Table of Contents

Chapter 1 About This Book ................................................................................................................................. 1
  1.1 Typographical Conventions .......................................................................................................................... 1
    1.1.1 Italic Text ............................................................................................................................................. 1
    1.1.2 Bold Text ............................................................................................................................................ 1
    1.1.3 Courier Text ....................................................................................................................................... 1
  1.2 UNPREDICTABLE and UNDEFINED .............................................................................................................. 2
    1.2.1 UNPREDICTABLE ........................................................................................................................... 2
    1.2.2 UNDEFINED ..................................................................................................................................... 2
  1.3 Special Symbols in Pseudocode Notation ........................................................................................................ 2
  1.4 For More Information .................................................................................................................................... 5

Chapter 2 Guide to the Instruction Set .................................................................................................................. 7
  2.1 Understanding the Instruction Fields ............................................................................................................. 7
    2.1.1 Instruction Fields ............................................................................................................................... 8
    2.1.2 Instruction Descriptive Name and Mnemonic ..................................................................................... 9
    2.1.3 Format Field ....................................................................................................................................... 9
    2.1.4 Purpose Field .................................................................................................................................... 10
    2.1.5 Description Field .............................................................................................................................. 10
    2.1.6 Restrictions Field ............................................................................................................................. 10
    2.1.7 Operation Field ................................................................................................................................ 11
    2.1.8 Exceptions Field ............................................................................................................................... 11
    2.1.9 Programming Notes and Implementation Notes Fields .................................................................... 11
  2.2 Operation Section Notation and Functions ................................................................................................... 12
    2.2.1 Instruction Execution Ordering .......................................................................................................... 12
    2.2.2 Pseudocode Functions ....................................................................................................................... 12
  2.3 Op and Function Subfield Notation ............................................................................................................... 20
  2.4 FPU Instructions .......................................................................................................................................... 20

Chapter 3 The MIPS32™ Instruction Set .............................................................................................................. 21
  3.1 Compliance and Subsetting .......................................................................................................................... 21
  3.2 Alphabetical List of Instructions .................................................................................................................. 21
    ABS.fmt ....................................................................................................................................................... 30
    ADD .......................................................................................................................................................... 31
    ADD.fmt .................................................................................................................................................... 33
    ADDI ......................................................................................................................................................... 34
    ADDIU ....................................................................................................................................................... 35
    ADDU ......................................................................................................................................................... 36
    AND ........................................................................................................................................................ 37
    ANDI ......................................................................................................................................................... 38
    B .............................................................................................................................................................. 39
    BAL .......................................................................................................................................................... 40
    BC1F ......................................................................................................................................................... 41
    BC1FL ....................................................................................................................................................... 43
    BC1T ........................................................................................................................................................ 45
    BC1TL ...................................................................................................................................................... 47
    BC2F ......................................................................................................................................................... 49
    BC2FL ...................................................................................................................................................... 50
    BC2T ........................................................................................................................................................ 52
    BC2TL ...................................................................................................................................................... 53
    BEQ .......................................................................................................................................................... 55
    BEQL ......................................................................................................................................................... 56
    BGEZ ......................................................................................................................................................... 58
    BGEZAL .................................................................................................................................................... 59
    BGEZALL .................................................................................................................................................. 60
    BGEZL ....................................................................................................................................................... 62
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>XOR</td>
<td>243</td>
</tr>
<tr>
<td>WAIT</td>
<td>241</td>
</tr>
<tr>
<td>TNEI</td>
<td>238</td>
</tr>
<tr>
<td>TLTU</td>
<td>236</td>
</tr>
<tr>
<td>TLBP</td>
<td>235</td>
</tr>
<tr>
<td>TLBR</td>
<td>236</td>
</tr>
<tr>
<td>TLBWI</td>
<td>237</td>
</tr>
<tr>
<td>TLTU</td>
<td>237</td>
</tr>
<tr>
<td>TNEI</td>
<td>238</td>
</tr>
<tr>
<td>TRUNC_W.fmt</td>
<td>239</td>
</tr>
<tr>
<td>WAIT</td>
<td>241</td>
</tr>
<tr>
<td>XOR</td>
<td>243</td>
</tr>
<tr>
<td>Xori</td>
<td>244</td>
</tr>
</tbody>
</table>

Appendix A Revision History ........................................................................................................ 245
List of Tables

Table 1-1: Symbols Used in Instruction Operation Statements .......................................................... 3
Table 2-1: AccessLength Specifications for Loads/Stores .................................................................. 16
Table 3-1: CPU Arithmetic Instructions ........................................................................................ 22
Table 3-2: CPU Branch and Jump Instructions ................................................................................ 22
Table 3-3: CPU Instruction Control Instructions ............................................................................. 23
Table 3-4: CPU Load, Store, and Memory Control Instructions ...................................................... 23
Table 3-5: CPU Logical Instructions .................................................................................................. 24
Table 3-6: CPU Move Instructions ........................................................................................................... 24
Table 3-7: CPU Shift Instructions ........................................................................................................ 24
Table 3-8: CPU Trap Instructions ......................................................................................................... 25
Table 3-9: Obsolete CPU Branch Instructions .................................................................................. 25
Table 3-10: FPU Arithmetic Instructions ......................................................................................... 26
Table 3-11: FPU Branch Instructions .................................................................................................. 26
Table 3-12: FPU Compare Instructions ............................................................................................ 26
Table 3-13: FPU Convert Instructions .................................................................................................. 26
Table 3-14: FPU Load, Store, and Memory Control Instructions ............................................................ 27
Table 3-15: FPU Move Instructions ....................................................................................................... 27
Table 3-16: Obsolete FPU Branch Instructions .................................................................................. 27
Table 3-17: Coprocessor Branch Instructions ....................................................................................... 27
Table 3-18: Coprocessor Execute Instructions ..................................................................................... 27
Table 3-19: Coprocessor Load and Store Instructions ......................................................................... 28
Table 3-20: Coprocessor Move Instructions .......................................................................................... 28
Table 3-21: Obsolete Coprocessor Branch Instructions ..................................................................... 28
Table 3-22: Privileged Instructions ....................................................................................................... 28
Table 3-23: EJTAG Instructions .......................................................................................................... 29
Table 3-24: FPU Comparisons Without Special Operand Exceptions .................................................. 81
Table 3-25: FPU Comparisons With Special Operand Exceptions for QNaNs ........................................ 82
Table 3-26: Usage of Effective Address ............................................................................................. 85
Table 3-27: Encoding of Bits[17:16] of CACHE Instruction ................................................................. 86
Table 3-28: Encoding of Bits [20:18] of the CACHE Instruction ............................................................. 87
Table 3-29: Values of the hint Field for the PREF Instruction ............................................................ 179
Chapter 1

About This Book

The MIPS32™ Architecture For Programmers Volume II comes as a multi-volume set.

- Volume I describes conventions used throughout the document set, and provides an introduction to the MIPS32™ Architecture
- Volume II provides detailed descriptions of each instruction in the MIPS32™ instruction set
- Volume III describes the MIPS32™ Privileged Resource Architecture which defines and governs the behavior of the privileged resources included in a MIPS32™ processor implementation
- Volume IV-a describes the MIPS16™ Application-Specific Extension to the MIPS32™ Architecture
- Volume IV-b describes the MDMX™ Application-Specific Extension to the MIPS32™ Architecture and is not applicable to the MIPS32™ document set
- Volume IV-c describes the MIPS-3D™ Application-Specific Extension to the MIPS64™ Architecture and is not applicable to the MIPS32™ document set
- Volume IV-d describes the SmartMIPS™ Application-Specific Extension to the MIPS32™ Architecture

1.1 Typographical Conventions

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

1.1.1 Italic Text

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

1.1.2 Bold Text

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

1.1.3 Courier Text

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

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

1.2.1 UNPREDICTABLE

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

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

1.2.2 UNDEFINED

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

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

1.3 Special Symbols in Pseudocode Notation

In this book, algorithmic descriptions of an operation are described as pseudocode in a high-level language notation resembling Pascal. Special symbols used in the pseudocode notation are listed in Table 1-1.
## 1.3 Special Symbols in Pseudocode Notation

Table 1-1 Symbols Used in Instruction Operation Statements

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>←</td>
<td>Assignment</td>
</tr>
<tr>
<td>=, ≠</td>
<td>Tests for equality and inequality</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>x³</td>
<td>A y-bit string formed by y copies of the single-bit value x</td>
</tr>
<tr>
<td>b#n</td>
<td>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.</td>
</tr>
<tr>
<td>x₂…ₙ</td>
<td>Selection of bits y through n of bit string x. Little-endian bit notation (rightmost bit is 0) is used. If y is less than n, this expression is an empty (zero length) bit string.</td>
</tr>
<tr>
<td>+, −</td>
<td>2’s complement or floating point arithmetic: addition, subtraction</td>
</tr>
<tr>
<td>*</td>
<td>2’s complement or floating point multiplication (both used for either)</td>
</tr>
<tr>
<td>div</td>
<td>2’s complement integer division</td>
</tr>
<tr>
<td>mod</td>
<td>2’s complement modulo</td>
</tr>
<tr>
<td>/</td>
<td>Floating point division</td>
</tr>
<tr>
<td>&lt;</td>
<td>2’s complement less-than comparison</td>
</tr>
<tr>
<td>&gt;</td>
<td>2’s complement greater-than comparison</td>
</tr>
<tr>
<td>≤</td>
<td>2’s complement less-than or equal comparison</td>
</tr>
<tr>
<td>≥</td>
<td>2’s complement greater-than or equal comparison</td>
</tr>
<tr>
<td>nor</td>
<td>Bitwise logical NOR</td>
</tr>
<tr>
<td>xor</td>
<td>Bitwise logical XOR</td>
</tr>
<tr>
<td>and</td>
<td>Bitwise logical AND</td>
</tr>
<tr>
<td>or</td>
<td>Bitwise logical OR</td>
</tr>
<tr>
<td>GPRLEN</td>
<td>The length in bits (32 or 64) of the CPU general-purpose registers</td>
</tr>
<tr>
<td>GPR[x]</td>
<td>CPU general-purpose register x. The content of GPR[0] is always zero.</td>
</tr>
<tr>
<td>FPR[x]</td>
<td>Floating Point operand register x</td>
</tr>
<tr>
<td>FCC[CC]</td>
<td>Floating Point condition code CC. FCC[0] has the same value as COC[1].</td>
</tr>
<tr>
<td>FPR[x]</td>
<td>Floating Point (Coprocessor unit 1), general register x</td>
</tr>
<tr>
<td>CPR[xy]</td>
<td>Coprocessor unit x, general register x, select s</td>
</tr>
<tr>
<td>CCR[xy]</td>
<td>Coprocessor unit x, control register x</td>
</tr>
<tr>
<td>COC[z]</td>
<td>Coprocessor unit z condition signal</td>
</tr>
<tr>
<td>Xlat[x]</td>
<td>Translation of the MIPS16 GPR number x into the corresponding 32-bit GPR number</td>
</tr>
<tr>
<td>BigEndianMem</td>
<td>Endian mode as configured at chip reset (0 → Little-Endian, 1 → Big-Endian). Specifies the endianness of the memory interface (see LoadMemory and StoreMemory pseudocode function descriptions), and the endianness of Kernel and Supervisor mode execution.</td>
</tr>
</tbody>
</table>
Chapter 1 About This Book

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 in the Status register. Thus, ReverseEndian may be computed as (SRRE 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; it is tested and cleared by the conditional store. It is cleared, during other CPU operation, when a store to the location would no longer be atomic. In particular, it is cleared by exception return instructions.

I, I+n, I-n:

This occurs as a prefix to Operation description lines and functions as a label. It indicates the instruction time during which the pseudocode appears to “execute.” Unless otherwise indicated, all effects of the current instruction appear to occur during the instruction time of the current instruction. No label is equivalent to a time label of I. Sometimes effects of an instruction appear to occur either earlier or later — that is, during the instruction time of another instruction. When this happens, the instruction operation is written in sections labeled with the instruction time, relative to the current instruction I, in which the effect of that pseudocode appears to occur. For example, an instruction may have a result that is not available until after the next instruction. Such an instruction has the portion of the instruction operation description that writes the result register in a section labeled I+1.

The effect of pseudocode statements for the current instruction labelled I+1 appears to occur “at the same time” as the effect of pseudocode statements labelled 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 MIPS16 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.

PABITS

The number of physical address bits implemented is represented by the symbol PABITS. As such, if 36 physical address bits were implemented, the size of the physical address space would be $2^{PABITS} = 2^{36}$ bytes.

FP32RegistersMode

Indicates whether the FPU has 32-bit or 64-bit floating point registers (FPRs). In MIPS32, the FPU has 32 32-bit FPRs in which 64-bit data types are stored in even-odd pairs of FPRs. In MIPS64, the FPU has 32 64-bit FPRs in which 64-bit data types are stored in any FPR.

In MIPS32 implementations, FP32RegistersMode is always a 0. MIPS64 implementations have a compatibility mode in which the processor references the FPRs as if it were a MIPS32 implementation. In such a case FP32RegisterMode is computed from the FR bit in the Status register. If this bit is a 0, the processor operates as if it had 32 32-bit FPRs. If this bit is a 1, the processor operates with 32 64-bit FPRs.

The value of FP32RegistersMode is computed from the FR bit in the Status register.

InstructionInBranchDelaySlot

Indicates whether the instruction at the Program Counter address was executed in the delay slot of a branch or jump. This condition reflects the dynamic state of the instruction, not the static state. That is, the value is false if a branch or jump occurs to an instruction whose PC immediately follows a branch or jump, but which is not executed in the delay slot of a branch or jump.

SignalException(exception, argument)

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

---

Table 1-1 Symbols Used in Instruction Operation Statements

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>BigEndianCPU</td>
<td>The endianness for load and store instructions (0 → Little-Endian, 1 → Big-Endian). In User mode, this endianness may be switched by setting the RE bit in the Status register. Thus, BigEndianCPU may be computed as (BigEndianMem XOR ReverseEndian).</td>
</tr>
<tr>
<td>ReverseEndian</td>
<td>Signal to reverse the endianness of load and store instructions. This feature is available in User mode only, and is implemented by setting the RE bit in the Status register. Thus, ReverseEndian may be computed as (SRRE and User mode).</td>
</tr>
<tr>
<td>LLbit</td>
<td>Bit of virtual state used to specify operation for instructions that provide atomic read-modify-write. LLbit is set when a linked load occurs; it is tested and cleared by the conditional store. It is cleared, during other CPU operation, when a store to the location would no longer be atomic. In particular, it is cleared by exception return instructions.</td>
</tr>
<tr>
<td>I, I+n, I-n:</td>
<td>This occurs as a prefix to Operation description lines and functions as a label. It indicates the instruction time during which the pseudocode appears to “execute.” Unless otherwise indicated, all effects of the current instruction appear to occur during the instruction time of the current instruction. No label is equivalent to a time label of I. Sometimes effects of an instruction appear to occur either earlier or later — that is, during the instruction time of another instruction. When this happens, the instruction operation is written in sections labeled with the instruction time, relative to the current instruction I, in which the effect of that pseudocode appears to occur. For example, an instruction may have a result that is not available until after the next instruction. Such an instruction has the portion of the instruction operation description that writes the result register in a section labeled I+1. The effect of pseudocode statements for the current instruction labelled I+1 appears to occur “at the same time” as the effect of pseudocode statements labelled I for the following instruction. Within one pseudocode sequence, the effects of the statements take place in order. However, between sequences of statements for different instructions that occur “at the same time,” there is no defined order. Programs must not depend on a particular order of evaluation between such sections.</td>
</tr>
<tr>
<td>PC</td>
<td>The Program Counter value. During the instruction time of an instruction, this is the address of the instruction word. The address of the instruction that occurs during the next instruction time is determined by assigning a value to PC during an instruction time. If no value is assigned to PC during an instruction time by any pseudocode statement, it is automatically incremented by either 2 (in the case of a 16-bit MIPS16 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.</td>
</tr>
<tr>
<td>PABITS</td>
<td>The number of physical address bits implemented is represented by the symbol PABITS. As such, if 36 physical address bits were implemented, the size of the physical address space would be $2^{PABITS} = 2^{36}$ bytes.</td>
</tr>
<tr>
<td>FP32RegistersMode</td>
<td>Indicates whether the FPU has 32-bit or 64-bit floating point registers (FPRs). In MIPS32, the FPU has 32 32-bit FPRs in which 64-bit data types are stored in even-odd pairs of FPRs. In MIPS64, the FPU has 32 64-bit FPRs in which 64-bit data types are stored in any FPR. In MIPS32 implementations, FP32RegistersMode is always a 0. MIPS64 implementations have a compatibility mode in which the processor references the FPRs as if it were a MIPS32 implementation. In such a case FP32RegisterMode is computed from the FR bit in the Status register. If this bit is a 0, the processor operates as if it had 32 32-bit FPRs. If this bit is a 1, the processor operates with 32 64-bit FPRs. The value of FP32RegistersMode is computed from the FR bit in the Status register.</td>
</tr>
<tr>
<td>InstructionInBranchDelaySlot</td>
<td>Indicates whether the instruction at the Program Counter address was executed in the delay slot of a branch or jump. This condition reflects the 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.</td>
</tr>
<tr>
<td>SignalException(exception, argument)</td>
<td>Causes an exception to be signaled, using the exception parameter as the type of exception and the argument parameter as an exception-specific argument). Control does not return from this pseudocode function - the exception is signaled at the point of the call.</td>
</tr>
</tbody>
</table>
1.4 For More Information

Various MIPS RISC processor manuals and additional information about MIPS products can be found at the MIPS URL:

http://www.mips.com

Comments or questions on the MIPS32™ Architecture or this document should be directed to

Director of MIPS Architecture
MIPS Technologies, Inc.
1225 Charleston Road
Mountain View, CA 94043

or via E-mail to architecture@mips.com.
Chapter 2

Guide to the Instruction Set

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

2.1 Understanding the Instruction Fields

Figure 2-1 shows an example instruction. Following the figure are descriptions of the fields listed below:

- “Instruction Fields” on page 8
- “Instruction Descriptive Name and Mnemonic” on page 9
- “Format Field” on page 9
- “Purpose Field” on page 10
- “Description Field” on page 10
- “Restrictions Field” on page 10
- “Operation Field” on page 11
- “Exceptions Field” on page 11
- “Programming Notes and Implementation Notes Fields” on page 11
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:
2.1 Understanding the Instruction Fields

• 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](image)

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](image)

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.

**Format:** 
ADD rd, rs, rt  
MIPS32 (MIPS I)

![Figure 2-4 Example of Instruction Format](image)

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. If the instruction was originally defined in the MIPS I through MIPS V levels of the architecture, that information is enclosed in parentheses.

There can be more than one assembler format for each architecture level. Floating point operations on formatted data show an assembly format with the actual assembler mnemonic for each valid value of the fmt field. For example, the ADD.fmt instruction lists both ADD.S and ADD.D.
The assembler format lines sometimes include parenthetical comments to help explain variations in the formats (once again, see C.cond.fmt). These comments are not a part of the assembler format.

### 2.1.4 Purpose Field

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

**Purpose:**

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

*Figure 2-5 Example of Instruction Purpose*

### 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.

**Description:** \( rd \leftarrow rs + rt \)

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 \( rd \)

*Figure 2-6 Example of Instruction Description*

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 DADD)
- 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)
2.1 Understanding the Instruction Fields

Restrictions:
None

Figure 2-7 Example of Instruction Restrictions

2.1.7 Operation Field

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

Operation:
\[
\text{temp} \leftarrow (\text{GPR}[rs]_{31} || \text{GPR}[rs]_{31..0}) + (\text{GPR}[rt]_{31} || \text{GPR}[rt]_{31..0}) \\
\text{if temp}_{32} \neq \text{temp}_{31} \text{ then} \\
\quad \text{SignalException(IntegerOverflow)} \\
\text{else} \\
\quad \text{GPR}[rd] \leftarrow \text{temp} \\
\text{endif}
\]

Figure 2-8 Example of Instruction Operation

See Section 2.2, "Operation Section Notation and Functions" on page 12 for more information on the formal notation used here.

2.1.8 Exceptions Field

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

Exceptions:
Integer Overflow

Figure 2-9 Example of Instruction Exception

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

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

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

**Figure 2-10 Example of Instruction Programming Notes**

### 2.2 Operation Section Notation and Functions

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

This section presents information about the following topics:

- “Instruction Execution Ordering” on page 12
- “Pseudocode Functions” on page 12

#### 2.2.1 Instruction Execution Ordering

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

#### 2.2.2 Pseudocode Functions

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

- “Coprocessor General Register Access Functions” on page 12
- “Load Memory and Store Memory Functions” on page 14
- “Access Functions for Floating Point Registers” on page 16
- “Miscellaneous Functions” on page 18

#### 2.2.2.1 Coprocessor General Register Access Functions

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

**COP_LW**

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

---

MIPS32™ Architecture For Programmers Volume II, Revision 0.95
2.2 Operation Section Notation and Functions

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

Figure 2-11 COP_LW Pseudocode Function

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.

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

Figure 2-12 COP_LD Pseudocode Function

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.

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

Figure 2-13 COP_SW Pseudocode Function

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.
2.2.2.2 Load Memory and Store Memory Functions

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

In the Operation pseudocode for load and store operations, the following functions summarize the handling of virtual addresses and the access of physical memory. The size of the data item to be loaded or stored is passed in the AccessLength field. The valid constant names and values are shown in Table 2-1. The bytes within the addressed unit of memory (word for 32-bit processors or doubleword for 64-bit processors) that are used can be determined directly from the AccessLength and the two or three low-order bits of the address.

AddressTranslation

The AddressTranslation function translates a virtual address to a physical address and its cache coherence algorithm, 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 cache coherence algorithm (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.

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

/* pAddr: physical address */
/* CCA: Cache Coherence Algorithm, the method used to access caches*/
/* 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

Figure 2-15 AddressTranslation Pseudocode Function

LoadMemory

The LoadMemory function loads a value from memory.
This action uses cache and main memory as specified in both the Cache Coherence Algorithm (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.

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: Cache Coherence Algorithm, the 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

Figure 2-16 LoadMemory Pseudocode Function

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 Cache Coherence Algorithm (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.

StoreMemory (CCA, AccessLength, MemElem, pAddr, vAddr)
/* CCA: Cache Coherence Algorithm, 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

Figure 2-17 StoreMemory Pseudocode Function

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.

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

/* CCA: Cache Coherence Algorithm, 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 */
```

eendfunction Prefetch

**Figure 2-18 Prefetch Pseudocode Function**

Table 2-1 lists the data access lengths and their labels for loads and stores.

### Table 2-1 AccessLength Specifications for Loads/Stores

<table>
<thead>
<tr>
<th>AccessLength Name</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOUBLEWORD</td>
<td>7</td>
<td>8 bytes (64 bits)</td>
</tr>
<tr>
<td>SEPTIBYTE</td>
<td>6</td>
<td>7 bytes (56 bits)</td>
</tr>
<tr>
<td>SEXTIBYTE</td>
<td>5</td>
<td>6 bytes (48 bits)</td>
</tr>
<tr>
<td>QUINTIBYTE</td>
<td>4</td>
<td>5 bytes (40 bits)</td>
</tr>
<tr>
<td>WORD</td>
<td>3</td>
<td>4 bytes (32 bits)</td>
</tr>
<tr>
<td>TRIPLEBYTE</td>
<td>2</td>
<td>3 bytes (24 bits)</td>
</tr>
<tr>
<td>HALFWORD</td>
<td>1</td>
<td>2 bytes (16 bits)</td>
</tr>
<tr>
<td>BYTE</td>
<td>0</td>
<td>1 byte (8 bits)</td>
</tr>
</tbody>
</table>

#### 2.2.2.3 Access Functions for Floating Point Registers

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

**ValueFPR**

The ValueFPR function returns a formatted value from the floating point registers.
value ← ValueFPR(fpr, fmt)

/* value: The formatted value from the FPR */
/* fpr: The FPR number */
/* fmt: The format of the data, one of: */
/*  S, D, W, */
/*  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 (fpr0 ≠ 0) then
      valueFPR ← UNPREDICTABLE
    else
      valueFPR ← FPR[fpr+1] || FPR[fpr]
    endif
  DEFAULT:
    valueFPR ← UNPREDICTABLE
endcase
endfunction ValueFPR

Figure 2-19 ValueFPR Pseudocode Function

**StoreFPR**

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

/* fpr: The FPR number */
/* fmt: The format of the data, one of: */
/* S, D, W, */
/* OB, QH, */
/* UNINTERPRETED_WORD, */
/* UNINTERPRETED_DOUBLEWORD */
/* value: The formatted 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 (fpr0 ≠ 0) then
      UNPREDICTABLE
    else
      FPR[fpr] ← value
      FPR[fpr+1] ← value
    endif
endcase

endfunction StoreFPR

Figure 2-20 StoreFPR Pseudocode Function

### 2.2.2.4 Miscellaneous Functions

This section lists miscellaneous functions not covered in previous sections.

**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.

SyncOperation(stype)

/* stype: Type of load/store ordering to perform. */
/* Perform implementation-dependent operation to complete the */
/* required synchronization operation */

endfunction SyncOperation

Figure 2-21 SyncOperation Pseudocode Function

**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.

```pseudocode
SignalException(Exception, argument)
/* Exception: The exception condition that exists. */
/* argument: A exception-dependent argument, if any */
endfunction SignalException

Figure 2-22 SignalException Pseudocode Function
```

**NullifyCurrentInstruction**

The NullifyCurrentInstruction function nullifies the current instruction.

The instruction is aborted. For branch-likely instructions, nullification kills the instruction in the delay slot during its execution.

```pseudocode
NullifyCurrentInstruction()
endfunction NullifyCurrentInstruction

Figure 2-23 NullifyCurrentInstruction Pseudocode Function
```

**CoprocessorOperation**

The CoprocessorOperation function performs the specified Coprocessor operation.

```pseudocode
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

Figure 2-24 CoprocessorOperation Pseudocode Function
```

**JumpDelaySlot**

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

```pseudocode
JumpDelaySlot(vAddr)
/* vAddr: Virtual address */
endfunction JumpDelaySlot

