MIPS IV Instruction Set

Revision 3.2

September, 1995

Charles Price
MIPS IV Instruction Set

CPU Instruction Set

Introduction ................................................. A-1

Functional Instruction Groups .......................... A-2
  Load and Store Instructions .......................... A-2
    Delayed Loads ....................................... A-3
    CPU Loads and Stores ................................ A-4
    Atomic Update Loads and Stores ..................... A-5
    Coprocessor Loads and Stores ......................... A-5
  Computational Instructions ............................ A-6
    ALU ................................................. A-6
    Shifts ............................................. A-7
    Multiply and Divide ................................ A-8
  Jump and Branch Instructions ........................ A-8
  Miscellaneous Instructions ............................ A-9
    Exception Instructions ............................. A-9
    Serialization Instructions ........................... A-10
    Conditional Move Instructions ...................... A-10
    Prefetch .......................................... A-10
  Coprocessor Instructions ............................. A-11
  Memory Access Types .................................. A-12
    Uncached ........................................... A-12
    Cached Noncoherent ................................ A-12
    Cached Coherent ........................................ A-13
    Cached ............................................. A-13
    Mixing References with Different Access Types. .... A-13
  Description of an Instruction ...................... A-15
    Instruction mnemonic and name .................... A-15
    Instruction encoding picture ....................... A-16
    Format ............................................ A-16
    Purpose ............................................ A-16
    Description ........................................ A-16
    Restrictions ......................................... A-17
    Operation .......................................... A-17
    Exceptions ......................................... A-17
    Programming Notes, Implementation Notes .......... A-18

Operation Section Notation and Functions ............ A-18
  Pseudocode Language ................................ A-18
  Pseudocode Symbols ................................ A-18
  Pseudocode Functions ................................ A-20
  Coprocessor General Register Access Functions .... A-20
  Load and Store Memory Functions ..................... A-21
FPU Instruction Set

Introduction ......................................................... B-1

FPU Data Types .................................................... B-2
  Floating-point formats ........................................ B-3
  Normalized and Denormalized Numbers ...................... B-4
  Reserved Operand Values — Infinity and NaN ............ B-4
  Fixed-point formats ........................................... B-6

Floating-Point Registers ......................................... B-6
  Organization .................................................... B-7
  Binary Data Transfers ....................................... B-7
  Formatted Operand Layout .................................... B-9
  Implementation and Revision Register ..................... B-10
  FPU Control and Status Register — FCSR ................. B-10

Values in FP Registers ........................................ B-13

FPU Exceptions ................................................... B-14
  Precise Exception Mode ...................................... B-15
  Imprecise Exception Mode ................................... B-16
  Exception Condition Definitions ............................ B-16
    Invalid Operation exception ............................. B-17
    Division By Zero exception .............................. B-18
    Overflow exception ....................................... B-18
    Underflow exception ..................................... B-18
    Inexact exception ......................................... B-19
    Unimplemented Operation exception ..................... B-19

Functional Instruction Groups ................................. B-19
  Data Transfer Instructions ................................ B-19
  Arithmetic Instructions .................................... B-21
  Conversion Instructions .................................... B-22
  Formatted Operand Value Move Instructions ................ B-23
  Conditional Branch Instructions ......................... B-23
Miscellaneous Instructions ........................................... B-24
CPU Conditional Move .................................................. B-24
Valid Operands for FP Instructions ................................. B-24
Description of an Instruction ....................................... B-26
Operation Notation Conventions and Functions ................. B-26
Individual FPU Instruction Descriptions ......................... B-27
FPU Instruction Formats .............................................. B-95
FPU (CP1) Instruction Opcode Bit Encoding ..................... B-98
  Instruction Decode .................................................. B-98
  COP1 Instruction Class ............................................. B-98
  COP1X Instruction Class .......................................... B-99
  SPECIAL Instruction Class ........................................ B-99
Instruction Subsets of MIPS III and MIPS IV Processors .... B-99
# List of Figures

| Figure A-1. | Example Instruction Description | A-15 |
| Figure A-2. | Unaligned Doubleword Load using LDL and LDR. | A-83 |
| Figure A-3. | Unaligned Doubleword Load using LDR and LDL. | A-85 |
| Figure A-4. | Unaligned Word Load using LWL and LWR. | A-97 |
| Figure A-5. | Unaligned Word Load using LWR and LWL. | A-100 |
| Figure A-6. | Unaligned Doubleword Store with SDL and SDR | A-129 |
| Figure A-7. | Unaligned Doubleword Store with SDR and SDL | A-131 |
| Figure A-8. | Unaligned Word Store using SWL and SWR. | A-149 |
| Figure A-9. | Unaligned Word Store using SWR and SWL. | A-152 |
| Figure A-10. | CPU Instruction Formats | A-174 |
| Figure B-1. | Single-Precision Floating-Point Format (S) | B-3 |
| Figure B-2. | Double-Precision Floating-Point Format (D) | B-4 |
| Figure B-3. | Word Fixed-Point Format (W) | B-6 |
| Figure B-4. | Longword Fixed-Point Format (L) | B-6 |
| Figure B-5. | Coprocessor 1 General Registers (FGRs) | B-7 |
| Figure B-6. | Effect of FPU Word Load or Move-to Operations | B-8 |
| Figure B-7. | Effect of FPU Doubleword Load or Move-to Operations | B-8 |
| Figure B-8. | Floating-point Operand Register (FPR) Organization | B-9 |
| Figure B-9. | Single Floating Point (S) or Word Fixed (W) Operand in an FPR | B-9 |
| Figure B-10. | Double Floating Point (D) or Long Fixed (L) Operand In an FPR | B-10 |
| Figure B-11. | FPU Implementation and Revision Register | B-10 |
| Figure B-12. | MIPS I - FPU Control and Status Register (FCSR) | B-11 |
| Figure B-13. | MIPS III - FPU Control and Status Register (FCSR) | B-11 |
| Figure B-14. | MIPS IV - FPU Control and Status Register (FCSR) | B-11 |
| Figure B-15. | The Effect of FPU Operations on the Format of Values Held in FPRs. | B-14 |
| Figure B-16. | FPU Instruction Formats | B-95 |
List of Tables

Table A-1. Load/Store Operations Using Register + Offset Addressing Mode. A-3
Table A-2. Load/Store Operations Using Register + Register Addressing Mode. A-3
Table A-3. Normal CPU Load/Store Instructions A-4
Table A-4. Unaligned CPU Load/Store Instructions A-4
Table A-5. Atomic Update CPU Load/Store Instructions A-5
Table A-6. Coprocessor Load/Store Instructions A-5
Table A-7. FPU Load/Store Instructions Using Register + Register Addressing A-5
Table A-8. ALU Instructions With an Immediate Operand A-6
Table A-9. 3-Operand ALU Instructions A-7
Table A-10. Shift Instructions A-7
Table A-11. Multiply/Divide Instructions A-8
Table A-12. Jump Instructions Jumping Within a 256 Megabyte Region A-9
Table A-13. Jump Instructions to Absolute Address A-9
Table A-14. PC-Relative Conditional Branch Instructions Comparing 2 Registers A-9
Table A-15. PC-Relative Conditional Branch Instructions Comparing Against Zero A-9
Table A-16. System Call and Breakpoint Instructions A-9
Table A-17. Trap-on-Condition Instructions Comparing Two Registers A-10
Table A-18. Trap-on-Condition Instructions Comparing an Immediate A-10
Table A-19. Serialization Instructions A-10
Table A-20. CPU Conditional Move Instructions A-10
Table A-21. Prefetch Using Register + Offset Address Mode A-11
Table A-22. Prefetch Using Register + Register Address Mode A-11
Table A-23. Coprocessor Definition and Use in the MIPS Architecture A-11
Table A-24. Coprocessor Operation Instructions A-12
Table A-25. Symbols in Instruction Operation Statements A-19
Table A-26. Coprocessor General Register Access Functions A-21
Table A-27. AccessLength Specifications for Loads/Stores A-24
Table A-28. Bytes Loaded by LDL Instruction A-84
Table A-29. Bytes Loaded by LDR Instruction A-86
Table A-30. Bytes Loaded by LWL Instruction A-98
Table A-31. Bytes Loaded by LWR Instruction A-101
Table A-32. Values of Hint Field for Prefetch Instruction A-117
Table A-33. Bytes Stored by SDL Instruction A-130
Table A-34. Bytes Stored by SDR Instruction A-132
Table A-35. Bytes Stored by SWL Instruction A-150
Table A-36. Bytes Stored by SWR Instruction A-153
Table A-37. CPU Instruction Encoding - MIPS I Architecture A-177
Table A-38. CPU Instruction Encoding - MIPS II Architecture A-178
Table A-39. CPU Instruction Encoding - MIPS III Architecture A-179
Revision History

2.0 (Jan 94): First General Release

This version contained incorrect definitions for MSUB and NMSUB. It did not contain the RECIP and RSQRT instructions. It contained incomplete or erroneous information for LL, LLD, SC, SCD, SYNC, PREF, and PREFX.

All copies of this version of the document should be destroyed

2.2 (Jul 94): Mandatory Replacement of Rev 2.0

This version should probably have been 3.0 since it is a major content change. This version is issued with no known errors. It includes the late changes to the MIPS IV definition including the reintroduction of RECIP and RSQRT and the definition of the multiply-accumulate instructions as unfused (rounded) operations.

3.0 (Oct 94):

Add itemized instruction lists in the discussion of instruction functional groups.
Add a more complete description of FPU operation
Correct problems discovered with Revision 2.2.

3.1 (Jan 95):

Correct minor problems discovered with Revision 3.0.

3.2 (Sep 95):

Revise the opcode encoding tables significantly.
Correct minor problems discovered with Revision 3.1.

Changes from previous revision:

Changes are generally marked by change bars in the outer margin of the page -- just like the bar to the side of this line. Minor corrections to punctuation and spelling are neither marked with change bars nor noted in this list. Some changes in figures are not marked by change bars due to limitations of the publishing tools.

CVT.D.fmt Instruction

Change the architecture level for the CVT.D.L version of the instruction from: MIPS IV

to: MIPS III
CVT.S.fmt Instruction

Change the architecture level for the CVT.S.L version of the instruction
from: MIPS IV

to: MIPS III

LWL Instruction

In the example in Fig. A-4 the sign extension “After executing LWL $24, 2 ($0)” should be changed
from: no cng or sign ext
to: sign bit (31) extend.

The information in the tables later in the instruction description is correct.

MOVF Instruction

Change the name of the constant value in the function field
from: MOVC

to: MOVCI

There is a corresponding change in the FPU opcode encoding table in section B.12
with opcode=SPECIAL and function=MOVC, changing the value to MOVCI.

MOVF.fmt Instruction

Change the name of the constant value in the function field
from: MOVC

to: MOVCF

There is a corresponding change in the FPU opcode encoding table in section B.12
with opcode=COP1, fmt = S or D, and function=MOVC, changing the value to MOVCI.

MOVF Instruction

Change the name of the constant value in the function field
from: MOVC

to: MOVCI

There is a corresponding change in the FPU opcode encoding table in section B.12
with opcode=SPECIAL and function=MOVC, changing the value to MOVCI.

MOVT.fmt Instruction

Change the name of the constant value in the function field
from: MOVC

to: MOVCF

There is a corresponding change in the FPU opcode encoding table in section B.12
with opcode=COP1, fmt = S or D, and function=MOVC, changing the value to MOVCI.
CPU Instruction Encoding tables

Revise the presentation of the opcode encoding in section A.8 for greater clarity when considering different architecture levels or operating a MIPS III or MIPS IV processor in the MIPS II or MIPS III instruction subset modes.

There is a separate encoding table for each architecture level. There is a table of the MIPS IV encodings showing the architecture level at which each opcode was first defined and subsequently modified or extended. There is a separate table for each architecture revision I→II, II→III, and III→IV showing the changes made in that revision.

FPU Instruction Encoding tables

Revise the presentation of the opcode encoding in section B.12 for greater clarity when considering different architecture levels or operating a MIPS III or MIPS IV processor in the MIPS II or MIPS III instruction subset modes.

There is a separate encoding table for each architecture level. There is a table of the MIPS IV encodings showing the architecture level at which each opcode was first defined and subsequently modified or extended. There is a separate table for each architecture revision I→II, II→III, and III→IV showing the changes made in that revision.
A 1 Introduction

This appendix describes the instruction set architecture (ISA) for the central processing unit (CPU) in the MIPS IV architecture. The CPU architecture defines the non-privileged instructions that execute in user mode. It does not define privileged instructions providing processor control executed by the implementation-specific System Control Processor. Instructions for the floating-point unit are described in Appendix B.

The original MIPS I CPU ISA has been extended in a backward-compatible fashion three times. The ISA extensions are inclusive as the diagram illustrates; each new architecture level (or version) includes the former levels. The description of an architectural feature includes the architecture level in which the feature is (first) defined or extended. The feature is also available in all later (higher) levels of the architecture.

The practical result is that a processor implementing MIPS IV is also able to run MIPS I, MIPS II, or MIPS III binary programs without change.
The CPU instruction set is first summarized by functional group then each instruction is described separately in alphabetical order. The appendix describes the organization of the individual instruction descriptions and the notation used in them (including FPU instructions). It concludes with the CPU instruction formats and opcode encoding tables.

A 2 Functional Instruction Groups

CPU instructions are divided into the following functional groups:

- Load and Store
- ALU
- Jump and Branch
- Miscellaneous
- Co processor

A 2.1 Load and Store Instructions

Load and store instructions transfer data between the memory system and the general register sets in the CPU and the coprocessors. There are separate instructions for different purposes: transferring various sized fields, treating loaded data as signed or unsigned integers, accessing unaligned fields, selecting the addressing mode, and providing atomic memory update (read-modify-write).

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

Except for the few specialized instructions listed in Table A-4, loads and stores must access naturally aligned objects. An attempt to load or store an object at an address that is not an even multiple of the size of the object will cause an Address Error exception.

Load and store operations have been added in each revision of the architecture:

MIPS II
- 64-bit coprocessor transfers
- atomic update

MIPS III
- 64-bit CPU transfers
- unsigned word load for CPU

MIPS IV
- register + register addressing mode for FPU
Tables A-1 and A-2 tabulate the supported load and store operations and indicate the MIPS architecture level at which each operation was first supported. The instructions themselves are listed in the following sections.

Table A-1  Load/Store Operations Using Register + Offset Addressing Mode.

<table>
<thead>
<tr>
<th>Data Size</th>
<th>Load Signed</th>
<th>Load Unsigned</th>
<th>Store</th>
<th>Store</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>I</td>
<td>I</td>
<td>I</td>
<td></td>
</tr>
<tr>
<td>halfword</td>
<td>I</td>
<td>I</td>
<td>I</td>
<td></td>
</tr>
<tr>
<td>word</td>
<td>I</td>
<td>III</td>
<td>I</td>
<td>I</td>
</tr>
<tr>
<td>doubleword</td>
<td>III</td>
<td>III</td>
<td>II</td>
<td>II</td>
</tr>
<tr>
<td>unaligned word</td>
<td>I</td>
<td>I</td>
<td></td>
<td></td>
</tr>
<tr>
<td>unaligned doubleword</td>
<td>III</td>
<td>III</td>
<td></td>
<td></td>
</tr>
<tr>
<td>linked word  (atomic modify)</td>
<td>II</td>
<td>II</td>
<td></td>
<td></td>
</tr>
<tr>
<td>linked doubleword  (atomic modify)</td>
<td>III</td>
<td>III</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table A-2  Load/Store Operations Using Register + Register Addressing Mode.

<table>
<thead>
<tr>
<th>Data Size</th>
<th>Load</th>
<th>Store</th>
</tr>
</thead>
<tbody>
<tr>
<td>word</td>
<td>IV</td>
<td>IV</td>
</tr>
<tr>
<td>doubleword</td>
<td>IV</td>
<td>IV</td>
</tr>
</tbody>
</table>

A 2.1.1 Delayed Loads

The MIPS I architecture defines delayed loads; an instruction scheduling restriction requires that an instruction immediately following a load into register \( Rn \) cannot use \( Rn \) as a source register. The time between the load instruction and the time the data is available is the “load delay slot”. If no useful instruction can be put into the load delay slot, then a null operation (assembler mnemonic NOP) must be inserted.

In MIPS II, this instruction scheduling restriction is removed. Programs will execute correctly when the loaded data is used by the instruction following the load, but this may require extra real cycles. Most processors cannot actually load data quickly enough for immediate use and the processor will be forced to wait until the data is available. Scheduling load delay slots is desirable for performance reasons even when it is not necessary for correctness.
A 2.1.2 CPU Loads and Stores

There are instructions to transfer different amounts of data: bytes, halfwords, words, and doublewords. Signed and unsigned integers of different sizes are supported by loads that either sign-extend or zero-extend the data loaded into the register.

Table A-3 Normal CPU Load/Store Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
<th>Defined in</th>
</tr>
</thead>
<tbody>
<tr>
<td>LB</td>
<td>Load Byte</td>
<td>MIPS I</td>
</tr>
<tr>
<td>LBU</td>
<td>Load Byte Unsigned</td>
<td>I</td>
</tr>
<tr>
<td>SB</td>
<td>Store Byte</td>
<td>I</td>
</tr>
<tr>
<td>LH</td>
<td>Load Halfword</td>
<td>I</td>
</tr>
<tr>
<td>LHU</td>
<td>Load Halfword Unsigned</td>
<td>I</td>
</tr>
<tr>
<td>SH</td>
<td>Store Halfword</td>
<td>I</td>
</tr>
<tr>
<td>LW</td>
<td>Load Word</td>
<td>I</td>
</tr>
<tr>
<td>LWU</td>
<td>Load Word Unsigned</td>
<td>III</td>
</tr>
<tr>
<td>SW</td>
<td>Store Word</td>
<td>I</td>
</tr>
<tr>
<td>LD</td>
<td>Load Doubleword</td>
<td>III</td>
</tr>
<tr>
<td>SD</td>
<td>Store Doubleword</td>
<td>III</td>
</tr>
</tbody>
</table>

Unaligned words and doublewords can be loaded or stored in only two instructions by using a pair of special instructions. The load instructions read the left-side or right-side bytes (left or right side of register) from an aligned word and merge them into the correct bytes of the destination register. MIPS I, though it prohibits other use of loaded data in the load delay slot, permits LWL and LWR instructions targeting the same destination register to be executed sequentially. Store instructions select the correct bytes from a source register and update only those bytes in an aligned memory word (or doubleword).

Table A-4 Unaligned CPU Load/Store Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
<th>Defined in</th>
</tr>
</thead>
<tbody>
<tr>
<td>LWL</td>
<td>Load Word Left</td>
<td>MIPS I</td>
</tr>
<tr>
<td>LWR</td>
<td>Load Word Right</td>
<td>I</td>
</tr>
<tr>
<td>SWL</td>
<td>Store Word Left</td>
<td>I</td>
</tr>
<tr>
<td>SWR</td>
<td>Store Word Right</td>
<td>I</td>
</tr>
<tr>
<td>LDL</td>
<td>Load Doubleword Left</td>
<td>III</td>
</tr>
<tr>
<td>LDR</td>
<td>Load Doubleword Right</td>
<td>III</td>
</tr>
<tr>
<td>SDL</td>
<td>Store Doubleword Left</td>
<td>III</td>
</tr>
<tr>
<td>SDR</td>
<td>Store Doubleword Right</td>
<td>III</td>
</tr>
</tbody>
</table>
A 2.1.3 Atomic Update Loads and Stores

There are paired instructions, Load Linked and Store Conditional, that can be used to perform atomic read-modify-write of word and doubleword cached memory locations. These instructions are used in carefully coded sequences to provide one of several synchronization primitives, including test-and-set, bit-level locks, semaphores, and sequencers/event counts. The individual instruction descriptions describe how to use them.

Table A-5 Atomic Update CPU Load/Store Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
<th>Defined in</th>
</tr>
</thead>
<tbody>
<tr>
<td>LL</td>
<td>Load Linked Word</td>
<td>MIPS II</td>
</tr>
<tr>
<td>SC</td>
<td>Store Conditional Word</td>
<td>II</td>
</tr>
<tr>
<td>LLD</td>
<td>Load Linked Doubleword</td>
<td>III</td>
</tr>
<tr>
<td>SCD</td>
<td>Store Conditional Doubleword</td>
<td>III</td>
</tr>
</tbody>
</table>

A 2.1.4 Coprocessor Loads and Stores

These loads and stores are coprocessor instructions, however it seems more useful to summarize all load and store instructions in one place instead of listing them in the coprocessor instructions functional group.

If a particular coprocessor is not enabled, loads and stores to that processor cannot execute and will cause a Coprocessor Unusable exception. Enabling a coprocessor is a privileged operation provided by the System Control Coprocessor.

Table A-6 Coprocessor Load/Store Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
<th>Defined in</th>
</tr>
</thead>
<tbody>
<tr>
<td>LWCz</td>
<td>Load Word to Coprocessor-z</td>
<td>MIPS I</td>
</tr>
<tr>
<td>SWCz</td>
<td>Store Word from Coprocessor-z</td>
<td>I</td>
</tr>
<tr>
<td>LDCz</td>
<td>Load Doubleword to Coprocessor-z</td>
<td>II</td>
</tr>
<tr>
<td>SDCz</td>
<td>Store Doubleword from Coprocessor-z</td>
<td>II</td>
</tr>
</tbody>
</table>

Table A-7 FPU Load/Store Instructions Using Register + Register Addressing

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
<th>Defined in</th>
</tr>
</thead>
<tbody>
<tr>
<td>LWXC1</td>
<td>Load Word Indexed to Floating Point</td>
<td>MIPS IV</td>
</tr>
<tr>
<td>SWXC1</td>
<td>Store Word Indexed from Floating Point</td>
<td>IV</td>
</tr>
<tr>
<td>LDXC1</td>
<td>Load Doubleword Indexed to Floating Point</td>
<td>IV</td>
</tr>
<tr>
<td>SDXC1</td>
<td>Store Doubleword Indexed from Floating Point</td>
<td>IV</td>
</tr>
</tbody>
</table>
A 2.2 Computational Instructions

Two’s complement arithmetic is performed on integers represented in two’s complement notation. There are signed versions of add, subtract, multiply, and divide. There are add and subtract operations, called “unsigned”, that are actually modulo arithmetic without overflow detection. There are unsigned versions of multiply and divide. There is a full complement of shift and logical operations.

MIPS I provides 32-bit integers and 32-bit arithmetic. MIPS III adds 64-bit integers and provides separate arithmetic and shift instructions for 64-bit operands. Logical operations are not sensitive to the width of the register.

A 2.2.5 ALU

Some arithmetic and logical instructions operate on one operand from a register and the other from a 16-bit immediate value in the instruction word. The immediate operand is treated as signed for the arithmetic and compare instructions, and treated as logical (zero-extended to register length) for the logical instructions.

Table A-8 ALU Instructions With an Immediate Operand

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
<th>Defined in</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDI</td>
<td>Add Immediate Word</td>
<td>MIPS I</td>
</tr>
<tr>
<td>ADDIU</td>
<td>Add Immediate Unsigned Word</td>
<td>I</td>
</tr>
<tr>
<td>SLTI</td>
<td>Set on Less Than Immediate</td>
<td>I</td>
</tr>
<tr>
<td>SLTIU</td>
<td>Set on Less Than Immediate Unsigned</td>
<td>I</td>
</tr>
<tr>
<td>ANDI</td>
<td>And Immediate</td>
<td>I</td>
</tr>
<tr>
<td>ORI</td>
<td>Or Immediate</td>
<td>I</td>
</tr>
<tr>
<td>XORI</td>
<td>Exclusive Or Immediate</td>
<td>I</td>
</tr>
<tr>
<td>LUI</td>
<td>Load Upper Immediate</td>
<td>I</td>
</tr>
<tr>
<td>DADDI</td>
<td>Doubleword Add Immediate</td>
<td>III</td>
</tr>
<tr>
<td>DADDIU</td>
<td>Doubleword Add Immediate Unsigned</td>
<td>III</td>
</tr>
</tbody>
</table>
A 2.2.6 Shifts

There are shift instructions that take the shift amount from a 5-bit field in the instruction word and shift instructions that take a shift amount from the low-order bits of a general register. The instructions with a fixed shift amount are limited to a 5-bit shift count, so there are separate instructions for doubleword shifts of 0-31 bits and 32-63 bits.

Table A-10 Shift Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
<th>Defined in</th>
</tr>
</thead>
<tbody>
<tr>
<td>SLL</td>
<td>Shift Word Left Logical</td>
<td>MIPS I</td>
</tr>
<tr>
<td>SRL</td>
<td>Shift Word Right Logical</td>
<td>I</td>
</tr>
<tr>
<td>SRA</td>
<td>Shift Word Right Arithmetic</td>
<td>I</td>
</tr>
<tr>
<td>SLLV</td>
<td>Shift Word Left Logical Variable</td>
<td>I</td>
</tr>
<tr>
<td>SRLV</td>
<td>Shift Word Right Logical Variable</td>
<td>I</td>
</tr>
<tr>
<td>SRAV</td>
<td>Shift Word Right Arithmetic Variable</td>
<td>I</td>
</tr>
<tr>
<td>DSLL</td>
<td>Doubleword Shift Left Logical</td>
<td>III</td>
</tr>
<tr>
<td>DSRL</td>
<td>Doubleword Shift Right Logical</td>
<td>III</td>
</tr>
<tr>
<td>DSRA</td>
<td>Doubleword Shift Right Arithmetic</td>
<td>III</td>
</tr>
<tr>
<td>DSLL32</td>
<td>Doubleword Shift Left Logical + 32</td>
<td>III</td>
</tr>
<tr>
<td>DSRL32</td>
<td>Doubleword Shift Right Logical + 32</td>
<td>III</td>
</tr>
<tr>
<td>DSRA32</td>
<td>Doubleword Shift Right Arithmetic + 32</td>
<td>III</td>
</tr>
<tr>
<td>DSLLV</td>
<td>Doubleword Shift Left Logical Variable</td>
<td>III</td>
</tr>
<tr>
<td>DSRLV</td>
<td>Doubleword Shift Right Logical Variable</td>
<td>III</td>
</tr>
<tr>
<td>DSRAV</td>
<td>Doubleword Shift Right Arithmetic Variable</td>
<td>III</td>
</tr>
</tbody>
</table>
A 2.2.7 Multiply and Divide

The multiply and divide instructions produce twice as many result bits as is typical with other processors and they deliver their results into the HI and LO special registers. Multiply produces a full-width product twice the width of the input operands; the low half is put in LO and the high half is put in HI. Divide produces both a quotient in LO and a remainder in HI. The results are accessed by instructions that transfer data between HI/LO and the general registers.

