MIPS® Architecture Extension: nanoMIPS32® Multithreading Technical Reference Manual

Revision 1.17
April 27, 2018
Public

Table of Contents

List of Figures

List of Tables

Chapter 1

About This Book

This chapter describes the terminology and conventions for describing features of the MIPS® Architecture such as instructions and control and status registers.

1.1 Typographical Conventions

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

1.1.1 Italic Text

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

1.1.2 Bold Text

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

1.1.3 Courier Text

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

1.2 UNPREDICTABLE and UNDEFINED

The terms **UNPREDICTABLE** and **UNDEFINED** are used throughout this book to describe the behavior of the processor in certain cases. **UNDEFINED** behavior or operations can occur only as the result of executing instructions in

a privileged mode (i.e., in Kernel Mode or Debug Mode, or with the CP0 usable bit set in the Status register). Unprivileged software can never cause **UNDEFINED** behavior or operations. Conversely, both privileged and unprivileged software can cause **UNPREDICTABLE** results or operations.

1.2.1 UNPREDICTABLE

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

UNPREDICTABLE results or operations have several implementation restrictions:

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

1.2.2 UNDEFINED

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

UNDEFINED operations or behavior has one implementation restriction:

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

1.2.3 UNSTABLE

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

UNSTABLE values have one implementation restriction:

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

1.3 Special Symbols in Pseudocode Notation

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

Table 1.1 Symbols Used in Instruction Operation Statements

Symbol	Meaning
♦	Assignment
=, ≠	Tests for equality and inequality
||	Bit string concatenation
x^y	A *y*-bit string formed by *y* copies of the single-bit value *x*
b#n	A constant value *n* in base *b*. For instance 10#100 represents the decimal value 100, 2#100 represents the binary value 100 (decimal 4), and 16#100 represents the hexadecimal value 100 (decimal 256). If the "b#" prefix is omitted, the default base is 10.
0bn	A constant value *n* in base *2*. For instance 0b100 represents the binary value 100 (decimal 4).
0xn	A constant value *n* in base *16*. For instance 0x100 represents the hexadecimal value 100 (decimal 256).
$x_{y\ z}$	Selection of bits *y* through *z* of bit string *x*. Little-endian bit notation (rightmost bit is 0) is used. If *y* is less than *z*, this expression is an empty (zero length) bit string.
x.bit[y]	Bit *y* of bitstring *x*. Alternative to the traditional MIPS notation x_y.
x.bits[y..z]	Selection of bits *y* through *z* of bit string *x*. Alternative to the traditional MIPS notation $x_{y\ z}$.
x.byte[y]	Byte *y* of bitstring *x*. Equivalent to the traditional MIPS notation $x_{8*y+7\ 8*y}$.
x.bytes[y..z]	Selection of bytes *y* through *z* of bit string *x*. Alternative to the traditional MIPS notation $x_{8*y+7\ 8*z}$
x halfword[y] x.word[i] x.doubleword[i]	Similar extraction of particular bitfields (used in e.g., MSA packed SIMD vectors).
x.bit31, x.byte0, etc.	Examples of abbreviated form of x.bit[y], etc. notation, when y is a constant.
x fieldy	Selection of a named subfield of bitstring *x*, typically a register or instruction encoding. More formally described as “Field y of register x”. For example, FIR.D = “the D bit of the Coprocessor 1 Floating-point Implementation Register (FIR)”.
+, −	2’s complement or floating point arithmetic: addition, subtraction
*, ∞	2’s complement or floating point multiplication (both used for either)
div	2’s complement integer division
mod	2’s complement modulo
/	Floating point division
<	2’s complement less-than comparison
>	2’s complement greater-than comparison
≤	2’s complement less-than or equal comparison
≥	2’s complement greater-than or equal comparison
nor	Bitwise logical NOR
xor	Bitwise logical XOR
and	Bitwise logical AND
or	Bitwise logical OR

Table 1.1 Symbols Used in Instruction Operation Statements (Continued)

Symbol	Meaning
not	Bitwise inversion
&&	Logical (non-Bitwise) AND
<<	Logical Shift left (shift in zeros at right-hand-side)
>>	Logical Shift right (shift in zeros at left-hand-side)
GPRLEN	The length in bits (32 or 64) of the CPU general-purpose registers
GPR[x]	CPU general-purpose register *x*. The content of *GPR[0]* is always zero. In Release 2 of the Architecture, GPR[x] is a short-hand notation for $SGPR[SRSCtl_{CSS}, x]$.
SGPR[s,x]	In Release 2 of the Architecture and subsequent releases, multiple copies of the CPU general-purpose registers may be implemented. *SGPR[s,x]* refers to GPR set *s*, register *x*.
FPR[x]	Floating Point operand register *x*
FCC[CC]	Floating Point condition code CC. *FCC[0]* has the same value as *COC[1]*. Release 6 removes the floating point condition codes.
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.

Table 1.1 Symbols Used in Instruction Operation Statements (Continued)

Symbol	Meaning
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 labeled **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.
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. Release 6 adds PC-relative address computation and load instructions. 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 or the microMIPS base architectures, 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 or microMIPS 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). It is optional if the FPU has 32 64-bit FPRs in which 64-bit data types are stored in any FPR. microMIPS64 implementations have a compatibility mode in which the processor references the FPRs as if it were a microMIPS32 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.

Table 1.1 Symbols Used in Instruction Operation Statements (Continued)

Symbol	Meaning
SignalException(exception, argument)	Causes an exception to be signaled, using the exception parameter as the type of exception and the argument parameter as an exception-specific argument). Control does not return from this pseudocode function—the exception is signaled at the point of the call.

1.4 Notation for Register Field Accessibility

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

Table 1.2 Read/Write Register Field Notation

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

Table 1.2 Read/Write Register Field Notation (Continued)

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

Table 1.2 Read/Write Register Field Notation (Continued)

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

1.5 For More Information

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

.
.

Chapter 2

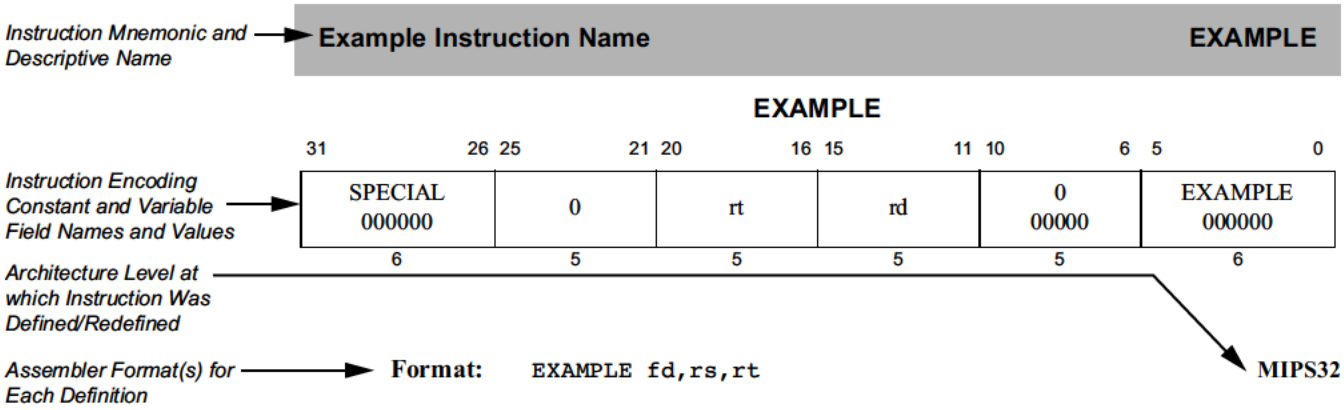
Guide to the Instruction Set

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:

Figure 2.1 Example of Instruction Description



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 the 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] ← sign extend(temp31..0)
```

Exceptions that the 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 Implementers → **Implementation Notes:**

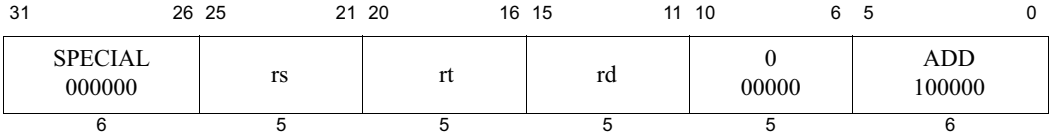
Like *Programming Notes*, except for processor implementors.

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. Instructions introduced at different times by different ISA family members, are indicated by markings such as “MIPS64, MIPS32 Release 2”. Instructions removed by particular architecture release are indicated in the Availability section.

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.

The term *decoded_immediate* is used if the immediate field is encoded within the binary format but the assembler format uses the decoded value. The term *left_shifted_offset* is used if the offset field is encoded within the binary format but the assembler format uses value after the appropriate amount of left shifting.

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 Availability and Compatibility Fields

The *Availability* and *Compatibility* sections are not provided for all instructions. These sections list considerations relevant to whether and how an implementation may implement some instructions, when software may use such instructions, and how software can determine if an instruction or feature is present. Such considerations include:

- Some instructions are not present on all architecture releases. Sometimes the implementation is required to signal a Reserved Instruction exception, but sometimes executing such an instruction encoding is architecturally defined to give UNPREDICTABLE results.
- Some instructions are available for implementations of a particular architecture release, but may be provided only if an optional feature is implemented. Control register bits typically allow software to determine if the feature is present.
- Some instructions may not behave the same way on all implementations. Typically this involves behavior that was UNPREDICTABLE in some implementations, but which is made architectural and guaranteed consistent so that software can rely on it in subsequent architecture releases.
- Some instructions are prohibited for certain architecture releases and/or optional feature combinations.
- Some instructions may be removed for certain architecture releases. Implementations may then be required to signal a Reserved Instruction exception for the removed instruction encoding; but sometimes the instruction encoding is reused for other instructions.

All of these considerations may apply to the same instruction. If such considerations applicable to an instruction are simple, the architecture level in which an instruction was defined or redefined in the *Format* field, and/or the *Restrictions* section, may be sufficient; but if the set of such considerations applicable to an instruction is complicated, the *Availability* and *Compatibility* sections may be provided.

2.1.8 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 22 for more information on the formal notation used here.

2.1.9 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.

2.1.10 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:

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:

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.

2.2.2.1.1 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
```

2.2.2.1.2 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
```

2.2.2.1.3 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
```

2.2.2.1.4 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
```

2.2.2.1.5 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.

2.2.2.2.1 Misaligned Support

MIPS processors originally required all memory accesses to be naturally aligned. MSA (the MIPS SIMD Architecture) supported misaligned memory accesses for its 128 bit packed SIMD vector loads and stores, from its introduction in MIPS Release 5. Release 6 requires systems to provide support for misaligned memory accesses for all ordinary memory reference instructions: the system must provide a mechanism to complete a misaligned memory reference for this instruction, ranging from full execution in hardware to trap-and-emulate.

The pseudocode function MisalignedSupport encapsulates the version number check to determine if misalignment is supported for an ordinary memory access.

Figure 2.16 MisalignedSupport Pseudocode Function

```
predicate ← MisalignedSupport ()
   return Config.AR ≥ 2 // Architecture Revision 2 corresponds to MIPS Release 6.
end function
```

See Appendix B, “Misaligned Memory Accesses” on page 511 for a more detailed discussion of misalignment, including pseudocode functions for the actual misaligned memory access.

2.2.2.2.2 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 2.17 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
```

2.2.2.2.3 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 2.18 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
```

2.2.2.2.4 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 2.19 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
```

2.2.2.2.5 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 2.20 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)

2.2.2.2.6 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 2.21 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).

2.2.2.3.1 ValueFPR

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

Figure 2.22 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:
         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.

2.2.2.3.2 StoreFPR

Figure 2.23 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]   ← UNPREDICTABLE32 || value31..0
                        FPR[fpr+1] ← UNPREDICTABLE32 || value63..32
                    endif
                else
                    FPR[fpr] ← value
                endif

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

        endcase

    endfunction StoreFPR
```

2.2.2.3.3 CheckFPException

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

Figure 2.24 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
```

2.2.2.3.4 FPConditionCode

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

Figure 2.25 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
```

2.2.2.3.5 SetFPConditionCode

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

Figure 2.26 SetFPConditionCode Pseudocode Function

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

endfunction SetFPConditionCode
```

2.2.2.4 Instruction Mode Checking Functions

2.2.2.4.1 Are64BitFPOperationsEnabled

The Are64BitFPOperationsEnabled function is used to determine if a 64-bit floating point instruction may be executed (and conversely, whether a Reserved Instruction exception should be signaled). On a Release 1 processor, such operations are never enabled and this function returns 0. On a Release 2 processor, which supports a 64-bit FPU on a 32-bit processors (and therefore, on a 64-bit processor running with 64-bit operations disabled), the function simply checks the *F64* bit in the *FIR* register.

Figure 2.27 Are64BitFPOperationsEnabled Pseudocode Function

```
enabled ←Are64BitFPOperationsEnabled()

   /* enabled: true if 64-bit floating point operations are enabled; */
   /* false if they are not */

   if (ArchitectureRevision() ≥ 2) then
      Are64BitFPOperationsEnabled ← FIRF64
   else
      Are64BitFPOperationsEnabled ← 0
   endif

endfunction Are64FPBitOperationsEnabled
```

2.2.2.4.2 IsCoprocessorEnabled

The IsCoprocessorEnabled function is used to determine if access is available to one of the four coprocessors. This is primarily done by looking at the value of the appropriate *CU* bit in the *Status* register, but complicated by the fact that access to coprocessor 0 is also enabled if the processor is running in Kernel Mode or Debug Mode.

Figure 2.28 IsCoprocessorEnabled PseudocodeFunction

```
enabled ← IsCoprocessorEnabled(z)

   /* enabled: true if the coprocessor is enabled; false if it is not */

   /* z: The coprocessor unit number in the range 0..3 */

   case z of
      0:
         IsCoprocessorEnabled ←
            (Status_KSU = 0b00) or (Debug_DM = 1) or
            (Status_EXL = 1) or (Status_ERL = 1)
      1:
         IsCoprocessorEnabled ← (Status_CU1 = 1)
      2:
         IsCoprocessorEnabled ← (Status_CU2 = 1)
      3:
         IsCoprocessorEnabled ← (Status_CU3 = 1)
   endcase

endfunction IsCoprocessorEnabled
```

2.2.2.4.3 IsCoprocessor2Implemented

The IsCoprocessor2Implemented function is used to determine if coprocessor 2 is implemented. This is determined by the state of the *C2* bit in the *Config1* register.

Figure 2.29 IsCoprocessor2 Pseudocode Function

```
impl ← IsCoprocessor2Implemented()

   /* impl: true if coprocessor 2 is implemented; false if it is not */

   IsCoprocessor2Implemented ← Config1_C2

endfunction IsCoprocessor2Implemented
```

2.2.2.4.4 IsEJTAGImplemented

The IsEJTAGImplemented function is used to determine if EJTAG is implemented by the processor. This is determined by the state of the *EP* bit in the *Config1* register.

Figure 2.30 IsEJTAGImplemented Pseudocode Function

```
impl ← IsEJTAGImplemented()

   /* impl: true if EJTAG is implemented; false if it is not */

   IsEJTAGImplemented ← Config1_EP

endfunction IsEJTAGImplemented
```

2.2.2.4.5 IsFloatingPointImplemented

The IsFloatingPointImplemented function is used to determine if floating point is implemented by the processor and, additionally, whether a particular floating point datatype is implemented. Whether floating point is implemented at all is determined by the state of the *FP* bit in the *Config1* register. The determination of whether a particular datatype is implemented is done by looking at the architecture of the chip (MIPS32 or MIPS64, as determined by the *AT* field in the *Config* register), and the state of the *S*, *D*, and *PS* bits in the *FIR* coprocessor 1 register.

Figure 2.31 IsFloatingPointImplemented Pseudocode Function

```
impl ← IsFloatingPointImplemented(fmt)

   /* impl: true if floating point is implemented; false  if it is not */

   /* fmt: The floating point datatype to be checked:/
   /*        0: Determine if any floating point datatype is implemented */
   /*        S, D, W, L, PS:  Determine if a specific datatype is */
   /*                         implemented

   if Config1FP = 0 then
      IsFloatingPointImplemented ← 0
   else
      case fmt of
         0:
            IsFloatingPointImplemented ← 1
         S:
            IsFloatingPointImplemented ← FIRS

         W:
            IsFloatingPointImplemented ←
               (   ((ArchitectureRevision() = 1) and FIRS)
                                 or
                   ((ArchitectureRevision() ≥ 2) and FIRW) )

         D:
            IsFloatingPointImplemented ← FIRD

         L: /* L datatype is valid on a MIPS64 Release 1 implementation */
            /* or on a Release 2 implementation with the L bit set in FIR */
            IsFloatingPointImplemented ←
               (   ((ArchitectureRevision() = 1) and
                     ((ConfigAT = 1) or (Config1AT = 2)))
                                 or
                   ((ArchitectureRevision() ≥ 2) and FIRL) )

         PS:
            IsFloatingPointImplemented ← FIRPS and
               (   ((ArchitectureRevision() = 1) and
                     ((ConfigAT = 1) or (Config1AT = 2)))
                                 or
                   (ArchitectureRevision() ≥ 2) )
      endcase
   endif

endfunction IsFloatingPointImplemented
```

2.2.2.5 Pseudocode Functions Related to Sign and Zero Extension

2.2.2.5.1 Sign extension and zero extension in pseudocode

Much pseudocode uses a generic function `sign_extend` without specifying from what bit position the extension is done, when the intention is obvious. E.g. `sign_extend(immediate16)` or `sign_extend(disp9)`.

However, sometimes it is necessary to specify the bit position. For example, $\texttt{sign\_extend(temp}_{31..0}\texttt{)}$ or the more complicated $\texttt{(offset}_{15}\texttt{)}^{\texttt{GPRLEN-(16+2)}}\ \texttt{||}\ \texttt{offset}\ \texttt{||}\ 0^2$.

The explicit notation `sign_extend.nbits(val)` or `sign_extend(val,nbits)` is suggested as a simplification. They say to sign extend as if an nbits-sized signed integer. The width to be sign extended to is usually apparent by context, and is usually GPRLEN, 32 or 64 bits. The previous examples then become.

$\texttt{sign\_extend(temp}_{31..0}\texttt{)}$
`= sign_extend.32(temp)`

and

$\texttt{(offset}_{15}\texttt{)}^{\texttt{GPRLEN-(16+2)}}\ \texttt{||}\ \texttt{offset}\ \texttt{||}\ 0^2$
`= sign_extend.16(offset)<<2`

Note that sign_extend.N(value) extends from bit position N-1, if the bits are numbered 0..N-1 as is typical.

The explicit notations `sign_extend.nbits(val)` or `sign_extend(val,nbits)` is used as a simplification. These notations say to sign extend as if an nbits-sized signed integer. The width to be sign extended to is usually apparent by context, and is usually GPRLEN, 32 or 64 bits.

Figure 2.32 sign_extend Pseudocode Functions

```
sign_extend.nbits(val) = sign_extend(val,nbits) /* syntactic equivalents */

function sign_extend(val,nbits)
   return (val_nbits-1)^GPRLEN-nbits || val_nbits-1..0
end function
```

The earlier examples can be expressed as

$\texttt{(offset}_{15}\texttt{)}^{\texttt{GPRLEN-(16+2)}}\ \texttt{||}\ \texttt{offset}\ \texttt{||}\ 0^2$
`= sign_extend.16(offset) << 2)`

and

$\texttt{sign\_extend(temp}_{31..0}\texttt{)}$
`= sign_extend.32(temp)`

Similarly for `zero_extension`, although zero extension is less common than sign extension in the MIPS ISA.

Floating point may use notations such as `zero_extend.fmt` corresponding to the format of the FPU instruction. E.g. `zero_extend.S` and `zero_extend.D` are equivalent to `zero_extend.32` and `zero_extend.64`.

Existing pseudocode may use any of these, or other, notations.

2.2.2.5.2 memory_address

The pseudocode function `memory_address` performs mode-dependent address space wrapping for compatibility between MIPS32 and MIPS64. It is applied to all memory references. It may be specified explicitly in some places, particularly for new memory reference instructions, but it is also declared to apply implicitly to all memory references as defined below. In addition, certain instructions that are used to calculate effective memory addresses but which are not themselves memory accesses specify memory_address explicitly in their pseudocode.

Figure 2.33 memory_address Pseudocode Function

```
function memory_address(ea)
   return ea
end function
```

On a 32-bit CPU, `memory_address` returns its 32-bit effective address argument unaffected.

In addition to the use of `memory_address` for all memory references (including load and store instructions, LL/SC), Release 6 extends this behavior to control transfers (branch and call instructions), and to the PC-relative address calculation instructions (ADDIUPC, AUIPC, ALUIPC). In newer instructions the function is explicit in the pseudocode.