Figure 2-25 JumpDelaySlot Pseudocode Function
```

**FPConditionCode**

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

Figure 2-26 FPConditionCode Pseudocode Function

SetFPConditionCode

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

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

Figure 2-27 SetFPConditionCode Pseudocode Function

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 MIPS16 instructions.

See Section 2.3, "Op and Function Subfield Notation" on page 20 for a description of the op and function subfields.
3.1 Compliance and Subsetting

To be compliant with the MIPS32 Architecture, designs must implement a set of required features, as described in this document set. To allow flexibility in implementations, the MIPS32 Architecture does provide subsetting rules. An implementation that follows these rules is compliant with the MIPS32 Architecture as long as it adheres strictly to the rules, and fully implements the remaining instructions.

The instruction set subsetting rules are as follows:

- All CPU instructions must be implemented - no subsetting is allowed.
- The FPU and related support instructions, including the MOVF and MOVT CPU instructions, may be omitted. Software may determine if an FPU is implemented by checking the state of the FP bit in the Config1 CP0 register. If the FPU is implemented, it must include S, D, and W formats, operate instructions, and all supporting instructions. Software may determine which FPU data types are implemented by checking the appropriate bit in the FIR CP1 register. The following allowable FPU subsets are compliant with the MIPS32 architecture:
  - No FPU
  - FPU with S, D, and W formats and all supporting instructions
- Coprocessor 2 is optional and may be omitted. Software may determine if Coprocessor 2 is implemented by checking the state of the C2 bit in the Config1 CP0 register. If Coprocessor 2 is implemented, the Coprocessor 2 interface instructions (BC2, CFC2, COP2, CTC2, LCD2, LWC2, MFC2, MTC2, SDC2, and SWC2) may be omitted on an instruction by instruction basis.
- Instruction fields that are marked “Reserved” or shown as “0” in the description of that field are reserved for future use by the architecture and are not available to implementations. Implementations may only use those fields that are explicitly reserved for implementation dependent use.
- Supported ASEs are optional and may be subsetted out. If most cases, software may determine if a supported ASE is implemented by checking the appropriate bit in the Config1 or Config3 CP0 register. If they are implemented, they must implement the entire ISA applicable to the component, or implement subsets that are approved by the ASE specifications.
- If any instruction is subsetted out based on the rules above, an attempt to execute that instruction must cause the appropriate exception (typically Reserved Instruction or Coprocessor Unusable).

Supersetting of the MIPS32 ISA is only allowed by adding functions to the SPECIAL2 major opcode or by adding instructions to support Coprocessor 2.

3.2 Alphabetical List of Instructions

Table 3-1 through Table 3-23 provide a list of instructions grouped by category. Individual instruction descriptions follow the tables, arranged in alphabetical order.
### Table 3-1 CPU Arithmetic Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>Add Word</td>
</tr>
<tr>
<td>ADDI</td>
<td>Add Immediate Word</td>
</tr>
<tr>
<td>ADDIU</td>
<td>Add Immediate Unsigned Word</td>
</tr>
<tr>
<td>ADDU</td>
<td>Add Unsigned Word</td>
</tr>
<tr>
<td>CLO</td>
<td>Count Leading Ones in Word</td>
</tr>
<tr>
<td>CLZ</td>
<td>Count Leading Zeros in Word</td>
</tr>
<tr>
<td>DIV</td>
<td>Divide Word</td>
</tr>
<tr>
<td>DIVU</td>
<td>Divide Unsigned Word</td>
</tr>
<tr>
<td>MADD</td>
<td>Multiply and Add Word to Hi, Lo</td>
</tr>
<tr>
<td>MADDU</td>
<td>Multiply and Add Unsigned Word to Hi, Lo</td>
</tr>
<tr>
<td>MSUB</td>
<td>Multiply and Subtract Word to Hi, Lo</td>
</tr>
<tr>
<td>MSUBU</td>
<td>Multiply and Subtract Unsigned Word to Hi, Lo</td>
</tr>
<tr>
<td>MUL</td>
<td>Multiply Word to GPR</td>
</tr>
<tr>
<td>MULT</td>
<td>Multiply Word</td>
</tr>
<tr>
<td>MULTU</td>
<td>Multiply Unsigned Word</td>
</tr>
<tr>
<td>SLT</td>
<td>Set on Less Than</td>
</tr>
<tr>
<td>SLTI</td>
<td>Set on Less Than Immediate</td>
</tr>
<tr>
<td>SLTIU</td>
<td>Set on Less Than Immediate Unsigned</td>
</tr>
<tr>
<td>SLTU</td>
<td>Set on Less Than Unsigned</td>
</tr>
<tr>
<td>SUB</td>
<td>Subtract Word</td>
</tr>
<tr>
<td>SUBU</td>
<td>Subtract Unsigned Word</td>
</tr>
</tbody>
</table>

### Table 3-2 CPU Branch and Jump Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>Unconditional Branch</td>
</tr>
<tr>
<td>BAL</td>
<td>Branch and Link</td>
</tr>
<tr>
<td>BEQ</td>
<td>Branch on Equal</td>
</tr>
<tr>
<td>BGEZ</td>
<td>Branch on Greater Than or Equal to Zero</td>
</tr>
<tr>
<td>BGEZAL</td>
<td>Branch on Greater Than or Equal to Zero and Link</td>
</tr>
<tr>
<td>BGTZ</td>
<td>Branch on Greater Than Zero</td>
</tr>
<tr>
<td>BLEZ</td>
<td>Branch on Less Than or Equal to Zero</td>
</tr>
<tr>
<td>BLTZ</td>
<td>Branch on Less Than Zero</td>
</tr>
</tbody>
</table>
### Table 3-2 CPU Branch and Jump Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLTZAL</td>
<td>Branch on Less Than Zero and Link</td>
</tr>
<tr>
<td>BNE</td>
<td>Branch on Not Equal</td>
</tr>
<tr>
<td>J</td>
<td>Jump</td>
</tr>
<tr>
<td>JAL</td>
<td>Jump and Link</td>
</tr>
<tr>
<td>JALR</td>
<td>Jump and Link Register</td>
</tr>
<tr>
<td>JR</td>
<td>Jump Register</td>
</tr>
</tbody>
</table>

### Table 3-3 CPU Instruction Control Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOP</td>
<td>No Operation</td>
</tr>
<tr>
<td>SSNOP</td>
<td>Superscalar No Operation</td>
</tr>
</tbody>
</table>

### Table 3-4 CPU Load, Store, and Memory Control Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>LB</td>
<td>Load Byte</td>
</tr>
<tr>
<td>LBU</td>
<td>Load Byte Unsigned</td>
</tr>
<tr>
<td>LH</td>
<td>Load Halfword</td>
</tr>
<tr>
<td>LHU</td>
<td>Load Halfword Unsigned</td>
</tr>
<tr>
<td>LL</td>
<td>Load Linked Word</td>
</tr>
<tr>
<td>LW</td>
<td>Load Word</td>
</tr>
<tr>
<td>LWL</td>
<td>Load Word Left</td>
</tr>
<tr>
<td>LWR</td>
<td>Load Word Right</td>
</tr>
<tr>
<td>PREF</td>
<td>Prefetch</td>
</tr>
<tr>
<td>SB</td>
<td>Store Byte</td>
</tr>
<tr>
<td>SC</td>
<td>Store Conditional Word</td>
</tr>
<tr>
<td>SD</td>
<td>Store Doubleword</td>
</tr>
<tr>
<td>SH</td>
<td>Store Halfword</td>
</tr>
<tr>
<td>SW</td>
<td>Store Word</td>
</tr>
<tr>
<td>SWL</td>
<td>Store Word Left</td>
</tr>
<tr>
<td>SWR</td>
<td>Store Word Right</td>
</tr>
<tr>
<td>SYNC</td>
<td>Synchronize Shared Memory</td>
</tr>
</tbody>
</table>
### Table 3-5 CPU Logical Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND</td>
<td>And</td>
</tr>
<tr>
<td>ANDI</td>
<td>And Immediate</td>
</tr>
<tr>
<td>LUI</td>
<td>Load Upper Immediate</td>
</tr>
<tr>
<td>NOR</td>
<td>Not Or</td>
</tr>
<tr>
<td>OR</td>
<td>Or</td>
</tr>
<tr>
<td>ORI</td>
<td>Or Immediate</td>
</tr>
<tr>
<td>XOR</td>
<td>Exclusive Or</td>
</tr>
<tr>
<td>XORI</td>
<td>Exclusive Or Immediate</td>
</tr>
</tbody>
</table>

### Table 3-6 CPU Move Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>MFHI</td>
<td>Move From HI Register</td>
</tr>
<tr>
<td>MFLO</td>
<td>Move From LO Register</td>
</tr>
<tr>
<td>MOVF</td>
<td>Move Conditional on Floating Point False</td>
</tr>
<tr>
<td>MOVN</td>
<td>Move Conditional on Not Zero</td>
</tr>
<tr>
<td>MOVT</td>
<td>Move Conditional on Floating Point True</td>
</tr>
<tr>
<td>MOVZ</td>
<td>Move Conditional on Zero</td>
</tr>
<tr>
<td>MTHI</td>
<td>Move To HI Register</td>
</tr>
<tr>
<td>MTLO</td>
<td>Move To LO Register</td>
</tr>
</tbody>
</table>

### Table 3-7 CPU Shift Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>SLL</td>
<td>Shift Word Left Logical</td>
</tr>
<tr>
<td>SLLV</td>
<td>Shift Word Left Logical Variable</td>
</tr>
<tr>
<td>SRA</td>
<td>Shift Word Right Arithmetic</td>
</tr>
<tr>
<td>SRAV</td>
<td>Shift Word Right Arithmetic Variable</td>
</tr>
<tr>
<td>SRL</td>
<td>Shift Word Right Logical</td>
</tr>
<tr>
<td>SRLV</td>
<td>Shift Word Right Logical Variable</td>
</tr>
</tbody>
</table>
### Table 3-8 CPU Trap Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>BREAK</td>
<td>Breakpoint</td>
</tr>
<tr>
<td>SYSCALL</td>
<td>System Call</td>
</tr>
<tr>
<td>TEQ</td>
<td>Trap if Equal</td>
</tr>
<tr>
<td>TEQI</td>
<td>Trap if Equal Immediate</td>
</tr>
<tr>
<td>TGE</td>
<td>Trap if Greater or Equal</td>
</tr>
<tr>
<td>TGEI</td>
<td>Trap if Greater of Equal Immediate</td>
</tr>
<tr>
<td>TGEIU</td>
<td>Trap if Greater or Equal Immediate Unsigned</td>
</tr>
<tr>
<td>TGEU</td>
<td>Trap if Greater or Equal Unsigned</td>
</tr>
<tr>
<td>TLT</td>
<td>Trap if Less Than</td>
</tr>
<tr>
<td>TLTI</td>
<td>Trap if Less Than Immediate</td>
</tr>
<tr>
<td>TLIU</td>
<td>Trap if Less Than Immediate Unsigned</td>
</tr>
<tr>
<td>TLTU</td>
<td>Trap if Less Than Unsigned</td>
</tr>
<tr>
<td>TNE</td>
<td>Trap if Not Equal</td>
</tr>
<tr>
<td>TNEI</td>
<td>Trap if Not Equal Immediate</td>
</tr>
</tbody>
</table>

### Table 3-9 Obsolete CPU Branch Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEQL</td>
<td>Branch on Equal Likely</td>
</tr>
<tr>
<td>BGEZALL</td>
<td>Branch on Greater Than or Equal to Zero and Link Likely</td>
</tr>
<tr>
<td>BGEZL</td>
<td>Branch on Greater Than or Equal to Zero Likely</td>
</tr>
<tr>
<td>BGTZL</td>
<td>Branch on Greater Than Zero Likely</td>
</tr>
<tr>
<td>BLEZL</td>
<td>Branch on Less Than or Equal to Zero Likely</td>
</tr>
<tr>
<td>BLTZALL</td>
<td>Branch on Less Than Zero and Link Likely</td>
</tr>
<tr>
<td>BLTZL</td>
<td>Branch on Less Than Zero Likely</td>
</tr>
<tr>
<td>BNEL</td>
<td>Branch on Not Equal Likely</td>
</tr>
</tbody>
</table>

---

a. Software is strongly encouraged to avoid use of the Branch Likely instructions, as they will be removed from a future revision of the MIPS32 architecture.
### Table 3-10 FPU Arithmetic Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABS.fmt</td>
<td>Floating Point Absolute Value</td>
</tr>
<tr>
<td>ADD.fmt</td>
<td>Floating Point Add</td>
</tr>
<tr>
<td>DIV.fmt</td>
<td>Floating Point Divide</td>
</tr>
<tr>
<td>MADD.fmt</td>
<td>Floating Point Multiply Add</td>
</tr>
<tr>
<td>MSUB.fmt</td>
<td>Floating Point Multiply Subtract</td>
</tr>
<tr>
<td>MUL.fmt</td>
<td>Floating Point Multiply</td>
</tr>
<tr>
<td>NEG.fmt</td>
<td>Floating Point Negate</td>
</tr>
<tr>
<td>NMADD.fmt</td>
<td>Floating Point Negative Multiply Add</td>
</tr>
<tr>
<td>NMSUB.fmt</td>
<td>Floating Point Negative Multiply Subtract</td>
</tr>
<tr>
<td>RECIP.fmt</td>
<td>Reciprocal Approximation</td>
</tr>
<tr>
<td>RSQRT.fmt</td>
<td>Reciprocal Square Root Approximation</td>
</tr>
<tr>
<td>SQRT</td>
<td>Floating Point Square Root</td>
</tr>
<tr>
<td>SUB.fmt</td>
<td>Floating Point Subtract</td>
</tr>
</tbody>
</table>

### Table 3-11 FPU Branch Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>BC1F</td>
<td>Branch on FP False</td>
</tr>
<tr>
<td>BC1T</td>
<td>Branch on FP True</td>
</tr>
</tbody>
</table>

### Table 3-12 FPU Compare Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>C.cond.fmt</td>
<td>Floating Point Compare</td>
</tr>
</tbody>
</table>

### Table 3-13 FPU Convert Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>CEIL.W.fmt</td>
<td>Floating Point Ceiling Convert to Word Fixed Point</td>
</tr>
<tr>
<td>CVT.D.fmt</td>
<td>Floating Point Convert to Double Floating Point</td>
</tr>
<tr>
<td>CVT.S.fmt</td>
<td>Floating Point Convert to Single Floating Point</td>
</tr>
<tr>
<td>CVT.W.fmt</td>
<td>Floating Point Convert to Word Fixed Point</td>
</tr>
<tr>
<td>FLOOR.W.fmt</td>
<td>Floating Point Floor Convert to Word Fixed Point</td>
</tr>
<tr>
<td>ROUND.W.fmt</td>
<td>Floating Point Round to Word Fixed Point</td>
</tr>
<tr>
<td>TRUNC.W.fmt</td>
<td>Floating Point Truncate to Word Fixed Point</td>
</tr>
</tbody>
</table>
### Table 3-14 FPU Load, Store, and Memory Control Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDC1</td>
<td>Load Doubleword to Floating Point</td>
</tr>
<tr>
<td>LWC1</td>
<td>Load Word to Floating Point</td>
</tr>
<tr>
<td>SDC1</td>
<td>Store Doubleword from Floating Point</td>
</tr>
<tr>
<td>SWC1</td>
<td>Store Word from Floating Point</td>
</tr>
</tbody>
</table>

### Table 3-15 FPU Move Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>CFC1</td>
<td>Move Control Word from Floating Point</td>
</tr>
<tr>
<td>CTC1</td>
<td>Move Control Word to Floating Point</td>
</tr>
<tr>
<td>MFC1</td>
<td>Move Word from Floating Point</td>
</tr>
<tr>
<td>MOV.fmt</td>
<td>Floating Point Move</td>
</tr>
<tr>
<td>MOVF.fmt</td>
<td>Floating Point Move Conditional on Floating Point False</td>
</tr>
<tr>
<td>MOVN.fmt</td>
<td>Floating Point Move Conditional on Not Zero</td>
</tr>
<tr>
<td>MOVT.fmt</td>
<td>Floating Point Move Conditional on Floating Point True</td>
</tr>
<tr>
<td>MOVZ.fmt</td>
<td>Floating Point Move Conditional on Zero</td>
</tr>
<tr>
<td>MTC1</td>
<td>Move Word to Floating Point</td>
</tr>
</tbody>
</table>

### Table 3-16 Obsolete\(^a\) FPU Branch Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>BC1FL</td>
<td>Branch on FP False Likely</td>
</tr>
<tr>
<td>BC1TL</td>
<td>Branch on FP True Likely</td>
</tr>
</tbody>
</table>

\(^a\) Software is strongly encouraged to avoid use of the Branch Likely instructions, as they will be removed from a future revision of the MIPS32 architecture.

### Table 3-17 Coprocessor Branch Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>BC2F</td>
<td>Branch on COP2 False</td>
</tr>
<tr>
<td>BC2T</td>
<td>Branch on COP2 True</td>
</tr>
</tbody>
</table>

### Table 3-18 Coprocessor Execute Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>COP2</td>
<td>Coprocessor Operation to Coprocessor 2</td>
</tr>
</tbody>
</table>
### Table 3-19 Coprocessor Load and Store Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDC2</td>
<td>Load Doubleword to Coprocessor 2</td>
</tr>
<tr>
<td>LWC2</td>
<td>Load Word to Coprocessor 2</td>
</tr>
<tr>
<td>SDC2</td>
<td>Store Doubleword from Coprocessor 2</td>
</tr>
<tr>
<td>SWC2</td>
<td>Store Word from Coprocessor 2</td>
</tr>
</tbody>
</table>

### Table 3-20 Coprocessor Move Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>CFC2</td>
<td>Move Control Word from Coprocessor 2</td>
</tr>
<tr>
<td>CTC2</td>
<td>Move Control Word to Coprocessor 2</td>
</tr>
<tr>
<td>MFC2</td>
<td>Move Word from Coprocessor 2</td>
</tr>
<tr>
<td>MTC2</td>
<td>Move Word to Coprocessor 2</td>
</tr>
</tbody>
</table>

### Table 3-21 Obsolete\(^a\) Coprocessor Branch Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>BC2FL</td>
<td>Branch on COP2 False Likely</td>
</tr>
<tr>
<td>BC2TL</td>
<td>Branch on COP2 True Likely</td>
</tr>
</tbody>
</table>

\(^a\) Software is strongly encouraged to avoid use of the Branch Likely instructions, as they will be removed from a future revision of the MIPS32 architecture.

### Table 3-22 Privileged Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>CACHE</td>
<td>Perform Cache Operation</td>
</tr>
<tr>
<td>ERET</td>
<td>Exception Return</td>
</tr>
<tr>
<td>MFC0</td>
<td>Move from Coprocessor 0</td>
</tr>
<tr>
<td>MTC0</td>
<td>Move to Coprocessor 0</td>
</tr>
<tr>
<td>TLBP</td>
<td>Probe TLB for Matching Entry</td>
</tr>
<tr>
<td>TLBR</td>
<td>Read Indexed TLB Entry</td>
</tr>
<tr>
<td>TLBWI</td>
<td>Write Indexed TLB Entry</td>
</tr>
<tr>
<td>TLBWR</td>
<td>Write Random TLB Entry</td>
</tr>
<tr>
<td>WAIT</td>
<td>Enter Standby Mode</td>
</tr>
</tbody>
</table>
### Table 3-23 EJTAG Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>DERET</td>
<td>Debug Exception Return</td>
</tr>
<tr>
<td>SDBBP</td>
<td>Software Debug Breakpoint</td>
</tr>
</tbody>
</table>
**Floating Point Absolute Value**

<table>
<thead>
<tr>
<th>COP1</th>
<th>fmt</th>
<th>0</th>
<th>fs</th>
<th>fd</th>
<th>ABS</th>
</tr>
</thead>
<tbody>
<tr>
<td>010001</td>
<td>00000</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

**Format:**
- ABS.S fd, fs
- ABS.D fd, fs

**MIPS32 (MIPS I)**

**Purpose:**
To compute the absolute value of an FP value

**Description:**
\[ fd \leftarrow \text{abs}(fs) \]

The absolute value of the value in FPR \( fs \) is placed in FPR \( fd \). The operand and result are values in format \( fmt \). Cause bits are ORed into the Flag bits if no exception is taken.

This operation is arithmetic; a NaN operand signals invalid operation.

**Restrictions:**
The fields \( fs \) and \( fd \) must specify FPRs valid for operands of type \( fmt \). If they are not valid, the result is UNPREDICTABLE.

The operand must be a value in format \( fmt \); if it is not, the result is UNPREDICTABLE and the value of the operand FPR becomes UNPREDICTABLE.

**Operation:**
- \( \text{StoreFPR}(fd, fmt, \text{AbsoluteValue}(\text{ValueFPR}(fs, fmt))) \)

**Exceptions:**
- Coprocessor Unusable, Reserved Instruction

**Floating Point Exceptions:**
- Unimplemented Operation, Invalid Operation
**Add Word**

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>0</td>
<td>00000</td>
<td>ADD</td>
<td>100000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000000</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** ADD rd, rs, rt

**MIPS32 (MIPS I)**

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

**Description:** rd $\leftarrow$ rs + 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 $rd$.

**Restrictions:**
None

**Operation:**

```
temp $\leftarrow$ (GPR[rs]31 || GPR[rs]31..0) + (GPR[rt]31 || GPR[rt]31..0)
if temp32 $\neq$ temp31 then
    SignalException(IntegerOverflow)
else
    GPR[rd] $\leftarrow$ temp
endif
```

**Exceptions:**

Integer Overflow

**Programming Notes:**

ADDU performs the same arithmetic operation but does not trap on overflow.
Floating Point Add

### ADD.fmt

#### Format:

- `ADD.S fd, fs, ft`
- `ADD.D fd, fs, ft`

#### MIPS32 (MIPS I)

#### Purpose:

To add floating point values

#### Description:

**fd** ← **fs** + **ft**

The value in FPR **ft** is added to the value in FPR **fs**. The result is calculated to infinite precision, rounded by using to the current rounding mode in **FCSR**, and placed into FPR **fd**. The operands and result are values in format **fmt**. Cause bits are ORed into the Flag bits if no exception is taken.

#### Restrictions:

The fields **fs**, **ft**, and **fd** must specify FPRs valid for operands of type **fmt**. If they are not valid, the result is **UNPREDICTABLE**.

The operands must be values in format **fmt**; if they are not, the result is **UNPREDICTABLE** and the value of the operand FPRs becomes **UNPREDICTABLE**.

#### Operation:

\[
\text{StoreFPR} (fd, fmt, \text{ValueFPR}(fs, fmt) +_{\text{fmt}} \text{ValueFPR}(ft, fmt))
\]

#### Exceptions:

Coprocessor Unusable, Reserved Instruction

#### Floating Point Exceptions:

Unimplemented Operation, Invalid Operation, Inexact, Overflow, Underflow
Add Immediate Word

<table>
<thead>
<tr>
<th>ADDI</th>
<th>001000</th>
<th>rs</th>
<th>rt</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>26 25</td>
<td>21 20</td>
<td>16 15</td>
<td>16</td>
</tr>
</tbody>
</table>

**Format:** ADDI rt, rs, immediate

**Purpose:**
To add a constant to a 32-bit integer. If overflow occurs, then trap.

**Description:** rt ← rs + immediate

The 16-bit signed immediate 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 rt.

**Restrictions:**
None

**Operation:**

\[
\text{temp} \leftarrow (\text{GPR}[rs]_{31} | \text{GPR}[rs]_{31..0}) + \text{sign\_extend}(\text{immediate})
\]

if temp

\[
\text{temp} \neq \text{temp}_{31}
\]

then

SignalException(IntegerOverflow)

else

GPR[rt] ← temp

endif

**Exceptions:**
Integer Overflow

**Programming Notes:**
ADDIU performs the same arithmetic operation but does not trap on overflow.
Add Immediate Unsigned Word

<table>
<thead>
<tr>
<th>Format: ADDIU rt, rs, immediate</th>
<th>MIPS32 (MIPS I)</th>
</tr>
</thead>
</table>

**Purpose:**
To add a constant to a 32-bit integer

**Description:** rt ← rs + immediate

The 16-bit signed immediate is added to the 32-bit value in GPR rs and the 32-bit arithmetic result is placed into GPR rt.

No Integer Overflow exception occurs under any circumstances.

**Restrictions:**
None

**Operation:**

\[
\text{temp} \leftarrow \text{GPR}[rs] + \text{sign} \_\text{extend}(\text{immediate})
\]

\[
\text{GPR}[rt] \leftarrow \text{temp}
\]

**Exceptions:**
None

**Programming Notes:**
The term "unsigned" in the instruction name is a misnomer; this operation is 32-bit modulo arithmetic that does not trap on overflow. This instruction is appropriate for unsigned arithmetic, such as address arithmetic, or integer arithmetic environments that ignore overflow, such as C language arithmetic.
Add Unsigned Word

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>0</td>
<td>ADDU</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000000</td>
<td></td>
<td></td>
<td></td>
<td>00000</td>
<td>100001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Format: ADDU rd, rs, rt

MIPS32 (MIPS I)

Purpose:
To add 32-bit integers

Description: rd ← rs + rt
The 32-bit word value in GPR rt is added to the 32-bit value in GPR rs and the 32-bit arithmetic result is placed into GPR rd.

No Integer Overflow exception occurs under any circumstances.

Restrictions:
None

Operation:
  temp ← GPR[rs] + GPR[rt]
  GPR[rd] ← temp

Exceptions:
None

Programming Notes:
The term "unsigned" in the instruction name is a misnomer; this operation is 32-bit modulo arithmetic that does not trap on overflow. This instruction is appropriate for unsigned arithmetic, such as address arithmetic, or integer arithmetic environments that ignore overflow, such as C language arithmetic.
**AND**

<table>
<thead>
<tr>
<th>Format: AND rd, rs, rt</th>
<th>MIPS32 (MIPS I)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose: To do a bitwise logical AND</td>
<td></td>
</tr>
<tr>
<td>Description: rd ← rs AND rt</td>
<td></td>
</tr>
<tr>
<td>The contents of GPR rs are combined with the contents of GPR rt in a bitwise logical AND operation. The result is placed into GPR rd.</td>
<td></td>
</tr>
<tr>
<td>Restrictions: None</td>
<td></td>
</tr>
<tr>
<td>Operation: GPR[rd] ← GPR[rs] and GPR[rt]</td>
<td></td>
</tr>
<tr>
<td>Exceptions: None</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL 000000</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>0</td>
<td>0000</td>
<td>AND 10100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
And Immediate

**Format:** \( \text{ANDI } rt, rs, \text{ immediate} \)

**MIPS32 (MIPS I)**

**Purpose:**
To do a bitwise logical AND with a constant

**Description:** \( rt \leftarrow rs \text{ AND immediate} \)

The 16-bit immediate is zero-extended to the left and combined with the contents of GPR \( rs \) in a bitwise logical AND operation. The result is placed into GPR \( rt \).

**Restrictions:**
None

**Operation:**
\[
GPR[rt] \leftarrow GPR[rs] \text{ and zero_extend(immediate)}
\]

**Exceptions:**
None
Unconditional Branch

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEQ</td>
<td>000100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>offset</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** $B \text{ offset}$

**Purpose:**
To do an unconditional branch

**Description:** $\text{branch}$

$B$ offset is the assembly idiom used to denote an unconditional branch. The actual instruction is interpreted by the hardware as BEQ r0, r0, offset.

An 18-bit signed offset (the 16-bit $\text{offset}$ field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

**Restrictions:**
Processor operation is **UNPREDICTABLE** if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

**Operation:**

$I$: $\text{target_offset} \leftarrow \text{sign_extend}(\text{offset} || 0^2)$

$I+1$: $\text{PC} \leftarrow \text{PC} + \text{target_offset}$

**Exceptions:**
None

**Programming Notes:**

With the 18-bit signed instruction offset, the conditional branch range is $\pm 128$ Kbytes. Use jump (J) or jump register (JR) instructions to branch to addresses outside this range.
Branch and Link BAL

| 31  | 26  | 25  | 21  | 20  | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|-----|-----|-----|-----|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| REGIMM | 0 | 0000 | BGEZAL | 1000 | offset |
| 6 | 5 | 5 | 16 |

Format: BAL rs, offset

Purpose:
To do an unconditional PC-relative procedure call

Description: procedure_call

BAL offset is the assembly idiom used to denote an unconditional branch. The actual instruction is interpreted by the hardware as BGEZAL r0, offset.

Place the return address link in GPR 31. The return link is the address of the second instruction following the branch, where execution continues after a procedure call.

An 18-bit signed offset (the 16-bit offset field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

Restrictions:
Processor operation is UNPREDICTABLE if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

GPR 31 must not be used for the source register rs, because such an instruction does not have the same effect when reexecuted. The result of executing such an instruction is UNPREDICTABLE. This restriction permits an exception handler to resume execution by reexecuting the branch when an exception occurs in the branch delay slot.

Operation:

I: target_offset ← sign_extend(offset || 0²)
GPR[31] ← PC + 8
I+1: PC ← PC + target_offset

Exceptions:
None

Programming Notes:
With the 18-bit signed instruction offset, the conditional branch range is ± 128 KBytes. Use jump and link (JAL) or jump and link register (JALR) instructions for procedure calls to addresses outside this range.
Branch on FP False

**Format:**

<table>
<thead>
<tr>
<th>Format</th>
<th>MIPS32 (MIPS I)</th>
<th>MIPS32 (MIPS IV)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BC1F</td>
<td>offset (cc = 0 implied)</td>
<td>cc, offset</td>
</tr>
</tbody>
</table>

**Purpose:**

To test an FP condition code and do a PC-relative conditional branch

**Description:**

if cc = 0 then branch

An 18-bit signed offset (the 16-bit offset field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself) in the branch delay slot to form a PC-relative effective target address. If the FP condition code bit CC is false (0), the program branches to the effective target address after the instruction in the delay slot is executed. An FP condition code is set by the FP compare instruction, C.cond.fmt.

**Restrictions:**

Processor operation is **UNPREDICTABLE** if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

**Operation:**

This operation specification is for the general Branch On Condition operation with the tf (true/false) and nd (nullify delay slot) fields as variables. The individual instructions BC1F, BC1FL, BC1T, and BC1TL have specific values for tf and nd.

```
I:    condition ← FPConditionCode(cc) = 0
      target_offset ← (offset15)GPRLEN-(16+2) | | offset | | 0^2
I+1:  if condition then
      PC ← PC + target_offset
      endif
```
Exceptions:
Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:
Unimplemented Operation

Programming Notes:
With the 18-bit signed instruction offset, the conditional branch range is ±128 KBytes. Use jump (J) or jump register (JR) instructions to branch to addresses outside this range.

Historical Information:
The MIPS I architecture defines a single floating point condition code, implemented as the coprocessor 1 condition signal (Cp1Cond) and the C bit in the FP Control/Status register. MIPS I, II, and III architectures must have the CC field set to 0, which is implied by the first format in the “Format” section.
The MIPS IV and MIPS32 architectures add seven more Condition Code bits to the original condition code 0. FP compare and conditional branch instructions specify the Condition Code bit to set or test. Both assembler formats are valid for MIPS IV and MIPS32.
In the MIPS I, II, and III architectures there must be at least one instruction between the compare instruction that sets the condition code and the branch instruction that tests it. Hardware does not detect a violation of this restriction.
### Branch on FP False Likely

**Format:**

- BC1FL   offset (cc = 0 implied)  \(\text{MIPS32 (MIPS II)}\)
- BC1FL   cc, offset             \(\text{MIPS32 (MIPS IV)}\)

**Purpose:**

To test an FP condition code and make a PC-relative conditional branch; execute the instruction in the delay slot only if the branch is taken.

**Description:** \(\text{if } cc = 0 \text{ then branch\_likely}\)

An 18-bit signed offset (the 16-bit offset field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself) in the branch delay slot to form a PC-relative effective target address. If the FP Condition Code bit CC is false (0), the program branches to the effective target address after the instruction in the delay slot is executed. If the branch is not taken, the instruction in the delay slot is not executed.

An FP condition code is set by the FP compare instruction, C.cond.fmt.

**Restrictions:**

Processor operation is **UNPREDICTABLE** if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

**Operation:**

This operation specification is for the general Branch On Condition operation with the \(tf\) (true/false) and \(nd\) (nullify delay slot) fields as variables. The individual instructions BC1F, BC1FL, BC1T, and BC1TL have specific values for \(tf\) and \(nd\).

\[
\begin{align*}
I & : & \text{condition} & \leftarrow \text{FPConditionCode}(cc) = 0 \\
& & \text{target\_offset} & \leftarrow (\text{offset}_{15})^{GPRLEN-(16+2)} \ || \ offset \ || \ 0^{2} \\
I+1 & : & \text{if condition then} \\
& & \text{PC} & \leftarrow \text{PC} + \text{target\_offset} \\
& & \text{else} \\
& & \text{NullifyCurrentInstruction()} \\
& & \text{endif}
\end{align*}
\]
Branch on FP False Likely (cont.)

**Exceptions:**
Coprocessor Unusable, Reserved Instruction

**Floating Point Exceptions:**
Unimplemented Operation

**Programming Notes:**
With the 18-bit signed instruction offset, the conditional branch range is ± 128 KBytes. Use jump (J) or jump register (JR) instructions to branch to addresses outside this range.

Software is strongly encouraged to avoid the use of the Branch Likely instructions, as they will be removed from a future revision of the MIPS Architecture.

Some implementations always predict the branch will be taken, so there is a significant penalty if the branch is not taken. Software should only use this instruction when there is a very high probability (98% or more) that the branch will be taken. If the branch is not likely to be taken or if the probability of a taken branch is unknown, software is encouraged to use the BC1F instruction instead.

**Historical Information:**
The MIPS I architecture defines a single floating point condition code, implemented as the coprocessor 1 condition signal (Cp1Cond) and the C bit in the FP Control/Status register. MIPS I, II, and III architectures must have the CC field set to 0, which is implied by the first format in the “Format” section.

The MIPS IV and MIPS32 architectures add seven more Condition Code bits to the original condition code 0. FP compare and conditional branch instructions specify the Condition Code bit to set or test. Both assembler formats are valid for MIPS IV and MIPS32.

In the MIPS II and III architectures there must be at least one instruction between the compare instruction that sets a condition code and the branch instruction that tests it. Hardware does not detect a violation of this restriction.
## Branch on FP True

### Format:

\[ \text{BC1T offset (cc = 0 implied)} \]

\[ \text{BC1T cc, offset} \]

**MIPS32 (MIPS I)**

**MIPS32 (MIPS IV)**

### Purpose:

To test an FP condition code and do a PC-relative conditional branch

### Description:

if cc = 1 then branch

An 18-bit signed offset (the 16-bit offset field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself) in the branch delay slot to form a PC-relative effective target address. If the FP condition code bit \( CC \) is true (1), the program branches to the effective target address after the instruction in the delay slot is executed. An FP condition code is set by the FP compare instruction, \( \text{C.cond.fmt} \).

### Restrictions:

Processor operation is \text{UNPREDICTABLE} if a branch, jump, ERET, DERET, or \text{WAIT} instruction is placed in the delay slot of a branch or jump.

### Operation:

This operation specification is for the general Branch On Condition operation with the \( tf \) (true/false) and \( nd \) (nullify delay slot) fields as variables. The individual instructions BC1F, BC1FL, BC1T, and BC1TL have specific values for \( tf \) and \( nd \).

\[
\begin{align*}
\text{I:} & \quad \text{condition} \leftarrow \text{FPConditionCode}(cc) = 1 \\
& \quad \text{target\_offset} \leftarrow (\text{offset}_{15})^{\text{GPRLEN}-16+2} \ | \ | \text{offset} \ | \ 0^2 \\
\text{I+1:} & \quad \text{if condition then} \\
& \quad \quad \text{PC} \leftarrow \text{PC} + \text{target\_offset} \\
& \quad \quad \text{endif}
\end{align*}
\]
Exceptions:
Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:
Unimplemented Operation

Programming Notes:
With the 18-bit signed instruction offset, the conditional branch range is ±128 KBytes. Use jump (J) or jump register (JR) instructions to branch to addresses outside this range.

Historical Information:
The MIPS I architecture defines a single floating point condition code, implemented as the coprocessor 1 condition signal (Cp1Cond) and the C bit in the FP Control/Status register. MIPS I, II, and III architectures must have the CC field set to 0, which is implied by the first format in the “Format” section.

The MIPS IV and MIPS32 architectures add seven more Condition Code bits to the original condition code 0. FP compare and conditional branch instructions specify the Condition Code bit to set or test. Both assembler formats are valid for MIPS IV and MIPS32.

In the MIPS I, II, and III architectures there must be at least one instruction between the compare instruction that sets the condition code and the branch instruction that tests it. Hardware does not detect a violation of this restriction.
Branch on FP True Likely

Format: BC1TL offset (cc = 0 implied)  
BC1TL cc, offset

MIPS32 (MIPS II)  
MIPS32 (MIPS IV)

Purpose:
To test an FP condition code and do a PC-relative conditional branch; execute the instruction in the delay slot only if the branch is taken.

Description: if cc = 1 then branch_likely
An 18-bit signed offset (the 16-bit offset field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself) in the branch delay slot to form a PC-relative effective target address. If the FP Condition Code bit CC is true (1), the program branches to the effective target address after the instruction in the delay slot is executed. If the branch is not taken, the instruction in the delay slot is not executed.

An FP condition code is set by the FP compare instruction, C.cond.fmt.

Restrictions:
Processor operation is UNPREDICTABLE if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

Operation:
This operation specification is for the general Branch On Condition operation with the tf (true/false) and nd (nullify delay slot) fields as variables. The individual instructions BC1F, BC1FL, BC1T, and BC1TL have specific values for tf and nd.

\[ \text{I: } \text{condition} \leftarrow \text{FPConditionCode}(cc) = 1 \]
\[ \text{target_offset} \leftarrow (\text{offset}_{16})^{GPRLEN-(16+2)} \ || \ \text{offset} \ || \ 0^2 \]
\[ \text{I+1: if condition then} \]
\[ \text{PC} \leftarrow \text{PC} + \text{target_offset} \]
\[ \text{else} \]
\[ \text{NullifyCurrentInstruction()} \]
\[ \text{endif} \]
Exceptions:
Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:
Unimplemented Operation

Programming Notes:
With the 18-bit signed instruction offset, the conditional branch range is ± 128 KBytes. Use jump (J) or jump register (JR) instructions to branch to addresses outside this range.

Software is strongly encouraged to avoid the use of the Branch Likely instructions, as they will be removed from a future revision of the MIPS Architecture.

Some implementations always predict the branch will be taken, so there is a significant penalty if the branch is not taken. Software should only use this instruction when there is a very high probability (98% or more) that the branch will be taken. If the branch is not likely to be taken or if the probability of a taken branch is unknown, software is encouraged to use the BC1T instruction instead.

Historical Information:
The MIPS I architecture defines a single floating point condition code, implemented as the coprocessor 1 condition signal (Cp1Cond) and the C bit in the FP Control/Status register. MIPS I, II, and III architectures must have the CC field set to 0, which is implied by the first format in the “Format” section.

The MIPS IV and MIPS32 architectures add seven more Condition Code bits to the original condition code 0. FP compare and conditional branch instructions specify the Condition Code bit to set or test. Both assembler formats are valid for MIPS IV and MIPS32.

In the MIPS II and III architectures there must be at least one instruction between the compare instruction that sets a condition code and the branch instruction that tests it. Hardware does not detect a violation of this restriction.
### Branch on COP2 False

**Format:**  
BC2F   offset (cc = 0 implied)  
BC2F   cc, offset  

**MIPS32 (MIPS I)**  
**MIPS32 (MIPS IV)**

**Purpose:**
To test a COP2 condition code and do a PC-relative conditional branch

**Description:**  
if cc = 0 then branch

An 18-bit signed offset (the 16-bit offset field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself) in the branch delay slot to form a PC-relative effective target address. If the COP2 condition specified by CC is false (0), the program branches to the effective target address after the instruction in the delay slot is executed.

**Restrictions:**
Processor operation is **UNPREDICTABLE** if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

**Operation:**
This operation specification is for the general Branch On Condition operation with the tf (true/false) and nd (nullify delay slot) fields as variables. The individual instructions BC2F, BC2FL, BC2T, and BC2TL have specific values for tf and nd.

I:  
condition ← COP2Condition(cc) = 0  
target_offset ← (offset_{15})^{\text{GPRLEN}=-(16+2)} || offset || 0^2  

I+1:  
if condition then  
PC ← PC + target_offset  
endif

**Exceptions:**
Coprocessor Unusable, Reserved Instruction

**Programming Notes:**
With the 18-bit signed instruction offset, the conditional branch range is ± 128 KBytes. Use jump (J) or jump register (JR) instructions to branch to addresses outside this range.
Branch on COP2 False Likely

**Format:**
- BC2FL offset (cc = 0 implied)
- BC2FL cc, offset

**MIPS32 (MIPS II)**
- MIPS32 (MIPS IV)

**Purpose:**
To test a COP2 condition code and make a PC-relative conditional branch; execute the instruction in the delay slot only if the branch is taken.

**Description:** if cc = 0 then branch\_likely

An 18-bit signed offset (the 16-bit offset field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself) in the branch delay slot to form a PC-relative effective target address. If the COP2 condition specified by \( CC \) is false (0), the program branches to the effective target address after the instruction in the delay slot is executed. If the branch is not taken, the instruction in the delay slot is not executed.

**Restrictions:**
Processor operation is **UNPREDICTABLE** if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

**Operation:**
This operation specification is for the general Branch On Condition operation with the \( tf \) (true/false) and \( nd \) (nullify delay slot) fields as variables. The individual instructions BC2F, BC2FL, BC2T, and BC2TL have specific values for \( tf \) and \( nd \).

\[
\begin{align*}
I: & \quad \text{condition} \leftarrow \text{COP2Condition}(cc) = 0 \\
& \quad \text{target\_offset} \leftarrow (\text{offset}_{15})_{GPRLEN-(16+2)} \mid \mid \text{offset} \mid \mid 0^2 \\
I+1: & \quad \text{if condition then} \\
& \quad \quad \text{PC} \leftarrow \text{PC} + \text{target\_offset} \\
& \quad \text{else} \\
& \quad \quad \text{NullifyCurrentInstruction()} \\
& \quad \text{endif}
\end{align*}
\]
Branch on COP2 False Likely (cont.)

**Exceptions:**
- Coprocessor Unusable, Reserved Instruction

**Programming Notes:**
- With the 18-bit signed instruction offset, the conditional branch range is ± 128 KBytes. Use jump (J) or jump register (JR) instructions to branch to addresses outside this range.
- Software is strongly encouraged to avoid the use of the Branch Likely instructions, as they will be removed from a future revision of the MIPS Architecture.
- Some implementations always predict the branch will be taken, so there is a significant penalty if the branch is not taken. Software should only use this instruction when there is a very high probability (98% or more) that the branch will be taken. If the branch is not likely to be taken or if the probability of a taken branch is unknown, software is encouraged to use the BC2F instruction instead.
Branch on COP2 True

<table>
<thead>
<tr>
<th>COP2</th>
<th>BC</th>
<th>cc</th>
<th>nd</th>
<th>tf</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>010010</td>
<td>01000</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>16</td>
</tr>
</tbody>
</table>

**Format:** BC2T offset (cc = 0 implied)  
BC2T cc, offset

**Purpose:**  
To test a COP2 condition code and do a PC-relative conditional branch

**Description:** if cc = 1 then branch  
An 18-bit signed offset (the 16-bit offset field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself) in the branch delay slot to form a PC-relative effective target address. If the COP2 condition specified by CC is true (1), the program branches to the effective target address after the instruction in the delay slot is executed.

**Restrictions:**  
Processor operation is UNPREDICTABLE if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

**Operation:**  
This operation specification is for the general Branch On Condition operation with the tf (true/false) and nd (nullify delay slot) fields as variables. The individual instructions BC2F, BC2FL, BC2T, and BC2TL have specific values for tf and nd.

I:  
condition ← COP2Condition(cc) = 1  
target_offset ← (offset_{15})^{GPRLEN-(16+2)} || offset || 0^2

I+1:  
if condition then  
PC ← PC + target_offset  
endif

**Exceptions:**  
Coprocessor Unusable, Reserved Instruction

**Programming Notes:**  
With the 18-bit signed instruction offset, the conditional branch range is ± 128 KBytes. Use jump (J) or jump register (JR) instructions to branch to addresses outside this range.
Branch on COP2 True Likely

**Format:**

<table>
<thead>
<tr>
<th>Format:</th>
<th>BC2TL   offset (cc = 0 implied)</th>
<th>MIPS32 (MIPS II)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>BC2TL cc, offset</td>
<td>MIPS32 (MIPS IV)</td>
</tr>
</tbody>
</table>

**Purpose:**

To test a COP2 condition code and do a PC-relative conditional branch; execute the instruction in the delay slot only if the branch is taken.

**Description:**

if cc = 1 then branch_likely

An 18-bit signed offset (the 16-bit offset field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself) in the branch delay slot to form a PC-relative effective target address. If the COP2 condition specified by CC is true (1), the program branches to the effective target address after the instruction in the delay slot is executed. If the branch is not taken, the instruction in the delay slot is not executed.

**Restrictions:**

Processor operation is **UNPREDICTABLE** if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

**Operation:**

This operation specification is for the general Branch On Condition operation with the tf (true/false) and nd (nullify delay slot) fields as variables. The individual instructions BC2F, BC2FL, BC2T, and BC2TL have specific values for tf and nd.

\[
\begin{align*}
I: & \quad \text{condition} \leftarrow \text{COP2Condition}(cc) = 1 \\
& \quad \text{target_offset} \leftarrow (\text{offset}_{15})_{\text{GPRLEN}-(16+2)} | | \text{offset} | | 0^2 \\
I+1: & \quad \text{if condition then} \\
& \quad \text{PC} \leftarrow \text{PC} + \text{target_offset} \\
& \quad \text{else} \\
& \quad \text{NullifyCurrentInstruction()} \\
& \quad \text{endif}
\end{align*}
\]
Branch on COP2 True Likely (cont.)

**Exceptions:**
Coprocessor Unusable, Reserved Instruction

**Programming Notes:**
With the 18-bit signed instruction offset, the conditional branch range is ± 128 KBytes. Use jump (J) or jump register (JR) instructions to branch to addresses outside this range.

Software is strongly encouraged to avoid the use of the Branch Likely instructions, as they will be removed from a future revision of the MIPS Architecture.

Some implementations always predict the branch will be taken, so there is a significant penalty if the branch is not taken. Software should only use this instruction when there is a very high probability (98% or more) that the branch will be taken. If the branch is not likely to be taken or if the probability of a taken branch is unknown, software is encouraged to use the BC2T instruction instead.
Branch on Equal

BEQ

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEQ</td>
<td>000100</td>
<td>rs</td>
<td>rt</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** BEQ rs, rt, offset

**Purpose:**
To compare GPRs then do a PC-relative conditional branch

**Description:** if rs = rt then branch

An 18-bit signed offset (the 16-bit offset field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR rs and GPR rt are equal, branch to the effective target address after the instruction in the delay slot is executed.

**Restrictions:**
Processor operation is **UNPREDICTABLE** if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

**Operation:**

I: 
\[ \text{target} \_\text{offset} \leftarrow \text{sign} \_\text{extend}(\text{offset} \mid | 0^2) \]
\[ \text{condition} \leftarrow (\text{GPR}[rs] = \text{GPR}[rt]) \]

I+1: 
\[ \text{if condition then} \]
\[ \quad \text{PC} \leftarrow \text{PC} + \text{target} \_\text{offset} \]
\[ \quad \text{endif} \]

**Exceptions:**
None

**Programming Notes:**
With the 18-bit signed instruction offset, the conditional branch range is ±128 Kbytes. Use jump (J) or jump register (JR) instructions to branch to addresses outside this range.

BEQ r0, r0 offset, expressed as B offset, is the assembly idiom used to denote an unconditional branch.
Branch on Equal Likely

**Format:** BEQL rs, rt, offset

**MIPS2 (MIPS II)**

**Purpose:**
To compare GPRs then do a PC-relative conditional branch; execute the delay slot only if the branch is taken.

**Description:** If rs = rt then branch likely

An 18-bit signed offset (the 16-bit offset field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR rs and GPR rt are equal, branch to the target address after the instruction in the delay slot is executed. If the branch is not taken, the instruction in the delay slot is not executed.

**Restrictions:**
Processor operation is UNPREDICTABLE if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

**Operation:**

| I: | target_offset ← sign_extend(offset || 02) |
|    | condition ← (GPR[rs] = GPR[rt]) |
| I+1: | if condition then |
|      | PC ← PC + target_offset |
|      | else |
|      | NullifyCurrentInstruction() |
|      | endif |

**Exceptions:**
None
Programming Notes:

With the 18-bit signed instruction offset, the conditional branch range is ± 128 KBytes. Use jump (J) or jump register (JR) instructions to branch to addresses outside this range.

Software is strongly encouraged to avoid the use of the Branch Likely instructions, as they will be removed from a future revision of the MIPS Architecture.

Some implementations always predict the branch will be taken, so there is a significant penalty if the branch is not taken. Software should only use this instruction when there is a very high probability (98% or more) that the branch will be taken. If the branch is not likely to be taken or if the probability of a taken branch is unknown, software is encouraged to use the BEQ instruction instead.

Historical Information:

In the MIPS I architecture, this instruction signaled a Reserved Instruction Exception.
## Branch on Greater Than or Equal to Zero

### BGEZ

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>REGIMM</td>
<td>rs</td>
<td>BGEZ</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000001</td>
<td>00001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** BGEZ rs, offset

**MIPS32 (MIPS I)**

**Purpose:**
To test a GPR then do a PC-relative conditional branch

**Description:** if rs ≥ 0 then branch

An 18-bit signed offset (the 16-bit offset field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR rs are greater than or equal to zero (sign bit is 0), branch to the effective target address after the instruction in the delay slot is executed.

**Restrictions:**
Processor operation is **UNPREDICTABLE** if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

**Operation:**

I: target_offset ← sign_extend(offset || 0²)
condition ← GPR[rs] ≥ 0

I+1: if condition then
PC ← PC + target_offset
endif

**Exceptions:**
None

**Programming Notes:**
With the 18-bit signed instruction offset, the conditional branch range is ±128 KBytes. Use jump (J) or jump register (JR) instructions to branch to addresses outside this range.
Branch on Greater Than or Equal to Zero and Link

**Format:** \texttt{BGEZAL} \texttt{rs, offset}

**Purpose:**
To test a GPR then do a PC-relative conditional procedure call

**Description:** if \( rs \geq 0 \) then \texttt{procedure_call}

Place the return address link in GPR 31. The return link is the address of the second instruction following the branch, where execution continues after a procedure call.

An 18-bit signed offset (the 16-bit \texttt{offset} field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR \( rs \) are greater than or equal to zero (sign bit is 0), branch to the effective target address after the instruction in the delay slot is executed.

**Restrictions:**
Processor operation is \texttt{UNPREDICTABLE} if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

GPR 31 must not be used for the source register \( rs \), because such an instruction does not have the same effect when reexecuted. The result of executing such an instruction is \texttt{UNPREDICTABLE}. This restriction permits an exception handler to resume execution by reexecuting the branch when an exception occurs in the branch delay slot.

**Operation:**

\[
\begin{align*}
\text{I:} & & \text{target offset} & \leftarrow \text{sign extend}(\text{offset} \mid | 0^2) \\
& & \text{condition} & \leftarrow \text{GPR}[rs] \geq 0^{\text{GPRLEN}} \\
& & \text{GPR}[31] & \leftarrow \text{PC} + 8 \\
\text{I+1:} & & \text{if condition then} \\
& & \text{PC} & \leftarrow \text{PC} + \text{target offset} \\
& & \text{endif}
\end{align*}
\]

**Exceptions:**
None

**Programming Notes:**
With the 18-bit signed instruction offset, the conditional branch range is \( \pm 128 \) KBytes. Use jump and link (JAL) or jump and link register (JALR) instructions for procedure calls to addresses outside this range.

\texttt{BGEZAL} \( r0, \text{offset} \), expressed as \texttt{BAL} \( \text{offset} \), is the assembly idiom used to denote a PC-relative branch and link. \texttt{BAL} is used in a manner similar to \texttt{JAL}, but provides PC-relative addressing and a more limited target PC range.
### Branch on Greater Than or Equal to Zero and Link Likely

**BGEZALL**

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>000001</td>
<td>rs</td>
<td>BGEZALL</td>
<td>10011</td>
<td></td>
<td>offset</td>
<td></td>
</tr>
</tbody>
</table>

**Format:** BGEZALL rs, offset

**MIPS32 (MIPS II)**

**Purpose:**
To test a GPR then do a PC-relative conditional procedure call; execute the delay slot only if the branch is taken.

**Description:** if rs ≥ 0 then procedure_call_likely

Place the return address link in GPR 31. The return link is the address of the second instruction following the branch, where execution continues after a procedure call.

An 18-bit signed offset (the 16-bit offset field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR rs are greater than or equal to zero (sign bit is 0), branch to the effective target address after the instruction in the delay slot is executed. If the branch is not taken, the instruction in the delay slot is not executed.

**Restrictions:**
GPR 31 must not be used for the source register rs, because such an instruction does not have the same effect when reexecuted. The result of executing such an instruction is UNPREDICTABLE. This restriction permits an exception handler to resume execution by reexecuting the branch when an exception occurs in the branch delay slot.

Processor operation is UNPREDICTABLE if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

**Operation:**

I:

```
target_offset ← sign_extend(offset || 0^2)
condition ← GPR[rs] ≥ 0^GPRLEN
GPR[31] ← PC + 8
```

I+1:

```
if condition then
    PC ← PC + target_offset
else
    NullifyCurrentInstruction()
endif
```

**Exceptions:**
None
Programming Notes:

With the 18-bit signed instruction offset, the conditional branch range is ± 128 KBytes. Use jump and link (JAL) or jump and link register (JALR) instructions for procedure calls to addresses outside this range.

Software is strongly encouraged to avoid the use of the Branch Likely instructions, as they will be removed from a future revision of the MIPS Architecture.

Some implementations always predict the branch will be taken, so there is a significant penalty if the branch is not taken. Software should only use this instruction when there is a very high probability (98% or more) that the branch will be taken. If the branch is not likely to be taken or if the probability of a taken branch is unknown, software is encouraged to use the BGEZAL instruction instead.

Historical Information:

In the MIPS I architecture, this instruction signaled a Reserved Instruction Exception.
Branch on Greater Than or Equal to Zero Likely

Format: \texttt{BGEZL \textit{rs}, offset}  

MIPS32 (MIPS II)

Purpose:
To test a GPR then do a PC-relative conditional branch; execute the delay slot only if the branch is taken.

Description: \texttt{if \textit{rs} \geq 0 then branch\_likely}
An 18-bit signed offset (the 16-bit \textit{offset} field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.
If the contents of GPR \textit{rs} are greater than or equal to zero (sign bit is 0), branch to the effective target address after the instruction in the delay slot is executed. If the branch is not taken, the instruction in the delay slot is not executed.

Restrictions:
Processor operation is \texttt{UNPREDICTABLE} if a branch, jump, ERET, DERET, or \texttt{WAIT} instruction is placed in the delay slot of a branch or jump.

Operation:
\begin{verbatim}
I:   target_offset ← sign_extend(offset || 0^2)
     condition ← GPR[rs] \geq 0^\texttt{GPRLEN}
I+1: if condition then
     PC ← PC + target_offset
     else
     NullifyCurrentInstruction()
endif
\end{verbatim}

Exceptions:
None
Programming Notes:

With the 18-bit signed instruction offset, the conditional branch range is ±128 KBytes. Use jump (J) or jump register (JR) instructions to branch to addresses outside this range.

Software is strongly encouraged to avoid the use of the Branch Likely instructions, as they will be removed from a future revision of the MIPS Architecture.

Some implementations always predict the branch will be taken, so there is a significant penalty if the branch is not taken. Software should only use this instruction when there is a very high probability (98% or more) that the branch will be taken. If the branch is not likely to be taken or if the probability of a taken branch is unknown, software is encouraged to use the BGEZ instruction instead.

Historical Information:

In the MIPS I architecture, this instruction signaled a Reserved Instruction Exception.
Branch on Greater Than Zero

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>BGTZ</td>
<td>rs</td>
<td>0</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000111</td>
<td>0</td>
<td>0000</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** BGTZ rs, offset

**MIPS32 (MIPS I)**

**Purpose:**
To test a GPR then do a PC-relative conditional branch

**Description:** if rs > 0 then branch

An 18-bit signed offset (the 16-bit offset field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR rs are greater than zero (sign bit is 0 but value not zero), branch to the effective target address after the instruction in the delay slot is executed.

**Restrictions:**
Processor operation is **UNPREDICTABLE** if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

**Operation:**

| I   | target_offset ← sign_extend(offset || 0²) |
|-----|------------------------------------------|
|     | condition ← GPR[rs] > 0GPRLEN            |
| I+1 | if condition then                        |
|     |   PC ← PC + target_offset               |
|     |   endif                                  |

**Exceptions:**
None

**Programming Notes:**
With the 18-bit signed instruction offset, the conditional branch range is ± 128 KBytes. Use jump (J) or jump register (JR) instructions to branch to addresses outside this range.
**Branch on Greater Than Zero Likely**

<table>
<thead>
<tr>
<th>bit</th>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>BGTZL</td>
<td>010111</td>
<td>rs</td>
<td>0</td>
<td>00000</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** `BGTZL rs, offset`

**MIPS32 (MIPS II)**

**Purpose:**
To test a GPR then do a PC-relative conditional branch; execute the delay slot only if the branch is taken.

**Description:**
if `rs > 0` then `branch_likely`

An 18-bit signed offset (the 16-bit `offset` field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR `rs` are greater than zero (sign bit is 0 but value not zero), branch to the effective target address after the instruction in the delay slot is executed. If the branch is not taken, the instruction in the delay slot is not executed.

**Restrictions:**
Processor operation is **UNPREDICTABLE** if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

**Operation:**

```
I: target_offset ← sign_extend(offset || \text{0}^2)
   condition ← \text{GPR[rs]} > \text{0}^{\text{GPRLEN}}

I+1: if condition then
      PC ← PC + target_offset
   else
      NullifyCurrentInstruction()
   endif
```

**Exceptions:**

None
Programming Notes:

With the 18-bit signed instruction offset, the conditional branch range is ± 128 KBytes. Use jump (J) or jump register (JR) instructions to branch to addresses outside this range.

Software is strongly encouraged to avoid the use of the Branch Likely instructions, as they will be removed from a future revision of the MIPS Architecture.

Some implementations always predict the branch will be taken, so there is a significant penalty if the branch is not taken. Software should only use this instruction when there is a very high probability (98% or more) that the branch will be taken. If the branch is not likely to be taken or if the probability of a taken branch is unknown, software is encouraged to use the BGTZ instruction instead.

Historical Information:

In the MIPS I architecture, this instruction signaled a Reserved Instruction Exception.
Branch on Less Than or Equal to Zero

### Format:

- **MIPS32 (MIPS I)**

#### Purpose:

To test a GPR then do a PC-relative conditional branch

#### Description:

- **if rs \leq 0 then branch**

An 18-bit signed offset (the 16-bit \textit{offset} field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR \( rs \) are less than or equal to zero (sign bit is 1 or value is zero), branch to the effective target address after the instruction in the delay slot is executed.

#### Restrictions:

Processor operation is **UNPREDICTABLE** if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

#### Operation:

**I:**

- \( \text{target} \_\text{offset} \leftarrow \text{sign} \_\text{extend}(\text{offset} \mid \mid 0^2) \)
- \( \text{condition} \leftarrow \text{GPR}[rs] \leq 0^\text{GPRLEN} \)

**I+1:**

- **if** condition **then**
  - \( \text{PC} \leftarrow \text{PC} + \text{target} \_\text{offset} \)
- **endif**

#### Exceptions:

None

#### Programming Notes:

With the 18-bit signed instruction offset, the conditional branch range is \( \pm 128 \text{ KBytes} \). Use jump (\( J \)) or jump register (\( JR \)) instructions to branch to addresses outside this range.
Branch on Less Than or Equal to Zero Likely

**Format:** \texttt{BLEZL rs, offset}  

**MIPS32 (MIPS II)**

**Purpose:**
To test a GPR then do a PC-relative conditional branch; execute the delay slot only if the branch is taken.

**Description:** if rs ≤ 0 then branch Likely

An 18-bit signed offset (the 16-bit \textit{offset} field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR \( rs \) are less than or equal to zero (sign bit is 1 or value is zero), branch to the effective target address after the instruction in the delay slot is executed. If the branch is not taken, the instruction in the delay slot is not executed.

**Restrictions:**
Processor operation is UNPREDICTABLE if a branch, jump, ERET, DERET, or \textsc{wait} instruction is placed in the delay slot of a branch or jump.

**Operation:**

\begin{verbatim}
I: target_offset ← sign_extend(offset || 0^2)
   condition ← GPR[rs] ≤ 0^GPRLEN
I+1: if condition then
      PC ← PC + target_offset
   else
      NullifyCurrentInstruction()
endif
\end{verbatim}

**Exceptions:**
None
Branch on Less Than or Equal to Zero Likely (cont.)  BLEZL

**Programming Notes:**

With the 18-bit signed instruction offset, the conditional branch range is $\pm 128$ KBytes. Use jump (J) or jump register (JR) instructions to branch to addresses outside this range.

Software is strongly encouraged to avoid the use of the Branch Likely instructions, as they will be removed from a future revision of the MIPS Architecture.

Some implementations always predict the branch will be taken, so there is a significant penalty if the branch is not taken. Software should only use this instruction when there is a very high probability (98% or more) that the branch will be taken. If the branch is not likely to be taken or if the probability of a taken branch is unknown, software is encouraged to use the BLEZ instruction instead.

**Historical Information:**

In the MIPS I architecture, this instruction signaled a Reserved Instruction Exception.
Branch on Less Than Zero

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>REGIMM</td>
<td>rs</td>
<td>BLTZ</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000001</td>
<td></td>
<td>0000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Format: `BLTZ rs, offset`

Purpose:
To test a GPR then do a PC-relative conditional branch

Description: if $rs < 0$ then branch

An 18-bit signed offset (the 16-bit `offset` field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR $rs$ are less than zero (sign bit is 1), branch to the effective target address after the instruction in the delay slot is executed.

Restrictions:
Processor operation is **UNPREDICTABLE** if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

Operation:

- **I**: target_offset ← sign_extend(offset $||$ 0$^2$)
  condition ← GPR[rs] < 0$^GPRLEN$

- **I+1**: if condition then
  PC ← PC + target_offset
  endif

Exceptions:
None

Programming Notes:
With the 18-bit signed instruction offset, the conditional branch range is ±128 KBytes. Use jump and link (JAL) or jump and link register (JALR) instructions for procedure calls to addresses outside this range.
Branch on Less Than Zero and Link

<table>
<thead>
<tr>
<th>REGIMM</th>
<th>rs</th>
<th>BLTZAL</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>000001</td>
<td>6</td>
<td>10000</td>
<td>16</td>
</tr>
</tbody>
</table>

Format: \texttt{BLTZAL rs, offset} \hspace{1cm} \texttt{MIPS32 (MIPS I)}

Purpose:
To test a GPR then do a PC-relative conditional procedure call

Description: if \( rs < 0 \) then \texttt{procedure\_call}

Place the return address link in GPR 31. The return link is the address of the second instruction following the branch, where execution continues after a procedure call.

An 18-bit signed offset (the 16-bit \texttt{offset} field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR \( rs \) are less than zero (sign bit is 1), branch to the effective target address after the instruction in the delay slot is executed.

Restrictions:
GPR 31 must not be used for the source register \( rs \), because such an instruction does not have the same effect when reexecuted. The result of executing such an instruction is UNPREDICTABLE. This restriction permits an exception handler to resume execution by reexecuting the branch when an exception occurs in the branch delay slot.

Processor operation is UNPREDICTABLE if a branch, jump, ERET, DERET, or \texttt{WAIT} instruction is placed in the delay slot of a branch or jump.

Operation:
\[
\begin{align*}
\text{I:} & \quad \text{target\_offset} \leftarrow \text{sign\_extend}(\text{offset} \mid \mid 0^2) \\
& \quad \text{condition} \leftarrow \text{GPR}[rs] < 0^{\text{GPRLEN}} \\
& \quad \text{GPR}[31] \leftarrow \text{PC} + 8 \\
\text{I+1:} & \quad \text{if condition then} \\
& \quad \quad \text{PC} \leftarrow \text{PC} + \text{target\_offset} \\
& \quad \text{endif}
\end{align*}
\]

Exceptions:
None

Programming Notes:
With the 18-bit signed instruction offset, the conditional branch range is \( \pm 128 \text{KBytes} \). Use jump and link (JAL) or jump and link register (JALR) instructions for procedure calls to addresses outside this range.
Branch on Less Than Zero and Link Likely

**Format:** \texttt{BLTZALL rs, offset}  

**MIPS32 (MIPS II)**

**Purpose:**
To test a GPR then do a PC-relative conditional procedure call; execute the delay slot only if the branch is taken.

**Description:** if \( rs < 0 \) then procedure\_call\_likely

Place the return address link in GPR 31. The return link is the address of the second instruction following the branch, where execution continues after a procedure call.

An 18-bit signed offset (the 16-bit \texttt{offset} field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR \( rs \) are less than zero (sign bit is 1), branch to the effective target address after the instruction in the delay slot is executed. If the branch is not taken, the instruction in the delay slot is not executed.

**Restrictions:**
GPR 31 must not be used for the source register \( rs \), because such an instruction does not have the same effect when reexecuted. The result of executing such an instruction is UNPREDICTABLE. This restriction permits an exception handler to resume execution by reexecuting the branch when an exception occurs in the branch delay slot.

Processor operation is **UNPREDICTABLE** if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

**Operation:**

\begin{align*}
\text{I:} & \quad \text{target\_offset} \leftarrow \text{sign\_extend}(\text{offset} \ || \ 0^2) \\
& \quad \text{condition} \leftarrow \text{GPR}[rs] < 0^{\text{GPRLEN}} \\
& \quad \text{GPR}[31] \leftarrow \text{PC} + 8 \\
\text{I+1:} & \quad \text{if condition then} \\
& \quad \quad \text{PC} \leftarrow \text{PC} + \text{target\_offset} \\
& \quad \text{else} \\
& \quad \quad \text{NullifyCurrentInstruction()} \\
& \quad \text{endif}
\end{align*}

**Exceptions:**
None
Programming Notes:

With the 18-bit signed instruction offset, the conditional branch range is ±128 KBytes. Use jump and link (JAL) or jump and link register (JALR) instructions for procedure calls to addresses outside this range.

Software is strongly encouraged to avoid the use of the Branch Likely instructions, as they will be removed from a future revision of the MIPS Architecture.

Some implementations always predict the branch will be taken, so there is a significant penalty if the branch is not taken. Software should only use this instruction when there is a very high probability (98% or more) that the branch will be taken. If the branch is not likely to be taken or if the probability of a taken branch is unknown, software is encouraged to use the BLTZAL instruction instead.

Historical Information:

In the MIPS I architecture, this instruction signaled a Reserved Instruction Exception.
Branch on Less Than Zero Likely

<table>
<thead>
<tr>
<th>REGIMM</th>
<th>rs</th>
<th>BLTZL</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>000001</td>
<td>6</td>
<td>5</td>
<td>16</td>
</tr>
<tr>
<td>5</td>
<td>15</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** BLTZL rs, offset

**MIPS32 (MIPS II)**

**Purpose:**
To test a GPR then do a PC-relative conditional branch; execute the delay slot only if the branch is taken.

**Description:** if rs < 0 then branch_likely
An 18-bit signed offset (the 16-bit offset field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.
If the contents of GPR rs are less than zero (sign bit is 1), branch to the effective target address after the instruction in the delay slot is executed. If the branch is not taken, the instruction in the delay slot is not executed.

**Restrictions:**
Processor operation is **UNPREDICTABLE** if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

**Operation:**

\[
\text{I: } \text{target\_offset} \leftarrow \text{sign\_extend}([\text{offset} \ || \ 0^2]) \\
\text{condition} \leftarrow \text{GPR}[\text{rs}] < 0^\text{GPR\_LEN} \\
\text{I+1: if condition then} \\
\text{PC} \leftarrow \text{PC} + \text{target\_offset} \\
\text{else} \\
\text{NullifyCurrentInstruction()} \\
\text{endif}
\]

**Exceptions:**
None
Programming Notes:

With the 18-bit signed instruction offset, the conditional branch range is ± 128 KBytes. Use jump (J) or jump register (JR) instructions to branch to addresses outside this range.

Software is strongly encouraged to avoid the use of the Branch Likely instructions, as they will be removed from a future revision of the MIPS Architecture.

Some implementations always predict the branch will be taken, so there is a significant penalty if the branch is not taken. Software should only use this instruction when there is a very high probability (98% or more) that the branch will be taken. If the branch is not likely to be taken or if the probability of a taken branch is unknown, software is encouraged to use the BLTZ instruction instead.

Historical Information:

In the MIPS I architecture, this instruction signaled a Reserved Instruction Exception.
Branch on Not Equal

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>BNE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>000101</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>16</td>
</tr>
</tbody>
</table>

Format: \( \text{BNE \, \text{rs, \, rt, \, offset}} \)

MIPS32 (MIPS I)

Purpose:

To compare GPRs then do a PC-relative conditional branch

Description: if \( \text{rs} \neq \text{rt} \) then branch

An 18-bit signed offset (the 16-bit offset field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR \( \text{rs} \) and GPR \( \text{rt} \) are not equal, branch to the effective target address after the instruction in the delay slot is executed.

Restrictions:

Processor operation is \textbf{UNPREDICTABLE} if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

Operation:

\[
\begin{align*}
\text{I:} & \quad \text{target\_offset} \leftarrow \text{sign\_extend}(\text{offset} || 0^2) \\
& \quad \text{condition} \leftarrow (\text{GPR}[\text{rs}] \neq \text{GPR}[\text{rt}]) \\
\text{I+1:} & \quad \text{if condition then} \\
& \quad \quad \text{PC} \leftarrow \text{PC} + \text{target\_offset} \\
& \quad \text{endif}
\end{align*}
\]

Exceptions:

None

Programming Notes:

With the 18-bit signed instruction offset, the conditional branch range is \( \pm 128 \text{ KBytes} \). Use jump (J) or jump register (JR) instructions to branch to addresses outside this range.
Branch on Not Equal Likely

**Format:** BNEL rs, rt, offset

**MIPS32 (MIPS II)**

**Purpose:**
To compare GPRs then do a PC-relative conditional branch; execute the delay slot only if the branch is taken.

**Description:** if rs ≠ rt then branch likely

An 18-bit signed offset (the 16-bit offset field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR rs and GPR rt are not equal, branch to the effective target address after the instruction in the delay slot is executed. If the branch is not taken, the instruction in the delay slot is not executed.

**Restrictions:**
Processor operation is **UNPREDICTABLE** if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

**Operation:**

| I:     | target_offset ← sign_extend(offset || 0²) |
|--------|---------------------------------|
| I+1:   | condition ← (GPR[rs] ≠ GPR[rt]) |
|        | if condition then |
|        | PC ← PC + target_offset |
|        | else |
|        | NullifyCurrentInstruction() |
|        | endif |

**Exceptions:**
None
Programming Notes:

With the 18-bit signed instruction offset, the conditional branch range is ±128 KBytes. Use jump (J) or jump register (JR) instructions to branch to addresses outside this range.

Software is strongly encouraged to avoid the use of the Branch Likely instructions, as they will be removed from a future revision of the MIPS Architecture.

Some implementations always predict the branch will be taken, so there is a significant penalty if the branch is not taken. Software should only use this instruction when there is a very high probability (98% or more) that the branch will be taken. If the branch is not likely to be taken or if the probability of a taken branch is unknown, software is encouraged to use the BNE instruction instead.

Historical Information:

In the MIPS I architecture, this instruction signaled a Reserved Instruction Exception.
Breakpoint

**Format:** BREAK

**Purpose:**
To cause a Breakpoint exception

**Description:**
A breakpoint exception occurs, immediately and unconditionally transferring control to the exception handler. The `code` field is available for use as software parameters, but is retrieved by the exception handler only by loading the contents of the memory word containing the instruction.

**Restrictions:**
None

**Operation:**
```
SignalException(Breakpoint)
```

**Exceptions:**
Breakpoint
Floating Point Compare

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>COP1</td>
<td>fmt</td>
<td>ft</td>
<td>fs</td>
<td>cc</td>
<td>0</td>
<td>A</td>
<td>FC</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| 6 | 5 | 5 | 5 | 3 | 1 | 1 |   |   |

**Format:**

- C.cond.S fs, ft (cc = 0 implied)  
- C.cond.D fs, ft (cc = 0 implied)  
- C.cond.S cc, fs, ft  
- C.cond.D cc, fs, ft

**MIPS32 (MIPS I)**

**Purpose:**

To compare FP values and record the Boolean result in a condition code.

**Description:**

`cc ← fs compare_cond ft`

The value in FPR `fs` is compared to the value in FPR `ft`; the values are in format `fmt`. The comparison is exact and neither overflows nor underflows.

If the comparison specified by `cond2..1` is true for the operand values, the result is true; otherwise, the result is false. If no exception is taken, the result is written into condition code `CC`; true is 1 and false is 0.

If one of the values is an SNaN, or `cond3` is set and at least one of the values is a QNaN, an Invalid Operation condition is raised and the Invalid Operation flag is set in the FCSR. If the Invalid Operation `Enable` bit is set in the FCSR, no result is written and an Invalid Operation exception is taken immediately. Otherwise, the Boolean result is written into condition code `CC`.

There are four mutually exclusive ordering relations for comparing floating point values; one relation is always true and the others are false. The familiar relations are `greater than`, `less than`, and `equal`. In addition, the IEEE floating point standard defines the relation `unordered`, which is true when at least one operand value is NaN; NaN compares unordered with everything, including itself. Comparisons ignore the sign of zero, so +0 equals -0.

The comparison condition is a logical predicate, or equation, of the ordering relations such as `less than or equal`, `equal`, `not less than`, or `unordered or equal`. Compare distinguishes among the 16 comparison predicates. The Boolean result of the instruction is obtained by substituting the Boolean value of each ordering relation for the two FP values in the equation. If the `equal` relation is true, for example, then all four example predicates above yield a true result. If the `unordered` relation is true then only the final predicate, `unordered or equal`, yields a true result.

Logical negation of a compare result allows eight distinct comparisons to test for the 16 predicates as shown in . Each mnemonic tests for both a predicate and its logical negation. For each mnemonic, `compare` tests the truth of the first predicate. When the first predicate is true, the result is true as shown in the “If Predicate Is True” column, and the second predicate must be false, and vice versa. (Note that the False predicate is never true and False/True do not follow the normal pattern.)

The truth of the second predicate is the logical negation of the instruction result. After a compare instruction, test for the truth of the first predicate can be made with the Branch on FP True (BC1T) instruction and the truth of the second can be made with Branch on FP False (BC1F).
Table 3-24 shows another set of eight compare operations, distinguished by a \( cond_3 \) value of 1 and testing the same 16 conditions. For these additional comparisons, if at least one of the operands is a NaN, including Quiet NaN, then an Invalid Operation condition is raised. If the Invalid Operation condition is enabled in the FCSR, an Invalid Operation exception occurs.

### Table 3-24 FPU Comparisons Without Special Operand Exceptions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Comparison Predicate</th>
<th>Cond Mnemonic</th>
<th>Comparison CC Result</th>
<th>Inv Op Excp. if QNaN?</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Name of Predicate and Logically Negated Predicate (Abbreviation)</td>
<td>Relation Values</td>
<td>If Predicate Is True</td>
<td>Condition Field</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&gt;</td>
<td>&lt;</td>
<td>=</td>
<td>?</td>
</tr>
<tr>
<td>F</td>
<td>False [this predicate is always False]</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td></td>
<td>True (T)</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>UN</td>
<td>Unordered</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td></td>
<td>Ordered (OR)</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>EQ</td>
<td>Equal</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td></td>
<td>Not Equal (NEQ)</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>UEQ</td>
<td>Unordered or Equal</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td></td>
<td>Ordered or Greater Than or Less Than (OGL)</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>OLT</td>
<td>Ordered or Less Than</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td></td>
<td>Unordered or Greater Than or Equal (UGE)</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>ULT</td>
<td>Unordered or Less Than</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td></td>
<td>Ordered or Greater Than or Equal (OGE)</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>OLE</td>
<td>Ordered or Less Than or Equal</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td></td>
<td>Unordered or Greater Than (UGT)</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>ULE</td>
<td>Unordered or Less Than or Equal</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td></td>
<td>Ordered or Greater Than (OGT)</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>

Key: \( ? = \text{unordered}, \geq = \text{greater than}, \leq = \text{less than}, = \text{equal}, T = \text{True}, F = \text{False} \)

Floating Point Compare (cont.)

C.cond.fmt
# Table 3-25 FPU Comparisons With Special Operand Exceptions for QNaNs

<table>
<thead>
<tr>
<th>Condition Mnemonic</th>
<th>Comparison Predicate</th>
<th>Relation Values</th>
<th>If Predicate Is True</th>
<th>Inv Op Excp If QNaN?</th>
<th>Instruction Result</th>
<th>Condition Field</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>Name of Predicate and Logically Negated Predicate (Abbreviation)</strong></td>
<td><strong>Relation Values</strong></td>
<td><strong>If Predicate Is True</strong></td>
<td><strong>Inv Op Excp If QNaN?</strong></td>
<td><strong>Instruction Result</strong></td>
<td><strong>Condition Field</strong></td>
</tr>
<tr>
<td>SF</td>
<td>Signaling False [this predicate always False]</td>
<td>F F F F</td>
<td>F</td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Signaling True (ST)</td>
<td>T T T T</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NGLE</td>
<td>Not Greater Than or Less Than or Equal</td>
<td>F F F T</td>
<td>T</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Greater Than or Less Than or Equal (GLE)</td>
<td>T T T F</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SEQ</td>
<td>Signaling Equal</td>
<td>F F T F</td>
<td>T</td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>Signaling Not Equal (SNE)</td>
<td>T T F T</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NGL</td>
<td>Not Greater Than or Less Than</td>
<td>F F T T</td>
<td>T</td>
<td></td>
<td>Yes</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>Greater Than or Less Than (GL)</td>
<td>T T F F</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LT</td>
<td>Less Than</td>
<td>F T F F</td>
<td>T</td>
<td></td>
<td></td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>Not Less Than (NLT)</td>
<td>T F T T</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NGE</td>
<td>Not Greater Than or Equal</td>
<td>F T F T</td>
<td>T</td>
<td></td>
<td></td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>Greater Than or Equal (GE)</td>
<td>T F T F</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LE</td>
<td>Less Than or Equal</td>
<td>F T T F</td>
<td>T</td>
<td></td>
<td></td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>Not Less Than or Equal (NLE)</td>
<td>T F F T</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NGT</td>
<td>Not Greater Than</td>
<td>F T T T</td>
<td>T</td>
<td></td>
<td></td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>Greater Than (GT)</td>
<td>T F F F</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Key: ? = unordered, > = greater than, < = less than, = is equal, T = True, F = False
Floating Point Compare (cont.)

**Restrictions:**

The fields *fs* and *ft* must specify FPRs valid for operands of type *fmt*; if they are not valid, the result is **UNPREDICTABLE**.

The operands must be values in format *fmt*; if they are not, the result is **UNPREDICTABLE** and the value of the operand FPRs becomes **UNPREDICTABLE**.

**Operation:**

\[
\begin{align*}
\text{if } & \text{NaN(ValueFPR}(fs, fmt)) \text{ or NaN(ValueFPR}(ft, fmt)) \text{ or} \\
& \text{QNaN(ValueFPR}(fs, fmt)) \text{ or QNaN(ValueFPR}(ft, fmt)) \text{ then} \\
& \text{less } \leftarrow \text{false} \\
& \text{equal } \leftarrow \text{false} \\
& \text{unordered } \leftarrow \text{true} \\
\text{if } & \text{(NaN(ValueFPR}(fs,fmt)) \text{ or NaN(ValueFPR}(ft,fmt)) \text{ or} \\
& \text{(cond 3 and (QNaN(ValueFPR}(fs,fmt)) \text{ or QNaN(ValueFPR}(ft,fmt)))) \text{ then} \\
& \text{SignalException(InvalidOperation)} \\
\text{endif} \\
\text{endif} \\
\text{else} \\
& \text{less } \leftarrow \text{ValueFPR}(fs, fmt) <_{fmt} \text{ValueFPR}(ft, fmt) \\
& \text{equal } \leftarrow \text{ValueFPR}(fs, fmt) =_{fmt} \text{ValueFPR}(ft, fmt) \\
& \text{unordered } \leftarrow \text{false} \\
\text{endif} \\
\text{condition } \leftarrow \text{(cond 2 and less) or (cond 1 and equal) or (cond 0 and unordered)} \\
\text{SetFPConditionCode}(cc, \text{condition})
\end{align*}
\]
Exceptions:
Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:
Unimplemented Operation, Invalid Operation

Programming Notes:
FP computational instructions, including compare, that receive an operand value of Signaling NaN raise the Invalid Operation condition. Comparisons that raise the Invalid Operation condition for Quiet NaNs in addition to SNaNs permit a simpler programming model if NaNs are errors. Using these compares, programs do not need explicit code to check for QNaNs causing the unordered relation. Instead, they take an exception and allow the exception handling system to deal with the error when it occurs. For example, consider a comparison in which we want to know if two numbers are equal, but for which unordered would be an error.