Table A-11 Multiply/Divide Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
<th>Defined in</th>
</tr>
</thead>
<tbody>
<tr>
<td>MULT</td>
<td>Multiply Word</td>
<td>MIPS I</td>
</tr>
<tr>
<td>MULTU</td>
<td>Multiply Unsigned Word</td>
<td>I</td>
</tr>
<tr>
<td>DIV</td>
<td>Divide Word</td>
<td>I</td>
</tr>
<tr>
<td>DIVU</td>
<td>Divide Unsigned Word</td>
<td>I</td>
</tr>
<tr>
<td>DMULT</td>
<td>Doubleword Multiply</td>
<td>III</td>
</tr>
<tr>
<td>DMULTU</td>
<td>Doubleword Multiply Unsigned</td>
<td>III</td>
</tr>
<tr>
<td>DDIV</td>
<td>Doubleword Divide</td>
<td>III</td>
</tr>
<tr>
<td>DDIVU</td>
<td>Doubleword Divide Unsigned</td>
<td>III</td>
</tr>
<tr>
<td>MFHI</td>
<td>Move From HI</td>
<td>I</td>
</tr>
<tr>
<td>MTHI</td>
<td>Move To HI</td>
<td>I</td>
</tr>
<tr>
<td>MFLO</td>
<td>Move From LO</td>
<td>I</td>
</tr>
<tr>
<td>MTLO</td>
<td>Move To LO</td>
<td>I</td>
</tr>
</tbody>
</table>

A 2.3 Jump and Branch Instructions

The architecture defines PC-relative conditional branches, a PC-region unconditional jump, an absolute (register) unconditional jump, and a similar set of procedure calls that record a return link address in a general register. For convenience this discussion refers to them all as branches.

All branches have an architectural delay of one instruction. When a branch is taken, the instruction immediately following the branch instruction, in the branch delay slot, is executed before the branch to the target instruction takes place. Conditional branches come in two versions that treat the instruction in the delay slot differently when the branch is not taken and execution falls through. The “branch” instructions execute the instruction in the delay slot, but the “branch likely” instructions do not (they are said to nullify it).

By convention, if an exception or interrupt prevents the completion of an instruction occupying a branch delay slot, the instruction stream is continued by re-executing the branch instruction. To permit this, branches must be restartable; procedure calls may not use the register in which the return link is stored (usually register 31) to determine the branch target address.
Exception instructions have as their sole purpose causing an exception that will transfer control to a software exception handler in the kernel. System call and breakpoint instructions cause exceptions unconditionally. The trap instructions cause exceptions conditionally based upon the result of a comparison.
Table A-17  Trap-on-Condition Instructions Comparing Two Registers

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
<th>Defined in</th>
</tr>
</thead>
<tbody>
<tr>
<td>TGE</td>
<td>Trap if Greater Than or Equal</td>
<td>MIPS II</td>
</tr>
<tr>
<td>TGEU</td>
<td>Trap if Greater Than or Equal Unsigned</td>
<td>II</td>
</tr>
<tr>
<td>TLT</td>
<td>Trap if Less Than</td>
<td>II</td>
</tr>
<tr>
<td>TLTU</td>
<td>Trap if Less Than Unsigned</td>
<td>II</td>
</tr>
<tr>
<td>TEQ</td>
<td>Trap if Equal</td>
<td>II</td>
</tr>
<tr>
<td>TNE</td>
<td>Trap if Not Equal</td>
<td>II</td>
</tr>
</tbody>
</table>

Table A-18  Trap-on-Condition Instructions Comparing an Immediate

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
<th>Defined in</th>
</tr>
</thead>
<tbody>
<tr>
<td>TGEI</td>
<td>Trap if Greater Than or Equal Immediate</td>
<td>MIPS II</td>
</tr>
<tr>
<td>TGEIU</td>
<td>Trap if Greater Than or Equal Unsigned Immediate</td>
<td>II</td>
</tr>
<tr>
<td>TLTI</td>
<td>Trap if Less Than Immediate</td>
<td>II</td>
</tr>
<tr>
<td>TLTIU</td>
<td>Trap if Less Than Unsigned Immediate</td>
<td>II</td>
</tr>
<tr>
<td>TEQI</td>
<td>Trap if Equal Immediate</td>
<td>II</td>
</tr>
<tr>
<td>TNEI</td>
<td>Trap if Not Equal Immediate</td>
<td>II</td>
</tr>
</tbody>
</table>

A 2.4.2  Serialization Instructions

The order in which memory accesses from load and store instruction appear outside the processor executing them, in a multiprocessor system for example, is not specified by the architecture. The SYNC instruction creates a point in the executing instruction stream at which the relative order of some loads and stores is known. Loads and stores executed before the SYNC are completed before loads and stores after the SYNC can start.

Table A-19  Serialization Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
<th>Defined in</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNC</td>
<td>Synchronize Shared Memory</td>
<td>MIPS II</td>
</tr>
</tbody>
</table>

A 2.4.3  Conditional Move Instructions

Instructions were added in MIPS IV to conditionally move one CPU general register to another based on the value in a third general register.

Table A-20  CPU Conditional Move Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
<th>Defined in</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVN</td>
<td>Move Conditional on Not Zero</td>
<td>MIPS IV</td>
</tr>
<tr>
<td>MOVZ</td>
<td>Move Conditional on Zero</td>
<td>IV</td>
</tr>
</tbody>
</table>

A 2.4.4  Prefetch

There are two prefetch advisory instructions; one with register+offset addressing and the other with register+register addressing. These instructions advise that memory is likely to be used in a particular way in the near future and should be
prefetched into the cache. The PREFX instruction using register+register addressing mode is coded in the FPU opcode space along with the other operations using register+register addressing.

Table A-21  Prefetch Using Register + Offset Address Mode

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
<th>Defined in</th>
</tr>
</thead>
<tbody>
<tr>
<td>PREF</td>
<td>Prefetch Indexed</td>
<td>MIPS IV</td>
</tr>
</tbody>
</table>

Table A-22  Prefetch Using Register + Register Address Mode

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
<th>Defined in</th>
</tr>
</thead>
<tbody>
<tr>
<td>PREFX</td>
<td>Prefetch Indexed</td>
<td>MIPS IV</td>
</tr>
</tbody>
</table>

A 2.5 Coprocessor Instructions

Coprocessors are alternate execution units, with register files separate from the CPU. The MIPS architecture provides an abstraction for up to 4 coprocessor units, numbered 0 to 3. Each architecture level defines some of these coprocessors as shown in Table A-23. Coprocessor 0 is always used for system control and coprocessor 1 is used for the floating-point unit. Other coprocessors are architecturally valid, but do not have a reserved use. Some coprocessors are not defined and their opcodes are either reserved or used for other purposes.

Table A-23  Coprocessor Definition and Use in the MIPS Architecture

<table>
<thead>
<tr>
<th>coprocessor</th>
<th>I</th>
<th>II</th>
<th>III</th>
<th>IV</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Sys Control</td>
<td>Sys Control</td>
<td>Sys Control</td>
<td>Sys Control</td>
</tr>
<tr>
<td>1</td>
<td>FPU</td>
<td>FPU</td>
<td>FPU</td>
<td>FPU</td>
</tr>
<tr>
<td>2</td>
<td>unused</td>
<td>unused</td>
<td>unused</td>
<td>unused</td>
</tr>
<tr>
<td>3</td>
<td>unused</td>
<td>unused</td>
<td>not defined</td>
<td>FPU (COP 1X)</td>
</tr>
</tbody>
</table>

The coprocessors may have two register sets, coprocessor general registers and coprocessor control registers, each set containing up to thirty two registers. Coprocessor computational instructions may alter registers in either set.

System control for all MIPS processors is implemented as coprocessor 0 (CP0), the System Control Coprocessor. It provides the processor control, memory management, and exception handling functions. The CP0 instructions are specific to each CPU and are documented with the CPU-specific information.

If a system includes a floating-point unit, it is implemented as coprocessor 1 (CP1). In MIPS IV, the FPU also uses the computation opcode space for coprocessor unit 3, renamed COP1X. The FPU instructions are documented in Appendix B.
The coprocessor instructions are divided into two main groups:

- Load and store instructions that are reserved in the main opcode space.
- Coprocessor-specific operations that are defined entirely by the coprocessor.

A 2.5.1 Coprocessor Load and Store

Load and store instructions are not defined for CP0; the move to/from coprocessor instructions are the only way to write and read the CP0 registers.

The loads and stores for coprocessors are summarized in Load and Store Instructions on page A-2.

A 2.5.2 Coprocessor Operations

There are up to four coprocessors and the instructions are shown generically for coprocessor-\(z\). Within the operation main opcode, the coprocessor has further coprocessor-specific instructions encoded.

Table A-24  Coprocessor Operation Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
<th>Defined in</th>
</tr>
</thead>
<tbody>
<tr>
<td>COPz</td>
<td>Coprocessor-(z) Operation</td>
<td>MIPS I</td>
</tr>
</tbody>
</table>

A 3 Memory Access Types

MIPS systems provide a few memory access types that are characteristic ways to use physical memory and caches to perform a memory access. The memory access type is specified as a cache coherence algorithm (CCA) in the TLB entry for a mapped virtual page. The access type used for a location is associated with the virtual address, not the physical address or the instruction making the reference. Implementations without multiprocessor (MP) support provide uncached and cached accesses. Implementations with MP support provide uncached, cached noncoherent and cached coherent accesses. The memory access types use the memory hierarchy as follows:

Uncached

Physical memory is used to resolve the access. Each reference causes a read or write to physical memory. Caches are neither examined nor modified.

Cached Noncoherent

Physical memory and the caches of the processor performing the access are used to resolve the access. Other caches are neither examined nor modified.
Cached Coherent

Physical memory and all caches in the system containing a coherent copy of the physical location are used to resolve the access. A copy of a location is coherent (noncoherent) if the copy was placed in the cache by a cached coherent (cached noncoherent) access. Caches containing a coherent copy of the location are examined and/or modified to keep the contents of the location coherent. It is unpredictable whether caches holding a noncoherent copy of the location are examined and/or modified during a cached coherent access.

Cached

For early 32-bit processors without MP support, cached is equivalent to cached noncoherent. If an instruction description mentions the cached noncoherent access type, the comment applies equally to the cached access type in a processor that has the cached access type.

For processors with MP support, cached is a collective term, e.g. “cached memory” or “cached access”, that includes both cached noncoherent and cached coherent. Such a collective use does not imply that cached is an access type, it means that the statement applies equally to cached noncoherent and cached coherent access types.

A 3.1 Mixing References with Different Access Types

It is possible to have more than one virtual location simultaneously mapped to the same physical location. The memory access type used for the virtual mappings may be different, but it is not generally possible to use mappings with different access types at the same time.

A processor executing load and store instructions must observe the effect of the load and store instructions to a physical location in the order that they occur in the instruction stream (i.e. program order) for all accesses to virtual locations with the same memory access type.

If a processor executes a load or store using one access type to a physical location, the behavior of a subsequent load or store to the same location using a different memory access type is undefined unless a privileged instruction sequence is executed between the two accesses. Each implementation has a privileged implementation-specific mechanism that must be used to change the access type being used to access a location.

The memory access type of a location affects the behavior of I-fetch, load, store, and prefetch operations to the location. In addition, memory access types affect some instruction descriptions. Load linked (LL, LLD) and store conditional (SC, SCD) have defined operation only for locations with cached memory access type. SYNC affects only load and stores made to locations with uncached or cached coherent memory access types.
A 3.2 Cache Coherence Algorithms and Access Types

The memory access types are specified by implementation-specific cache coherence algorithms (CCAs) in TLB entries. Slightly different cache coherence algorithms such as “cached coherent, update on write” and “cached coherent, exclusive on write” can map to the same memory access type, in this case they both map to cached coherent. In order to map to the same access type the fundamental mechanism of both CCAs must be the same. When it affects the operation of the instruction, the instructions are described in terms of the memory access types. The load and store operations in a processor proceeds according to the specific CCA of the reference, however, and the pseudocode for load and store common functions in the section Load and Store Memory Functions on page A-21 use the CCA value rather than the corresponding memory access type.

A 3.3 Implementation-Specific Access Types

An implementation may provide memory access types other than uncached, cached noncoherent, or cached coherent. Implementation-specific documentation will define the properties of the new access types and their effect on all memory-related operations.
A 4 Description of an Instruction

The CPU instructions are described in alphabetic order. Each description contains several sections that contain specific information about the instruction. The content of the section is described in detail below. An example description is shown in Figure A-1.

Figure A-1 Example Instruction Description

A 4.1 Instruction mnemonic and name

The instruction mnemonic and name are printed as page headings for each page in the instruction description.
A 4.2 Instruction encoding picture

The instruction word encoding is shown in pictorial form at the top of the instruction description. This picture shows the values of all constant fields and the opcode names for opcode fields in upper-case. It labels all variable fields with lower-case names that are used in the instruction description. Fields that contain zeroes but are not named are unused fields that are required to be zero. A summary of the instruction formats and a definition of the terms used to describe the contents can be found in CPU Instruction Formats on page A-174.

A 4.3 Format

The assembler formats for the instruction and the architecture level at which the instruction was originally defined are shown. 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 order of extension. 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.

The assembler format is shown with literal parts of the assembler instruction in upper-case characters. The variable parts, the operands, are shown as the lower-case names of the appropriate fields in the instruction encoding picture. The architecture level at which the instruction was first defined, e.g. “MIPS I”, is shown at the right side of the page.

There can be more than one assembler format per architecture level. This is sometimes an alternate form of the instruction. 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 shows ADD.S and ADD.D.

The assembler format lines sometimes have comments to the right in parentheses to help explain variations in the formats. The comments are not a part of the assembler format.

A 4.4 Purpose

This is a very short statement of the purpose of the instruction.

A 4.5 Description

If a one-line symbolic description of the instruction is feasible, it will appear 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.

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 and status register.

A 4.6 Restrictions

This section documents the restrictions on the instruction. Most restrictions fall into one of six categories:

- The valid values for instruction fields (see floating-point ADD.fmt).
- The alignment requirements for memory addresses (see LW).
- The valid values of operands (see DADD).
- The valid operand formats (see floating-point ADD.fmt).
- The order of instructions necessary to guarantee correct execution. These ordering constraints avoid pipeline hazards for which some processors do not have hardware interlocks (see MUL).
- The valid memory access types (see LL/SC).

A 4.7 Operation

This section describes the operation of the instruction as pseudocode in a high-level language notation resembling Pascal. The purpose of this section is to describe the operation of the instruction clearly in a form with less ambiguity than prose. 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 omitted for readability.

There will be separate Operation sections for 32-bit and 64-bit processors if the operation is different. This is usually necessary because the path to memory is a different size on these processors.

See Operation Section Notation and Functions on page A-18 for more information on the formal notation.

A 4.8 Exceptions

This section lists the exceptions that can be caused by operation of the instruction. It omits exceptions that can be caused by instruction fetch, e.g. TLB Refill. It omits exceptions that can be caused by asynchronous external events, e.g. Interrupt. Although the 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 implementation dependent.
Reserved Instruction is listed for every instruction not in MIPS I because the instruction will cause this exception on a MIPS I processor. To execute a MIPS II, MIPS III, or MIPS IV instruction, the processor must both support the architecture level and have it enabled. The mechanism to do this is implementation specific.

The mechanism used to signal a floating-point unit (FPU) exception is implementation specific. Some implementations use the exception named “Floating Point”. Others use external interrupts (the Interrupt exception). This section lists Floating Point to represent all such mechanisms. The specific FPU traps possible are listed, indented, under the Floating Point entry.

The usual floating-point exception model for MIPS architecture processors is precise exceptions. However, the R8000 processor, the first implementation of the MIPS IV architecture, normally operates with imprecise floating-point exceptions. It also has a mode in which it operates with degraded floating-point performance but provides precise exceptions compatible with other MIPS processors. This is mentioned in the description of some floating-point instructions. A general description of this exception model is not included in this document. See the “MIPS R8000 Microprocessor Chip Set Users Manual” for more information.

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

A 4.9 Programming Notes, Implementation Notes

These 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.

A 5 Operation Section Notation and Functions

In an instruction description, the Operation section describes the operation performed by each instruction using a high-level language notation. The contents of the Operation section are described here. The special symbols and functions used are documented here.

A 5.1 Pseudocode Language

Each of the high-level language statements is executed in sequential order (as modified by conditional and loop constructs).

A 5.2 Pseudocode Symbols

Special symbols used in the notation are described in Table A-25.
<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>x^y</td>
<td>A y-bit string formed by y copies of the single-bit value x.</td>
</tr>
<tr>
<td>x_y..z</td>
<td>Selection of bits y through z of bit string x. Little-endian bit notation (rightmost bit is 0) is used. If y is less than z, this expression is an empty (zero length) bit string.</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>nor</td>
<td>Bit-wise logical NOR.</td>
</tr>
<tr>
<td>xor</td>
<td>Bit-wise logical XOR.</td>
</tr>
<tr>
<td>and</td>
<td>Bit-wise logical AND.</td>
</tr>
<tr>
<td>or</td>
<td>Bit-wise 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>FGR[x]</td>
<td>Floating-Point (Coprocessor unit1), general register x.</td>
</tr>
<tr>
<td>CPR[z,x]</td>
<td>Coprocessor unit z, general register x.</td>
</tr>
<tr>
<td>CCR[z,x]</td>
<td>Coprocessor unit z, control register x.</td>
</tr>
<tr>
<td>COC[z]</td>
<td>Coprocessor unit z condition signal.</td>
</tr>
<tr>
<td>BigEndianMem</td>
<td>Endianness mode as configured at chip reset (0 → Little, 1 → Big). Specifies the endianness of the memory interface (see LoadMemory and StoreMemory), and the endianness of Kernel and Supervisor mode execution.</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 effected by setting the RE bit of the Status register. Thus, ReverseEndian may be computed as (SRRE and User mode).</td>
</tr>
<tr>
<td>BigEndianCPU</td>
<td>The endianness for load and store instructions (0 → Little, 1 → Big). 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>LLbit</td>
<td>Bit of virtual state used to specify operation for instructions that provide atomic read-modify-write. It 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>
</tbody>
</table>


<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
</table>
| \( I \_m \) | This occurs as a prefix to operation description lines and functions as a label. It indicates the instruction time during which the effects of the pseudocode lines appears to occur (i.e. when the pseudocode is “executed”). 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 – during the instruction time of another instruction. When that happens, the instruction operation is written in sections labelled 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 will have the portion of the instruction operation description that writes the result register in a section labelled “\( I+1 \):”.

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 takes place in order. However, between sequences of statements for different instructions that occur “at the same time”, there is no order defined. Programs must not depend on a particular order of evaluation between such sections.

\( I+n \) | 
| \( I-n \) | 

\( \text{PC} \) | 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 \( \text{PC} \) during an instruction time. If no value is assigned to \( \text{PC} \) during an instruction time by any pseudocode statement, it is automatically incremented by 4 before the next instruction time. A taken branch assigns the target address to \( \text{PC} \) during the instruction time of the instruction in the branch delay slot.

\( \text{PSIZE} \) | The \text{SIZE}, number of bits, of Physical address in an implementation.

A 5.3 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. The functions are defined in this section.

A 5.3.1 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 functions:
Table A-26  Coprocessor General Register Access Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>COP_LW (z, rt, memword)</td>
<td>The coprocessor unit number. (z) Coprocessor general register specifier. (rt) A 32-bit word value supplied to the coprocessor. This is 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.</td>
</tr>
<tr>
<td>COP_LD (z, rt, memdouble)</td>
<td>The coprocessor unit number. (z) Coprocessor general register specifier. (rt) 64-bit doubleword value supplied to the coprocessor. This is 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.</td>
</tr>
<tr>
<td>dataword ← COP_SW (z, rt)</td>
<td>The coprocessor unit number. (z) Coprocessor general register specifier. (rt) 32-bit word value. This 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.</td>
</tr>
<tr>
<td>datadouble ← COP_SD (z, rt)</td>
<td>The coprocessor unit number. (z) Coprocessor general register specifier. (rt) 64-bit doubleword value. This 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 doubleword in coprocessor general register rt.</td>
</tr>
</tbody>
</table>

A 5.3.2 Load 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 among the bytes forming 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 description pseudocode for load and store operations, the functions shown below are used to summarize the handling of virtual addresses and accessing 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 A-27. The bytes within the addressed unit of memory (word for 32-bit processors or doubleword for 64-bit processors) which are used can be determined directly from the AccessLength and the two or three low-order bits of the address.
Translate a virtual address to a physical address and a cache coherence algorithm describing the mechanism used to resolve the memory reference. Given the virtual address \(v\text{Addr}\), and whether the reference is to Instructions or Data (\(IorD\)), find the corresponding physical address (\(p\text{Addr}\)) 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 is used to determine 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.

\[
\text{(pAddr, CCA)} \leftarrow \text{AddressTranslation (vAddr, IorD, LorS)}
\]

- \(p\text{Addr}\): Physical Address.
- \(CCA\): Cache Coherence Algorithm: the method used to access caches and memory and resolve the reference.
- \(v\text{Addr}\): Virtual Address.
- \(IorD\): Indicates whether access is for INSTRUCTION or DATA.
- \(LorS\): Indicates whether access is for LOAD or STORE.

Load a value from memory. Uses the cache and main memory as specified in the Cache Coherence Algorithm (\(CCA\)) and the sort of access (\(IorD\)) to find the contents of \(AccessLength\) memory bytes starting at physical location \(p\text{Addr}\). The data is returned in the fixed width naturally-aligned memory element (\(MemElem\)). The low-order two (or three) bits of the address and the \(AccessLength\) indicate which of the bytes within \(MemElem\) needs to be given to the processor. If the memory access type of the reference is uncached then only the referenced bytes are read from memory and valid within the memory element. If the access type is cached, and 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, the block is the entire memory element.

\[
\text{MemElem} \leftarrow \text{LoadMemory (CCA, AccessLength, pAddr, vAddr, IorD)}
\]

- \(\text{MemElem}\): 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.
- \(p\text{Addr}\): Physical Address.
- \(v\text{Addr}\): Virtual Address.
- \(IorD\): Indicates whether access is for Instructions or Data.
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, 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.

Store 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 (word for 32-bit processors, doubleword for 64-bit processors), though only the bytes that will actually be stored to memory need to be valid. The low-order two (or three) bits of pAddr and the AccessLength field indicates which of the bytes within the MemElem data should actually be stored; only these bytes in memory will be changed.

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.

Prefetch 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.
Table A-27 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>

A 5.3.3 Access Functions for Floating-Point Registers

The details of the relationship between CP1 general registers and floating-point operand registers is encapsulated in the functions included in this section. See Valid Operands for FP Instructions on page B-24 for more information.

This function returns the current logical width, in bits, of the CP1 general registers. All 32-bit processors will return “32”. 64-bit processors will return “32” when in 32-bit-CP1-register emulation mode and “64” when in native 64-bit mode.

The following pseudocode referring to the StatusFR bit is valid for all existing MIPS 64-bit processors at the time of this writing, however this is a privileged processor-specific mechanism and it may be different in some future processor.

```plaintext
SizeFGR() -- current size, in bits, of the CP1 general registers
size ← SizeFGR()
if 32_bit_processor then
    size ← 32
else
    /* 64-bit processor */
    if StatusFR = 1 then
        size ← 64
    else
        size ← 32
    endif
endif
```
This pseudocode 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()  --  Get a formatted value from an FPR.
value ← ValueFPR (fpr, fmt) /* get a formatted value from an FPR */
  if SizeFGR() = 64 then
    case fmt of
      S, W:
        value ← FGR[fpr]31..0
      D, L:
        value ← FGR[fpr]
    endcase
  elseif fpr0 = 0 then  /* fpr is valid (even), 32-bit wide FGRs */
    case fmt of
      S, W:
        value ← FGR[fpr]
      D, L:
        value ← FGR[fpr+1] || FGR[fpr]
    endcase
  else  /* undefined for odd 32-bit FGRs */
    UndefinedResult
  endif
```

This pseudocode specifies the way that 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 contains a value via StoreFPR(), it is not valid to interpret the value with ValueFPR() in a different format.