Implicit address space wrapping for all instruction fetches is described by the following pseudocode fragment which should be considered part of instruction fetch:

Figure 2.34 Instruction Fetch Implicit memory_address Wrapping

```
PC ← memory_address( PC )
( instruction_data, length ) ← instruction_fetch( PC )
/* decode and execute instruction */
```

Implicit address space wrapping for all data memory accesses is described by the following pseudocode, which is inserted at the top of the AddressTranslation pseudocode function:

Figure 2.35 AddressTranslation implicit memory_address Wrapping

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

In addition to its use in instruction pseudocode,

2.2.2.6 Miscellaneous Functions

This section lists miscellaneous functions not covered in previous sections.

2.2.2.6.1 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 2.36 SignalException Pseudocode Function

```
SignalException(Exception, argument)

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

endfunction SignalException
```

2.2.2.6.2 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 2.37 SignalDebugBreakpointException Pseudocode Function

```
SignalDebugBreakpointException()

endfunction SignalDebugBreakpointException
```

2.2.2.6.3 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 2.38 SignalDebugModeBreakpointException Pseudocode Function

```
SignalDebugModeBreakpointException()

endfunction SignalDebugModeBreakpointException
```

2.2.2.6.4 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 2.39 NullifyCurrentInstruction PseudoCode Function

```
NullifyCurrentInstruction()

endfunction NullifyCurrentInstruction
```

2.2.2.6.5 PolyMult

The PolyMult function multiplies two binary polynomial coefficients.

Figure 2.40 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 35 for a description of the *op* and *function* subfields.

Chapter 3

Introduction to the MIPS® MT Architecture Extension

3.5 Background

Multi-threading, or the concurrent presence of multiple active threads or contexts of execution on the same CPU, is an increasingly widely-used technique for tolerating memory and execution latency and for getting higher utilization out of processor functional units. The MIPS® Multi-threading (MT) Module is an extension to the nanoMIPS32™ Architecture which provides a framework for multi-threading the MIPS processor architecture.

3.6 Definitions and General Description

A *thread context*, for the purposes of this document, is a collection of processor state necessary to describe the state of execution of an instruction stream in the nanoMIPS32 Instruction Set Architecture. It includes a set of general purpose registers (GPRs), the MIPS Hi/Lo multiplier result registers, some internal representation of a program counter, and some associated nanoMIPS32 privileged system coprocessor (CP0) state, specifically:

- The CU3..CU0, MX, and KSU fields of the CP0 Status register
- The ASID field of the CP0 EntryHi register.
- The SSt and OffLine fields of the EJTAG Debug register.
- The CP0 UserLocal register, if implemented.

A thread context also contains some new privileged resource state, to allow software to manage the new multi-threading capabilities. Thread Context will be abbreviated as *TC*, both in the interests of brevity, and to minimize the confusion between a TC as state/storage and a thread of execution as a sequence of instructions.

A *processor context* is a larger collection of processor state, which includes at least one TC, but also the CP0 and system state necessary to describe an instantiation of the full nanoMIPS32 Privileged Resource Architecture.

The MIPS MT Module allows two distinct, but not mutually-exclusive, multi-threading capabilities. A single MIPS processor or core can contain some number of *Virtual Processing Elements* (VPEs), each of which supports at least one thread context. To software, an *N* VPE processor looks like an *N*-way symmetric multiprocessor. All legacy nanoMIPS32 read-write CP0 state must be implemented per-VPE. This allows existing SMP-capable operating systems to manage the set of VPEs, which transparently share the processor's execution units and other resources. A processor or core implementing multiple MIPS MT VPEs is referred to as a *Virtual Multiprocessor*, or VMP.

Each VPE can also contain some number of TCs beyond the single TC implicitly required by the base architecture. Multi-threaded VPEs require explicit operating system support, but with such support they provide a lightweight, fine-grained multi-threaded programming model wherein threads can be created and destroyed, without operating system intervention in the typical cases, using new FORK and YIELD instructions, and where system service threads can be scheduled in response to external events with zero interrupt latency.

A TC may be in one of two allocation states, *free* or *activated*. A **free** TC has no valid content and cannot be scheduled to issue instructions. An **activated** TC will be scheduled according to the implemented policies to fetch and issue instructions from its program counter. Only activated TCs may be scheduled. Only free TCs may be allocated to create new threads. Allocation and deallocation of TCs may be done explicitly by privileged software, or automatically via FORK and YIELD instructions that can be executed in user mode. Only TCs which have been explicitly designated as *Dynamically Allocatable* (DA) may be allocated or deallocated by FORK and YIELD.

An activated TC may be *running* or *blocked*. A **running** TC will fetch and issue instructions according to the thread scheduling policy in effect for the processor. Any or all running TCs may have instructions in the pipeline of a processor at a given point of time, but it is not knowable to software precisely which ones. A **blocked** TC is one which has issued an instruction which performs an explicit synchronization that has not yet been satisfied. While a running, activated TC may be stalled momentarily due to functional unit delays, memory load dependencies, or scheduling rules, its instruction stream will advance on its own within the limitations of the pipeline implementation. The instruction stream of a blocked TC cannot advance without some change in system state being effected by another thread or by external hardware, and as such it may remain blocked for an unbounded period of time.

Independently of whether it is free or activated, a TC may be *halted*. A **halted** TC is inhibited from being allocated by a FORK instruction, even if free, and inhibited from fetching and issuing instructions, even if activated. Only a TC in a halted state is guaranteed to be stable as seen by other TCs. Multi-threaded execution may be temporarily inhibited on a VPE due to exceptions or explicit software interventions, but the activated threads that are inhibited in such cases are considered to be *suspended*, rather than implicitly halted. A **suspended** thread is inhibited from any action which might cause exceptions or otherwise change global VPE privileged resource state, but, unlike a halted thread, it may still have instructions active in the pipeline, and its internal TC state, including GPR values, may still be unstable.

And independently of whether an activated TC is halted, it will not be scheduled to fetch or issue if it has been set *offline* by code executing in EJTAG Debug mode, via the *OffLine* bit of the *Debug* register (see the EJTAG specification).

If executing in a sufficiently privileged mode, one TC can access another TC's register state, via new instructions to move to/from the registers of a "target" TC.

To allow for fine-grain synchronization of cooperating threads, an inter-thread communication (ITC) memory space can be created in virtual memory, with gating-storage semantics that allow threads to be blocked on loads or stores until data has been produced or consumed by other threads. These gating storage semantics can also be applied to I/O devices such as FIFOs to provide a data-driven execution model.

The thread creation/destruction and synchronization capabilities function without operating system intervention in the general case, but the resources they manipulate are all virtualizable via an operating system. This allows the execution of multi-threaded programs with more "virtual" threads than there are TCs on a VPE, and for the migration of threads to balance load in multiprocessor systems. At any particular point in its execution, a thread is bound to a particular TC on a particular VPE. The number of that TC provides a unique identifier *at that point in time*. But context switching and migration can cause a single sequential thread of execution to have a series of different TCs, possibly on a series of different VPEs.

Dynamic binding of TCs, TLB entries, and other resources to multiple VPEs on the same processor can be performed in a special processor configuration state. By default, one VPE of each processor enters its reset vector as if it were a standard nanoMIPS32 core.

Chapter 4

MIPS® MT Multi-Threaded Execution and Exception Model

4.1 Multi-Threaded Execution

The MIPS Multi-threading Module does not impose any particular implementation or scheduling model on the execution of parallel threads and VPEs. Scheduling may be round-robin, time-sliced to an arbitrary granularity, or simultaneous. An implementation must not, however, allow a thread which is blocked or suspended by an external or software dependency to monopolize any shared processor resource which could produce a hardware deadlock.

4.2 MIPS® MT Exception Model

All multiple threads executing on a single VPE share the same system coprocessor, the same TLB, and the same virtual address space. Each TC has an independent Kernel/Supervisor/User state and ASID for the purposes of instruction decode and memory access. When an exception of any kind is taken, all TCs of the affected VPE other than the one taking the exception are stopped and suspended until the EXL and ERL bits of the Status word are cleared, or, in the case of an EJTAG Debug exception, the Debug state is exited. Debug exceptions have the broader effect of suspending the TCs of other VPEs of the processor as well. See Section 10.3 "Debug Exception Handling". All sources of additional synchronous exceptions must be quiesced before the exception handler begins execution. If simultaneous exception conditions occur across multiple threads, only a single exception, one with the highest relative priority, will be dispatched to a handler. The others will be deferred until EXL/ERL or the Debug state are cleared, and the associated instructions replayed.

Implementations which pre-fetch instructions for suspended TCs must not use the exception values of ERL, EXL, or DM for instruction fetches for the suspended TCs. If fetching is to continue for the suspended threads, per-TC copies of the pre-suspension state of these bits must be used.

Exception handlers for synchronous exceptions caused by the execution of an instruction stream, such as TLB miss and floating-point exceptions, are executed using the GPRs of the TC associated with the instruction stream, unless they are configured to be executed using a Shadow Register Set. When an unmasked asynchronous exception, such as an interrupt, is raised to a VPE, it is implementation-dependent which eligible TC is used to execute the exception handler, but TCs can be selectively exempted from use by asynchronous exception handlers.

Imprecise, synchronous exceptions are not permitted on a MIPS MT processor. All exceptions are either precise and synchronous, or asynchronous.

Each exception is associated with an activated TC, even if shadow register sets are used to run the exception handler. This associated TC is referenced whenever a SRSCtl PSS value of 0 is used by RDPGPR and WRPGPR instructions executed by the exception handler.

4.3 New Exception Conditions

The Multi-threading Module introduces 6 new exception conditions.

- Thread Overflow condition, where a TC allocation request cannot be satisfied.
- Thread Underflow condition, where the termination and deallocation of a thread leaves no dynamically allocatable TCs activated on a VPE.
- Invalid Qualifier condition, where a YIELD instruction specifies an invalid condition for resuming execution.
- Gating Storage exception condition, where implementation-dependent logic associated with gating or inter-thread communication (ITC) storage requires software intervention.
- YIELD Scheduler exception condition, where a valid YIELD instruction would have caused a rescheduling of a TC, and the YIELD Intercept bit is set.
- GS Scheduler exception, where a Gating Storage load or store would have blocked and caused a rescheduling of a TC, and the GS Intercept bit is set.

These exception conditions are mapped to a single new *Thread* exception. They can be distinguished based on the CP0 *VPEControl EXCPT* field value when the exception is raised.

4.4 New Exception Priority

The Thread exception groups together a number of possible exception conditions which can be detected at different stages of a processor pipeline. Thus, different Thread exception conditions may have different priorities relative to other nanoMIPS32 exceptions. The following table describes where Thread exceptions fit in to the nanoMIPS32 priority scheme.

Table 4.1 Priority of Exceptions in MIPS® MT

Exception	Description	Type
Reset	The Cold Reset signal was asserted to the processor	Asynchronous Reset
Soft Reset	The Reset signal was asserted to the processor	
Debug Single Step	An EJTAG Single Step occurred. Prioritized above other exceptions, including asynchronous exceptions, so that one can single-step into interrupt (or other asynchronous) handlers.	Synchronous Debug
Debug Interrupt	An EJTAG interrupt (EjtagBrk or DINT) was asserted.	Asynchronous Debug
Imprecise Debug Data Break	An imprecise EJTAG data break condition was asserted.	
Nonmaskable Interrupt (NMI)	The NMI signal was asserted to the processor.	Asynchronous
Machine Check	An internal inconsistency was detected by the processor.	
Interrupt	An enabled interrupt occurred.	
Deferred Watch	A watch exception, deferred because EXL was one when the exception was detected, was asserted after EXL went to zero.	
Debug Instruction Break	An EJTAG instruction break condition was asserted. Prioritized above instruction fetch exceptions to allow break on illegal instruction addresses.	Synchronous Debug

Table 4.1 Priority of Exceptions in MIPS® MT (Continued)

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

4.5 Interrupts

In general, the binding of hardware interrupts to VPEs is implementation-dependent. Interrupt inputs to a processor may be presented in common to all VPEs, leaving it up to software whether any or all VPEs enable and service a

given interrupt. A processor may also provide distinct interrupt signals per supported VPE, and/or extend the External Interrupt Controller (EIC) interface to express a VPE identifier in addition to the Exception Vector Offset and Shadow Set Number.

The exception to the above is the hardware interrupt generated by the Count/Compare registers. This logic must be replicated per-VPE, and interrupt events associated with the Count/Compare values of a specific VPE result in interrupt requests only to that VPE.

Depending on the implementation, Performance Counter interrupts may be local to a VPE or "broadcast" to all VPEs of a processor.

Software interrupts IP1 and IP0 must by default be local to a VPE.

4.6 Bus Error Exceptions

Bus error exceptions on instruction fetch (IBE) in a MIPS MT processor are synchronous and must be precise as per Section 4.2 "MIPS® MT Exception Model". Bus errors on load/store operations (DBE) are considered to be imprecise and are therefore non-maskable asynchronous exceptions delivered to the VPE where the operation was issued. A DBE exception may thus be taken by a TC other than the one which issued the failing operation. A per-TC TBE bit is defined to allow exception handlers to determine which TC(s) were associated with the failed bus transaction (see Section 6.13 "TCBind Register (CP0 Register 2, Select 2)").

If a DBE results from an operation that was combined across VPEs, a DBE exception must be delivered to all VPEs affected. Where the origin of the failure cannot be determined, all VPEs in a processor must take a DBE exception.

Implementations may provide additional bus error diagnostic information in implementation-dependent CP0 register fields. The DBE state, including the per-TC TBE state, should be analyzed in the context of this information.

4.7 Cache Error Exceptions

Cache memories may be shared between multiple VPEs on a virtual multiprocessor. In the event of a cache parity or other data integrity error, all VPEs sharing the cache may be affected, and all must take a Cache Error exception. It is the responsibility of software to coordinate any diagnostics or re-initialization of the shared cache, communicating by means other than cached storage.

4.8 EJTAG Debug Exceptions

EJTAG Debug exceptions override MIPS MT scheduling and TC management. See Section 10.3 "Debug Exception Handling".

4.9 Shadow Register Sets

MIPS MT optionally allows TCs to be assigned for use as Shadow Register Set (SRS) storage. This is accomplished by writing the TC number into a programmable field of one of the *SRSConf* registers (see Section 6.20 "SRSConf0 (CP0 Register 6, Select 1)"). A TC assigned for use as SRS storage must never be Activated, nor may it be programmed to be Dynamically Allocatable. *Because SRS management and control is performed on a per-VPE basis, with only a single SRSCtl register per VPE, multi-threading should never be explicitly re-enabled in an exception handler which executes using an SRS.*

Chapter 5

MIPS® MT Instructions

5.1 New Instructions

The MIPS MT Module contains 8 new instructions.

FORK and **YIELD** control thread allocation, deallocation, and scheduling, and are available in all execution modes if implemented and enabled.

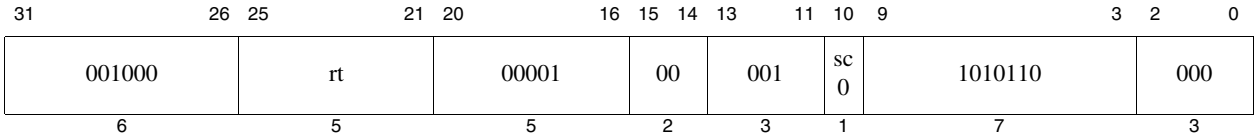
MFTR and **MTTR** are system coprocessor (Cop0) instructions available to privileged system software for managing thread state.

EMT and **DMT** are privileged Cop0 instructions for enabling and disabling multi-threaded operation of a VPE.

EVPE and **DVPE** are privileged Cop0 instructions for enabling and disabling multi-VPE operation of a processor.

These instructions will cause a **Reserved Instruction** exception if executed by a processor not implementing the MIPS MT Module.

DMT — Disable Multi-Threaded Execution



Format: `DMT` **MIPS MT**

`DMT rt`

Purpose: Disable Multi-Threaded Execution

To return the previous value of the *VPEControl* register (see Section 6.5) and disable multi-threaded execution. If DMT is specified without an argument, GPR *r0* is implied, which discards the previous value of the *VPEControl* register.

Description: `GPR[rt] ← VPEControl; VPEControl`$_{TE}$ `← 0`

The current value of the *VPEControl* register is loaded into general register *rt*. The Threads Enable (*TE*) bit in the *VPEControl* register is then cleared, suspending concurrent execution of instruction streams other than that which issues the DMT. This is independent of any per-TC halted state.

Restrictions:

If access to Coprocessor 0 is not enabled, a Coprocessor Unusable Exception is signaled.

In implementations that do not implement the MT Module, this instruction results in a Reserved Instruction Exception.

Operation:

This operation specification is for the general multi-threading enable/disable operation, with the *sc* (set/clear) field as a variable. The individual instructions EMT and DMT have a specific value for the *sc* field.

```
if IsCoprocessorEnabled(0) then
   if Config3MT then
      data ← VPEControl
      GPR[rt] ← data
      VPEControlTE ← sc
   else
      SignalException(ReservedInstruction)
   endif
else
   SignalException(CoprocessorUnusable, 0)
endif
```

Exceptions:

Coprocessor Unusable
Reserved Instruction (Implementations that do not include the MT Module)

Implementation Notes:

DMT accesses a COP0 register and assumes a hard-coded value of *rd*=1 and *sel*=1 for *VPEControl*.

The *sc* field indicates whether the operation is a bit clear or set, as follows:

sc	Operation
0	Clear bit specified by the *pos* field
1	Set bit specified by the *pos* field

The general description of the final operation provided by this operation (with Coprocessor Unusable and Reserved

Instruction exception checks excluded for clarity) is:

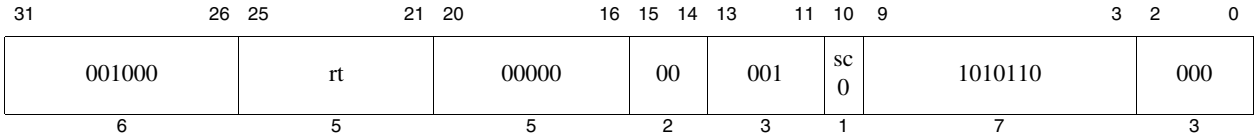
```
data ← CPR[0,rd, sel]
GPR[rt] ← data
CPR[0, rd, sel]pos ← sc
```

Programming Notes:

The effects of this instruction are identical to those accomplished by the sequence of reading *VPEControl* into a GPR, clearing the *TE* bit to create a temporary value in a second GPR, and writing that value back to *VPEControl*. Unlike the multiple instruction sequence, however, the DMT instruction does not consume a temporary register, and cannot be aborted by an interrupt or exception.

The effect of a DMT instruction may not be instantaneous. An instruction hazard barrier, e.g., JR.HB, is required to guarantee that all other threads have been suspended. If a DMT instruction is followed in the same instruction stream by an MFC0 or MFTR from the *VPEControl* register, a JALR.HB, JR.HB, EHB, or ERET instruction must be issued between the DMT and the read of *VPEControl* to guarantee that the new state of *TE* will be accessed by the read.

DVPE — Disable Virtual Processor Execution



Format: `DVPE` **MIPS MT**

`DVPE rt`

Purpose: Disable Virtual Processor Execution

To return the previous value of the *MVPControl* register (see Section 6.2) and disable multi-VPE execution. If DVPE is specified without an argument, GPR r0 is implied, which discards the previous value of the *MVPControl* register.

Description: `GPR[rt] ← MVPControl; MVPControl`$_{EVP}$ `← 0`

The current value of the *MVPControl* register is loaded into general register *rt*. The Enable Virtual Processors (*EVP*) bit in the *MVPControl* register is then cleared, suspending concurrent execution of instruction streams other than the instruction stream that issues the DVPE.

Restrictions:

If access to Coprocessor 0 is not enabled, a Coprocessor Unusable Exception is signaled.

If the VPE executing the instruction is not a Master VPE, with the *MVP* bit of the *VPEConf0* register set, the *EVP* bit is unchanged by the instruction.

In implementations that do not implement the MT Module, this instruction results in a Reserved Instruction Exception.

Operation:

This operation specification is for the general VPE enable/disable operation, with the *sc* (set/clear) field as a variable. The individual instructions EVPE and DVPE have a specific value for the *sc* field.

```
if IsCoprocessorEnabled(0) then
   if Config3MT then
      data ← MVPControl
      GPR[rt] ← data
      if(VPEConf0MVP = 1) then
         MVPControlEVP ← sc
      endif
   else
      SignalException(ReservedInstruction)
   endif
else
   SignalException(CoprocessorUnusable, 0)
endif
```

Exceptions:

Coprocessor Unusable
Reserved Instruction (Implementations that do not include the MT Module)

Implementation Notes:

DVPE accesses a COP0 register and assumes a hard-coded value of *rd*=0 and *sel*=1 for *MVPControl*.