```
# comparisons using explicit tests for QNaN
  c.eq.d $f2,$f4# check for equal
  nop
  bclt  L2    # it is equal
  c.un.d $f2,$f4# it is not equal,
    # but might be unordered
  bclt  ERROR  # unordered goes off to an error handler
# not-equal-case code here
...
# equal-case code here
L2:
# --------------------------------------------------------------
# comparison using comparisons that signal QNaN
  c.seq.d $f2,$f4# check for equal
  nop
  bclt  L2    # it is equal
  nop
# it is not unordered here
...
# not-equal-case code here
...
# equal-case code here
```

Historical Information:
The MIPS I architecture defines a single floating point condition code, implemented as the coprocessor 1 condition signal (Cp1Cond) and the C bit in the FP Control/Status register. MIPS I, II, and III architectures must have the CC field set to 0, which is implied by the first format in the “Format” section.

The MIPS IV and MIPS32 architectures add seven more Condition Code bits to the original condition code 0. FP compare and conditional branch instructions specify the Condition Code bit to set or test. Both assembler formats are valid for MIPS IV and MIPS32.

In the MIPS I, II, and III architectures there must be at least one instruction between the compare instruction that sets the condition code and the branch instruction that tests it. Hardware does not detect a violation of this restriction.
Perform Cache Operation

### Format: CACHE op, offset(base)

#### Purpose:
To perform the cache operation specified by op.

#### Description:
The 16-bit offset is sign-extended and added to the contents of the base register to form an effective address. The effective address is used in one of the following ways based on the operation to be performed and the type of cache as described in the following table.

<table>
<thead>
<tr>
<th>Operation Requires an Type of Cache</th>
<th>Usage of Effective Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address Virtual</td>
<td>The effective address is used to address the cache. It is implementation dependent whether an address translation is performed on the effective address (with the possibility that a TLB Refill or TLB Invalid exception might occur)</td>
</tr>
<tr>
<td>Address Physical</td>
<td>The effective address is translated by the MMU to a physical address. The physical address is then used to address the cache</td>
</tr>
<tr>
<td>Index N/A</td>
<td>The effective address is translated by the MMU to a physical address. It is implementation dependent whether the effective address or the translated physical address is used to index the cache. Assuming that the total cache size in bytes is CS, the associativity is A, and the number of bytes per tag is BPT, the following calculations give the fields of the address which specify the way and the index: OffsetBit ← Log2(BPT) IndexBit ← Log2(CS / A) WayBit ← IndexBit + Ceiling(Log2(A)) Way ← AddrWayBit-1..IndexBit Index ← AddrIndexBit-1..OffsetBit For a direct-mapped cache, the Way calculation is ignored and the Index value fully specifies the cache tag. This is shown symbolically in the figure below.</td>
</tr>
</tbody>
</table>

### Table 3-26 Usage of Effective Address
A TLB Refill and TLB Invalid (both with cause code equal TLBL) exception can occur on any operation. For index operations (where the address is used to index the cache but need not match the cache tag) software should use unmapped addresses to avoid TLB exceptions. This instruction never causes TLB Modified exceptions nor TLB Refill exceptions with a cause code of TLBS, nor data Watch exceptions.

A Cache Error exception may occur as a byproduct of some operations performed by this instruction. For example, if a Writeback operation detects a cache or bus error during the processing of the operation, that error is reported via a Cache Error exception. Similarly, a Bus Error Exception may occur if a bus operation invoked by this instruction is terminated in an error.

An Address Error Exception (with cause code equal AdEL) may occur if the effective address references a portion of the kernel address space which would normally result in such an exception. It is implementation dependent whether such an exception does occur.

It is implementation dependent whether a data watch is triggered by a cache instruction whose address matches the Watch register address match conditions.

Bits [17:16] of the instruction specify the cache on which to perform the operation, as follows:

**Table 3-27 Encoding of Bits[17:16] of CACHE Instruction**

<table>
<thead>
<tr>
<th>Code</th>
<th>Name</th>
<th>Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>2#00</td>
<td>I</td>
<td>Primary Instruction</td>
</tr>
<tr>
<td>2#01</td>
<td>D</td>
<td>Primary Data or Unified Primary</td>
</tr>
<tr>
<td>2#10</td>
<td>T</td>
<td>Tertiary</td>
</tr>
<tr>
<td>2#11</td>
<td>S</td>
<td>Secondary</td>
</tr>
</tbody>
</table>

Bits [20:18] of the instruction specify the operation to perform. To provide software with a consistent base of cache operations, certain encodings must be supported on all processors. The remaining encodings are recommended.
<table>
<thead>
<tr>
<th>Code</th>
<th>Caches</th>
<th>Name</th>
<th>Effective Address Operand Type</th>
<th>Operation</th>
<th>Compliance</th>
</tr>
</thead>
<tbody>
<tr>
<td>2#000</td>
<td>I</td>
<td>Index Invalidate</td>
<td>Index</td>
<td>Set the state of the cache block at the specified index to invalid. This required encoding may be used by software to invalidate the entire instruction cache by stepping through all valid indices.</td>
<td>Required</td>
</tr>
<tr>
<td></td>
<td>D</td>
<td>Index Writeback Invalidate / Index Invalidate</td>
<td>Index</td>
<td>For a write-back cache: If the state of the cache block at the specified index is valid and dirty, write the block back to the memory address specified by the cache tag. After that operation is completed, set the state of the cache block to invalid. If the block is valid but not dirty, set the state of the block to invalid.</td>
<td>Required</td>
</tr>
<tr>
<td></td>
<td>S, T</td>
<td>Index Writeback Invalidate / Index Invalidate</td>
<td>Index</td>
<td>For a write-through cache: Set the state of the cache block at the specified index to invalid. This required encoding may be used by software to invalidate the entire data cache by stepping through all valid indices. Note that Index Store Tag should be used to initialize the cache at powerup.</td>
<td>Optional</td>
</tr>
<tr>
<td>2#001</td>
<td>All</td>
<td>Index Load Tag</td>
<td>Index</td>
<td>Read the tag for the cache block at the specified index into the TagLo and TagHi Coprocessor 0 registers. If the DataLo and DataHi registers are implemented, also read the data corresponding to the byte index into the DataLo and DataHi registers. The granularity and alignment of the data read into the DataLo and DataHi registers is implementation-dependent, but is typically the result of an aligned access to the cache, ignoring the appropriate low-order bits of the byte index.</td>
<td>Recommended</td>
</tr>
</tbody>
</table>
### Table 3-28 Encoding of Bits [20:18] of the CACHE Instruction

<table>
<thead>
<tr>
<th>Code</th>
<th>Caches</th>
<th>Name</th>
<th>Effective Address Operand Type</th>
<th>Operation</th>
<th>Compliance</th>
</tr>
</thead>
<tbody>
<tr>
<td>2#010</td>
<td>All</td>
<td>Index Store Tag</td>
<td>Index</td>
<td>Write the tag for the cache block at the specified index from the TagLo and TagHi Coprocessor 0 registers. This required encoding may be used by software to initialize the entire instruction of data caches by stepping through all valid indices. Doing so requires that the TagLo and TagHi registers associated with the cache be initialized first.</td>
<td>Required</td>
</tr>
<tr>
<td>2#011</td>
<td>All</td>
<td>Implementation Dependent</td>
<td>Unspecified</td>
<td>Available for implementation-dependent operation.</td>
<td>Optional</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2#100</td>
<td>I, D</td>
<td>Hit Invalidate</td>
<td>Address</td>
<td>If the cache block contains the specified address, set the state of the cache block to invalid. This required encoding may be used by software to invalidate a range of addresses from the instruction cache by stepping through the address range by the line size of the cache.</td>
<td>Required (Instruction Cache Encoding Only), Recommended otherwise</td>
</tr>
<tr>
<td></td>
<td>S, T</td>
<td>Hit Invalidate</td>
<td>Address</td>
<td></td>
<td>Optional</td>
</tr>
<tr>
<td></td>
<td>I</td>
<td>Fill</td>
<td>Address</td>
<td>Fill the cache from the specified address.</td>
<td>Recommended</td>
</tr>
<tr>
<td></td>
<td>D</td>
<td>Hit Writeback Invalidate / Hit Invalidate</td>
<td>Address</td>
<td>For a write-back cache: If the cache block contains the specified address and it is valid and dirty, write the contents back to memory. After that operation is completed, set the state of the cache block to invalid. If the block is valid but not dirty, set the state of the block to invalid. For a write-through cache: If the cache block contains the specified address, set the state of the cache block to invalid. This required encoding may be used by software to invalidate a range of addresses from the data cache by stepping through the address range by the line size of the cache.</td>
<td>Required</td>
</tr>
<tr>
<td>2#101</td>
<td>S, T</td>
<td>Hit Writeback Invalidate / Hit Invalidate</td>
<td>Address</td>
<td></td>
<td>Optional</td>
</tr>
</tbody>
</table>
D Hit Writeback Address

If the cache block contains the specified address and it is valid and dirty, write the contents back to memory. After the operation is completed, leave the state of the line valid, but clear the dirty state. For a write-through cache, this operation may be treated as a nop.

Recommended

S, T Hit Writeback Address

If the cache does not contain the specified address, fill it from memory, performing a writeback if required, and set the state to valid and locked. If the cache already contains the specified address, set the state to locked. In set-associative or fully-associative caches, the way selected on a fill from memory is implementation dependent.

The lock state may be cleared by executing an Index Invalidate, Index Writeback Invalidate, Hit Invalidate, or Hit Writeback Invalidate operation to the locked line, or via an Index Store Tag operation to the line that clears the lock bit. Note that clearing the lock state via Index Store Tag is dependent on the implementation-dependent cache tag and cache line organization, and that Index and Index Writeback Invalidate operations are dependent on cache line organization. Only Hit and Hit Writeback Invalidate operations are generally portable across implementations.

Recommended

I

Fetch and Lock Address

If the cache does not contain the specified address, fill it from memory, performing a writeback if required, and set the state to valid and locked. If the cache already contains the specified address, set the state to locked. In set-associative or fully-associative caches, the way selected on a fill from memory is implementation dependent.

The lock state may be cleared by executing an Index Invalidate, Index Writeback Invalidate, Hit Invalidate, or Hit Writeback Invalidate operation to the locked line, or via an Index Store Tag operation to the line that clears the lock bit. Note that clearing the lock state via Index Store Tag is dependent on the implementation-dependent cache tag and cache line organization, and that Index and Index Writeback Invalidate operations are dependent on cache line organization. Only Hit and Hit Writeback Invalidate operations are generally portable across implementations.

Recommended

It is implementation dependent whether a locked line is displaced as the result of an external invalidate or intervention that hits on the locked line. Software must not depend on the locked line remaining in the cache if an external invalidate or intervention would invalidate the line if it were not locked.

It is implementation dependent whether a Fetch and Lock operation affects more than one line. For example, more than one line around the referenced address may be fetched and locked. It is recommended that only the single line containing the referenced address be affected.

Table 3-28 Encoding of Bits [20:18] of the CACHE Instruction

<table>
<thead>
<tr>
<th>Code</th>
<th>Caches</th>
<th>Name</th>
<th>Effective Address Operand Type</th>
<th>Operation</th>
<th>Compliance</th>
</tr>
</thead>
<tbody>
<tr>
<td>2#110</td>
<td>D</td>
<td>Hit Writeback</td>
<td>Address</td>
<td>If the cache block contains the specified address and it is valid and dirty, write the contents back to memory. After the operation is completed, leave the state of the line valid, but clear the dirty state. For a write-through cache, this operation may be treated as a nop.</td>
<td>Recommended</td>
</tr>
<tr>
<td></td>
<td>S, T</td>
<td>Hit Writeback</td>
<td>Address</td>
<td>If the cache does not contain the specified address, fill it from memory, performing a writeback if required, and set the state to valid and locked. If the cache already contains the specified address, set the state to locked. In set-associative or fully-associative caches, the way selected on a fill from memory is implementation dependent. The lock state may be cleared by executing an Index Invalidate, Index Writeback Invalidate, Hit Invalidate, or Hit Writeback Invalidate operation to the locked line, or via an Index Store Tag operation to the line that clears the lock bit. Note that clearing the lock state via Index Store Tag is dependent on the implementation-dependent cache tag and cache line organization, and that Index and Index Writeback Invalidate operations are dependent on cache line organization. Only Hit and Hit Writeback Invalidate operations are generally portable across implementations. It is implementation dependent whether a locked line is displaced as the result of an external invalidate or intervention that hits on the locked line. Software must not depend on the locked line remaining in the cache if an external invalidate or intervention would invalidate the line if it were not locked. It is implementation dependent whether a Fetch and Lock operation affects more than one line. For example, more than one line around the referenced address may be fetched and locked. It is recommended that only the single line containing the referenced address be affected.</td>
<td>Optional</td>
</tr>
</tbody>
</table>

MIPS32™ Architecture For Programmers Volume II, Revision 0.95
Restrictions:
The operation of this instruction is **UNDEFINED** for any operation/cache combination that is not implemented.
The operation of this instruction is **UNDEFINED** if the operation requires an address, and that address is uncacheable.

Operation:

\[
\begin{align*}
\text{vAddr} & \leftarrow \text{GPR}[\text{base}] + \text{sign}_\text{extend}(\text{offset}) \\
(\text{pAddr, uncached}) & \leftarrow \text{AddressTranslation(vAddr, DataReadReference)} \\
\text{CacheOp}(\text{op, vAddr, pAddr})
\end{align*}
\]

Exceptions:
TLB Refill Exception.
TLB Invalid Exception
Coprocessor Unusable Exception
Address Error Exception
Cache Error Exception
Bus Error Exception
Floating Point Ceiling Convert to Word Fixed Point

<table>
<thead>
<tr>
<th>COP1</th>
<th>fmt</th>
<th>0</th>
<th>fs</th>
<th>fd</th>
<th>CEIL.W</th>
<th>MIPS32 (MIPS II)</th>
</tr>
</thead>
<tbody>
<tr>
<td>010001</td>
<td>00000</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>001110</td>
</tr>
</tbody>
</table>

**Format:**

- CEIL.W.S fd, fs
- CEIL.W.D fd, fs

**MIPS32 (MIPS II)**

**Purpose:**

To convert an FP value to 32-bit fixed point, rounding up

**Description:**

\[ fd \leftarrow \text{convert\_and\_round}(fs) \]

The value in FPR \( fs \), in format \( fmt \), is converted to a value in 32-bit word fixed point format and rounding toward \(+\infty\) (rounding mode 2). The result is placed in FPR \( fd \).

When the source value is Infinity, NaN, or rounds to an integer outside the range \(-2^{31}\) to \(2^{31}-1\), the result cannot be represented correctly, an IEEE Invalid Operation condition exists, and the Invalid Operation flag is set in the FCSR. If the Invalid Operation \textit{Enable} bit is set in the FCSR, no result is written to \( fd \) and an Invalid Operation exception is taken immediately. Otherwise, the default result, \(2^{31}-1\), is written to \( fd \).

**Restrictions:**

The fields \( fs \) and \( fd \) must specify valid FPRs; \( fs \) for type \( fmt \) and \( fd \) for word fixed point; if they are not valid, the result is UNPREDICTABLE.

The operand must be a value in format \( fmt \); if it is not, the result is UNPREDICTABLE and the value of the operand FPR becomes UNPREDICTABLE.

**Operation:**

\[
\text{StoreFPR}(fd, W, \text{ConvertFmt}(\text{ValueFPR}(fs, fmt), fmt, W))
\]

**Exceptions:**

Coprocessor Unusable, Reserved Instruction

**Floating Point Exceptions:**

Invalid Operation, Unimplemented Operation, Inexact, Overflow
Move Control Word From Floating Point

**Format:** CFC1 rt, fs

**Purpose:**
To copy a word from an FPU control register to a GPR

**Description:**
rt ← FP_Control[fs]

Copy the 32-bit word from FP (coprocessor 1) control register fs into GPR rt.

**Restrictions:**
There are a few control registers defined for the floating point unit. The result is UNPREDICTABLE if fs specifies a register that does not exist.

**Operation:**

if fs = 0 then
    temp ← FIR
elseif fs = 25 then
    temp ← 0^24 || FCSR31..25 || FCSR23
elseif fs = 26 then
    temp ← 0^14 || FCSR17..12 || 0^5 || FCSR6..2 || 0^2
elseif fs = 28 then
    temp ← 0^20 || FCSR11.7 || 0^4 || FCSR24 || FCSR1..0
elseif fs = 31 then
    temp ← FCSR
else
    temp ← UNPREDICTABLE
endif
GPR[rt] ← temp
Exceptions:
Coprocessor Unusable, Reserved Instruction

Historical Information:
For the MIPS I, II and III architectures, the contents of GPR $rt$ are **UNPREDICTABLE** for the instruction immediately following CFC1.

MIPS V and MIPS32 introduced the three control registers that access portions of FCSR. These registers were not available in MIPS I, II, III, or IV.
Move Control Word From Coprocessor 2

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>COP2</td>
<td>CF</td>
<td>rt</td>
<td>rd</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010010</td>
<td>00010</td>
<td>000 0000 000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** CFC2 rt, rd

**Purpose:**
To copy a word from a Coprocessor 2 control register to a GPR

**Description:** \( \text{rt} \leftarrow \text{CCR}[2, \text{rd}] \)
Copy the 32-bit word from Coprocessor 2 control register \( \text{rd} \) into GPR \( \text{rt} \).

**Restrictions:**
The result is **UNPREDICTABLE** if \( \text{fs} \) specifies a register that does not exist.

**Operation:**
\[
\begin{align*}
\text{temp} & \leftarrow \text{CCR}[2, \text{rd}] \\
\text{GPR}[\text{rt}] & \leftarrow \text{temp}
\end{align*}
\]

**Exceptions:**
Coprocessor Unusable, Reserved Instruction
### Count Leading Ones in Word

**Format:** CLO rd, rs

**Purpose:**
To Count the number of leading ones in a word

**Description:** rd ← count_leading_ones rs

Bits 31..0 of GPR rs are scanned from most significant to least significant bit. The number of leading ones is counted and the result is written to GPR rd. If all of bits 31..0 were set in GPR rs, the result written to GPR rd is 32.

**Restrictions:**
To be compliant with the MIPS32 and MIPS64 Architecture, software must place the same GPR number in both the rt and rd fields of the instruction. The operation of the instruction is **UNPREDICTABLE** if the rt and rd fields of the instruction contain different values.

**Operation:**

```
temp ← 32
for i in 31 .. 0
    if GPR[rs]_i = 0 then
        temp ← 31 - i
        break
    endif
endfor
GPR[rd] ← temp
```

**Exceptions:**
None
Count Leading Zeros in Word

<table>
<thead>
<tr>
<th>Format:</th>
<th>CLZ rd, rs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Count the number of leading zeros in a word</td>
</tr>
<tr>
<td>Description:</td>
<td>rd ← count_leading_zeros rs</td>
</tr>
<tr>
<td>Bits 31..0 of GPR rs are scanned from most significant to least significant bit. The number of leading zeros is counted and the result is written to GPR rd. If no bits were set in GPR rs, the result written to GPR rt is 32.</td>
<td></td>
</tr>
<tr>
<td>Restrictions:</td>
<td>To be compliant with the MIPS32 and MIPS64 Architecture, software must place the same GPR number in both the rt and rd fields of the instruction. The operation of the instruction is UNPREDICTABLE if the rt and rd fields of the instruction contain different values.</td>
</tr>
</tbody>
</table>
| Operation:       | temp ← 32  
|                  | for i in 31 .. 0  
|                  |   if GPR[rs]_i = 1 then  
|                  |     temp ← 31 - i  
|                  |     break  
|                  | endif  
|                  | endiffor  
|                  | GPR[rd] ← temp  |
| Exceptions:      | None   |
COP2

Format: COP2 func

Purpose:
To perform an operation to Coprocessor 2

Description: CoprocessorOperation(2, cofun)
An implementation-dependent operation is performed to Coprocessor 2, with the cofun value passed as an argument. The operation may specify and reference internal coprocessor registers, and may change the state of the coprocessor conditions, but does not modify state within the processor. Details of coprocessor operation and internal state are described in the documentation for each Coprocessor 2 implementation.

Restrictions:

Operation:
CoprocessorOperation(2, cofun)

Exceptions:
Coprocessor Unusable
Reserved Instruction
Move Control Word to Floating Point

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>COP1</td>
<td>CT</td>
<td>rt</td>
<td>fs</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010001</td>
<td>00110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000 0000 0000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:**  
CTC1 rt, fs  

**Purpose:**  
To copy a word from a GPR to an FPU control register

**Description:**  
FP-Control[fs] ← rt  

Copy the low word from GPR rt into the FP (coprocessor 1) control register indicated by fs.

Writing to the floating point Control/Status register, the FCSR, causes the appropriate exception if any Cause bit and its corresponding Enable bit are both set. The register is written before the exception occurs. Writing to FEXR to set a cause bit whose enable bit is already set, or writing to FENR to set an enable bit whose cause bit is already set causes the appropriate exception. The register is written before the exception occurs.

**Restrictions:**  
There are a few control registers defined for the floating point unit. The result is UNPREDICTABLE if fs specifies a register that does not exist.
Operation:

\[
\text{temp} \leftarrow \text{GPR}[rt]_{31..0}
\]

if \(fs = 25\) then
  if \(\text{temp}_{31..8} \neq 0^24\) then
    UNPREDICTABLE
  else
    \(\text{FCSR} \leftarrow \text{temp}_{7..1} \mid \mid \text{FCSR}_{24} \mid \mid \text{temp}_0 \mid \mid \text{FCSR}_{22..0}\)
  endif
else if \(fs = 26\) then
  if \(\text{temp}_{22..18} \neq 0\) then
    UNPREDICTABLE
  else
    \(\text{FCSR} \leftarrow \text{FCSR}_{31..18} \mid \mid \text{temp}_{17..12} \mid \mid \text{FCSR}_{11..7} \mid \mid \text{temp}_{6..2} \mid \mid \text{FCSR}_{1..0}\)
  endif
else if \(fs = 28\) then
  if \(\text{temp}_{22..18} \neq 0\) then
    UNPREDICTABLE
  else
    \(\text{FCSR} \leftarrow \text{FCSR}_{31..25} \mid \mid \text{temp}_2 \mid \mid \text{FCSR}_{23..12} \mid \mid \text{temp}_{11..7} \mid \mid \text{FCSR}_6..2 \mid \mid \text{temp}_{1..0}\)
  endif
else if \(fs = 31\) then
  if \(\text{temp}_{22..18} \neq 0\) then
    UNPREDICTABLE
  else
    \(\text{FCSR} \leftarrow \text{temp}\)
  endif
else
  UNPREDICTABLE
endif

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Unimplemented Operation, Invalid Operation, Division-by-zero, Inexact, Overflow, Underflow

Historical Information:

For the MIPS I, II and III architectures, the contents of floating point control register \(fs\) are undefined for the instruction immediately following CTC1.

MIPS V and MIPS32 introduced the three control registers that access portions of FCSR. These registers were not available in MIPS I, II, III, or IV.
Move Control Word to Coprocessor 2

CTC2

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>COP2</td>
<td>CT</td>
<td>rt</td>
<td>rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>010010</td>
<td>00110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0000000000</td>
</tr>
</tbody>
</table>

**Format:** CTC2 rt, rd

**Purpose:**
To copy a word from a GPR to a Coprocessor 2 control register

**Description:** CCR[2,rd] ← rt
Copy the low word from GPR rt into the Coprocessor 2 control register indicated by rd.

**Restrictions:**
The result is UNPREDICTABLE if rd specifies a register that does not exist.

**Operation:**

\[
\text{temp} \leftarrow \text{GPR[rt]}
\]

\[
\text{CCR}[2,\text{rd}] \leftarrow \text{temp}
\]

**Exceptions:**
Coprocessor Unusable, Reserved Instruction
Floating Point Convert to Double Floating Point

<table>
<thead>
<tr>
<th>COP1</th>
<th>fmt</th>
<th>0</th>
<th>fs</th>
<th>fd</th>
<th>CVT.D</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>00000</td>
<td>5</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

**Format:**
- CVT.D.S fd, fs
- CVT.D.W fd, fs
- CVT.D.L fd, fs

**MIPS32 (MIPS I)**
- CVT.D.S
- CVT.D.W
- CVT.D.L

**MIPS64 (MIPS III)**

**Purpose:**
To convert an FP or fixed point value to double FP

**Description:**
\[ fd \leftarrow \text{convert\_and\_round}(fs) \]

The value in FPR \( fs \), in format \( fmt \), is converted to a value in double floating point format and rounded according to the current rounding mode in \( FCSR \). The result is placed in FPR \( fd \). If \( fmt \) is S or W, then the operation is always exact.

**Restrictions:**
The fields \( fs \) and \( fd \) must specify valid FPRs—\( fs \) for type \( fmt \) and \( fd \) for double floating point—if they are not valid, the result is \textsc{unpredictable}.

The operand must be a value in format \( fmt \); if it is not, the result is \textsc{unpredictable} and the value of the operand FPR becomes \textsc{unpredictable}.

**Operation:**
\[
\text{StoreFPR} (fd, D, \text{ConvertFmt} (\text{ValueFPR}(fs, fmt), fmt, D))
\]

**Exceptions:**
- Coprocessor Unusable, Reserved Instruction

**Floating Point Exceptions:**
- Invalid Operation, Unimplemented Operation, Inexact
### Floating Point Convert to Single Floating Point

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>COP1</td>
<td>fmt</td>
<td>0</td>
<td>fs</td>
<td>fd</td>
<td>CVT.S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0000</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Format:
- CVT.S.D fd, fs
- CVT.S.W fd, fs
- CVT.S.L fd, fs

#### MIPS32 (MIPS I)
- MIPS32 (MIPS I)
- MIPS64 (MIPS III)

#### Purpose:
To convert an FP or fixed point value to single FP

#### Description:
- fd ← convert_and_round(fs)

The value in FPR fs, in format fmt, is converted to a value in single floating point format and rounded according to the current rounding mode in FCSR. The result is placed in FPR fd.

#### Restrictions:
- The fields fs and fd must specify valid FPRs—fs for type fmt and fd for single floating point. If they are not valid, the result is UNPREDICTABLE.
- The operand must be a value in format fmt; if it is not, the result is UNPREDICTABLE and the value of the operand FPR becomes UNPREDICTABLE.

#### Operation:
- StoreFPR(fd, S, ConvertFmt(ValueFPR(fs, fmt), fmt, S))

#### Exceptions:
- Coprocessor Unusable, Reserved Instruction

#### Floating Point Exceptions:
- Invalid Operation, Unimplemented Operation, Inexact, Overflow, Underflow
Floating Point Convert to Word Fixed Point

CVT.W.fmt

Format:  
CVT.W.S fd, fs  
CVT.W.D fd, fs  

MIPS32 (MIPS I)

Purpose:
To convert an FP value to 32-bit fixed point

Description:
fd ← convert_and_round(fs)

The value in FPR fs, in format fmt, is converted to a value in 32-bit word fixed point format and rounded according to the current rounding mode in FCSR. The result is placed in FPR fd.

When the source value is Infinity, NaN, or rounds to an integer outside the range -2^{31} to 2^{31}-1, the result cannot be represented correctly, an IEEE Invalid Operation condition exists, and the Invalid Operation flag is set in the FCSR. If the Invalid Operation Enable bit is set in the FCSR, no result is written to fd and an Invalid Operation exception is taken immediately. Otherwise, the default result, 2^{31}−1, is written to fd.

Restrictions:
The fields fs and fd must specify valid FPRs—fs for type fmt and fd for word fixed point—if they are not valid, the result is UNPREDICTABLE.

The operand must be a value in format fmt; if it is not, the result is UNPREDICTABLE and the value of the operand FPR becomes UNPREDICTABLE.

Operation:

StoreFPR(fd, W, ConvertFmt(ValueFPR(fs, fmt), fmt, W))

Exceptions:
Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:
Invalid Operation, Unimplemented Operation, Inexact, Overflow
**DERET**

**Format:** DERET

**Purpose:**
To Return from a debug exception.

**Description:**
DERET returns from Debug Mode and resumes non-debug execution at the instruction whose address is contained in the DEPC register. DERET does not execute the next instruction (i.e., it has no delay slot).

**Restrictions:**
A DERET placed between an LL and SC instruction does not cause the SC to fail.

If the DEPC register with the return address for the DERET was modified by an MTC0 or a DMTC0 instruction, a CP0 hazard exists that must be removed via software insertion of the appropriate number of SSNOP instructions.

The DERET instruction implements a software barrier for all changes in the CP0 state that could affect the fetch and decode of the instruction at the PC to which the DERET returns, such as changes to the effective ASID, user-mode state, and addressing mode.

This instruction is legal only if the processor is executing in Debug Mode. The operation of the processor is **UNDEFINED** if a DERET is executed in the delay slot of a branch or jump instruction.
Operation:

\[
\begin{align*}
\text{DebugDM} & \leftarrow 0 \\
\text{DebugIEXI} & \leftarrow 0 \\
\text{if IsMIPS16Implemented()} & \text{ then} \\
& \quad \text{PC} \leftarrow \text{DEPC}_{31..1} \parallel 0 \\
& \quad \text{ISAMode} \leftarrow 0 \parallel \text{DEPC}_0 \\
\text{else} & \quad \text{PC} \leftarrow \text{DEPC} \\
\text{endif}
\end{align*}
\]

Exceptions:

- Coprocessor Unusable Exception
- Reserved Instruction Exception
### Divide Word

#### Format: \( \text{DIV } rs, rt \)  
MIPS32 (MIPS I)

#### Purpose:
To divide a 32-bit signed integers

#### Description:
\((LO, HI) \leftarrow rs / rt\)

The 32-bit word value in GPR \( rs \) is divided by the 32-bit value in GPR \( rt \), treating both operands as signed values. The 32-bit quotient is placed into special register \( LO \) and the 32-bit remainder is placed into special register \( HI \).

No arithmetic exception occurs under any circumstances.

#### Restrictions:
If the divisor in GPR \( rt \) is zero, the arithmetic result value is **UNPREDICTABLE**.

#### Operation:
\[
q \leftarrow GPR[rs]_{31..0} \text{ div } GPR[rt]_{31..0} \\
LO \leftarrow q \\
r \leftarrow GPR[rs]_{31..0} \text{ mod } GPR[rt]_{31..0} \\
HI \leftarrow r
\]

#### Exceptions:
None
Programming Notes:

No arithmetic exception occurs under any circumstances. If divide-by-zero or overflow conditions are detected and some action taken, then the divide instruction is typically followed by additional instructions to check for a zero divisor and/or for overflow. If the divide is asynchronous then the zero-divisor check can execute in parallel with the divide. The action taken on either divide-by-zero or overflow is either a convention within the program itself, or more typically within the system software; one possibility is to take a BREAK exception with a code field value to signal the problem to the system software.

As an example, the C programming language in a UNIX® environment expects division by zero to either terminate the program or execute a program-specified signal handler. C does not expect overflow to cause any exceptional condition. If the C compiler uses a divide instruction, it also emits code to test for a zero divisor and execute a BREAK instruction to inform the operating system if a zero is detected.

Where the size of the operands are known, software should place the shorter operand in GPR rt. This may reduce the latency of the instruction on those processors which implement data-dependent instruction latencies.

In some processors the integer divide operation may proceed asynchronously and allow other CPU instructions to execute before it is complete. An attempt to read LO or HI before the results are written interlocks until the results are ready. Asynchronous execution does not affect the program result, but offers an opportunity for performance improvement by scheduling the divide so that other instructions can execute in parallel.

Historical Perspective:

In MIPS I through MIPS III, if either of the two instructions preceding the divide is an MFHI or MFLO, the result of the MFHI or MFLO is UNPREDICTABLE. Reads of the HI or LO special register must be separated from subsequent instructions that write to them by two or more instructions. This restriction was removed in MIPS IV and MIPS32 and all subsequent levels of the architecture.
Floating Point Divide

<table>
<thead>
<tr>
<th>COP1</th>
<th>fmt</th>
<th>ft</th>
<th>fs</th>
<th>fd</th>
<th>DIV</th>
</tr>
</thead>
<tbody>
<tr>
<td>010001</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

**Format:**
DIV.S fd, fs, ft
DIV.D fd, fs, ft

**MIPS32 (MIPS I)**
DIV.D fd, fs, ft

**Purpose:**
To divide FP values

**Description:**
fd ← fs / ft

The value in FPR fs is divided by the value in FPR ft. The result is calculated to infinite precision, rounded according to the current rounding mode in FCSR, and placed into FPR fd. The operands and result are values in format fmt.

**Restrictions:**
The fields fs, ft, and fd must specify FPRs valid for operands of type fmt; if they are not valid, the result is UNPREDICTABLE.

The operands must be values in format fmt; if they are not, the result is UNPREDICTABLE and the value of the operand FPRs becomes UNPREDICTABLE.

**Operation:**
StoreFPR (fd, fmt, ValueFPR(fs, fmt) / ValueFPR(ft, fmt))

**Exceptions:**
Coprocessor Unusable, Reserved Instruction

**Floating Point Exceptions:**
Inexact, Invalid Operation, Unimplemented Operation, Division-by-zero, Overflow, Underflow
**Divide Unsigned Word**

**Format:**  DIVU rs, rt  

**MIPS32 (MIPS I)**

**Purpose:**
To divide a 32-bit unsigned integers

**Description:**  (LO, HI) ← rs / rt

The 32-bit word value in GPR rs is divided by the 32-bit value in GPR rt, treating both operands as unsigned values. The 32-bit quotient is placed into special register LO and the 32-bit remainder is placed into special register HI.

No arithmetic exception occurs under any circumstances.

**Restrictions:**
If the divisor in GPR rt is zero, the arithmetic result value is undefined.

**Operation:**

\[
\begin{align*}
q &\leftarrow (0 | | \text{GPR}[rs]_{31..0}) \text{ div } (0 | | \text{GPR}[rt]_{31..0}) \\
r &\leftarrow (0 | | \text{GPR}[rs]_{31..0}) \text{ mod } (0 | | \text{GPR}[rt]_{31..0}) \\
\text{LO} &\leftarrow \text{sign\_extend}(q_{31..0}) \\
\text{HI} &\leftarrow \text{sign\_extend}(r_{31..0})
\end{align*}
\]

**Exceptions:**
None

**Programming Notes:**
See “Programming Notes” for the DIV instruction.

**Historical Perspective:**
In MIPS I through MIPS III, if either of the two instructions preceding the divide is an MFHI or MFLO, the result of the MFHI or MFLO is UNPREDICTABLE. Reads of the HI or LO special register must be separated from subsequent instructions that write to them by two or more instructions. This restriction was removed in MIPS IV and MIPS32 and all subsequent levels of the architecture.
Format: ERET

Purpose:
To return from interrupt, exception, or error trap.

Description:
ERET returns to the interrupted instruction at the completion of interrupt, exception, or error trap processing. ERET does not execute the next instruction (i.e., it has no delay slot).

Restrictions:
The operation of the processor is **UNDEFINED** if an ERET is executed in the delay slot of a branch or jump instruction.

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

ERET implements a software barrier for all changes in the CP0 state that could affect the fetch and decode of the instruction at the PC to which the ERET returns, such as changes to the effective ASID, user-mode state, and addressing mode.

Operation:

```
if Status_{ERL} = 1 then
    temp ← ErrorEPC
    Status_{ERL} ← 0
else
    temp ← EPC
    Status_{EXL} ← 0
endif
if IsMIPS16Implemented() then
    PC ← temp_{31..1} || 0
    ISAMode ← temp_{0}
else
    PC ← temp
endif
LLbit ← 0
```

Exceptions:
- Coprocessor Usable Exception
### Floating Point Floor Convert to Word Fixed Point

**FLOOR.W.fmt**

<table>
<thead>
<tr>
<th>COP1</th>
<th>fmt</th>
<th>0</th>
<th>fs</th>
<th>fd</th>
<th>FLOOR.W</th>
<th>001111</th>
</tr>
</thead>
<tbody>
<tr>
<td>010001</td>
<td>0</td>
<td>00000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:**

- `FLOOR.W.S fd, fs`  
- `FLOOR.W.D fd, fs`

**MIPS32 (MIPS II)**

**Purpose:**
To convert an FP value to 32-bit fixed point, rounding down

**Description:**

\[
fd \leftarrow \text{convert\_and\_round}(fs)
\]

The value in FPR `fs`, in format `fmt`, is converted to a value in 32-bit word fixed point format and rounded toward \(-\infty\) (rounding mode 3). The result is placed in FPR `fd`.

When the source value is Infinity, NaN, or rounds to an integer outside the range \(-2^{31}\) to \(2^{31}-1\), the result cannot be represented correctly, an IEEE Invalid Operation condition exists, and the Invalid Operation flag is set in the `FCSR`. If the Invalid Operation `Enable` bit is set in the `FCSR`, no result is written to `fd` and an Invalid Operation exception is taken immediately. Otherwise, the default result, \(2^{31}-1\), is written to `fd`.

**Restrictions:**

The fields `fs` and `fd` must specify valid FPRs—`fs` for type `fmt` and `fd` for word fixed point—if they are not valid, the result is UNPREDICTABLE.

The operand must be a value in format `fmt`; if it is not, the result is UNPREDICTABLE and the value of the operand FPR becomes UNPREDICTABLE.

**Operation:**

\[
\text{StoreFPR}(fd, \ W, \ \text{ConvertFmt}(\text{ValueFPR}(fs, \ \text{fmt}), \ \text{fmt}, \ \text{W}))
\]

**Exceptions:**

Coprocessor Unusable, Reserved Instruction

**Floating Point Exceptions:**

Invalid Operation, Unimplemented Operation, Inexact, Overflow
Jump

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>J</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000010</td>
<td>instr_index</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>26</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** J target

**MIPS32 (MIPS I)**

**Purpose:**
To branch within the current 256 MB-aligned region

**Description:**
This is a PC-region branch (not PC-relative); the effective target address is in the “current” 256 MB-aligned region. The low 28 bits of the target address is the instr_index field shifted left 2 bits. The remaining upper bits are the corresponding bits of the address of the instruction in the delay slot (not the branch itself).

Jump to the effective target address. Execute the instruction that follows the jump, in the branch delay slot, before executing the jump itself.

**Restrictions:**
Processor operation is **UNPREDICTABLE** if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

**Operation:**

\[
\begin{align*}
I & : \\
I+1 : & PC \leftarrow PC_{GPRLEN..28} || instr_{index} || 0^2
\end{align*}
\]

**Exceptions:**
None

**Programming Notes:**
Forming the branch target address by catenating PC and index bits rather than adding a signed offset to the PC is an advantage if all program code addresses fit into a 256 MB region aligned on a 256 MB boundary. It allows a branch from anywhere in the region to anywhere in the region, an action not allowed by a signed relative offset.

This definition creates the following boundary case: When the jump instruction is in the last word of a 256 MB region, it can branch only to the following 256 MB region containing the branch delay slot.
**Jump and Link**

**Format:** JAL target

**Purpose:**
To execute a procedure call within the current 256 MB-aligned region

**Description:**
Place the return address link in GPR 31. The return link is the address of the second instruction following the branch, at which location execution continues after a procedure call.

This is a PC-region branch (not PC-relative); the effective target address is in the “current” 256 MB-aligned region. The low 28 bits of the target address is the instr_index field shifted left 2 bits. The remaining upper bits are the corresponding bits of the address of the instruction in the delay slot (not the branch itself).

Jump to the effective target address. Execute the instruction that follows the jump, in the branch delay slot, before executing the jump itself.

**Restrictions:**
Processor operation is **UNPREDICTABLE** if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

**Operation:**

\[
\begin{align*}
I &: \text{GPR}[31] &\leftarrow & \text{PC} + 8 \\
I+1 &: \text{PC} &\leftarrow & \text{PC}_{\text{GPRLN..28}} || \text{instr\_index} || 0^2
\end{align*}
\]

**Exceptions:**
None

**Programming Notes:**
Forming the branch target address by catenating PC and index bits rather than adding a signed offset to the PC is an advantage if all program code addresses fit into a 256 MB region aligned on a 256 MB boundary. It allows a branch from anywhere in the region to anywhere in the region, an action not allowed by a signed relative offset.

This definition creates the following boundary case: When the branch instruction is in the last word of a 256 MB region, it can branch only to the following 256 MB region containing the branch delay slot.
Jump and Link Register

### Format:

JALR rs (rd = 31 implied)

JALR rd, rs

### MIPS32 (MIPS I)

**Purpose:**

To execute a procedure call to an instruction address in a register

**Description:**

rd ← return_addr, PC ← rs

Place the return address link in GPR rd. The return link is the address of the second instruction following the branch, where execution continues after a procedure call.

*For processors that do not implement the MIPS16 ASE:*

- Jump to the effective target address in GPR rs. Execute the instruction that follows the jump, in the branch delay slot, before executing the jump itself.

*For processors that do implement the MIPS16 ASE:*

- Jump to the effective target address in GPR rs. Set the *ISA Mode* bit to the value in GPR rs bit 0. Bit 0 of the target address is always zero so that no Address Exceptions occur when bit 0 of the source register is one

At this time the only defined hint field value is 0, which sets default handling of JALR. Future versions of the architecture may define additional hint values.

### Restrictions:

Register specifiers rs and rd must not be equal, because such an instruction does not have the same effect when reexecuted. The result of executing such an instruction is undefined. This restriction permits an exception handler to resume execution by reexecuting the branch when an exception occurs in the branch delay slot.

The effective target address in GPR rs must be naturally-aligned. For processors that do not implement the MIPS16 ASE, if either of the two least-significant bits are not zero, an Address Error exception occurs when the branch target is subsequently fetched as an instruction. For processors that do implement the MIPS16 ASE, if bit 0 is zero and bit 1 is one, an Address Error exception occurs when the jump target is subsequently fetched as an instruction.

Processor operation is **UNPREDICTABLE** if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.
Operation:

I: \[ \text{temp} \leftarrow \text{GPR}[rs] \]
\[ \text{GPR}[rd] \leftarrow \text{PC} + 8 \]

I+1: if Config\textsubscript{CA} = 0 then
  \[ \text{PC} \leftarrow \text{temp} \]
else
  \[ \text{PC} \leftarrow \text{temp}_{\text{GPRLEN}-1..1} \mid 0 \]
  ISAMode \leftarrow temp\textsubscript{0}
endif

Exceptions:
None

Programming Notes:
This is the only branch-and-link instruction that can select a register for the return link; all other link instructions use GPR 31. The default register for GPR \( rd \), if omitted in the assembly language instruction, is GPR 31.
Jump Register

<table>
<thead>
<tr>
<th>Format:</th>
<th>JR rs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose:</td>
<td>To execute a branch to an instruction address in a register</td>
</tr>
<tr>
<td>Description:</td>
<td>PC ← rs</td>
</tr>
<tr>
<td></td>
<td>Jump to the effective target address in GPR rs. Execute the instruction following the jump, in the branch delay slot, before jumping.</td>
</tr>
<tr>
<td>Restrictions:</td>
<td>The effective target address in GPR rs must be naturally-aligned. For processors that do not implement the MIPS16 ASE, if either of the two least-significant bits are not zero, an Address Error exception occurs when the branch target is subsequently fetched as an instruction. For processors that do implement the MIPS16 ASE, if bit 0 is zero and bit 1 is one, an Address Error exception occurs when the jump target is subsequently fetched as an instruction.</td>
</tr>
<tr>
<td>Exceptions:</td>
<td>None</td>
</tr>
</tbody>
</table>

![Jump Register](image)
Programming Notes:
Software should use the value 31 for the rs field of the instruction word on return from a JAL, JALR, or BGEZAL, and should use a value other than 31 for remaining uses of JR.
Load Byte

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>LB</td>
<td>10000</td>
<td>base</td>
<td>rt</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** \( \text{LB } rt, \text{ offset(base)} \)

**MIPS32 (MIPS I)**

**Purpose:**
To load a byte from memory as a signed value

**Description:** \( rt \leftarrow \text{memory[base+offset]} \)

The contents of the 8-bit byte at the memory location specified by the effective address are fetched, sign-extended, and placed in GPR \( rt \). The 16-bit signed \( \text{offset} \) is added to the contents of GPR \( \text{base} \) to form the effective address.

**Restrictions:**
None

**Operation:**
\[
\begin{align*}
\text{vAddr} & \leftarrow \text{sign_extend(} \text{offset}) + \text{GPR[base]} \\
\text{pAddr} & \leftarrow \text{pAddrPSIZE-1..2 || (pAddr}
\text{1..0 xor ReverseEndian}\text{)2} \\
\text{memword} & \leftarrow \text{LoadMemory (CCA, BYTE, pAddr, vAddr, DATA)} \\
\text{byte} & \leftarrow \text{vAddr}
\text{1..0 xor BigEndianCPU}\text{2} \\
\text{GPR[rt]} & \leftarrow \text{sign_extend(} \text{memword}7..8*byte\text{..8*byte})
\end{align*}
\]

**Exceptions:**
TLB Refill, TLB Invalid, Address Error
Load Byte Unsigned

Format: \( \text{LBU } rt, \text{offset}(\text{base}) \)

Purpose:
To load a byte from memory as an unsigned value

Description: \( rt \leftarrow \text{memory}[\text{base}+\text{offset}] \)
The contents of the 8-bit byte at the memory location specified by the effective address are fetched, zero-extended, and placed in GPR \( rt \). The 16-bit signed \( \text{offset} \) is added to the contents of GPR \( \text{base} \) to form the effective address.

Restrictions:
None

Operation:
\[
\begin{align*}
\text{vAddr} & \leftarrow \text{sign\_extend}(\text{offset}) + \text{GPR[base]} \\
(\text{pAddr, CCA}) & \leftarrow \text{AddressTranslation (vAddr, DATA, LOAD)} \\
\text{pAddr} & \leftarrow \text{pAddr}_{\text{PSIZE}-1..2} \mid (\text{pAddr}_{1..0} \text{xor ReverseEndian}^2) \\
\text{memword} & \leftarrow \text{LoadMemory (CCA, BYTE, pAddr, vAddr, DATA)} \\
\text{byte} & \leftarrow \text{vAddr}_{1..0} \text{xor BigEndianCPU}^2 \\
\text{GPR[rt]} & \leftarrow \text{zero\_extend} (\text{memword}_{7+8*\text{byte}..8*\text{byte}})
\end{align*}
\]

Exceptions:
TLB Refill, TLB Invalid, Address Error

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>LBU</td>
<td>base</td>
<td>rt</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100100</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Load Doubleword to Floating Point

LDC1

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDC1</td>
<td>base</td>
<td>ft</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>110101</td>
<td>5</td>
<td>5</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Format: LDC1 ft, offset(base)

Purpose:
To load a doubleword from memory to an FPR

Description: ft ← memory[base+offset]

The contents of the 64-bit doubleword at the memory location specified by the aligned effective address are fetched and placed in FPR ft. The 16-bit signed offset is added to the contents of GPR base to form the effective address.

Restrictions:
An Address Error exception occurs if EffectiveAddress₂..₀ ≠ 0 (not doubleword-aligned).

Operation:
```plaintext
vAddr ← sign_extend(offset) + GPR[base]
if vAddr₂..₀ ≠ 0³ then
    SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
memdoubleword ← LoadMemory(CCA, DOUBLEWORD, pAddr, vAddr, DATA)
StoreFPR(ft, UNINTERPRETED_DOUBLEWORD, memdoubleword)
```

Exceptions:
Coprocessor Unusable, Reserved Instruction, TLB Refill, TLB Invalid, Address Error
### Load Doubleword to Coprocessor 2

#### Format:
LDC2 rt, offset(base)

#### Purpose:
To load a doubleword from memory to a Coprocessor 2 register

#### Description:
rt ← memory[base+offset]

The contents of the 64-bit doubleword at the memory location specified by the aligned effective address are fetched and placed in Coprocessor 2 register rt. The 16-bit signed offset is added to the contents of GPR base to form the effective address.

#### Restrictions:
An Address Error exception occurs if EffectiveAddress2..0 ≠ 0 (not doubleword-aligned).

#### Operation:

\[
\text{vAddr} \leftarrow \text{sign}_{\text{extend}}(\text{offset}) + \text{GPR[base]}
\]

if vAddr2..0 ≠ 0 \(3^3\) then SignalException(AddressError) endif

(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)

memdoubleword ← LoadMemory(CCA, DOUBLEWORD, pAddr, vAddr, DATA)

CPR[2,rt,0] ← memdoubleword

#### Exceptions:
Coprocessor Unusable, Reserved Instruction, TLB Refill, TLB Invalid, Address Error
## Load Halfword

<table>
<thead>
<tr>
<th>Format:</th>
<th>LH rt, offset(base)</th>
<th>MIPS32 (MIPS I)</th>
</tr>
</thead>
</table>

**Purpose:**
To load a halfword from memory as a signed value

**Description:**
rt ← memory[base+offset]
The contents of the 16-bit halfword at the memory location specified by the aligned effective address are fetched, sign-extended, and placed in GPR rt. The 16-bit signed offset is added to the contents of GPR base to form the effective address.

**Restrictions:**
The effective address must be naturally-aligned. If the least-significant bit of the address is non-zero, an Address Error exception occurs.

**Operation:**

```
vAddr ← sign_extend(offset) + GPR[base]
if vAddr₀ ≠ 0 then
    SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← pAddr_{PSIZE-1..2} || (pAddr_{1..0} xor (ReverseEndian || 0))
memword ← LoadMemory (CCA, HALFWORD, pAddr, vAddr, DATA)
byte ← vAddr_{1..0} xor (BigEndianCPU || 0)
GPR[rt] ← sign_extend(memword_{15+8*byte..8*byte})
```

**Exceptions:**
TLB Refill, TLB Invalid, Bus Error, Address Error
Load Halfword Unsigned

<table>
<thead>
<tr>
<th>LHU</th>
<th>base</th>
<th>rt</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>1001011</td>
<td>6</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

**Format:** LHU rt, offset(base)  
MIPS32 (MIPS I)

**Purpose:**
To load a halfword from memory as an unsigned value

**Description:** rt ← memory[base+offset]

The contents of the 16-bit halfword at the memory location specified by the aligned effective address are fetched, zero-extended, and placed in GPR rt. The 16-bit signed offset is added to the contents of GPR base to form the effective address.

**Restrictions:**
The effective address must be naturally-aligned. If the least-significant bit of the address is non-zero, an Address Error exception occurs.

**Operation:**

\[
vAddr \leftarrow \text{sign\_extend}(\text{offset}) + \text{GPR}\{\text{base}\}
\]

if \( vAddr_0 \neq 0 \) then

\[\text{SignalException(AddressError)}\]
endif

\((\text{pAddr}, \text{CCA}) \leftarrow \text{AddressTranslation} (vAddr, \text{DATA, LOAD})\)

\(\text{pAddr} \leftarrow \text{pAddr}_{\text{PSIZE}-1..2} \mid\mid (\text{pAddr}_{1..0} \oplus (\text{ReverseEndian} \mid\mid 0))\)

\(\text{memword} \leftarrow \text{LoadMemory} (\text{CCA, HALFWORD, pAddr, vAddr, DATA})\)

\(\text{byte} \leftarrow vAddr_{1..0} \oplus (\text{BigEndianCPU} \mid\mid 0)\)

\(\text{GPR}\{\text{rt}\} \leftarrow \text{zero\_extend}(\text{memword}_{15+8*\text{byte}..8*\text{byte}})\)

**Exceptions:**

TLB Refill, TLB Invalid, Address Error
Load Linked Word

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>LL</td>
<td>base</td>
<td>rt</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>110000</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:**  LL rt, offset(base)

**Purpose:**
To load a word from memory for an atomic read-modify-write

**Description:**  rt ← memory[base+offset]

The LL and SC instructions provide the primitives to implement atomic read-modify-write (RMW) operations for cached memory locations.

The 16-bit signed offset is added to the contents of GPR base to form an effective address. The contents of the 32-bit word at the memory location specified by the aligned effective address are fetched, sign-extended to the GPR register length if necessary, and written into GPR rt.

This begins a RMW sequence on the current processor. There can be only one active RMW sequence per processor.

When an LL is executed it starts an active RMW sequence replacing any other sequence that was active.

The RMW sequence is completed by a subsequent SC instruction that either completes the RMW sequence atomically and succeeds, or does not and fails.

Executing LL on one processor does not cause an action that, by itself, causes an SC for the same block to fail on another processor.

An execution of LL does not have to be followed by execution of SC; a program is free to abandon the RMW sequence without attempting a write.

**Restrictions:**
The addressed location must be cached; if it is not, the result is undefined.

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

**Operation:**

\[
\text{vAddr} \leftarrow \text{sign}_\text{extend}(\text{offset}) + \text{GPR}[\text{base}]
\]

if \( \text{vAddr}_{1\ldots0} \neq 0^2 \) then
  SignalException(AddressError)
endif

(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)

memword ← LoadMemory (CCA, WORD, pAddr, vAddr, DATA)

GPR[rt] ← memword

LLbit ← 1

MIPS32™ Architecture For Programmers Volume II, Revision 0.95 127
Exceptions:
TLB Refill, TLB Invalid, Address Error, Reserved Instruction

Programming Notes:
There is no Load Linked Word Unsigned operation corresponding to Load Word Unsigned.
## Load Upper Immediate

**Format:** LUI rt, immediate

**Purpose:**
To load a constant into the upper half of a word

**Description:**
\( rt \leftarrow \text{immediate} \mid \mid 0^{16} \)

The 16-bit `immediate` is shifted left 16 bits and concatenated with 16 bits of low-order zeros. The 32-bit result is placed into GPR `rt`.

**Restrictions:**
None

**Operation:**
\[ \text{GPR}[rt] \leftarrow \text{immediate} \mid \mid 0^{16} \]

**Exceptions:**
None

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>LUI</td>
<td>0</td>
<td>0</td>
<td>rt</td>
<td></td>
<td>immediate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>001111</td>
<td>00000</td>
<td>5</td>
<td></td>
<td>16</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Load Word**

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>LW</td>
<td>base</td>
<td>rt</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** $\text{LW} \ rt, \ \text{offset}(\text{base})$

**MIPS32 (MIPS I)**

**Purpose:**
To load a word from memory as a signed value

**Description:** $rt \leftarrow \text{memory}[\text{base} + \text{offset}]$

The contents of the 32-bit word at the memory location specified by the aligned effective address are fetched, sign-extended to the GPR register length if necessary, and placed in GPR $rt$. The 16-bit signed offset is added to the contents of GPR base to form the effective address.

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

**Operation:**

\[
\begin{align*}
\text{vAddr} & \leftarrow \text{sign\_extend}(\text{offset}) + \text{GPR}[\text{base}] \\
\text{if } \text{vAddr}_{1..0} & \neq 0^2 \text{ then} \\
\quad & \text{SignalException(AddressError)} \\
\text{endif} \\
(p\text{Addr}, CCA) & \leftarrow \text{AddressTranslation (vAddr, DATA, LOAD)} \\
\text{memword} & \leftarrow \text{LoadMemory (CCA, WORD, p\text{Addr}, v\text{Addr}, DATA)} \\
\text{GPR}[rt] & \leftarrow \text{memword}
\end{align*}
\]

**Exceptions:**
TLB Refill, TLB Invalid, Bus Error, Address Error
Load Word to Floating Point

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>LWC1</td>
<td>110001</td>
<td>base</td>
<td>rt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>5</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>16</td>
</tr>
</tbody>
</table>

**Format:** LWC1 ft, offset(base)

**MIPS32 (MIPS I)**

**Purpose:**
To load a word from memory to an FPR

**Description:**
\[
ft \leftarrow \text{memory}\left[\text{base} + \text{offset}\right]
\]
The contents of the 32-bit word at the memory location specified by the aligned effective address are fetched and placed into the low word of coprocessor 1 general register \( ft \). The 16-bit signed offset is added to the contents of GPR base to form the effective address.

**Restrictions:**
An Address Error exception occurs if EffectiveAddress\(_{1,0}\) \( \neq 0 \) (not word-aligned).

**Operation:**
```c
/* mem is aligned 64 bits from memory. Pick out correct bytes. */
vAddr \leftarrow \text{sign\_extend}(\text{offset}) + \text{GPR}[\text{base}]
if vAddr\(_{1,0}\) \( \neq 0 \) then
    \text{SignalException(AddressError)}
endif
(pAddr, CCA) \leftarrow \text{AddressTranslation (vAddr, DATA, LOAD)}
memword \leftarrow \text{LoadMemory(CCA, WORD, pAddr, vAddr, DATA)}
StoreFPR(ft, UNINTERPRETED\_WORD, memword)
```

**Exceptions:**
TLB Refill, TLB Invalid, Address Error, Reserved Instruction, Coprocessor Unusable
Load Word to Coprocessor 2

**Format:** LWC2 rt, offset(base)  

**MIPS32 (MIPS I)**

**Purpose:**  
To load a word from memory to a COP2 register

**Description:**  
rt ← memory[base+offset]  
The contents of the 32-bit word at the memory location specified by the aligned effective address are fetched and placed into the low word of COP2 (Coprocessor 2) general register rt. The 16-bit signed offset is added to the contents of GPR base to form the effective address.

**Restrictions:**  
An Address Error exception occurs if EffectiveAddress1..0 ≠ 0 (not word-aligned).

**Operation:**

\[
\text{vAddr} \leftarrow \text{sign}_\text{extend}(\text{offset}) + \text{GPR[base]}
\]

\[
\text{if } \text{vAddr}_{12..0} \neq 0^2 \text{ then}
\]

\[
\text{SignalException(AddressError)}
\]

\[
\text{endif}
\]

\[
(p\text{Addr}, \text{CCA}) \leftarrow \text{AddressTranslation (vAddr, DATA, LOAD)}
\]

\[
\text{memword} \leftarrow \text{LoadMemory(CCA, DOUBLEWORD, p\text{Addr}, v\text{Addr}, DATA)}
\]

\[
\text{CPR}[2,\text{rt},0] \leftarrow \text{memword}
\]

**Exceptions:**

TLB Refill, TLB Invalid, Address Error, Reserved Instruction, Coprocessor Unusable
Load Word Left

**Format:** \( \text{LWL rt, offset(base)} \)  

**MIPS32 (MIPS I)**

**Purpose:**
To load the most-significant part of a word as a signed value from an unaligned memory address.

**Description:**
\[ \text{rt} \leftarrow \text{rt MERGE memory[base+offset]} \]

The 16-bit signed `offset` is added to the contents of GPR `base` to form an effective address (`EffAddr`). `EffAddr` is the address of the most-significant of 4 consecutive bytes forming a word (`W`) in memory starting at an arbitrary byte boundary. The most-significant 1 to 4 bytes of `W` is in the aligned word containing the `EffAddr`. This part of `W` is loaded into the most-significant (left) part of the word in GPR `rt`. The remaining least-significant part of the word in GPR `rt` is unchanged.

The figure below illustrates this operation using big-endian byte ordering for 32-bit and 64-bit registers. The 4 consecutive bytes in 2..5 form an unaligned word starting at location 2. A part of `W`, 2 bytes, is in the aligned word containing the most-significant byte at 2. First, LWL loads these 2 bytes into the left part of the destination register word and leaves the right part of the destination word unchanged. Next, the complementary LWR loads the remainder of the unaligned word.

![Figure 3-2 Unaligned Word Load Using LWL and LWR](image-url)
The bytes loaded from memory to the destination register depend on both the offset of the effective address within an aligned word, that is, the low 2 bits of the address (\(v\text{Addr}_{1..0}\)), and the current byte-ordering mode of the processor (big- or little-endian). The figure below shows the bytes loaded for every combination of offset and byte ordering.

**Figure 3-3 Bytes Loaded by LWL Instruction**

<table>
<thead>
<tr>
<th>Memory contents and byte offsets</th>
<th>Initial contents of Dest Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 ← big-endian</td>
<td>e f g h</td>
</tr>
<tr>
<td>3 2 1 0 ← little-endian</td>
<td>most least</td>
</tr>
<tr>
<td>most least</td>
<td>— significance —</td>
</tr>
<tr>
<td>Destination register contents after instruction (shaded is unchanged)</td>
<td></td>
</tr>
<tr>
<td>Big-endian (v\text{Addr}_{1..0})</td>
<td>Little-endian</td>
</tr>
<tr>
<td>I J K L</td>
<td>L f g h</td>
</tr>
<tr>
<td>J K L h</td>
<td>K L g h</td>
</tr>
<tr>
<td>K L g h</td>
<td>J K L h</td>
</tr>
<tr>
<td>L f g h</td>
<td>I J K L</td>
</tr>
</tbody>
</table>
Load Word Left (con’t)

**Restrictions:**

None

**Operation:**

\[
\begin{align*}
\text{vAddr} & \leftarrow \text{sign\_extend}(\text{offset}) + \text{GPR}[\text{base}] \\
(p\text{Addr}, \text{CCA}) & \leftarrow \text{AddressTranslation}(\text{vAddr}, \text{DATA}, \text{LOAD}) \\
p\text{Addr} & \leftarrow p\text{Addr}_{\text{PSIZE}-1..2} \ || (p\text{Addr}_{1..0} \text{ xor ReverseEndian}^2) \\
\text{if BigEndianMem} = 0 \text{ then} & \\
\quad p\text{Addr} & \leftarrow p\text{Addr}_{\text{PSIZE}-1..2} \ || 0^2 \\
\text{endif} \\
\text{byte} & \leftarrow \text{vAddr}_{1..0} \text{ xor BigEndianCPU}^2 \\
\text{memword} & \leftarrow \text{LoadMemory}(\text{CCA}, \text{byte}, p\text{Addr}, \text{vAddr}, \text{DATA}) \\
\text{temp} & \leftarrow \text{memword}_{7+8*\text{byte}..0} \ || \ \text{GPR}[\text{rt}]_{23-8*\text{byte}..0} \\
\text{GPR}[\text{rt}] & \leftarrow \text{temp}
\end{align*}
\]

**Exceptions:**

None

TLB Refill, TLB Invalid, Bus Error, Address Error

**Programming Notes:**

The architecture provides no direct support for treating unaligned words as unsigned values, that is, zeroing bits 63..32 of the destination register when bit 31 is loaded.

**Historical Information**

In the MIPS I architecture, the LWL and LWR instructions were exceptions to the load-delay scheduling restriction. A LWL or LWR instruction which was immediately followed by another LWL or LWR instruction, and used the same destination register would correctly merge the 1 to 4 loaded bytes with the data loaded by the previous instruction. All such restrictions were removed from the architecture in MIPS II.
Load Word Right

<table>
<thead>
<tr>
<th>LWR</th>
<th>base</th>
<th>rt</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>100110</td>
<td>5</td>
<td>5</td>
<td>16</td>
</tr>
</tbody>
</table>

**Format:**  \( \text{LWR } rt, \text{ offset}(\text{base}) \)  

**MIPS32 (MIPS I)**

**Purpose:**
To load the least-significant part of a word from an unaligned memory address as a signed value

**Description:**  
\( rt \leftarrow \text{rt MERGE memory[base+offset]} \)

The 16-bit signed \( \text{offset} \) is added to the contents of GPR \( \text{base} \) to form an effective address (\( \text{EffAddr} \)). \( \text{EffAddr} \) is the address of the least-significant of 4 consecutive bytes forming a word (\( W \)) in memory starting at an arbitrary byte boundary.

A part of \( W \), the least-significant 1 to 4 bytes, is in the aligned word containing \( \text{EffAddr} \). This part of \( W \) is loaded into the least-significant (right) part of the word in GPR \( rt \). The remaining most-significant part of the word in GPR \( rt \) is unchanged.

Executing both LWR and LWL, in either order, delivers a sign-extended word value in the destination register.

The figure below illustrates this operation using big-endian byte ordering for 32-bit and 64-bit registers. The 4 consecutive bytes in 2..5 form an unaligned word starting at location 2. A part of \( W \), 2 bytes, is in the aligned word containing the least-significant byte at 5. First, LWR loads these 2 bytes into the right part of the destination register. Next, the complementary LWL loads the remainder of the unaligned word.
The bytes loaded from memory to the destination register depend on both the offset of the effective address within an aligned word, that is, the low 2 bits of the address (\(\text{vAddr}_{1,0}\)), and the current byte-ordering mode of the processor (big- or little-endian). The figure below shows the bytes loaded for every combination of offset and byte ordering.
# Figure 3-5 Bytes Loaded by LWL Instruction

<table>
<thead>
<tr>
<th>Memory contents and byte offsets</th>
<th>Initial contents of Dest Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 ←big-endian</td>
<td>e f g h</td>
</tr>
<tr>
<td>I J K L</td>
<td>offset (vAddr₁,₀)</td>
</tr>
<tr>
<td>3 2 1 0 ←little-endian</td>
<td>most least</td>
</tr>
<tr>
<td>most</td>
<td>significance—</td>
</tr>
<tr>
<td>least</td>
<td></td>
</tr>
<tr>
<td>— significance —</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Destination register contents after instruction (shaded is unchanged)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Big-endian                                      Little-endian</td>
</tr>
<tr>
<td>e f g I                                           I J K L</td>
</tr>
<tr>
<td>e f I J                                          0</td>
</tr>
<tr>
<td>e I J K                                          1</td>
</tr>
<tr>
<td>I J K L                                          2</td>
</tr>
<tr>
<td>3                                                 e f g I</td>
</tr>
<tr>
<td>3</td>
</tr>
</tbody>
</table>
Restrictions:
None

Operation:

\[
\begin{align*}
vAddr & \leftarrow \text{sign\_extend}(\text{offset}) + \text{GPR}[\text{base}] \\
(p\Addr, \text{CCA}) & \leftarrow \text{AddressTranslation} (vAddr, \text{DATA, LOAD}) \\
p\Addr & \leftarrow p\Addr_{\text{PSIZE}\ldots1.2} \mid\mid (p\Addr_{1\ldots0} \text{ xor ReverseEndian}^2) \\
\text{if } \text{BigEndianMem} = 0 \text{ then} \\
& \quad p\Addr \leftarrow p\Addr_{\text{PSIZE}\ldots1.2} \mid\mid 0^2 \\
\text{endif} \\
\text{byte} & \leftarrow v\Addr_{1\ldots0} \text{ xor BigEndianCPU}^2 \\
\text{memword} & \leftarrow \text{LoadMemory} (\text{CCA, byte, } p\Addr, \text{vAddr, DATA}) \\
\text{temp} & \leftarrow \text{memword}_{31\ldots32\cdot\text{byte}} \mid\mid \text{GPR}[\text{rt}]_{31\ldots8\cdot\text{byte}}.0 \\
\text{GPR}[\text{rt}] & \leftarrow \text{temp}
\end{align*}
\]

Exceptions:
TLB Refill, TLB Invalid, Bus Error, Address Error

Programming Notes:
The architecture provides no direct support for treating unaligned words as unsigned values, that is, zeroing bits 63..32 of the destination register when bit 31 is loaded.

Historical Information
In the MIPS I architecture, the LWL and LWR instructions were exceptions to the load-delay scheduling restriction. A LWL or LWR instruction which was immediately followed by another LWL or LWR instruction, and used the same destination register would correctly merge the 1 to 4 loaded bytes with the data loaded by the previous instruction. All such restrictions were removed from the architecture in MIPS II.
### Multiply and Add Word to Hi,Lo

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>011100</td>
<td>rs</td>
<td>rt</td>
<td>0</td>
<td>0</td>
<td>MADD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** \texttt{MADD \ rs, \ rt} \hspace{1cm} \text{MIPS32}

**Purpose:**
To multiply two words and add the result to Hi, Lo

**Description:** \((LO, HI) \leftarrow (rs \times rt) + (LO, HI)\)
The 32-bit word value in GPR \(rs\) is multiplied by the 32-bit word value in GPR \(rt\), treating both operands as signed values, to produce a 64-bit result. The product is added to the 64-bit concatenated values of \(HI\) and \(LO\). The most significant 32 bits of the result are written into \(HI\) and the least significant 32 bits are written into \(LO\). No arithmetic exception occurs under any circumstances.

**Restrictions:**
None

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

**Operation:**
\[
\begin{align*}
temp & \leftarrow (HI \ || \ LO) + (GPR[rs] \times GPR[rt]) \\
HI & \leftarrow temp_{63..32} \\
LO & \leftarrow temp_{31..0}
\end{align*}
\]

**Exceptions:**
None

**Programming Notes:**
Where the size of the operands are known, software should place the shorter operand in GPR \(rt\). This may reduce the latency of the instruction on those processors which implement data-dependent instruction latencies.
Multiply and Add Unsigned Word to Hi,Lo

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL2</td>
<td>rs</td>
<td>rt</td>
<td>0</td>
<td>0</td>
<td>MADDU</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011100</td>
<td>00000</td>
<td>00000</td>
<td>00001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** MADDU rs, rt

**Purpose:**
To multiply two unsigned words and add the result to Hi, Lo.

**Description:**
\[(LO, HI) \leftarrow (rs \times rt) + (LO, HI)\]
The 32-bit word value in GPR rs is multiplied by the 32-bit word value in GPR rt, treating both operands as unsigned values, to produce a 64-bit result. The product is added to the 64-bit concatenated values of HI and LO. The most significant 32 bits of the result are written into HI and the least significant 32 bits are written into LO. No arithmetic exception occurs under any circumstances.

**Restrictions:**
None
This instruction does not provide the capability of writing directly to a target GPR.

**Operation:**
\[
temp \leftarrow (HI || LO) + (GPR[rs] \times GPR[rt]) \\
HI \leftarrow temp_{63...32} \\
LO \leftarrow temp_{31...0}
\]

**Exceptions:**
None

**Programming Notes:**
Where the size of the operands are known, software should place the shorter operand in GPR rt. This may reduce the latency of the instruction on those processors which implement data-dependent instruction latencies.
Move from Coprocessor 0

**Format:** MFC0 rt, rd

**Purpose:**
To move the contents of a coprocessor 0 register to a general register.

**Description:** rt ← CPR[0, rd, sel]

The contents of the coprocessor 0 register specified by the combination of rd and sel are loaded into general register rt. Note that not all coprocessor 0 registers support the sel field. In those instances, the sel field must be zero.

**Restrictions:**
The results are **UNDEFINED** if coprocessor 0 does not contain a register as specified by rd and sel.

**Operation:**

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

**Exceptions:**

- Coprocessor Unusable
- Reserved Instruction
Move Word From Floating Point  

**Format:**  
MFC1 rt, fs  

**Purpose:**  
To copy a word from an FPU (CP1) general register to a GPR  

**Description:** rt ← fs  
The contents of FPR fs are loaded into general register rt.  

**Restrictions:**  

**Operation:**  
data ← ValueFPR(fs, UNINTERPRETED_WORD)  
GPR[rt] ← data  

**Exceptions:**  
Coprocessor Unusable, Reserved Instruction  

**Historical Information:**  
For MIPS I, MIPS II, and MIPS III the contents of GPR rt are undefined for the instruction immediately following MFC1.
Move Word From Coprocessor 2

**Format:**

<table>
<thead>
<tr>
<th>COP2</th>
<th>MF</th>
<th>rt</th>
<th>rd</th>
<th>0</th>
<th>sel</th>
</tr>
</thead>
<tbody>
<tr>
<td>010010</td>
<td>00000</td>
<td>5</td>
<td>5</td>
<td>8</td>
<td>3</td>
</tr>
</tbody>
</table>

MFC2 rt, rd

MFC2, rt, rd, sel

**Purpose:**

To copy a word from a COP2 general register to a GPR

**Description:** rt ← rd

The contents of GPR rt are and placed into the coprocessor 2 register specified by the rd and sel fields. Note that not all coprocessor 2 registers may support the sel field. In those instances, the sel field must be zero.

**Restrictions:**

The results are **UNPREDICTABLE** is coprocessor 2 does not contain a register as specified by rd and sel.

**Operation:**