```
StoreFPR()  --  store a formatted value into an FPR.
StoreFPR(fpr, fmt, value): /* place a formatted value into an FPR */
    if SizeFGR() = 64 then /* 64-bit wide FGRs */
        case fmt of
            S, W:
                FGR[fpr] ← undefined || value
            D, L:
                FGR[fpr] ← value
        endcase
    elseif fpr = 0 then /* fpr is valid (even), 32-bit wide FGRs */
        case fmt of
            S, W:
                FGR[fpr+1] ← undefined32
                FGR[fpr] ← value
            D, L:
                FGR[fpr+1] ← value63..32
                FGR[fpr] ← value31..0
        endcase
    else /* undefined for odd 32-bit FGRs */
        UndefinedResult
    endif
```

A 5.3.4 Miscellaneous Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SyncOperation(stype)</td>
<td>stype: Type of load/store ordering to perform. Order loads and stores to synchronize shared memory. Perform the action necessary to make the effects of groups synchronizable loads and stores indicated by stype occur in the same order for all processors.</td>
</tr>
<tr>
<td>SignalException(Exception)</td>
<td>Exception The exception condition that exists. Signal an exception condition. This will result in an exception that aborts the instruction. The instruction operation pseudocode will never see a return from this function call.</td>
</tr>
<tr>
<td>UndefinedResult()</td>
<td>This function indicates that the result of the operation is undefined.</td>
</tr>
</tbody>
</table>
A 6 Individual CPU Instruction Descriptions

The user-mode CPU instructions are described in alphabetic order. See Description of an Instruction on page A-15 for a description of the information in each instruction description.
ADD

Add Word

Format:
ADD   rd, rs, rt

MIPS I

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

Description:
rd ← 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 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:
On 64-bit processors, if either GPR rt or GPR rs do not contain sign-extended 32-bit values (bits 63..31 equal), then the result of the operation is undefined.

Operation:
if (NotWordValue(GPR[rs]) or NotWordValue(GPR[rt])) then UndefinedResult() endif
temp ← GPR[rs] + GPR[rt]
if (32_bit_arithmetic_overflow) then
   SignalException(IntegerOverflow)
else
   GPR[rd] ← sign_extend(temp31..0)
endif

Exceptions:
Integer Overflow

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

**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 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 `rt`.

**Restrictions:**

On 64-bit processors, if GPR `rs` does not contain a sign-extended 32-bit value (bits 63..31 equal), then the result of the operation is undefined.

**Operation:**

```
if (NotWordValue(GPR[rs])) then UndefinedResult() endif

temp ← GPR[rs] + sign_extend(immediate)
if (32_bit_arithmetic_overflow) then
    SignalException(IntegerOverflow)
else
    GPR[rt] ← sign_extend(temp[31..0])
endif
```

**Exceptions:**

Integer Overflow

**Programming Notes:**

ADDIU performs the same arithmetic operation but, does not trap on overflow.
ADDIU

Add Immediate 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>ADDIU</td>
<td>rs</td>
<td>rt</td>
<td>immediate</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 1 0 0 1</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Format: ADDIU rt, rs, immediate

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:

On 64-bit processors, if GPR rs does not contain a sign-extended 32-bit value (bits 63..31 equal), then the result of the operation is undefined.

Operation:

if (NotWordValue(GPR[rs])) then UndefinedResult() endif

temp ← GPR[rs] + sign_extend(immediate)

GPR[rt] ← sign_extend(temp31..0)

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 arithmetic which is not signed, such as address arithmetic, or integer arithmetic environments that ignore overflow, such as “C” language arithmetic.
**Add Unsigned Word**

### Format

```
ADDU   rd, rs, rt
```

### 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:**

On 64-bit processors, if either GPR \( rt \) or GPR \( rs \) do not contain sign-extended 32-bit values (bits 63..31 equal), then the result of the operation is undefined.

**Operation:**

```
if (NotWordValue(GPR[rs]) or NotWordValue(GPR[rt])) then UndefinedResult() endif

temp ← GPR[rs] + GPR[rt]
GPR[rd] ← sign_extend(temp 31..0)
```

**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 arithmetic which is not signed, such as address arithmetic, or integer arithmetic environments that ignore overflow, such as “C” language arithmetic.
AND

AND   rd, rs, rt

Purpose: To do a bitwise logical AND.

Description: rd ← rs AND rt

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.

Restrictions: None

Operation:

GPR[rd] ← GPR[rs] and GPR[rt]

Exceptions: None
And Immediate **ANDI**

<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>ANDI</td>
<td>rs</td>
<td>rt</td>
<td>immediate</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** `ANDI rt, rs, immediate`

**MIPS I**

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

**Description:** `rt ← rs 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:**

\[ GPR[rt] \leftarrow \text{zero\_extend}(\text{immediate}) \text{ and } GPR[rs] \]

**Exceptions:** None
BEQ Branch on Equal

Format:      BEQ  rs, rt, offset
MIPS I
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: None

Operation:
I:  tgt_offset ← sign_extend(offset || 0^2)
   condition ← (GPR[rs] = GPR[rt])
I+ 1: if condition then
      PC ← PC + tgt_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 more distant addresses.
### Branch on Equal Likely \textbf{BEQL}

<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>BEQL</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** \texttt{BEQL rs, rt, offset}

**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:**
None

**Operation:**

1. \(\text{tgt\_offset} \leftarrow \text{sign\_extend}(\text{offset} \| 0^2)\)
2. \(\text{condition} \leftarrow (\text{GPR}[rs] = \text{GPR}[rt])\)
3. If condition then
   1. \(\text{PC} \leftarrow \text{PC} + \text{tgt\_offset}\)
   2. Else
      1. \(\text{NullifyCurrentInstruction}()\)

**Exceptions:**
Reserved Instruction

**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 more distant addresses.
BGEZ Branch on Greater Than or Equal to Zero

Format: \texttt{BGEZ rs, offset}

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

Description: if (rs \geq 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 \texttt{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:
None

Operation:

\begin{verbatim}
I: tgt_offset ← sign_extend(offset || 0²)
   condition ← GPR[rs] \geq 0^{GPRLEN}
I+1: if condition then
      PC ← PC + tgt_offset
   endif
\end{verbatim}

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 more distant addresses.
Branch on Greater Than or Equal to Zero and Link BGEZAL

Format: \[ \text{BGEZAL} \quad \text{rs}, \text{offset} \]

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

Description: if \( \text{rs} \geq 0 \) then 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 would continue 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.

Restrictions:

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

Operation:

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

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 more distant addresses.

---

CPU Instruction Set  
MIPS IV Instruction Set. Rev 3.2  
A-37
BGEZALL  Branch on Greater Than or Equal to Zero and Link Likely

<table>
<thead>
<tr>
<th>REGIMM</th>
<th>0 0 0 0 1</th>
<th>rs</th>
<th>BGEZALL</th>
<th>1 0 0 1 1</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
<td>16</td>
<td></td>
</tr>
</tbody>
</table>

Format:  BGEZALL  rs, offset

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 would continue 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 re-executed. The result of executing such an instruction
is undefined. This restriction permits an exception handler to resume execution by re-
executing the branch when an exception occurs in the branch delay slot.

Operation:

I:  tgt_offset ← sign_extend(offset || 0^2)
    condition ← GPR[rs] ≥ 0^{GPRLEN}
    GPR[31] ← PC + 8
    I+1: if condition then
    PC ← PC + tgt_offset
    else
    NullifyCurrentInstruction()
    endif

Exceptions:

Reserved Instruction

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 more distant addresses.
Branch on Greater Than or Equal to Zero Likely

BGEZL

Format: BGEZL rs, offset

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 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:
None

Operation:

1: tgt_offset ← sign_extend(offset || 0^2)
   condition ← GPR[rs] ≥ 0^GPRLEN
1+ 1: if condition then
   PC ← PC + tgt_offset
else
   NullifyCurrentInstruction()
endif

Exceptions:
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 more distant addresses.
BGTZ

Branch on Greater Than Zero

Format: BGTZ 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 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:
None

Operation:

I: tgt_offset ← sign_extend(offset || 0^2)
   condition ← GPR[rs] > 0^{GPRLEN}
   I + 1: if condition then
      PC ← PC + tgt_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 more distant addresses.
Branch on Greater Than Zero Likely BGTZL

<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>BGTZL</td>
<td>rs</td>
<td>0</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 0 1 1 1</td>
<td>0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** BGTZL rs, offset

**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:** None

**Operation:**

1: tgt_offset ← sign_extend(offset || 0^2)
   condition ← GPR[rs] > 0^GPRLEN
1 + 1: if condition then
   PC ← PC + tgt_offset
else
   NullifyCurrentInstruction()
endif

**Exceptions:**

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 more distant addresses.
**BLEZ**

*Branch on Less Than or Equal to 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>BLEZ</td>
<td>rs</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>offset</td>
<td></td>
</tr>
</tbody>
</table>

| 6 | 5 | 5 | 16 |

**Format:**

```
BLEZ   rs, offset
```

**MIPS I**

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

**Description:**

\[ \text{if } (rs \leq 0) \text{ then branch} \]

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 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:**

None

**Operation:**

```
I:   tgt_offset ← sign_extend(offset || 0^2)
    condition ← GPR[rs] \leq 0^{GPRLEN}
I+1: if condition then
    PC ← PC + tgt_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 more distant addresses.
Branch on Less Than or Equal to Zero Likely

**Format:**
BLEZL  rs, offset

**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 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:**
None

**Operation:**
I: tgt_offset ← sign_extend(offset || 0^2)
   condition ← GPR[rs] ≤ 0^GPRLEN
I+1: if condition then
     PC ← PC + tgt_offset
   else
     NullifyCurrentInstruction()
   endif

**Exceptions:**
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 more distant addresses.
BLTZ

Branch on Less Than Zero

Format: \texttt{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 \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 zero (sign bit is 1), branch to the effective target address after the instruction in the delay slot is executed.

Restrictions: None

Operation:

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

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 more distant addresses.
Branch on Less Than Zero And Link BLTZAL

Format:    BLTZAL   rs, offset
Purpose:   To test a GPR then do a PC-relative conditional procedure call.

Description:  if (rs < 0) then procedure_call

Place the return address link in GPR 31. The return link is the address of the second
instruction following the branch (not the branch itself), where execution would
continue 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, 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 re-executed. The result of executing such an instruction
is undefined. This restriction permits an exception handler to resume execution by re-
executing the branch when an exception occurs in the branch delay slot.

Operation:

I: tgt_offset ← sign_extend(offset || 0^2)
    condition ← GPR[rs] < 0^GPRLEN
    GPR[31] ← PC + 8
    I + 1:if condition then
        PC ← PC + tgt_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 more distant addresses.
BLTZALL Branch on Less Than Zero And Link Likely

Format: \texttt{BLTZALL \ rs, offset}

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 \texttt{procedure\_call\_likely}

Place the return address link in GPR 31. The return link is the address of the second instruction following the branch (not the branch itself), where execution would continue 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, in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR \texttt{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 \texttt{rs}, because such an instruction does not have the same effect when re-executed. The result of executing such an instruction is undefined. This restriction permits an exception handler to resume execution by re-executing the branch when an exception occurs in the branch delay slot.

Operation:

\begin{verbatim}
I:  tgt_offset ← sign_extend(offset || 0^2)
    condition ← GPR[rs] < 0^{GPRLEN}
    GPR[31] ← PC + 8
    I+ 1:
    if condition then
        PC ← PC + tgt_offset
    else
        NullifyCurrentInstruction()
    endif
\end{verbatim}

Exceptions:

Reserved Instruction

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 more distant addresses.
Branch on Less Than Zero Likely \textbf{BLTZL}

| Format: | BLTZ rs, offset \hspace{1cm} 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:
None

Operation:

\[
1: \quad \text{tgt\_offset} \leftarrow \text{sign\_extend}(\text{offset} \mid 0^2) \\
\text{condition} \leftarrow \text{GPR}[rs] < 0^\text{GPRLEN} \\
1+1: \quad \text{if condition then} \\
\quad \text{PC} \leftarrow \text{PC} + \text{tgt\_offset} \\
\text{else} \\
\quad \text{NullifyCurrentInstruction()} \\
\text{endif}
\]

Exceptions:
Reserved Instruction

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 more distant addresses.
BNE 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>rs</td>
<td>rt</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 0 1 0 1</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:**

BNE rs, rt, offset

**MIPS I**

**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 not equal, branch to the effective target address after the instruction in the delay slot is executed.

**Restrictions:**

None

**Operation:**

I:  tgt_offset ← sign_extend(offset || 0^2)
    condition ← (GPR[rs] ≠ GPR[rt])
    I+1: if condition then
          PC ← PC + tgt_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 more distant addresses.
Branch on Not Equal Likely

**BNEL**

<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>BNEL</td>
<td>rs</td>
<td>rt</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:**

`BNEL rs, rt, offset`

**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:**

None

**Operation:**

1: `tgt_offset ← sign_extend(offset || 0^2)`
   condition ← (GPR[rs] ≠ GPR[rt])
1+ 1: if condition then
   PC ← PC + tgt_offset
   else
   NullifyCurrentInstruction()
endif

**Exceptions:**

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 more distant addresses.
BREAK

MIPS I

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
Coprocessor Operation **COPz**

<table>
<thead>
<tr>
<th>31</th>
<th>26 25</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>COPz</td>
<td>cop_fun</td>
<td></td>
</tr>
<tr>
<td>0 1 0 0 z z</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Format:  

- COP0 cop_fun  
- COP1 cop_fun  
- COP2 cop_fun  
- COP3 cop_fun

Purpose: To execute a coprocessor instruction.

Description:  
The coprocessor operation specified by *cop_fun* is performed by coprocessor unit *zz*.  
Details of coprocessor operations must be found in the specification for each coprocessor.

Each MIPS architecture level defines up to 4 coprocessor units, numbered 0 to 3 (see Coprocessor Instructions on page A-11). The opcodes corresponding to coprocessors that are not defined by an architecture level may be used for other instructions.

Restrictions:  
Access to the coprocessors is controlled by system software. Each coprocessor has a “coprocessor usable” bit in the System Control coprocessor. The usable bit must be set for a user program to execute a coprocessor instruction. If the usable bit is not set, an attempt to execute the instruction will result in a Coprocessor Unusable exception. An unimplemented coprocessor must never be enabled. The result of executing this instruction for an unimplemented coprocessor when the usable bit is set, is undefined.

See specification for the specific coprocessor being programmed.

Operation:  

CoprocessorOperation (z, cop_fun)

Exceptions:  
- Reserved Instruction  
- Coprocessor Unusable  
- Coprocessor interrupt or Floating-Point Exception (CP1 only for some processors)
DADD Doubleword Add

Format:        DADD   rd, rs, rt
Purpose:       To add 64-bit integers. If overflow occurs, then trap.
Description:   rd ← rs + rt
               The 64-bit doubleword value in GPR rt is added to the 64-bit value in GPR rs to
               produce a 64-bit result. If the addition results in 64-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 64-bit result is placed into GPR rd.

Restrictions:  None

Operation:     64-bit processors
               temp ← GPR[rs] + GPR[rt]
               if (64_bit_arithmetic_overflow) then
                   SignalException(IntegerOverflow)
               else
                   GPR[rd] ← temp
               endif

Exceptions:    Integer Overflow
               Reserved Instruction

Programming Notes:
               DADDU performs the same arithmetic operation but, does not trap on overflow.
Doubleword Add Immediate \textbf{DADDI}

\begin{tabular}{|c|c|c|c|}
\hline
31 & 26 & 25 & 21 20 16 15 0 \\
\hline
DADDI & rs & rt & immediate \\
0 1 1 0 0 0 & 6 & 5 & 5 & 16 \\
\hline
\end{tabular}

Format: \textbf{DADDI} \textbf{rt, rs, immediate} \hspace{1cm} \textbf{MIPS III}

Purpose: To add a constant to a 64-bit integer. If overflow occurs, then trap.

Description: \textbf{rt} \leftarrow \textbf{rs} + \textbf{immediate}

The 16-bit signed \textit{immediate} is added to the 64-bit value in GPR \textit{rs} to produce a 64-bit result. If the addition results in 64-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 64-bit result is placed into GPR \textit{rt}.

Restrictions:
None

Operation: 64-bit processors

\begin{verbatim}
temp ← GPR[rs] + sign_extend(immediate)
if (64_bit_arithmetic_overflow) then
    SignalException(IntegerOverflow)
else
    GPR[rt] ← temp
endif
\end{verbatim}

Exceptions:
Integer Overflow
Reserved Instruction

Programming Notes:
\textbf{DADDIU} performs the same arithmetic operation but, does not trap on overflow.
DADDIU Doubleword Add 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>DADDIU</td>
<td>rs</td>
<td>rt</td>
<td>immediate</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 1 0 0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| 6 | 5 | 5 | 16 |

Format: DADDIU  rt, rs, immediate
Purpose: To add a constant to a 64-bit integer.

Description: rt ← rs + immediate
The 16-bit signed immediate is added to the 64-bit value in GPR rs and the 64-bit arithmetic result is placed into GPR rt.
No Integer Overflow exception occurs under any circumstances.

Restrictions:
None

Operation: 64-bit processors
GPR[rt] ← GPR[rs] + sign_extend(immediate)

Exceptions:
Reserved Instruction

Programming Notes:
The term “unsigned” in the instruction name is a misnomer; this operation is 64-bit modulo arithmetic that does not trap on overflow. It is appropriate for arithmetic which is not signed, such as address arithmetic, or integer arithmetic environments that ignore overflow, such as “C” language arithmetic.
### Doubleword Add Unsigned DADDU

<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>DADDU</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0 0 0 0 0</td>
<td>1 0 1 1 0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:**
DADDU   rd, rs, rt       
**MIPS III**

**Purpose:**
To add 64-bit integers.

**Description:**
\[ rd \leftarrow rs + rt \]

The 64-bit doubleword value in GPR \( rt \) is added to the 64-bit value in GPR \( rs \) and the 64-bit arithmetic result is placed into GPR \( rd \).

No Integer Overflow exception occurs under any circumstances.

**Restrictions:**
None

**Operation:**
64-bit processors
\[ \text{GPR}[rd] \leftarrow \text{GPR}[rs] + \text{GPR}[rt] \]

**Exceptions:**
Reserved Instruction

**Programming Notes:**
The term “unsigned” in the instruction name is a misnomer; this operation is 64-bit modulo arithmetic that does not trap on overflow. It is appropriate for arithmetic which is not signed, such as address arithmetic, or integer arithmetic environments that ignore overflow, such as “C” language arithmetic.
**DDIV**

### Doubleword Divide

<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>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DDIV</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:**

```
DDIV   rs, rt
```

**Purpose:**

To divide 64-bit signed integers.

**Description:**

```
(LO, HI) ← rs / rt
```

The 64-bit doubleword in GPR rs is divided by the 64-bit doubleword in GPR rt, treating both operands as signed values. The 64-bit quotient is placed into special register LO and the 64-bit remainder is placed into special register HI.

No arithmetic exception occurs under any circumstances.

**Restrictions:**

If either of the two preceding instructions is MFHI or MFLO, the result of the MFHI or MFLO is undefined. Reads of the HI or LO special registers must be separated from subsequent instructions that write to them by two or more other instructions.

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

**Operation:**

64-bit processors

1-2; 1-1:  
```
LO, HI ← undefined
```

I:  
```
LO ← GPR[rs] div GPR[rt]
HI ← GPR[rs] mod GPR[rt]
```

**Exceptions:**

Reserved Instruction

**Programming Notes:**

See the Programming Notes for the DIV instruction.
Doubleword Divide Unsigned \texttt{DDIVU}

<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>6</th>
<th>5</th>
<th>10</th>
<th>0</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Format: \texttt{DDIVU} \ rs, \ rt

Purpose: To divide 64-bit unsigned integers.

Description: \((\text{LO, HI}) \leftarrow \text{rs} / \text{rt}\)

The 64-bit doubleword in GPR \(\text{rs}\) is divided by the 64-bit doubleword in GPR \(\text{rt}\), treating both operands as unsigned values. The 64-bit quotient is placed into special register \(\text{LO}\) and the 64-bit remainder is placed into special register \(\text{HI}\).

No arithmetic exception occurs under any circumstances.

Restrictions:

If either of the two preceding instructions is \texttt{MFHI} or \texttt{MFLO}, the result of the \texttt{MFHI} or \texttt{MFLO} is undefined. Reads of the \(\text{HI}\) or \(\text{LO}\) special registers must be separated from subsequent instructions that write to them by two or more other instructions.

If the divisor in GPR \(\text{rt}\) is zero, the arithmetic result value is undefined.

Operation: 64-bit processors

\begin{align*}
\text{I-2:}, \text{I-1:} & : \hspace{1cm} \text{LO, HI} \leftarrow \text{undefined} \\
\text{I:} & : \hspace{1cm} \text{LO} \leftarrow (0 || \text{GPR}[\text{rs}]) \div (0 || \text{GPR}[\text{rt}]) \\
& : \hspace{1cm} \text{HI} \leftarrow (0 || \text{GPR}[\text{rs}]) \mod (0 || \text{GPR}[\text{rt}])
\end{align*}

Exceptions:

Reserved instruction

Programming Notes:

See the Programming Notes for the \texttt{DIV} instruction.
DIV

Divide Word

<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>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL</td>
<td>0 0 0 0 0 0</td>
<td>rs</td>
<td>rt</td>
<td>0 0 0 0 0 0 0 0</td>
<td>DIV</td>
<td>0 1 1 0 1 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Format: \( \text{DIV} \ rs, \ rt \)

Purpose: To divide 32-bit signed integers.

Description:

\[
(\text{LO}, \text{HI}) \leftarrow \frac{\text{rs}}{\text{rt}}
\]

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:

On 64-bit processors, if either GPR \( rt \) or GPR \( rs \) do not contain sign-extended 32-bit values (bits 63..31 equal), then the result of the operation is undefined.

If either of the two preceding instructions is \( \text{MFHI} \) or \( \text{MFLO} \), the result of the \( \text{MFHI} \) or \( \text{MFLO} \) is undefined. Reads of the \( HI \) or \( LO \) special registers must be separated from subsequent instructions that write to them by two or more other instructions.

If the divisor in GPR \( rt \) is zero, the arithmetic result value is undefined.

Operation:

\[
\text{if} \ (\text{NotWordValue}(\text{GPR}[rs]) \text{ or NotWordValue}(\text{GPR}[rt])) \text{ then UndefinedResult() endif}
\]

\[
\text{I - 2, I - 1:} \quad \text{LO, HI} \leftarrow \text{undefined}
\]

\[
\text{I:} \quad q \leftarrow \text{GPR}[rs]_{31..0} \text{ div GPR}[rt]_{31..0}
\]

\[
\text{LO} \leftarrow \text{sign\_extend}(q_{31..0})
\]

\[
\text{r} \leftarrow \text{GPR}[rs]_{31..0} \mod \text{GPR}[rt]_{31..0}
\]

\[
\text{HI} \leftarrow \text{sign\_extend}(r_{31..0})
\]

Exceptions:

None

Programming Notes:

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 will wait (interlock) 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.
No arithmetic exception occurs under any circumstances. If divide-by-zero or overflow conditions should be 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, 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 one is detected.
**DIVU**

Divide 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>6</th>
<th>5</th>
<th>10</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL</td>
<td>rs</td>
<td>rt</td>
<td>00000000</td>
<td>011011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Format: \texttt{DIVU} rs, rt \quad \textbf{MIPS I}

Purpose: To divide 32-bit unsigned 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 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:

- On 64-bit processors, if either GPR \( rt \) or GPR \( rs \) do not contain sign-extended 32-bit values (bits 63..31 equal), then the result of the operation is undefined.

- If either of the two preceding instructions is MFHI or MFLO, the result of the MFHI or MFLO is undefined. Reads of the \( HI \) or \( LO \) special registers must be separated from subsequent instructions that write to them, like this one, by two or more other instructions.

- If the divisor in GPR \( rt \) is zero, the arithmetic result is undefined.

Operation:

\[
\begin{align*}
&\text{if (NotWordValue(GPR[rs]) or NotWordValue(GPR[rt])) then UndefinedResult() endif} \\
&I - 2: , I - 1: \quad \text{LO, HI} \leftarrow \text{undefined} \\
&I: \quad q \leftarrow (0 || \text{GPR[rs]}_{31..0}) \text{ div } (0 || \text{GPR[rt]}_{31..0}) \\
&\quad \text{LO} \leftarrow \text{sign\_extend}(q_{31..0}) \\
&\quad r \leftarrow (0 || \text{GPR[rs]}_{31..0}) \text{ mod } (0 || \text{GPR[rt]}_{31..0}) \\
&\quad \text{HI} \leftarrow \text{sign\_extend}(r_{31..0})
\end{align*}
\]

Exceptions:

None

Programming Notes:

See the Programming Notes for the DIV instruction.
## Doubleword Multiply

### DMULT

#### Format:
- DMULT rs, rt

#### Purpose:
To multiply 64-bit signed integers.

#### Description:
\[(LO, HI) \leftarrow rs \times rt\]

The 64-bit doubleword value in GPR rt is multiplied by the 64-bit value in GPR rs, treating both operands as signed values, to produce a 128-bit result. The low-order 64-bit doubleword of the result is placed into special register \(LO\), and the high-order 64-bit doubleword is placed into special register \(HI\).

No arithmetic exception occurs under any circumstances.

#### Restrictions:
If either of the two preceding instructions is MFHI or MFLO, the result of the MFHI or MFLO is undefined. Reads of the \(HI\) or \(LO\) special registers must be separated from subsequent instructions that write to them by two or more other instructions.

#### Operation:
**64-bit processors**

1. \(31::22, 21::12:LO, HI \leftarrow \text{undefined}\)
2. \(1:: \text{prod} \leftarrow \text{GPR}[rs] \times \text{GPR}[rt]\)
3. \(LO \leftarrow \text{prod}_{63..0}\)
4. \(HI \leftarrow \text{prod}_{127..64}\)

#### Exceptions:
- Reserved Instruction

#### 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 will wait (interlock) 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.
DMULTU

Doubleword Multiply Unsigned

Format:
DMULTU rs, rt

Purpose:
To multiply 64-bit unsigned integers.

Description:
(LO, HI) ← rs × rt

The 64-bit doubleword value in GPR rt is multiplied by the 64-bit value in GPR rs, 
treating both operands as unsigned values, to produce a 128-bit result. The low-order 
64-bit doubleword of the result is placed into special register LO, and the high-order 
64-bit doubleword is placed into special register HI.

No arithmetic exception occurs under any circumstances.

Restrictions:
If either of the two preceding instructions is MFHI or MFLO, the result of the MFHI or 
MFLO is undefined. Reads of the HI or LO special registers must be separated from 
subsequent instructions that write to them by two or more other instructions.

Operation: 64-bit processors

I-2, I-1: LO, HI ← undefined
I: prod ← (0 || GPR[rs]) * (0 || GPR[rt])
   LO ← prod_{63..0}
   HI ← prod_{127..64}

Exceptions:
Reserved Instruction

A-62  MIPS IV Instruction Set. Rev 3.2  CPU Instruction Set
**Doubleword Shift Left Logical**

**DSLL**

<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>0</td>
<td>0</td>
<td>0</td>
<td>rt</td>
<td>rd</td>
<td>sa</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Format:**

`DSLL   rd, rt, sa`

**Purpose:**

To left shift a doubleword by a fixed amount — 0 to 31 bits.

**Description:**

`rd ← rt << sa`

The 64-bit doubleword contents of GPR `rt` are shifted left, inserting zeros into the emptied bits; the result is placed in GPR `rd`. The bit shift count in the range 0 to 31 is specified by `sa`.

**Restrictions:**

None

**Operation:**

64-bit processors

`s ← 0 || sa`

`GPR[rd] ← GPR[rt]_{63-s}..0 || 0^s`

**Exceptions:**

Reserved Instruction
DSLL32 Doubleword Shift Left Logical Plus 32

<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>rt</td>
<td>rd</td>
<td>sa</td>
<td>DSLL32</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Format: \texttt{DSLL32 rd, rt, sa} \quad \textbf{MIPS III}

Purpose: To left shift a doubleword by a fixed amount — 32 to 63 bits.