The *sc* field indicates whether the operation is a bit clear or set, as follows:

sc	Operation
0	Clear bit specified by the pos field
1	Set bit specified by the pos field

The general description of the final operation provided by this operation (with Coprocessor Unusable and Reserved Instruction exception checks excluded for clarity) is:

```
data ← CPR[0,rd, sel]
GPR[rt] ← data
CPR[0, rd, sel]pos ← sc
```

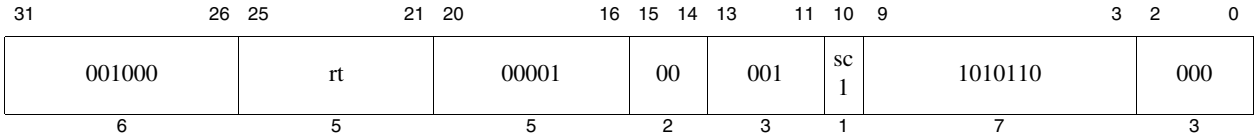
Programming Notes:

The effects of this instruction are identical to those accomplished by the sequence of reading *MVPControl* into a GPR, clearing the *EVP* bit to create a temporary value in a second GPR, and writing that value back to *MVPControl*. Unlike the multiple instruction sequence, however, the DVPE instruction does not consume a temporary register, and cannot be aborted by an interrupt or exception, nor by the scheduling of a different instruction stream.

The effect of a DVPE instruction may not be instantaneous. An instruction hazard barrier, e.g., JR.HB, is required to guarantee that all other TCs have been suspended.

If a DVPE instruction is followed in the same instruction stream by an MFC0 or MFTR from the *MVPControl* register, a JALR.HB, JR.HB, EHB, or ERET instruction must be issued between the DVPE and the read of *MVPControl* to guarantee that the new state of *EVP* will be accessed by the read.

EMT — Enable Multi-Threaded Execution



Format: `EMT` **MIPS MT**

`EMT rt`

Purpose: Enable Multi-Threaded Execution

To return the previous value of the *VPEControl* register (see Section 6.5) and to enable multi-threaded execution. If EMT is specified without an argument, GPR *r0* is implied, which discards the previous value of the *VPEControl* register.

Description: `GPR[rt] ← VPEControl; VPEControl`$_{TE}$ `← 1`

The current value of the *VPEControl* register is loaded into general register *rt*. The Threads Enable (*TE*) bit in the *VPEControl* register is then set, allowing multiple instruction streams to execute concurrently.

Restrictions:

If access to Coprocessor 0 is not enabled, a **Coprocessor Unusable Exception** is signaled.

In implementations that do not implement the MT Module, this instruction results in a **Reserved Instruction Exception**.

Operation:

This operation specification is for the general multi-threading enable/disable operation, with the *sc* (set/clear) field as a variable. The individual instructions EMT and DMT have a specific value for the *sc* field.

```
if IsCoprocessorEnabled(0) then
   if Config3MT then
      data ← VPEControl
      GPR[rt] ← data
      VPEControlTE ← sc
   else
      SignalException(ReservedInstruction)
   endif
else
   SignalException(CoprocessorUnusable, 0)
endif
```

Exceptions:

Coprocessor Unusable
Reserved Instruction (Implementations that do not include the MT Module)

Implementation Notes:

EMT accesses a COP0 register and assumes a hard-coded value of *rd*=1 and *sel*=1 for *VPEControl*.

The *sc* field indicates whether the operation is a bit clear or set, as follows:

sc	Operation
0	Clear bit specified by the *pos* field
1	Set bit specified by the *pos* field

The general description of the final operation provided by this operation (with Coprocessor Unusable and Reserved

Instruction exception checks excluded for clarity) is:

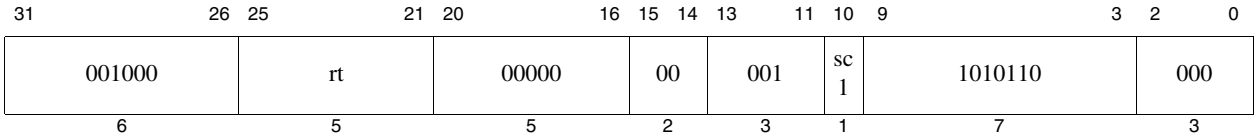
```
data ← CPR[0,rd, sel]
GPR[rt] ← data
CPR[0, rd, sel]pos ← sc
```

Programming Notes:

The effects of this instruction are identical to those accomplished by the sequence of reading *VPEControl* into a GPR, setting the *TE* bit to create a temporary value in a second GPR, and writing that value back to *VPEControl*. Unlike the multiple instruction sequence, however, the EMT instruction does not consume a temporary register, and cannot be aborted by an interrupt or exception.

If an EMT instruction is followed in the same instruction stream by an MFC0 or MFTR from the *VPEControl* register, a JALR.HB, JR.HB, EHB, or ERET instruction must be issued between the EMT and the read of *VPEControl* to guarantee that the new state of *TE* will be accessed by the read.

EVPE — Enable Virtual Processor Execution



Format: `EVPE` **MIPS MT**

`EVPE rt`

Purpose: Enable Virtual Processor Execution

To return the previous value of the *MVPControl* register (see Section 6.2) and enable multi-VPE execution. If EVPE is specified without an argument, GPR r0 is implied, which discards the previous value of the *MVPControl* register.

Description: `GPR[rt] ← MVPControl; MVPControl`$_{EVP}$ `← 1`

The current value of the *MVPControl* register is loaded into general register *rt*. The Enable Virtual Processors (*EVP*) bit in the *MVPControl* register is then set, enabling concurrent execution of instruction streams on all non-inhibited Virtual Processing Elements (VPEs) on a processor.

Restrictions:

If access to Coprocessor 0 is not enabled, a Coprocessor Unusable Exception is signaled.

If the VPE executing the instruction is not a Master VPE, with the *MVP* bit of the *VPEConf0* register set, the *EVP* bit is unchanged by the instruction.

In implementations that do not implement the MT Module, this instruction results in a Reserved Instruction Exception.

Operation:

This operation specification is for the general VPE enable/disable operation, with the *sc* (set/clear) field as a variable. The individual instructions EVPE and DVPE have a specific value for the *sc* field.

```
if IsCoprocessorEnabled(0) then
   if Config3MT then
      data ← MVPControl
      GPR[rt] ← data
      if(VPEConf0MVP = 1) then
         MVPControlEVP ← sc
      endif
   else
      SignalException(ReservedInstruction)
   endif
else
   SignalException(CoprocessorUnusable, 0)
endif
```

Exceptions:

Coprocessor Unusable
Reserved Instruction (Implementations that do not include the MT Module)

Implementation Notes:

EVPE accesses a COP0 register and assumes a hard-coded value of *rd*=0 and *sel*=1 for *MVPControl*.

The *sc* field indicates whether the operation is a bit clear or set, as follows:

sc	Operation
0	Clear bit specified by the *pos* field
1	Set bit specified by the *pos* field

The general description of the final operation provided by this operation (with Coprocessor Unusable and Reserved Instruction exception checks excluded for clarity) is:

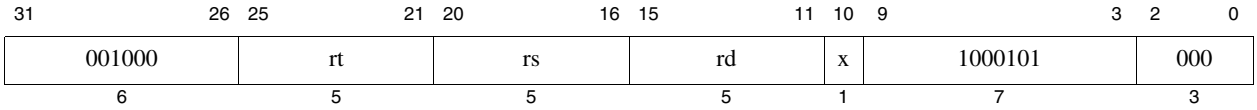
```
data ← CPR[0,rd, sel]
GPR[rt] ← data
CPR[0, rd, sel]pos ← sc
```

Programming Notes:

The effects of this instruction are identical to those accomplished by the sequence of reading *MVPControl* into a GPR, setting the *EVP* bit to create a temporary value in a second GPR, and writing that value back to *MVPControl*. Unlike the multiple instruction sequence, however, the EVPE instruction does not consume a temporary register, and cannot be aborted by an interrupt or exception, nor by the scheduling of a different instruction stream.

If an EVPE instruction is followed in the same instruction stream by an MFC0 or MFTR from the *MVPControl* register, a JALR.HB, JR.HB, EHB, or ERET instruction must be issued between the EVPE and the read of *MVPControl* to guarantee that the new state of *EVP* will be accessed by the read.

FORK — Allocate and Schedule a New Thread



Format: `FORK rd, rs, rt` **MIPS MT**

Purpose: Allocate and Schedule a New Thread

To cause a thread context to be allocated and associated with a new instruction stream.

Description: `NewThread's GPR[rd] ← GPR[rt], NewThread's TCRestart ← GPR[rs]`

The FORK instruction causes a free dynamically allocatable thread context (TC) to be allocated and activated on the issuing VPE. It takes two operand values from GPRs. The *rs* value is used as the starting fetch address and execution mode for the new thread. The *rt* value is copied into GPR *rd* of the new TC. The *TCStatus* register of the new TC is set up as a function of the FORKing TC as described in Section 6.12. If the *UserLocal* register is implemented, the *UserLocal* value of the FORKing TC is also copied to the new TC. The newly allocated TC will begin executing instructions according to the implemented scheduling policy if and when multi-threaded execution is otherwise enabled.

Restrictions:

If no free, non-halted, dynamically allocatable TC is available for the fork, a Thread Exception is raised for the FORK instruction, with the *VPEControl.EXCPT* CP0 register field set to 1 to indicate the Thread Overflow case.

Processors which implement only a single TC per VPE may implement FORK by simply raising the Thread Exception and indicating the Overflow.

Any exceptions associated with the virtual address passed in *rs* will be taken by the new thread of execution.

Operation:

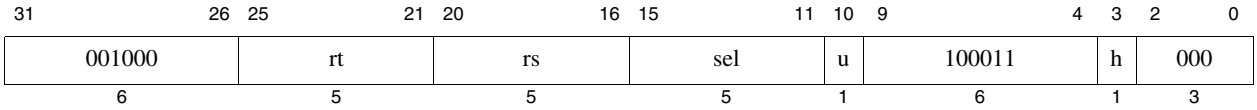
```
if Config3MT = 1 then
    success ← 0
    for t in 0...MVPConf0PTC
        if TC[t].TCBindCurVPE = TCBindCurVPE then
            if (TC[t].TCStatusDA = 1)
                and (TC[t].TCHaltH = 0)
                and (TC[t].TCStatusA = 0)
                and (success = 0) then
                    TC[t].TCRestart ← GPR[rs]
                    TC[t].GPR[rd] ← GPR[rt]
                    if (Config3ULRI = 1) then
                        TC[t].UserLocal ← UserLocal
                    endif
                    activated ← 1
                    priorcu ← TC[t].TCStatusTCU3..TCU0
                    priormx ← TC[t].TCStatusTMX
                    priorixmt ← TC[t].TCStatusIXMT
                    TC[t].TCStatus = priorcu || priormx || StatusFR || 05 || 1 ||
ImpDep4
                                    || 1 || 0 || activated|| StatusKSU || priorixmt
                                    || 02 || TCStatusTASID
                    success ←1
            endif
        endif
    endfor
    if success = 0
        VPEControlEXCPT ← 1
        SignalException(Thread)
```

```
        endif
    else
        SignalException(ReservedInstruction)
    endif
```

Exceptions:

Reserved Instruction
Thread

MFTR — Move from Thread Context



Format: `MFTR rt, rs, u, sel, h` **MIPS MT**

See also the *Idiom(s)* column of Table 5.1.

Purpose: Move from Thread Context

To move the contents of a register within a targeted thread context or VPE into a general register of the current thread.

Description: `GPR[rt] ← TC[VPEControl`$_{\texttt{TargTC}}$`[u,rs,sel,h]`

The contents of the register specified are loaded into general register *rt*. The target context to be read is determined by the value of the TargTC field of the CP0 *VPEControl* register (see Section 6.5). The register to be read within the selected context is determined by the value in the *rs* operand register, in conjunction with the *u* and *sel* bits of the MFTR instruction, according to table Table 5.1. If the register to be read is instantiated per-processor or per-VPE, rather than per-TC, the register selected is that of the processor within which the target TC is instantiated, or the VPE to which the target TC is bound (see Section 6.13), respectively.

Coprocessor 1 and 2 registers and DSP accumulators referenced by the MFTR instruction are those bound to the target TC. The *TCUx* bits and *TMX* bit of the target TC's *TCStatus* register are ignored.

If the selected register is not implemented on the processor, or otherwise not accessible to the TC that issued the MFTR, as in the case of references to TCs and coprocessor resources bound to other VPEs when the VPE executing the MFTR does not have MVP set in *VPConfig0*, the resulting *rt* value is -1.

Release 5 adds the instruction MFTHC0.

The *Idiom(s)* column in Table 5.1 specifies the assembler idiom that is used to express an access to the particular register.

Table 5.1 MFTR Source Decode

u Value	*sel* Value	Register Selected	Idiom(s)
0	n	Coprocessor 0 Register number *rs*, sel = *sel* h=0 signifies MFTC0, while h=1 signifies MFTHC0	MFTC0 rt, rs MFTHC0 rt, rs (Release 5)
			MFTC0 rt, rs, sel MFTHC0 rt, rs, sel (Release 5)
1	0	GPR[*rs*]	MFTGPR rt, rs

Table 5.1 MFTR Source Decode (Continued)

u Value	*sel* Value	Register Selected		Idiom(s)
1	1	*rs* Value	Selection	
		0	Lo Register / Lo component of DSP Accumulator 0	MFTLO rt
				MFTLO rs, ac0
		1	Hi Register / Hi component of DSP Accumulator 0	MFTHI rt
				MFTHI rt, ac0
		2	ACX Register / ACX component of Accumulator 0	MFTACX rt
				MFTACX rt, ac0
		4	Lo component of DSP Accumulator 1	MFTLO rt, ac1
		5	Hi component of DSP Accumulator 1	MFTHI rt, ac1
		6	Reserved for ACX of DSP Accumulator 1	MFTACX rt, ac1
		8	Lo component of DSP Accumulator 2	MFTLO rt, ac2
		9	Hi component of DSP Accumulator 2	MFTHI rt, ac2
		10	Reserved for ACX of DSP Accumulator 2	MFTACX rt, ac2
		12	Lo component of DSP Accumulator 3	MFTLO rt, ac3
		13	Hi component of DSP Accumulator 3	MFTHI rt, ac3
		14	Reserved for ACX of DSP Accumulator 3	MFTACX rt, ac3
		16	DSPControl register	MFTDSP rt
		Other Values of *rs*, Reserved, Unpredictable		
1	2	FPR[*rs*]		MFTC1 rt, ft
				MFTHC1 rt, ft
1	3	FPCR[*rs*]		CFTC1 rt, ft
1	4	Cop2 Data[*n*], where *n* is composed by concatenating *rx* with *rs*, with *rx* providing the most significant bits.		
1	5	Cop2 Control[*n*], where *n* is composed by concatenating *rx* with *rs*, with *rx* providing the most significant bits.		
1	>5	Reserved, Unpredictable		

The selected value is written into the target register *rt*. If the precision of the source register is less than the precision of the target GPR, the value is sign-extended.

The *h* bit of the instruction word selects the high-order half of the source register in instances where the source is a register of greater precision than the target GPR.

Restrictions:

An MFTR instruction where the target TC is not in a Halted state (i.e., *TCHalt.H* is not set), or where a TC other than the one issuing the MFTR is active in the target VPE on a reference to a per-VPE CP0 register, may result in an UNSTABLE value.

If the target TC is blocked but not halted, then the thread issuing the MFTR instruction may be blocked indefinitely. This is due to the target TC waiting on an external event that may never happen. It is recommended that the *TCStatus*$_{RNST}$ bit of the target TC be checked before issuing the MFTR instruction.

If access to Coprocessor 0 is not enabled, a Coprocessor Unusable Exception is signaled.

In any implementation with Floating-Point-Unit, if *TCStatus*$_{TFR}$ is set so the effective FPR width matches the GPR width, a MFTR instruction targetting one of the FPRs with h=1 will cause UNPREDICTABLE results.

Operation:

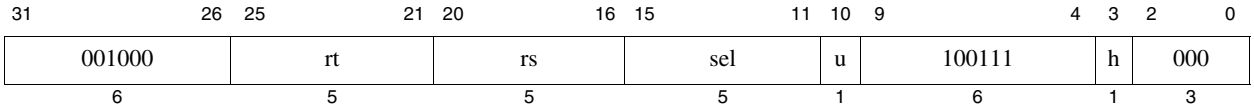
```
if IsCoprocessorEnabled(0) then
    if VPEConf0MVP = 0 and ( TC[VPEControlTargTC].TCBindCurVPE ≠ TCBindCurVPE ) then
        data ← -1
    else if VPEControlTargTC > MVPConf0PTC then
        data ← -1
    else if u = 0 then
        data ← TC[VPEControlTargTC].CPR[0,rs,sel]
    else
        if h = 1 then
            topbit ← 63
            bottombit ← 32
        else
            topbit ← 31
            bottombit ← 0
        endif
        case sel
            0: data ← TC[VPEControlTargTC].GPR[rs]topbit..bottombit
            1: case rs
                0: data ← TC[VPEControlTargTC].Lo
                1: data ← TC[VPEControlTargTC].Hi
                2: data ← TC[VPEControlTargTC].ACX
                4: data ← TC[VPEControlTargTC].DSPLo[1]
                5: data ← TC[VPEControlTargTC].DSPHi[1]
                6: data ← TC[VPEControlTargTC].DSPACX[1]
                8: data ← TC[VPEControlTargTC].DSPLo[2]
                9: data ← TC[VPEControlTargTC].DSPHI[2]
                10:data ← TC[VPEControlTargTC].DSPACX[2]
                12 data ← TC[VPEControlTargTC].DSPLo[3]
                13:data ← TC[VPEControlTargTC].DSPHi[3]
                14:data ← TC[VPEControlTargTC].DSPACX[3]
                16:data ← TC[VPEControlTargTC].DSPControl
                otherwise: data ← UNPREDICTABLE
                2: if ( ( ConfigAT = 0 and StatusFR = 0) or
                        ( ConfigAT = 1 or ConfigAT = 2) )
                       // GPR and FPR widths match
                       if (h = 0 )
                           data ← TC[VPEControlTargTC].FPR[rs]
                       else
                           UNPREDICTABLE
                       endif
                   elseif (ConfigAT = 0 and StatusFR = 1)
                       // 32-bit GPRs and 64-bit FPRs
                       data ← TC[VPEControlTargTC].FPR[rs]topbit..bottombit
                   endif
            3: data ← TC[VPEControlTargTC].FPCR[rs]
            4: data ← TC[VPEControlTargTC].CP2CPR[rx||rs]topbit..bottombit
            5: data ← TC[VPEControlTargTC].CP2CCR[rx||rs]topbit..bottombit
            otherwise: data ← UNPREDICTABLE
    endif
```

```
        if h = 1 then
            data ← data63..32
        endif
        GPR[rt] ← data31..0
    else
        SignalException(CoprocessorUnusable, 0)
    endif
```

Exceptions:

Coprocessor Unusable
Reserved Instruction

MTTR — Move to Thread Context



Format: `MTTR rt, rs, u, sel, h` **MIPS MT**

See also *Idiom(s)* column of Table 5.2.

Purpose: Move to Thread Context

To move the contents of a general register of the current thread into a register within a targeted thread context.

Description: $\texttt{TC[VPEControl}_{\texttt{TargTC}}\texttt{][u,rs,sel,h]} \leftarrow \texttt{GPR[rt]}$

The contents of the *rt* register specified are written into a register of an arbitrary thread context (TC) or virtual processor (VPE).

The target context to be written is determined by the value of the *TargTC* field of the CP0 *VPEControl* register (see Section 6.5). The register to be written within the selected context is determined by the value in the *rs* operand register, in conjunction with the *u* and *sel* bits of the MTTR instruction, according to Table 5.2. If the register to be written is instantiated per-processor or per-VPE, rather than per-TC, the register selected is that of the processor within which the target TC is instantiated, or the VPE to which the target TC is bound (see "6.13 TCBind Register (CP0 Register 2, Select 2)" on page 81), respectively.

Coprocessor 1 and 2 registers and DSP accumulators referenced by the MTTR instruction are those bound to the target TC. The *TCUx* bits and *TMX* bit of the target TC's *TCStatus* register are ignored.

If the selected register is not implemented on the processor, or otherwise not accessible to the TC issuing the MTTR, as in the case of references to TCs and coprocessor resources bound to other VPEs when the VPE executing the MTTR does not have *MVP* set in *VPConfig0*, MTTR has no effect.

Release 5 adds the MTTHC0 instruction.

The *Idiom(s)* column in Table 5.2 specifies the assembler idiom that is used to express an access to the particular register.

