmartMIPS ASE

The SmartMIPS ASE extends the MIPS32® Architecture with set of new instructions combined with a set of backward-compatible modifications to existing MIPS32 instructions, designed to improve the performance and reduce the memory consumption of MIPS-based smart card or “smart object” systems. The SmartMIPS ASE contains enhancements in several distinct areas: cryptographic processing, code density, and virtual machine performance.

3.4.1 Support for Cryptographic Algorithms in the SmartMIPS ASE

The SmartMIPS ASE includes a package of extensions to MIPS32 to enhance the performance of cryptographic algorithms. Cryptographic algorithms can be generally divided into two categories - public key algorithms and secret key algorithms. Secret key algorithms, also known as symmetric algorithms, encrypt and decrypt with the same key, while public key algorithms operate in terms of key pairs, one for encryption and the other for decryption.

3.4.1.1 Secret Key Cryptography

Secret key algorithms are generally computationally relatively simple and frequently reducible to simple hardware solutions performing sequences of XORs, rotations, and permutations on blocks of data.

The SmartMIPS ASE contains the following elements for accelerating software implementations of secret key cryptography:

- A partial permutation instruction, capable of permuting 6 bits per instruction.
- A single-instruction bitwise rotate capability.

3.4.1.2 Public Key Cryptography

Public key cryptosystems are mathematically more subtle and computationally more difficult than private-key systems. While different schemes have different bases in mathematics, they tend to have a common need for integer computation across very large ranges of values, on the order of 1024 bits. This extended precision arithmetic is often modular (operating modulo the value range), and in some cases polynomial instead of twos-complement. Research conducted with industry partners has led us to conclude that accelerating extended-precision modular arithmetic provides a significant improvement in performance across a range of public key cryptography schemes.

The SmartMIPS ASE contains the following elements for accelerating public key cryptography:

- Additional architecturally visible accumulator state containing extended carry information.
- Instructions to allow the extended carry state to be initialized, extracted, saved, and restored.
- Extension of the definition of the MIPS32 MADDU instruction to generate and use the extended carry state.
- Instructions which allow multiplication and accumulation of polynomial-basis values.

3.4.2 Code Density Optimization

The SmartMIPS ASE includes the new and enhanced version of the MIPS16 ASE. This ASE is fully documented in Volume IVa of the MIPS Architecture reference, and is outside the scope of this document. Relative to the earlier version of MIPS16, there are enhancements in the following areas:

3.4.2.1 Data Type Conversion

In virtual machines and other software that very frequently handles data elements smaller than a 32-bit word, sign and zero extension operations must be performed on those data elements before they can be used computationally. In MIPS16, these operations require relatively large instruction sequences, on the order of 8 bytes per conversion. The Enhanced MIPS16 ASE provides a set of specific instructions to perform zero and sign-extension of bytes and 16-bit halfwords, bringing the footprint down to 2 bytes per conversion.

3.4.2.2 Jump Delay Slot Suppression

The MIPS16 ASE preserved the “delay slot” following the jump instructions used for subroutine call and return. The compiler can frequently, but by no means always, fill these delay slots with a useful instruction. Where it cannot, the MIPS16 ASE required that a no-op instruction be inserted into the instruction stream, at a cost of 2 bytes of footprint. The Enhanced MIPS16 ASE provides variant jump-via-register instructions that suppress these visible delay slots and eliminate the need for those no-ops.

3.4.2.3 Stack Frame Set-up and Tear-down

In generating code compatible with the MIPS Application Binary Interface (ABI) calling conventions, the compiler must assure that each subroutine set up and tear down a stack frame on which register values are saved and where local variables can be stored. The process of storing register values and updating the stack pointer on subroutine entry, and of restoring the values of the registers and the stack pointer on subroutine exit, can consume a significant amount of code, particularly in system composed of many small subroutines.

While a convention exists for trapping into the operating system from MIPS16 code, and having the operating system perform the necessary stack frame maintenance, this imposes a significant burden on small operating systems and has a significant impact on performance. The Enhanced MIPS16 ISA provides instructions that allow stack frame setup and restoration each to be done in a single compressed instruction.

3.4.3 Other ISA Enhancements

In order to accelerate the interpretation of JavaCard bytecodes and similar interpretive languages, SmartMIPS introduces a scaled, indexed 32-bit load instruction.

3.4.4 Privileged Resource Architecture Enhancements

In addition to an augmented instruction set, SmartMIPS defines an augmented privileged resource architecture with augmented memory management capabilities.

- True Read-only, Write-only, and Execute-only page protection are supported.
- 2K or 1K virtual memory pages can be supported.
- A more flexible CP0 *Context* register is provided to accelerate TLB lookups in small memory systems.

3.5 Instruction Bit Encoding

Table 3.2 through Table 3.11 describe the encoding used for the SmartMIPS ASE. Table 3.1 describes the meaning of the symbols used in the tables. These tables only list the instruction encodings for the SmartMIPS instructions. See Volume I of this multi-volume set for a full encoding of all instructions.

Table 3.1 Symbols Used in the Instruction Encoding Tables

Symbol	Meaning
*	Operation or field codes marked with this symbol are reserved for future use. Executing such an instruction must cause a Reserved Instruction Exception.
δ	(Also *italic* field name.) Operation or field codes marked with this symbol denotes a field class. The instruction word must be further decoded by examining additional tables that show values for another instruction field.
β	Operation or field codes marked with this symbol represent a valid encoding for a higher-order MIPS ISA level. Executing such an instruction must cause a Reserved Instruction Exception.
θ	Operation or field codes marked with this symbol are available to licensed MIPS partners. To avoid multiple conflicting instruction definitions, MIPS Technologies will assist the partner in selecting appropriate encodings if requested by the partner. The partner is not required to consult with MIPS Technologies when one of these encodings is used. If no instruction is encoded with this value, executing such an instruction must cause a Reserved Instruction Exception (*SPECIAL2* encodings or coprocessor instruction encodings for a coprocessor to which access is allowed) or a Coprocessor Unusable Exception (coprocessor instruction encodings for a coprocessor to which access is not allowed).
σ	Field codes marked with this symbol represent an EJTAG support instruction and implementation of this encoding is optional for each implementation. If the encoding is not implemented, executing such an instruction must cause a Reserved Instruction Exception. If the encoding is implemented, it must match the instruction encoding as shown in the table.

Table 3.1 Symbols Used in the Instruction Encoding Tables

Symbol	Meaning
ε	Operation or field codes marked with this symbol are reserved for MIPS Application Specific Extensions. If the ASE is not implemented, executing such an instruction must cause a Reserved Instruction Exception.
ϕ	Operation or field codes marked with this symbol are obsolete and will be removed from a future revision of the MIPS32 ISA. Software should avoid using these operation or field codes.

Table 3.2 SmartMIPS ASE Encoding of the Opcode Field

opcode		*bits 28..26*							
		0	1	2	3	4	5	6	7
bits 31..29		000	001	010	011	100	101	110	111
0	000	*SPECIAL* δ							
1	001								
2	010								
3	011					*SPECIAL2* δ			
4	100								
5	101								
6	110								
7	111								

Table 3.3 SmartMIPS ASE *SPECIAL* Opcode Encoding of Function Field

function		*bits 2..0*							
		0	1	2	3	4	5	6	7
bits 5..3		000	001	010	011	100	101	110	111
0	000			*SRL* δ				*SRLV* δ	
1	001								
2	010			*MFLO* δ	*MTLO* δ				
3	011		*MULTU* δ						
4	100								
5	101								
6	110								
7	111								

Table 3.4 SmartMIPS ASE *SPECIAL2* Encoding of Function Field

function		*bits 2..0*							
		0	1	2	3	4	5	6	7
bits 5..3		000	001	010	011	100	101	110	111
0	000		*MADDU* δ						
1	001	*LXS* δ							
2	010								
3	011								
4	100								
5	101								
6	110								
7	111								

Table 3.5 SmartMIPS ASE *SRL* Encoding of Shift/Rotate

R	*bit 21*	
	0	1
	SRL	ROTR

Table 3.6 SmartMIPS ASE *SRLV* Encoding of Shift/Rotate

R	*bit 6*	
	0	1
	SRLV	ROTRV

Table 3.7 SmartMIPS ASE *MFLO* Encoding of MFLO/MFLHXU

sa	*bits 10..6*	
	0b00000	0b00001
	MFLO	MFLHXU

Table 3.8 SmartMIPS ASE *MTLO* Encoding of MTLO/MTLHX

sa	*bits 10..6*	
	0b00000	0b00001
	MTLO	MTLHX

Table 3.9 SmartMIPS ASE *MULTU* Encoding of MULTU/MULTP

sa	*bits 10..6*	
	0b00000	0b10001
	MULTU	MULTP

Table 3.10 SmartMIPS ASE *MADDU* Encoding of MADDU/MADDP/PPERM

sa	*bits 10..6*		
	0b00000	0b10001	0b10010
	MADDU	MADDP	PPERM

Table 3.11 SmartMIPS ASE *LXS* Encoding of LWXS

sa	*bits 10..6*
	0b00001
	LWXS

Chapter 4

The SmartMIPS® Cryptographic Feature Set

The SmartMIPS ASE includes a set of features that form a cryptographic extension to the MIPS32 Architecture. This adds the following new architecturally visible state:

- A special accumulator extension register, ACX.