```
data ← CPR[2, rd, sel]
GPR[rt] ← data
```

**Exceptions:**

Coprocessor Unusable
Move From HI Register

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>0</td>
<td>000000 0000</td>
<td>rd</td>
<td>0</td>
<td>00000</td>
<td>MFHI</td>
<td>010000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>10</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** \( \text{MFHI} \ rd \)

**Purpose:**
To copy the special purpose \( HI \) register to a GPR

**Description:** \( rd \leftarrow HI \)
The contents of special register \( HI \) are loaded into GPR \( rd \).

**Restrictions:**
None

**Operation:**
\[ \text{GPR}[rd] \leftarrow HI \]

**Exceptions:**
None

**Historical Information:**
In the MIPS I, II, and III architectures, the two instructions which follow the MFHI must not modify the HI register. If this restriction is violated, the result of the MFHI is \text{UNPREDICTABLE}. This restriction was removed in MIPS IV and MIPS32, and all subsequent levels of the architecture.
**Move From LO Register**

**Format:**

\[
\text{MFLO} \quad \text{rd}
\]

**Purpose:**

To copy the special purpose \textit{LO} register to a GPR

**Description:** \( \text{rd} \leftarrow \text{LO} \)

The contents of special register \textit{LO} are loaded into GPR \textit{rd}.

**Restrictions:** None

**Operation:**

\[
\text{GPR}[\text{rd}] \leftarrow \text{LO}
\]

**Exceptions:**

None

**Historical Information:**

In the MIPS I, II, and III architectures, the two instructions which follow the MFHI must not modify the HI register. If this restriction is violated, the result of the MFHI is **UNPREDICTABLE**. This restriction was removed in MIPS IV and MIPS32, and all subsequent levels of the architecture.
Floating Point Move

<table>
<thead>
<tr>
<th>COP1</th>
<th>fmt</th>
<th>0</th>
<th>fs</th>
<th>fd</th>
<th>MOV</th>
</tr>
</thead>
<tbody>
<tr>
<td>010001</td>
<td>00000</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>000110</td>
</tr>
</tbody>
</table>

**Format:** MOV.S fd, fs  
`MIPS32` (MIPS I)

MOV.D fd, fs  
`MIPS32` (MIPS I)

**Purpose:**
To move an FP value between FPRs

**Description:** fd ← fs

The value in FPR fs is placed into FPR fd. The source and destination are values in format fmt.

The move is non-arithmetic; it causes no IEEE 754 exceptions.

**Restrictions:**
The fields fs and fd must specify FPRs valid for operands of type fmt; if they are not valid, the result is UNPREDICTABLE.

The operand must be a value in format fmt; if it is not, the result is UNPREDICTABLE and the value of the operand FPR becomes UNPREDICTABLE.

**Operation:**

\[
\text{StoreFPR}(fd, fmt, \text{ValueFPR}(fs, fmt))
\]

**Exceptions:**
Coprocessor Unusable, Reserved Instruction

**Floating Point Exceptions:**
Unimplemented Operation
Move Conditional on Floating Point False

MOVF

Format:  MOVF rd, rs, cc

MIPS32 (MIPS IV)

Purpose:
To test an FP condition code then conditionally move a GPR

Description: if cc = 0 then rd ← rs
If the floating point condition code specified by CC is zero, then the contents of GPR rs are placed into GPR rd.

Restrictions:

Operation:
if FPConditionCode(cc) = 0 then
    GPR[rd] ← GPR[rs]
endif

Exceptions:
Reserved Instruction, Coprocessor Unusable
Floating Point Move Conditional on Floating Point False

**MOVF.fmt**

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>COP1</td>
<td>fmt</td>
<td>cc</td>
<td>0</td>
<td>tf</td>
<td>0</td>
<td>0</td>
<td>fs</td>
<td>fd</td>
<td>MOVCF</td>
<td>010001</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** MOVF.S fd, fs, cc MIPS32 (MIPS IV)

**MOVF.D fd, fs, cc MIPS32 (MIPS IV)**

**Purpose:**

To test an FP condition code then conditionally move an FP value

**Description:** if cc = 0 then fd ← fs

If the floating point condition code specified by CC is zero, then the value in FPR fs is placed into FPR fd. The source and destination are values in format fmt.

If the condition code is not zero, then FPR fs is not copied and FPR fd retains its previous value in format fmt. If fd did not contain a value either in format fmt or previously unused data from a load or move-to operation that could be interpreted in format fmt, then the value of fd becomes UNPREDICTABLE.

The move is non-arithmetic; it causes no IEEE 754 exceptions.

**Restrictions:**

The fields fs and fd must specify FPRs valid for operands of type fmt; if they are not valid, the result is UNPREDICTABLE. The operand must be a value in format fmt; if it is not, the result is UNPREDICTABLE and the value of the operand FPR becomes UNPREDICTABLE.
Floating Point Move Conditional on Floating Point False (cont.)

**Operation:**

```plaintext
if FPConditionCode(cc) = 0 then
    StoreFPR(fd, fmt, ValueFPR(fs, fmt))
else
    StoreFPR(fd, fmt, ValueFPR(fd, fmt))
```

**Exceptions:**

Coprocessor Unusable, Reserved Instruction

**Floating Point Exceptions:**

Unimplemented Operation
**Move Conditional on Not Zero**

**MOVN**

### Format:

```plaintext
MOVN rd, rs, rt
```

**Purpose:**

To conditionally move a GPR after testing a GPR value

**Description:**

if rt ≠ 0 then rd ← rs

If the value in GPR `rt` is not equal to zero, then the contents of GPR `rs` are placed into GPR `rd`.

**Restrictions:**

None

**Operation:**

```plaintext
if GPR[rt] ≠ 0 then
    GPR[rd] ← GPR[rs]
endif
```

**Exceptions:**

None

**Programming Notes:**

The non-zero value tested here is the *condition true* result from the SLT, SLTI, SLTU, and SLTIU comparison instructions.
Floating Point Move Conditional on Not Zero

Format:  
MOVN.S fd, fs, rt  
MOVN.D fd, fs, rt

MIPS32 (MIPS IV)

Purpose:
To test a GPR then conditionally move an FP value

Description:
if rt \neq 0 then fd \leftarrow fs

If the value in GPR rt is not equal to zero, then the value in FPR fs is placed in FPR fd. The source and destination are values in format fmt.

If GPR rt contains zero, then FPR fs is not copied and FPR fd contains its previous value in format fmt. If fd did not contain a value either in format fmt or previously unused data from a load or move-to operation that could be interpreted in format fmt, then the value of fd becomes UNPREDICTABLE.

The move is non-arithmetic; it causes no IEEE 754 exceptions.

Restrictions:
The fields fs and fd must specify FPRs valid for operands of type fmt; if they are not valid, the result is UNPREDICTABLE.

The operand must be a value in format fmt; if it is not, the result is UNPREDICTABLE and the value of the operand FPR becomes UNPREDICTABLE.

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>COP1</td>
<td>fmt</td>
<td>rt</td>
<td>fs</td>
<td>fd</td>
<td>MOVN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010001</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Floating Point Move Conditional on Not Zero

**Operation:**

\[
\text{if } \text{GPR}[rt] \neq 0 \text{ then }\\
\quad \text{StoreFPR}(fd, \text{fmt}, \text{ValueFPR}(fs, \text{fmt}))\\
\text{else }\\
\quad \text{StoreFPR}(fd, \text{fmt}, \text{ValueFPR}(fd, \text{fmt}))\\
\text{endif}
\]

**Exceptions:**

- Coprocessor Unusable, Reserved Instruction

**Floating Point Exceptions:**

- Unimplemented Operation
Move Conditional on Floating Point True

### MOVT

**Format:** MOVT rd, rs, cc

**MIPS32 (MIPS IV)**

**Purpose:**
To test an FP condition code then conditionally move a GPR

**Description:**
if \( cc = 1 \) then \( rd \leftarrow rs \)

If the floating point condition code specified by \( CC \) is one, then the contents of GPR \( rs \) are placed into GPR \( rd \).

**Restrictions:**

**Operation:**

```
if FPConditionCode(cc) = 1 then
  GPR[rd] ← GPR[rs]
endif
```

**Exceptions:**
Reserved Instruction, Coprocessor Unusable
Floating Point Move Conditional on Floating Point True

MOVT.fmt

<table>
<thead>
<tr>
<th>COP1</th>
<th>fmt</th>
<th>cc</th>
<th>tf</th>
<th>fs</th>
<th>fd</th>
<th>MOVCF</th>
</tr>
</thead>
<tbody>
<tr>
<td>010001</td>
<td></td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>010001</td>
<td></td>
<td>5</td>
<td>6</td>
<td>0</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

**Format:**

- MOVT.S fd, fs, cc
- MOVT.D fd, fs, cc

**Purpose:**

To test an FP condition code then conditionally move an FP value

**Description:** if cc = 1 then fd ← fs

If the floating point condition code specified by CC is one, then the value in FPR fs is placed into FPR fd. The source and destination are values in format fmt.

If the condition code is not one, then FPR fs is not copied and FPR fd contains its previous value in format fmt. If fd did not contain a value either in format fmt or previously unused data from a load or move-to operation that could be interpreted in format fmt, then the value of fd becomes undefined.

The move is non-arithmetic; it causes no IEEE 754 exceptions.

**Restrictions:**

The fields fs and fd must specify FPRs valid for operands of type fmt; if they are not valid, the result is UNPREDICTABLE. The operand must be a value in format fmt; if it is not, the result is UNPREDICTABLE and the value of the operand FPR becomes UNPREDICTABLE.
Floating Point Move Conditional on Floating Point True

**Operation:**

```c
    if FPConditionCode(cc) = 0 then
        StoreFPR(fd, fmt, ValueFPR(fs, fmt))
    else
        StoreFPR(fd, fmt, ValueFPR(fd, fmt))
    endif
```

**Exceptions:**

Coprocessor Unusable, Reserved Instruction

**Floating Point Exceptions:**

Unimplemented Operation
Move Conditional on Zero

Format: MOVZ rd, rs, rt

Purpose:
To conditionally move a GPR after testing a GPR value

Description: if rt = 0 then rd ← rs
If the value in GPR rt is equal to zero, then the contents of GPR rs are placed into GPR rd.

Restrictions:
None

Operation:
if GPR[rt] = 0 then
    GPR[rd] ← GPR[rs]
endif

Exceptions:
None

Programming Notes:
The zero value tested here is the condition false result from the SLT, SLTI, SLTU, and SLTIU comparison instructions.
Floating Point Move Conditional on Zero

### MOVZ.fmt

<table>
<thead>
<tr>
<th>COP1</th>
<th>fmt</th>
<th>rt</th>
<th>fs</th>
<th>fd</th>
<th>MOVZ</th>
<th>010001</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>010010</td>
</tr>
</tbody>
</table>

**Format:**

- MOVZ.S fd, fs, rt
- MOVZ.D fd, fs, rt

**MIPS32 (MIPS IV)**

**Purpose:**
To test a GPR then conditionally move an FP value

**Description:**

If the value in GPR rt is equal to zero then the value in FPR fs is placed in FPR fd. The source and destination are values in format fmt.

If GPR rt is not zero, then FPR fs is not copied and FPR fd contains its previous value in format fmt. If fd did not contain a value either in format fmt or previously unused data from a load or move-to operation that could be interpreted in format fmt, then the value of fd becomes UNPREDICTABLE.

The move is non-arithmetic; it causes no IEEE 754 exceptions.

**Restrictions:**

The fields fs and fd must specify FPRs valid for operands of type fmt; if they are not valid, the result is UNPREDICTABLE.

The operand must be a value in format fmt; if it is not, the result is UNPREDICTABLE and the value of the operand FPR becomes UNPREDICTABLE.
Floating Point Move Conditional on Zero (cont.)

**Operation:**

if GPR[rt] = 0 then
  StoreFPR(fd, fmt, ValueFPR(fs, fmt))
else
  StoreFPR(fd, fmt, ValueFPR(fd, fmt))
endif

**Exceptions:**

Coprocessor Unusable, Reserved Instruction

**Floating Point Exceptions:**

Unimplemented Operation
Multiply and Subtract Word to Hi,Lo

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL2</td>
<td>rs</td>
<td>rt</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>MSUB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011100</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** MSUB rs, rt

**Purpose:**
To multiply two words and subtract the result from Hi, Lo

**Description:**
\((\text{LO}, \text{HI}) \leftarrow (rs \times rt) - (\text{LO}, \text{HI})\)
The 32-bit word value in GPR rs is multiplied by the 32-bit value in GPR rt, treating both operands as signed values, to produce a 64-bit result. The product is subtracted from the 64-bit concatenated values of HI and LO. The most significant 32 bits of the result are written into HI and the least significant 32 bits are written into LO. No arithmetic exception occurs under any circumstances.

**Restrictions:**
None

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

**Operation:**
\[
\text{temp} \leftarrow (\text{HI} \mid \text{LO}) - (\text{GPR}[rs] \times \text{GPR}[rt])
\]
\[
\text{HI} \leftarrow \text{temp}_{63..32}
\]
\[
\text{LO} \leftarrow \text{temp}_{31..0}
\]

**Exceptions:**
None

**Programming Notes:**
Where the size of the operands are known, software should place the shorter operand in GPR rt. This may reduce the latency of the instruction on those processors which implement data-dependent instruction latencies.
Multiply and Subtract Word to Hi, Lo

**Format:** \texttt{MSUBU \textsubscript{rs}, \textsubscript{rt}}

**Purpose:**
To multiply two words and subtract the result from Hi, Lo

**Description:** \((LO, HI) \leftarrow (\text{rs} \times \text{rt}) - (LO, HI)\)

The 32-bit word value in GPR \textsubscript{rs} is multiplied by the 32-bit word value in GPR \textsubscript{rt}, treating both operands as unsigned values, to produce a 64-bit result. The product is subtracted from the 64-bit concatenated values of \textit{HI} and \textit{LO}. The most significant 32 bits of the result are written into \textit{HI} and the least significant 32 bits are written into \textit{LO}. No arithmetic exception occurs under any circumstances.

**Restrictions:**
None

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

**Operation:**
\[
\text{temp} \leftarrow (\text{HI} \mathbin{||} \text{LO}) - (\text{GPR}[\text{rs}] \times \text{GPR}[\text{rt}])
\]
\[
\text{HI} \leftarrow \text{temp}_{63..32}
\]
\[
\text{LO} \leftarrow \text{temp}_{31..0}
\]

**Exceptions:**
None

**Programming Notes:**
Where the size of the operands are known, software should place the shorter operand in GPR \textsubscript{rt}. This may reduce the latency of the instruction on those processors which implement data-dependent instruction latencies.
**Move to Coprocessor 0**

**Format:** MTC0 rt, rd  

**Purpose:**
To move the contents of a general register to a coprocessor 0 register.

**Description:** CPR[r0, rd, sel] ← rt  
The contents of general register rt are loaded into the coprocessor 0 register specified by the combination of rd and sel. Not all coprocessor 0 registers support the sel field. In those instances, the sel field must be set to zero.

**Restrictions:**
The results are **UNDEFINED** if coprocessor 0 does not contain a register as specified by rd and sel.

**Operation:**
CPR[0, rd, sel] ← data

**Exceptions:**
Coprocessor Unusable  
Reserved Instruction
Move Word to Floating Point

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>COP1</td>
<td>MT</td>
<td>rt</td>
<td>fs</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010001</td>
<td>00100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** MTC1 rt, fs

**Purpose:**

To copy a word from a GPR to an FPU (CP1) general register

**Description:** fs ← rt

The low word in GPR rt is placed into the low word of floating point (Coprocessor 1) general register fs.

**Restrictions:**

**Operation:**

\[
\text{data} \leftarrow \text{GPR}[rt]_{31..0} \\
\text{StoreFPR}(fs, \text{UNINTERPRETED\_WORD}, \text{data})
\]

**Exceptions:**

Coprocessor Unusable

**Historical Information:**

For MIPS I, MIPS II, and MIPS III the value of FPR fs is UNPREDICTABLE for the instruction immediately following MTC1.
Move Word to Coprocessor 2

MTC2

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>COP2</td>
<td>MT</td>
<td>rt</td>
<td>rd</td>
<td>0</td>
<td>sel</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010010</td>
<td>00100</td>
<td>5</td>
<td>5</td>
<td>8</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:**
MTC2 rt, rd
MTC2 rt, rd, sel

**MIPS32**

**MIPS32**

**Purpose:**
To copy a word from a GPR to a COP2 general register

**Description:** rd ← rt
The low word in GPR rt is placed into the low word of coprocessor 2 general register specified by the rd and sel fields. Note that not all coprocessor 2 registers may support the sel field. In those instances, the sel field must be zero.

**Restrictions:**
The results are UNPREDICTABLE if coprocessor 2 does not contain a register as specified by rd and sel.

**Operation:**
\[
data \leftarrow \text{GPR}[rt]_{31..0} \\
\text{CPR}[2, rd, sel] \leftarrow data
\]

**Exceptions:**
Coprocessor Unusable
**Move to HI Register**

**Format:** MTHI rs

**Purpose:**
To copy a GPR to the special purpose HI register

**Description:** HI ← rs

The contents of GPR rs are loaded into special register HI.

**Restrictions:**
A computed result written to the HI/LO pair by DIV, DIVU,MULT, or MULTU must be read by MFHI or MFLO before a new result can be written into either HI or LO.

If an MTHI instruction is executed following one of these arithmetic instructions, but before an MFLO or MFHI instruction, the contents of LO are UNPREDICTABLE.

The following example shows this illegal situation:

```
MUL r2,r4 # start operation that will eventually write to HI,LO
...        # code not containing mfhi or mflo
MTHI r6
...        # code not containing mflo
MFLO r3   # this mflo would get an UNPREDICTABLE value
```

**Operation:**

HI ← GPR[rs]

**Exceptions:**
None

**Historical Information:**
In MIPS I-III, if either of the two preceding instructions is MFHI, the result of that MFHI is UNPREDICTABLE. Reads of the HI or LO special register must be separated from any subsequent instructions that write to them by two or more instructions. In MIPS IV and later, including MIPS32 and MIPS64, this restriction does not exist.
Move to LO Register

### MTLO

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>6</th>
<th>5</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL</td>
<td>rs</td>
<td>0</td>
<td>000000</td>
<td>000000000000</td>
<td>MTLO</td>
<td>010011</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>15</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** `MTLO rs`

**Purpose:**
To copy a GPR to the special purpose `LO` register

**Description:** `LO ← rs`

The contents of GPR `rs` are loaded into special register `LO`.

**Restrictions:**
A computed result written to the `HI/LO` pair by DIV, DIVU, MULT, or MULTU must be read by MFHI or MFLO before a new result can be written into either `HI` or `LO`.

If an MTLO instruction is executed following one of these arithmetic instructions, but before an MFLO or MFHI instruction, the contents of `HI` are UNPREDICTABLE. The following example shows this illegal situation:

```
MUL r2,r4  # start operation that will eventually write to HI,LO
...       # code not containing mfhi or mflo
MTLO r6   # code not containing mfhi
MFHI r3    # this mfhi would get an UNPREDICTABLE value
```

**Operation:**

`LO ← GPR[rs]`

**Exceptions:**
None

**Historical Information:**
In MIPS I-III, if either of the two preceding instructions is MFHI, the result of that MFHI is UNPREDICTABLE. Reads of the `HI` or `LO` special register must be separated from any subsequent instructions that write to them by two or more instructions. In MIPS IV and later, including MIPS32 and MIPS64, this restriction does not exist.
MIPS32™ Architecture For Programmers Volume II, Revision 0.95

Multiply Word to GPR

<table>
<thead>
<tr>
<th>Format:</th>
<th>MUL rd, rs, rt</th>
<th>MIPS32</th>
</tr>
</thead>
</table>

**Purpose:**
To multiply two words and write the result to a GPR.

**Description:**
rd ← rs × rt

The 32-bit word value in GPR rs is multiplied by the 32-bit value in GPR rt, treating both operands as signed values, to produce a 64-bit result. The least significant 32 bits of the product are written to GPR rd. The contents of HI and LO are UNPREDICTABLE after the operation. No arithmetic exception occurs under any circumstances.

**Restrictions:**
Note that this instruction does not provide the capability of writing the result to the HI and LO registers.

**Operation:**

1. `temp <- GPR[rs] * GPR[rt]`
2. `GPR[rd] <- temp31..0`
3. `HI <- UNPREDICTABLE`
4. `LO <- UNPREDICTABLE`

**Exceptions:**
None

**Programming Notes:**

In some processors the integer multiply operation may proceed asynchronously and allow other CPU instructions to execute before it is complete. An attempt to read LO or HI before the results are written interlocks until the results are ready. Asynchronous execution does not affect the program result, but offers an opportunity for performance improvement by scheduling the multiply so that other instructions can execute in parallel.

Programs that require overflow detection must check for it explicitly.

Where the size of the operands are known, software should place the shorter operand in GPR rt. This may reduce the latency of the instruction on those processors which implement data-dependent instruction latencies.
<table>
<thead>
<tr>
<th>COP1</th>
<th>fmt</th>
<th>ft</th>
<th>fs</th>
<th>fd</th>
<th>MUL</th>
<th>000010</th>
</tr>
</thead>
<tbody>
<tr>
<td>010001</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

### Format:
- MUL.S fd, fs, ft
- MUL.D fd, fs, ft

### Purpose:
To multiply FP values

### Description:
\[ fd \leftarrow fs \times ft \]
The value in FPR \( fs \) is multiplied by the value in FPR \( ft \). The result is calculated to infinite precision, rounded according to the current rounding mode in \( FCSR \), and placed into FPR \( fd \). The operands and result are values in format \( fmt \).

### Restrictions:
The fields \( fs, ft, \) and \( fd \) must specify FPRs valid for operands of type \( fmt \); if they are not valid, the result is UNPREDICTABLE.

The operands must be values in format \( fmt \); if they are not, the result is UNPREDICTABLE and the value of the operand FPRs becomes UNPREDICTABLE.

### Operation:
\[
\text{StoreFPR}(fd, fmt, \text{ValueFPR}(fs, fmt) \times fmt, \text{ValueFPR}(ft, fmt))
\]

### Exceptions:
Coprocessor Unusable, Reserved Instruction

### Floating Point Exceptions:
Inexact, Unimplemented Operation, Invalid Operation, Overflow, Underflow
MULT

Format: MULT rs, rt

Purpose:
To multiply 32-bit signed integers

Description: (LO, HI) ← rs × rt
The 32-bit word value in GPR rt is multiplied by the 32-bit value in GPR rs, treating both operands as signed values, to produce a 64-bit result. The low-order 32-bit word of the result is placed into special register LO, and the high-order 32-bit word is placed into special register HI.
No arithmetic exception occurs under any circumstances.

Restrictions:
None

Operation:
prod ← GPR[rs]31..0 × GPR[rt]31..0
LO ← prod31..0
HI ← prod63..32

Exceptions:
None

Programming Notes:
In some processors the integer multiply operation may proceed asynchronously and allow other CPU instructions to execute before it is complete. An attempt to read LO or HI before the results are written interlocks until the results are ready. Asynchronous execution does not affect the program result, but offers an opportunity for performance improvement by scheduling the multiply so that other instructions can execute in parallel.

Programs that require overflow detection must check for it explicitly.

Where the size of the operands are known, software should place the shorter operand in GPR rt. This may reduce the latency of the instruction on those processors which implement data-dependent instruction latencies.
Multiply Unsigned Word

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL</td>
<td>rs</td>
<td>rt</td>
<td>0</td>
<td>00 0000 0000</td>
<td>MULTU</td>
<td>011001</td>
<td></td>
</tr>
</tbody>
</table>

**Format:** MULTU rs, rt  
**MIPS32 (MIPS I)**

**Purpose:**
To multiply 32-bit unsigned integers

**Description:**  
(LO, HI) ← rs × rt

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

No arithmetic exception occurs under any circumstances.

**Restrictions:**
None

**Operation:**

\[
\text{prod} \leftarrow (0 || \text{GPR}[\text{rs}]_{31..0}) \times (0 || \text{GPR}[\text{rt}]_{31..0})
\]

\[
\text{LO} \leftarrow \text{prod}_{31..0}
\]

\[
\text{HI} \leftarrow \text{prod}_{63..32}
\]

**Exceptions:**
None

**Programming Notes:**
In some processors the integer multiply operation may proceed asynchronously and allow other CPU instructions to execute before it is complete. An attempt to read LO or HI before the results are written interlocks until the results are ready. Asynchronous execution does not affect the program result, but offers an opportunity for performance improvement by scheduling the multiply so that other instructions can execute in parallel.

Programs that require overflow detection must check for it explicitly.

Where the size of the operands are known, software should place the shorter operand in GPR rt. This may reduce the latency of the instruction on those processors which implement data-dependent instruction latencies.
Floating Point Negate  

<table>
<thead>
<tr>
<th>COP1</th>
<th>fmt</th>
<th>0</th>
<th>fs</th>
<th>fd</th>
<th>NEG</th>
</tr>
</thead>
<tbody>
<tr>
<td>010001</td>
<td>0</td>
<td>0000</td>
<td>5</td>
<td>5</td>
<td>000111</td>
</tr>
</tbody>
</table>

**Format:**
- `NEG.S fd, fs`
- `NEG.D fd, fs`

**MIPS32 (MIPS I)**

**Purpose:**
To negate an FP value

**Description:** $fd \leftarrow -fs$

The value in FPR $fs$ is negated and placed into FPR $fd$. The value is negated by changing the sign bit value. The operand and result are values in format $fmt$. This operation is arithmetic; a NaN operand signals invalid operation.

**Restrictions:**

The fields $fs$ and $fd$ must specify FPRs valid for operands of type $fmt$; if they are not valid, the result is UNPREDICTABLE. The operand must be a value in format $fmt$; if it is not, the result is UNPREDICTABLE and the value of the operand FPR becomes UNPREDICTABLE.

**Operation:**