Table 5.2 MTTR Destination Decode

u Value	*sel* Value	Register Selected	Idiom(s)
0	n	Coprocessor 0 Register number *rs*, sel = *sel* h=0 signifies MTTC0, while h=1 signifies MTTHC0	MTTC0 rt, rs MTTHC0 rt, rs (Release 5)
			MTTC0 rt, rs, sel MTTHC0 rt, rs, sel (Release 5)
1	0	GPR[*rt*]	MTTGPR rt, rs

Table 5.2 MTTR Destination Decode (Continued)

u Value	*sel* Value	Register Selected		Idiom(s)
1	1	rs Value	Selection	
		0	Lo Register / Lo component of DSP Accumulator 0	MTTLO rt
				MTTLO rt, ac0
		1	Hi Register / Hi component of DSP Accumulator 0	MTTHI rt
				MTTHI rt, ac0
		2	ACX Register / ACX component of Accumulator 0	MTTACX rt
				MTTACX rt ac0
		4	Lo component of DSP Accumulator 1	MTTLO rt, ac1
		5	Hi component of DSP Accumulator 1	MTTHI rt, ac1
		6	Reserved for ACX of DSP Accumulator 1	MTTACX rt, ac1
		8	Lo component of DSP Accumulator 2	MTTLO rt, ac2
		9	Hi component of DSP Accumulator 2	MTTHI rt, ac2
		10	Reserved for ACX of DSP Accumulator 2	MTTACX rt, ac2
		12	Lo component of DSP Accumulator 3	MTTLO rt, ac3
		13	Hi component of DSP Accumulator 3	MTTHI rt, ac3
		14	Reserved for ACX of DSP Accumulator 3	MTTACX rt, ac3
		16	DSPControl register	MTTDSP rt
		Other Values of *rs*, Reserved		
1	2	FPR[rs]		MTTC1 rt, ft
				MTTHC1 rt, ft
1	3	FPCR[rs]		CTTC1 rt, ft
1	4	Cop2 Data[*n*], where *n* is composed by concatenating *rx* with *rs*, with *rx* providing the most significant bits.		
1	5	Cop2 Control[*n*], where *n* is composed by concatenating *rx* with *rs*, with *rx* providing the most significant bits.		
1	>5	Reserved		

The *h* bit of the instruction word selects the high-order half of the target register in instances where the target is a register of greater precision than the source GPR. The source value is not sign-extended on an MTTR operation.

Restrictions:

The effect on a TC that is not in a Halted state (i.e., *TCHalt.H* is 0) of an MTTR instruction targeting that TC may be transient and unstable, but MTTRs setting a *TCHalt H* bit are always effective until overridden by another MTTR.

Processor state following an MTTR instruction modifying a per-VPE CP0 register is UNPREDICTABLE if a TC other than the one issuing the MTTR is concurrently active on the targeted VPE.

If the target TC is blocked but not halted, then the thread issuing the MTTR instrution may be blocked indefinitely.

This is due to the target TC waiting on an external event that may never happen. It is recommended that the $TCStatus_{RNST}$ bit of the target TC be checked before issuing the MTTR instruction.

If access to Coprocessor 0 is not enabled, a Coprocessor Unusable Exception is signaled.

If the target register is a Floating-Point Control register, hardware must not generate a Floating-Point Exception due to any value written into Floating Point Exception Cause Field by the MTTR instruction.

In any implementation with Floating-Point-Unit, if $TCStatus_{TFR}$ is set so the effective FPR width matches the GPR width, a MFTR instruction targeting one of the FPRs with h=1 will cause UNPREDICTABLE results.

Operation:

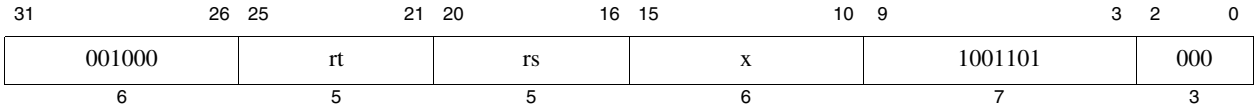
```
if IsCoprocessorEnabled(0) then
    if VPEConf0MVP = 0 and ( TC[VPEControlTargTC].TCBindCurVPE ≠ TCBindCurVPE ) then
        NOOP
    else if VPEControlTargTC > MVPConf0PTC then
        NOOP
    else
        if h = 1 then
            topbit ← 63
            bottombit ← 32
        else
            topbit ← 31
            bottombit ← 0
        endif
        if u = 0 then
            TC[VPEControlTargTC].CPR[0,rs,sel]topbit..bottombit ← GPR[rt]
        else
            case sel
                0: TC[VPEControlTargTC].GPR[rs] ← GPR[rt]topbit..bottombit
                1: case rs
                    0: TC[VPEControlTargTC].Lo ← GPR[rt]
                    1: TC[VPEControlTargTC].Hi ← GPR[rt]
                    2: TC[VPEControlTargTC].ACX ← GPR[rt]
                    4: TC[VPEControlTargTC].DSPLo[1] ← GPR[rt]
                    5: TC[VPEControlTargTC].DSPHi[1] ← GPR[rt]
                    6: TC[VPEControlTargTC].DSPACX[1] ← GPR[rt]
                    8: TC[VPEControlTargTC].DSPLo[2] ← GPR[rt]
                    9: TC[VPEControlTargTC].DSPHi[2] ← GPR[rt]
                    10:TC[VPEControlTargTC].DSPACX[2] ← GPR[rt]
                    12:TC[VPEControlTargTC].DSPLo[3] ← GPR[rt]
                    13:TC[VPEControlTargTC].DSPHi[3] ← GPR[rt]
                    14:TC[VPEControlTargTC].DSPACX[3] ← GPR[rt]
                    16:TC[VPEControlTargTC].DSPControl ← GPR[rt]
                    otherwise: UNPREDICTABLE
                2: if ( ( ConfigAT = 0 and StatusFR = 0) or
                         ( ConfigAT = 1 or ConfigAT = 2) )
                        // GPR and FPR widths match
                        if (h = 0)
                            TC[VPEControlTargTC].FPR[rs] ← GPR[rt]
                        else
                            UNPREDICTABLE
                        endif
                    elseif (ConfigAT = 0 and StatusFR = 1)
                        // 32-bit GPRs and 64-bit FPRs
                        TC[VPEControlTargTC].FPR[rs]topbit..bottombit ← GPR[rt]
                    endif
                3: TC[VPEControlTargTC].FPCR[rs] ← GPR[rt]
                4: TC[VPEControlTargTC].CP2CPR[rx||rs]topbit..bottombit ← GPR[rt]
```

```
                5: TC[VPEControl_TargTC].CP2CCR[rx||rs]_topbit..bottombit ← GPR[rt]
                otherwise: UNPREDICTABLE
            endif
        endif
    else
        SignalException(CoprocessorUnusable, 0)
    endif
```

Exceptions:

Coprocessor Unusable
Reserved Instruction

YIELD — Conditionally Deschedule or Deallocate the Current Thread



Format: `YIELD rt, rs` **MIPS MT**

`yield rs`

Purpose: Conditionally Deschedule or Deallocate the Current Thread

To suspend the current thread of execution, and conditionally deallocate the associated thread context.

Description:

The YIELD instruction takes a single input operand value from a GPR *rs*. This value is a descriptor of the circumstances under which the issuing thread should be rescheduled.

If GPR *rs* is zero, the thread is not to be rescheduled at all, and it is instead deallocated and its associated TC storage freed for allocation by a subsequent FORK issued by some other thread.

If GPR *rs* is negative one (-1), the thread remains eligible for scheduling at the next opportunity, but invokes the processor's scheduling logic and relinquishes the CPU for any other threads which ought to execute first according to the implemented scheduling policy.

If GPR *rs* is negative two (-2), the processor's scheduling logic is not invoked, and the only effect of the instruction is to retrieve the *rt* value (see below).

All other negative values of the *rs* register are reserved for future architectural definition by MIPS.

Positive values of *rs* are treated as a vector of *YIELD qualifier* (YQ) bits which describe an implementation-dependent set of external or internal core signal conditions under which the YIELDing thread is to be rescheduled. Up to 31 bits of YIELD qualifier state may be supported by a processor, but implementations may provide fewer. To be usable, a YIELD qualifier bit must be enabled in the *YQMask* register (see Section 6.8).

If no set bit of *rs* matches with a set, enabled *YQ* bit, the TC is blocked until one or more active bits of enabled *YQ* input match corresponding *rs* bits. If and when one or more bits match, the TC resumes a running state, and may be rescheduled for execution in accordance with the thread scheduling policy in effect.

The *rt* output operand specifies a GPR which is to receive a result value. This result contains the bit vector of *YQ* inputs values enabled by the *YQMask* register at the time the YIELD completes. Thus, any *YQ* state that can be waited upon by a YIELD with a positive *rs* value can also be polled via a YIELD with an *rs* value of -1 or -2. The value of any *rt* bits that do not correspond to set bits in the *YQMask* register is implementation-dependent, typically 0. A zero value of the *rt* operand field, selecting GPR 0, indicates that no result value is desired.

Implementation Notes:

The writeback of the destination register should be scheduled only when it is known that the YIELD is not blocked. Accesses to the register via MTTR or MFTR targeting a TC blocked on a YIELD should not be blocked by a dependency on the YIELD completion.

Restrictions:

Bits 15:10 must be set to 0 by software. Hardware must ignore these bits.

If a positive *rs* value includes a set bit that is not also set in the *YQMask* register, a Thread exception is raised for the YIELD instruction, with the *EXCPT* field of the *VPEControl* register set to 2 to indicate the Invalid Qualifier case.

If no non-halted dynamically allocatable TC would be activated after a YIELD whose *rs* value is 0, a Thread exception is raised for the YIELD instruction, with the *EXCPT* field of the *VPEControl* register set to 0 to indicate the Thread Underflow case.

If the processor's scheduling logic would be invoked as a consequence of an otherwise unexceptional YIELD, one whose *rs* value is 0 (excluding the Underflow case), -1, or positive (excluding the Invalid Qualifier case), and both the

YSI bit of *VPEControl* and the *DT* bit of *TCStatus* are set, a Thread exception is raised for the YIELD instruction, with the *VPEControl EXCPT* field set to 4 to indicate the YIELD Scheduler case.

If multi-threaded operation is unsupported, a Reserved Instruction Exception is raised for the YIELD instruction.

Processor behavior is UNPREDICTABLE if a YIELD instruction is placed in a branch or jump delay slot.

Operation:

```
if Config3MT = 1 then
    if GPR[rs] = 0 then
        ok ← 0
        for t in 0...MVPConf0PTC
            if (TC[t].TCBindCurVPE = TCBindCurVPE )
             and (TC[t].TCBindCurTC ≠ TCBindCurTC )
             and (TC[t].TCStatusDA = 1)
             and (TC[t].TCHaltH = 0)
             and (TC[t].TCStatusA = 1) then
                ok ← 1
            endif
        endfor
        if ((ok = 1) and not ((VPEControlYSI = 1) and (TCStatusDT = 1))) then
            TCStatusA ← 0
        else
            VPEControlEXCPT ← 0
            SignalException(Thread)
        endif
    else if GPR[rs] > 0 then
        if (GPR[rs] and (not YQMask)) ≠ 0 then
            VPEControlEXCPT ← 2
            SignalException(Thread)
        else
            SetThreadRescheduleCondition(GPR[rs] and YQMask)
        endif
    endif
    if GPR[rs] ≠ -2 then
        if (VPEControlYSI = 1) and (TCStatusDT = 1) then
            VPEControlEXCPT ← 4
            SignalException(Thread)
        else
            ScheduleOtherThreads()
        endif
    endif
    if rt ≠ 0 then
        GPR[rt] ← GetThreadRescheduleCondition()
    endif
else
    SignalException(ReservedInstruction)
endif
```

Exceptions:

Reserved Instruction
Thread

Chapter 6

MIPS® MT Privileged Resource Architecture

6.1 Privileged Resource Architecture for MIPS® MT

Table 6.1 summarizes the system coprocessor privileged resources associated with the MIPS MT Module.

Table 6.1 MIPS® MT PRA

Register Name	New or Modified	CP0 Register Number	Register Select Number	Description
MVPControl	New	0	1	**Per-Processor** register containing global MIPS MT configuration data. See Section 6.2.
MVPConf0	New	0	2	**Per-Processor** multi-VPE dynamic configuration information. See Section 6.3.
MVPConf1	New	0	3	Optional **Per-Processor** multi-VPE dynamic configuration information. See Section 6.4
VPEControl	New	1	1	**Per-VPE** register containing relatively volatile thread configuration data. See Section 6.5.
VPEConf0	New	1	2	**Per-VPE** multi-thread configuration information. See Section 6.6.
VPEConf1	New	1	3	**Per-VPE** multi-thread configuration information. See Section 6.7.
YQMask	New	1	4	**Per-VPE** register defining which YIELD qualifier bits may be used without generating an exception. See Section 6.8
VPESchedule	New	1	5	Optional **Per-VPE** register to manage scheduling of a VPE within a processor. See Section 6.9.
VPEScheFBack	New	1	6	Optional **Per-VPE** register to provide scheduling feedback to software. See Section 6.10.
VPEOpt	New	1	7	Optional **Per-VPE** register to provide control over optional features, such as cache partitioning control. See Section 6.11
TCStatus	New	2	1	**Per-TC** status information, includes copies of thread-specific bits of Status and EntryHi registers. See Section 6.12
TCBind	New	2	2	**Per-TC** information about TC ID and VPE binding. See Section 6.13
TCRestart	New	2	3	**Per-TC** value of restart instruction address for the associated thread of execution. See Section 6.14
TCHalt	New	2	4	**Per-TC** register controlling Halt state of TC. See Section 6.15.

Table 6.1 MIPS® MT PRA (Continued)

Register Name	New or Modified	CP0 Register Number	Register Select Number	Description
TCContext	New	2	5	**Per-TC** Read/Write Storage for OS use. See Section 6.16.
TCSchedule	New	2	6	Optional **Per-TC** register to manage scheduling of a TC. See Section 6.17.
TCScheFBack	New	2	7	Optional **Per-TC** register to provide scheduling feedback to software. See Section 6.18.
TCOpt	New	3	7	Optional **Per-TC** register to provide control over optional features, such as cache partitioning control. See Section 6.19
SRSConf0	New	6	1	**Per-VPE** register indicating and optionally controlling shadow register set configuration. See Section 6.20.
SRSConf1	New	6	2	Optional **Per-VPE** register indicating and optionally controlling shadow register set configuration. See Section 6.21.
SRSConf2	New	6	3	Optional **Per-VPE** register indicating and optionally controlling shadow register set configuration. See Section 6.22.
SRSConf3	New	6	4	Optional **Per-VPE** register indicating and optionally controlling shadow register set configuration. See Section 6.23.
SRSConf4	New	6	5	Optional **Per-VPE** register indicating and optionally controlling shadow register set configuration. See Section 6.24.
SRSCtl	Modified	12	2	Previously hard-wired field now optionally “soft”, and a function of the SRSConf registers. See Section 6.20.
Cause	Modified	13	0	New Cause code. See Section 6.25.2.
EBase	Modified	15	1	Distinct *CPUNum* value required per VPE. See Section 6.25.5.
Config3	Modified	16	3	Fields added to describe and control MT Module configuration. See Section 6.25.7.
Debug	Modified	23	0	Register accessed by MFTR/MTTR as being per-TC, with distinct *SSt* and *OffLine* values. See Sections 6.25.4 and 10.2.

6.2 MVPControl Register (CP0 Register 0, Select 1)

Compliance Level: *Required for MIPS MT.*

The *MVPControl* register is instantiated per-processor, and provides an interface for global control and configuration of a multi-VPE MIPS MT core.

Figure 6.1 shows the format of the *MVPControl* register; Table 6.2 describes the *MVPControl* register fields.

Figure 6.1 MVPControl Register Format

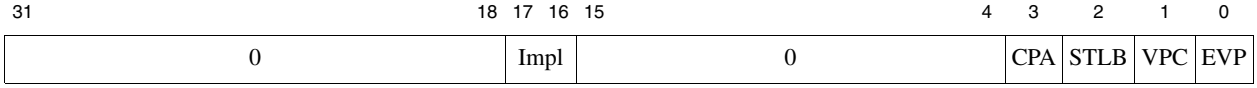


Table 6.2 MVPControl Register Field Descriptions

Fields		Description	Read/Write		Reset State	Compliance
Name	Bits		MVP=0	MVP=1		
0	31:18	Must be written as zero; return zero on read.	0		0	Reserved
Impl	17:16	This field is reserved for implementations. Refer to the processor specification for the format and definition of this field.			Undefined	Optional
0	15:4	Must be written as zero; return zero on read.	0		0	Reserved
CPA	3	Cache Partitioning Active. If set, the *IWX* and *DWX* fields of the *VPEOpt* register and/or the *IWX* and *DWX* fields of the *TCOpt* register control the allocation of cache lines as described in Sections 6.11 amd 6.19. If clear, the *IWX* and *DWX* fields of both registers are ignored.	R	R/W	0	Optional
STLB	2	Share TLBs. Modifiable only if the VPC bit was set prior to the write to the register of a new value. When set, the full complement of TLBs of a processor is shared by all VPEs on the processor having access to the TLB, regardless of the programming of the *Config1 MMU_Size* register fields. When STLB is set: • The virtual address and ASID spaces are unified across all VPEs sharing the TLB. • The TLB logic must ensure that a TLBWR instruction can never write to a TLB entry which corresponds to the valid Index register value of any VPE sharing the TLB. • TLBWRs may have UNPREDICTABLE results if there are fewer total unwired TLB entries than there are operational VPEs sharing the TLB. • TLBWRs may have UNPREDICTABLE results if the Wired register values are not identical across all VPEs sharing the TLB. • If Segmentation Control is used, all of the SegCtl registers to be programmed identically across the VPEs. When not in use for TLB maintenance, software should leave the *Index* register set to an invalid value, with the *P* bit set, for all VPEs having TLB access.	R if VPC = 0, R/W if VPC = 1		0	Optional

Table 6.2 MVPControl Register Field Descriptions (Continued)

Fields		Description	Read/Write		Reset State	Compliance
Name	Bits		MVP=0	MVP=1		
VPC	1	Indicates that Processor is in a VPE Configuration State. When *VPC* is set, some normally "Preset" configuration register fields become writable, to allow for dynamic configuration of processor resources (See Section 8.2). Writable by software only if the *VPEConf0 MVP* bit is set for the VPE issuing the modifying instruction. Processor behavior is UNDEFINED if *VPC* and *EVP* are both in a set state at the same time.	R	R/W	0	Required if run-time VPE configuration supported
EVP	0	Enable Virtual Processors. Modifiable only if the *VPEConf0 MVP* bit is set for the VPE issuing the modifying instruction. Set by EVPE instruction and cleared by DVPE instruction. If set, all activated VPEs (see Section 6.6) on a processor fetch and execute independently. If cleared, only a single instruction stream on a single VPE can run.	R	R/W	0	Required

So long as the *EVP* bit is zero, no thread scheduling will be performed by the processor. On a processor reset, only the reset thread, TC 0, will execute. If *EVP* is cleared by software, only the thread which issued the DVPE or MTC0 instruction which cleared the bit will issue further instructions. All other TCs of the processor are suspended (see Section 3.6).

The effect of clearing *EVP* in software may not be instantaneous. An instruction hazard barrier, e.g., JR.HB, is required to guarantee that all other VPEs have been quiesced.

The *STLB* bit affects only VPEs using a TLB MMU. The operation of VPEs using FMT MMUs is unaffected.

For nanoMIPS32-compatible software operation, all *MMU_Size* fields must indicate the size of the shared TLB when *STLB* is set. This may either be done automatically by hardware, or, on processors implementing configurable *MMU_Size*, by software rewriting the *MMU_Size* fields of the *Config1* registers of the affected VPEs to the correct value while the processor has the VPC bit set. When *STLB* is set, the restriction that the sum of *Config1 MMU_Size* fields not exceed the total number of configurable TLB entry pairs as indicated by the *PTLBE* field of the *MVPConf0* register no longer applies. If TLB entries are not otherwise dynamically configurable, i.e., *PTLBE* is zero, hardware must automatically maintain the correct *MMU_Size* values according to the value of *STLB*.

Programming Notes

The TLB should always be flushed of valid entries between any setting or clearing of *STLB* and the first subsequent TLB-mapped memory reference.

6.3 MVPConf0 Register (CP0 Register 0, Select 2)

Compliance Level: *Required.*

The *MVPConf0* Register is instantiated per-processor. It contains configuration information for dynamic multi-VPE processor configuration. All fields in the *MVPConf0* register are read-only.

Figure 6.2 shows the format of the *MVPConf0* register; Table 6.3 describes the *MVPConf0* register fields.