It modifies the following MIPS32/MIPS64 instructions to generate and consume the new ACX state:

- MADDU
- MULTU

It adds the following new instructions to extract and restore the new ACX state:

- MFLHXU
- MTLHX

It adds the following new instructions to implement efficient binary polynomial arithmetic in the multiply/divide unit:

- MADDP
- MULTP

It adds a partial permutation instruction to accelerate bit-permutation of data.

- PPERM

And it adds bitwise rotate instructions that operates on the general register set:

- ROTR
- ROTRV

4.1 The Special Register ACX

The special register ACX contains some number of bits of additional integer precision beyond those contained in the Hi/Lo special register pair. The precise number of bits is implementation dependent, but can be trivially determined at run-time by software. The minimum architectural size of the ACX register is 8 bits. The maximum architectural size of the ACX register is 64 bits for a MIPS64 processor and 32 bits for a MIPS32 processor. The currently recommended implementation size of the ACX register is 8 bits.

4.2 Change to MADDU Semantics

Whereas the MIPS32 MADDU instruction is defined to produce a 64-bit result, the SmartMIPS ASE MADDU instruction produces a higher-precision result, with the carry out of the Hi register propagating into the ACX register. The behavior of the instruction as seen by the MIPS32 ISA is unchanged.

4.3 Change to MULTU Semantics

The definition of the MIPS32 MULTU instruction is extended to clear the ACX register to all zeroes.

4.4 Possible Changes to other Multiply/Accumulate Semantics

While only MADDU, MULTU, MADDP, and MULTP need affect the ACX register to implement the extended-precision modular arithmetic algorithms targeted by the ASE, in the interests of consistency and design simplicity, it is possible that the MADD instruction will also generate ACX bits should the Hi register overflow, and that the MULT, DIV, DIVU, DMULT, DMULTU, DDIV and DDIVU instructions will clear the ACX bits. *Code written to the SmartMIPS ASE should make no assumptions about the behavior of the ACX bits during the execution of any instruction that writes both the Hi and Lo special registers, other than MULTU, MADDU, MULTP, and MADDP as described in this document.*

4.5 New Instructions

4.5.1 MFLHXU

The MFLHXU, or Move-from-LO-HI-ACX-Unsigned instruction, in effect shifts the extended precision accumulator formed by the ACX, Hi, and Lo registers to the right by one register position: the Lo register is copied into the specified GPR, the Hi register is copied into the Lo register, and the ACX register is zero-extended and copied into the Hi register. The instruction is designated as “unsigned” because the ACX bits are zero-extended and not sign-extended when they are transferred to a Hi register of higher precision.

4.5.2 MTLHX

The MTLHXI, or Move-to-LO-HI-ACX instruction, is the inverse operation from MFLHXU. It effectively shifts the extended-precision accumulator formed by the ACX, Hi, and Lo registers to the left by one register position: The Hi register is truncated to the width of the ACX register and the remaining lower bits are copied to the ACX register, the Lo register is copied to the Hi register, and the specified GPR is copied to the Lo register.

4.5.3 MADDP

Binary polynomial addition and multiplication form the basis for an important family of elliptical curve (EC) cryptosystems. The MADDP instruction performs a polynomial basis multiply of a pair of general registers, and then performs a polynomial basis add of the resulting product to the contents of the Hi/Lo register pair.

4.5.4 MULTP

The MULTP instruction performs a binary polynomial basis multiply of a pair of general registers, placing the result in the Hi/Lo register pair.

4.5.5 PPERM

The PPERM instruction performs a partial permutation of a value in a general register into the ACX-HI-LO registers. The ACX-HI-LO registers are shifted left by six bit positions, and the least significant six bits of LO are set to the values of arbitrary bits in an input register, based on a set of six 5-bit bit specifiers in the second general purpose input register. This allows for an arbitrary permutation of 32 bits of a GPR in 7 instructions: 6 PPERMS and a MFLO or MFLHX.

4.5.6 ROTR

DES and the candidate AES secret-key block ciphers all require rotations of 32-bit quantities. In the MIPS32 ISA, this requires a sequence of 3 instructions to perform. The ROTR instruction performs a bitwise rotation of a general purpose register of up to 31 bits in a single instruction. The definition and encoding of the ROTR instruction in the SmartMIPS ASE is identical to that of the same instruction in Release 2 of the MIPS Architecture.

4.5.7 ROTRV

The ROTRV instruction performs a variable-length rotation of a general purpose register, with the number of bits to be rotated determined at run-time by the value of another general purpose register. The definition and encoding of the ROTRV instruction in the SmartMIPS ASE is identical to that of the same instruction in Release 2 of the MIPS Architecture.

Chapter 5

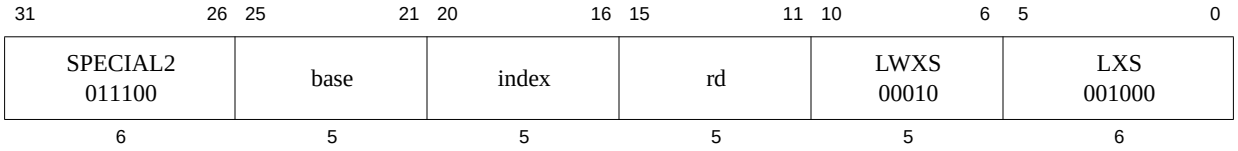
Other ISA Elements of the SmartMIPS® ASE

In addition to the cryptography feature set and the enhanced MIPS16 ASE, the SmartMIPS ASE contains the following instructions:

- LWXS - Scaled, indexed word load.

5.1 LWXS Instruction

The inner-loop function of interpreters for JavaCard bytecodes and similar interpretive languages requires a dispatch to a function based on an integer value. The LWXS instruction reduces and accelerates such loops, by integrating a scaling of an integer operand into a word offset and an address generation based on using the scaled value as an offset relative to a base register.



Format: `LWXS rd, index(base)` **SmartMIPS**

Purpose: Load Word Indexed, Scaled

To load a word from memory as a signed value, using scaled indexed addressing**.**

Description: `GPR[rd] ← memory[GPR[base] + (GPR[index] × 4)]`

The contents of GPR *index* is multiplied by 4 and the result is added to the contents of GPR *base* to form an effective address. The contents of the 32-bit word at the memory location specified by the aligned effective address are fetched, sign-extended to the GPR register length if necessary, and placed in GPR *rd*.

Restrictions:

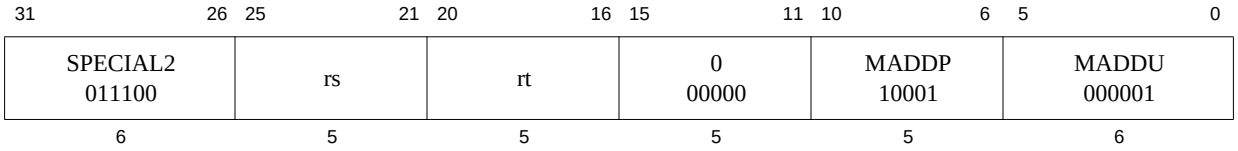
The effective address must be naturally-aligned. If either of the 2 least-significant bits of the address is non-zero, an Address Error exception occurs.

Operation:

```
vAddr ← (GPR[index]29..0 || 02) + GPR[base]
if vAddr1..0 ≠ 02 then
   SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
memword ← LoadMemory (CCA, WORD, pAddr, vAddr, DATA)
GPR[rd] ← memword
```

Exceptions:

TLB Refill, TLB Invalid, Bus Error, Address Error



Format: `MADDP rs, rt` **SmartMIPS Crypto**

Purpose: Multiply and Add Polynomial Basis Word to Hi,Lo

To multiply two 32-bit binary polynomial values and polynomial-basis add the result to Hi, Lo.