```plaintext```
StoreFPR(fd, fmt, Negate(ValueFPR(fs, fmt)))
```

**Exceptions:**

Coprocessor Unusable, Reserved Instruction

**Floating Point Exceptions:**

Unimplemented Operation, Invalid Operation
### No Operation

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>SLL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000000</td>
<td>00000</td>
<td>00000</td>
<td>00000</td>
<td>00000</td>
<td>00000</td>
<td>00000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:**  NOP

**Assembly Idiom**

**Purpose:**

To perform no operation.

**Description:**

NOP is the assembly idiom used to denote no operation. The actual instruction is interpreted by the hardware as SLL r0, r0, 0.

**Restrictions:**

None

**Operation:**

None

**Exceptions:**

None

**Programming Notes:**

The zero instruction word, which represents SLL, r0, r0, 0, is the preferred NOP for software to use to fill branch and jump delay slots and to pad out alignment sequences.
Not Or

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>0</td>
<td>NOR</td>
<td>00000</td>
<td>100111</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** NOR rd, rs, rt

**Purpose:**
To do a bitwise logical NOT OR

**Description:** rd ← rs NOR rt
The contents of GPR rs are combined with the contents of GPR rt in a bitwise logical NOR operation. The result is placed into GPR rd.

**Restrictions:**
None

**Operation:**

GPR[rd] ← GPR[rs] nor GPR[rt]

**Exceptions:**
None
**Or**

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>0</td>
<td>OR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000000</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** OR rd, rs, rt

**MIPS32 (MIPS I)**

**Purpose:**

To do a bitwise logical OR

**Description:** rd ← rs or rt

The contents of GPR rs are combined with the contents of GPR rt in a bitwise logical OR operation. The result is placed into GPR rd.

**Restrictions:**

None

**Operation:**

GPR[rd] ← GPR[rs] or GPR[rt]

**Exceptions:**

None
## Or Immediate

### MIPS32 (MIPS I)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>001101</th>
<th></th>
<th></th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>26</td>
<td>25</td>
<td>21</td>
<td>20</td>
<td>16</td>
</tr>
</tbody>
</table>

### Format: ORI rt, rs, immediate

### Purpose:
To do a bitwise logical OR with a constant

### Description:
rt ← rs or immediate

The 16-bit immediate is zero-extended to the left and combined with the contents of GPR rs in a bitwise logical OR operation. The result is placed into GPR rt.

### Restrictions:
None

### Operation:

GPR[rt] ← GPR[rs] or zero_extend(immediate)

### Exceptions:
None
Prefetch

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** \( \text{PREF} \text{ hint,offset(base)} \)

**Purpose:**
To move data between memory and cache.

**Description:** \( \text{prefetch\_memory(base+offset)} \)

PREF adds the 16-bit signed \( \text{offset} \) to the contents of GPR \( \text{base} \) to form an effective byte address. The \( \text{hint} \) field supplies information about the way that the data is expected to be used.

PREF enables the processor to take some action, typically prefetching the data into cache, to improve program performance. The action taken for a specific PREF instruction is both system and context dependent. Any action, including doing nothing, is permitted as long as it does not change architecturally visible state or alter the meaning of a program. Implementations are expected either to do nothing, or to take an action that increases the performance of the program.

PREF does not cause addressing-related exceptions. If the address specified would cause an addressing exception, the exception condition is ignored and no data movement occurs. However even if no data is prefetched, some action that is not architecturally visible, such as writeback of a dirty cache line, can take place.

PREF never generates a memory operation for a location with an \( \text{uncached} \) memory access type.

If PREF results in a memory operation, the memory access type used for the operation is determined by the memory access type of the effective address, just as it would be if the memory operation had been caused by a load or store to the effective address.

For a cached location, the expected and useful action for the processor is to prefetch a block of data that includes the effective address. The size of the block and the level of the memory hierarchy it is fetched into are implementation specific.

The \( \text{hint} \) field supplies information about the way the data is expected to be used. A \( \text{hint} \) value cannot cause an action to modify architecturally visible state. A processor may use a \( \text{hint} \) value to improve the effectiveness of the prefetch action.
**Table 3-29 Values of the hint Field for the PREF Instruction**

<table>
<thead>
<tr>
<th>Value</th>
<th>Name</th>
<th>Data Use and Desired Prefetch Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>load</td>
<td>Use: Prefetched data is expected to be read (not modified). Action: Fetch data as if for a load.</td>
</tr>
<tr>
<td>1</td>
<td>store</td>
<td>Use: Prefetched data is expected to be stored or modified. Action: Fetch data as if for a store.</td>
</tr>
<tr>
<td>2-3</td>
<td>Reserved</td>
<td>Reserved for future use - not available to implementations.</td>
</tr>
<tr>
<td>4</td>
<td>load_streamed</td>
<td>Use: Prefetched data is expected to be read (not modified) but not reused extensively; it “streams” through cache. Action: Fetch data as if for a load and place it in the cache so that it does not displace data prefetched as “retained.”</td>
</tr>
<tr>
<td>5</td>
<td>store_streamed</td>
<td>Use: Prefetched data is expected to be stored or modified but not reused extensively; it “streams” through cache. Action: Fetch data as if for a store and place it in the cache so that it does not displace data prefetched as “retained.”</td>
</tr>
<tr>
<td>6</td>
<td>load_retained</td>
<td>Use: Prefetched data is expected to be read (not modified) and reused extensively; it should be “retained” in the cache. Action: Fetch data as if for a load and place it in the cache so that it is not displaced by data prefetched as “streamed.”</td>
</tr>
<tr>
<td>7</td>
<td>store_retained</td>
<td>Use: Prefetched data is expected to be stored or modified and reused extensively; it should be “retained” in the cache. Action: Fetch data as if for a store and place it in the cache so that it is not displaced by data prefetched as “streamed.”</td>
</tr>
</tbody>
</table>
### Table 3-29 Values of the hint Field for the PREF Instruction

<table>
<thead>
<tr>
<th>8-24</th>
<th>Reserved</th>
<th>Reserved for future use - not available to implementations.</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>writeback_invalidate (also known as &quot;nudge&quot;)</td>
<td>Use: Data is no longer expected to be used. Action: For a writeback cache, schedule a writeback of any dirty data. At the completion of the writeback, mark the state of any cache lines written back as invalid.</td>
</tr>
<tr>
<td>26-29</td>
<td>Implementation Dependent</td>
<td>Unassigned by the Architecture - available for implementation-dependent use.</td>
</tr>
<tr>
<td>30</td>
<td>PrepareForStore</td>
<td>Use: Prepare the cache for writing an entire line, without the overhead involved in filling the line from memory. Action: If the reference hits in the cache, no action is taken. If the reference misses in the cache, a line is selected for replacement, any valid and dirty victim is written back to memory, the entire line is filled with zero data, and the state of the line is marked as valid and dirty.</td>
</tr>
<tr>
<td>31</td>
<td>Implementation Dependent</td>
<td>Unassigned by the Architecture - available for implementation-dependent use.</td>
</tr>
</tbody>
</table>
Prefetch (cont.)

Restrictions:
None

Operation:
\[
\text{vAddr} \leftarrow \text{GPR}[\text{base}] + \text{sign}_{-}\text{extend}(\text{offset}) \\
(p\text{Addr}, \text{CCA}) \leftarrow \text{AddressTranslation}(\text{vAddr, DATA, LOAD}) \\
\text{Prefetch}(\text{CCA, pAddr, vAddr, DATA, hint})
\]

Exceptions:
Prefetch does not take any TLB-related or address-related exceptions under any circumstances.

Programming Notes:
Prefetch cannot prefetch data from a mapped location unless the translation for that location is present in the TLB. Locations in memory pages that have not been accessed recently may not have translations in the TLB, so prefetch may not be effective for such locations.

Prefetch does not cause addressing exceptions. It does not cause an exception to prefetch using an address pointer value before the validity of a pointer is determined.

*Hint* field encodings whose function is described as “streamed” or “retained” convey usage intent from software to hardware. Software should not assume that hardware will always prefetch data in an optimal way. If data is to be truly retained, software should use the Cache instruction to lock data into the cache.
Floating Point Round to Word Fixed Point

### ROUND.W.fmt

<table>
<thead>
<tr>
<th>COP1</th>
<th>fmt</th>
<th>0</th>
<th>fs</th>
<th>fd</th>
<th>ROUND.W</th>
</tr>
</thead>
<tbody>
<tr>
<td>010001</td>
<td>00000</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

**Format:**

- **ROUND.W.S**  fd, fs
- **ROUND.W.D**  fd, fs

**MIPS32 (MIPS II)**

**Purpose:**
To convert an FP value to 32-bit fixed point, rounding to nearest

**Description:**

\[ fd \leftarrow \text{convert\_and\_round}(fs) \]

The value in FPR \( fs \), in format \( fmt \), is converted to a value in 32-bit word fixed point format rounding to nearest/even (rounding mode 0). The result is placed in FPR \( fd \).

When the source value is Infinity, NaN, or rounds to an integer outside the range \(-2^{31} \) to \( 2^{31}-1 \), the result cannot be represented correctly and an IEEE Invalid Operation condition exists. In this case the Invalid Operation flag is set in the FCSR. If the Invalid Operation Enable bit is set in the FCSR, no result is written to \( fd \) and an Invalid Operation exception is taken immediately. Otherwise, the default result, \( 2^{31} - 1 \), is written to \( fd \).

**Restrictions:**

The fields \( fs \) and \( fd \) must specify valid FPRs; \( fs \) for type \( fmt \) and \( fd \) for word fixed point; if they are not valid, the result is UNPREDICTABLE.

The operand must be a value in format \( fmt \); if it is not, the result is UNPREDICTABLE and the value of the operand FPR becomes UNPREDICTABLE.

**Operation:**

\[
\text{StoreFPR}(fd, W, \text{ConvertFmt(}\text{ValueFPR}(fs, fmt), fmt, W))
\]
Floating Point Round to Word Fixed Point (cont).

**Exceptions:**
Coprocessor Unusable, Reserved Instruction

**Floating Point Exceptions:**
Inexact, Unimplemented Operation, Invalid Operation, Overflow
**Store Byte**

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SB</td>
<td>base</td>
<td>rt</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>101000</td>
<td>5</td>
<td>5</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:**  \( SB \ rt, \ offset(base) \)  

**MIPS32 (MIPS I)**

**Purpose:**  
To store a byte to memory

**Description:**  
\( memory[base+offset] \leftarrow rt \)  
The least-significant 8-bit byte of GPR \( rt \) is stored in memory at the location specified by the effective address. The 16-bit signed \( offset \) is added to the contents of GPR \( base \) to form the effective address.

**Restrictions:**  
None

**Operation:**  
\[
\begin{align*}
vAddr & \leftarrow \text{sign\_extend}(offset) + \text{GPR}[base] \\
(pAddr, CCA) & \leftarrow \text{AddressTranslation}(vAddr, DATA, STORE) \\
pAddr & \leftarrow pAddr_{PSIZE-1..2} || (pAddr_{1..0} \text{ xor ReverseEndian}^2) \\
\text{bytesel} & \leftarrow vAddr_{1..0} \text{ xor BigEndianCPU}^2 \\
\text{dataword} & \leftarrow \text{GPR}[rt]_{31-8*\text{bytesel}.0} || 0^{8*\text{bytesel}} \\
\text{StoreMemory}(CCA, \text{BYTE}, \text{dataword}, pAddr, vAddr, DATA)
\end{align*}
\]

**Exceptions:**  
TLB Refill, TLB Invalid, TLB Modified, Bus Error, Address Error
Store Conditional Word

```
<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SC</td>
<td>111000</td>
<td>base</td>
<td>rt</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**Format:** \( SC \ rt, \ offset(base) \)  

**Purpose:**  
To store a word to memory to complete an atomic read-modify-write

**Description:**  
if \( atomic\_update \) then \( memory[base+offset] \leftarrow rt \), \( rt \leftarrow 1 \) else \( rt \leftarrow 0 \)

The LL and SC instructions provide primitives to implement atomic read-modify-write (RMW) operations for cached memory locations.

The 16-bit signed \( offset \) is added to the contents of GPR \( base \) to form an effective address.

The SC completes the RMW sequence begun by the preceding LL instruction executed on the processor. To complete the RMW sequence atomically, the following occur:

- The least-significant 32-bit word of GPR \( rt \) is stored into memory at the location specified by the aligned effective address.
- A 1, indicating success, is written into GPR \( rt \).

Otherwise, memory is not modified and a 0, indicating failure, is written into GPR \( rt \).

If either of the following events occurs between the execution of LL and SC, the SC fails:

- A coherent store is completed by another processor or coherent I/O module into the block of physical memory containing the word. The size and alignment of the block is implementation dependent, but it is at least one word and at most the minimum page size.
- An exception occurs on the processor executing the LL/SC.

If either of the following events occurs between the execution of LL and SC, the SC may succeed or it may fail; the success or failure is not predictable. Portable programs should not cause one of these events.

- A load, store, or prefetch is executed on the processor executing the LL/SC.
- The instructions executed starting with the LL and ending with the SC do not lie in a 2048-byte contiguous region of virtual memory. The region does not have to be aligned, other than the alignment required for instruction words.

The following conditions must be true or the result of the SC is undefined:

- Execution of SC must have been preceded by execution of an LL instruction.
- A RMW sequence executed without intervening exceptions must use the same address in the LL and SC. The address is the same if the virtual address, physical address, and cache-coherence algorithm are identical.
Atomic RMW is provided only for cached memory locations. The extent to which the detection of atomicity operates correctly depends on the system implementation and the memory access type used for the location:

- **MP atomicity:** To provide atomic RMW among multiple processors, all accesses to the location must be made with a memory access type of *cached coherent*.

- **Uniprocessor atomicity:** To provide atomic RMW on a single processor, all accesses to the location must be made with memory access type of either *cached noncoherent* or *cached coherent*. All accesses must be to one or the other access type, and they may not be mixed.

**I/O System:** To provide atomic RMW with a coherent I/O system, all accesses to the location must be made with a memory access type of *cached coherent*. If the I/O system does not use coherent memory operations, then atomic RMW cannot be provided with respect to the I/O reads and writes.

**Restrictions:**

The addressed location must have a memory access type of *cached noncoherent* or *cached coherent*; if it does not, the result is undefined.

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

**Operation:**

\[
\text{vAddr} \leftarrow \text{sign extend}(\text{offset}) + \text{GPR}[\text{base}]
\]

\[
\text{if } \text{vAddr}_{1..0} \neq 0^2 \text{ then}
\]

\[
\text{SignalException(AddressError)}
\]

\[
\text{endif}
\]

\[
(p\text{Addr}, CCA) \leftarrow \text{AddressTranslation(vAddr, DATA, STORE)}
\]

\[
data\text{word} \leftarrow \text{GPR}[rt]
\]

\[
\text{if LLbit then}
\]

\[
\text{StoreMemory(CCA, WORD, dataword, pAddr, vAddr, DATA)}
\]

\[
\text{endif}
\]

\[
\text{GPR}[rt] \leftarrow 31 || \text{LLbit}
\]
Exceptions:
TLB Refill, TLB Invalid, TLB Modified, Address Error, Reserved Instruction

Programming Notes:
LL and SC are used to atomically update memory locations, as shown below.

```
L1:
  LL  T1, (T0)  # load counter
  ADDI  T2, T1, 1  # increment
  SC  T2, (T0)  # try to store, checking for atomicity
  BEQ  T2, 0, L1  # if not atomic (0), try again
  NOP  # branch-delay slot
```

Exceptions between the LL and SC cause SC to fail, so persistent exceptions must be avoided. Some examples of these are arithmetic operations that trap, system calls, and floating point operations that trap or require software emulation assistance.

LL and SC function on a single processor for cached noncoherent memory so that parallel programs can be run on uniprocessor systems that do not support cached coherent memory access types.
Software Debug Breakpoint

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL2</td>
<td>code</td>
<td>SDBBP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011100</td>
<td></td>
<td>111111</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** SDBBP code

**Purpose:**
To cause a debug breakpoint exception

**Description:**
This instruction causes a debug exception, passing control to the debug exception handler. The code field can be used for passing information to the debug exception handler, and is retrieved by the debug exception handler only by loading the contents of the memory word containing the instruction, using the DEPC register. The CODE field is not used in any way by the hardware.

**Restrictions:**

**Operation:**

```plaintext
If DebugDM = 0 then
   SignalDebugBreakpointException()
else
   SignalDebugModeBreakpointException()
endif
```

**Exceptions:**
Debug Breakpoint Exception
Store Doubleword from Floating Point

Format: SDC1 ft, offset(base)

Purpose:
To store a doubleword from an FPR to memory

Description: memory[base+offset] ← ft
The 64-bit doubleword in FPR ft is stored in memory at the location specified by the aligned effective address. The 16-bit signed offset is added to the contents of GPR base to form the effective address.

Restrictions:
An Address Error exception occurs if EffectiveAddress₂.₀ ≠ 0 (not doubleword-aligned).

Operation:

\[
\begin{align*}
\text{vAddr} & \leftarrow \text{sign_extend(offset)} + \text{GPR[base]} \\
& \text{if vAddr}_{2..0} \neq 0 \text{ then} \\
& \quad \text{SignalException(AddressError)} \\
& \text{endif} \\
& \quad (pAddr, \text{CCA}) \leftarrow \text{AddressTranslation(vAddr, DATA, STORE)} \\
& \quad \text{datadoubleword} \leftarrow \text{ValueFPR(ft, UNINTERPRETED_DOUBLEWORD)} \\
& \quad \text{StoreMemory(CCA, DOUBLEWORD, datadoubleword, pAddr, vAddr, DATA)}
\end{align*}
\]

Exceptions:
Coprocessor Unusable, Reserved Instruction, TLB Refill, TLB Invalid, TLB Modified, Address Error
Store Doubleword from Coprocessor 2

SDC2

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SDC</td>
<td>111110</td>
<td>base</td>
<td>rt</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** SDC2 rt, offset(base)

**Purpose:**
To store a doubleword from a Coprocessor 2 register to memory

**Description:** memory[base+offset] ← rt

The 64-bit doubleword in Coprocessor 2 register rt is stored in memory at the location specified by the aligned effective address. The 16-bit signed offset is added to the contents of GPR base to form the effective address.

**Restrictions:**
An Address Error exception occurs if EffectiveAddress2..0 ≠ 0 (not doubleword-aligned).

**Operation:**
```
vAddr ← sign_extend(offset) + GPR[base]
if vAddr2..0 ≠ 0 then
    SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation(vAddr, DATA, STORE)
datadoubleword ← CPR[2,rt,0]
StoreMemory(CCA, DOUBLEWORD, datadoubleword, pAddr, vAddr, DATA)
```

**Exceptions:**
Coprocessor Unusable, Reserved Instruction, TLB Refill, TLB Invalid, TLB Modified, Address Error
Store Halfword

Format: \( \text{SH rt, offset(base)} \)  

MIPS32 (MIPS I)

Purpose:
To store a halfword to memory

Description: \( \text{memory[base+offset]} \leftarrow \text{rt} \)
The least-significant 16-bit halfword of register \( rt \) is stored in memory at the location specified by the aligned effective address. The 16-bit signed \( offset \) is added to the contents of GPR \( base \) to form the effective address.

Restrictions:
The effective address must be naturally-aligned. If the least-significant bit of the address is non-zero, an Address Error exception occurs.

Operation:
\[
\begin{align*}
\text{vAddr} & \leftarrow \text{sign\_extend(offset)} + \text{GPR[base]} \\
& \text{if } \text{vAddr}_0 \neq 0 \text{ then} \\
& \quad \text{SignalException(AddressError)} \\
& \text{endif} \\
& (\text{pAddr, CCA}) \leftarrow \text{AddressTranslation (vAddr, DATA, STORE)} \\
& \text{pAddr} \leftarrow \text{pAddr}_{\text{PSIZE}} \ || \ (\text{pAddr}_{1..0} \text{xor (ReverseEndian } || \ 0)) \\
& \text{bytesel} \leftarrow \text{vAddr}_{1..0} \text{xor (BigEndianCPU } || \ 0) \\
& \text{dataword} \leftarrow \text{GPR[rt]}_{31-8*\text{bytesel}.0} \ || \ 0^{8*\text{bytesel}} \\
& \text{StoreMemory (CCA, HALFWORD, dataword, pAddr, vAddr, DATA)}
\end{align*}
\]

Exceptions:
TLB Refill, TLB Invalid, TLB Modified, Address Error
Shift Word Left Logical

**SLL**

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL</td>
<td>0</td>
<td>0</td>
<td>rt</td>
<td>rd</td>
<td>sa</td>
<td>SLL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000000</td>
<td>00000</td>
<td>00000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Format: SLL rd, rt, sa

Purpose:
To left-shift a word by a fixed number of bits

Description: rd ← rt << sa
The contents of the low-order 32-bit word of GPR rt are shifted left, inserting zeros into the emptied bits; the word result is placed in GPR rd. The bit-shift amount is specified by sa.

Restrictions:
None

Operation:
s ← sa
temp ← GPR[rt](31-s)..0 || 0^s
GPR[rd] ← temp

Exceptions:
None

Programming Notes:
SLL r0, r0, 0, expressed as NOP, is the assembly idiom used to denote no operation.
SLL r0, r0, 1, expressed as SSNOP, is the assembly idiom used to denote no operation that causes an issue break on superscalar processors.
Shift Word Left Logical Variable

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>0</td>
<td>SLLV</td>
<td>000000</td>
<td>000100</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| 6 | 5 | 5 | 5 | 6 |

Format: SLLV rd, rt, rs

MIPS32 (MIPS I)

Purpose: To left-shift a word by a variable number of bits

Description: rd ← rt << rs

The contents of the low-order 32-bit word of GPR rt are shifted left, inserting zeros into the emptied bits; the result word is placed in GPR rd. The bit-shift amount is specified by the low-order 5 bits of GPR rs.

Restrictions: None

Operation:

s ← GPR[rs]4..0  
temp ← GPR[rt](31-s)..0 || 0s  
GPR[rd] ← temp

Exceptions: None

Programming Notes:

None
Set on Less Than

### SLT

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 31 | 26 | 25 | 21 | 20 | 16 | 15 | 11 | 10 |  6 |  5 |  6 |  5 |  5 |  5 |  0 |  0 | 00000 | 101010 |
|     | SPECIAL | rs | rt | rd | 0 | SLT |
| 000000 | 6 | 5 | 5 | 5 | 6 |

#### Format:

\[ \text{SLT } rd, rs, rt \]

#### Purpose:

To record the result of a less-than comparison.

#### Description:

\[ rd \leftarrow (rs < rt) \]

Compare the contents of GPR \( rs \) and GPR \( rt \) as signed integers and record the Boolean result of the comparison in GPR \( rd \). If GPR \( rs \) is less than GPR \( rt \), the result is 1 (true); otherwise, it is 0 (false).

The arithmetic comparison does not cause an Integer Overflow exception.

#### Restrictions:

None

#### Operation:

```plaintext
if GPR[rs] < GPR[rt] then
    GPR[rd] \leftarrow 0^{GPRLEN-1} | 1
else
    GPR[rd] \leftarrow 0^{GPRLEN}
endif
```

#### Exceptions:

None
Set on Less Than Immediate

```
SLTI 001010
001010
```

**Format:** \( SLTI \) \( rt, rs, \) immediate

**MIPS32 (MIPS I)**

**Purpose:**
To record the result of a less-than comparison with a constant

**Description:** \( rt \leftarrow (rs < \text{immediate}) \)

Compare the contents of GPR \( rs \) and the 16-bit signed \( \text{immediate} \) as signed integers and record the Boolean result of the comparison in GPR \( rt \). If GPR \( rs \) is less than \( \text{immediate} \), the result is 1 (true); otherwise, it is 0 (false).

The arithmetic comparison does not cause an Integer Overflow exception.

**Restrictions:**
None

**Operation:**

\[
\begin{align*}
\text{if } \text{GPR}[rs] < \text{sign extend}(\text{immediate}) \text{ then} \\
\text{GPR}[rd] & \leftarrow 0_{\text{GPRLEN}-1}||1 \\
\text{else} & \\
\text{GPR}[rd] & \leftarrow 0_{\text{GPRLEN}}
\end{align*}
\]

**Exceptions:**
None
Set on Less Than Immediate Unsigned

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SLTIU</td>
<td>rs</td>
<td>rt</td>
<td>immediate</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001011</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** SLTIU rt, rs, immediate

**Purpose:**
To record the result of an unsigned less-than comparison with a constant

**Description:** rt ← (rs < immediate)

Compare the contents of GPR rs and the sign-extended 16-bit immediate as unsigned integers and record the Boolean result of the comparison in GPR rt. If GPR rs is less than immediate, the result is 1 (true); otherwise, it is 0 (false).

Because the 16-bit immediate is sign-extended before comparison, the instruction can represent the smallest or largest unsigned numbers. The representable values are at the minimum [0, 32767] or maximum [max_unsigned-32767, max_unsigned] end of the unsigned range.

The arithmetic comparison does not cause an Integer Overflow exception.

**Restrictions:**
None

**Operation:**

if (0 || GPR[rs]) < (0 || sign_extend(immediate)) then
  GPR[rd] ← 0^GPRLEN-1 || 1
else
  GPR[rd] ← 0^GPRLEN
endif

**Exceptions:**
None
Set on Less Than Unsigned

<table>
<thead>
<tr>
<th>Format:</th>
<th>SLTU rd, rs, rt</th>
<th>MIPS32 (MIPS I)</th>
</tr>
</thead>
</table>

**Purpose:**
To record the result of an unsigned less-than comparison

**Description:**
\[ \text{rd} \leftarrow (rs < rt) \]

Compare the contents of GPR \( rs \) and GPR \( rt \) as unsigned integers and record the Boolean result of the comparison in GPR \( rd \). If GPR \( rs \) is less than GPR \( rt \), the result is 1 (true); otherwise, it is 0 (false).

The arithmetic comparison does not cause an Integer Overflow exception.

**Restrictions:**
None

**Operation:**
\[
\text{if (0} | | \text{GPR}[rs]) < (0 | | \text{GPR}[rt]) \text{ then}
\]
\[
\text{GPR}[rd] \leftarrow 0^{\text{GPRLEN}-1} | | 1
\]
\[
\text{else}
\]
\[
\text{GPR}[rd] \leftarrow 0^{\text{GPRLEN}}
\]
\[
\text{endif}
\]

**Exceptions:**
None
**Floating Point Square Root**

### SQRT.fmt

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>COP1</td>
<td>fmt</td>
<td>0</td>
<td>fs</td>
<td>fd</td>
<td>SQRT</td>
<td>00100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010001</td>
<td>00000</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:**  
SQRT.S fd, fs  
SQRT.D fd, fs

**MIPS32 (MIPS II)**

**Purpose:**
To compute the square root of an FP value

**Description:**  
\[ fd \leftarrow \text{SQRT}(fs) \]  
The square root of the value in FPR \( fs \) is calculated to infinite precision, rounded according to the current rounding mode in \( FCSR \), and placed into FPR \( fd \). The operand and result are values in format \( fmt \).

If the value in FPR \( fs \) corresponds to \(-0\), the result is \(-0\).

**Restrictions:**
If the value in FPR \( fs \) is less than 0, an Invalid Operation condition is raised.

The fields \( fs \) and \( fd \) must specify FPRs valid for operands of type \( fmt \); if they are not valid, the result is **UNPREDICTABLE**.

The operand must be a value in format \( fmt \); if it is not, the result is **UNPREDICTABLE** and the value of the operand FPR becomes **UNPREDICTABLE**.

**Operation:**
\[ \text{StoreFPR}(fd, \text{fmt}, \text{SquareRoot(ValueFPR}(fs, \text{fmt})) \]

**Exceptions:**
Coprocessor Unusable, Reserved Instruction

**Floating Point Exceptions:**
Invalid Operation, Inexact, Unimplemented Operation
Shift Word Right Arithmetic

Format: SRA rd, rt, sa

Purpose:
To execute an arithmetic right-shift of a word by a fixed number of bits

Description: rd ← rt >> sa  (arithmetic)
The contents of the low-order 32-bit word of GPR rt are shifted right, duplicating the sign-bit (bit 31) in the emptied bits; the word result is placed in GPR rd. The bit-shift amount is specified by sa.

Restrictions:
None

Operation:

$$s \leftarrow sa$$

$$\text{temp} \leftarrow (\text{GPR}[rt]_{31})^s \ || \ \text{GPR}[rt]_{31...s}$$

$$\text{GPR}[rd] \leftarrow \text{temp}$$

Exceptions: None
Shift Word Right Arithmetic Variable  

**Format:**  
SRAV rd, rt, rs  

**MIPS32 (MIPS I)**  

**Purpose:**  
To execute an arithmetic right-shift of a word by a variable number of bits  

**Description:**  
rd ← rt >> rs      (arithmetic)  
The contents of the low-order 32-bit word of GPR rt are shifted right, duplicating the sign-bit (bit 31) in the emptied bits; the word result is placed in GPR rd. The bit-shift amount is specified by the low-order 5 bits of GPR rs.  

**Restrictions:**  
None  

**Operation:**  

g ← GPR[rs]4..0  

\[ \text{temp} \leftarrow (GPR[rt]_{31})^2 \mid GPR[rt]_{31..s} \]  

GPR[rd] ← temp  

**Exceptions:**  
None
Shift Word Right Logical

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL</td>
<td>0</td>
<td>00000</td>
<td>rt</td>
<td>rd</td>
<td>sa</td>
<td>SRL</td>
<td>000010</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Format: SRL rd, rt, sa

Purpose:
To execute a logical right-shift of a word by a fixed number of bits

Description: rd ← rt >> sa (logical)
The contents of the low-order 32-bit word of GPR rt are shifted right, inserting zeros into the emptied bits; the word result is placed in GPR rd. The bit-shift amount is specified by sa.

Restrictions:
None

Operation:
\[ s \leftarrow sa \]
\[ \text{temp} \leftarrow \text{0}\text{s} \| \ GPR[rt]_{31..s} \]
\[ \text{GPR[rd]} \leftarrow \text{temp} \]

Exceptions:
None
Shift Word Right Logical Variable

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>0</td>
<td>SRLV</td>
<td>000000</td>
<td>000110</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** SRLV rd, rt, rs

**MIPS32 (MIPS I)**

**Purpose:**
To execute a logical right-shift of a word by a variable number of bits

**Description:**
rd ← rt >> rs \( \text{(logical)} \)

The contents of the low-order 32-bit word of GPR rt are shifted right, inserting zeros into the emptied bits; the word result is placed in GPR rd. The bit-shift amount is specified by the low-order 5 bits of GPR rs.

**Restrictions:**
None

**Operation:**
\[
\begin{align*}
s & \leftarrow \text{GPR}[rs]_{4..0} \\
temp & \leftarrow 0^s \ || \ | \text{GPR}[rt]_{31..s} \\
\text{GPR}[rd] & \leftarrow temp
\end{align*}
\]

**Exceptions:**
None
Superscalar No Operation

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SLL</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** SSNOP

**MIPS32**

**Purpose:**
Break superscalar issue on a superscalar processor.

**Description:**
SSNOP is the assembly idiom used to denote superscalar no operation. The actual instruction is interpreted by the hardware as SLL r0, r0, 1.

This instruction alters the instruction issue behavior on a superscalar processor by forcing the SSNOP instruction to single-issue. The processor must then end the current instruction issue between the instruction previous to the SSNOP and the SSNOP. The SSNOP then issues alone in the next issue slot.

On a single-issue processor, this instruction is a NOP that takes an issue slot.

**Restrictions:**
None

**Operation:**
None

**Exceptions:**
None

**Programming Notes:**
SSNOP is intended for use primarily to allow the programmer control over CP0 hazards by converting instructions into cycles in a superscalar processor. For example, to insert at least two cycles between an MTC0 and an ERET, one would use the following sequence:

```
mtc0 x,y
ssnop
ssnop
eret
```

Based on the normal issues rules of the processor, the MTC0 issues in cycle T. Because the SSNOP instructions must issue alone, they may issue no earlier than cycle T+1 and cycle T+2, respectively. Finally, the ERET issues no earlier than cycle T+3. Note that although the instruction after an SSNOP may issue no earlier than the cycle after the SSNOP is issued, that instruction may issue later. This is because other implementation-dependent issue rules may apply that prevent an issue in the next cycle. Processors should not introduce any unnecessary delay in issuing SSNOP instructions.
Subtract Word

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>0</td>
<td>SUB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000000</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>00000</td>
<td>100010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Format: \( \text{SUB} \, rd, \, rs, \, rt \)

Purpose:
To subtract 32-bit integers. If overflow occurs, then trap

Description: \( rd \leftarrow rs - rt \)

The 32-bit word value in GPR \( rt \) is subtracted from the 32-bit value in GPR \( rs \) to produce a 32-bit result. If the subtraction results in 32-bit 2’s complement arithmetic overflow, then the destination register is not modified and an Integer Overflow exception occurs. If it does not overflow, the 32-bit result is placed into GPR \( rd \).

Restrictions:
None

Operation:
\[
\text{_temp} \leftarrow (\text{GPR}[rs]_{31} | \text{GPR}[rs]_{31..0}) - (\text{GPR}[rt]_{31} | \text{GPR}[rt]_{31..0})
\]
\[
\text{if temp}_{32} \neq \text{temp}_{31} \text{ then}
\]
\[
\text{SignalException(IntegerOverflow)}
\]
\[
\text{else}
\]
\[
\text{GPR}[rd] \leftarrow \text{temp}_{31..0}
\]
\[
\text{endif}
\]

Exceptions:
Integer Overflow

Programming Notes:
SUBU performs the same arithmetic operation but does not trap on overflow.
Floating Point Subtract

Format: SUB.S fd, fs, ft
        SUB.D fd, fs, ft

MIPS32 (MIPS I)

Purpose: To subtract FP values

Description: fd ← fs - ft

The value in FPR ft is subtracted from the value in FPR fs. The result is calculated to infinite precision, rounded according to the current rounding mode in FCSR, and placed into FPR fd. The operands and result are values in format fmt. Restrictions:

The fields fs, ft, and fd must specify FPRs valid for operands of type fmt. If they are not valid, the result is UNPREDICTABLE.

The operands must be values in format fmt; if they are not, the result is UNPREDICTABLE and the value of the operand FPRs becomes UNPREDICTABLE.

Operation:

StoreFPR (fd, fmt, ValueFPR(fs, fmt) -fmt ValueFPR(ft, fmt))

CPU Exceptions:

Coprocessor Unusable, Reserved Instruction

FPU Exceptions:

Inexact, Overflow, Underflow, Invalid Op, Unimplemented Op
Subtract Unsigned Word

<table>
<thead>
<tr>
<th>Special</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>0</th>
<th>SUBU</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

**Format:** SUBU rd, rs, rt

**MIPS32 (MIPS I)**

**Purpose:**
To subtract 32-bit integers

**Description:** rd ← rs - rt

The 32-bit word value in GPR rt is subtracted from the 32-bit value in GPR rs and the 32-bit arithmetic result is and placed into GPR rd.

No integer overflow exception occurs under any circumstances.

**Restrictions:**
None

**Operation:**

\[
\text{temp} \leftarrow \text{GPR}[rs] - \text{GPR}[rt]
\]
\[
\text{GPR}[rd] \leftarrow \text{temp}
\]

**Exceptions:**
None

**Programming Notes:**
The term “unsigned” in the instruction name is a misnomer; this operation is 32-bit modulo arithmetic that does not trap on overflow. It is appropriate for unsigned arithmetic, such as address arithmetic, or integer arithmetic environments that ignore overflow, such as C language arithmetic.
Store Word

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SW</td>
<td>base</td>
<td>rt</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>101011</td>
<td>5</td>
<td>5</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** \( \text{SW \ rt, offset(base)} \)  

**Purpose:** To store a word to memory

**Description:** \( \text{memory[base+offset] \leftarrow rt} \)

The least-significant 32-bit word of register \( rt \) is stored in memory at the location specified by the aligned effective address. The 16-bit signed \( \text{offset} \) is added to the contents of GPR \( \text{base} \) to form the effective address.

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

**Operation:**

\[
\begin{align*}
vAddr & \leftarrow \text{sign_extend(offset)} + \text{GPR}[base] \\
& \text{if } vAddr_{1..0} \neq 0^2 \text{ then} \\
& \quad \text{SignalException(AddressError)} \\
& \text{endif} \\
(pAddr, CCA) & \leftarrow \text{AddressTranslation (vAddr, DATA, STORE)} \\
\text{dataword} & \leftarrow \text{GPR}[rt] \\
\text{StoreMemory (CCA, WORD, dataword, pAddr, vAddr, DATA)}
\end{align*}
\]

**Exceptions:** TLB Refill, TLB Invalid, TLB Modified, Address Error
Store Word from Floating Point

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SWC1</td>
<td>base</td>
<td>ft</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>111001</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:**  SWC1 ft, offset(base)  

**Purpose:**  
To store a word from an FPR to memory  

**Description:** memory[base+offset] ← ft  

The low 32-bit word from FPR ft is stored in memory at the location specified by the aligned effective address. The 16-bit signed offset is added to the contents of GPR base to form the effective address.

**Restrictions:**  
An Address Error exception occurs if EffectiveAddress1..0 ≠ 0 (not word-aligned).

**Operation:**  
vAddr ← sign_extend(offset) + GPR[base]  
if vAddr1..0 ≠ 0 then  
    SignalException(AddressError)  
endif  
(pAddr, CCA) ← AddressTranslation(vAddr, DATA, STORE)  
dataword ← ValueFPR(ft, UNINTERPRETED_WORD)  
StoreMemory(CCA, WORD, dataword, pAddr, vAddr, DATA)

**Exceptions:**  
Coprocessor Unusable, Reserved Instruction, TLB Refill, TLB Invalid, TLB Modified, Address Error
Store Word from Coprocessor 2

<table>
<thead>
<tr>
<th>SWC2</th>
<th>base</th>
<th>rt</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>111010</td>
<td>5</td>
<td>5</td>
<td>16</td>
</tr>
</tbody>
</table>

**Format:** SWC2 rt, offset(base)  

**MIPS32 (MIPS I)**

**Purpose:**

To store a word from a COP2 register to memory

**Description:** memory[base+offset] ← ft

The low 32-bit word from COP2 (Coprocessor 2) register rt is stored in memory at the location specified by the aligned effective address. The 16-bit signed offset is added to the contents of GPR base to form the effective address.

**Restrictions:**

An Address Error exception occurs if EffectiveAddress1,0 ≠ 0 (not word-aligned).

**Operation:**

```
vAddr ← sign_extend(offset) + GPR[base]
if vAddr2..0 ≠ 03 then
    SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation(vAddr, DATA, STORE)
dataword ← CPR[2,rt,0]
StoreMemory(CCA, WORD, dataword, pAddr, vAddr, DATA)
```

**Exceptions:**

Coprocessor Unusable, Reserved Instruction, TLB Refill, TLB Invalid, TLB Modified, Address Error
Store Word Left

<table>
<thead>
<tr>
<th>SWL</th>
<th>base</th>
<th>rt</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>101010</td>
<td>6</td>
<td>5</td>
<td>16</td>
</tr>
</tbody>
</table>

**Format:** SWL rt, offset(base)  

**MIPS32 (MIPS I)**

**Purpose:**
To store the most-significant part of a word to an unaligned memory address

**Description:** 
memory[base+offset] ← rt

The 16-bit signed offset is added to the contents of GPR base to form an effective address (EffAddr). EffAddr is the address of the most-significant of 4 consecutive bytes forming a word (W) in memory starting at an arbitrary byte boundary.

A part of W, the most-significant 1 to 4 bytes, is in the aligned word containing EffAddr. The same number of the most-significant (left) bytes from the word in GPR rt are stored into these bytes of W.

The following figure illustrates this operation using big-endian byte ordering for 32-bit and 64-bit registers. The 4 consecutive bytes in 2..5 form an unaligned word starting at location 2. A part of W, 2 bytes, is located in the aligned word containing the most-significant byte at 2. First, SWL stores the most-significant 2 bytes of the low word from the source register into these 2 bytes in memory. Next, the complementary SWR stores the remainder of the unaligned word.