Figure 6.2 MVPConf0 Register Format

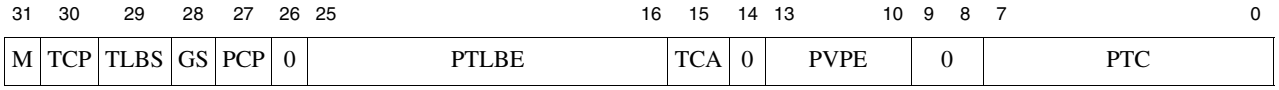


Table 6.3 MVPConf0 Register Field Descriptions

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
M	31	This bit indicates that a *MVPConf1* register (see Section 6.4) is present. If the *MVPConf1* register is not implemented, this bit should read as a 0. If the *MVPConf1* register is implemented, this bit should read as a 1.	R	Preset by hardware	Required
TCP	30	Programmable Cache Partitioning per TC. If set, indicates that the allocation behavior of the "ways" of the primary instruction and data caches can be controlled via the *TCOpt* register's *IWX* and *DWX* fields. See Section 6.19.	R	Preset by hardware	Required
TLBS	29	TLB Sharable. If set, indicates that TLB sharing amongst all VPEs of a VMP is possible. TLB sharing is enabled by the *STLB* bit of the *MVPControl* register. See Section 6.2.	R	Preset by hardware	Required
GS	28	Gating Storage Support present. If set, indicates that the processor is configured to support gating storage operations. See Section 9.1.	R	Preset by hardware	Required
PCP	27	Programmable Cache Partitioning per VPE. If set, indicates that the allocation behavior of the "ways" of the primary instruction and data caches can be controlled via the *VPEOpt* register's *IWX* and *DWX* fields. See Section 6.11.	R	Preset by hardware	Required
PTLBE	25:16	Total processor complement of allocatable TLB entry pairs. See Section 8.2. If TLB configuration is fixed, *PTLBE* is zero.	R	Preset by hardware	Required
TCA	15	TCs Allocatable. If set, TCs may be assigned to VPEs by writing the *CurVPE* field of each TC's *TCBind* register while the *VPC* bit of *MVPControl* is set. See Section 6.13.	R	Preset by hardware	Required
PVPE	13:10	Total processor complement of VPE contexts - 1. Valid VPE numbers are from 0 to *PVPE*, inclusive.	R	Preset by hardware	Required
PTC	7:0	Total processor complement of TCs - 1. Valid TC numbers are from zero to *PTC*, inclusive.	R	Preset by hardware	Required
0	30, 26, 14, 9:8	Reserved. Reads as zero, must be written as zero.	R	0	Reserved

6.4 MVPConf1 Register (CP0 Register 0, Select 3)

Compliance Level: *Optional*.

The *MVPConf1* register is optionally instantiated per processor. It indicates the coprocessor and UDI resources available for dynamic allocation to VPEs. All fields in the *MVPConf1* register are read-only.

Figure 6.3 shows the format of the *MVPConf1* register; Table 6.4 describes the *MVPConf1* register fields.

Figure 6.3 MVPConf1 Register Format

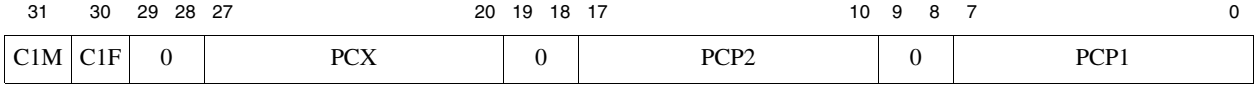


Table 6.4 MVPConf1 Register Field Descriptions

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
C1M	31	Allocatable CP1 coprocessors are media-extension capable	R	Preset by hardware	Required
C1F	30	Allocatable CP1 coprocessors are floating-point capable	R	Preset by hardware	Required
PCX	27:20	Total processor complement of CorExtend™ UDI state instantiations available, for UDI blocks with persistent state.	R	Preset by hardware	Required
PCP2	17:10	Total processor complement of integrated and allocatable Coprocessor 2 contexts	R	Preset by hardware	Required
PCP1	7:0	Total processor complement of integrated and allocatable FP/MDMX Coprocessors contexts	R	Preset by hardware	Required
0	29:28, 19:18, 9:8	Reserved. Reads as zero, must be written as zero.	R	0	Reserved

Allocatable resources can be bound to specific VPEs, as described in Section 8.2.

6.5 VPEControl Register (CP0 Register 1, Select 1)

Compliance Level: *Required for MIPS MT.*

The *VPEControl* register is instantiated per VPE as part of the system coprocessor.

Figure 6.4 shows the format of the *VPEControl* register; Table 6.5 describes the *VPEControl* register fields.

Figure 6.4 VPEControl Register Format

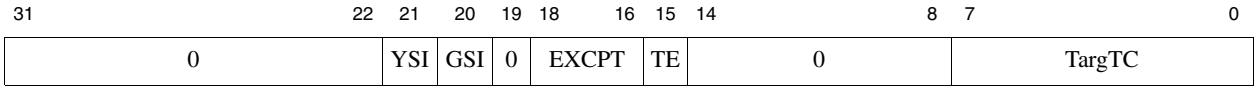


Table 6.5 VPEControl Register Field Descriptions

Fields		Description			Read / Write	Reset State	Compliance
Name	Bits						
YSI	21	YIELD Scheduler Intercept. If set, and the *TCStatus DT* bit is also set, valid YIELD instructions that could otherwise cause a rescheduling cause a Thread exception with a YIELD Scheduler Exception sub-code (see below).			R/W	0	Required
GSI	20	Gating Storage Scheduler Intercept. If set, and the *TCStatus DT* bit is also set, Gating Storage load and store operations that would otherwise block the issuing TC cause a Thread exception with a GS Scheduler Exception sub-code (see below).			R/W	0	Required
EXCPT	18:16	Exception sub-code of most recently dispatched Thread exception	**Value**	**Meaning**	R	Undefined	Required
			0	Thread Underflow			
			1	Thread Overflow			
			2	Invalid YIELD Qualifier			
			3	Gating Storage Exception			
			4	YIELD Scheduler Exception			
			5	GS Scheduler Exception			
			6-7	Reserved			
TE	15	Threads Enabled. Set by EMT instruction, cleared by DMT instruction. If set, multiple TCs may be simultaneously active. If cleared, only one thread may execute on the VPE.			R/W	0	Required
TargTC	7:0	TC number to be used on MTTR and MFTR instructions.			R/W	Undefined	Required
0	31:22, 19,14:8	Must be written as zero; return zero on read.			0	0	Reserved

So long as the *TE* bit is zero, no thread scheduling will be performed by the VPE. On a processor reset, only the reset thread, TC 0, will execute. If *TE* is cleared by software, only the thread which issued the DMT or MTC0 instruction which cleared the bit will issue further instructions. All other TCs of the VPE are suspended (see Section 3.6).

The effect of clearing *TE* in software may not be instantaneous. An instruction hazard barrier, e.g., JR.HB, is required to guarantee that all other threads have been quiesced.

6.6 VPEConf0 Register(CP0 Register 1, Select 2)

Compliance Level: *Required for MIPS MT.*

The *VPEConf0* register is instantiated per VPE. It indicates the activation state and privilege level of the VPE. All fields in the *VPEConf0* register are read-only in normal execution, but the *MVP* and *VPA* fields are writable while the *MVP* bit is set for the VPE performing the modification.

Figure 6.5 shows the format of the *VPEConf0* register; Table 6.6 describes the *VPEConf0* register fields.

Figure 6.5 VPEConf0 Register Format

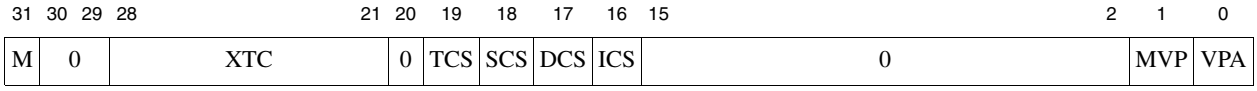


Table 6.6 VPEConf0 Register Field Descriptions

Fields		Description	Read/Write		Reset State	Compliance
Name	Bits		MVP=0	MVP=1		
M	31	This bit is reserved to indicate that a *VPEConf1* register is present. If the *VPEConf1* register is not implemented, this bit should read as a 0. If the *VPEConf1* register is implemented, this bit should read as a 1.	R		Preset by hardware	Required
XTC	28:21	Exclusive TC. Set by hardware when execution is restricted within a VPE to a single TC, due to *EXL*/*ERL* being set in the *Status* register, or *TE* being cleared in the *VPEControl* register, this field contains the TC number of the TC eligible to run. Read by hardware when the *VPA* bit is written set by software. For cross-VPE initialization, *XTC* is writable by MTTR if the issuing VPE has *MVP* set **and** the target VPE has *VPA* clear.	R	R/W (if *VPA* not set for target)	0 for VPE 0, Undefined for all others	Required
TCS	19	Tertiary Cache Shared. Indicates that the tertiary cache described in the *Config2* register is shared with at least one other VPE.	R		Preset by hardware	Required
SCS	18	Secondary Cache Shared. Indicates that the secondary cache described in the *Config2* register is shared with at least one other VPE.	R		Preset by hardware	Required
DCS	17	Data Cache Shared. Indicates that the primary data cache described in the *Config1* register is shared with at least one other VPE.	R		Preset by hardware	Required
ICS	16	Instruction Cache Shared. Indicates that the primary instruction cache described in the *Config1* register is shared with at least one other VPE.	R		Preset by hardware	Required
MVP	1	Master Virtual Processor. If set, the VPE can access the registers of other VPEs of the same VMP, using MTTR/MFTR, and can modify the contents of the *MVPControl* and *VPEConf0* registers, thus acquiring the capability to manipulate and configure other VPEs sharing the same processor (see Section 8.2).	R	R/W	1 for VPE 0, 0 for all others	Required
VPA	0	Virtual Processor Activated. If set, the VPE will schedule threads and execute instructions so long as the *EVP* bit of the *MVPControl* register enables multi-VPE execution.	R	R/W	1 for VPE 0, 0 for all others	Required
0	30:29, 20, 15:2	Reserved. Reads as zero, must be written as zero.	R		0	Reserved

The *XTC* field is set by hardware on an exception setting *EXL* or *ERL* of the *Status* register, or on an MTC0 or DMT instruction clearing the *TE* bit of *VPEControl*. It may be set by software if and only if both *MVP* of the writing VPE is set and *VPA* of the written VPE is clear, which implies a cross-VPE MTTR operation. It is read by hardware when *VPA* is set, and if the initial state of the VPE is such that only one activated TC may issue, i.e., if *EXL* or *ERL* are set, or *TE* is clear, the TC designated by the *XTC* field will be the TC selected for exclusive execution on the VPE. This allows initialization of one VPE by another, such that the initialized VPE can begin execution in an exception or single-threaded state, and the full context save/restore of one VPE by another, even if the target VPE is in an exception or single-threaded state.

Implementations may set the *XTC* field on the clearing of the *EVP* field of the *MVPControl* register by MTC0 or DVPE instructions if this simplifies the design, but given that *XTC*'s utility is in cross-VPE references (a TC running single-threaded in a VPE can always determine its identity by reading its own *TCBind* register), and given that no other VPEs can be executing when *EVP* is set, it is not particularly useful and is not required.

6.7 VPEConf1 Register(CP0 Register 1, Select 3)

Compliance Level: *Optional*.

The *VPEConf1* register is instantiated per VPE. It indicates the coprocessor and UDI resources available to the VPE. All fields in the *VPEConf1* register are read-only in normal operation, but may be writable while the *MVPControl VPC* bit is set. See Section 8.2.

Figure 6.6 shows the format of the *VPEConf1* register; Table 6.7 describes the *VPEConf1* register fields.

Figure 6.6 VPEConf1 Register Format

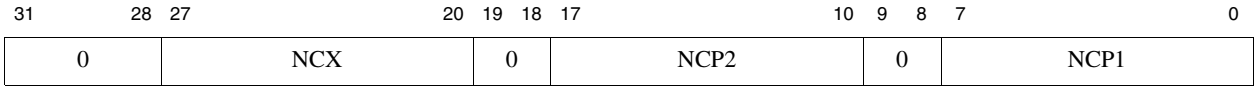


Table 6.7 VPEConf1 Register Field Descriptions

Fields		Description	Read/Write		Reset State	Compliance
Name	Bits		VPC=0	VPC=1		
NCX	27:20	Number of CorExtend™ UDI state instantiations available, for UDI blocks with persistent state.	R	R/W	Preset by hardware	Required
NCP2	17:10	Number of Coprocessor 2 contexts available.	R	R/W	Preset by hardware	Required
NCP1	7:0	Number of Coprocessor 1 contexts available.	R	R/W	Preset by hardware	Required
0	31:28, 19:18, 9:8	Reserved. Reads as zero, must be written as zero.	R		0	Reserved

6.8 YQMask Register (CP0 Register 1, Select 4)

Compliance Level: *Required for MIPS MT.*

The *YQMask* register is instantiated per VPE.

Figure 6.7 shows the format of the *YQMask* register; Table 6.8 describes the *YQMask* register fields.

Figure 6.7 YQMask Register Format

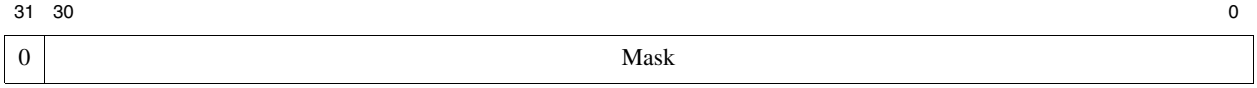


Table 6.8 YQMask Register Field Descriptions

Fields		Description	Read/Write	Reset State	Compliance
Name	Bits				
Mask	30:0	Bit vector which determines which values may be used as external state qualifiers by YIELD instructions.	R/W	0	Required
0	31	Must be written as zero; return zero on read.	0	0	Reserved

The *YQMask* register allows software control over values used to select external qualifier states for YIELD instructions. If a YIELD instruction has a positive value of its *rs* parameter, and any bit that is set in *rs* is not also set in *YQMask*, a Thread exception is raised on the YIELD instruction, with the *VPEControl EXCPT* field set to 3 to indicate the illegal qualifier condition.

If a processor implementation supports fewer than 31 qualifier state inputs, the *YQMask* bits corresponding to unimplemented inputs should be hard-wired to zero, so that attempts to suspend pending an impossible state are certain to cause an exception to be raised.

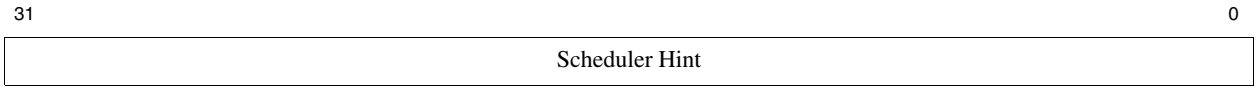
6.9 VPESchedule Register (CP0 Register 1, Select 5)

Compliance Level: *Optional*.

The *VPESchedule* register is optional, and is instantiated per-VPE.

Figure 6.8 shows the format of the *VPESchedule* register.

Figure 6.8 VPESchedule Register Format



The *Scheduler Hint* is a per-VPE value whose interpretation is scheduler implementation-dependent. For example, it could encode a description of the overall requested issue bandwidth for the associated VPE, or it could encode a priority level.

A *VPESchedule* register value of zero is the default, and should result in a well-behaved default scheduling of the associated VPE.

The *VPESchedule* register and the *TCSchedule* register create a hierarchy of issue bandwidth allocation. The set of *VPESchedule* registers assigns bandwidth to VPEs as a proportion of the total available on a processor or core, while the *TCSchedule* register can only assign bandwidth to threads as a function of that which is available to the VPE containing the thread.

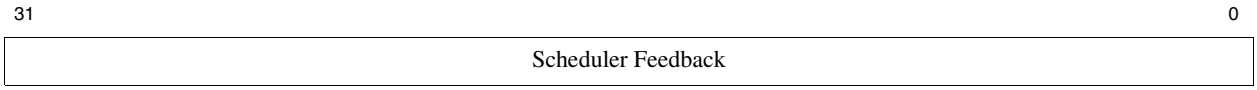
6.10 VPEScheFBack Register (CP0 Register 1, Select 6)

Compliance Level: *Optional*.

The *VPEScheFBack* register is an optional, per-VPE register.

Figure 6.9 shows the format of the *VPEScheFBack* register.

Figure 6.9 VPEScheFBack Register Format



The *Scheduler Feedback* is a per-VPE feedback value from scheduler hardware to software, whose interpretation is scheduler implementation-dependent. For example, it might encode the total number of instructions retired in the instruction streams on the associated VPE since the last time the value was cleared by software.

6.11 VPEOpt Register(CP0 Register 1, Select 7)

Compliance Level: *Optional*.

The *VPEOpt* register is instantiated per VPE. It provides control over optional per-VPE capabilities, such as cache "way" allocation management.

Figure 6.10 shows the format of the *VPEOpt* register; Table 6.9 describes the *VPEOpt* register fields.

Figure 6.10 VPEOpt Register Format

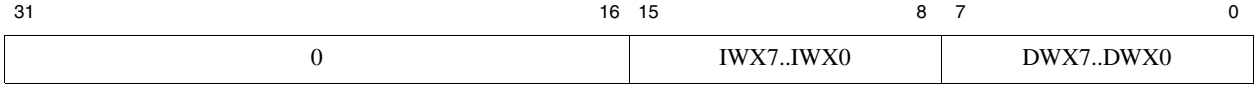


Table 6.9 VPEOpt Register Field Descriptions

Fields		Description	Reset State	Compliance
Name	**Bits**			
IWX7 .. IWX0	15:8	Instruction cache way exclusion mask. If programmable cache partitioning is supported by the processor (see Section 6.3) and enabled in the *MVPControl* register (see Section 6.2), a VPE can exclude an arbitrary subset of the first 8 ways of the primary instruction cache from allocation by the cache controller on behalf of the VPE. The existence of this register field is denoted by the PCP field of the *MVPConf0* register.	0	Optional
		Bit / **Name** / **Meaning**		
		15 / IWX7 / If set, I-cache way 7 will not be allocated for the VPE		
		14 / IWX6 / If set, I-cache way 6 will not be allocated for the VPE		
		13 / IWX5 / If set, I-cache way 5 will not be allocated for the VPE		
		12 / IWX4 / If set, I-cache way 4 will not be allocated for the VPE		
		11 / IWX3 / If set, I-cache way 3 will not be allocated for the VPE		
		10 / IWX2 / If set, I-cache way 2 will not be allocated for the VPE		
		9 / IWX1 / If set, I-cache way 1 will not be allocated for the VPE		
		8 / IWX0 / If set, I-cache way 0 will not be allocated for the VPE		
DWX7..DWX0	7:0	Data cache way exclusion mask. If programmable cache partitioning is supported by the processor (see Section 6.3) and enabled in the *MVPControl* register (see Section 6.2), a VPE can exclude an arbitrary subset of the first 8 ways of the primary data cache from allocation by the cache controller on behalf of the VPE. The existence of this register field is denoted by the PCP field of the *MVPConf0* register.	0	Optional
		Bit / **Name** / **Meaning**		
		7 / DWX7 / If set, D-cache way 7 will not be allocated for the VPE		
		6 / DWX6 / If set, D-cache way 6 will not be allocated for the VPE		
		5 / DWX5 / If set, D-cache way 5 will not be allocated for the VPE		
		4 / DWX4 / If set, D-cache way 4 will not be allocated for the VPE		
		3 / DWX3 / If set, D-cache way 3 will not be allocated for the VPE		
		2 / DWX2 / If set, D-cache way 2 will not be allocated for the VPE		
		1 / DWX1 / If set, D-cache way 1 will not be allocated for the VPE		
		0 / DWX0 / If set, D-cache way 0 will not be allocated for the VPE		
0	31:16	Reserved. Reads as zero, must be written as zero.	0	Reserved

The *IWX* and *DWX* bits inhibit *allocation* of cache lines in the specified way. They do not prevent fetches and loads by the VPE from hitting in those lines if the requested physical address is present, nor do they prevent stores from modifying the contents of a line already present in the cache.

If fewer than 8 ways are implemented by a processor's instruction or data cache, the *IWX* and *DWX* bits corresponding to unimplemented cache ways may be implemented as read-only (RO) zero bits.

Behavior of the processor is **UNDEFINED** if references are made to cached address spaces by a VPE which has excluded all implemented cache ways from allocation.

Whether or not a cache line in a way that is excluded from allocation by a VPE can be locked by a CACHE instruction issued by that VPE is implementation-dependent.

If per-TC cache partitioning is also also used (through the use of the *TCOpt* register), care must be taken not to exclude all ways of the cache through the usage of both per-VPE cache partitioning and per-TC cache partitioning.

6.12 TCStatus Register (CP0 Register 2, Select 1)

Compliance Level: *Required for MIPS MT.*

The *TCStatus* register is instantiated per TC as part of the system coprocessor.

Figure 6.11 shows the format of the *TCStatus* register; Table 6.10 describes the *TCStatus* register fields.