Description: `(LO,HI,ACX) ← PolyMult(GPR[rs], GPR[rt]) xor (LO,HI,ACX)`

The 32-bit word value in GPR *rs* is polynomial-basis multiplied by the 32-bit value in GPR *rt*, treating both operands as binary polynomial values, to produce a 64-bit result. The product is polynomial-basis added (XORed) to the 64-bit concatenated value of *HI* and *LO*, and the zero-extended result is written back into *HI* and *LO*. Although MADDP is formally defined to operate on special register *ACX* as well, its value can never be changed by the operation, nor can its input value affect the result. No arithmetic exception occurs under any circumstances.

Restrictions:

This instruction does not provide the capability of writing directly to a target GPR.

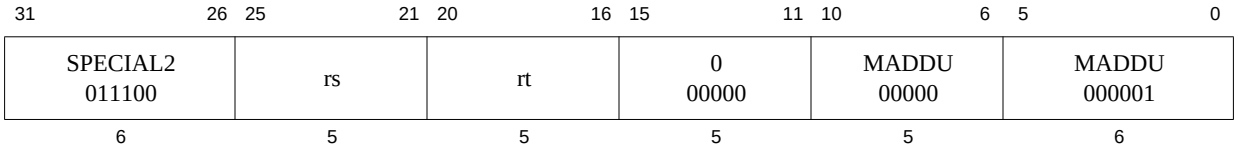
Operation:

```
temp ← (HI31..0 || LO31..0) xor PolyMult(GPR[rs]31..0,GPR[rt]31..0)
HI ← sign_extend(temp63..32)
LO ← sign_extend(temp31..0)
ACX ← ACX
```

Exceptions:

None

.



Format: `MADDU rs, rt` **SmartMIPS Crypto**

Purpose: Multiply and Add Unsigned Word to Hi,Lo

To multiply two unsigned words and add the result to ACX, HI, LO.

Description: `(LO,HI,ACX) ← (GPR[rs] × GPR[rt]) + (LO,HI,ACX)`

The 32-bit word value in GPR *rs* is multiplied by the 32-bit value in GPR *rt*, treating both operands as unsigned values, to produce a 64-bit result. The product is added to the 72-or-more-bit concatenated value of *ACX*, *HI*, and *LO*, and the carry-extended result is written back into *ACX*, *HI*, and *LO*. No arithmetic exception occurs under any circumstances.

Restrictions:

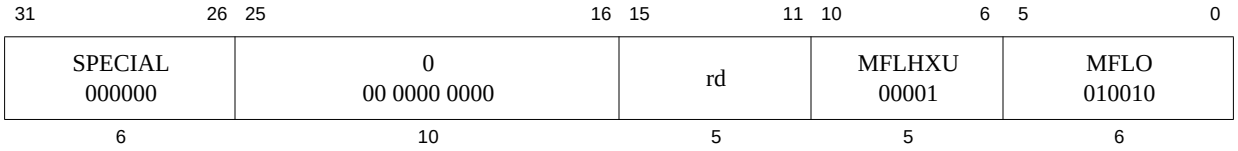
This instruction does not provide the capability of writing directly to a target GPR.

Operation:

```
temp ← (ACX_{ACXMSB..0} || HI_{31..0} || LO_{31..0}) + (GPR[rs]_{31..0} × GPR[rt]_{31..0})
ACX ← zero_extend(temp_{ACXMSB+64..64})
HI ← sign_extend(temp_{63..32})
LO ← sign_extend(temp_{31..0})
```

Exceptions:

None



Format: `MFLHXU rd` **SmartMIPS Crypto**

Purpose: Move from Extended Carry, Hi and Lo (Unsigned)

Extract extended Hi/Lo state.

Description: `GPR[rd] ← LO; LO ← HI; HI ← ACX; ACX ← 0;`

The value in special register *LO* is written to GPR *rd*. The value in special register *HI* is then written to special register *LO*, the extended accumulator bits *ACX* are zero-extended and copied to *HI*, and the extended accumulator bits *ACX* are cleared

- The number of *ACX* extended accumulator bits is implementation dependent, ranging from 8 to 64 bits.
- If 64-bit operations are not available and enabled, at most the least-significant 32 bits of *ACX* will be copied to *HI*, but the whole *ACX* field will then be cleared.

Restrictions:

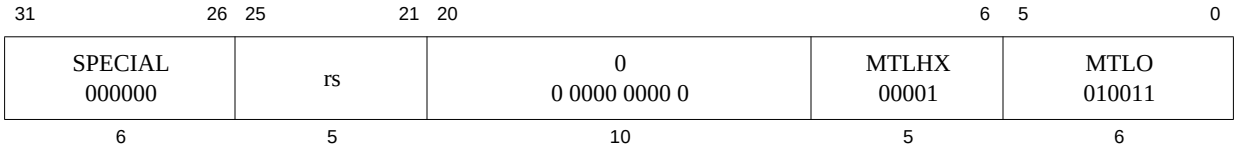
None

Operation:

```
newHI ← zero_extend(ACX)
newLO ← HI
GPR[rd] ← LO
LO ← newLO
HI ← newHI
ACX ← 0
```

Exceptions:

None



Format: `MTLHX rs` **SmartMIPS Crypto**

Purpose: Move to Lo, Hi, and Extended Carry

Set extended Hi/Lo state.

Description: `ACX ← HI; HI ← LO; LO ← GPR[rs] ;`

The value special register *HI* is written to the extended accumulator bits *ACX*. The value in special register *LO* is then written to special register *HI*, and the value in GPR *rs* is written to special register *LO*. This is the reverse of the operation of the MFLHXU instruction.

- The number of *ACX* extended accumulator bits is implementation-dependent, ranging from 8 to 64 bits. If the *HI* register contains more significant bits than the number of implemented *ACX* bits, that information is discarded without raising an exception.
- If 64-bit operations are not enabled, at most the least significant 32 bits of *HI* will be copied to *ACX*.

Restrictions:

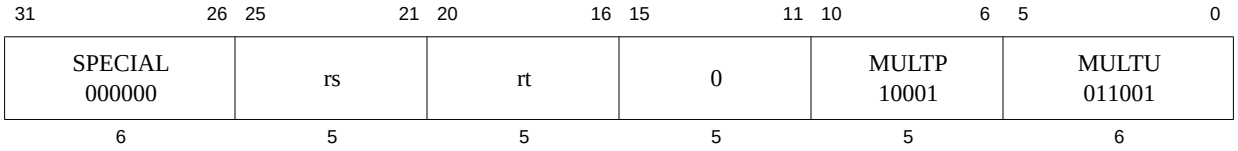
None

Operation:

```
newLO ← GPR[rs]
newHI ← LO
ACX ← HI_ACXMSB..0
HI ← newHI
LO ← newLO
```

Exceptions:

None



Format: `MULTP rs, rt` **SmartMIPS Crypto**

Purpose: Multiply Binary Polynomial Basis Word

To multiply two 32-bit binary polynomial values

Description: `(LO, HI) ← BinPolyMult(GPR[rs], GPR[rt])`

The 32-bit word value in GPR *rt* is polynomial-basis multiplied by the 32-bit value in GPR *rs*, treating both operands as binary polynomial values, to produce a 64-bit result. The low-order 32-bit word of the result is placed into special register *LO*, and the high-order 32-bit word is placed into special register *HI*. The special register *ACX* is cleared.

No arithmetic exception occurs under any circumstances.