Description: \[ rd \leftarrow rt \ll (sa+32) \]

The 64-bit doubleword contents of GPR \( rt \) are shifted left, inserting zeros into the emptied bits; the result is placed in GPR \( rd \). The bit shift count in the range 32 to 63 is specified by \( sa+32 \).

Restrictions: None

Operation: 64-bit processors
\[
\begin{align*}
    s & \leftarrow 1 || sa \quad /\!/ 32+sa /\!\\
    \text{GPR}[rd] & \leftarrow \text{GPR}[rt]_{(63-s)..0} || 0^s
\end{align*}
\]

Exceptions:
Reserved Instruction
Doubleword Shift Left Logical Variable

<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>26</td>
<td>25</td>
<td>21</td>
<td>20</td>
<td>16</td>
<td>15</td>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td></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>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

SPECIAL  rs  rt  rd  0 0 0 0 0 0 0 0 0 0 0 0 0 0 DSLLV

Format: DSLLV rd, rt, rs
Purpose: To left shift a doubleword by a variable number of bits.
Description: rd ← rt << rs

The 64-bit doubleword contents of GPR rt are shifted left, inserting zeros into the emptied bits; the result is placed in GPR rd. The bit shift count in the range 0 to 63 is specified by the low-order six bits in GPR rs.

Restrictions:
None

Operation: 64-bit processors
s ← 0 || GPR[rs]5..0
GPR[rd] ← GPR[rt](63–s)..0 || 0^s

Exceptions:
Reserved Instruction
### DSRA

**Doubleword Shift Right Arithmetic**

<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>rt</td>
<td>rd</td>
<td>sa</td>
<td>DSRA</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Format:**

```
DSRA   rd, rt, sa
```

**MIPS III**

**Purpose:**

To arithmetic right shift a doubleword by a fixed amount — 0 to 31 bits.

**Description:**

```
rd ← rt >> sa      (arithmetic)
```

The 64-bit doubleword contents of GPR `rt` are shifted right, duplicating the sign bit (63) into the emptied bits; the result is placed in GPR `rd`. The bit shift count in the range 0 to 31 is specified by `sa`.

**Restrictions:**

None

**Operation:** 64-bit processors

```
s ← 0 || sa
GPR[rd] ← (GPR[rt]_63)_s || GPR[rt]_63..s
```

**Exceptions:**

Reserved Instruction
Doubleword Shift Right Arithmetic Plus 32 DSRA32

Format: DSRA32 rd, rt, sa

MIPS III

Purpose: To arithmetic right shift a doubleword by a fixed amount — 32-63 bits.

Description: rd ← rt >> (sa+32) (arithmetic)

The doubleword contents of GPR rt are shifted right, duplicating the sign bit (63) into the emptied bits; the result is placed in GPR rd. The bit shift count in the range 32 to 63 is specified by sa+32.

Restrictions:
None

Operation: 64-bit processors

s ← 1 || sa /* 32+sa */
GPR[rd] ← (GPR[rt]_{63})^s || GPR[rt]_{63..s}

Exceptions:
Reserved Instruction
**DSRAV**

Doubleword Shift Right Arithmetic Variable

| 31 | 26 | 25 | 21 | 20 | 16 | 15 | 11 | 10 | 6 | 5 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|
| SPECIAL | rs | rt | rd | 0 | 00000 | DSRAV | 010111 |

**Format:**

```
DSRAV   rd, rt, rs
```

**MIPS III**

**Purpose:**

To arithmetic right shift a doubleword by a variable number of bits.

**Description:**

```
rd ← rt >> rs       (arithmetic)
```

The doubleword contents of GPR `rt` are shifted right, duplicating the sign bit (63) into the emptied bits; the result is placed in GPR `rd`. The bit shift count in the range 0 to 63 is specified by the low-order six bits in GPR `rs`.

**Restrictions:**

None

**Operation:** 64-bit processors

```
s ← GPR[rs]_5..0
GPR[rd]← (GPR[rt]_63)^s || GPR[rt]_63..s
```

**Exceptions:**

Reserved Instruction
Doubleword Shift Right Logical DSRL

### Format:

```
DSRL   rd, rt, sa
```

### MIPS III

### Purpose:
To logical right shift a doubleword by a fixed amount — 0 to 31 bits.

### Description:

```
rd ← rt >> sa      (logical)
```

The doubleword contents of GPR `rt` are shifted right, inserting zeros into the emptied bits; the result is placed in GPR `rd`. The bit shift count in the range 0 to 31 is specified by `sa`.

### Restrictions:

None

### Operation:

64-bit processors

```
s ← 0 || sa
GPR[rd] ← 0^s || GPR[rt]_{63..s}
```

### Exceptions:

Reserved Instruction
## DSRL32

### Doubleword Shift Right Logical Plus 32

<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>rt</td>
<td>rd</td>
<td>sa</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**Format:**

DSRL32   rd, rt, sa

**Purpose:**

To logical right shift a doubleword by a fixed amount — 32 to 63 bits.

**Description:**

\[
\text{rd} \leftarrow \text{rt} \gg (sa+32) \quad \text{(logical)}
\]

The 64-bit doubleword contents of GPR \( rt \) are shifted right, inserting zeros into the emptied bits; the result is placed in GPR \( rd \). The bit shift count in the range 32 to 63 is specified by \( sa+32 \).

**Restrictions:**

None

**Operation:**

64-bit processors

\[
\text{s} \leftarrow 1 || sa \quad /\!\!/ 32+sa */\!
\]

\[
\text{GPR}[rd] \leftarrow 0^a || \text{GPR}[rt]_{63..s}
\]

**Exceptions:**

Reserved Instruction
Doubleword Shift Right Logical Variable **DSRLV**

Format: \[ \text{DSRLV} \quad \text{rd, rt, rs} \]

Purpose: To logical right shift a doubleword by a variable number of bits.

Description: \[ \text{rd} \leftarrow \text{rt} \gg \text{rs} \] (logical)

The 64-bit doubleword contents of GPR \text{rt} are shifted right, inserting zeros into the emptied bits; the result is placed in GPR \text{rd}. The bit shift count in the range 0 to 63 is specified by the low-order six bits in GPR \text{rs}.

Restrictions:
None

Operation: 64-bit processors

\[ s \leftarrow \text{GPR}[\text{rs}]_{5..0} \]
\[ \text{GPR}[\text{rd}] \leftarrow 0^s || \text{GPR}[\text{rt}]_{63..s} \]

Exceptions:
Reserved Instruction
DSUB Doubleword Subtract

Format:

\[
\text{DSUB} \quad rd, \, rs, \, rt
\]

MIPS III

Purpose:

To subtract 64-bit integers; trap if overflow.

Description:

\[
rd \leftarrow rs - rt
\]

The 64-bit doubleword value in GPR \(rt\) is subtracted from the 64-bit value in GPR \(rs\) to produce a 64-bit result. If the subtraction results in 64-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 64-bit result is placed into GPR \(rd\).

Restrictions:

None

Operation:

64-bit processors

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

if (64_bit_arithmetic_overflow) then

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

else

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

endif

Exceptions:

Integer Overflow
Reserved Instruction

Programming Notes:

DSUBU performs the same arithmetic operation but, does not trap on overflow.
Doubleword Subtract Unsigned DSUBU

<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>DSUBU</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>
</tr>
</tbody>
</table>

Format: DSUBU  rd, rs, rt
Purpose: To subtract 64-bit integers.
Description: rd ← rs - rt
The 64-bit doubleword value in GPR rt is subtracted from the 64-bit value in GPR rs and the 64-bit arithmetic result is placed into GPR rd.
No Integer Overflow exception occurs under any circumstances.
Restrictions:
None
Operation: 64-bit processors
GPR[rd] ← GPR[rs] – GPR[rt]
Exceptions:
Reserved Instruction
Programming Notes:
The term “unsigned” in the instruction name is a misnomer; this operation is 64-bit modulo arithmetic that does not trap on overflow. It is appropriate for arithmetic which is not signed, such as address arithmetic, or integer arithmetic environments that ignore overflow, such as “C” language arithmetic.
Jump

J target

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 following the jump, in the branch delay slot, before jumping.

Restrictions:
None

Operation:

I:
I+1 : PC ← PC_GPRLEN..28 || instr_index || 0^2

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 to anywhere in the region from anywhere in the region which a signed relative offset would not allow.

This definition creates the boundary case where the branch instruction is in the last word of a 256 MB region and can therefore only branch to the following 256 MB region containing the branch delay slot.
Jump And Link JAL

**Format:**

```
JAL    target
```

**Purpose:**

To 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, where execution would continue 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 following the jump, in the branch delay slot, before jumping.

**Restrictions:**

None

**Operation:**

```
I:  GPR[31] ← PC + 8
I+1: PC ← PC_GPRLEN.28 || instr_index || 0^2
```

**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 to anywhere in the region from anywhere in the region which a signed relative offset would not allow.

This definition creates the boundary case where the branch instruction is in the last word of a 256 MB region and can therefore only branch to the following 256 MB region containing the branch delay slot.
JALR   Jump And Link Register

Format:  JALR   rs                      (rd = 31 implied)  
          JALR   rd, rs

Purpose: To 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 would continue after a procedure
call.

Jump to the effective target address in GPR rs. Execute the instruction following the
jump, in the branch delay slot, before jumping.

Restrictions:

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

The effective target address in GPR rs must be naturally aligned. If either of the two
least-significant bits are not -zero, then an Address Error exception occurs, not for the
jump instruction, but when the branch target is subsequently fetched as an instruction.

Operation:

I:  temp ← GPR[rs]
       GPR[rd] ← PC + 8
I+1: PC ← temp

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 JR

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>15</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>rs</td>
<td>000000000000000000</td>
<td>JR</td>
<td>001000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Format: JR rs  

Purpose: To branch to an instruction address in a register.

Description: PC ← rs  

Jump to the effective target address in GPR rs. Execute the instruction following the jump, in the branch delay slot, before jumping.

Restrictions:  
The effective target address in GPR rs must be naturally aligned. If either of the two least-significant bits are not -zero, then an Address Error exception occurs, not for the jump instruction, but when the branch target is subsequently fetched as an instruction.

Operation:  
I: temp ← GPR[rs]  
I+1: PC ← temp

Exceptions: None
**LB Load 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>LB</td>
<td></td>
<td></td>
<td>rt</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Format:**  
```
LB rt, offset(base)
```

**MIPS I**

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

**Description:**  
```
rt ← 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 offset is added to the contents of GPR base to form the effective address.

**Restrictions:**  
None

**Operation:**  
**32-bit processors**
```
vAddr ← sign_extend(offset) + GPR[base]
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← pAddr_{PSIZE-1..2} || (pAddr_{1,0} xor ReverseEndian^2)
memword ← LoadMemory (uncached, BYTE, pAddr, vAddr, DATA)
byte ← vAddr_{1,0} xor BigEndianCPU^2
GPR[rt] ← sign_extend(memword_{7+8*byte..8*byte})
```

**Operation:**  
**64-bit processors**
```
vAddr ← sign_extend(offset) + GPR[base]
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← pAddr_{PSIZE-1..3} || (pAddr_{2,0} xor ReverseEndian^3)
memdouble ← LoadMemory (uncached, BYTE, pAddr, vAddr, DATA)
byte ← vAddr_{2,0} xor BigEndianCPU^3
GPR[rt] ← sign_extend(memdouble_{7+8*byte..8*byte})
```

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

31 26 25 21 20 16 15  0

<table>
<thead>
<tr>
<th>LBU</th>
<th>base</th>
<th>rt</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 0 1 0 0</td>
<td>6</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

Format: LBU rt, offset(base)

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

Description: rt ← memory[base+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 offset is added to the contents of GPR base to form the effective address.

Restrictions:
None

Operation: 32-bit processors

vAddr ← sign_extend(offset) + GPR[base]
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← pAddrPSIZE−1..2 || (pAddr1..0 xor ReverseEndian)
memword ← LoadMemory (uncached, BYTE, pAddr, vAddr, DATA)
byte ← vAddr1..0 xor BigEndianCPU
GPR[rt] ← zero_extend(memword7..8*byte..8*byte)

Operation: 64-bit processors

vAddr ← sign_extend(offset) + GPR[base]
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← pAddrPSIZE−1..3 || (pAddr2..0 xor ReverseEndian)
memdouble ← LoadMemory (uncached, BYTE, pAddr, vAddr, DATA)
byte ← vAddr2..0 xor BigEndianCPU
GPR[rt] ← zero_extend(memdouble7..8*byte..8*byte)

Exceptions: TLB Refill, TLB Invalid, Address Error
LD Load Doubleword

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

Purpose: To load a doubleword from memory.

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

The contents of the 64-bit doubleword at the memory location specified by the aligned effective address are fetched 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:

The effective address must be naturally aligned. If any of the three least-significant bits of the address are non-zero, an Address Error exception occurs.

MIPS IV: The low-order 3 bits of the \( \text{offset} \) field must be zero. If they are not, the result of the instruction is undefined.

Operation: 64-bit processors

\[
\begin{align*}
\text{vAddr} & \leftarrow \text{sign\_extend(\text{offset})} + \text{GPR[base]} \\
\text{if } (\text{vAddr}_{2..0}) \neq 0^3 & \text{ then SignalException(AddressError) endif} \\
(\text{pAddr, uncached}) & \leftarrow \text{AddressTranslation (vAddr, DATA, LOAD)} \\
\text{memdouble} & \leftarrow \text{LoadMemory (uncached, DOUBLEWORD, pAddr, vAddr, DATA)} \\
\text{GPR[rt]} & \leftarrow \text{memdouble}
\end{align*}
\]

Exceptions:

TLB Refill, TLB Invalid
Bus Error
Address Error
Reserved Instruction
Load Doubleword to Coprocessor LDCz

Format:
- LDC1  rt, offset(base)
- LDC2  rt, offset(base)

Purpose:
To load a doubleword from memory to a coprocessor general register.

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

The contents of the 64-bit doubleword at the memory location specified by the aligned effective address are fetched and made available to coprocessor unit \( zz \). The 16-bit signed \( \text{offset} \) is added to the contents of GPR \( \text{base} \) to form the effective address.

The manner in which each coprocessor uses the data is defined by the individual coprocessor specifications. The usual operation would place the data into coprocessor general register \( rt \).

Each MIPS architecture level defines up to 4 coprocessor units, numbered 0 to 3 (see Coprocessor Instructions on page A-11). The opcodes corresponding to coprocessors that are not defined by an architecture level may be used for other instructions.

Restrictions:
Access to the coprocessors is controlled by system software. Each coprocessor has a "coprocessor usable" bit in the System Control coprocessor. The usable bit must be set for a user program to execute a coprocessor instruction. If the usable bit is not set, an attempt to execute the instruction will result in a Coprocessor Unusable exception. An unimplemented coprocessor must never be enabled. The result of executing this instruction for an unimplemented coprocessor when the usable bit is set, is undefined.

This instruction is not available for coprocessor 0, the System Control coprocessor, and the opcode may be used for other instructions.

The effective address must be naturally aligned. If any of the three least-significant bits of the effective address are non-zero, an Address Error exception occurs.

MIPS IV: The low-order 3 bits of the \( \text{offset} \) field must be zero. If they are not, the result of the instruction is undefined.

Operation:
- 32-bit processors
  - \( \text{vAddr} \leftarrow \text{sign\_extend} (\text{offset}) + \text{GPR}[\text{base}] \)
  - if \( (\text{vAddr}_{2:0}) \neq 0^3 \) then \( \text{Signal\_Exception\(\text{(Address\_Error)} \) endif} \)
  - \( (\text{pAddr}, \text{uncached}) \leftarrow \text{Address\_Translation} (\text{vAddr}, \text{DATA, LOAD}) \)
  - \( \text{memdouble} \leftarrow \text{Load\_Memory} (\text{uncached}, \text{DOUBLEWORD, pAddr, vAddr, DATA}) \)
  - \( \text{COP\_LD} (z, rt, \text{memdouble}) \)
LDCz

Load Doubleword to Coprocessor

Operation: 64-bit processors

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

if \( (\text{vAddr}_{2..0}) \neq 0^3 \) then SignalException(\( \text{AddressError} \)) endif

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

\[ \text{memdouble} \leftarrow \text{LoadMemory}(\text{uncached}, \text{DOUBLEWORD}, \text{pAddr}, \text{vAddr}, \text{DATA}) \]

\[ \text{COP\_LD}(z, \text{rt}, \text{memdouble}) \]

Exceptions:

- TLB Refill, TLB Invalid
- Bus Error
- Address Error
- Reserved Instruction
- Coprocessor Unusable
Load Doubleword Left \textbf{LDL}

Format: \texttt{LDL rt, offset(base)}

Purpose: To load the most-significant part of a doubleword from an unaligned memory address.

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

The 16-bit signed \texttt{offset} is added to the contents of GPR \texttt{base} to form an effective address \( \text{EffAddr} \). \text{EffAddr} is the address of the most-significant of eight consecutive bytes forming a doubleword in memory \( (DW) \) starting at an arbitrary byte boundary. A part of \( DW \), the most-significant one to eight bytes, is in the aligned doubleword containing \( \text{EffAddr} \). This part of \( DW \) is loaded appropriately into the most-significant (left) part of GPR \( rt \) leaving the remainder of GPR \( rt \) unchanged.

The figure below illustrates this operation for big-endian byte ordering. The eight consecutive bytes in 2..9 form an unaligned doubleword starting at location 2. A part of \( DW \), six bytes, is contained in the aligned doubleword containing the most-significant byte at 2. First, LDL loads these six bytes into the left part of the destination register and leaves the remainder of the destination unchanged. Next, the complementary LDR loads the remainder of the unaligned doubleword.

\begin{figure}[h]
\centering
\begin{tabular}{|c|c|c|c|c|c|c|c|}
\hline
31 & 26 & 25 & 21 & 20 & 16 & 15 & 0 \\
\hline
LDL & 21 & 20 & 16 & 15 & 0 \\
0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 \\
\hline
6 & 5 & 5 & 16 \\
\hline
\end{tabular}
\caption{Doubleword at byte 2 in memory, big-endian byte order, - each mem byte contains its address}
\end{figure}

\textbf{Figure A-2} Unaligned Doubleword Load using LDL and LDR.
The bytes loaded from memory to the destination register depend on both the offset of the effective address within an aligned doubleword, i.e. the low three bits of the address (vAddr_{2..0}), and the current byte ordering mode of the processor (big- or little-endian). The table below shows the bytes loaded for every combination of offset and byte ordering.

**Table A-28  Bytes Loaded by LDL Instruction**

<table>
<thead>
<tr>
<th>vAddr_{2..0}</th>
<th>Initial contents of Destination Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7</td>
<td>big-endian offset</td>
</tr>
<tr>
<td>I J K L M N O P</td>
<td>a b c d e f g h</td>
</tr>
<tr>
<td>7 6 5 4 3 2 1 0</td>
<td>little-endian offset</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Big-endian byte ordering</th>
<th>Little-endian byte ordering</th>
</tr>
</thead>
<tbody>
<tr>
<td>I J K L M N O P</td>
<td>P b c d e f g h</td>
</tr>
<tr>
<td>J K L M N O P h</td>
<td>O P c d e f g h</td>
</tr>
<tr>
<td>K L M N O P g h</td>
<td>N O P d e f g h</td>
</tr>
<tr>
<td>L M N O P f g h</td>
<td>M N O P e f g h</td>
</tr>
<tr>
<td>M N O P e f g h</td>
<td>L M N O P f g h</td>
</tr>
<tr>
<td>N O P d e f g h</td>
<td>K L M N O P g h</td>
</tr>
<tr>
<td>O P c d e f g h</td>
<td>J K L M N O P h</td>
</tr>
<tr>
<td>P b c d e f g h</td>
<td>I J K L M N O P</td>
</tr>
</tbody>
</table>

Restrictions:
None

Operation: 64-bit processors

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

(pAddr, uncached) \leftarrow AddressTranslation (vAddr, DATA, LOAD)

pAddr \leftarrow pAddr_{(PSIZE-1)\ldots3} || (pAddr_{2..0} \text{xor ReverseEndian}^3)

if BigEndianMem = 0 then
    pAddr \leftarrow pAddr_{(PSIZE-1)\ldots3} || 0^3
endif

byte \leftarrow vAddr_{2..0} \text{xor BigEndianCPU}^3

memdouble \leftarrow LoadMemory (uncached, byte, pAddr, vAddr, DATA)

GPR[rt] \leftarrow memdouble_{7\ldots8*byte..0} || GPR[rt]_{55\ldots8*byte..0}

Exceptions:
TLB Refill, TLB Invalid
Bus Error
Address Error
Reserved Instruction
**Load Doubleword Right**

### Format:
```
LDR   rt, offset(base)
```

### MIPS III

#### Purpose:
To load the least-significant part of a doubleword from an unaligned memory address.

#### Description:
```
rt ← 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 least-significant of eight consecutive bytes forming a doubleword in memory (`DW`) starting at an arbitrary byte boundary. A part of `DW`, the least-significant one to eight bytes, is in the aligned doubleword containing `EffAddr`. This part of `DW` is loaded appropriately into the least-significant (right) part of GPR `rt` leaving the remainder of GPR `rt` unchanged.

The figure below illustrates this operation for big-endian byte ordering. The eight consecutive bytes in 2..9 form an unaligned doubleword starting at location 2. A part of `DW`, two bytes, is contained in the aligned doubleword containing the least-significant byte at 9. First, LDR loads these two bytes into the right part of the destination register and leaves the remainder of the destination unchanged. Next, the complementary LDL loads the remainder of the unaligned doubleword.

**Figure A-3** Unaligned Doubleword Load using LDR and LDL.
The bytes loaded from memory to the destination register depend on both the offset of the effective address within an aligned doubleword, i.e. the low three bits of the address (vAddr_{2..0}), and the current byte ordering mode of the processor (big- or little-endian). The table below shows the bytes loaded for every combination of offset and byte ordering.

### Table A-29  Bytes Loaded by LDR Instruction

<table>
<thead>
<tr>
<th>Memory contents and byte offsets (vAddr_{2..0})</th>
<th>Initial contents of Destination Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>most  — significance — least</td>
<td>most  — significance — least</td>
</tr>
<tr>
<td>0 1 2 3 4 5 6 7 ← big-endian offset</td>
<td>a b c d e f g h</td>
</tr>
<tr>
<td>7 6 5 4 3 2 1 0 ← little-endian offset</td>
<td></td>
</tr>
</tbody>
</table>

| Destination register contents after instruction (shaded is unchanged) |
|-----------------------------|-----------------------------|
| Big-endian byte ordering    | Little-endian byte ordering |
| vAddr_{2..0}                | 0                           |
| a b c d e f g i             | I J K L M N O P              |
| a b c d e f i J             | 1                           |
| a b c d e I J K             | 2                           |
| a b c d I J K L             | 3                           |
| a b c I J K L M             | 4                           |
| a b I J K L M N             | 5                           |
| a I J K L M N O             | 6                           |
| I J K L M N O P             | 7                           |

Restrictions:

None

Operation: 64-bit processors

\[
\begin{align*}
v\text{Addr} & \leftarrow \text{sign}_\text{extend}(\text{offset}) + \text{GPR[base]} \\
(p\text{Addr}, \text{uncached}) & \leftarrow \text{AddressTranslation}(v\text{Addr}, \text{DATA}, \text{LOAD}) \\
p\text{Addr} & \leftarrow p\text{Addr}(\text{PSIZE}-1..3) \text{||} (p\text{Addr}_{2..0} \text{ xor ReverseEndian}^3) \\
\text{if BigEndianMem} = 1 & \text{ then} \\
\text{pAddr} & \leftarrow p\text{Addr}(\text{PSIZE}-1..3) \text{||} 0^3 \\
\text{endif} \\
\text{byte} & \leftarrow v\text{Addr}_{2..0} \text{ xor BigEndianCPU}^3 \\
\text{memdouble} & \leftarrow \text{LoadMemory}(\text{uncached}, \text{byte}, p\text{Addr}, v\text{Addr}, \text{DATA}) \\
\text{GPR}[rt] & \leftarrow \text{GPR}[rt]_{63..64-8^*\text{byte} \text{||} \text{memdouble}_{63..8^*\text{byte}}} 
\end{align*}

Exceptions:

TLB Refill, TLB Invalid
Bus Error
Address Error
Reserved Instruction
Load Halfword

Format: \texttt{LH \ rt, offset(base)} \\
Purpose: To load a halfword from memory as a signed value.

Description: \texttt{rt} \leftarrow \text{memory}[\text{base}+\text{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 \texttt{rt}. The 16-bit signed \texttt{offset} is added to the contents of GPR \texttt{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.

MIPS IV: The low-order bit of the \texttt{offset} field must be zero. If it is not, the result of the instruction is undefined.

Operation: 32-bit processors

\begin{align*}
\text{vAddr} & \leftarrow \text{sign\_extend}(\text{offset}) + \text{GPR}[\text{base}] \\
\text{if } (\text{vAddr}_{0}) & \neq 0 \text{ then } \text{SignalException(AddressError) endif} \\
(\text{pAddr, uncached}) & \leftarrow \text{AddressTranslation (vAddr, DATA, LOAD)} \\
\text{pAddr} & \leftarrow \text{pAddr}_{\text{PSIZE - 1..2} \ || \ (\text{pAddr}_{1..0} \ xor (\text{ReverseEndian} \ || \ 0))} \\
\text{memword} & \leftarrow \text{LoadMemory (uncached, HALFWORD, pAddr, vAddr, DATA)} \\
\text{byte} & \leftarrow \text{vAddr}_{1..0} \ xor (\text{BigEndianCPU} \ || \ 0) \\
\text{GPR}[\text{rt}] & \leftarrow \text{sign\_extend}(\text{memword}_{15+8*\text{byte}..8*\text{byte}})
\end{align*}

Operation: 64-bit processors

\begin{align*}
\text{vAddr} & \leftarrow \text{sign\_extend}(\text{offset}) + \text{GPR}[\text{base}] \\
\text{if } (\text{vAddr}_{0}) & \neq 0 \text{ then } \text{SignalException(AddressError) endif} \\
(\text{pAddr, uncached}) & \leftarrow \text{AddressTranslation (vAddr, DATA, LOAD)} \\
\text{pAddr} & \leftarrow \text{pAddr}_{\text{PSIZE - 1..3} \ || \ (\text{pAddr}_{2..0} \ xor (\text{ReverseEndian} \ || \ 0))} \\
\text{memdouble} & \leftarrow \text{LoadMemory (uncached, HALFWORD, pAddr, vAddr, DATA)} \\
\text{byte} & \leftarrow \text{vAddr}_{2..0} \ xor (\text{BigEndianCPU}^2 \ || \ 0) \\
\text{GPR}[\text{rt}] & \leftarrow \text{sign\_extend}(\text{memdouble}_{15+8*\text{byte}..8*\text{byte}})
\end{align*}

Exceptions:

TLB Refill, TLB Invalid
Bus Error
Address Error
LHU Load Halfword Unsigned

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

**Description:**

\[
GPR[rt] \leftarrow \text{memory}[\text{base} + \text{offset}]
\]

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 \(\text{offset}\) is added to the contents of GPR \(\text{base}\) to form the effective address.