Figure 6.11 TCStatus Register Format

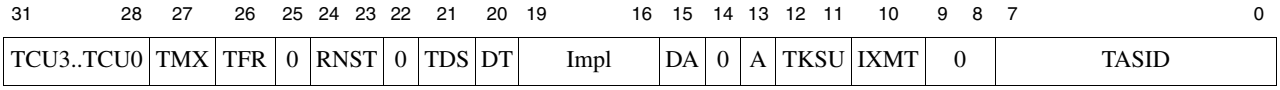


Table 6.10 TCStatus Register Field Descriptions

Fields		Description			Read / Write	Reset State	Fork State	Compliance
Name	Bits							
TCU (TCU3.. TCU0)	31:28	Controls access of a TC to coprocessors 3,2,1, and 0 respectively. *Status* bits *CU3..CU0* are identical to *TCStatus* bits *TCU3..TCU0* of the thread referencing that *Status* with an MFC0 operation. The modification of either must be visible in both.			R/W	Undefined	Unchanged by FORK	Required
TMX	27	Controls access of a TC to extended media processing state, such as MDMX and DSP Module accumulators. *Status* bit *MX* is identical to *TCStatus* bit *TMX* of the thread referencing that *Status* with an MFC0 operation. The modification of either must be visible in both.			R/W	0	Unchanged by FORK	Required for MDMX and DSP Modules
TFR	26	TC Floating-Point Register Mode for Multi-threaded 64-bit FPU.	**Value**	**Meaning**	R/W	Undefined	Copied from forking thread	Required if 64-bit MT-FPU is implemented
			0	FPRs can only hold 32-bit data values. 64-bit data values held in even-odd pairs of FPRs.				
			1	FPRs can hold either 32-bit or 64-bit data values				
RNST	24:23	Run State of TC. Indicates the Running vs. Blocked state of the TC (see Section 3.6) and the reason for blockage. Value is stable only if TC is Halted and examined by another TC using an MFTR operation.	**Value**	**Meaning**	R	0	0	Required
			0	Running				
			1	Blocked on WAIT				
			2	Blocked on YIELD				
			3	Blocked on Gating Storage				
TDS	21	Thread stopped in branch Delay Slot. If a TC is Halted such that the next instruction to issue would be an instruction in a branch delay slot, the *TCRestart* register will contain the address of the branch instruction, and the *TDS* bit will be set. Otherwise *TDS* is cleared on a Halt, or on a software write to the *TCRestart* register.			R	0	0	Required

Table 6.10 TCStatus Register Field Descriptions

Fields		Description	Read / Write	Reset State	Fork State	Compliance
Name	Bits					
DT	20	Dirty TC. This bit is set by hardware whenever an instruction is retired using the associated TC, and on successful dispatch of the TC via a FORK instruction. The setting of *DT* by the retirement of instructions is inhibited if the instructions are issued with the *EXL* or *ERL* bits of *Status* set, or with the processor in Debug mode.	R/W	0	1	Required
Impl	19:16	These bits are implementation-dependent and are not defined by the architecture. If they are not implemented, they must be ignored on write and read as zero	Impl. Dep.	Impl. Dep.	Impl. Dep.	Optional
DA	15	Dynamic Allocation enable. If set, TC may be allocated/deallocated/scheduled by the FORK and YIELD instructions.	R/W	0	FORK allocate only possible if DA = 1	Required
A	13	Thread Activated. Set automatically when a FORK instruction allocates the TC, and cleared automatically when a YIELD $0 instruction deallocates it.	R/W	1 for TC 0, 0 for all others.	1	Required
TKSU	12:11	Defined as per the *Status* register *KSU* field. This is the per-TC Kernel/Supervisor/User state. The *Status KSU* field is identical to the *TCStatus TKSU* field of the thread referencing *Status*. The modification of either must be visible in both.	R/W	Undefined	Copied from forking thread	Required
IXMT	10	Interrupt Exempt. If set, the associated TC will not be used to handle Interrupt exceptions. Debug Interrupt exceptions are not affected.	R/W	0	Unchanged by FORK	Required
TASID	7:0	Defined as per the *EntryHi* register *ASID* field. This is the per-TC *ASID* value. The *EntryHi ASID* is identical to the *TCStatus TASID* of the thread referencing *EntryHi* with an MFC0 operation. The modification of either must be visible in both.	R/W if TLB implemented; 0 if TLB not implemented	Undefined	Copied from forking thread	Required if TLB implemented., Reserved otherwise
0	25, 22, 14, 9:8	Must be written as zero; return zero on read.	0	0	0	Reserved

The *(T)CUx*, *(T)MX*, and *(T)KSU* fields of the *TCStatus* and *Status* registers always display the correct state. That is, if the field is written via *TCStatus*, the new value may be read via *Status*, and vice-versa. Similarly, the *(T)ASID* field of the *TCStatus* and *EntryHi* always display the same current value for the TC.

6.13 TCBind Register (CP0 Register 2, Select 2)

Compliance Level: *Required for MIPS MT.*

The *TCBind* register is instantiated per TC as part of the system coprocessor.

Figure 6.12 shows the format of the *TCBind* register; Table 6.11 describes the *TCBind* register fields.

Figure 6.12 TCBind Register Format

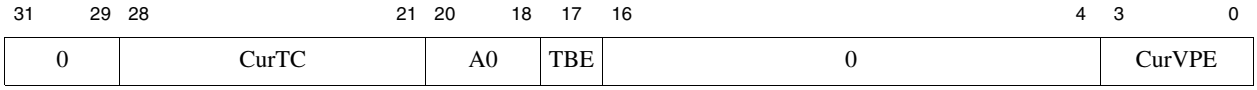


Table 6.11 TCBind Register Field Descriptions

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
CurTC	28:21	Indicates the number (index) of the TC.	R	TC #	Required
A0	20:18	Architecturally zero-valued field providing least-significant bits when a *TCBind* value is shifted right to be used as a scaled index into arrays of 32 or 64-bit values.	R	0	Required
TBE	17	TC Bus Error. Set by hardware when a transaction causing a bus error is identified as resulting from a load or store issued by the TC. Implementations may set the *TBE* bits of multiple TCs on a single DBE exception if multiple memory requests to the same memory location or cache line from the different TCs were merged. Implementations may generate bus error exceptions without setting a *TBE* bit if it is not possible to associate the failing transaction with a particular TC.	R/W	0	Required
CurVPE	3:0	Indicates and controls the binding of the TC to a VPE. Field is optionally Read/Write only when the *VPC* bit of the *MVPControl* register is set.	R or R/W	0 for TC 0, preset by hardware for all others	Required
0	31:29, 16:4	Must be written as zero; return zero on read.	0	0	Reserved

In reconfigurable MIPS MT processors, the binding of TCs to VPEs is managed via the *CurVPE* field of *TCBind*. If TC assignment to VPEs is configurable, the *CurVPE* fields of all TCs in the processor are writable if the *VPC* bit of the *MVPControl* register is set. At all other times, *CurVPE* is a read-only indication of which VPE contains the TC. Software can thus determine on which VPE it is running by executing an MFC0 instruction from *TCBind* and inspecting *CurVPE*. While implementations may allow for it under well-defined circumstances, behavior of a processor may be **UNPREDICTABLE** if software executing on a given TC changes its own VPE binding “on the fly”.

6.14 TCRestart Register (CP0 Register 2, Select 3)

Compliance Level: *Required for MIPS MT.*

The *TCRestart* register is instantiated per-TC, with the same width as the processor GPRs.

Figure 6.13 shows the format of the *TCRestart* register. Table 6.12 describes the *TCRestart* register fields.

Figure 6.13 TCRestart Register Format

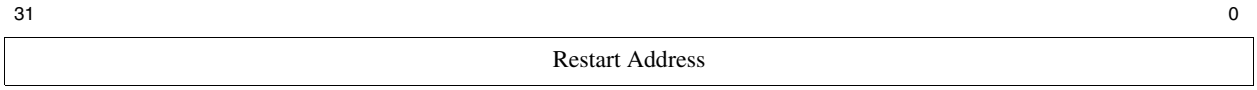


Table 6.12 TCRestart Register Field Descriptions

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
Restart Address	31..0	Address at which execution of the TC is restarted.	R/W	Undefined	Required

When a TC is in a Halted state, a read of the *TCRestart* register returns the instruction address at which the TC will start execution when it is restarted. The *TCRestart* register can be written while the associated TC is in a Halted state to change the address at which the TC will restart.

Reading the *TCRestart* register of a non-Halted TC will return the **UNSTABLE** address of some instruction that the TC was executing in the past, but which may no longer be valid. Writing the *TCRestart* register of a non-Halted TC will result in an **UNDEFINED** TC state.

In the case of branch and jump instructions with architectural delay slots, the restart address will advance beyond the address of the branch or jump instruction only after the instruction in the delay slot has been retired. If halted between the execution of a branch and the associated delay slot instruction, the branch delay slot is indicated by the *TDS* bit of the *TCStatus* register (see Section 6.12).

Software writes to the *TCRestart* register cause the *TDS* bit of the *TCStatus* register to be cleared. If a software write of the *TCRestart* register of a TC intervenes between the execution of an LL instruction and an SC instruction on the target TC, the SC operation must fail.

When the processor writes the *TCRestart* register, it combines the address at which the TC will resume execution with the value of the *ISAMode* register:

$$\mathrm{TCRestart} \leftarrow \mathrm{resumePC}_{31..1} \parallel \mathrm{ISAMode}_{0}$$

“resumePC” is the address at which the TC will resume execution, as described above.

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

$$\mathrm{PC} \leftarrow \mathrm{TCRestart}_{31..1} \parallel 0$$
$$\mathrm{ISAMode} \leftarrow \mathrm{TCRestart}_{0}$$

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

6.15 TCHalt Register (CP0 Register 2, Select 4)

Compliance Level: *Required for MIPS MT.*

The *TCHalt* register is instantiated per TC as part of the system coprocessor.

Figure 6.14 shows the format of the *TCHalt* register; Table 6.13 describes the *TCHalt* register fields.

Figure 6.14 TCHalt Register Format

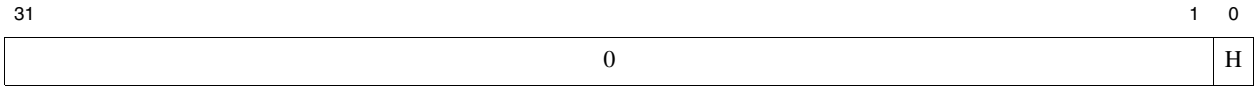


Table 6.13 TCHalt Register Field Descriptions

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
H	0	Thread Halted. When set, the associated thread has been halted and cannot be allocated, activated, or scheduled.	R/W	0 for TC 0, 1 for all others	Required
0	31:1	Must be written as zero; return zero on read.	0	0	Reserved

Writing a one to the *Halted* bit of an activated TC causes the associated thread to cease fetching instructions and to set its Restart Address in the *TCRestart* register (see Section 6.14) to the address of the next instruction to be issued. If the instruction stream associated with the TC is blocked waiting on a response from Gating Storage (see Chapter 9, "Data-Driven Scheduling of MIPS® MT Threads" on page 102), the load or store is aborted, and the TC resolves to a state where the *TCRestart* register and *TDS* field of the *TCStatus* register (see Section 6.12) reflect a restart at the blocked load or store. If the TC is blocked on a WAIT or YIELD instruction, it resolves to a stable restart state. If the TC was blocked at the time it is Halted, the *RNST* field of *TCStatus* indicates the blocked state, and the reason for blocking, even if that reason was an operation aborted by the Halt. Writing a zero to the *Halted* bit of an activated TC allows the associated thread of execution to be scheduled, fetching and executing as indicated by *TCRestart*. A one in the *Halted* bit (*TCHalt.H*) of a TC prevents that TC from being allocated and activated by a FORK instruction.

Any fetched/decoded but unissued instruction state associated with a TC must be discarded when a TC is *Halted* by a write to its *TCHalt* register.

The effect of writing a one to the *Halted* bit of a TC may not be instantaneous. An instruction hazard barrier, e.g., JR.HB, is required to guarantee that the target thread has been fully halted. As MFTR semantics (see MFTR) require that a halted TC have stable state, the hazard barrier must assure that all long-latency operations have completed, or at least appear to have completed to software (e.g., GPRs scoreboarded to ensure that MFTRs receive only the correct and final value).

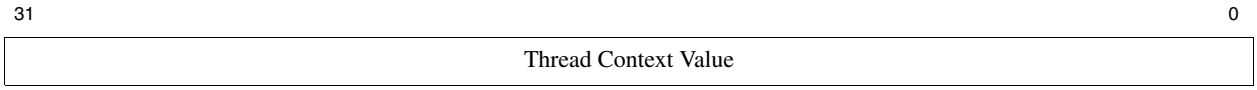
6.16 TCContext Register (CP0 Register 2, Select 5)

Compliance Level: *Required for MIPS MT.*

The *TCContext* register is instantiated per-TC, with the same width as the processor GPRs.

Figure 6.15 shows the format of the *TCContext* register.

Figure 6.15 TCContext Register Format



TCContext is purely a software read/write register, usable by the operating system as a pointer to thread-specific storage, e.g., a thread context save area.

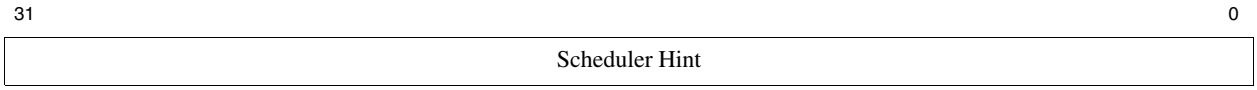
6.17 TCSchedule Register (CP0 Register 2, Select 6)

Compliance Level: *Optional*.

The *TCSchedule* register is optional, but when implemented must be implemented per-TC.

Figure 6.16 shows the format of the *TCSchedule* register.

Figure 6.16 TCSchedule Register Format



The *Scheduler Hint* is a per-TC value whose interpretation is scheduler implementation-dependent. For example, it could encode a description of the requested issue bandwidth for the associated thread, as in the *VPESchedule* register, or it could encode a priority level.

A *TCSchedule* register value of zero is the default, and should result in a well-behaved default scheduling of the associated thread.

The *VPESchedule* register and the *TCSchedule* register create a hierarchy of issue bandwidth allocation. The set of *VPESchedule* registers assigns bandwidth to VPEs as a proportion of the total available on a processor or core, while the *TCSchedule* register can only assign bandwidth to threads as a function of that which is available to the VPE containing the thread.

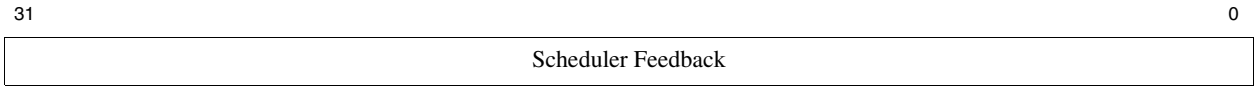
6.18 TCScheFBack Register (CP0 Register 2, Select 7)

Compliance Level: *Optional*.

The *TCScheFBack* register is optional, but when implemented must be implemented per-TC.

Figure 6.17 shows the format of the *TCScheFBack* register.

Figure 6.17 TCScheFBack Register Format



The *Scheduler Feedback* is a per-TC feedback value from scheduler hardware to software, whose interpretation is scheduler implementation-dependent. For example, it might encode the number of instructions retired in the instruction stream corresponding to the TC since the last time the value was cleared by software.

6.19 TCOpt Register(CP0 Register 3, Select 7)

Compliance Level: *Optional*.

The *TCOpt* register is instantiated per TC. It provides control over optional per-TC capabilities, such as cache "way" allocation management.

Figure 6.18 shows the format of the *TCOpt* register; Table 6.14 describes the *TCOpt* register fields.

Figure 6.18 TCOpt Register Format

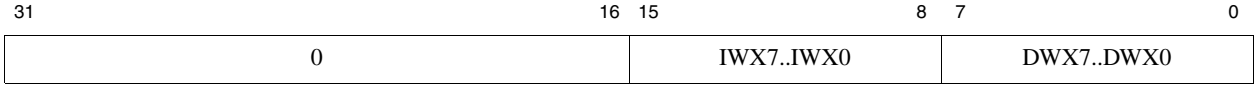


Table 6.14 TCOpt Register Field Descriptions

Fields		Description	Reset State	Compliance
Name	**Bits**			
IWX7 .. IWX0	15:8	Instruction cache way exclusion mask. If programmable cache partitioning is supported by the processor (see Section 6.3) and enabled in the *MVPControl* register (see Section 6.2), a TC can exclude an arbitrary subset of the first 8 ways of the primary instruction cache from allocation by the cache controller on behalf of the TC. The existence of this register field is denoted by the TCP field of the *MVPConf0* register.	0	Optional
		Bit / **Name** / **Meaning**		
		15 / IWX7 / If set, I-cache way 7 will not be allocated for the TC		
		14 / IWX6 / If set, I-cache way 6 will not be allocated for the TC		
		13 / IWX5 / If set, I-cache way 5 will not be allocated for the TC		
		12 / IWX4 / If set, I-cache way 4 will not be allocated for the TC		
		11 / IWX3 / If set, I-cache way 3 will not be allocated for the TC		
		10 / IWX2 / If set, I-cache way 2 will not be allocated for the TC		
		9 / IWX1 / If set, I-cache way 1 will not be allocated for the TC		
		8 / IWX0 / If set, I-cache way 0 will not be allocated for the TC		
DWX7..DWX0	7:0	Data cache way exclusion mask. If programmable cache partitioning is supported by the processor (see Section 6.3) and enabled in the *MVPControl* register (see Section 6.2), a TC can exclude an arbitrary subset of the first 8 ways of the primary data cache from allocation by the cache controller on behalf of the TC. The existence of this register field is denoted by the TCP field of the *MVPConf0* register.	0	Optional
		Bit / **Name** / **Meaning**		
		7 / DWX7 / If set, D-cache way 7 will not be allocated for the TC		
		6 / DWX6 / If set, D-cache way 6 will not be allocated for the TC		
		5 / DWX5 / If set, D-cache way 5 will not be allocated for the TC		
		4 / DWX4 / If set, D-cache way 4 will not be allocated for the TC		
		3 / DWX3 / If set, D-cache way 3 will not be allocated for the TC		
		2 / DWX2 / If set, D-cache way 2 will not be allocated for the TC		
		1 / DWX1 / If set, D-cache way 1 will not be allocated for the TC		
		0 / DWX0 / If set, D-cache way 0 will not be allocated for the TC		
0	31:16	Reserved. Reads as zero, must be written as zero.	0	Reserved

The *IWX* and *DWX* bits inhibit *allocation* of cache lines in the specified way. They do not prevent fetches and loads by the TC from hitting in those lines if the requested physical address is present, nor do they prevent stores from modifying the contents of a line already present in the cache.

If fewer than 8 ways are implemented by a processor's instruction or data cache, the *IWX* and *DWX* bits corresponding to unimplemented cache ways may be implemented as read-only (RO) zero bits.

Behavior of the processor is **UNDEFINED** if references are made to cached address spaces by a TC which has excluded all implemented cache ways from allocation.

Whether or not a cache line in a way that is excluded from allocation by a TC can be locked by a CACHE instruction issued by that TC is implementation-dependent.

If per-VPE cache partitioning is also also used (through the use of the *VPEOpt* register), care must be taken not to exclude all ways of the cache through the usage of both per-VPE cache partitioning and per-TC cache partitioning.

6.20 SRSConf0 (CP0 Register 6, Select 1)

Compliance Level: *Required for MIPS MT.*

The *SRSConf0* register is instantiated per VPE. It indicates the binding of TCs or other GPR resources to Shadow Register Sets 1 through 3.

Figure 6.19 shows the format of the *SRSConf0* register; Table 6.15 describes the *SRSConf0* register fields.

Figure 6.19 SRSConf0 Register Format

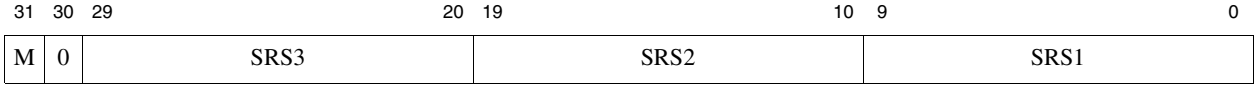


Table 6.15 SRSConf0 Register Field Descriptions

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
M	31	If set, *SRSConf1* register is implemented. If clear, no more than 3 shadow sets may be configured.	R	Preset by hardware	Required
SRS3	29:20	GPR set to be used if CSS = 3. See below for encoding.	RW or R	Preset by hardware	Required
SRS2	19:10	GPR set to be used if CSS = 2. See below for encoding.	RW or R	Preset by hardware	Required
SRS1	9:0	GPR set to be used if CSS = 1. See below for encoding.	RW or R	Preset by hardware	Required
0	30	Reserved. Reads as zero, must be written as zero.	R	0	Reserved

Each *SRSx* field of the *SRSConf0* register identifies which GPR will be used for references to Shadow Register Set *x*. There is no field for SRS0, as "Shadow Set 0" is taken in MIPS MT to mean the GPR set of the TC associated with entry into an exception handler. An *SRSx* field value may be hard-wired to all-ones (0x3ff) to indicate that the processor logic does not support the associated SRS number. If any SRS numbers are uninstantiated, they should be in a contiguous range starting from the highest number, i.e., *SRS3* may be uninstantiated while *SRS2* and *SRS1* are instantiated, but *SRS1* must be instantiated if *SRS2* is instantiated. The *M* bit should only be set, and the *SRSConf1* register should only be implemented, if all three *SRSx* fields of *SRSConf0* are instantiated.

Instantiated *SRSx* fields may be hard-wired or programmable. Hard wired fields represent dedicated shadow sets that are statically configured into the VPE, and contain distinct unsigned values greater than the total complement of TCs on a processor, but less than 0x3fe. Programmable *SRSx* fields have a reset value of 0x3fe. A value of 0x3ff or 0x3fe in an *SRSx* field means that SRS x is invalid. The *HSS* field of the *SRSCtl* register always indicates the number of the highest numbered valid SRS, i.e., one less than *x* for the lowest numbered invalid *SRSx* field. A programmable entry may be made valid by writing a value less than 0x3fe into it. A TC is assigned for its GPRs to be used as a Shadow Register set by writing the number of the TC, zero extended, into the *SRSx* field corresponding to the shadow set number for which the TC is to be used. Only a TC bound to a VPE may be used as an SRS on that VPE. If the CurVPE field of the *TCBind* register of a TC being assigned to an SRS is does not contain the number of the VPE associated with the *SRSConf0...4* register being programmed, the *SRSx* field is not updated. The effect of writing an *SRSx* value greater than the number of the highest numbered TC on a processor is implementation-dependent.