Restrictions:

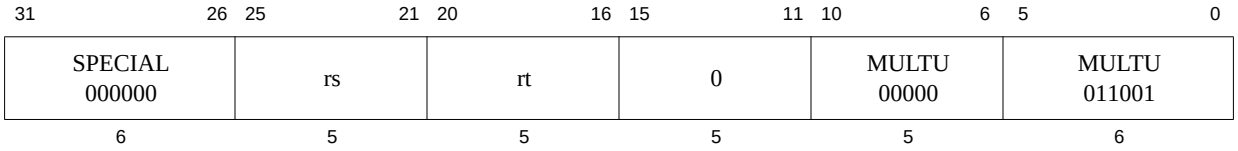
None

Operation:

```
prod ← PolyMult(GPR[rs]31..0, GPR[rt]31..0)
LO ← sign_extend(prod31..0)
HI ← sign_extend(prod63..32)
ACX ← 0
```

Exceptions:

None



Format: `MULTU rs, rt` **SmartMIPS Crypto**

Purpose: Multiply Unsigned Word

To multiply 32-bit unsigned integers

Description: `(LO, HI) ← GPR[rs] × GPR[rt]`

The 32-bit word value in GPR *rt* is multiplied by the 32-bit value in GPR *rs*, treating both operands as unsigned values, to produce a 64-bit result. The low-order 32-bit word of the result is placed into special register *LO*, and the high-order 32-bit word is placed into special register *HI*. The special register *ACX* is cleared.

No arithmetic exception occurs under any circumstances.

Restrictions:

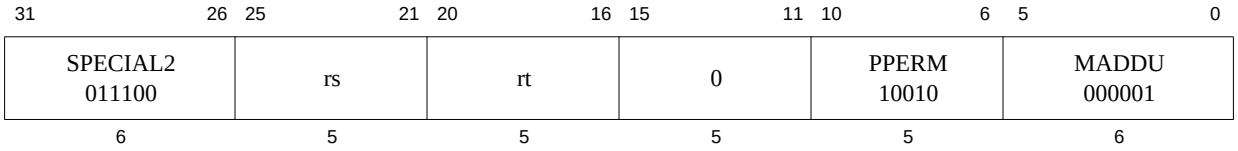
None

Operation:

```
prod ← (0 || GPR[rs]31..0) × (0 || GPR[rt]31..0)
LO ← sign_extend(prod31..0)
HI ← sign_extend(prod63..32)
ACX ← 0
```

Exceptions:

None



Format: `PPERM rs, rt` **SmartMIPS Crypto**

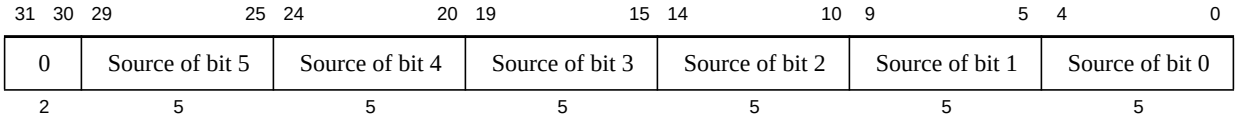
Purpose: Partial Permutation of Word Data into ACX-Hi-Lo Accumulator

Perform a partial permutation of a 32-bit value into the ACX/Hi/Lo registers

Description: `(LO, HI, ACX) ← (LO, HI, ACX) << 6 | GPR[rs]` **bits specified by contents of** `GPR[rt]`

The extended accumulator formed by the *ACX*, *HI*, and *LO* registers is shifted left by six bits, and 32-bit word value in GPR *rt* is used as a permutation descriptor to select a set of six bits from GPR *rs*, to be written into the least significant six bits of the *LO* register.

The contents of the *rt* register are interpreted as follows:



No arithmetic exception occurs under any circumstances.

Restrictions:

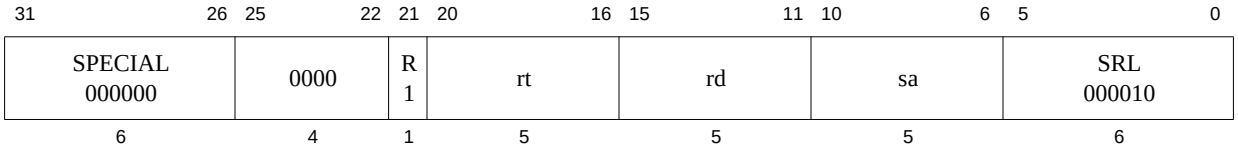
None

Operation:

```
TEMP ← ACX(ACXBITS-6)..0 || HI31..26
ACX ← TEMPACXBITS..0
TEMP ← HI25..0 || LO31..26
HI ← sign_extend(TEMP)
TEMP31...6 ← LO25..0
BITSEL ← GPR[rt]29..25
TEMP5 ← GPR[rs]BITSEL
BITSEL ← GPR[rt]24..20
TEMP4 ← GPR[rs]BITSEL
BITSEL ← GPR[rt]19..15
TEMP3 ← GPR[rs]BITSEL
BITSEL ← GPR[rt]14..10
TEMP2 ← GPR[rs]BITSEL
BITSEL ← GPR[rt]9..5
TEMP1 ← GPR[rs]BITSEL
BITSEL ← GPR[rt]4..0
TEMP0 ← GPR[rs]BITSEL
LO ← sign_extend(TEMP)
```

Exceptions:

None



Format: `ROTR rd, rt, sa` **SmartMIPS Crypto**

Purpose: Rotate Word Right

To execute a logical right-rotate of a word by a fixed number of bits

Description: `GPR[rd] ← GPR[rt] ↔ (right) sa`

The contents of the low-order 32-bit word of GPR *rt* are rotated right; the word result is placed in GPR *rd*. The bit-rotate amount is specified by *sa*.

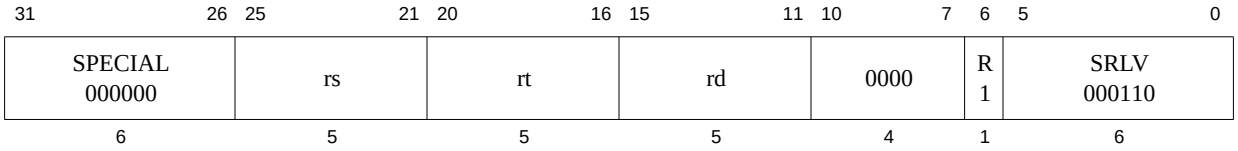
Restrictions:

Operation:

```
if ((ArchitectureRevision() < 2) and (Config3SM = 0)) then
   UNPREDICTABLE
endif
s ← sa
temp ← GPR[rt]s-1..0 || GPR[rt]31..s
GPR[rd] ← temp
```

Exceptions:

Reserved Instruction



Format: `ROTRV rd, rt, rs` **SmartMIPS Crypto**

Purpose: Rotate Word Right Variable

To execute a logical right-rotate of a word by a variable number of bits

Description: `GPR[rd] ← GPR[rt] ↔ (right) GPR[rs]`

The contents of the low-order 32-bit word of GPR *rt* are rotated right; the word result is placed in GPR *rd*. The bit-rotate amount is specified by the low-order 5 bits of GPR *rs*.

Restrictions:

Operation:

```
if ((ArchitectureRevision() < 2) and (Config3SM = 0)) then
   UNPREDICTABLE
endif
s ← GPR[rs]4..0
temp ← GPR[rt]s-1..0 || GPR[rt]31..s
GPR[rd] ← temp
```

Exceptions:

Reserved Instruction

Chapter 6

The SmartMIPS® Privileged Resource Architecture

6.1 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. SmartMIPS defines extensions to the MIPS32 PRA that are desirable in a smart card 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.

6.2 Interaction between the SmartMIPS ASE and Release 2 of the MIPS32 Architecture

Some features that are included in the SmartMIPS ASE (e.g., the ROTR and ROTRV) were subsequently added to Release 2 of the MIPS32 Architecture. In such cases, the features are implemented identically. Some features that are included in the SmartMIPS ASE (e.g., 1KB page support) were implemented in Release 2 of the MIPS32 Architecture in such a way that there is conflict between the specifications. In such a case, the conflict is resolved in favor of the SmartMIPS ASE specification. That is, an implementation of the SmartMIPS ASE in a processor that also implements Release 2 of the MIPS32 Architecture obeys the rules of the SmartMIPS ASE whenever the specifications have a conflict.

6.3 Overview

The SmartMIPS PRA extends MIPS32 in three specific regards:

- Protection of Virtual Memory Pages
- Virtual Memory Page Size
- Detection of SmartMIPS Features

The standard MIPS32 PRA provides for Virtual memory pages to be invalid, readable and executable, or readable, executable, and writable. SmartMIPS extends this to allow true read-only, execute-only, and write-only pages.

The minimum virtual memory page size supported by the MIPS32 PRA is 4K (4096) bytes. SmartMIPS allows for the TLB to be configured to optimally support 4K, 2K, and 1K virtual memory pages, and to accelerate lookups of multilevel page tables.

The presence of SmartMIPS features is indicated in the CP0 *Config3* register.