---

**Figure 3-6 Unaligned Word Store Using SWL and SWR**

Word at byte 2 in memory, big-endian byte order; each memory byte contains its own address  

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | ... |

**Memory:** Initial contents  

| E | F | G | H |

**GPR 24**  

| 0 | 1 | E | F | 4 | 5 | 6 | ... | After executing SWL $24, 2 ($0)  

| 0 | 1 | E | F | G | H | 6 | ... | Then after SWR $24, 5 ($0)

The bytes stored from the source register to memory depend on both the offset of the effective address within an aligned word—that is, the low 2 bits of the address (vAddr1..0)—and the current byte-ordering mode of the processor (big- or little-endian). The following figure shows the bytes stored for every combination of offset and byte ordering.
Figure 3-7 Bytes Stored by an SWL Instruction

Restrictions:
None

Operation:

\[ \text{vAddr } \leftarrow \text{sign} \text{-extend}\text{(offset)} + \text{GPR}\text{[base]} \]
\( (\text{pAddr, CCA}) \leftarrow \text{AddressTranslation (vAddr, DATA, STORE)} \)
\( \text{pAddr } \leftarrow \text{pAddr}_{PSIZE\text{-}1..2} \ || \ (\text{pAddr}_{1..0} \ xor \ \text{ReverseEndian}^2) \)
If BigEndianMem = 0 then
\( \text{pAddr } \leftarrow \text{pAddr}_{PSIZE\text{-}1..2} \ || \ 0^2 \)
endif
\( \text{byte } \leftarrow \text{vAddr}_{1..0} \ xor \ \text{BigEndianCPU}^2 \)
\( \text{dataword} \leftarrow 0^{24-8}\text{byte} \ || \ \text{GPR}[rt]_{31..24\text{-}8}\text{byte} \)
\( \text{StoreMemory(CCA, byte, dataword, pAddr, vAddr, DATA)} \)

Exceptions:
TLB Refill, TLB Invalid, TLB Modified, Bus Error, Address Error
Store Word Right

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SWR</td>
<td>base</td>
<td>rt</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>101110</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** SWR rt, offset(base)

**MIPS32 (MIPS I)**

**Purpose:**
To store the least-significant part of a word to an unaligned memory address

**Description:** memory[base+offset] ← rt

The 16-bit signed offset is added to the contents of GPR base to form an effective address (EffAddr). EffAddr is the address of the least-significant of 4 consecutive bytes forming a word (W) in memory starting at an arbitrary byte boundary.

A part of W, the least-significant 1 to 4 bytes, is in the aligned word containing EffAddr. The same number of the least-significant (right) bytes from the word in GPR rt are stored into these bytes of W.

The following figure illustrates this operation using big-endian byte ordering for 32-bit and 64-bit registers. The 4 consecutive bytes in 2..5 form an unaligned word starting at location 2. A part of W, 2 bytes, is contained in the aligned word containing the least-significant byte at 5. First, SWR stores the least-significant 2 bytes of the low word from the source register into these 2 bytes in memory. Next, the complementary SWL stores the remainder of the unaligned word.

**Figure 3-8 Unaligned Word Store Using SWR and SWL**
The bytes stored from the source register to memory depend on both the offset of the effective address within an aligned word—that is, the low 2 bits of the address ($vAddr_{1..0}$)—and the current byte-ordering mode of the processor (big- or little-endian). The following figure shows the bytes stored for every combination of offset and byte-ordering.

**Figure 3-9 Bytes Stored by SWR Instruction**

<table>
<thead>
<tr>
<th>Memory contents and byte offsets</th>
<th>Initial contents of Dest Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3</td>
<td>A B C D E F G H 64-bit register</td>
</tr>
<tr>
<td>i j k l (vAddr_{1..0})</td>
<td></td>
</tr>
<tr>
<td>3 2 1 0</td>
<td>most — significance — least</td>
</tr>
<tr>
<td>offset ($vAddr_{1..0}$)</td>
<td></td>
</tr>
<tr>
<td>i j k l (vAddr_{1..0})</td>
<td>most least — significance — least</td>
</tr>
<tr>
<td>i j k l (vAddr_{1..0})</td>
<td></td>
</tr>
<tr>
<td>Big-endian byte ordering</td>
<td>Little-endian byte ordering</td>
</tr>
<tr>
<td>H j k l 0</td>
<td>E F G H 32-bit register</td>
</tr>
<tr>
<td>G H k l 1</td>
<td>F G H l</td>
</tr>
<tr>
<td>F G H l 2</td>
<td>G H k l</td>
</tr>
<tr>
<td>E F G H 3</td>
<td>H j k l</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Restrictions:</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operation:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$vAddr \leftarrow \text{sign}._\text{extend}(\text{offset}) + \text{GPR[base]}$</td>
</tr>
<tr>
<td>$(pAddr, \text{CCA})\leftarrow \text{AddressTranslation} (vAddr, \text{DATA, STORE})$</td>
</tr>
<tr>
<td>$pAddr \leftarrow pAddr_{FSIZE-1..2} \</td>
</tr>
<tr>
<td>If BigEndianMem = 0 then</td>
</tr>
<tr>
<td>$pAddr \leftarrow pAddr_{FSIZE-1..2} \</td>
</tr>
<tr>
<td>endif</td>
</tr>
<tr>
<td>$byte\leftarrow vAddr_{1..0} \ xor \ BigEndianCPU^2$</td>
</tr>
<tr>
<td>dataword$\leftarrow \text{GPR[rt]_{31-8*byte}} \</td>
</tr>
<tr>
<td>StoreMemory(CCA, WORD-byte, dataword, pAddr, vAddr, DATA)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Exceptions:</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLB Refill, TLB Invalid, TLB Modified, Bus Error, Address Error</td>
</tr>
</tbody>
</table>
Synchronize Shared Memory

**SYNC**

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21 20</th>
<th>16 15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL</td>
<td>000000</td>
<td>0</td>
<td>00 0000 0000 0000 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>stype</td>
<td>001111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** SYNC (stype = 0 implied)

**Purpose:**
To order loads and stores.

**Description:**

Simple Description:
- SYNC affects only *uncached* and *cached coherent* loads and stores. The loads and stores that occur before the SYNC must be completed before the loads and stores after the SYNC are allowed to start.
- Loads are completed when the destination register is written. Stores are completed when the stored value is visible to every other processor in the system.
- SYNC is required, potentially in conjunction with SSNOP, to guarantee that memory reference results are visible across operating mode changes. For example, a SYNC is required on some implementations on entry to and exit from Debug Mode to guarantee that memory affects are handled correctly.

Detailed Description:
- When the *stype* field has a value of zero, every synchronizable load and store that occurs in the instruction stream before the SYNC instruction must be globally performed before any synchronizable load or store that occurs after the SYNC can be performed, with respect to any other processor or coherent I/O module.
- SYNC does not guarantee the order in which instruction fetches are performed. The *stype* values 1-31 are reserved; they produce the same result as the value zero.
-
Terms:

Synchronous: A load or store instruction is synchronizable if the load or store occurs to a physical location in shared memory using a virtual location with a memory access type of either uncached or cached coherent. Shared memory is memory that can be accessed by more than one processor or by a coherent I/O system module.

Performed load: A load instruction is performed when the value returned by the load has been determined. The result of a load on processor A has been determined with respect to processor or coherent I/O module B when a subsequent store to the location by B cannot affect the value returned by the load. The store by B must use the same memory access type as the load.

Performed store: A store instruction is performed when the store is observable. A store on processor A is observable with respect to processor or coherent I/O module B when a subsequent load of the location by B returns the value written by the store. The load by B must use the same memory access type as the store.

Globally performed load: A load instruction is globally performed when it is performed with respect to all processors and coherent I/O modules capable of storing to the location.

Globally performed store: A store instruction is globally performed when it is globally observable. It is globally observable when it is observable by all processors and I/O modules capable of loading from the location.

Coherent I/O module: A coherent I/O module is an Input/Output system component that performs coherent Direct Memory Access (DMA). It reads and writes memory independently as though it were a processor doing loads and stores to locations with a memory access type of cached coherent.
Restrictions:
The effect of SYNC on the global order of loads and stores for memory access types other than uncached and cached coherent is UNPREDICTABLE.

Operation:
SyncOperation(stype)

Exceptions:
None

Programming Notes:
A processor executing load and store instructions observes the order in which loads and stores using the same memory access type occur in the instruction stream; this is known as program order.

A parallel program has multiple instruction streams that can execute simultaneously on different processors. In multiprocessor (MP) systems, the order in which the effects of loads and stores are observed by other processors—the global order of the loads and store—determines the actions necessary to reliably share data in parallel programs.

When all processors observe the effects of loads and stores in program order, the system is strongly ordered. On such systems, parallel programs can reliably share data without explicit actions in the programs. For such a system, SYNC has the same effect as a NOP. Executing SYNC on such a system is not necessary, but neither is it an error.

If a multiprocessor system is not strongly ordered, the effects of load and store instructions executed by one processor may be observed out of program order by other processors. On such systems, parallel programs must take explicit actions to reliably share data. At critical points in the program, the effects of loads and stores from an instruction stream must occur in the same order for all processors. SYNC separates the loads and stores executed on the processor into two groups, and the effect of all loads and stores in one group is seen by all processors before the effect of any load or store in the subsequent group. In effect, SYNC causes the system to be strongly ordered for the executing processor at the instant that the SYNC is executed.

Many MIPS-based multiprocessor systems are strongly ordered or have a mode in which they operate as strongly ordered for at least one memory access type. The MIPS architecture also permits implementation of MP systems that are not strongly ordered; SYNC enables the reliable use of shared memory on such systems. A parallel program that does not use SYNC generally does not operate on a system that is not strongly ordered. However, a program that does use SYNC works on both types of systems. (System-specific documentation describes the actions needed to reliably share data in parallel programs for that system.)

The behavior of a load or store using one memory access type is undefined if a load or store was previously made to the same physical location using a different memory access type. The presence of a SYNC between the references does not alter this behavior.
SYNC affects the order in which the effects of load and store instructions appear to all processors; it does not generally affect the physical memory-system ordering or synchronization issues that arise in system programming. The effect of SYNC on implementation-specific aspects of the cached memory system, such as writeback buffers, is not defined. The effect of SYNC on reads or writes to memory caused by privileged implementation-specific instructions, such as CACHE, also is not defined.

```
# Processor A (writer)
# Conditions at entry:
# The value 0 has been stored in FLAG and that value is observable by B
SW R1, DATA       # change shared DATA value
LI R2, 1
SYNC              # Perform DATA store before performing FLAG store
SW R2, FLAG       # say that the shared DATA value is valid

# Processor B (reader)
LI R2, 1
1: LW R1, FLAG    # Get FLAG
BNE R2, R1, 1B    # if it says that DATA is not valid, poll again
NOP
SYNC             # FLAG value checked before doing DATA read
LW R1, DATA      # Read (valid) shared DATA value
```

Prefetch operations have no effect detectable by User-mode programs, so ordering the effects of prefetch operations is not meaningful.

The code fragments above shows how SYNC can be used to coordinate the use of shared data between separate writer and reader instruction streams in a multiprocessor environment. The FLAG location is used by the instruction streams to determine whether the shared data item DATA is valid. The SYNC executed by processor A forces the store of DATA to be performed globally before the store to FLAG is performed. The SYNC executed by processor B ensures that DATA is not read until after the FLAG value indicates that the shared data is valid.
**System Call**

### SYSCALL

#### Format: SYSCALL

#### MIPS32 (MIPS I)

#### Purpose:
To cause a System Call exception

#### Description:
A system call exception occurs, immediately and unconditionally transferring control to the exception handler.

The `code` field is available for use as software parameters, but is retrieved by the exception handler only by loading the contents of the memory word containing the instruction.

#### Restrictions:
None

#### Operation:
`SignalException(SystemCall)`

#### Exceptions:
System Call
Trap if Equal

Format: TEQ rs, rt

Purpose:
To compare GPRs and do a conditional trap

Description: if rs = rt then Trap
Compare the contents of GPR rs and GPR rt as signed integers; if GPR rs is equal to GPR rt, then take a Trap exception.
The contents of the code field are ignored by hardware and may be used to encode information for system software. To retrieve the information, system software must load the instruction word from memory.

Restrictions:
None

Operation:
if GPR[rs] = GPR[rt] then
    SignalException(Trap)
endif

Exceptions:
Trap
**Trap if Equal Immediate**

| Format: | TEQI rs, immediate |
| Purpose: | To compare a GPR to a constant and do a conditional trap |
| Description: | if rs = immediate then Trap |

Compare the contents of GPR rs and the 16-bit signed immediate as signed integers; if GPR rs is equal to immediate, then take a Trap exception.

| Restrictions: | None |
| Operation: | if GPR[rs] = sign_extend(immediate) then SignalException(Trap) endif |
| Exceptions: | Trap |
### Trap if Greater or Equal

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL</td>
<td>rs</td>
<td>rt</td>
<td>code</td>
<td>TGE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000000</td>
<td>5</td>
<td>5</td>
<td>10</td>
<td>110000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Format:

- `TGE rs, rt`

#### Purpose:

To compare GPRs and do a conditional trap

#### Description:

- `if rs ≥ rt then Trap`

Compare the contents of GPR `rs` and GPR `rt` as signed integers; if GPR `rs` is greater than or equal to GPR `rt`, then take a Trap exception.

The contents of the `code` field are ignored by hardware and may be used to encode information for system software. To retrieve the information, system software must load the instruction word from memory.

#### Restrictions:

None

#### Operation:

```c
if GPR[rs] ≥ GPR[rt] then
    SignalException(Trap)
endif
```

#### Exceptions:

- Trap
**Trap if Greater or Equal Immediate**

**TGEI**

<table>
<thead>
<tr>
<th>Format:</th>
<th>TGEI rs, immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Purpose:</strong></td>
<td>To compare a GPR to a constant and do a conditional trap</td>
</tr>
</tbody>
</table>

**Description:** if rs ≥ immediate then Trap

Compare the contents of GPR rs and the 16-bit signed immediate as signed integers; if GPR rs is greater than or equal to immediate, then take a Trap exception.

**Restrictions:**
None

**Operation:**

```plaintext
if GPR[rs] ≥ sign_extend(immediate) then
    SignalException(Trap)
endif
```

** Exceptions:**
Trap
Trap if Greater or Equal Immediate Unsigned

TGEIU

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>REGIMM</td>
<td>rs</td>
<td>TGEIU</td>
<td>immediate</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000001</td>
<td>01001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Format: TGEIU rs, immediate

Purpose:
To compare a GPR to a constant and do a conditional trap

Description: if rs ≥ immediate then Trap

Compare the contents of GPR rs and the 16-bit sign-extended immediate as unsigned integers; if GPR rs is greater than or equal to immediate, then take a Trap exception.

Because the 16-bit immediate is sign-extended before comparison, the instruction can represent the smallest or largest unsigned numbers. The representable values are at the minimum [0, 32767] or maximum [max_unsigned-32767, max_unsigned] end of the unsigned range.

Restrictions:
None

Operation:
if (0 || GPR[rs]) ≥ (0 || sign_extend(immediate)) then
    SignalException(Trap)
endif

Exceptions:
Trap
Trap if Greater or Equal Unsigned

**Format:** \texttt{TGEU rs, rt} \\

**MIPS32 (MIPS II)**

**Purpose:**
To compare GPRs and do a conditional trap

**Description:** if \( rs \geq rt \) then Trap

Compare the contents of GPR \( rs \) and GPR \( rt \) as unsigned integers; if GPR \( rs \) is greater than or equal to GPR \( rt \), then take a Trap exception.

The contents of the code field are ignored by hardware and may be used to encode information for system software. To retrieve the information, system software must load the instruction word from memory.

**Restrictions:**
None

**Operation:**
\[
\text{if } (0 || \text{GPR}[rs]) \geq (0 || \text{GPR}[rt]) \text{ then} \\
\quad \text{SignalException(Trap)} \\
\text{endif}
\]

**Exceptions:**
Trap
Probe TLB for Matching Entry

<table>
<thead>
<tr>
<th>COP0</th>
<th>CO</th>
<th>0</th>
<th>TLBP</th>
</tr>
</thead>
<tbody>
<tr>
<td>010000</td>
<td>1</td>
<td>000 0000 0000 0000 0000</td>
<td>001000</td>
</tr>
</tbody>
</table>

Format: TLBP

Purpose:
To find a matching entry in the TLB.

Description:
The Index register is loaded with the address of the TLB entry whose contents match the contents of the EntryHi register. If no TLB entry matches, the high-order bit of the Index register is set.

Restrictions:

Operation:

\[
\text{Index} \leftarrow 1 \mid | \quad \text{UNPREDICTABLE}^{31} \\
\text{for } i \text{ in } 0 \ldots \text{TLBEntries-1} \\
\quad \text{if } ((\text{TLB}[i]_{\text{VPN2}} \text{ and not } (\text{TLB}[i]_{\text{Mask}})) = \\
\quad \quad (\text{EntryHi}_{\text{VPN2}} \text{ and not } (\text{TLB}[i]_{\text{Mask}}))) \text{ and } \\
\quad \quad ((\text{TLB}[i]_{G} = 1) \text{ or } (\text{TLB}[i]_{\text{ASID}} = \text{EntryHi}_{\text{ASID}})) \text{ then} \\
\quad \quad \quad \text{Index} \leftarrow i \\
\quad \quad \text{endif} \\
\text{endfor}
\]

Exceptions:
Coprocessor Unusable
**Read Indexed TLB Entry**

<table>
<thead>
<tr>
<th>COP0</th>
<th>CO</th>
<th>EntryHi</th>
<th>EntryLo0</th>
<th>EntryLo1</th>
<th>PageMask</th>
<th>TLBR</th>
</tr>
</thead>
<tbody>
<tr>
<td>010000</td>
<td>1</td>
<td>0000000 0000000 0000000 0000000</td>
<td>000001</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** TLBR

**Purpose:**
To read an entry from the TLB.

**Description:**
The *EntryHi*, *EntryLo0*, *EntryLo1*, and *PageMask* registers are loaded with the contents of the TLB entry pointed to by the Index register. Note that the value written to the *EntryHi*, *EntryLo0*, and *EntryLo1* registers may be different from that originally written to the TLB via these registers in that:

- The value returned in the VPN2 field of the *EntryHi* register may have those bits set to zero corresponding to the one bits in the Mask field of the TLB entry (the least significant bit of VPN2 corresponds to the least significant bit of the Mask field). It is implementation dependent whether these bits are preserved or zeroed after a TLB entry is written and then read.

- The value returned in the PFN field of the *EntryLo0* and *EntryLo1* registers may have those bits set to zero corresponding to the one bits in the Mask field of the TLB entry (the least significant bit of PFN corresponds to the least significant bit of the Mask field). It is implementation dependent whether these bits are preserved or zeroed after a TLB entry is written and then read.

- The value returned in the G bit in both the *EntryLo0* and *EntryLo1* registers comes from the single G bit in the TLB entry. Recall that this bit was set from the logical AND of the two G bits in *EntryLo0* and *EntryLo1* when the TLB was written.

**Restrictions:**
The operation is **UNDEFINED** if the contents of the Index register are greater than or equal to the number of TLB entries in the processor.
Operation:

\[
i \leftarrow \text{Index} \\
\text{if } i > (\text{TLBEntries} - 1) \text{ then} \\
\quad \text{UNDEFINED} \\
\text{endif} \\
\text{PageMask}_{\text{Mask}} \leftarrow \text{TLB}[i]_{\text{Mask}} \\
\text{EntryHi} \leftarrow \text{TLB}[i]_{\text{VPN}} \quad \text{and} \quad \text{not TLB}[i]_{\text{Mask}} \quad \text{||} \quad \# \text{Masking implementation dependent} \\
\quad 0^5 \quad \text{||} \quad \text{TLB}[i]_{\text{ASID}} \\
\text{EntryLo1} \leftarrow 0^2 \quad || \quad \text{TLB}[i]_{\text{PFN1}} \quad \text{and} \quad \text{not TLB}[i]_{\text{Mask}} \quad \text{||} \quad \# \text{Masking implementation dependent} \\
\quad \text{TLB}[i]_{\text{C1}} \quad || \quad \text{TLB}[i]_{\text{D1}} \quad || \quad \text{TLB}[i]_{\text{V1}} \quad || \quad \text{TLB}[i]_{\text{G}} \\
\text{EntryLo0} \leftarrow 0^2 \quad || \quad \text{TLB}[i]_{\text{PFN0}} \quad \text{and} \quad \text{not TLB}[i]_{\text{Mask}} \quad \text{||} \quad \# \text{Masking implementation dependent} \\
\quad \text{TLB}[i]_{\text{C0}} \quad || \quad \text{TLB}[i]_{\text{D0}} \quad || \quad \text{TLB}[i]_{\text{V0}} \quad || \quad \text{TLB}[i]_{\text{G}}
\]

Exceptions:

- Coprocessor Unusable
Write Indexed TLB Entry

<table>
<thead>
<tr>
<th>COP0</th>
<th>CO</th>
<th>0</th>
<th>000 0000 0000 0000 0000</th>
<th>TLBWI</th>
</tr>
</thead>
<tbody>
<tr>
<td>010000</td>
<td>1</td>
<td>000 0000 0000 0000 0000</td>
<td>000010</td>
<td></td>
</tr>
</tbody>
</table>

**Format:** TLBWI  

**Purpose:**

To write a TLB entry indexed by the Index register.

**Description:**

The TLB entry pointed to by the Index register is written from the contents of the EntryHi, EntryLo0, EntryLo1, and PageMask registers. The information written to the TLB entry may be different from that in the EntryHi, EntryLo0, and EntryLo1 registers, in that:

- The value written to the VPN2 field of the TLB entry may have those bits set to zero corresponding to the one bits in the Mask field of the PageMask register (the least significant bit of VPN2 corresponds to the least significant bit of the Mask field). It is implementation dependent whether these bits are preserved or zeroed during a TLB write.

- The value written to the PFN0 and PFN1 fields of the TLB entry may have those bits set to zero corresponding to the one bits in the Mask field of PageMask register (the least significant bit of PFN corresponds to the least significant bit of the Mask field). It is implementation dependent whether these bits are preserved or zeroed during a TLB write.

- The single G bit in the TLB entry is set from the logical AND of the G bits in the EntryLo0 and EntryLo1 registers.

**Restrictions:**

The operation is **UNDEFINED** if the contents of the Index register are greater than or equal to the number of TLB entries in the processor.
Write Indexed TLB Entry

Operation:

\[ i \leftarrow \text{Index} \]
\[ \text{TLB}[i]_{\text{Mask}} \leftarrow \text{PageMask}_{\text{Mask}} \]
\[ \text{TLB}[i]_{\text{VPN2}} \leftarrow \text{EntryHi}_{\text{VPN2}} \quad \text{and not PageMask}_{\text{Mask}} \quad \# \text{Implementation dependent} \]
\[ \text{TLB}[i]_{\text{ASID}} \leftarrow \text{EntryHi}_{\text{ASID}} \]
\[ \text{TLB}[i]_{\text{C1}} \leftarrow \text{EntryLo1}_{\text{C}} \]
\[ \text{TLB}[i]_{\text{D1}} \leftarrow \text{EntryLo1}_{\text{D}} \]
\[ \text{TLB}[i]_{\text{V1}} \leftarrow \text{EntryLo1}_{\text{V}} \]
\[ \text{TLB}[i]_{\text{PFN1}} \leftarrow \text{EntryLo1}_{\text{PFN}} \quad \text{and not PageMask}_{\text{Mask}} \quad \# \text{Implementation dependent} \]
\[ \text{TLB}[i]_{\text{PFN0}} \leftarrow \text{EntryLo0}_{\text{PFN}} \quad \text{and not PageMask}_{\text{Mask}} \quad \# \text{Implementation dependent} \]
\[ \text{TLB}[i]_{\text{C0}} \leftarrow \text{EntryLo0}_{\text{C}} \]
\[ \text{TLB}[i]_{\text{D0}} \leftarrow \text{EntryLo0}_{\text{D}} \]
\[ \text{TLB}[i]_{\text{V0}} \leftarrow \text{EntryLo0}_{\text{V}} \]

Exceptions:

Coprocessor Unusable
## Write Random TLB Entry

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>COP0</td>
<td>CO</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010000</td>
<td>1</td>
<td>0000000000000000</td>
<td></td>
<td></td>
<td></td>
<td>TLBWR</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>19</td>
<td></td>
<td></td>
<td></td>
<td>MIPS32</td>
</tr>
</tbody>
</table>

**Format:** TLBWR

**Purpose:**

To write a TLB entry indexed by the Random register.

**Description:**

The TLB entry pointed to by the Random register is written from the contents of the EntryHi, EntryLo0, EntryLo1, and PageMask registers. The information written to the TLB entry may be different from that in the EntryHi, EntryLo0, and EntryLo1 registers, in that:

- The value written to the VPN2 field of the TLB entry may have those bits set to zero corresponding to the one bits in the Mask field of the PageMask register (the least significant bit of VPN2 corresponds to the least significant bit of the Mask field). It is implementation dependent whether these bits are preserved or zeroed during a TLB write.

- The value written to the PFN0 and PFN1 fields of the TLB entry may have those bits set to zero corresponding to the one bits in the Mask field of PageMask register (the least significant bit of PFN corresponds to the least significant bit of the Mask field). It is implementation dependent whether these bits are preserved or zeroed during a TLB write.

- The value returned in the G bit in both the EntryLo0 and EntryLo1 registers comes from the single G bit in the TLB entry. Recall that this bit was set from the logical AND of the two G bits in EntryLo0 and EntryLo1 when the TLB was written.

**Restrictions:**

The operation is **UNDEFINED** if the contents of the Index register are greater than or equal to the number of TLB entries in the processor.
### Operation:

\[
\begin{align*}
&i \leftarrow \text{Random} \\
&TLB[i]\_Mask \leftarrow \text{PageMask}_\text{Mask} \\
&TLB[i]\_\text{VPN2} \leftarrow \text{EntryHi}_\text{VPN2} \text{ and not } \text{PageMask}_\text{Mask} \quad \# \text{Implementation dependent} \\
&TLB[i]\_\text{ASID} \leftarrow \text{EntryHi}_\text{ASID} \\
&TLB[i]\_C \leftarrow \text{EntryLo}_0\_C \text{ and } \text{EntryLo}_0\_D \\
&TLB[i]\_\text{PFN1} \leftarrow \text{EntryLo}_1\_\text{PFN} \text{ and not } \text{PageMask}_\text{Mask} \quad \# \text{Implementation dependent} \\
&TLB[i]\_C_1 \leftarrow \text{EntryLo}_1\_C \\
&TLB[i]\_D_1 \leftarrow \text{EntryLo}_1\_D \\
&TLB[i]\_V_1 \leftarrow \text{EntryLo}_1\_V \\
&TLB[i]\_\text{PFN0} \leftarrow \text{EntryLo}_0\_\text{PFN} \text{ and not } \text{PageMask}_\text{Mask} \quad \# \text{Implementation dependent} \\
&TLB[i]\_C_0 \leftarrow \text{EntryLo}_0\_C \\
&TLB[i]\_D_0 \leftarrow \text{EntryLo}_0\_D \\
&TLB[i]\_V_0 \leftarrow \text{EntryLo}_0\_V
\end{align*}
\]

### Exceptions:

- Coprocessor Unusable
**TLT**

**Format:** TLT rs, rt

**Purpose:**
To compare GPRs and do a conditional trap

**Description:** if rs < rt then Trap

Compare the contents of GPR rs and GPR rt as signed integers; if GPR rs is less than GPR rt, then take a Trap exception.

The contents of the code field are ignored by hardware and may be used to encode information for system software. To retrieve the information, system software must load the instruction word from memory.

**Restrictions:**
None

**Operation:**

if GPR[rs] < GPR[rt] then
   SignalException(Trap)
endif

**Exceptions:**

Trap
TLTI

**Format:** TLTI rs, immediate

**Purpose:**
To compare a GPR to a constant and do a conditional trap

**Description:** if rs < immediate then Trap

Compare the contents of GPR rs and the 16-bit signed immediate as signed integers; if GPR rs is less than immediate, then take a Trap exception.

**Restrictions:**
None

**Operation:**
if GPR[rs] < sign_extend(immediate) then
    SignalException(Trap)
endif

**Exceptions:**
Trap
TLTIU

**Format:** TLTIU rs, immediate

**Purpose:**
To compare a GPR to a constant and do a conditional trap

**Description:** if rs < immediate then Trap

Compare the contents of GPR rs and the 16-bit sign-extended immediate as unsigned integers; if GPR rs is less than immediate, then take a Trap exception.

Because the 16-bit immediate is sign-extended before comparison, the instruction can represent the smallest or largest unsigned numbers. The representable values are at the minimum [0, 32767] or maximum [max_unsigned-32767, max_unsigned] end of the unsigned range.

**Restrictions:**
None

**Operation:**

```plaintext
if (0 || GPR[rs]) < (0 || sign_extend(immediate)) then
    SignalException(Trap)
endif
```

**Exceptions:**

Trap
**Trap if Less Than Unsigned**

**TLTU**

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL</td>
<td>rs</td>
<td>rt</td>
<td>code</td>
<td>TLTU</td>
<td>110011</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000000</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>10</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** TLTU, rs, rt

**MIPS32 (MIPS II)**

**Purpose:**
To compare GPRs and do a conditional trap

**Description:** if rs < rt then Trap

Compare the contents of GPR rs and GPR rt as unsigned integers; if GPR rs is less than GPR rt, then take a Trap exception.

The contents of the code field are ignored by hardware and may be used to encode information for system software. To retrieve the information, system software must load the instruction word from memory.

**Restrictions:**
None

**Operation:**

```c
if (0 || GPR[rs]) < (0 || GPR[rt]) then
    SignalException(Trap)
endif
```

**Exceptions:**
Trap
Trap if Not Equal

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL</td>
<td>rs</td>
<td>rt</td>
<td>code</td>
<td>TNE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000000</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>10</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** TNE rs, rt

**MIPS32 (MIPS II)**

**Purpose:**
To compare GPRs and do a conditional trap

**Description:** if rs ≠ rt then Trap

Compare the contents of GPR rs and GPR rt as signed integers; if GPR rs is not equal to GPR rt, then take a Trap exception.

The contents of the code field are ignored by hardware and may be used to encode information for system software. To retrieve the information, system software must load the instruction word from memory.

**Restrictions:**
None

**Operation:**

```plaintext
if GPR[rs] ≠ GPR[rt] then
    SignalException(Trap)
endif
```

**Exceptions:**
Trap

```plaintext
31 26 25 21 20 16 15 6 5 0
SPECIAL  rs rt code TNE
000000  6 5 5 10 6
```
**Trap if Not Equal**

<table>
<thead>
<tr>
<th>REGIMM</th>
<th>rs</th>
<th>TNEI</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>000001</td>
<td></td>
<td>0110</td>
<td></td>
</tr>
</tbody>
</table>

Format: TNEI rs, immediate

Purpose:
To compare a GPR to a constant and do a conditional trap

Description: if rs $\neq$ immediate then Trap

Compare the contents of GPR rs and the 16-bit signed immediate as signed integers; if GPR rs is not equal to immediate, then take a Trap exception.

Restrictions:
None

Operation:
if GPR[rs] $\neq$ sign_extend(immediate) then
    SignalException(Trap)
endif

Exceptions:
Trap
Floating Point Truncate to Word Fixed Point

<table>
<thead>
<tr>
<th>COP1</th>
<th>fmt</th>
<th>0</th>
<th>fs</th>
<th>fd</th>
<th>TRUNC.W</th>
</tr>
</thead>
<tbody>
<tr>
<td>010001</td>
<td></td>
<td>00000</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

Format:
- TRUNC.W.S fd, fs
- TRUNC.W.D fd, fs

MIPS32 (MIPS II)

Purpose:
To convert an FP value to 32-bit fixed point, rounding toward zero

Description:
fd ← convert_and_round(fs)

The value in FPR fs, in format fmt, is converted to a value in 32-bit word fixed point format using rounding toward zero (rounding mode 1). The result is placed in FPR fd.

When the source value is Infinity, NaN, or rounds to an integer outside the range -2^{31} to 2^{31}-1, the result cannot be represented correctly and an IEEE Invalid Operation condition exists. In this case the Invalid Operation flag is set in the FCSR. If the Invalid Operation Enable bit is set in the FCSR, no result is written to fd and an Invalid Operation exception is taken immediately. Otherwise, the default result, 2^{31}–1, is written to fd.

Restrictions:
The fields fs and fd must specify valid FPRs; fs for type fmt and fd for word fixed point; if they are not valid, the result is UNPREDICTABLE.

The operand must be a value in format fmt; if it is not, the result is UNPREDICTABLE and the value of the operand FPR becomes UNPREDICTABLE.

Operation:

StoreFPR(fd, W, ConvertFmt(ValueFPR(fs, fmt), fmt, W))
Exceptions:
Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:
Inexact, Invalid Operation, Overflow, Unimplemented Operation
Enter Standby Mode

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>COP0</td>
<td>CO</td>
<td>Implementation-Dependent Code</td>
<td>WAIT</td>
<td>100000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>19</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** \texttt{WAIT}

**Purpose:**

Wait for Event

**Description:**

The \texttt{WAIT} instruction performs an implementation-dependent operation, usually involving a lower power mode. Software may use bits 24:6 of the instruction to communicate additional information to the processor, and the processor may use this information as control for the lower power mode. A value of zero for bits 24:6 is the default and must be valid in all implementations.

The \texttt{WAIT} instruction is typically implemented by stalling the pipeline at the completion of the instruction and entering a lower power mode. The pipeline is restarted when an external event, such as an interrupt or external request occurs, and execution continues with the instruction following the \texttt{WAIT} instruction. It is implementation-dependent whether the pipeline restarts when a non-enabled interrupt is requested. In this case, software must poll for the cause of the restart. If the pipeline restarts as the result of an enabled interrupt, that interrupt is taken between the \texttt{WAIT} instruction and the following instruction (EPC for the interrupt points at the instruction following the \texttt{WAIT} instruction).

The assertion of any reset or NMI must restart the pipeline and the corresponding exception must be taken.

**Restrictions:**

The operation of the processor is \textbf{UNDEFINED} if a \texttt{WAIT} instruction is placed in the delay slot of a branch or a jump.
Operation:

Enter implementation dependent lower power mode

Exceptions:

Coprocessor Unusable Exception
**XOR**

**Format:** XOR rd, rs, rt

**Purpose:**
To do a bitwise logical Exclusive OR

**Description:**
rd ← rs XOR rt

Combine the contents of GPR rs and GPR rt in a bitwise logical Exclusive OR operation and place the result into GPR rd.

**Restrictions:**
None

**Operation:**

GPR[rd] ← GPR[rs] xor GPR[rt]

**Exceptions:**
None
Format: XORI rt, rs, immediate

Purpose:
To do a bitwise logical Exclusive OR with a constant

Description: rt ← rs XOR immediate
Combine the contents of GPR rs and the 16-bit zero-extended immediate in a bitwise logical Exclusive OR operation and place the result into GPR rt.

Restrictions:
None

Operation:
GPR[rt] ← GPR[rs] xor zero_extend(immediate)

Exceptions:
None
# Revision History

<table>
<thead>
<tr>
<th>Revision</th>
<th>Date</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.90</td>
<td>November 1, 2000</td>
<td>Internal review copy of reorganized and updated architecture documentation.</td>
</tr>
<tr>
<td>0.91</td>
<td>November 15, 2000</td>
<td>External review copy of reorganized and updated architecture documentation.</td>
</tr>
</tbody>
</table>
| 0.92     | December 15, 2000| Changes in this revision:  
  • Correct sign in description of MSUBU.  
  • Update JR and JALR instructions to reflect the changes required by MIPS16. |
| 0.95     | March 12, 2001  | Update for second external review release.                                 |