Behavior of the processor is **UNDEFINED** in the face of exceptions and FORK instructions if a TC is assigned to Shadow Register use when the DA bit is set in its *TCStatus* register.

Behavior of the processor is **UNDEFINED** if writing an invalid *SRSx* field value causes the *SRSCtl HSS* field to take on a value that is less than the current value of the *SRSCtl CSS* or *PSS* fields. Behavior of the processor is **UNDEFINED** under exceptions if the *SRSCtl HSS* field takes on a value less than the *SRSCtl ESS* field. Behavior of the pro-

cessor is **UNDEFINED** under EIC interrupts if the *SRSCtl HSS* field takes on a value less than the *SRSCtl EICSS* field. Software must thus take care to modify the *ESS* and *EISS* fields as necessary prior to de-allocating a TC from SRS service.

A TC may be reclaimed from use as a shadow set by writing some other value, possibly 0x3fe, into the *SRSx* field which had contained the TC's number.

At no time should the same value, other than the values 0x3ff and 0x3fe, be present more than one distinct *SRSx* field.

The sequence of shadow set numbers to be used by software is a monotonically increasing sequence starting with zero. To assure correct and backward-compatible software operation, there must be no invalid (0x3ff/0x3fe) *SRSx* field at a lower *x* index than that of a valid *SRSx* field.

6.21 SRSConf1 (CP0 Register 6, Select 2)

Compliance Level: *Optional*.

The *SRSConf1* register is instantiated per VPE. It indicates the binding of TCs or other GPR resources to Shadow Register Sets 4 through 6.

Figure 6.20 shows the format of the *SRSConf1* register; Table 6.16 describes the *SRSConf1* register fields.

Figure 6.20 SRSConf1 Register Format

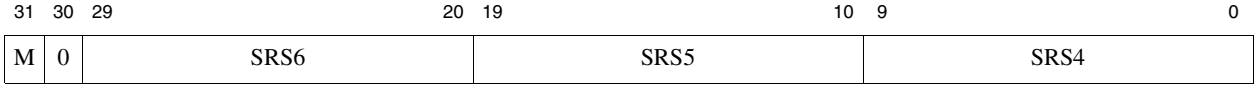


Table 6.16 SRSConf1 Register Field Descriptions

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
M	31	If set, *SRSConf2* register is implemented. If clear, no more than 6 shadow sets may be configured.	R	Preset by hardware	Required
SRS6	29:20	GPR set to be used if CSS = 6. See below for encoding.	RW or R	Preset by hardware	Required
SRS5	19:10	GPR set to be used if CSS = 5. See below for encoding.	RW or R	Preset by hardware	Required
SRS4	9:0	GPR set to be used if CSS = 4. See below for encoding.	RW or R	Preset by hardware	Required
0	30	Reserved. Reads as zero, must be written as zero.	R	0	Reserved

Each *SRSx* field of the *SRSConf1* register identifies which GPR will be used for references to Shadow Register Set *x*. An *SRSx* field value may be hard-wired to all-ones (0x3ff) to indicate that the processor logic does not support the associated SRS number. If any SRS numbers are uninstantiated, they should be in a contiguous range starting from the highest number, i.e., *SRS6* may be uninstantiated while *SRS5* and *SRS4* are instantiated, but *SRS4* must be instantiated if *SRS5* is instantiated. The *M* bit should only be set, and the *SRSConf2* register should only be implemented, if all three *SRSx* fields of *SRSConf1* are instantiated.

The semantics and encodings of the *SRSx* fields of the *SRSConf1* register are the same as those of the *SRSConf0* register, except in that they are applied to Shadow Register Sets 4 through 6. See Section 6.20.

6.22 SRSConf2 (CP0 Register 6, Select 3)

Compliance Level: *Optional*.

The *SRSConf2* register is instantiated per VPE. It indicates the binding of TCs or other GPR resources to Shadow Register Sets 7 through 9.

Figure 6.21 shows the format of the *SRSConf2* register; Table 6.17 describes the *SRSConf2* register fields.

Figure 6.21 SRSConf2 Register Format

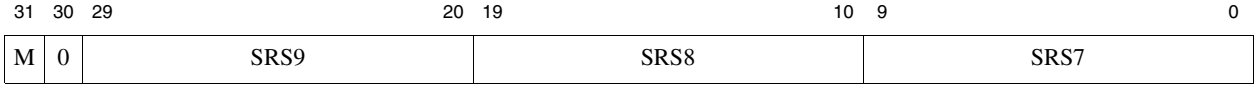


Table 6.17 SRSConf2 Register Field Descriptions

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
M	31	If set, *SRSConf3* register is implemented. If clear, no more than 9 shadow sets may be configured.	R	Preset by hardware	Required
SRS9	29:20	GPR set to be used if CSS = 9. See below for encoding.	RW or R	Preset by hardware	Required
SRS8	19:10	GPR set to be used if CSS = 8. See below for encoding.	RW or R	Preset by hardware	Required
SRS7	9:0	GPR set to be used if CSS = 7. See below for encoding.	RW or R	Preset by hardware	Required
0	30	Reserved. Reads as zero, must be written as zero.	R	0	Reserved

Each *SRSx* field of the *SRSConf2* register identifies which GPR will be used for references to Shadow Register Set *x*. An *SRSx* field value may be hard-wired to all-ones (0x3ff) to indicate that the processor logic does not support the associated SRS number. If any SRS numbers are uninstantiated, they should be in a contiguous range starting from the highest number, i.e., *SRS9* may be uninstantiated while *SRS8* and *SRS7* are instantiated, but *SRS7* must be instantiated if *SRS8* is instantiated. The *M* bit should only be set, and the *SRSConf3* register should only be implemented, if all three *SRSx* fields of *SRSConf2* are instantiated.

The semantics and encodings of the *SRSx* fields of the *SRSConf2* register are the same as those of the *SRSConf0* register, except in that they are applied to Shadow Register Sets 7 through 9. See Section 6.20.

6.23 SRSConf3 (CP0 Register 6, Select 4)

Compliance Level: *Optional*.

The *SRSConf3* register is instantiated per VPE. It indicates the binding of TCs or other GPR resources to Shadow Register Sets 10 through 12.

Figure 6.22 shows the format of the *SRSConf3* register; Table 6.18 describes the *SRSConf3* register fields.

Figure 6.22 SRSConf3 Register Format

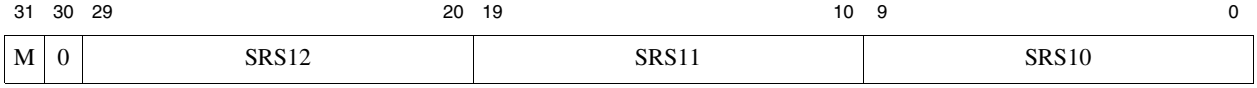


Table 6.18 SRSConf3 Register Field Descriptions

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
M	31	If set, *SRSConf4* register is implemented. If clear, no more than 9 shadow sets may be configured.	R	Preset by hardware	Required
SRS12	29:20	GPR set to be used if CSS = 12. See below for encoding.	RW or R	Preset by hardware	Required
SRS11	19:10	GPR set to be used if CSS = 11. See below for encoding.	RW or R	Preset by hardware	Required
SRS10	9:0	GPR set to be used if CSS = 10. See below for encoding.	RW or R	Preset by hardware	Required
0	30	Reserved. Reads as zero, must be written as zero.	R	0	Reserved

Each *SRSx* field of the *SRSConf3* register identifies which GPR will be used for references to Shadow Register Set *x*. An *SRSx* field value may be hard-wired to all-ones (0x3ff) to indicate that the processor logic does not support the associated SRS number. If any SRS numbers are uninstantiated, they should be in a contiguous range starting from the highest number, i.e., *SRS12* may be uninstantiated while *SRS11* and *SRS10* are instantiated, but *SRS10* must be instantiated if *SRS11* is instantiated. The *M* bit should only be set, and the *SRSConf4* register should only be implemented, if all three *SRSx* fields of *SRSConf3* are instantiated.

The semantics and encodings of the *SRSx* fields of the *SRSConf3* register are the same as those of the *SRSConf0* register, except in that they are applied to Shadow Register Sets 10 through 12. See Section 6.20.

6.24 SRSConf4 (CP0 Register 6, Select 5)

Compliance Level: *Optional.*

The *SRSConf4* register is instantiated per VPE. It indicates the binding of TCs or other GPR resources to Shadow Register Sets 13 through 15.

Figure 6.23 shows the format of the *SRSConf4* register; Table 6.19 describes the *SRSConf4* register fields.

Figure 6.23 SRSConf4 Register Format

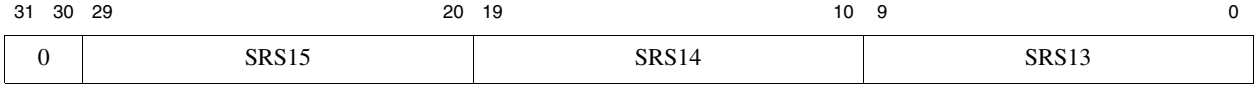


Table 6.19 SRSConf4 Register Field Descriptions

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
SRS15	29:20	GPR set to be used if CSS = 15. See below for encoding.	RW or R	Preset by hardware	Required
SRS14	19:10	GPR set to be used if CSS = 14. See below for encoding.	RW or R	Preset by hardware	Required
SRS13	9:0	GPR set to be used if CSS = 13. See below for encoding.	RW or R	Preset by hardware	Required
0	31,30	Reserved. Reads as zero, must be written as zero.	R	0	Reserved

Each *SRSx* field of the *SRSConf4* register identifies which GPR will be used for references to Shadow Register Set *x*. An *SRSx* field value may be hard-wired to all-ones (0x3ff) to indicate that the processor logic does not support the associated SRS number. If any SRS numbers are uninstantiated, they should be in a contiguous range starting from the highest number, i.e., *SRS15* may be uninstantiated while *SRS14* and *SRS13* are instantiated, but *SRS13* must be instantiated if *SRS14* is instantiated.

The semantics and encodings of the *SRSx* fields of the *SRSConf4* register are the same as those of the *SRSConf0* register, except in that they are applied to Shadow Register Sets 13 through 15. See Section 6.20.

6.25 Modifications to Existing MIPS® Privileged Resource Architecture

The Multi-threading Module modifies some elements of the existing nanoMIPS32 PRA.

6.25.1 SRSCtl Register

The *HSS* field value can change at run-time if an implementation allows TCs to be assigned to SRSs via the *SRSConf0-SRSConf4* registers. The *HSS* value tracks the highest valid *SRSx* field of an *SRSConf* register. Software must ensure that the *HSS* field does not take on a value that makes the value of any of the *PSS*, *CSS*, *ESS*, or *EISS* fields of the *SRSCtl* register illegal (see Section 6.20).

A zero value in the *PSS* or *CSS* field of the *SRSCtl* register indicates that the previous or current “shadow set” is not a built-in SRS or a TC register file allocated to a Shadow Set, but is in fact the register set belonging to the TC servicing the exception, whose number can be found in the *CurTC* field of the *TCBind* register, as read with an MFC0 instruction by the exception handler.

6.25.2 Cause Register

There is a new *Cause* register *ExcCode* value required for the Thread exceptions

Table 6.20 MIPS® MT Thread Exception

Exception Code Value		Mnemonic	Description
Decimal	Hexadecimal		
25	16#19	Thread	Thread Allocation, Deallocation, or Scheduling Exceptions

6.25.3 Machine Check Exceptions

A MIPS MT processor does not generate Machine Check exceptions on duplicate TLB entries. Duplicate entries must be detected and suppressed on TLB writes, without causing an exception.

6.25.4 Debug Register

On a MIPS MT processor, the *SSt* and *OffLine* fields of the EJTAG *Debug* register are instantiated per-TC. All other read/write fields are implemented per-VPE. See Section 10.2.

6.25.5 EBase Register

Each VPE sees a distinct value in the *CPUNum* field of the *EBase* register.

6.25.6 Config1 Register

The normally read-only *MMU_Size*, *C2*, *MD*, and *FP* fields of the *Config1* register may be modifiable by software while a processor is in a configuration state, as defined by the *VPC* bit of the *MVPControl* register (see Section 8.2).

6.25.7 Config3 Register

A new *Config3* register field is defined to express and control the availability of the MIPS MT Module.

Table 6.21 New Config3 Fields for MIPS® MT

Field		Description	Read / Write	Reset State
Name	Bit			
MT	2	Indicates that the MT Module is implemented on the processor.	R	Preset

6.25.8 LLAddr Register

It is implementation-dependent whether the *LLAddr* register is implemented per-TC or per VPE.

6.26 Thread State as a Function of Privileged Resource State

The following table summarizes the TC state definitions of Section 3.6 in terms of the associated Module privileged resource state.

Table 6.22 TC State as Function of MIPS® MT PRA State

TCHalt.H	TCStatus.A	TCStatus.RNST	TC State	
1	x	x	Halted	
0	0	x	Free	
0	1	0	Activated	Running
0	1	>0		Blocked

6.27 Thread Allocation and Initialization Without FORK

The procedure for an operating system to create a thread “by hand” would be:

1. Execute a DMT to stop other threads from executing and possibly FORKing or Halting threads.
2. Execute a JR.HB to ensure that other threads have quiesced.
3. Identify an available TC by setting the *TargTC* field of the *VPEControl* register to successive values from 0 to *PTC*, reading the *TCBind* registers with an MFTR instruction to identify those belonging to the same VPE (those having the same value in the *TCBind CurVPE* field as the current “parent” thread), and reading their *TCStatus* and *TCHalt* registers with MFTR instructions. A free TC will have neither the *H* bit of *TCHalt* nor the *Activated* bit of *TCStatus* set, as per Table 6.22. TCs that have been assigned for use as shadow register storage must be skipped in this search.
4. Perform an MTTR of a value of 1 to the selected TC’s *TCHalt* register to prevent it being allocated by another thread.
5. Execute an EMT instruction to re-enable multi-threading.
6. Copy any desired GPRs or other program state into the selected TC using MTTR instructions.
7. Write the desired starting execution address into the thread’s restart address register using an MTTR instruction to the selected TC’s *TCRestart* register.

8. Write a value with a 1 in the *Activated* bit position to the selected *TCStatus* register using an MTTR instruction.

9. Write a value of zero into the selected *TCHalt* register using an MTTR instruction.

The newly allocated thread will then be schedulable. The steps of executing DMT and EMT can be skipped if *EXL* or *ERL* are known to be set during the procedure, as they implicitly inhibit multi-threaded execution.

6.28 Thread Termination and Deallocation without YIELD

The procedure for an operating system to terminate the current thread would be:

1. Write a value with *EXL* = 0, *ERL* = 0, and *KSU* = 0 to the *Status* register using MTC0, setting Kernel mode for the retiring TC and removing the inhibition of multi-threaded execution due to *EXL*/*ERL*.

2. Write a value with zero in the *Activated* bit position to the *TCStatus* register, using a standard MTC0 instruction.

One thread, running in a privileged mode, could also terminate another, using MTTR instructions, but it would present an additional problem to the OS to determine which TC should be deallocated and at what point the state of the thread's computation is stable.

6.29 Multi-threading and Coprocessors

Coprocessors attached to a multi-threaded VPE may have a single context, which must be shared among processor threads, or it may have multiple contexts, such that distinct instruction streams executing concurrently from multiple TCs can likewise have concurrent use of coprocessor resources. A "multi-threaded" coprocessor, with multiple coprocessor contexts, need not have the same number of contexts as the VPE to which it is attached has TCs. For VPE to use a coprocessor, some mapping, which may or may not be dynamic, must exist between a TC and an associated coprocessor context. This could be an implicit 1:1 or many-to-one mapping, an even/odd or other hash mapping, or a programmable mapping. A coprocessor context is *bound* to a TC if a mapping exists from the TC to the coprocessor context, and access to the coprocessor context by the TC's instruction stream is mediated by the CU bit of the TC. Coprocessor instructions in the instruction stream associated with the TC reference the bound coprocessor context.

The mechanisms by which coprocessor contexts are bound to TCs are implementation-dependent. It is possible for a coprocessor context to be bound to multiple TCs, as in the case where a single coprocessor context is implemented with a many-to-one mapping from all TCs of a VPE. In such configurations, it is the responsibility of software to coordinate the use of the shared resource by managing the state of CU bits.

The Coprocessor Usable bits $CU_{3..0}$ are instantiated per TC, and are also visible as the $TCU_{3..0}$ bits of the *TCStatus* register (see Section 6.12) of each TC. Access to the coprocessor context bound to a TC is granted to instructions executing on that TC only if the *CU*/*TCU* bit corresponding to the coprocessor is set, otherwise a Coprocessor Unusable exception is delivered to the TC. The FORK operation preserves the CU_x values of each TC, so that bindings between coprocessor contexts and TCs can be preserved across FORK/YIELD 0 thread instantiations.

Coprocessor context state is accessible via MFTR and MTTR instructions which target the TC to which the coprocessor context is bound (see MFTR, MTTR). MFTR and MTTR access is unaffected by the state of *CU* bits, neither those of the TC issuing the MFTR/MTTR (which control access to coprocessors bound to that TC only), nor those of the target TC. Any exceptions enabled, unmasked, or created by MTTR operations on a coprocessor context must be serviced at some appropriate point by the TC to which the coprocessor context is bound, not the TC issuing the MTTR.

While the means of binding coprocessor contexts to thread contexts are coprocessor-specific, a multi-threaded coprocessor must provide sufficient means for diagnostic and operating system software to access selectively any context instantiated on the coprocessor.

Chapter 7

MIPS® MT Restrictions on MIPS32 Implementation

7.1 WAIT Instructions

The nanoMIPS32 ISA allows for implementation-dependent semantics of the WAIT instruction. MIPS MT adds the restriction that a WAIT issued by one TC does *not* shut down the processor or VPE if other TCs are still in a Running state.

7.2 SC Instructions

nanoMIPS32 SC instruction semantics may be extended by MIPS MT gating storage implementations to support "try" operations. See Section A.2 for an example. Gating storage is not cacheable, so LL/SC sequences to gating storage would normally have **UNPREDICTABLE** results in the MIPS32 architecture. MIPS MT gating storage extensions may overload the normal LL/SC semantics, such that the reported success or failure of a conditional store operation is completely independent of any prior LL instructions and/or stores to coherent cacheable (or otherwise "synchronizable") memory.

Any write of the per-TC *TCRestart* CP0 register clears the LLBit. Any write of that register between the execution of a LL instruction and a SC instruction on the target TC, will cause the SC write operation to fail. When a TC is re-assigned to another software thread, the new thread does not inherit the previous state of the LLBit.

7.3 LL Instructions

nanoMIPS32 LL /SC instruction semantics are extended. If per-TC resources are made available within an implementation, it is allowed to have one LL/SC RMW sequence in progress at any one time for each TC. If the implementation does not allow one LL/SC RMW sequence per TC, it must preclude live-lock of LL/SC sequences among the multiple TCs.

7.4 SYNC Instructions

For the MT Module, Cacheability and Coherency Attribute 3, named "Cacheable", is considered coherent among the different threads. For this reason, the ordering and completion rules defined by the SYNC instruction apply to load/store instructions using CCA=3.

Chapter 8

Multiple Virtual Processors in MIPS® MT

8.1 Multi-VPE Processors

A core or processor may implement multiple VPEs sharing resources such as functional units. Each VPE sees its own instantiation of the nanoMIPS32 instruction and privileged resource architecture. Each sees its own register file or TC array, its own CP0 system coprocessor, and its own set of TLB entries. Two VPEs on the same processor can be operated by the same systems software as for a 2-CPU cache-coherent SMP multiprocessor. While each VPE on a processor has a distinct set of CP0 resources, these sets of resources need not be identical. Each must have a minimum complement as defined by those privileged resources which are required by the architecture, but some may have more. The privileged resources of at least one VPE per processor (VPE 0) reset to a sane reset state as per the nanoMIPS32 privileged resource architecture specification.

Each VPE on a processor sees a distinct value in the *EBase.CPUNum* CP0 register field, as if it were a distinct core in a multi-core SoC.

Processor architectural resources such as TC and TLB storage and coprocessors may be statically bound to VPEs in a hard-wired configuration, or they may be configured dynamically in a processor supporting the necessary configuration capability.

8.2 Reset and Virtual Processor Configuration

To be backward compatible with the nanoMIPS32 PRA, a configurably multi-threaded//multi-VPE processor must have a sane and nanoMIPS32-compatible default TC/VPE configuration at reset, that of a single active VPE with a single activated TC.