6.4 Compliance

Features described as *Required* in this document are required of all processors claiming compatibility with SmartMIPS. Any features described as *Recommended* should be implemented unless there is an overriding need not to do so. Features described as *Optional* provide a standardization of features that may or may not be appropriate for a particular SmartMIPS processor implementation. If such a feature is implemented, it must be implemented as described in this document if a processor is to claim compatibility with SmartMIPS.

In some cases, there are features within features that have different levels of compliance. For example, if there is an *Optional* field within a *Required* register, this means that the register must be implemented, but the field may or may not be, depending on the needs of the implementation. Similarly, if there is a *Required* field within an *Optional* register, this means that if the register is implemented, it must have the specified field.

6.5 The SmartMIPS System Coprocessor

Except as defined below, the SmartMIPS system coprocessor interface and functionality is identical to MIPS32.

6.5.1 CP0 Register Summary

Table 6.1 lists the CP0 registers affected by the SmartMIPS 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 6.1 SmartMIPS Changes to Coprocessor 0 Registers in Numerical Order

Register Number	Sel	Register Name	Modification	Reference	Compliance Level
2	0	*EntryLo0*	Two additional bits per page to provide greater variety of access modes. PFN field definition modified by PageGrain register.	Section 6.7.1	Required
3	0	*EntryLo1*	Two additional bits per page to provide greater variety of access modes. PFN field definition modified by PageGrain register.	Section 6.7.1	Required
4	0	*Context*	Layout controlled by ContextConfig	Section 6.7.2	Required
4	1	*ContextConfig*	New Register. Controls layout of Context register	Section 6.7.3	Required
5	0	*PageMask*	Qualified by PageGrain register	Section 6.7.4	Required
5	1	*PageGrain*	New register. Controls granularity of virtual pages in EntryLo, PageMask, and EntryHi registers	Section 6.7.5	Required
10	0	*EntryHi*	Qualified by PageGrain register	Section 6.7.6	Required
16	2	*Config3*	Identifies SmartMIPS feature set.	Section 6.7.7	Required

6.6 Virtual Memory

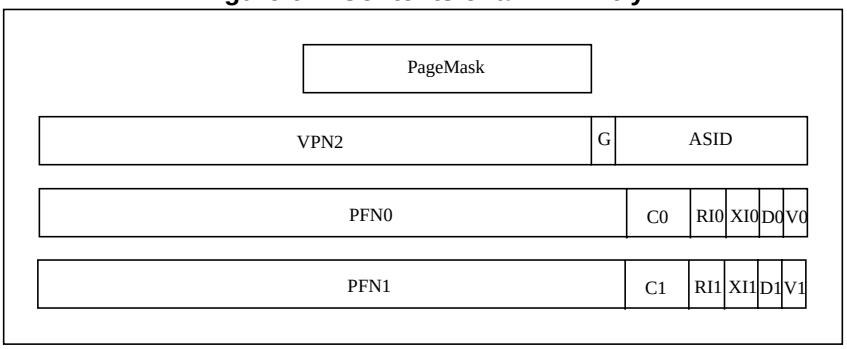
6.6.1 TLB-Based Virtual Address Translation

This section describes the SmartMIPS changes and additions to the MIPS32 TLB-based virtual address translation mechanism.

6.6.1.1 TLB Organization

SmartMIPS extends the TLB organization defined by the MIPS32 architecture. The size of the field containing the virtual page number in the comparison section must accommodate a wider range of virtual page sizes. In all profiles, the translation section is augmented by *two additional bits, RI (Read Inhibit) and XI (Execute Inhibit)*, which can be thought of as qualifiers for the existing V (Valid) bit. There are still two entries in the translation section for each TLB entry because each TLB entry maps an aligned pair of virtual pages and the pair of physical translation entries corresponds to the even and odd pages of the pair. Figure 6.1 shows the logical arrangement of a TLB entry.

Figure 6.1 Contents of a TLB Entry



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

6.6.1.2 Address Translation

The address translation process in SmartMIPS varies from the standard MIPS32 and MIPS64 address translation process in two specific regards:

- The number and position of the bits that form the virtual page number, physical page frame number, and page mask may vary from MIPS32 and provide 4K, 2K or 1K page granularity, depending on the state of the PageGrain register.

- The RI and XI bits serve to inhibit the V (Valid) bit for Read and Instruction Fetch accesses respectively. Fetching instructions from a page with the XI bit set will generate a TLB Invalid exception even if the V bit is set. Similarly, attempting to load data from a page with the RI bit set will generate a TLB Invalid exception even if the V bit is set. The set of the RI, XI, D, and V bits allows for any combination of read/write/execute protections to be enforced by hardware.

The modified TLB lookup process can be described as follows:

```
found ← 0
for i in 0...TLBEntries-1
          if((TLB[i]VPN2 and not (TLB[i]Mask)) = (va31..11 and not (TLB[i]Mask))) and
           (TLB[i]G or (TLB[i]ASID = EntryHiASID)) then
             # EvenOddBit selects between even and odd halves of the TLB as
             # a function of the page size in the matching TLB entry
             effective_mask = TLB[i]Mask OR (0 || PageGrainMask)
             case effective_mask
                 000000000000002: EvenOddBit ← 10
                 000000000000012: EvenOddBit ← 11
                 000000000000112: EvenOddBit ← 12
                 000000000011112: EvenOddBit ← 14
                 000000001111112: EvenOddBit ← 16
```

```
                    00000011111111₂: EvenOddBit ← 18
                    00001111111111₂: EvenOddBit ← 20
                    00111111111111₂: EvenOddBit ← 22
                    11111111111111₂: EvenOddBit ← 24
                    otherwise: UNDEFINED
                endcase
                if vaEvenOddBit = 0 then
                    pfn ← TLB[i]PFN0
                    v ← TLB[i]V0
                    c ← TLB[i]C0
                    d ← TLB[i]D0
                    ri ← TLB[i]RI0
                    xi ← TLB[i]XI0
                else
                    pfn ← TLB[i]PFN1
                    v ← TLB[i]V1
                    c ← TLB[i]C1
                    d ← TLB[i]D1
                    ri ← TLB[i]RI1
                    xi ← TLB[i]XI1
                endif
                if v = 0 then
                    SignalException(TLBInvalid, reftype)
                endif
                if (ri = 1) and (reftype = load) then
                    if (xi = 0) and (IsPCRelativeLoad(PC))
                        # PC relative loads are allowed where execute is allowed
                    else
                        SignalException(TLBInvalid, reftype)
                    endif
                endif
                if (xi = 1) and (reftype = fetch) then
                    SignalException(TLBInvalid, reftype)
                endif
                if (d = 0) and (reftype = store) then
                    SignalException(TLBModified)
                endif
                case PageGrainMask
                    00₂: pa_pfn  ← 00₂ || pfn
                    01₂: pa_pfn  ← 0₂ || pfn || 0₂
                    11₂: pa_pfn  ← pfn || 00₂
                endcase
                # pa_pfn(PABITS-1)-10..0 corresponds to paPABITS-1..10
                pa ← pa_pfn(PABITS-1)-10..EvenOddBit-10 || vaEvenOddBit-1..0
                found ← 1
                break
            endif
    endfor
    if found = 0 then
            SignalException(TLBMiss, reftype)
    endif
```

Table 6.2 shows how the physical address is generated as a function of the page size of the TLB entry matching the virtual address. The “Even/Odd Select” column of the table indicates which virtual address bit is used to select between the even (*EntryLo0*) or odd (*EntryLo1*) entry in the matching TLB entry. The “PA generated from” column specifies how the physical address is generated from the selected PFN and the offset-in-page bits in the virtual address. PFN is the physical page number as loaded into the TLB from the *EntryLo0* or *EntryLo1* registers, and has

the bit range $PFN_{(PABITS\text{-}1)\text{-}12..0}$, corresponding to $PA_{PABITS\text{-}1..12}$, $PA_{PABITS\text{-}2..11}$, or $PA_{PABITS\text{-}3..10}$, depending on the value of *PageGrain*$_{Mask}$. Note that there are multiple combinations of *PageMask* and *PageGrain* that result in the same effective virtual page size.