**Restrictions:**

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

MIPS IV: The low-order bit of the \(\text{offset}\) field must be zero. If it is not, the result of the instruction is undefined.

**Operation:**

**32-bit processors**

\[
\begin{align*}
  \text{vAddr} & \leftarrow \text{sign extend}(\text{offset}) + \text{GPR[base]} \\
  \text{if } (\text{vAddr}_0) & \neq 0 \text{ then SIGNAL EXCEPTION(AddressError) endif} \\
  (\text{pAddr}, \text{uncached}) & \leftarrow \text{AddressTranslation(} \text{vAddr, DATA, LOAD}) \\
  \text{pAddr} & \leftarrow \text{pAddr}_{\text{PSIZE} - 1..2} || (\text{pAddr}_{1..0} \text{xor (ReverseEndian} || 0)) \\
  \text{memword} & \leftarrow \text{LoadMemory(uncached, HALFWORD, pAddr, vAddr, DATA)} \\
  \text{byte} & \leftarrow \text{vAddr}_{1..0} \text{xor (BigEndianCPU} || 0) \\
  \text{GPR[rt]} & \leftarrow \text{zero extend(memword}_{15+8*\text{byte}..8*\text{byte}})
\end{align*}
\]

**Operation:**

**64-bit processors**

\[
\begin{align*}
  \text{vAddr} & \leftarrow \text{sign extend}(\text{offset}) + \text{GPR[base]} \\
  \text{if } (\text{vAddr}_0) & \neq 0 \text{ then SIGNAL EXCEPTION(AddressError) endif} \\
  (\text{pAddr}, \text{uncached}) & \leftarrow \text{AddressTranslation(} \text{vAddr, DATA, LOAD}) \\
  \text{pAddr} & \leftarrow \text{pAddr}_{\text{PSIZE} - 1..3} || (\text{pAddr}_{2..0} \text{xor (ReverseEndian_2} || 0)) \\
  \text{memdouble} & \leftarrow \text{LoadMemory(uncached, HALFWORD, pAddr, vAddr, DATA)} \\
  \text{byte} & \leftarrow \text{vAddr}_{2..0} \text{xor (BigEndianCPU_2} || 0) \\
  \text{GPR[rt]} & \leftarrow \text{zero extend(memdouble}_{15+8*\text{byte}..8*\text{byte}})
\end{align*}
\]

**Exceptions:**

- TLB Refill, TLB Invalid
- Address Error
Load Linked Word \texttt{LL}

Format: \texttt{LL \ rt, offset(base)}

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

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

The LL and SC instructions provide primitives to implement atomic Read-Modify-Write (RMW) operations for cached memory locations.

The 16-bit signed \texttt{offset} is added to the contents of GPR \texttt{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 \texttt{rt}. This begins a RMW sequence on the current processor.

There is one active RMW sequence per processor. When an LL is executed it starts the 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. See the description of SC for a list of events and conditions that cause the SC to fail and an example instruction sequence using LL and SC.

Executing LL on one processor does not cause an action that, by itself, would cause 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 (see Memory Access Types on page A-12).

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

- MIPS IV: The low-order 2 bits of the \texttt{offset} field must be zero. If they are not, the result of the instruction is undefined.
Load Linked Word

Operation: 32-bit processors

\[
vAddr \leftarrow \text{sign\_extend}(\text{offset}) + \text{GPR}[\text{base}]
\]
if \((\text{vAddr}_{1,0}) \neq 0^2\) then SignalException(AddressError) endif

\[
(p\text{Addr}, \text{uncached}) \leftarrow \text{AddressTranslation}(v\text{Addr}, \text{DATA}, \text{LOAD})
\]
\[
\text{memword} \leftarrow \text{LoadMemory}(\text{uncached}, \text{WORD}, p\text{Addr}, v\text{Addr}, \text{DATA})
\]
\[
\text{GPR}[rt] \leftarrow \text{memword}
\]
LLbit \leftarrow 1

Operation: 64-bit processors

\[
vAddr \leftarrow \text{sign\_extend}(\text{offset}) + \text{GPR}[\text{base}]
\]
if \((\text{vAddr}_{1,0}) \neq 0^2\) then SignalException(AddressError) endif

\[
(p\text{Addr}, \text{uncached}) \leftarrow \text{AddressTranslation}(v\text{Addr}, \text{DATA}, \text{LOAD})
\]
\[
p\text{Addr} \leftarrow p\text{Addr}_{\text{PSIZE}-1..3} \ || \ (p\text{Addr}_{2,0} \ \text{xor} \ (\text{ReverseEndian} \ || \ 0^2))
\]
\[
\text{memdouble} \leftarrow \text{LoadMemory}(\text{uncached}, \text{WORD}, p\text{Addr}, v\text{Addr}, \text{DATA})
\]
\[
\text{byte} \leftarrow v\text{Addr}_{2,0} \ \text{xor} \ (\text{BigEndianCPU} \ || \ 0^2)
\]
\[
\text{GPR}[rt] \leftarrow \text{sign\_extend}(\text{memdouble}_{31+8*\text{byte}..8*\text{byte}})
\]
LLbit \leftarrow 1

Exceptions:

- TLB Refill, TLB Invalid
- Address Error
- Reserved Instruction

Programming Notes:

There is no Load Linked Word Unsigned operation corresponding to Load Word Unsigned.

Implementation Notes:

An LL on one processor must not take action that, by itself, would cause an SC for the same block on another processor to fail. If an implementation depends on retaining the data in cache during the RMW sequence, cache misses caused by LL must not fetch data in the exclusive state, thus removing it from the cache, if it is present in another cache.
Load Linked Doubleword \textbf{LLD}

\begin{tabular}{|c|c|c|c|}
\hline
31 & 26 & 25 & 21 20 16 15 0 \\
\hline
LLD & base & rt & offset \\
1 1 0 1 0 0 & 6 & 5 & 5 & 16 \\
\hline
\end{tabular}

\textbf{Format:} LLD rt, offset(base) \quad \textbf{MIPS III}

\textbf{Purpose:} To load a doubleword from memory for an atomic read-modify-write.

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

The LLD and SCD instructions provide primitives to implement atomic Read-Modify-Write (RMW) operations for cached memory locations.

The 16-bit signed \textit{offset} is added to the contents of GPR \textit{base} to form an effective address.

The contents of the 64-bit doubleword at the memory location specified by the aligned effective address are fetched and written into GPR \textit{rt}. This begins a RMW sequence on the current processor.

There is one active RMW sequence per processor. When an LLD is executed it starts the active RMW sequence replacing any other sequence that was active.

The RMW sequence is completed by a subsequent SCD instruction that either completes the RMW sequence atomically and succeeds, or does not and fails. See the description of SCD for a list of events and conditions that cause the SCD to fail and an example instruction sequence using LLD and SCD.

Executing LLD on one processor does not cause an action that, by itself, would cause an SCD for the same block to fail on another processor.

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

\textbf{Restrictions:}

The addressed location must be cached; if it is not, the result is undefined (see Memory Access Types on page A-12).

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

MIPS IV: The low-order 3 bits of the \textit{offset} field must be zero. If they are not, the result of the instruction is undefined.
**LLD**

Load Linked Doubleword

**Operation:** 64-bit processors

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

if \((v\text{Addr}_{3..0}) \neq 0\) then SignalException(AddressError) endif

\((p\text{Addr, uncached}) \leftarrow \text{AddressTranslation}(v\text{Addr, DATA, LOAD})\)

\(\text{memdouble} \leftarrow \text{LoadMemory(uncached, DOUBLEWORD, p\text{Addr, v\text{Addr, DATA}})}\)

\(\text{GPR}[rt] \leftarrow \text{memdouble}\)

\(\text{LLbit} \leftarrow 1\)

**Exceptions:**
- TLB Refill, TLB Invalid
- Address Error
- Reserved Instruction

**Programming Notes:**

**Implementation Notes:**

An LLD on one processor must not take action that, by itself, would cause an SCD for the same block on another processor to fail. If an implementation depends on retaining the data in cache during the RMW sequence, cache misses caused by LLD must not fetch data in the exclusive state, thus removing it from the cache, if it is present in another cache.
Load Upper Immediate \textbf{LUI} \\

<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>10</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>LUI</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Format: \textbf{LUI} \( \text{rt, immediate} \) \quad \text{MIPS I}

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

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

The 16-bit \textit{immediate} is shifted left 16 bits and concatenated with 16 bits of low-order zeros. The 32-bit result is sign-extended and placed into GPR \textit{rt}.

Restrictions: None

Operation:

\[ \text{GPR}[\textit{rt}] \leftarrow \text{sign\_extend(} \text{immediate} \| 0^{16} \) \]

Exceptions: None
Load Word

Format: \texttt{LW \ rt, offset(base)}

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

Description: \texttt{rt} \leftarrow \text{memory}[\text{base+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 \texttt{rt}. The 16-bit signed \texttt{offset} is added to the contents of GPR \texttt{base} to form the effective address.

Restrictions:

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

MIPS IV: The low-order 2 bits of the \texttt{offset} field must be zero. If they are not, the result of the instruction is undefined.

Operation: 32-bit processors

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

\[ \text{if } (\text{vAddr}_1,0) \neq 0^2 \text{ then } \text{SignalException(AddressError) endif} \]

\[ \text{(pAddr, uncached)} \leftarrow \text{AddressTranslation (vAddr, DATA, LOAD)} \]

\[ \text{memword} \leftarrow \text{LoadMemory (uncached, WORD, pAddr, vAddr, DATA)} \]

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

Operation: 64-bit processors

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

\[ \text{if } (\text{vAddr}_1,0) \neq 0^2 \text{ then } \text{SignalException(AddressError) endif} \]

\[ \text{(pAddr, uncached)} \leftarrow \text{AddressTranslation (vAddr, DATA, LOAD)} \]

\[ \text{pAddr} \leftarrow \text{pAddr}_{\text{SIZE}\_1,3} \ || \ (\text{pAddr}_{2,0} \text{ xor (ReverseEndian } || \ 0^2)) \]

\[ \text{memdouble} \leftarrow \text{LoadMemory (uncached, WORD, pAddr, vAddr, DATA)} \]

\[ \text{byte} \leftarrow \text{vAddr}_{2,0} \text{ xor (BigEndianCPU } || \ 0^2) \]

\[ \text{GPR[rt]} \leftarrow \text{sign\_extend(memdouble}_{31+8^\text{byte}.8^\text{byte})} \]

Exceptions:

TLB Refill, TLB Invalid

Bus Error

Address Error
Load Word To Coprocessor \textbf{LWCz}

<table>
<thead>
<tr>
<th>LWCz</th>
<th>base</th>
<th>rt</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>1100zz</td>
<td>6</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

**Format:**

LWC1 \( rt, \text{offset}(\text{base}) \)  
LWC2 \( rt, \text{offset}(\text{base}) \)  
LWC3 \( rt, \text{offset}(\text{base}) \)

**Purpose:**
To load a word from memory to a coprocessor general register.

**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 and made available to coprocessor unit \( zz \). The 16-bit signed \( \text{offset} \) is added to the contents of GPR \( \text{base} \) to form the effective address.

The manner in which each coprocessor uses the data is defined by the individual coprocessor specification. The usual operation would place the data into coprocessor general register \( rt \).

Each MIPS architecture level defines up to 4 coprocessor units, numbered 0 to 3 (see Coprocessor Instructions on page A-11). The opcodes corresponding to coprocessors that are not defined by an architecture level may be used for other instructions.

**Restrictions:**

Access to the coprocessors is controlled by system software. Each coprocessor has a “coprocessor usable” bit in the System Control coprocessor. The usable bit must be set for a user program to execute a coprocessor instruction. If the usable bit is not set, an attempt to execute the instruction will result in a Coprocessor Unusable exception. An unimplemented coprocessor must never be enabled. The result of executing this instruction for an unimplemented coprocessor when the usable bit is set, is undefined.

This instruction is not available for coprocessor 0, the System Control coprocessor, and the opcode may be used for other instructions.

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

MIPS IV: The low-order 2 bits of the \( \text{offset} \) field must be zero. If they are not, the result of the instruction is undefined.

**Operation:**

\[
\begin{align*}
1: & \quad \text{vAddr} \leftarrow \text{sign}	ext{-}\text{extend} (\text{offset}) + \text{GPR}[\text{base}] \\
& \quad \text{if } (\text{vAddr}_{1,0}) \neq 0^2 \text{ then } \text{SignalException} (\text{AddressError}) \text{ endif} \\
& \quad (p\text{Addr, uncached}) \leftarrow \text{AddressTranslation} (\text{vAddr, DATA, LOAD}) \\
& \quad \text{memword} \leftarrow \text{LoadMemory} (\text{uncached, WORD, pAddr, vAddr, DATA}) \\
1+1: & \quad \text{COP\_LW} (z, rt, \text{memword})
\end{align*}
\]
LWCz Load Word To Coprocessor

Operation: 64-bit processors

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

if \( (\text{vAddr}_{1.0}) \neq 0^2 \) then SignalException(AddressError) endif

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

\( \text{pAddr} \leftarrow \text{pAddr}_{PSIZE-1..3} \ || \ (\text{pAddr}_{2..0} \ xor \ (\text{ReverseEndian} \ || \ 0^2)) \)

\( \text{memdouble} \leftarrow \text{LoadMemory (uncached, DOUBLEWORD, pAddr, vAddr, DATA)} \)

\( \text{byte} \leftarrow \text{vAddr}_{2..0} \ xor \ (\text{BigEndianCPU} \ || \ 0^2) \)

\( \text{memword} \leftarrow \text{memdouble}_{31+8*\text{byte..8*\text{byte}} \}

\text{COP\_LW (z, rt, memdouble)}

Exceptions:
- TLB Refill, TLB Invalid
- Bus Error
- Address Error
- Coprocessor Unusable
Load Word Left LWL

**Format:**

LWL   rt, offset(base)  

**MIPS I**

**Purpose:**

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

**Description:**

rt ← rt MERGE memory[base+offset]

The 16-bit signed offset is added to the contents of GPR base to form an effective address \((Eff\Addr)\). \(Eff\Addr\) is the address of the most-significant of four consecutive bytes forming a word in memory \((W)\) starting at an arbitrary byte boundary. A part of \(W\), the most-significant one to four bytes, is in the aligned word containing \(Eff\Addr\). 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.

If GPR rt is a 64-bit register, the destination word is the low-order word of the register. The loaded value is treated as a signed value; the word sign bit (bit 31) is always loaded from memory and the new sign bit value is copied into bits 63..32.

---

Figure A-4  Unaligned Word Load using LWL and LWR.

---

CPU Instruction Set  

MIPS IV Instruction Set. Rev 3.2  

A-97
The figure above illustrates this operation for big-endian byte ordering for 32-bit and 64-bit registers. The four consecutive bytes in 2..5 form an unaligned word starting at location 2. A part of \( W \), two bytes, is in the aligned word containing the most-significant byte at 2. First, LWL loads these two 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.

The bytes loaded from memory to the destination register depend on both the offset of the effective address within an aligned word, i.e. the low two bits of the address (vAddr\(_{1,0}\)), and the current byte ordering mode of the processor (big- or little-endian). The table below shows the bytes loaded for every combination of offset and byte ordering.

### Table A-30  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>( \begin{array}{cccc} I &amp; J &amp; K &amp; L \end{array} ) offset (vAddr(_{1,0}))</td>
<td>( \begin{array}{cccc} a &amp; b &amp; c &amp; d \end{array} )( \begin{array}{cccc} e &amp; f &amp; g &amp; h \end{array} )</td>
</tr>
</tbody>
</table>

#### Destination 64-bit register contents after instruction (shaded is unchanged)

<table>
<thead>
<tr>
<th>Big-endian byte ordering</th>
<th>vAddr(_{1,0})</th>
<th>Little-endian byte ordering</th>
</tr>
</thead>
<tbody>
<tr>
<td>sign bit (31) extended</td>
<td>( \begin{array}{cccc} I &amp; J &amp; K &amp; L \end{array} )</td>
<td>sign bit (31) extended</td>
</tr>
<tr>
<td>sign bit (31) extended</td>
<td>( \begin{array}{cccc} J &amp; K &amp; L &amp; h \end{array} )</td>
<td>sign bit (31) extended</td>
</tr>
<tr>
<td>sign bit (31) extended</td>
<td>( \begin{array}{cccc} K &amp; L &amp; g &amp; h \end{array} )</td>
<td>sign bit (31) extended</td>
</tr>
<tr>
<td>sign bit (31) extended</td>
<td>( \begin{array}{cccc} L &amp; f &amp; g &amp; h \end{array} )</td>
<td>sign bit (31) extended</td>
</tr>
</tbody>
</table>

The word sign (31) is always loaded and the value is copied into bits 63..32.

The unaligned loads, LWL and LWR, are exceptions to the load-delay scheduling restriction in the MIPS I architecture. An unaligned load instruction to GPR \( rt \) that immediately follows another load to GPR \( rt \) can “read” the loaded data. It will correctly merge the 1 to 4 loaded bytes with the data loaded by the previous instruction.
Load Word Left LWL

Restrictions:

MIPS I scheduling restriction: The loaded data is not available for use by the following instruction. The instruction immediately following this one, unless it is an unaligned load (LWL, LWR), may not use GPR rt as a source register. If this restriction is violated, the result of the operation is undefined.

Operation: 32-bit processors

\[
\begin{align*}
vAddr & \leftarrow \text{sign\_extend}(\text{offset}) + \text{GPR}[\text{base}] \\
(pAddr, \text{uncached}) & \leftarrow \text{AddressTranslation}(vAddr, \text{DATA, LOAD}) \\
pAddr & \leftarrow \text{pAddr}_{(\text{PSIZE}\_1)\_2} || (\text{pAddr}_{1..0} \text{xor ReverseEndian}^2) \\
\text{if BigEndianMem} & = 0 \text{ then} \\
\quad pAddr & \leftarrow \text{pAddr}_{(\text{PSIZE}\_1)\_2} || 0^2 \\
\text{endif} \\
\text{byte} & \leftarrow vAddr_{1..0} \text{xor BigEndianCPU}^2 \\
\text{memword} & \leftarrow \text{LoadMemory}(\text{uncached, byte, pAddr, vAddr, DATA}) \\
\text{GPR}[rt] & \leftarrow \text{memword}_{7+8*\text{byte..0} || \text{GPR}[rt]_{23-8*\text{byte..0}}}
\end{align*}
\]

Operation: 64-bit processors

\[
\begin{align*}
vAddr & \leftarrow \text{sign\_extend}(\text{offset}) + \text{GPR}[\text{base}] \\
(pAddr, \text{uncached}) & \leftarrow \text{AddressTranslation}(vAddr, \text{DATA, LOAD}) \\
pAddr & \leftarrow \text{pAddr}_{(\text{PSIZE}\_1)\_3} || (\text{pAddr}_{2..0} \text{xor ReverseEndian}^3) \\
\text{if BigEndianMem} & = 0 \text{ then} \\
\quad pAddr & \leftarrow \text{pAddr}_{(\text{PSIZE}\_1)\_3} || 0^3 \\
\text{endif} \\
\text{byte} & \leftarrow 0 || (vAddr_{1..0} \text{xor BigEndianCPU}^2) \\
\text{word} & \leftarrow vAddr_2 \text{xor BigEndianCPU} \\
\text{memdouble} & \leftarrow \text{LoadMemory}(\text{uncached, byte, pAddr, vAddr, DATA}) \\
\text{temp} & \leftarrow \text{memdouble}_{31+32*\text{word}-8*\text{byte..32*word} || \text{GPR}[rt]_{23-8*\text{byte..0}}} \\
\text{GPR}[rt] & \leftarrow (\text{temp}_{31})^{32} || \text{temp}
\end{align*}
\]

Exceptions:

TLB Refill, TLB Invalid
Bus Error
Address Error

Programming Notes:

The architecture provides no direct support for treating unaligned words as unsigned values, i.e. zeroing bits 63..32 of the destination register when bit 31 is loaded. See SLL or SLLV for a single-instruction method of propagating the word sign bit in a register into the upper half of a 64-bit register.
### LWR

**Load Word Right**

<table>
<thead>
<tr>
<th>Base</th>
<th>rt</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

**Format:**

LWR  rt, offset(base)

**MIPS I**

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

**Description:**

\[
rt \leftarrow rt \text{ MERGE memory}\left[\text{base}+\text{offset}\right]
\]

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 four consecutive bytes forming a word in memory (W) starting at an arbitrary byte boundary. A part of W, the least-significant one to four bytes, is in the aligned word containing 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.

If GPR rt is a 64-bit register, the destination word is the low-order word of the register. The loaded value is treated as a signed value; if the word sign bit (bit 31) is loaded (i.e. when all four bytes are loaded) then the new sign bit value is copied into bits 63..32. If bit 31 is not loaded then the value of bits 63..32 is implementation dependent; the value is either unchanged or a copy of the current value of bit 31. Executing both LWR and LWL, in either order, delivers in a sign-extended word value in the destination register.

**Figure A-5  Unaligned Word Load using LWR and LWL.**

![Figure A-5 Unaligned Word Load using LWR and LWL.](image-url)
The figure above illustrates this operation for big-endian byte ordering for 32-bit and 64-bit registers. The four consecutive bytes in 2..5 form an unaligned word starting at location 2. A part of \( W \), two bytes, is in the aligned word containing the least-significant byte at 5. First, LWR loads these two 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, i.e. the low two bits of the address (\( v\text{Addr}_{1..0} \)), and the current byte ordering mode of the processor (big- or little-endian). The table below shows the bytes loaded for every combination of offset and byte ordering.

<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 ) ( \leftarrow ) big-endian</td>
<td>( 64\text{-bit register} )</td>
</tr>
<tr>
<td>( I \ J \ K \ L ) ( \text{offset} (v\text{Addr}_{1..0}) )</td>
<td>( a \ b \ c \ d \ e \ f \ g \ h )</td>
</tr>
<tr>
<td>( 3 \ 2 \ 1 \ 0 ) ( \leftarrow ) little-endian</td>
<td>most ( \rightarrow ) significance ( \rightarrow ) least</td>
</tr>
</tbody>
</table>

| Destination 64-bit register contents after instruction (shaded is unchanged) |
|-----------------------------|-----------------------------|
| Big-endian byte ordering    | Little-endian byte ordering |
| \( v\text{Addr}_{1..0} \)   | \( v\text{Addr}_{1..0} \)   |
| No cng or sign-extend \( e \ f \ g \ I \) \( 0 \) | sign bit (31) extended \( I \ J \ K \ L \) |
| No cng or sign-extend \( e \ f \ I \ J \) \( 1 \) | No cng or sign-extend \( e \ f \ I \ J \) |
| No cng or sign-extend \( e \ I \ J \ K \) \( 2 \) | No cng or sign-extend \( e \ f \ I \ J \) |
| sign bit (31) extended \( I \ J \ K \ L \) \( 3 \) | No cng or sign-extend \( e \ f \ g \ I \) |

When the word sign bit (31) is loaded, its value is copied into bits 63..32. When it is not loaded, the behavior is implementation specific. Bits 63..32 are either unchanged or a the value of the unloaded bit 31 is copied into them.

<table>
<thead>
<tr>
<th>32-bit register</th>
<th>big-endian</th>
<th>( v\text{Addr}_{1..0} )</th>
<th>little-endian</th>
</tr>
</thead>
<tbody>
<tr>
<td>( e \ f \ g \ I ) ( 0 )</td>
<td>( I \ J \ K \ L )</td>
<td>( e \ f \ I \ J )</td>
<td></td>
</tr>
<tr>
<td>( e \ f \ I \ J ) ( 1 )</td>
<td>( e \ I \ J \ K )</td>
<td>( e \ f \ I \ J )</td>
<td></td>
</tr>
<tr>
<td>( e \ I \ J \ K ) ( 2 )</td>
<td>( e \ I \ J \ K )</td>
<td>( e \ f \ g \ I )</td>
<td></td>
</tr>
<tr>
<td>( I \ J \ K \ L ) ( 3 )</td>
<td>( I \ J \ K \ L )</td>
<td>( e \ f \ g \ I )</td>
<td></td>
</tr>
</tbody>
</table>

The unaligned loads, LWL and LWR, are exceptions to the load-delay scheduling restriction in the MIPS I architecture. An unaligned load to GPR \( rt \) that immediately follows another load to GPR \( rt \) can “read” the loaded data. It will correctly merge the 1 to 4 loaded bytes with the data loaded by the previous instruction.
Restrictions:

MIPS I scheduling restriction: The loaded data is not available for use by the following instruction. The instruction immediately following this one, unless it is an unaligned load (LWL, LWR), may not use GPR rt as a source register. If this restriction is violated, the result of the operation is undefined.