A VPE has the ability to access and directly manipulate another VPE's processor resources, or to enable or disable another VPE's execution, only if it is a "Master" VPE, designated by having the *VPEConf0.MVP* bit set (see Section 6.6). At reset, only one VPE may have the *MVP* bit set, though implementations may allow it to be set for other VPEs as part of post-reset software configuration. If its *MVP* bit is set, a VPE may:

- Read and write per-TC registers of TCs bound to other VPEs by using MFTR/MTTR instructions with appropriate values in the *TargTC* field of *VPEControl* (see Section 6.5).
- Read and write per-VPE registers of other VPEs by using MFTR/MTTR instructions with values in *TargTC* that correspond to TCs bound to the target VPE (see Section 6.13).
- Set or clear the *EVP* bit of the global *MVPControl* register (see Section 6.2) using MTC0 or DVPE/EVPE instructions.
- Set or clear the *VPA* bit of the per-VPE *VPEConf0* registers using MTTR instructions to put VPEs on or off-line.
- Set or clear the *MVP* bit of other VPEs using MTTR instructions, or clear the local VPE's MVP bit using MTC0.

- Set the *VPC* bit of *MVPControl*, if it is implemented, allowing reconfiguration of processor hardware resources and capabilities.

- Set the *XTC* field of *VPEConf0* of other VPEs (see Section 6.6) using MTTR instructions. *MVP* may also enable the modification of *XTC* of the local VPE's *VPEConf0* register using MTC0 instructions, but such a modification of a running VPE is unsafe and should not be done by software.

If this capability is ignored, as by legacy software, the processor will behave as per specification for the default configuration.

Modification of one VPE's state by another is only guaranteed safe if the *EVP* bit has been cleared and a hazard barrier executed. This applies to both per-VPE state and per-TC state of TCs outside the scope of the modifying TC.

Setting the *MVPControl.VPC* (Virtual Processor Configuration) bit puts the processor into a configuration state in which the contents of certain normally read-only "preset" fields of *Config* and other registers become writable. Implementations may impose restrictions on configuration-state instruction streams; e.g., they may be forbidden to use cached or TLB-mapped memory addresses.

The total number of VPEs is encoded in the *MVPConf0.PVPE* field. VPEs are numbered from 0 to *MVPConf0.PVPE*. A "Master" VPE may select another VPE as a target of an MFTR or MTTR operation by selecting (or setting up) a TC bound to the target VPE, and using that TC as the target of the MFTR/MTTR. If *VPC* is set, the normally read-only register fields outlined in Table 8.1 can potentially be modified by writing to them with MTTR instructions.

Table 8.1 Dynamic Virtual Processor Configuration Options

Register	Field	Meaning	Indicator of Configurability
Config1	MMU_Size	Number of TLB Entry Pairs	MVPConf0 PTLBE > 0
Config1	C2	Coprocessor 2 Present	MVPConf1 PCP2 > 0
Config1	MD	Media Accelerator Present	MVPConf1 PCP1 > 0 and MVPConf1 C1M = 1
Config1	FP	FPU Present	MVPConf1 PCP1 > 0 and MVPConf1 C1F = 1
MVPControl	STLB	TLB Shared across VPES	MVPConf0 TLBS = 1
VPEConf1	NCP1	Number of FP/Media Coprocessor contexts available	MVPConf1 PCP1 > 0
VPEConf1	NCP2	Number of Coprocessor 2 Contexts available	MVPConf1 PCP2 > 0
VPEConf1	NCX	Number of CorExtend Contexts available	MVPConf1 PCx > 0
TCBind	CurVPE	VPE binding of TC	MVPConf0 TCA = 1

Not all of the above configuration parameters need be configurable. For example, the number of TLB entries per VPE may be fixed, FPUs may be pre-allocated and hard-wired per VPE, etc. Statically assigned resources are reflected in the reset-time values in the *Config*, *Config1*, *VPEConf*, and *TCBind* registers. The existence of dynamically assignable resources is indicated in the *MVPConf0* and *MVPConf1* registers, and these resources are assigned to VPEs by writing new values to the *Config* and *VPConf* registers that reflect the allocation of resources. In the event that an implementation cannot provide the resource allocation or configuration implied by a write to one of the per-VPE configurable fields (e.g., if TLB entries are assignable only in blocks of 4, and an attempt is made to allocate 18 entry pairs to a VPE), a subsequent read will reflect the actual resource configuration. If a field containing a quantitative value is written to an implementation which cannot support that value, the implementation will set and subsequently return a supported value. It is recommended that the value be as close to the requested value as the implementation can provide.

A VPE is enabled for execution by setting the *VPEConf0.VPA* activation bit with an MTTR to that register.

The configuration state is exited by clearing *MVPControl.VPC*, which makes the configuration register fields read-only with their new values. Multi-VPE execution is enabled by setting *MVPControl.EVP*, either explicitly or via an EVPE instruction. This causes all Activated VPEs to begin fetching and executing concurrently. If a VPE's *MVP* bit is cleared, the *VPC* and *EVP* bits can no longer be manipulated by that VPE. If *MVP* is cleared for all VPEs, the processor configuration is effectively frozen until the next processor reset. If *MVP* remains set, an operating system may re-enter the configuration mode by clearing *EVP* (to stop other VPEs from running concurrently) and again setting the *VPC* bit.

8.3 MIPS® MT and Cache Configuration

Whether or not cache tags and data can be shared between VPEs is implementation-dependent. Simultaneous line-locking by multiple VPEs sharing a cache may result in undesirable behavior. Sharing of virtually tagged caches by multiple VPEs implies that a VPE number or other unique VPE tag must be concatenated with the *ASID* in the cache tags. Cache errors in shared caches must be signalled to all VPEs sharing the cache (see Section 4.7).

CACHE instruction operations in MIPS MT processors must be atomic with respect to concurrent threads of execution; e.g., a load from one TC must not be allowed to reference a memory location between its invalidation in the cache and its write-back to memory due to a writeback-invalidate CACHE instruction from another TC.

Chapter 9

Data-Driven Scheduling of MIPS® MT Threads

Multithreaded execution models lend themselves to data-driven algorithms, where the availability or absence of data in a storage or I/O location determines whether or not an instruction stream can advance. This paradigm requires some architectural and nanoarchitectural support.

9.1 Gating Storage

Gating Storage is an attribute of memory which may optionally be supported by processors implementing the MT Module. The user-mode load/store semantics of gating storage are identical with those of normal memory, except that completion of the operation may be blocked for unbounded periods of time. The distinguishing feature of gating storage is that outstanding load or store operations can be aborted and restarted. It is a TLB-mediated property of a virtual page whether or not a location is treated as gating storage. Gating storage support may be restricted to certain ranges of physical addresses, and may require special page attributes in some implementations, but any mapped virtual page may resolve to gating storage.

When a load or store operation is performed on gating storage, no instructions beyond the load/store in program order are allowed to alter the software-visible state of the system until a load result, a store confirmation, or an exception is returned from storage. An exception returned by gating storage logic in response to a load or store is delivered as a Thread exception on the load or store, with a value of 3 in the *EXCPT* field of the *VPEControl* register to indicate the Gating Storage exception (see Section 4.3). In the event that an exception is taken using the TC of an instruction stream which is blocked on a load/store to gating storage, whether or not that exception originates from the gating storage logic, or in the event where such a thread is halted by setting the *H* bit of the *TCHalt* register of the associated TC, the pending load/store operation is aborted.

If both the *GSI* bit of the *VPEControl* register and the *DT* bit of the *TCStatus* register are set when a load or store operation from the associated VPE is determined to be blocked by gating storage, a Thread exception is delivered on the load/store, preempting the memory operation, with a value of 5 in the *EXCPT* field of *VPEControl* to indicate a GS Scheduler exception, which allows a software scheduler to take control of the VPE and override the default hardware scheduling logic. The conditioning of *GSI* by the *DT* bit allows software to explicitly allow a blocking gating storage reference to be resumed without causing an exception, by clearing *DT* before restarting the TC.

When a load or store is aborted, the abort is signalled to the storage subsystem, such that the operation can unambiguously either complete or be abandoned without any side-effects. If a load operation is abandoned, any hardware interlocks on the load dependence are released, so that the destination register can be used as an operand source, with its pre-load value.

On an exception resulting in an aborted and abandoned load/store, the program counter as seen by the *EPC* register and the branch delay state as seen by the *Cause.BD* bit are set so that the execution of an ERET by the instruction stream associated with the TC, or a clearing of the TC halted state, will cause a re-issue of the gating load/store.

Gating storage accesses are never cached, and multiple stores to a gating storage address are never merged by a processor.

Chapter 10

EJTAG and MIPS® MT

10.2 EJTAG Debug Resources

The MIPS EJTAG resources are instantiated per VPE, with the exception of the *Debug* register. The *SSt* and *OffLine* bits of the *Debug* register are instantiated per TC. MFC0s and MTC0s of the *Debug* register reference the *SSt* and *OffLine* bit values corresponding to the bits of the TC issuing the MFC0, with the rest of the register field values being those of the VPE to which the issuing TC is bound. MFTRs and MTTRs of the *Debug* register of the target TC reference the *Debug* register as seen by the target TC: the *SSt* and *OffLine* bits are those of the target TC, and the rest of the register field values are those of the VPE to which the target TC is bound at the time the MFTR/MTTR is issued.

The *SSt* bit state is unaffected by a FORK instruction.

It is implementation-dependent whether EJTAG hardware breakpoint facilities are instantiated per-VPE or shared. If they are shared, however, the associated Debug exceptions must be delivered to the VPE containing the TC which triggered the breakpoint.

10.3 Debug Exception Handling

EJTAG Debug exception handling overrides the basic thread scheduling mechanisms of MIPS MT. When a Debug exception occurs, all thread scheduling is suspended across all VPEs of a processor until Debug mode is cleared. The *XTC* fields of the *VPEConf0* registers are not affected. If a TC is executing in Debug mode, its Activated and Halted states are ignored, as are the effects of any DMT or DVPE instruction issued by another TC which may have caused it to be suspended. This concerns mostly asynchronous Debug exceptions (see below), but it also resolves any races between a TC being Halted or de-Activated by the action of another TC and the dispatch of a synchronous Debug exception. A DERET by an otherwise Halted TC is an implicit instruction hazard barrier, so that even if the first instruction dispatched by the multi-threading scheduler is an MFTR access to the Halted TC, the per-TC state is stable.

So long as any VPE is running in Debug mode, asynchronous Debug exception requests, e.g., DINT, are ignored by all VPEs of a processor.

If the *SSt* bit of a TC is set, a Debug exception will be taken by that TC after any non-Debug mode instruction is executed. Other TCs with *SSt* cleared are scheduled and issue instructions normally according to the scheduling policy in force. Global single-step operation of a VPE can be achieved by setting *SSt* for all TCs.

Debug exceptions from data-value EJTAG hardware breakpoints are treated as asynchronous exceptions by a MIPS MT processor, as imprecise synchronous exceptions are not permitted.

Asynchronous Debug exceptions such as DINT and data-value breakpoints may be serviced by any TC that is bound to the VPE taking the exception, as the hardware implementation sees fit. This includes TCs that are otherwise Halted, non-Activated, off-line via the *Debug* register *OffLine* bit or bound for use as shadow register sets. This

allows an EJTAG debugger to get control of VPEs that are otherwise locked-up due to programming errors that result in no schedulable TCs on the VPE.

While entry into Debug mode does not affect any software-visible MIPS MT state, execution in Debug mode confers privilege equivalent to the *MVP* bit being set in the *VPEConf0* register.

Appendix A

Inter-Thread Communication Storage

Inter-Thread Communication (ITC) Storage is a Gating Storage capability which provides an alternative to Load-Linked/Store-Conditional synchronization for fine-grained multi-threading. It is invisible to the instruction set architecture, as it is manipulated by loads and stores, but it is visible to the Privileged Resource Architecture.

A.1 Basic Concepts

As described in the Gating Storage chapter of this specification, the fundamental property of Gating Storage is that it synchronizes execution streams. Loads and stores to/from gating storage may block unless and until the state of the storage location corresponds to some set of required conditions for completion. A blocked load or store can be precisely aborted if necessary, and restarted by the controlling operating system if appropriate.

The main chapters of this specification goes no further in defining Gating Storage semantics. This appendix describes a reference ITC storage model, an instance of Gating Storage which provides lightweight support for a number of standard interprocessor and interprocess communication and synchronization primitives.

References to memory pages which map to ITC storage resolve not to main memory, but to a gating store with special attributes. Each page maps a set of 1 to 32 64-bit storage locations, called *cells*, each of which can be accessed in one of 16 ways, called *views*, using standard load and store instructions. The view is encoded in the low order (and untranslated) bits 6:3 of the generated memory address, such that the successive views of a cell correspond to successive 64-bit-aligned addresses.

A.2 An ITC Storage Reference Model

In the MIPS MT ITC reference model, each cell of the ITC store has Empty and Full boolean states associated with it in addition to the data value of the cell. The cell views are then defined by Table A.1.

Table A.1 ITC Reference Cell Views

Address Bits 6:3 Value	ITC Storage Behavior	
2#0000	**Bypass**. Loads and stores do not block, and do not affect Empty/Full states.	
2#0001	**Control**. Read or Write of Status/Control Information:	
	Data Bit(s)	**Meaning**
	0	If set, cell is Empty and will block on an attempt to load as synchronized storage.
	1	If set, cell is Full and will block on an attempt to store as synchronized storage.
	15:2	Reserved for future architectural definition
	63:16	Implementation Dependent State

Table A.1 ITC Reference Cell Views

Address Bits 6:3 Value	ITC Storage Behavior
2#0010	**Empty/Full Synchronized** view. Loads will cause the issuing thread to block if cell is Empty, and set the Empty state on returning the last available load value. Stores will block if the cell is Full, and set the Full state on the cell accepting the last possible store value. Minimally, a cell can contain a single value.
2#0011	**Empty/Full "Try"** view. Loads will return a value of zero if cell is Empty, regardless of the actual data contained. Otherwise load behavior is same as in Empty/Full Synchronized view. Normal stores to Full locations through the E/F Try view fail silently to update the contents of the cell, rather than block the thread of execution. SC (Store Conditional) instructions referencing the E/F Try view will indicate success or failure based solely upon whether the ITC store succeeds or fails due to the Full state. Otherwise store behavior is same as in Empty/Full Synchronized view.
2#0100	**P/V Synchronized** view. Loads return the current cell data value if the value is non-zero, and cause an atomic post-decrement of the cell value. If the cell value is zero, loads block until the cell takes a non-zero value. Stores cause an atomic increment of the cell value, up to a maximal value at which they saturate, regardless of the register value stored. P/V loads and stores do not modify the Empty and Full bits, both of which should be cleared as part of cell initialization for P/V semaphore use. The width of the incremented/decremented field within the ITC cell need not be the full 32 or 64-bit width of the cell. It must, however, implement at least 15 bits of unsigned value. Bits more significant than the width of the incremented/decremented field are ignored for the purposes of computing zero/non-zero values in P/V operations.
2#0101	**P/V "Try"** view. Loads return the current cell data value, even if zero. If the load value is non-zero, an atomic post-decrement is performed on the cell value. Stores cause a saturating atomic increment of the cell value, as described for the P/V Synchronized view, and cannot fail. Loads and stores do not modify the Empty and Full bits, both of which should be cleared as part of cell initialization for P/V semaphore use.
2#0110	Architecturally Reserved View 0
2#0111	Architecturally Reserved View 1
2#1000	Architecturally Reserved View 2
2#1001	Architecturally Reserved View 3
2#1010	Architecturally Reserved View 4
2#1011	Architecturally Reserved View 5
2#1100	Architecturally Reserved View 6
2#1101	Architecturally Reserved View 7
2#1110	Architecturally Reserved View 8
2#1111	Architecturally Reserved View 9

Each storage cell could thus be described by the C structure:

```
struct {
   uint64 bypass_cell;
   uint64 ctl_cell;
   uint64 ef_sync_cell;
   uint64 ef_try_cell;
   uint64 pv_sync_cell;
   uint64 pv_try_cell;
   uint64 res_arch[10];
} ITC_cell;
```

Where all of the defined elements except ctl_cell reference the same underlying storage. implementation-dependent views may reference additional per-cell state. References to the cell storage may have access types of less than the cell data width (e.g., LW, LH, LB), with the same Empty/Full and semaphore protocols being enforced on a

per-access basis. Store/Load pairs of the same data type to a given ITC address will always reference the same data, but the byte and halfword ordering within words, and the word ordering within 64-bit doublewords, may be implementation and endianness-dependent, i.e., a SW followed by a LB from the same ITC address is not guaranteed to be portable. The effect of writing less than the implemented width of the control view of an ITC cell is implementation-dependent, and such stores may have **UNPREDICTABLE** results.

While the design of ITC storage allows references to be expressed in terms of C language constructs, compiler optimizations may generate sequences that break ITC protocols, and great care must be taken if ITC is directly referenced as “memory” in a high-level language.

Systems which do not support 64-bit loads and stores need not implement all 64 bits of each ITC cell as storage. If only 32 bits of storage are instantiated per cell, it must be visible in the least significant 32-bit word of each view, regardless of the endinanness of the processor. The results of referencing the most significant 32 bits of such a cell view are implementation-dependent. These requirements can be satisfied by ignoring the 2^2 bit of the address on each access. In this way a C language cast from a uint64 to a uint32 reference will acquire the data in both big-endian and little-endian CPU configurations.

Empty and Full bits are distinct so that decoupled multi-entry data buffers, such as FIFOs can be mapped into ITC storage.

ITC storage can be saved and restored by copying the {bypass_cell, ctl_cell} pair to and from general storage. In the case of multi-entry FIFO data buffers, each cell must be read using and Empty/Full view until the Control view shows the cell to be Empty to drain the buffer on a copy. The FIFI state can then be restored by performing a series of Empty/Full stores to an equivalent FIFO cell, starting in an Empty state. Implementations may provide depth counters in the implementation-specific bits of the Control view to optimize this process.

The “Try” view exploits the ability of the standard MIPS32 SC instructions to indicate failure of a store operation. The behavior of conditional stores to non-Try ITC views is implementation-dependent.

A.3 Multiprocessor/Multicore ITC

ITC storage may be strictly local to a processor/core or it may be shared across multiple processors. The “physical address space” of shared ITC storage should be consistent across all processors sharing the storage. Processors or cores designed for uniprocessor applications need not export a physical interface to the ITC storage, and can treat it as a processor-internal resource.

A.4 Interaction with EJTAG Debug Facilities

The Debug state of a processor is not visible to ITC storage logic, and no exceptions are made for Debug mode execution. If a load or store is issued by a processor in Debug mode to an ITC cell view which stalls, the processor is effectively halted until an exception of sufficiently high priority is delivered to the processor.

Appendix B

Revision History

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

Please note: Limitations on the authoring tools make it difficult to place change bars on changes to figures. Change bars on figure titles are used to denote a potential change in the figure itself. Certain parts of this document (Instruction set descriptions, EJTAG register definitions) are references to Architecture specifications, and the change bars within these sections indicate alterations since the previous version of the relevant Architecture document.

Revision	Date	Description
1.00	September 28, 2005	First official release
1.01	July 28, 2006	Converted to nB1.01 template.
1.02	January 25, 2007	Clarify Status.IXMT definition and converge MIPS64 and MIPS32 semantics for MFTR and MTTR.
1.04	June 25, 2008	• Add UserLocal to set of non-MIPS MT CP0 resources replicated per TC, and add copy of UserLocal to FORK semantics. • Section 5 - Write of TCRestart register clears LLBit. • Section 5 - multiple LL/SC RMW sequences allowed for multi-TC implementations. • Section 5 - SYNC instruction applies to load/store instructions using CCA3
1.05	June 25, 2009	• VPEOpt Table 4.9 - the DWX bits were mislabeled as IWX. • Added warnings on using MTTR, MFTR instructions on non-HALTED TCs - might stall indefinitely.
1.06	April 05, 2010	• Make Gating Storage text less MT specific, can be also used by MP systems as well. • Added “About This Book” and “Guide to ISA” chapters. • Added TCOpt Register. • nanoMIPS edits.
1.10	December 14, 2012	• TCSTatus.TFR bit is inherited from Forking thread. • R5 changes - MT ASE now MT Module • MVPControl.STLB - all VPEs now use same SegCtl programmed values when using Shared TLB. • Add restrictions to MTTR and MFTR instructions when dealing with 64-bit FPU. Clean-up of pseudo-code when dealing with 32/64-bit FPRs.
1.11	December 16, 2012	• No Technical content changed: • Update logos on Cover page • Update copyright text.
1.12	July 16, 2013	• New cover page and legal text.
1.13	August 30, 2017	• New cover page and legal text. • Updated page template. • Updated DMT/EMT and DVPE/EVPE for nanoMIPS,

Revision	Date	Description
1.14	January 29, 2018	• MFTR rt, rs, u, sel, h (rt is destination) • MTTR rt, rs, u, sel, h (rt may be destination, rs is source) • YIELD rt, rs (rt is destination)
1.15	February 20, 2018	• MFTR: fixed typo in the opcode table • MTTR: exchanged rs and rt in the pseudocode to be consistent with MD01247
1.16	March 26, 2018	• Updated the Yield instruction.
1.17	April 27, 2018	• Changed confidentiality level to Public.