Table 6.2 Physical Address Generation

Page Size	Even/Odd Select	PageGrain Mask value	$PA_{(PABITS-1)..0}$ generated from
1K Bytes	VA_{10}	00_2	$00_2 \| PFN_{(PABITS-1)-12..0} \| VA_{09..0}$
2K Bytes	VA_{11}	00_2	$00_2 \| PFN_{(PABITS-1)-12..1} \| VA_{10..0}$
		01_2	$0_2 \| PFN_{(PABITS-1)-12..0} \| VA_{10..0}$
4K Bytes	VA_{12}	00_2	$00_2 \| PFN_{(PABITS-1)-12..2} \| VA_{11..0}$
		01_2	$0_2 \| PFN_{(PABITS-1)-12..1} \| VA_{11..0}$
		11_2	$PFN_{(PABITS-1)-12..0} \| VA_{11..0}$
16K Bytes	VA_{14}	00_2	$00_2 \| PFN_{(PABITS-1)-12..4} \| VA_{13..0}$
		01_2	$0_2 \| PFN_{(PABITS-1)-12..3} \| VA_{13..0}$
		11_2	$PFN_{(PABITS-1)-12..2} \| VA_{13..0}$
64K Bytes	VA_{16}	00_2	$00_2 \| PFN_{(PABITS-1)-12..6} \| VA_{15..0}$
		01_2	$0_2 \| PFN_{(PABITS-1)-12..5} \| VA_{15..0}$
		11_2	$PFN_{(PABITS-1)-12..4} \| VA_{15..0}$
256K Bytes	VA_{18}	00_2	$00_2 \| PFN_{(PABITS-1)-12..8} \| VA_{17..0}$
		01_2	$0_2 \| PFN_{(PABITS-1)-12..7} \| VA_{17..0}$
		11_2	$PFN_{(PABITS-1)-12..6} \| VA_{17..0}$
1M Bytes	VA_{20}	00_2	$00_2 \| PFN_{(PABITS-1)-12..10} \| VA_{19..0}$
		01_2	$0_2 \| PFN_{(PABITS-1)-12..9} \| VA_{19..0}$
		11_2	$PFN_{(PABITS-1)-12..8} \| VA_{19..0}$
4M Bytes	VA_{22}	00_2	$00_2 \| PFN_{(PABITS-1)-12..12} \| VA_{21..0}$
		01_2	$0_2 \| PFN_{(PABITS-1)-12..11} \| VA_{21..0}$
		11_2	$PFN_{(PABITS-1)-12..10} \| VA_{21..0}$
16M Bytes	VA_{24}	11_2	$PFN_{(PABITS-1)-12..12} \| VA_{23..0}$

6.6.2 General Exception Processing

SmartMIPS optimizations for security and for dealing with small memories and small pages require modifications to the definitions of the TLB Refill, TLB Invalid, and TLB Modified exceptions, where new qualifying conditions exist for virtual pages to be considered to be valid, and where the setup of the *Context* register is now configurable by the *ContextConfig* register, and the *EntryHi* contents varies according to the page granularity specified by the *PageGrain* register.

6.6.3 TLB Refill Exception

As in MIPS32, a TLB refill exception occurs in a TLB-based MMU when no TLB entry matches a reference to a mapped address space and the *EXL* bit is zero in the *Status* register. SmartMIPS CPUs can differ from MIPS32 in the information provided on a TLB Refill exception in the *Context* and *EntryHi* registers, depending on the *Context* register configuration and the page granularity supported.

Table 6.3 TLB Refill Exception State Saved in Addition to the Cause Register

Register State	Value
BadVAddr	Failing address
Context	The bits of the *Context* register corresponding to the set bits of the *VirtualIndex* field of the *ContextConfig* register are loaded with the high-order bits of the virtual address that missed.
EntryHi	Bits 31:13 contain $VA_{31:13}$ of the failing address. Bits 12:11 contain $VA_{12:11}$ ANDed with the compliment (logical negation) of $PageGrain_{Mask}$; ASID field contains ASID of the reference that missed.
EntryLo0	UNPREDICTABLE
EntryLo1	UNPREDICTABLE

6.6.4 TLB Invalid Exception

As in MIPS32, a TLB invalid exception occurs when a TLB entry matches a reference to a mapped address space, but the matched entry has the V (valid) bit off. On a SmartMIPS CPU, however, if a valid, matching TLB entry is found with the RI (Read Inhibit) set on a read reference, or with XI (Execute Inhibit) set on an instruction fetch reference, a TLB Invalid exception will occur despite the presence of the valid bit. MIPS16 PC-relative loads are a special case, and are not affected by the RI bit. In addition, SmartMIPS can differ from MIPS32 in the information provided on a TLB Invalid exception in the *Context* and *EntryHi* registers.

Table 6.4 TLB Invalid Exception State Saved in Addition to the Cause Register

Register State	Value
BadVAddr	Failing address
Context	The bits of the *Context* register corresponding to the set bits of the *VirtualIndex* field of the *ContextConfig* register are loaded with the high-order bits of the invalid virtual address.
EntryHi	Bits 31:13 contain $VA_{31:13}$ of the failing address. Bits 12:11 contain $VA_{12:11}$ ANDed with the compliment (logical negation) of $PageGrain_{Mask}$; ASID field contains ASID of the invalid reference.
EntryLo0	UNPREDICTABLE
EntryLo1	UNPREDICTABLE

6.6.5 TLB Modified Exception

As in MIPS32, TLB modified exception occurs on a *store* reference to a mapped address when the matching TLB entry is valid, but the entry's D bit is zero, indicating that the page is not writable. SmartMIPS CPUs can differ from MIPS32 in the information provided on a TLB Refill exception in the *Context* and *EntryHi* registers.

Table 6.5 TLB Modified Exception State Saved in Addition to the Cause Register

Register State	Value
BadVAddr	Failing address

Table 6.5 TLB Modified Exception State Saved in Addition to the Cause Register

Register State	Value
Context	The bits of the *Context* register corresponding to the set bits of the *VirtualIndex* field of the *ContextConfig* register are loaded with the high-order bits of the virtual address being written.
EntryHi	Bits 31:13 contain $VA_{31:13}$ of the failing address. Bits 12:11 contain $VA_{12:11}$ ANDed with the compliment (logical negation) of $PageGrain_{Mask}$; ASID field contains ASID of the modifying reference.
EntryLo0	UNPREDICTABLE
EntryLo1	UNPREDICTABLE

6.7 CP0 Registers

The CP0 registers provide the interface between the ISA and the PRA. Those CP0 registers that are extended or redefined for SmartMIPS 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.

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

Compliance Level: *EntryLo0* modifications are *Required* for a SmartMIPS MMU.

Compliance Level: *EntryLo1* modifications are *Required* for a SmartMIPS MMU.

As in MIPS32, the pair of *EntryLo* registers act as the interface between the TLB and the TLBR, TLBWI, and TLBWR instructions. *EntryLo0* holds the entries for even pages and EntryLo1 holds the entries for odd pages.

In a SmartMIPS MMU, the previously reserved bits 31 and 30 are defined for further access control.

The interpretation, though not the size or location, of the PFN field in a SmartMIPS MMU varies with the content of the PageGrain register.

Figure 6.2 shows the format of the *EntryLo0* and *EntryLo1* registers; Table 6.6 describes the *EntryLo0* and *EntryLo1* register fields.

Figure 6.2 SmartMIPS EntryLo0, EntryLo1 Register Format

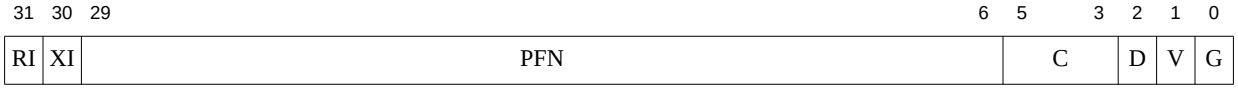


Table 6.6 SmartMIPS EntryLo0, EntryLo1 Register Field Descriptions

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
RI	31	Read Inhibit. If this bit is set in a TLB entry, an attempt, **other than a MIPS16 PC-relative load,** to read data on the virtual page causes a TLB Invalid exception, even if the *V* (Valid) bit is set. The *RI* bit is writable only if the *RIE* bit of the *PageGrain* register is set. If the *RIE* bit of *PageGrain* is not set, the *RI* bit of *EntryLo0*/*EntryLo1* is set to zero on any write to the register, regardless of the value written.	R/W	0	Required