Restrictions:

None

Operation: 32-bit processors

vAddr ← sign_extend(offset) + GPR[base]
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← pAddr[PSIZE-1..2 || (pAddr[1..0] xor ReverseEndian^2)
if BigEndianMem = 0 then
    pAddr ← pAddr[PSIZE-1..2 || 0^2
endif
byte ← vAddr[1..0] xor BigEndianCPU^2
memword ← LoadMemory (uncached, byte, pAddr, vAddr, DATA)
GPR[rt] ← memword[31..32-8*byte || GPR[rt][31–8*byte..0]

Operation: 64-bit processors

vAddr ← sign_extend(offset) + GPR[base]
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← pAddr[PSIZE-1..3 || (pAddr[2..0] xor ReverseEndian^3)
if BigEndianMem = 1 then
    pAddr ← pAddr[PSIZE-1..3 || 0^3
endif
byte ← vAddr[1..0] xor BigEndianCPU^2
word ← vAddr[2] xor BigEndianCPU
memdouble ← LoadMemory (uncached, 0 || byte, pAddr, vAddr, DATA)
temp ← GPR[rt][31..32-8*byte || memdouble[31+32*word..32*word+8*byte
if byte = 4 then
    utemp ← (temp[31])^32 /* loaded bit 31, must sign extend */
else
    one of the following two behaviors:
    utemp ← GPR[rt][63..32] /* leave what was there alone */
    utemp ← (GPR[rt][31])^32 /* sign-extend bit 31 */
endif
GPR[rt] ← utemp || temp

Exceptions:

TLB Refill, TLB Invalid
Bus Error
Address Error
Load Word Right LWR

Programming Notes:

The architecture provides no direct support for treating unaligned words as unsigned values, i.e. zeroing bits 63..32 of the destination register when bit 31 is loaded. See SLL or SLLV for a single-instruction method of propagating the word sign bit into the upper half of a 64-bit register.
Load Word Unsigned

**Format:**

LWU   rt, offset(base)  

**Purpose:**

To load a word from memory as an unsigned value.

**Description:**

rt ← memory[base+offset]  

The contents of the 32-bit word 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 either of the two least-significant bits of the address are non-zero, an Address Error exception occurs.

MIPS IV: The low-order 2 bits of the offset field must be zero. If they are not, the result of the instruction is undefined.

**Operation:**

64-bit processors

vAddr ← sign_extend(offset) + GPR[base]  

if (vAddr_{1,0}) ≠ 0^2 then SignalException(AddressError) endif  

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

pAddr ← pAddr_{PSIZE-1..3} || (pAddr_{2..0} xor (ReverseEndian || 0^2))  

memdouble ← LoadMemory (uncached, WORD, pAddr, vAddr, DATA)  

byte ← vAddr_{2..0} xor (BigEndianCPU || 0^2)  

GPR[rt] ← 0^{32} || memdouble_{31+8*byte..8*byte}  

**Exceptions:**

TLB Refill, TLB Invalid  
Bus Error  
Address Error  
Reserved Instruction
Move From HI Register\textbf{MFHI}

<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>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Format: \textbf{MFHI} \textbf{rd}  
Purpose: To copy the special purpose HI register to a GPR.  
Description: \textbf{rd} \leftarrow \textbf{HI}  
The contents of special register \textbf{HI} are loaded into GPR \textbf{rd}.  
Restrictions:  
The two instructions that follow an MFHI instruction must not be instructions that modify the \textbf{HI} register: DDIV, DDIVU, DIV, DIVU, DMULT, DMULTU, MTHI, MULT, MULTU. If this restriction is violated, the result of the MFHI is undefined.  
Operation:  
\text{GPR[rd]} \leftarrow \text{HI}  
Exceptions: None
MFLO
Move From LO Register

31 26 25 16 15 11 10 6 5 0

<table>
<thead>
<tr>
<th>SPECIAL</th>
<th>000000</th>
<th>000000000000</th>
<th>rd</th>
<th>000000</th>
<th>MFLO</th>
<th>010010</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>10</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Format: MFLO rd

Purpose: To copy the special purpose LO register to a GPR.

Description: rd ← LO

The contents of special register LO are loaded into GPR rd.

Restrictions:

The two instructions that follow an MFLO instruction must not be instructions that modify the LO register: DDIV, DDIVU, DIV, DIVU, DMULT, DMULTU, MTLO, MULT, MULTU. If this restriction is violated, the result of the MFLO is undefined.

Operation:

GPR[rd] ← LO

Exceptions:

None
**Move Conditional on Not Zero**

**Format:**

```
MOVN     rd, rs, rt
```

**MIPS IV**

**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:**

```
if GPR[rt] ≠ 0 then
    GPR[rd] ← GPR[rs]
endif
```

**Exceptions:**
Reserved Instruction

**Programming Notes:**

The nonzero value tested here is the “condition true” result from the SLT, SLTI, SLTU, and SLTIU comparison instructions.
MOVZ
Move Conditional on Zero

Format: MOVZ rd, rs, rt

MIPS IV

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: Reserved Instruction

Programming Notes:
The zero value tested here is the “condition false” result from the SLT, SLTI, SLTU, and SLTIU comparison instructions.
Move To HI Register

**MTHI**

| 31 | 26 | 25 | 24 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| SPECIAL | rs | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | MTHI | 0 | 1 | 0 | 0 | 0 | 1 |

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:

If either of the two preceding instructions is MFHI, the result of that MFHI is undefined. Reads of the HI or LO special registers must be separated from subsequent instructions that write to them by two or more other instructions.

A computed result written to the HI/LO pair by DDIV, DDIVU, DIV, DIVU, DMULT, DMULTU, MULT, or MULTU must be read by MFHI or MFLO before another result is written into either HI or LO. If an MTHI instruction is executed following one of these arithmetic instructions, but before a MFLO or MFHI instruction, the contents of LO are undefined. 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 undefined value
```

Operation:

1-2: 1-1: HI ← undefined
1: HI ← GPR[rs]

Exceptions:

None
MTLO

Move To LO Register

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:

If either of the two preceding instructions is MFLO, the result of that MFLO is undefined. Reads of the HI or LO special registers must be separated from subsequent instructions that write to them by two or more other instructions.

A computed result written to the HI/LO pair by DDIV, DDIVU, DIV, DIVU, DMULT, DMULTU, MULT, or MULTU must be read by MFHI or MFLO before another result is written into either HI or LO. If an MTLO instruction is executed following one of these arithmetic instructions, but before a MFLO or MFHI instruction, the contents of HI are undefined. 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 undefined value

Operation:

I-2 : I-1 : LO ← undefined
I:   LO ← GPR[rs]

Exceptions:

None
Multiply Word \texttt{MULT}

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|c|c|c|c|c|}
\hline
31 & 26 & 25 & 21 & 20 & 16 & 15 & 0 & \multicolumn{2}{c|}{MULT} \\
\hline
SPECIAL & rs & rt & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\
\hline
\end{tabular}
\end{table}

Format: \texttt{MULT rs, rt} \hspace{1cm} \texttt{MIPS I}

Purpose: To multiply 32-bit signed integers.

Description: \((LO, HI) \leftarrow rs \times rt\)

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:

On 64-bit processors, if either GPR \(rt\) or GPR \(rs\) do not contain sign-extended 32-bit values (bits 63..31 equal), then the result of the operation is undefined.

If either of the two preceding instructions is MFHI or MFLO, the result of the MFHI or MFLO is undefined. Reads of the \(HI\) or \(LO\) special registers must be separated from subsequent instructions that write to them by two or more other instructions.

Operation:

\begin{verbatim}
if (NotWordValue(GPR[rs]) or NotWordValue(GPR[rt])) then UndefinedResult() endif
I-2., I-1: LO, HI \leftarrow undefined
I: prod \leftarrow GPR[rs]_{31..0} \times GPR[rt]_{31..0}
    LO \leftarrow sign_extend(prod_{31..0})
    HI \leftarrow sign_extend(prod_{63..32})
\end{verbatim}

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 will wait (interlock) 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.
MULTU Multiply Unsigned Word

Format: MULTU rs, rt

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:

On 64-bit processors, if either GPR rt or GPR rs do not contain sign-extended 32-bit values (bits 63..31 equal), then the result of the operation is undefined.

If either of the two preceding instructions is MFHI or MFLO, the result of the MFHI or MFLO is undefined. Reads of the HI or LO special registers must be separated from subsequent instructions that write to them by two or more other instructions.

Operation:

if (NotWordValue(GPR[rs]) or NotWordValue(GPR[rt])) then UndefinedResult() endif
I - 2.; I - 1: LO, HI ← undefined
I: prod ← (0 || GPR[rs]31..0) × (0 || GPR[rt]31..0)
   LO ← sign_extend(prod31..0)
   HI ← sign_extend(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 will wait (interlock) 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.
**Not Or**

### 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

MIPS IV Instruction Set. Rev 3.2

Format: \( \text{OR} \quad rd, \; rs, \; rt \)

Purpose: To do a bitwise logical OR.

Description: \( rd \leftarrow rs \text{ OR } rt \)

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:

\[
\text{GPR}[rd] \leftarrow \text{GPR}[rs] \text{ OR GPR}[rt]
\]

Exceptions:

None
Or Immediate  \texttt{ORI}

<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>O R I</td>
<td>rs</td>
<td>rt</td>
<td>immediate</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

6 5 5 16

Format: \texttt{ORI \ rt, rs, immediate}  

MIPS I

Purpose: To do a bitwise logical OR with a constant.

Description: \(rd \leftarrow rs \text{ OR immediate}\)

The 16-bit \textit{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:

\[\text{GPR}[rt] \leftarrow \text{zero\_extend(immediate)} \text{ or GPR}[rs]\]

Exceptions:

None
**PREF**

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>1 0 0 1 1</td>
<td>base</td>
<td>hint</td>
<td>offset</td>
<td>PREF</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Summary:

- **Format:** `PREF hint, offset(base)`
- **MIPS IV**
- **Purpose:** To prefetch data from memory.
- **Description:** `prefetch_memory(base+offset)`

**Description:**

PREF adds the 16-bit signed `offset` to the contents of GPR `base` to form an effective byte address. It advises that data at the effective address may be used in the near future. The `hint` field supplies information about the way that the data is expected to be used.

PREF is an advisory instruction. It may change the performance of the program. For all `hint` values and all effective addresses, it neither changes architecturally-visible state nor alters the meaning of the program. An implementation may do nothing when executing a PREF instruction.

If MIPS IV instructions are supported and enabled, PREF does not cause addressing-related exceptions. If it raises an exception condition, the exception condition is ignored. If an addressing-related exception condition is raised and ignored, no data will be prefetched. Even if no data is prefetched in such a case, some action that is not architecturally-visible, such as writeback of a dirty cache line, might take place.

PREF will never generate a memory operation for a location with an uncached memory access type (see Memory Access Types on page A-12).

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.

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 that does not change architecturally-visible state or alter the meaning of a program. It is expected that implementations will either do nothing or take an action that will increase the performance of the program.

For a cached location, the expected, and useful, action is for the processor 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 hint field supplies information about the way the data is expected to be used. No hint value causes an action that modifies architecturally-visible state. A processor may use a hint value to improve the effectiveness of the prefetch action. The defined hint values and the recommended prefetch action are shown in the table below. The hint table may be extended in future implementations.

**Table A-32 Values of Hint Field for Prefetch 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>Data is expected to be loaded (not modified). Fetch data as if for a load.</td>
</tr>
<tr>
<td>1</td>
<td>store</td>
<td>Data is expected to be stored or modified. Fetch data as if for a store.</td>
</tr>
<tr>
<td>2-3</td>
<td></td>
<td>Not yet defined.</td>
</tr>
<tr>
<td>4</td>
<td>load_streamed</td>
<td>Data is expected to be loaded (not modified) but not reused extensively; it will “stream” through cache. Fetch data as if for a load and place it in the cache so that it will not displace data prefetched as “retained”.</td>
</tr>
<tr>
<td>5</td>
<td>store_streamed</td>
<td>Data is expected to be stored or modified but not reused extensively; it will “stream” through cache. Fetch data as if for a store and place it in the cache so that it will not displace data prefetched as “retained”.</td>
</tr>
<tr>
<td>6</td>
<td>load_retained</td>
<td>Data is expected to be loaded (not modified) and reused extensively; it should be “retained” in the cache. Fetch data as if for a load and place it in the cache so that it will not be displaced by data prefetched as “streamed”.</td>
</tr>
<tr>
<td>7</td>
<td>store_retained</td>
<td>Data is expected to be stored or modified and reused extensively; it should be “retained” in the cache. Fetch data as if for a store and place it in the cache so that will not be displaced by data prefetched as “streamed”.</td>
</tr>
<tr>
<td>8-31</td>
<td></td>
<td>Not yet defined.</td>
</tr>
</tbody>
</table>

Restrictions:
- None

Operation:
- \( v\text{Addr} \leftarrow \text{GPR}\{\text{base}\} + \text{sign\_extend}(\text{offset}) \)
- \((p\text{Addr}, \text{uncached}) \leftarrow \text{AddressTranslation}(v\text{Addr}, \text{DATA}, \text{LOAD}) \)
- Prefetch(uncached, p\text{Addr}, v\text{Addr}, \text{DATA}, \text{hint})

Exceptions:
- Reserved Instruction
Prefetch can not 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 will not cause an exception to prefetch using an address pointer value before the validity of a pointer is determined.

It is recommended that a reserved hint field value either cause a default prefetch action that is expected to be useful for most cases of data use, such as the “load” hint, or cause the instruction to be treated as a NOP.
Store Byte \(SB\)

**Format:** \(SB \ rt, \text{offset(base)}\)  
**Purpose:** To store a byte to memory.  
**Description:** \(\text{memory[base+offset]} \leftarrow rt\)  

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 \(\text{offset}\) is added to the contents of GPR \(\text{base}\) to form the effective address.

**Restrictions:**  
None

**Operation:**  
32-bit processors  
\[
\begin{align*}
\text{vAddr} &\leftarrow \text{sign\_extend}(\text{offset}) + \text{GPR[base]} \\
(p\text{Addr, uncached}) &\leftarrow \text{AddressTranslation (vAddr, DATA, STORE)} \\
p\text{Addr} &\leftarrow p\text{Addr}\_\text{PSIZE-1..2 || (pAddr\_1..0 xor ReverseEndian)}^2 \\
\text{byte} &\leftarrow v\text{Addr}\_1..0 \text{ xor BigEndianCPU}^2 \\
\text{dataword} &\leftarrow \text{GPR[rt]}_{31–8*\text{byte..0} || 0^8\text{byte}} \\
\text{StoreMemory (uncached, BYTE, dataword, pAddr, vAddr, DATA)}
\end{align*}
\]

64-bit processors  
\[
\begin{align*}
\text{vAddr} &\leftarrow \text{sign\_extend}(\text{offset}) + \text{GPR[base]} \\
(p\text{Addr, uncached}) &\leftarrow \text{AddressTranslation (vAddr, DATA, STORE)} \\
p\text{Addr} &\leftarrow p\text{Addr}\_\text{PSIZE-1..3 || (pAddr\_2..0 xor ReverseEndian)}^3 \\
\text{byte} &\leftarrow v\text{Addr}\_2..0 \text{ xor BigEndianCPU}^3 \\
\text{datadouble} &\leftarrow \text{GPR[rt]}_{63–8*\text{byte..0} || 0^8\text{byte}} \\
\text{StoreMemory (uncached, BYTE, datadouble, pAddr, vAddr, DATA)}
\end{align*}
\]

**Exceptions:**  
TLB Refill, TLB Invalid  
TLB Modified  
Bus Error  
Address Error
Store Conditional Word

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

Purpose: To store a word to memory to complete an atomic read-modify-write.

Description: \texttt{if (atomic\_update) then memory[base+offset] \leftarrow rt, rt \leftarrow 1 \text{ 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 \texttt{offset} is added to the contents of GPR \texttt{base} to form an effective address.

The SC completes the RMW sequence begun by the preceding LL instruction executed on the processor. If it would complete the RMW sequence atomically, then the least-significant 32-bit word of GPR \texttt{rt} is stored into memory at the location specified by the aligned effective address and a one, indicating success, is written into GPR \texttt{rt}.

Otherwise, memory is not modified and a zero, indicating failure, is written into GPR \texttt{rt}.

If any of the following events occurs between the execution of LL and SC, the SC will fail:

- 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. It is at least one word and is at most the minimum page size.

- An exception occurs on the processor executing the LL/SC.
  An implementation may detect “an exception” in one of three ways:
  1) Detect exceptions and fail when an exception occurs.
  2) Fail after the return-from-interrupt instruction (RFE or ERET) is executed.
  3) Do both 1 and 2.

If any 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 unpredictable. 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 will be undefined:

- Execution of SC must have been preceded by execution of an LL instruction.
Store Conditional Word SC

- 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. See Memory Access Types on page A-12.

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, 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.

The definition above applies to user-mode operation on all MIPS processors that support the MIPS II architecture. There may be other implementation-specific events, such as privileged CP0 instructions, that will cause an SC instruction to fail in some cases. System programmers using LL/SC should consult implementation-specific documentation.

Restrictions:

The addressed location must have a memory access type of cached noncoherent or cached coherent; if it does not, the result is undefined (see Memory Access Types on page A-12).

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

MIPS IV: The low-order 2 bits of the offset field must be zero. If they are not, the result of the instruction is undefined.

Operation: 32-bit processors

\[
\begin{align*}
\text{vAddr} & \leftarrow \text{sign\_extend(\text{offset}) + GPR[base]} \\
\text{if } (\text{vAddr}_{1:0}) \neq 0^2 \text{ then SignalException(AddressError) endif} \\
(p\text{Addr, uncached}) & \leftarrow \text{AddressTranslation (vAddr, DATA, STORE)} \\
\text{dataword} & \leftarrow \text{GPR[rt]} \\
\text{if LLbit then} \\
\quad & \text{StoreMemory (uncached, WORD, dataword, pAddr, vAddr, DATA)} \\
\text{endif} \\
\text{GPR[rt]} & \leftarrow 0^{31} || \text{LLbit}
\end{align*}
\]
**Store Conditional Word**

Operation: 64-bit processors

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

\[
\text{if } (vAddr_{1,0}) \neq 0^2 \text{ then } \text{SignalException(AddressError) endif}
\]

\[
(pAddr, \text{uncached}) \leftarrow \text{AddressTranslation} (vAddr, \text{DATA, STORE})
\]

\[
pAddr \leftarrow pAddr_{\text{PSIZE}-1,3} || (pAddr_{2,0} \text{ xor (ReverseEndian} \| 0^2))
\]

\[
\text{byte} \leftarrow vAddr_{2,0} \text{ xor (BigEndianCPU} \| 0^2)
\]

\[
\text{datadouble} \leftarrow \text{GPR}[rt]_{63-8\text{byte},0} || 0^8\text{byte}
\]

\[
\text{if LLbit then}
\]

\[
\text{StoreMemory (uncached, WORD, datadouble, pAddr, vAddr, DATA)}
\]

\[
\text{endif}
\]

\[
\text{GPR}[rt] \leftarrow 0^{63} || \text{LLbit}
\]

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 in the example atomic increment operation 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
```

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, 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.

Implementation Notes:

The block of memory that is “locked” for LL/SC is typically the largest cache line in use.
Store Conditional Doubleword \textbf{SCD}

<table>
<thead>
<tr>
<th>31 26 25 21 20 16 15</th>
<th>base rt offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 0 0</td>
<td>6 5 5 16</td>
</tr>
</tbody>
</table>

**Format:** \texttt{SCD \ rt, offset(base)} \hspace{1cm} \textbf{MIPS III}

**Purpose:** To store a doubleword to memory to complete an atomic read-modify-write.

**Description:**
\[
\text{if (atomic_update) then } \text{memory[base+offset]} \leftarrow \text{rt, rt} \leftarrow 1 \text{ else rt} \leftarrow 0
\]

The 16-bit signed \textit{offset} is added to the contents of GPR \textit{base} to form an effective address.

The SCD completes the RMW sequence begun by the preceding LLD instruction executed on the processor. If it would complete the RMW sequence atomically, then the 64-bit doubleword of GPR \textit{rt} is stored into memory at the location specified by the aligned effective address and a one, indicating success, is written into GPR \textit{rt}. Otherwise, memory is not modified and a zero, indicating failure, is written into GPR \textit{rt}.

If any of the following events occurs between the execution of LLD and SCD, the SCD will fail:

- 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. It is at least one doubleword and is at most the minimum page size.
- An exception occurs on the processor executing the LLD/SCD. An implementation may detect “an exception” in one of three ways:
  1) Detect exceptions and fail when an exception occurs.
  2) Fail after the return-from-interrupt instruction (RFE or ERET) is executed.
  3) Do both 1 and 2.

If any of the following events occurs between the execution of LLD and SCD, the SCD may succeed or it may fail; the success or failure is unpredictable. Portable programs should not cause one of these events.

- A memory access instruction (load, store, or prefetch) is executed on the processor executing the LLD/SCD.
- The instructions executed starting with the LLD and ending with the SCD 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 SCD will be undefined:

- Execution of SCD must have been preceded by execution of an LLD instruction.
SCD

Store Conditional Doubleword

- A RMW sequence executed without intervening exceptions must use the same address in the LLD and SCD. The address is the same if the virtual address, physical address, and cache-coherence algorithm are identical.

Atomic RMW is provided only for memory locations with cached noncoherent or cached coherent memory access types. The extent to which the detection of atomicity operates correctly depends on the system implementation and the memory access type used for the location. See Memory Access Types on page A-12.

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, 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.

The definition above applies to user-mode operation on all MIPS processors that support the MIPS III architecture. There may be other implementation-specific events, such as privileged CP0 instructions, that will cause an SCD instruction to fail in some cases. System programmers using LLD/SCD should consult implementation-specific documentation.

Restrictions:

The addressed location must have a memory access type of cached noncoherent or cached coherent; if it does not, the result is undefined (see Memory Access Types on page A-12). The 64-bit doubleword of register rt is conditionally 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.

The effective address must be naturally aligned. If any of the three least-significant bits of the address are non-zero, an Address Error exception occurs.

MIPS IV: The low-order 3 bits of the offset field must be zero. If they are not, the result of the instruction is undefined.

Operation: 64-bit processors

\[
\begin{align*}
vAddr & \leftarrow \text{sign\_extend}(\text{offset}) + \text{GPR}[\text{base}] \\
\text{if } (vAddr_{2,0}) & \neq 0^3 \text{ then } \text{SignalException(AddressError) endif} \\
(pAddr, \text{uncached}) & \leftarrow \text{AddressTranslation}(vAddr, \text{DATA, STORE}) \\
datadouble & \leftarrow \text{GPR}[rt] \\
\text{if LLbit then} \\
\quad \text{StoreMemory(uncached, DOUBLEWORD, datadouble, pAddr, vAddr, DATA)} \\
\text{endif} \\
\text{GPR}[rt] & \leftarrow 0^{63} \| \text{LLbit}
\end{align*}
\]
Store Conditional Doubleword SCD

Exceptions:
- TLB Refill, TLB Invalid
- TLB Modified
- Address Error
- Reserved Instruction