Table 6.6 SmartMIPS EntryLo0, EntryLo1 Register Field Descriptions

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
XI	30	Execute Inhibit. If this bit is set in a TLB entry, an attempt to fetch an instruction or to load MIPS16 PC-relative data from the virtual page causes a TLB Invalid exception, even if the *V* (Valid) bit is set. The *XI* bit is writable only if the *XIE* bit of the *PageGrain* register is set. If the *XIE* bit of *PageGrain* is not set, the *XI* bit of *EntryLo0*/*EntryLo1* is set to zero on any write to the register, regardless of the value written.	R/W	0	Required
PFN	29:6	Up to 24 bits of the physical address associated with the page. The binding of *PFN* bits to physical address bits depends on the value of the *Mask* field of the *PageGrain* register: $PageGrain_{Mask}$= 0 0: PFN corresponds to $PA_{PABITS-3..10}$ $PageGrain_{Mask}$= 0 1: PFN corresponds to $PA_{PABITS-2..11}$ $PageGrain_{Mask}$= 1 1: PFN corresponds to $PA_{PABITS-1..12}$ $PageGrain_{Mask}$= 1 0: **UNDEFINED** The width of this field implicitly limits the range of physical addresses to 36 bits, 35 bits, or 34 bits for minimum page sizes of 4K, 2K, and 1K bytes respectively. If the processor implements fewer physical address bits than this limit, the unimplemented bits must be written as zero, and return zero on read. If the processor implements more physical address bits than are defined by *PFN*, given a non-zero value of $PageGrain_{Mask}$, the bits to the left of the MSB of *PFN* are zero in the generated physical address.	R/W	Undefined	Required
C	5:3	Coherency attribute. Unchanged from MIPS32.			
D	2	"Dirty" bit. Unchanged from MIPS32.			
V	1	Valid bit, indicating that the TLB entry, and thus the virtual page mapping are valid. If this bit is a one, accesses to the page are permitted. If this bit is a zero, accesses to the page cause a TLB Invalid exception. In SmartMIPS, this bit is further qualified by the *RI* and *XI* bits.	R/W	Undefined	Required
G	0	Global bit. Unchanged from MIPS32.			

6.7.2 Context Register (CP0 Register 4, Select 0)

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

In SmartMIPS, the *Context* register is a read/write register containing a pointer to an arbitrary power-of-two aligned data structure in memory, such as an entry in the page table entry (PTE) array. Unlike MIPS32, where this pointer was defined to reference a 16-byte structure in memory within a linear array containing an entry for each even/odd virtual page pair, the SmartMIPS *Context* register can be used far more generally. Depending on the value in the *ContextConfig* register, it may point to an 8-byte pair of 32-bit PTEs within a single-level page table scheme, or to a first level page directory entry in a two-level lookup scheme.

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

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

Figure 6.3 shows the format of the *Context* Register; Table 6.7 describes the *Context* register fields.

Figure 6.3 SmartMIPS Context Register Format

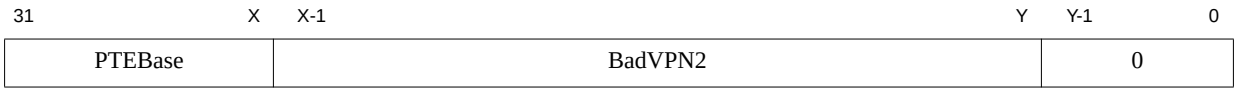


Table 6.7 SmartMIPS Context Register Field Descriptions

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
PTEBase	Variable, 31:X where X in {31..0}. May be null.	This field is for use by the operating system and is normally written with a value that allows the operating system to use the *Context* Register as a pointer to an array of data structures in memory corresponding to the address region containing the virtual address which caused the exception.	R/W	Undefined	Required
BadVPN2	Variable, (X-1):Y where X in {32..1} and Y in {31..0}. May be null.	This field is written by hardware on a TLB exception. It contains bits $VA_{31:31-((X-Y)-1)}$ of the virtual address that caused the exception.	R	Undefined	Required
0	Variable, (Y-1):0 where Y in {31:1}. May be null.	Must be written as zero; returns zero on read.	0	0	Reserved

6.7.3 ContextConfig Register (CP0 Register 4, Select 1)

Compliance Level: *Required* for a SmartMIPS MMU.

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

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

A value of all ones in the *ContextConfig* register means that the full 32 bits of the faulting virtual address will be copied into the context register, making it duplicate the *BadVAddr* register. A value of all zeroes means that the full 32 bits of the *Context* register are R/W for software and unaffected by TLB exceptions.

Figure 6.4 shows the SmartMIPS formats of the *ContextConfig* Register; Table 6.8 describes the *ContextConfig* register fields.

Figure 6.4 SmartMIPS ContextConfig Register Format

Table 6.8 SmartMIPS ContextConfig Register Field Descriptions

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

It is permissible to implement a subset of the *ContextConfig* register, in which some number of bits are read-only and set to one or zero as appropriate. It is possible for software to determine which bits are implemented by alternately writing all zeroes and all ones to the register, and reading back the resulting values. All implementations of the *ContextConfig* register must allow for the emulation of the MIPS32 fixed *Context* register configuration. Table 6.9 describes some useful *ContextConfig* values.

Table 6.9 Recommended ContextConfig Values for SmartMIPS

Value	Page Table Organization	Page Size	PTE Size	Compliance
0x007ffff0	Single Level	4K	64 bits/page	REQUIRED
0x003ffff8	Single Level	4K	32 bits/page	RECOMMENDED
0x007ffff8	Single Level	2K	32 bits/page	RECOMMENDED
0x00fffff8	Single Level	1K	32 bits/page	RECOMMENDED

6.7.4 PageMask Register (CP0 Register 5, Select 0)

Compliance Level: *PageMask* register modifications are *Required* for SmartMIPS MMUs.

As in MIPS32, the *PageMask* register is a read/write register used for reading from and writing to the TLB. SmartMIPS allows implementation of page sizes smaller than 4K bytes, and the *PageMask* register must be extended to accommodate them, as shown in Table 6.11. To assure backward compatibility with MIPS32, the *Mask* field extension bits 12 ad 11 can be inhibited and overridden by the corresponding bits of the *PageGrain* register. Inhibited *PageMask* bits are treated as 1 bits for the purposes of virtual address translation - the corresponding virtual address bits are not used for TLB match comparisons - but read as zeroes to software to preserve backward compatibility.

Figure 6.5 shows the format of the *PageMask* register; Table 6.10 describes the *PageMask* register fields.

Figure 6.5 SmartMIPS PageMask Register Format

Table 6.10 PageMask Register Field Descriptions

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
Mask	24:11	The *Mask* field is a bit mask in which a "1" bit indicates that the corresponding bit of the virtual address should not participate in the TLB match. Bits 12 and 11 of the *Mask* field can be overridden by the *Mask* field of the *PageGrain* register: If set in the *PageGrain* register, the corresponding bit in the *PageMask* register is unwritable and reads as a zero to software, but the corresponding bit is excluded from address comparison as if it were set in the *PageMask Mask* field.	R/W	0 for bits 12..1; Undefined for bits 24..13	Required
0	31:25, 10:0	Must be written as zero; return zero on read.	0	0	Reserved

Table 6.11 Values for the Mask Field of the PageMask Register

Page Size	Bit													
	24	23	22	21	20	19	18	17	16	15	14	13	12*	11*
1 KBytes	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2 KBytes	0	0	0	0	0	0	0	0	0	0	0	0	0	1
4 KBytes	0	0	0	0	0	0	0	0	0	0	0	0	1	1
16 KBytes	0	0	0	0	0	0	0	0	0	0	1	1	1	1
64 KBytes	0	0	0	0	0	0	0	0	1	1	1	1	1	1
256 KBytes	0	0	0	0	0	0	1	1	1	1	1	1	1	1
1 MByte	0	0	0	0	1	1	1	1	1	1	1	1	1	1
4 MByte	0	0	1	1	1	1	1	1	1	1	1	1	1	1
16 Mbyte	1	1	1	1	1	1	1	1	1	1	1	1	1	1

The columns marked with an asterix (*) are those which can be disabled and overridden by the *PageGrain* register.

It is implementation-dependent how many of the encodings described in Table 6.11 are implemented. All processors must implement the 4KB page size, and the implemented Mask bits must span the contiguous range of values from 4KB to the smallest page granularity that can be specified by the implemented *PageGrain* register. If a particular page size encoding is not implemented by a processor, a read of the *PageMask* register must return zeros in all bits that correspond to encodings that are not implemented. Software can determine which page sizes are supported by writing the encoding for a 16MB page to the *PageMask* register, then examine the value returned from a read of the

PageMask register. If a pair of bits reads back as ones, the processor implements that page size. The operation of the processor is **UNDEFINED** if software loads the *PageMask* register with a value other than one of those listed in Table 6.11.