Programming Notes:
LLD and SCD are used to atomically update memory locations as shown in the example atomic increment operation below.

```
L1:

  LLD T1, (T0)  # load counter
  ADDI T2, T1, 1  # increment
  SCD 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 LLD and SCD cause SCD to fail, so persistent exceptions must be avoided. Some examples of these are arithmetic operations that trap, system calls, floating-point operations that trap or require software emulation assistance.

LLD and SCD 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.

Implementation Notes:
The block of memory that is “locked” for LLD/SCD is typically the largest cache line in use.
**SD**

**Store Doubleword**

<table>
<thead>
<tr>
<th>Format:</th>
<th>SD rt, offset(base)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose:</td>
<td>To store a doubleword to memory.</td>
</tr>
</tbody>
</table>
| Description: | memory[base+offset] ← rt  

The 64-bit doubleword in GPR 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.

<table>
<thead>
<tr>
<th>Restrictions:</th>
</tr>
</thead>
<tbody>
<tr>
<td>The effective address must be naturally aligned. If any of the three least-significant bits of the effective address are non-zero, an Address Error exception occurs.</td>
</tr>
<tr>
<td>MIPS IV: The low-order 3 bits of the offset field must be zero. If they are not, the result of the instruction is undefined.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operation:</th>
<th>64-bit processors</th>
</tr>
</thead>
<tbody>
<tr>
<td>vAddr ← sign_extend(offset) + GPR[base]</td>
<td></td>
</tr>
<tr>
<td>if (vAddr₂₀) ≠ 0³ then SignalException(AddressError) endif</td>
<td></td>
</tr>
<tr>
<td>(pAddr, uncached) ← AddressTranslation (vAddr, DATA, STORE)</td>
<td></td>
</tr>
<tr>
<td>datadouble ← GPR[rt]</td>
<td></td>
</tr>
<tr>
<td>StoreMemory (uncached, DOUBLEWORD, datadouble, pAddr, vAddr, DATA)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Exceptions:</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLB Refill, TLB Invalid</td>
</tr>
<tr>
<td>TLB Modified</td>
</tr>
<tr>
<td>Address Error</td>
</tr>
<tr>
<td>Reserved Instruction</td>
</tr>
</tbody>
</table>
Store Doubleword From Coprocessor SDCz

Format:

SDC1 \( \text{rt, offset} \) (base)  
SDC2 \( \text{rt, offset} \) (base)  

MIPS II

Purpose: To store a doubleword from a coprocessor general register to memory.

Description:

The data supplied by each coprocessor is defined by the individual coprocessor specifications. The usual operation would read the data from coprocessor general register \( \text{rt} \).

Each MIPS architecture level defines up to 4 coprocessor units, numbered 0 to 3 (see Coprocessor Instructions on page A-11). The opcodes corresponding to coprocessors that are not defined by an architecture level may be used for other instructions.

Restrictions:

Access to the coprocessors is controlled by system software. Each coprocessor has a “coprocessor usable” bit in the System Control coprocessor. The usable bit must be set for a user program to execute a coprocessor instruction. If the usable bit is not set, an attempt to execute the instruction will result in a Coprocessor Unusable exception. An unimplemented coprocessor must never be enabled. The result of executing this instruction for an unimplemented coprocessor when the usable bit is set, is undefined.

This instruction is not defined for coprocessor 0, the System Control coprocessor, and the opcode may be used for other instructions.

The effective address must be naturally aligned. If any of the three least-significant bits of the effective address are non-zero, an Address Error exception occurs.

MIPS IV: The low-order 3 bits of the offset field must be zero. If they are not, the result of the instruction is undefined.

Operation: 32-bit processors

\[
\begin{align*}
\text{vAddr} & \leftarrow \text{sign extend}(\text{offset}) + \text{GPR}[\text{base}] \\
\text{if} \ (\text{vAddr}_{2..0}) & \neq 0^3 \ \text{then} \ \text{SignalException(AddressError)} \ \text{endif} \\
(\text{pAddr, uncached}) & \leftarrow \text{AddressTranslation (vAddr, DATA, STORE)} \\
\text{datadouble} & \leftarrow \text{COP_SD(z, rt)} \\
\text{StoreMemory (uncached, DOUBLEWORD, datadouble, pAddr, vAddr, DATA)}
\end{align*}
\]
SDCz Store Doubleword From Coprocessor

Operation: 64-bit processors

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

\[
\text{if } (vAddr_2..0) \neq 03 \text{ then SignalException(AddressError) endif}
\]

\[
(p\text{Addr}, \text{uncached}) \leftarrow \text{AddressTranslation}(v\text{Addr}, \text{DATA}, \text{STORE})
\]

\[
datadouble \leftarrow \text{COP_SD}(z, rt)
\]

\[
\text{StoreMemory(uncached, DOUBLEWORD, datadouble, p\text{Addr}, v\text{Addr}, \text{DATA})}
\]

Exceptions:

- TLB Refill, TLB Invalid
- TLB Modified
- Address Error
- Reserved Instruction
- Coprocessor Unusable
Store Doubleword Left SDL

**Format:** SDL $rt$, offset(base)  
**MIPS III**

**Purpose:** To store the most-significant part of a doubleword to an unaligned memory address.

**Description:**

$$ memory[base+offset] \leftarrow \text{Some\_Bytes\_From\ rt} $$

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 eight consecutive bytes forming a doubleword in memory (DW) starting at an arbitrary byte boundary. A part of DW, the most-significant one to eight bytes, is in the aligned doubleword containing EffAddr. The same number of most-significant (left) bytes of GPR rt are stored into these bytes of DW.

The figure below illustrates this operation for big-endian byte ordering. The eight consecutive bytes in 2..9 form an unaligned doubleword starting at location 2. A part of DW, six bytes, is contained in the aligned doubleword containing the most-significant byte at 2. First, SDL stores the six most-significant bytes of the source register into these bytes in memory. Next, the complementary SDR instruction stores the remainder of DW.

**Figure A-6  Unaligned Doubleword Store with SDL and SDR**
The bytes stored from the source register to memory depend on both the offset of the effective address within an aligned doubleword, i.e. the low three bits of the address (vAddr_{2,0}), and the current byte ordering mode of the processor (big- or little-endian). The table below shows the bytes stored for every combination of offset and byte ordering.

**Table A-33  Bytes Stored by SDL Instruction**

<table>
<thead>
<tr>
<th>Initial Memory contents and byte offsets</th>
<th>Contents of Source Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>most — significance — least</td>
<td>A B C D E F G H</td>
</tr>
<tr>
<td>0 1 2 3 4 5 6 7 ← big-</td>
<td>i j k l m n o p</td>
</tr>
<tr>
<td>7 6 5 4 3 2 1 0 ← little-endian</td>
<td>A B C D E F G H</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory contents after instruction (shaded is unchanged)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Big-endian byte ordering vAddr_{2,0} Little-endian byte ordering</td>
</tr>
<tr>
<td>A B C D E F G H</td>
</tr>
<tr>
<td>i A B C D E F G</td>
</tr>
<tr>
<td>i j A B C D E F</td>
</tr>
<tr>
<td>i j k A B C D E</td>
</tr>
<tr>
<td>i j k l A B C D</td>
</tr>
<tr>
<td>i j k l m A B C</td>
</tr>
<tr>
<td>i j k l m n A B</td>
</tr>
<tr>
<td>i j k l m n o A</td>
</tr>
</tbody>
</table>

Restrictions:

None

Operation: 64-bit processors

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

\[
(p\text{Addr, uncached}) \leftarrow \text{AddressTranslation}\left(\text{vAddr, DATA, STORE}\right)
\]

\[
p\text{Addr} \leftarrow p\text{Addr}_{\left(\text{PSIZE}\leftarrow 1\right).3} \left\| \left(\text{pAddr}_{2..0} \times\text{ReverseEndian}^3\right)\right.
\]

If BigEndianMem = 0 then

\[
p\text{Addr} \leftarrow p\text{Addr}_{\left(\text{PSIZE}\leftarrow 1\right).3} \left\| 0^3\right.
\]

endif

\[
\text{byte} \leftarrow v\text{Addr}_{2.0} \times\text{BigEndianCPU}^3
\]

\[
\text{datadouble} \leftarrow 0^{56-8\times\text{byte}} \left\| \text{GPR}\left[r\right]_{56-8\times\text{byte}}
\]

\[
\text{StoreMemory}\left(\text{uncached, byte, datadouble, pAddr, vAddr, DATA}\right)
\]

Exceptions:

TLB Refill, TLB Invalid
TLB Modified
Bus Error
Address Error
Reserved Instruction
Store Doubleword Right SDR

Format: SDR rt, offset(base)

Purpose: To store the least-significant part of a doubleword to an unaligned memory address.

Description: 

```
memory[base+offset] ← Some_Bytes_From 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 one to eight bytes forming a doubleword in memory (DW) starting at an arbitrary byte boundary. A part of DW, the least-significant one to eight bytes, is in the aligned doubleword containing EffAddr. The same number of least-significant (right) bytes of GPR rt are stored into these bytes of DW.

The figure below illustrates this operation for big-endian byte ordering. The eight consecutive bytes in 2..9 form an unaligned doubleword starting at location 2. A part of DW, two bytes, is contained in the aligned doubleword containing the least-significant byte at 9. First, SDR stores the two least-significant bytes of the source register into these bytes in memory. Next, the complementary SDL stores the remainder of DW.

*Figure A-7 Unaligned Doubleword Store with SDR and SDL*
The bytes stored from the source register to memory depend on both the offset of the effective address within an aligned doubleword, i.e. the low three bits of the address (vAddr2_0), and the current byte ordering mode of the processor (big- or little-endian). The table below shows the bytes stored for every combination of offset and byte ordering.

**Table A-34  Bytes Stored by SDR Instruction**

<table>
<thead>
<tr>
<th>Initial Memory contents and byte offsets</th>
<th>Contents of Source Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>most — significance — least</td>
<td>most — significance — least</td>
</tr>
<tr>
<td>0 1 2 3 4 5 6 7 ← big-</td>
<td>A B C D E F G H</td>
</tr>
<tr>
<td>7 6 5 4 3 2 1 0 ← little-endian</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory contents after instruction (shaded is unchanged)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Big-endian byte ordering</td>
</tr>
<tr>
<td>----------------------------</td>
</tr>
<tr>
<td>H j k l m n o p</td>
</tr>
<tr>
<td>G H k l m n o p</td>
</tr>
<tr>
<td>F G H k l m n o p</td>
</tr>
<tr>
<td>E F G H l m n o p</td>
</tr>
<tr>
<td>D E F G H n o p</td>
</tr>
<tr>
<td>C D E F G H o p</td>
</tr>
<tr>
<td>B C D E F G H p</td>
</tr>
<tr>
<td>A B C D E F G H</td>
</tr>
</tbody>
</table>

**Restrictions:**
None

**Operation:** 64-bit processors

vAddr ← sign_extend(offset) + GPR[base]
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, STORE)
pAddr ← pAddr(PSIZE-1,...) || (pAddr2_0 xor ReverseEndian)
If BigEndianMem = 0 then
  pAddr ← pAddr(PSIZE-1,...) || 0
endif
byte ← vAddr1_0 xor BigEndianCPU
mdatadouble ← GPR[rt]_83-8 byte || 0^byte
StoreMemory (uncached, DOUBLEWORD-byte, datadouble, pAddr, vAddr, DATA)

**Exceptions:**
TLB Refill, TLB Invalid
TLB Modified
Bus Error
Address Error
Reserved Instruction
### Store Halfword \textit{SH}

<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>SH</td>
<td>1 0 1 0 0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** \texttt{SH \ rt, offset(base)}

**Purpose:** To store a halfword to memory.

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

The least-significant 16-bit halfword of register \texttt{rt} is stored in memory at the location specified by the aligned effective address. The 16-bit signed \texttt{offset} is added to the contents of GPR \texttt{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.
- MIPS IV: The low-order bit of the \texttt{offset} field must be zero. If it is not, the result of the instruction is undefined.

**Operation:**

32-bit processors

- \texttt{vAddr \leftarrow \texttt{sign\_extend(offset)} + GPR[base]}
- \texttt{if (vAddr_p) \neq 0 then SignalException(AddressError) endif}
- \texttt{(pAddr, uncached) \leftarrow AddressTranslation (vAddr, DATA, STORE)}
- \texttt{pAddr \leftarrow pAddr_{PSIZE-1..2} \parallel (pAddr_{1..0} \ xor (ReverseEndian || 0))}
- \texttt{byte \leftarrow vAddr_{1..0} \ xor (BigEndianCPU || 0)}
- \texttt{dataword \leftarrow GPR[rt]_{31-8*byte..0} \parallel 0^{8*byte}}
- \texttt{StoreMemory (uncached, HALFWORD, dataword, pAddr, vAddr, DATA)}

64-bit processors

- \texttt{vAddr \leftarrow \texttt{sign\_extend(offset)} + GPR[base]}
- \texttt{if (vAddr_p) \neq 0 then SignalException(AddressError) endif}
- \texttt{(pAddr, uncached) \leftarrow AddressTranslation (vAddr, DATA, STORE)}
- \texttt{pAddr \leftarrow pAddr_{PSIZE-1..3} \parallel (pAddr_{2..0} \ xor (ReverseEndian^2 || 0))}
- \texttt{byte \leftarrow vAddr_{2..0} \ xor (BigEndianCPU^2 || 0)}
- \texttt{datadouble \leftarrow GPR[rt]_{63-8*byte..0} \parallel 0^{8*byte}}
- \texttt{StoreMemory (uncached, HALFWORD, datadouble, pAddr, vAddr, DATA)}

**Exceptions:**

- TLB Refill, TLB Invalid
- TLB Modified
- Address Error
SLL  Shift Word Left Logical

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 zeroes into the emptied bits; the word result is placed in GPR rd. The bit shift count is specified by sa. If rd is a 64-bit register, the result word is sign-extended.

Restrictions:
None

Operation:
s ← sa
temp ← GPR[rt][31-s].0 || 0sa
GPR[rd] ← sign_extend(temp)

Exceptions:
None

Programming Notes:

Unlike nearly all other word operations the input operand does not have to be a properly sign-extended word value to produce a valid sign-extended 32-bit result. The result word is always sign extended into a 64-bit destination register; this instruction with a zero shift amount truncates a 64-bit value to 32 bits and sign extends it.

Some assemblers, particularly 32-bit assemblers, treat this instruction with a shift amount of zero as a NOP and either delete it or replace it with an actual NOP.
Shift Word Left Logical Variable SLLV

<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>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Format: SLLV rd, rt, rs

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 zeroes into the emptied bits; the result word is placed in GPR rd. The bit shift count is specified by the low-order five bits of GPR rs. If rd is a 64-bit register, the result word is sign-extended.

Restrictions:

None

Operation:

s ← GP[rs]4..0

temp ← GPR[rt](31..s)0 || 0s

GPR[rd] ← sign_extend(temp)

Exceptions:

None

Programming Notes:

Unlike nearly all other word operations the input operand does not have to be a properly sign-extended word value to produce a valid sign-extended 32-bit result. The result word is always sign extended into a 64-bit destination register; this instruction with a zero shift amount truncates a 64-bit value to 32 bits and sign extends it.

Some assemblers, particularly 32-bit assemblers, treat this instruction with a shift amount of zero as a NOP and either delete it or replace it with an actual NOP.
**SLT**

**Set On Less Than**

<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 0 0 0 0 0 0</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>0 0 0 0 0 0</td>
<td>SLT 1 0 1 0 1 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** SLT rd, rs, rt

**Purpose:** To record the result of a less-than comparison.

**Description:** rd ← (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 0 (false).

The arithmetic comparison does not cause an Integer Overflow exception.

**Restrictions:** None

**Operation:**

```plaintext
if GPR[rs] < GPR[rt] then
    GPR[rd] ← 0^GPRLEN-1 || 1
else
    GPR[rd] ← 0^GPRLEN
endif
```

**Exceptions:** None
**Set on Less Than Immediate**

**SLTI**

<table>
<thead>
<tr>
<th>Format:</th>
<th>SLTI rt, rs, immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose:</td>
<td>To record the result of a less-than comparison with a constant.</td>
</tr>
<tr>
<td>Description:</td>
<td>$rt \leftarrow (rs \ &lt; \ immediate)$</td>
</tr>
<tr>
<td></td>
<td>Compare the contents of GPR rs and the 16-bit signed immediate as signed 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 0 (false).</td>
</tr>
<tr>
<td></td>
<td>The arithmetic comparison does not cause an Integer Overflow exception.</td>
</tr>
<tr>
<td>Restrictions:</td>
<td>None</td>
</tr>
<tr>
<td>Operation:</td>
<td>if GPR[rs] &lt; sign_extend(immediate) then GPR[rd] ← $0^{GPRLEN-1}$</td>
</tr>
<tr>
<td>Exceptions:</td>
<td>None</td>
</tr>
</tbody>
</table>
SLTIU  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>
</tbody>
</table>

**Format:** SLTIU  rt, rs, immediate  
**MIPS I**

**Purpose:** To record the result of an unsigned less-than comparison with a constant.

**Description:**

\[ rt \leftarrow (rs < \text{immediate}) \]

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 0 (false).

Because the 16-bit immediate is sign-extended before comparison, the instruction is able to represent the smallest or largest unsigned numbers. The representable values are at the minimum \([0, 32767]\) or maximum \([\text{max\_unsigned}-32767, \text{max\_unsigned}]\) end of the unsigned range.

The arithmetic comparison does not cause an Integer Overflow exception.

**Restrictions:**

None

**Operation:**

\[
\text{if} \ (0 \ || \ GPR[rs]) < (0 \ || \ \text{sign\_extend(immediate)}) \text{ then} \\
\quad GPR[rd] \leftarrow 0^{\text{GPRLEN}-1} || 1 \\
\text{else} \\
\quad GPR[rd] \leftarrow 0^{\text{GPRLEN}} \\
\text{endif}
\]

**Exceptions:**

None
**Set on Less Than Unsigned**

**SLTU**

<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>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>101011</td>
<td></td>
</tr>
</tbody>
</table>

**Format:** 
SLTU   rd, rs, rt

**MIPS I**

**Purpose:** To record the result of an unsigned less-than comparison.

**Description:** 
rd ← (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 0 (false).

The arithmetic comparison does not cause an Integer Overflow exception.

**Restrictions:**
None

**Operation:**

```c
if (0 || GPR[rs]) < (0 || GPR[rt]) then
    GPR[rd] ← 0^GPRLEN-1 || 1
else
    GPR[rd] ← 0^GPRLEN
endif
```

**Exceptions:**
None
**SRA**

**Shift Word Right Arithmetic**

<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>0</td>
<td>rt</td>
<td>rd</td>
<td>sa</td>
<td>SRA</td>
<td>0</td>
</tr>
</tbody>
</table>

**Format:**

SRA rd, rt, sa

**MIPS I**

**Purpose:** To arithmetic right shift 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 count is specified by sa. If rd is a 64-bit register, the result word is sign-extended.

**Restrictions:**

On 64-bit processors, if GPR rt does not contain a sign-extended 32-bit value (bits 63..31 equal) then the result of the operation is undefined.

**Operation:**

if (NotWordValue(GPR[rt])) then UndefinedResult() endif

s ← sa

temp ← (GPR[rt]31)8 || GPR[rt]31..s

GPR[rd] ← sign_extend(temp)

**Exceptions:**

None
Shift Word Right Arithmetic Variable `SRAV`

<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>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

**Format:**  `SRAV   rd, rt, rs`  
**MIPS I**

**Purpose:** To arithmetic right shift a word by a variable number of bits.

**Description:**  
\[ rd \leftarrow rt >> rs \]  
(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 count is specified by the low-order five bits of GPR `rs`. If `rd` is a 64-bit register, the result word is sign-extended.

**Restrictions:**

On 64-bit processors, if GPR `rt` does not contain a sign-extended 32-bit value (bits 63..31 equal) then the result of the operation is undefined.

**Operation:**

\[
\begin{align*}
\text{if } (\text{NotWordValue(GPR[rt]))} & \text{ then UndefinedResult()} \text{ endif} \\
 s & \leftarrow \text{GPR[rs]}_{4..0} \\
temp & \leftarrow (GPR[rt]_{31})^s || GPR[rt]_{31..s} \\
\text{GPR[rd]} & \leftarrow \text{sign\_extend(temp)}
\end{align*}
\]

**Exceptions:**

None
SRL
Shift Word Right Logical

Format: \texttt{SRL \, rd, rt, sa} ~ \textit{MIPS I}

Purpose: To logical right shift a word by a fixed number of bits.

Description: \texttt{rd} $\leftarrow$ \texttt{rt} $\gg$ \texttt{sa} \ (logical)

The contents of the low-order 32-bit word of GPR \texttt{rt} are shifted right, inserting zeros into the emptied bits; the word result is placed in GPR \texttt{rd}. The bit shift count is specified by \texttt{sa}. If \texttt{rd} is a 64-bit register, the result word is sign-extended.

Restrictions:

On 64-bit processors, if GPR \texttt{rt} does not contain a sign-extended 32-bit value (bits 63..31 equal) then the result of the operation is undefined.

Operation:

\begin{verbatim}
    if (NotWordValue(GPR[rt])) then UndefinedResult() endif
    s $\leftarrow$ sa
    temp $\leftarrow$ \texttt{0}^s \| GPR[rt]_{31..s}
    GPR[rd]$\leftarrow$ sign_extend(temp)
\end{verbatim}

Exceptions:

None
Shift Word Right Logical Variable SRLV

<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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Format: SRLV rd, rt, rs  
MIPS I

Purpose: To logical right shift a word by a variable number of bits.

Description: rd ← rt >> rs  (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 count is specified by the low-order five bits of GPR rs. If rd is a 64-bit register, the result word is sign-extended.

Restrictions: 
On 64-bit processors, if GPR rt does not contain a sign-extended 32-bit value (bits 63..31 equal) then the result of the operation is undefined.

Operation:  
if (NotWordValue(GPR[rt])) then UndefinedResult() endif  
s ← GPR[rs]4..0  
temp ← 0s || GPR[rt]31..s  
GPR[rd] ← sign_extend(temp)

Exceptions: 
None
## SUB

**Subtract Word**

### Format:

```
SUB rd, rs, rt
```

### MIPS I

#### Purpose:
To subtract 32-bit integers. If overflow occurs, then trap.

#### Description:

```
rd ← 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:

On 64-bit processors, if either GPR `rt` or GPR `rs` do not contain sign-extended 32-bit values (bits 63..31 equal), then the result of the operation is undefined.

#### Operation:

```
if (NotWordValue(GPR[rs]) or NotWordValue(GPR[rt])) then UndefinedResult() endif