The value of the *PageMask* register is **UNPREDICTABLE** following a modification of the contents of the *PageGrain* register.

6.7.5 PageGrain Register (CP0 Register 5, Select 1)

Compliance Level: *Required* for SmartMIPS MMUs.

The *PageGrain* register is a read/write register used to configure the SmartMIPS MMU to operate on pages smaller than 4K bytes. It's value is used when reading from and writing to the TLB. SmartMIPS allows implementation of page sizes smaller than 4K bytes, and in those implementations, the *PageMask* register must be extended to accommodate them, as shown in Table 6.11. The *PageGrain* register also contains enable bits for the read-inhibit (*RI*) and execute-inhibit (*XI*) bits of the *EntryLo* registers.

It is not required that the contents of the *PageGrain* register be reflected in the contents of the TLB. Therefore, the TLB must be flushed before any change to the *PageGrain* register is made. The operation of the processor is **UNDEFINED** if software modifies any field of the *PageGrain* register while valid entries are present in the TLB.

Figure 6.6 shows the format of the *PageGrain* register; Table 6.10 describes the *PageGrain* register fields.

Figure 6.6 SmartMIPS PageGrain Register Format

Table 6.12 SmartMIPS PageGrain Register Field Descriptions

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
RIE	31	Read Inhibit Enable. If this bit is set, the *RI* bit of the *EntryLo0* and *EntryLo1* registers is enabled. If the bit is clear, the *RI* bit is disabled and not writable by software. See section **6.7.1**.	R/W	0	Required
XIE	30	Execute Inhibit Enable. If this bit is set, the *XI* bit of the *EntryLo0* and *EntryLo1* registers is enabled. If the bit is clear, the *XI* bit is disabled and not writable by software. See section **6.7.1**.	R/W	0	Required

Table 6.12 SmartMIPS PageGrain Register Field Descriptions

Fields		Description		Read / Write	Reset State	Compliance
Name	Bits					
Mask	12:11	Determines whether the corresponding bits of a virtual address are to be used for address translation purposes. This affects the behavior of the *EntryLo0*/*EntryLo1*, *EntryHi*, and *PageMask* registers according to the following scheme:		R/W	1 1	Required
		0 0	Bits 12 and 11 of *PageMask* and *EntryHi* are R/W to software and used in address translation. *PFN* field of *EntryLo0*/*EntryLo1* is treated as $PA_{PABITS-3..10}$			
		0 1	Bit 12 of *PageMask* and *EntryHi* is R/W to software and used in address translation. Bit 11 of *PageMask* and *EntryHi* reads as zero, and is not used in address translation. PFN field of *EntryLo0*/*EntryLo1* is treated as $PA_{PABITS-2..11}$			
		1 1	Bits 12 and 11 of *PageMask* and *EntryHi* read as zero, and are not used in address translation. *PFN* field of *EntryLo0*/*EntryLo1* is treated as $PA_{PABITS-1..12}$. In this setting, virtual address translation is identical to that of MIPS32.			
		1 0	UNDEFINED			
1	10:8	Reserved bits for future *Mask* expansion. Must be written as one, return one on read.				
0	29:13, 7:0	Must be written as zero; return zero on read.		0	0	Reserved

It is not required that all bits of the *PageGrain* Mask field be fully implemented. Unimplemented low-order bits must be read-only, and must read and function as having a value of 1. Unimplemented high-order bits must read and function as having the same value as the highest-order implemented bit. Table 6.13 shows some the read/write and functional behavior of the possible SmartMIPS *PageGrain* subsets.

Table 6.13 PageGrain Implementation Subset Behavior

Subset	Mask Value Written	Mask Value Read Back	Page Granularity
2K Byte Page Grain but not 1K Byte Page Grain	0 0	0 1	2K Bytes
	0 1	0 1	2K Bytes
	1 0	1 1	4K Bytes
	1 1	1 1	4K Bytes
1K Byte Page Grain but not 2K Byte Page Grain	0 0	0 0	1K Bytes
	0 1	1 1	4K Bytes
	1 0	0 0	1K Bytes
	1 1	1 1	4K Bytes

6.7.6 EntryHi Register (CP0 Register 10, Select 0

Compliance Level: *EntryHi* register modifications are *Required* for SmartMIPS MMUs.

As in MIPS32, the *EntryHi* register contains the virtual address match information used for TLB read, write, and access operations.

For SmartMIPS implementations supporting pages sizes smaller than 4K, the *VPN2* field of *EntryHi* must be extended to allow for the greater number of VPNs in an address space divided into smaller pages.

Figure 6.7 shows the format of the modified SmartMIPS *EntryHi* register; Table 6.14 describes the *EntryHi* register fields.

Figure 6.7 SmartMIPS EntryHi Register Format

Table 6.14 EntryHi Register Field Descriptions

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
VPN2	31:11	$VA_{31:11}$ of the virtual address (virtual page number / 2). This field is written by hardware on a TLB exception or on a TLB read, and is written by software before a TLB write. Bits 12 and 11 can take a non-zero value only if the corresponding bits of the *PageGrain* register are zeroes.	R/W	0 for bits 12..1; Undefined for bits 31..13	Required
ASID	7:0	*ASID*. Unchanged from MIPS32	R/W	Undefined	Required
0	10:8	Must be written as zero; returns zero on read.	0	0	Reserved

The value of the *EntryHi* register is **UNPREDICTABLE** following a modification of the contents of the *PageGrain* register.

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

Compliance Level: *Required* for SmartMIPS.

The *Config3* register is fully defined in Volume III of this multi-volume set. Bit 1 (named *SM*) of the *Config3* register denotes the presence of the SmartMIPS ASE.

Appendix A

Revision History

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

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

Version	Date	Comments
0.10	27 September 2000	Initial review draft.
0.20	27 October 2000	PRA section added, MIPS16 instructions migrated to Volume IV of MIPS Architecture specification. SmartMIPS no longer referred to as an “ASE” in the text.
0.90	1 November 2000	Conversion to new specification format. First external review draft.
0.91	December 15, 2000	Changes in this revision: • Correct temp variable indexing in the pseudo code for MADDU. • Features listed in the Config2 register should have been included in the Config3 register instead.
0.92	December 21, 2000	Changes in this revision • Make effects of MULTP and MADDP on ACX explicit and mandatory. • Reduce size of reserved 1’s field in PageGrain register to align with EntryHi ASID field as intended. • Clarification of “binary polynomial basis” nomenclature.
0.93	January 30, 2001	Elimination of references to a “MIPS Crypto” ASE as a subset of SmartMIPS. Clarify UNPREDICTABLE state of PageMask and EntryHi registers following modification of PageGrain register.
0.94	February 28, 2001	Addition of RIE and XIE bits to PageGrain register as enables for the RI and XI bits in the EntryLo registers, to enhance backward compatibility.
0.95	March 12, 2001	Update for next external review revision.
1.00	August 1, 2001	Update based on all fieldback received from external distribution.
1.01	April 23, 2002	Create and release an External Confidential version of the Architecture for Programmers manual.
1.20	August 29, 2002	Add bit encoding tables to describe the SmartMIPS insructions.
2.00	May 15, 2003	Changes in this revision: • Add base architecture requirements, software detection of the ASE, and compliance and subsetting sections to the introduction. • Update the ROTR and ROTRV instruction encoding to specify a 1-bit difference between shift and rotate, and to align with the Release 2 instruction descriptions. • Note that conflicts between the SmartMIPS ASE specification and Release 2 of the MIPS32 Architecture are resolved in favor of the SmartMIPS ASE. • Remove the register description of the Config3 register, as this is now fully described in Volume III.

Version	Date	Comments
2.50	July 1, 2005	Changes in this revision: • Update all files to FrameMaker 7.1. • Replace the use of the BinPolyMult function in MULTP with PolyMult, and define that function as pseudo code. • Change reset state of bits 12..11 of the *PageMask* and *EntryHi* registers to 0 from Undefined. This is simply a clarification because the reset state of *PageGrain* forces those bits of *PageMask* and *EntryHi* to be 0.
2.51	July 15, 2008	Changes in this revision: • Update copyrights. • Update contact information.