temp ← GPR[rs] - GPR[rt]
if (32_bit_arithmetic_overflow) then
    SignalException(IntegerOverflow)
else
    GPR[rd] ← temp
endif
```

#### Exceptions:

- Integer Overflow

#### Programming Notes:

- SUBU performs the same arithmetic operation but, does not trap on overflow.
Subtract Unsigned Word \textbf{SUBU}

Format: \texttt{SUBU \ rd, rs, rt} \\
Purpose: To subtract 32-bit integers.

Description: \( \text{rd} \leftarrow \text{rs} - \text{rt} \) \\
\hspace{1cm} The 32-bit word value in GPR \textit{rt} is subtracted from the 32-bit value in GPR \textit{rs} and the 32-bit arithmetic result is placed into GPR \textit{rd}.
\hspace{1cm} No integer overflow exception occurs under any circumstances.

Restrictions: 
\hspace{1cm} On 64-bit processors, if either GPR \textit{rt} or GPR \textit{rs} do not contain sign-extended 32-bit values (bits 63..31 equal), then the result of the operation is undefined.

Operation: 
\hspace{1cm} \text{if (NotWordValue(GPR[rs]) or NotWordValue(GPR[rt])) then UndefinedResult() endif} \\
\hspace{1cm} \text{temp} \leftarrow \text{GPR[rs]} - \text{GPR[rt]} \\
\hspace{1cm} \text{GPR[rd]} \leftarrow \text{temp}

Exceptions: 
\hspace{1cm} None

Programming Notes: 
\hspace{1cm} 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 arithmetic which is not signed, such as address arithmetic, or integer arithmetic environments that ignore overflow, such as “C” language arithmetic.
SW 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>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>base rt offset</td>
</tr>
</tbody>
</table>

Format: SW rt, offset(base)

Purpose: To store a word to memory.

Description: memory[base+offset] ← 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 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 two least-significant bits of the address are non-zero, an Address Error exception occurs.

MIPS IV: The low-order 2 bits of the offset field must be zero. If they are not, the result of the instruction is undefined.

Operation: 32-bit Processors

vAddr ← sign_extend(offset) + GPR[base]
if (vAddr1..0) ≠ 02 then SignalException(AddressError) endif
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, STORE)
dataword ← GPR[rt]
StoreMemory (uncached, WORD, dataword, pAddr, vAddr, DATA)

Operation: 64-bit Processors

vAddr ← sign_extend(offset) + GPR[base]
if (vAddr1..0) ≠ 02 then SignalException(AddressError) endif
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, STORE)
pAddr ← pAddrPSIZE-1..3 || (pAddr2..0 xor (ReverseEndian || 02))
byte ← vAddrf2..0 xor (BigEndianCPU || 02)
datadouble ← GPR[rt]63-8*byte || 08*byte
StoreMemory (uncached, WORD, datadouble, pAddr, vAddr, DATA)

Exceptions:

TLB Refill, TLB Invalid
TLB Modified
Address Error
Store Word From Coprocessor $SWC_z$

<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>SWCz</td>
<td>base</td>
<td>rt</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1110zz</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Format:  
- SWC1 $rt$, offset(base)
- SWC2 $rt$, offset(base)
- SWC3 $rt$, offset(base)

Purpose: To store a word from a coprocessor general register to memory.

Description:  
$memory[base+offset] \leftarrow rt$

Coprocessor unit $zz$ supplies a 32-bit word which is stored at the memory 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.

The data supplied by each coprocessor is defined by the individual coprocessor specifications. The usual operation would read the data from coprocessor general register $rt$.

Each MIPS architecture level defines up to 4 coprocessor units, numbered 0 to 3 (see Coprocessor Instructions on page A-11). The opcodes corresponding to coprocessors that are not defined by an architecture level may be used for other instructions.

Restrictions:

Access to the coprocessors is controlled by system software. Each coprocessor has a “coprocessor usable” bit in the System Control coprocessor. The usable bit must be set for a user program to execute a coprocessor instruction. If the usable bit is not set, an attempt to execute the instruction will result in a Coprocessor Unusable exception. An unimplemented coprocessor must never be enabled. The result of executing this instruction for an unimplemented coprocessor when the usable bit is set, is undefined.

This instruction is not available for coprocessor 0, the System Control coprocessor, and the opcode may be used for other instructions.

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

MIPS IV: The low-order 2 bits of the $offset$ field must be zero. If they are not, the result of the instruction is undefined.

Operation: 32-bit processors

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

if $(vAddr_{1..0}) \neq 0^2$ then $\text{SignalException}(\text{AddressError})$ endif

$(pAddr, \text{uncached}) \leftarrow \text{AddressTranslation}(vAddr, \text{DATA}, \text{STORE})$

dataword $\leftarrow \text{COP\_SW}(z, rt)$

$\text{StoreMemory}(\text{uncached, WORD}, \text{dataword, pAddr, vAddr, DATA})$
SWCz

Store Word From Coprocessor

Operation: 64-bit processors

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

if \((v\text{Addr}_{1..0}) \neq 0^2\) then \text{SignalException}(\text{AddressError}) endif

\[
(p\text{Addr, uncached}) \leftarrow \text{AddressTranslation}(v\text{Addr, DATA, STORE})
\]

\[
p\text{Addr} \leftarrow p\text{Addr}_{P\text{SIZE}-1..3} || (p\text{Addr}_{2..0} \text{xor}(\text{ReverseEndian} || 0^2))
\]

\[
\text{byte} \leftarrow v\text{Addr}_{2..0} \text{xor}(\text{BigEndianCPU} || 0^2)
\]

\[
data\text{word} \leftarrow \text{COP\_SW}(z, rt)
\]

\[
data\text{double} \leftarrow 0^{32-8\text{byte}} || data\text{word} || 0^{8\text{byte}}
\]

\[
\text{StoreMemory(uncached, WORD, datadouble, pAddr, vAddr DATA)}
\]

Exceptions:

- TLB Refill, TLB Invalid
- TLB Modified
- Address Error
- Reserved Instruction
- Coprocessor Unusable
Store Word Left SWL

<table>
<thead>
<tr>
<th></th>
<th>SWL</th>
<th>rt</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>15</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Format: SWL rt, offset(base)  
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 four consecutive bytes forming a word in memory (W) starting at an arbitrary byte boundary. A part of W, the most-significant one to four 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.

If GPR rt is a 64-bit register, the source word is the low word of the register.

Figures A-4 illustrates this operation for big-endian byte ordering for 32-bit and 64-bit registers. The four consecutive bytes in 2..5 form an unaligned word starting at location 2. A part of W, two bytes, is contained in the aligned word containing the most-significant byte at 2. First, SWL stores the most-significant two bytes of the low-word from the source register into these two bytes in memory. Next, the complementary SWR stores the remainder of the unaligned word.

Figure A-8 Unaligned Word Store using SWL and SWR.
The bytes stored from the source register to memory depend on both the offset of the effective address within an aligned word, i.e. the low two bits of the address (vAddr\(_{1..0}\)), and the current byte ordering mode of the processor (big- or little-endian). The table below shows the bytes stored for every combination of offset and byte ordering.

**Table A-35  Bytes Stored by SWL 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 (\leftarrow) big-endian</td>
<td>64-bit register</td>
</tr>
<tr>
<td>i j k l</td>
<td>A B C D E F G H</td>
</tr>
<tr>
<td>3 2 1 0 (\leftarrow) little-endian</td>
<td>most significance least</td>
</tr>
<tr>
<td>most least</td>
<td>32-bit register E F G H</td>
</tr>
</tbody>
</table>

Memory contents after instruction (shaded is unchanged)

<table>
<thead>
<tr>
<th>Big-endian byte ordering</th>
<th>vAddr(_{1..0})</th>
<th>Little-endian byte ordering</th>
</tr>
</thead>
<tbody>
<tr>
<td>E F G H</td>
<td>0</td>
<td>i j k E</td>
</tr>
<tr>
<td>i E F G</td>
<td>1</td>
<td>i j E F</td>
</tr>
<tr>
<td>i j E F</td>
<td>2</td>
<td>i E F G</td>
</tr>
<tr>
<td>i j k E</td>
<td>3</td>
<td>E F G H</td>
</tr>
</tbody>
</table>

**Operation:** 32-bit Processors

vAddr \(\leftarrow\) sign_extend(offset) + GPR[base]

(pAddr, uncached) \(\leftarrow\) AddressTranslation (vAddr, DATA, STORE)

pAddr \(\leftarrow\) pAddr\((\text{PSIZE}-1)\ldots2\) || (pAddr\(_{1..0}\) xor ReverseEndian\(^2\))

If BigEndianMem = 0 then

pAddr \(\leftarrow\) pAddr\((\text{PSIZE}-1)\ldots2\) || 0\(^2\)

endif

byte \(\leftarrow\) vAddr\(_{1..0}\) xor BigEndianCPU\(^2\)

dataword \(\leftarrow\) \(0^{24-8\text{byte}}\) || GPR[rt]\(_{31..24-8\text{byte}}\)

StoreMemory (uncached, byte, dataword, pAddr, vAddr, DATA)
Store Word Left \textbf{SWL}

Operation: 64-bit Processors

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

\[
(pAddr, \text{uncached}) \leftarrow \text{AddressTranslation}(vAddr, \text{DATA, STORE})
\]

\[
pAddr \leftarrow pAddr_{(\text{PSIZE-1})..3} \parallel (pAddr_{2..0} \text{ xor } \text{ReverseEndian}^3)
\]

If BigEndianMem = 0 then

\[
pAddr \leftarrow pAddr_{(\text{PSIZE-1})..2} \parallel 0^2
\]

endif

\[
\text{byte} \leftarrow vAddr_{1..0} \text{ xor } \text{BigEndianCPU}^2
\]

if \((vAddr_{2} \text{ xor } \text{BigEndianCPU}) = 0\) then

\[
\text{datadouble} \leftarrow 0^{32} \parallel 0^{24-8*\text{byte}} \parallel \text{GPR}[rt]_{31..24-8*\text{byte}}
\]

else

\[
\text{datadouble} \leftarrow 0^{24-8*\text{byte}} \parallel \text{GPR}[rt]_{31..24-8*\text{byte}} \parallel 0^{32}
\]

endif

\[
\text{StoreMemory}(\text{uncached, byte, datadouble, pAddr, vAddr, DATA})
\]

Exceptions:

TLB Refill, TLB Invalid
TLB Modified
Bus Error
Address Error
SWR

Store Word Right

<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>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SWR</td>
<td>1 0 1 1 1 0</td>
<td>base</td>
<td>rt</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:**

SWR  rt, offset(base)

**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 four consecutive bytes forming a word in memory (W) starting at an arbitrary byte boundary. A part of W, the least-significant one to four 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.

If GPR rt is a 64-bit register, the source word is the low word of the register.

Figures A-4 illustrates this operation for big-endian byte ordering for 32-bit and 64-bit registers. The four consecutive bytes in 2..5 form an unaligned word starting at location 2. A part of W, two bytes, is contained in the aligned word containing the least-significant byte at 5. First, SWR stores the least-significant two bytes of the low-word from the source register into these two bytes in memory. Next, the complementary SWL stores the remainder of the unaligned word.

**Figure A-9  Unaligned Word Store using SWR and SWL.**
Store Word Right (SWR)

The bytes stored from the source register to memory depend on both the offset of the effective address within an aligned word, i.e. the low two bits of the address (vAddr_{1,0}), and the current byte ordering mode of the processor (big- or little-endian). The table below shows the bytes stored for every combination of offset and byte ordering.

Table A-36  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 ← big-endian</td>
<td>64-bit register</td>
</tr>
<tr>
<td>i  j  k  l</td>
<td></td>
</tr>
<tr>
<td>offset (vAddr_{1,0})</td>
<td>A  B  C  D  E  F  G  H</td>
</tr>
</tbody>
</table>

| 3  2  1  0 ← little-endian      | most — significance — least       |
| most least                      | 32-bit register                   |
| — significance —               |                                   |

<table>
<thead>
<tr>
<th>Memory contents after instruction (shaded is unchanged)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Big-endian byte ordering vAddr_{1,0}</td>
</tr>
<tr>
<td>--------------------------------------</td>
</tr>
<tr>
<td>H  j  k  l</td>
</tr>
<tr>
<td>G  H  k  l</td>
</tr>
<tr>
<td>F  G  H  l</td>
</tr>
<tr>
<td>E  F  G  H</td>
</tr>
</tbody>
</table>

Restrictions:
None

Operation:  32-bit Processors

vAddr ← sign_extend(offset) + GPR[base]
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, STORE)
pAddr ← pAddr\_{(P.SIZE-1..2)} || (pAddr\_{1,0} xor ReverseEndian^2)
BigEndianMem = 0 then
pAddr ← pAddr\_{(P.SIZE-1..2)} || 0^2
endif

byte ← vAddr\_{1,0} xor BigEndianCPU^2
dataword ← GPR[rt]_{31–8*byte} || 0^{8*byte}

StoreMemory (uncached, WORD-byte, dataword, pAddr, vAddr, DATA)
Operation: 64-bit Processors

vAddr ← sign_extend(offset) + GPR[base]
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, STORE)
pAddr ← pAddr(PSIZE-1).3 || (pAddr2..0 xor ReverseEndian3)
If BigEndianMem = 0 then
   pAddr ← pAddr(PSIZE-1).2 || 0^2
endif
byte ← vAddr1..0 xor BigEndianCPU^2
if (vAddr2 xor BigEndianCPU) = 0 then
   datadouble ← 0^32 || GPR[rt]31-8*byte..0 || 0^8*byte
else
   datadouble ← GPR[rt]31-8*byte..0 || 0^8*byte || 0^32
endif
StoreMemory(uncached, WORD-byte, datadouble, pAddr, vAddr, DATA)

Exceptions:
TLB Refill, TLB Invalid
TLB Modified
Bus Error
Address Error
# Synchronize Shared Memory

**SYNC**

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</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>000000</td>
<td>000000</td>
<td>0000</td>
<td>stype</td>
<td>SYNC 001111</td>
</tr>
<tr>
<td>6</td>
<td>15</td>
<td>5</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:**

SYNC \((\text{stype} = 0 \text{ implied})\)  

**MIPS II**

**Purpose:**

To order loads and stores to shared memory in a multiprocessor system.

**Description:**

To serve a broad audience, two descriptions are given. A simple description of SYNC that appeals to intuition is followed by a precise and detailed description.

**A Simple Description:**

SYNC affects only uncached and cached coherent loads and stores. The loads and stores that occur prior to 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.

**A Precise Description:**

If the \(\text{stype}\) field has a value of zero, every synchronizable load and store that occurs in the instruction stream prior to the SYNC instruction must be globally performed before any synchronizable load or store that occurs after the SYNC may 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 \(\text{stype}\) values 1-31 are reserved; they produce the same result as the value zero.

**Synchronizable:** 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.

Memory Access Types on page A-12 contains information on memory access types.

**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.
SYNC
Synchronize Shared Memory

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 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 the effects of loads and stores for memory access types other than uncached and cached coherent is not defined.

Operation:

SyncOperation(stype)

Exceptions:

Reserved Instruction

Programming Notes:

A processor executing load and store instructions observes the effects of the loads and stores that use the same memory access type in the order that they occur in the instruction stream; this is known as program order. A parallel program has multiple instruction streams that can execute at the same time 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 stores, 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 is also not 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 in order 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 effects of these groups are seen in program order by all processors. 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 other 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 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 will generally not operate on a system that is not strongly ordered, however a program that does use SYNC will work on both types of systems. System-specific documentation will describe the actions necessary 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. See page A-13 for a more complete discussion.

SYNC affects the order in which the effects of load and store instructions appears to all processors; it 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, is not defined.

Prefetch operations have no effects detectable by user-mode programs so ordering the effects of prefetch operations is not meaningful.
SYNC

Synchronize Shared Memory

EXAMPLE: These code fragments show 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.

<table>
<thead>
<tr>
<th>Processor A (writer)</th>
</tr>
</thead>
<tbody>
<tr>
<td># Conditions at entry:</td>
</tr>
<tr>
<td># The value 0 has been stored in FLAG and that value is observable by B.</td>
</tr>
<tr>
<td>SW R1, DATA # change shared DATA value</td>
</tr>
<tr>
<td>LI R2, 1</td>
</tr>
<tr>
<td>SYNC # perform DATA store before performing FLAG store</td>
</tr>
<tr>
<td>SW R2, FLAG # say that the shared DATA value is valid</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Processor B (reader)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LI R2, 1</td>
</tr>
<tr>
<td>1: LW R1, FLAG # get FLAG</td>
</tr>
<tr>
<td>BNE R2, R1, 1B # if it says that DATA is not valid, poll again</td>
</tr>
<tr>
<td>NOP</td>
</tr>
<tr>
<td>SYNC # FLAG value checked before doing DATA reads</td>
</tr>
<tr>
<td>LW R1, DATA # read (valid) shared DATA values</td>
</tr>
</tbody>
</table>

Implementation Notes:

There may be side effects of uncached loads and stores that affect cached coherent load and store operations. To permit the reliable use of such side effects, buffered uncached stores that occur before the SYNC must be written to memory before cached coherent loads and stores after the SYNC may be performed.
**System Call** SYSCALL

<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>SPECIAL</td>
<td>Code</td>
<td>SYSCALL</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 0 0 0 0</td>
<td></td>
<td>0 0 1 1 0 0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** SYSCALL

**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
TEQ  Trap if Equal

MIPS II

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:
  Reserved Instruction
  Trap
Trap if Equal Immediate \textbf{TEQI}

Format: \textbf{TEQI} \text{ rs, immediate}  

Purpose: To compare a GPR to a constant and do a conditional Trap.

Description: if (rs = immediate) then Trap

\begin{itemize}
  \item Compare the contents of GPR \textit{rs} and the 16-bit signed \textit{immediate} as signed integers; if 
  GPR \textit{rs} is equal to \textit{immediate} then take a Trap exception.
\end{itemize}

Restrictions:

None

Operation:

\begin{verbatim}
  if GPR[rs] = sign_extend(immediate) then
    SignalException(Trap)
  endif
\end{verbatim}

Exceptions:

\begin{itemize}
  \item Reserved Instruction
  \item Trap
\end{itemize}
**TGE** Trap if Greater or Equal

**Format:**
```
TGE   rs, rt
```

**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 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:**
```
if GPR[rs] \geq GPR[rt] then
    SignalException(Trap)
endif
```

**Exceptions:**
- Reserved Instruction
- Trap
### Trap if Greater or Equal Immediate (TGEI)

**Format:**

```
TGEI   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 greater than or equal to `immediate` then take a Trap exception.

**Restrictions:**

None

**Operation:**

```
if GPR[rs] ≥ sign_extend(immediate) then
    SignalException(Trap)
endif
```

**Exceptions:**

- Reserved Instruction
- Trap

---

<table>
<thead>
<tr>
<th>REGIMM</th>
<th>rs</th>
<th>TGEI</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 1</td>
<td>6 5</td>
<td>0 1 0 0 0</td>
<td>16</td>
</tr>
</tbody>
</table>
**TGEIU**

**Trap If Greater Or Equal Immediate Unsigned**

**Format:**

```
TGEIU   rs, immediate
```

**MIPS II**

**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 is able to 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:**

- Reserved Instruction
- Trap
**Trap If Greater or Equal Unsigned TGEU**

<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>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL</td>
<td>0 0 0 0 0 0</td>
<td>rs</td>
<td>rt</td>
<td>code</td>
<td>TGEU</td>
<td>1 1 0 0 0 0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Format: TGEU rs, rt

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 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:

\[
\text{if } (0 || \text{GPR}[rs]) \geq (0 || \text{GPR}[rt]) \text{ then}
\]

SignalException(Trap)

endif

Exceptions:
Reserved Instruction
Trap
**TLT** Trap if Less Than

**MIPS II**

<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 0 0 0 0 0 0</td>
<td>rs</td>
<td>rt</td>
<td>code</td>
<td>TLT</td>
<td>1 1 0 0 1 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**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:**

- Reserved Instruction
- Trap
Trap if Less Than Immediate 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:

   Reserved Instruction
   Trap
TLTIU

Trap if Less Than Immediate Unsigned

31 26 25 21 20 16 15 0

REGIMM | rs | TLTIU | immediate
---|---|---|---
0 0 0 0 0 1 | 0 1 0 1 1 | 0 1 0 1 1

6 5 5 16

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 is able to 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:

Reserved Instruction
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></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 0 0 0 0</td>
<td>11 0 0 1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** TLTU rs, rt

**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:**

if (0 || GPR[rs]) < (0 || GPR[rt]) then
    SignalException(Trap)
endif

**Exceptions:**

Reserved Instruction
Trap
**TNE**  
**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>0 0 0 0 0 0</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>

**MIPS II**

**Format:**

TNE 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 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:**

if GPR[rs] ≠ GPR[rt] then
    SignalException(Trap)
endif

**Exceptions:**

Reserved Instruction
Trap
Trap if Not Equal Immediate \textbf{TNEI}

\begin{center}
\begin{tabular}{|c|c|c|c|c|c|c|c|}
\hline
31 & 26 & 25 & 21 & 20 & 16 & 15 & 0 \\
\hline
REGIMM & rs & TNEI & immediate & & & & \\
0 & 0 & 0 & 0 & 0 & 1 & & \\
\hline
6 & 5 & 5 & 16 & & & & \\
\hline
\end{tabular}
\end{center}

Format: \textbf{TNEI rs, immediate}

Purpose: To compare a GPR to a constant and do a conditional Trap.

Description: if \((rs \neq \text{immediate})\) then Trap

\hspace{1cm} Compare the contents of GPR \(rs\) and the 16-bit signed \textit{immediate} as signed integers; if GPR \(rs\) is not equal to \textit{immediate} then take a Trap exception.

Restrictions:
None

Operation:
\begin{verbatim}
if GPR[rs] \neq \text{sign\_extend}(\text{immediate}) then
  \text{SignalException(Trap)}
endif
\end{verbatim}

Exceptions:
- Reserved Instruction
- Trap
**XOR**

**Exclusive 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>XOR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 0 0 0 0</td>
<td>0 0 0 0 0 1 0 0 1 1 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** XOR rd, rs, rt

**Purpose:** To do a bitwise logical EXCLUSIVE OR.

**Description:**

\[
rd \leftarrow rs \text{ XOR } rt
\]

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:**

\[
\text{GPR}[rd] \leftarrow \text{GPR}[rs] \text{ XOR } \text{GPR}[rt]
\]

**Exceptions:**

None
**Exclusive OR Immediate XORI**

<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>XORI</td>
<td>rs</td>
<td>rt</td>
<td>immediate</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 1 1 1 0</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** XORI  rt, rs, immediate  
**MIPS I**

**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
A 7  CPU Instruction Formats

A CPU instruction is a single 32-bit aligned word. The major instruction formats are shown in Figure A-10.

I-Type (Immediate).

<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>opcode</td>
<td>rs</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>

J-Type (Jump).

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

R-Type (Register).

<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>opcode</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>sa</td>
<td>function</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>
</tr>
</tbody>
</table>

- **opcode**: 6-bit primary operation code
- **rd**: 5-bit destination register specifier
- **rs**: 5-bit source register specifier
- **rt**: 5-bit target (source/destination) register specifier or used to specify functions within the primary opcode value REGIMM
- **immediate**: 16-bit signed immediate used for: logical operands, arithmetic signed operands, load/store address byte offsets, PC-relative branch signed instruction displacement
- **instr_index**: 26-bit index shifted left two bits to supply the low-order 28 bits of the jump target address.
- **sa**: 5-bit shift amount
- **function**: 6-bit function field used to specify functions within the primary operation code value SPECIAL.

*Figure A-10  CPU Instruction Formats*
A 8  CPU Instruction Encoding

This section describes the encoding of user-level, i.e. non-privileged, CPU instructions for the four levels of the MIPS architecture, MIPS I through MIPS IV. Each architecture level includes the instructions in the previous level;† MIPS IV includes all instructions in MIPS I, MIPS II, and MIPS III. This section presents eight different views of the instruction encoding.

• Separate encoding tables for each architecture level.
• A MIPS IV encoding table showing the architecture level at which each opcode was originally defined and subsequently modified (if modified).
• Separate encoding tables for each architecture revision showing the changes made during that revision.

A 8.1  Instruction Decode

Instruction field names are printed in bold in this section.

The primary opcode field is decoded first. Most opcode values completely specify an instruction that has an immediate value or offset. Opcode values that do not specify an instruction specify an instruction class. Instructions within a class are further specified by values in other fields. The opcode values SPECIAL and REGIMM specify instruction classes. The COP0, COP1, COP2, COP3, and COP1X instruction classes are not CPU instructions; they are discussed in section A 8.3.

A 8.1.1  SPECIAL Instruction Class

The opcode =SPECIAL instruction class encodes 3-register computational instructions, jump register, and some special purpose instructions. The class is further decoded by examining the format field. The format values fully specify the CPU instructions; the MOVCI instruction class is not a CPU instruction class.

A 8.1.2  REGIMM Instruction Class

The opcode =REGIMM instruction class encodes conditional branch and trap immediate instructions. The class is further decode, and the instructions fully specified, by examining the rt field.

A 8.2  Instruction Subsets of MIPS III and MIPS IV Processors.

MIPS III processors, such as the R4000, R4200, R4300, R4400, and R4600, have a processor mode in which only the MIPS II instructions are valid. The MIPS II encoding table describes the MIPS II-only mode except that the Coprocessor 3 instructions (COP3, LWC3, SWC3, LDC3, SDC3) are not available and cause a Reserved Instruction exception.

† An exception to this rule is that the reserved, but never implemented, Coprocessor 3 instructions were removed or changed to another use starting in MIPS III.
MIPS IV processors, such as the R8000 and R10000, have processor modes in which only the MIPS II or MIPS III instructions are valid. The MIPS II encoding table describes the MIPS II-only mode except that the Coprocessor 3 instructions (COP3, LWC3, SWC3, LDC3, SDC3) are not available and cause a Reserved Instruction exception. The MIPS III encoding table describes the MIPS III-only mode.

A 8.3 Non-CPU Instructions in the Tables

The encoding tables show all values for the field they describe and by doing this they include some entries that are not user-level CPU instructions. The primary opcode table includes coprocessor instruction classes (COP0, COP1, COP2, COP3/COP1X) and coprocessor load/store instructions (LWCx, SWCx, LDCx, SDCx for x=1, 2, or 3). The opcode =SPECIAL + function =MOVCI instruction class is an FPU instruction.

A 8.3.1 Coprocessor 0 - COP0

COP0 encodes privileged instructions for Coprocessor 0, the System Control Coprocessor. The definition of the System Control Coprocessor is processor-specific and further information on these instructions are not included in this document.

A 8.3.2 Coprocessor 1 - COP1, COP1X, MOVCI, and CP1 load/store.

Coprocessor 1 is the floating-point unit in the MIPS architecture. COP1, COP1X, and the (opcode =SPECIAL + function =MOVCI) instruction classes encode floating-point instructions. LWC1, SWC1, LDC1, and SDC1 are floating-point loads and stores. The FPU instruction encoding is documented in section B.12.

A 8.3.3 Coprocessor 2 - COP2 and CP2 load/store.

Coprocessor 2 is optional and implementation-specific. No standard processor from MIPS has implemented coprocessor 2, but MIPS' semiconductor licensees may have implemented it in a product based on one of the standard MIPS processors. At this time the standard processors are: R2000, R3000, R4000, R4200, R4300, R4400, R4600, R6000, R8000, and R10000.

A 8.3.4 Coprocessor 3 - COP3 and CP3 load/store.

Coprocessor 3 is optional and implementation-specific in the MIPS I and MIPS II architecture levels. It was removed from MIPS III and later architecture levels. Note that in MIPS IV the COP3 primary opcode was reused for the COP1X instruction class. No standard processor from MIPS has implemented coprocessor 2, but MIPS' semiconductor licensees may have implemented it in a product based on one of the standard MIPS processors. At this time the standard processors are: R2000, R3000, R4000, R4200, R4300, R4400, R4600, R6000, R8000, and R10000.
Table A-37  CPU Instruction Encoding - MIPS I Architecture

<table>
<thead>
<tr>
<th>opcod</th>
<th>bits 28..26</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode</td>
<td>Instructions encoded by opcode field.</td>
</tr>
<tr>
<td>bits</td>
<td>0</td>
</tr>
<tr>
<td>31..29</td>
<td>000</td>
</tr>
<tr>
<td>0</td>
<td>SPECIAL δ</td>
</tr>
<tr>
<td>1</td>
<td>ADDI</td>
</tr>
<tr>
<td>2</td>
<td>COP0 δ,π</td>
</tr>
<tr>
<td>3</td>
<td>011</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
</tr>
<tr>
<td>5</td>
<td>101</td>
</tr>
<tr>
<td>6</td>
<td>110</td>
</tr>
<tr>
<td>7</td>
<td>111</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>functi</th>
<th>bits 2..0</th>
</tr>
</thead>
<tbody>
<tr>
<td>function</td>
<td>Instructions encoded by function field when opcode field = SPECIAL.</td>
</tr>
<tr>
<td>bits</td>
<td>0</td>
</tr>
<tr>
<td>5..3</td>
<td>000</td>
</tr>
<tr>
<td>0</td>
<td>SLL</td>
</tr>
<tr>
<td>1</td>
<td>JR</td>
</tr>
<tr>
<td>2</td>
<td>MFHI</td>
</tr>
<tr>
<td>3</td>
<td>MULT</td>
</tr>
<tr>
<td>4</td>
<td>ADD</td>
</tr>
<tr>
<td>5</td>
<td>101</td>
</tr>
<tr>
<td>6</td>
<td>110</td>
</tr>
<tr>
<td>7</td>
<td>111</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>rt</th>
<th>bits 18..16</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode</td>
<td>Instructions encoded by the rt field when opcode field = REGIMMM.</td>
</tr>
<tr>
<td>bits</td>
<td>0</td>
</tr>
<tr>
<td>20..19</td>
<td>000</td>
</tr>
<tr>
<td>0</td>
<td>BLTZ</td>
</tr>
<tr>
<td>1</td>
<td>=</td>
</tr>
<tr>
<td>2</td>
<td>BLTZAL</td>
</tr>
<tr>
<td>3</td>
<td>=</td>
</tr>
</tbody>
</table>
Table A-38  CPU Instruction Encoding - MIPS II Architecture

<table>
<thead>
<tr>
<th>opcod</th>
<th>opcode</th>
<th>bits 28..26</th>
<th>Instructions encoded by opcode field.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>31 26 0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>000</td>
<td>SPECIAL δ</td>
<td>REGIMM δ</td>
</tr>
<tr>
<td>1</td>
<td>001</td>
<td>ADDI</td>
<td>ADDIU</td>
</tr>
<tr>
<td>2</td>
<td>010</td>
<td>COP0 δ,π</td>
<td>COP1 δ,π</td>
</tr>
<tr>
<td></td>
<td>011</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>LB</td>
<td>LH</td>
</tr>
<tr>
<td>5</td>
<td>101</td>
<td>SB</td>
<td>SH</td>
</tr>
<tr>
<td>6</td>
<td>110</td>
<td>LL</td>
<td>LWC1 δ</td>
</tr>
<tr>
<td>7</td>
<td>111</td>
<td>SC</td>
<td>SWC1 δ</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>functi</th>
<th>function</th>
<th>bits 2..0</th>
<th>Instructions encoded by function field when opcode field = SPECIAL.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>31 26 0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>000</td>
<td>SLL</td>
<td>*</td>
</tr>
<tr>
<td>1</td>
<td>001</td>
<td>JR</td>
<td>JALR</td>
</tr>
<tr>
<td>2</td>
<td>010</td>
<td>MFHI</td>
<td>MTHI</td>
</tr>
<tr>
<td>3</td>
<td>011</td>
<td>MULT</td>
<td>MULTU</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>ADD</td>
<td>ADDU</td>
</tr>
<tr>
<td>5</td>
<td>101</td>
<td>*</td>
<td>SLT</td>
</tr>
<tr>
<td>6</td>
<td>110</td>
<td>TGE</td>
<td>TGEU</td>
</tr>
<tr>
<td>7</td>
<td>111</td>
<td>*</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>rt</th>
<th>rt</th>
<th>bits 18..16</th>
<th>Instructions encoded by the rt field when opcode field = REGIMM.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>31 26 0 20 16 0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>00</td>
<td>BLTZ</td>
<td>BGEZ</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>TGEI</td>
<td>TGEIU</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>BLTZAL</td>
<td>BGEZAL</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>*</td>
<td>*</td>
</tr>
</tbody>
</table>

|         |         |           |                                             |
| opcode |       | 31 26 0   |                                               |
| rt      |       | 31 26 0   |                                               |
| rt      |       | 31 26 0   |                                               |

A-178  MIPS IV Instruction Set. Rev 3.2  CPU Instruction Set
### CPU Instruction Encoding - MIPS III Architecture

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>opcode field</th>
<th>function field</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL</td>
<td>REGIMM</td>
<td>000</td>
<td>SLL * SRL * SRA * SLLV * SRLV * SRAV</td>
</tr>
<tr>
<td>ADDI</td>
<td>JAL</td>
<td>010</td>
<td>JR * JALR * SYSCALL * BREAK * SYNC</td>
</tr>
<tr>
<td>COP0 &amp; COP1 &amp; COP2</td>
<td>BEQ</td>
<td>011</td>
<td>MFHI * MTHI * MFLO * MTLO * DSSLV * DSRLV * DSRAV</td>
</tr>
<tr>
<td>DADDI</td>
<td>LSL</td>
<td>100</td>
<td>ADDI</td>
</tr>
<tr>
<td>LB</td>
<td>LWL</td>
<td>101</td>
<td>ADD</td>
</tr>
<tr>
<td>LL</td>
<td>LW</td>
<td>110</td>
<td>TGE</td>
</tr>
<tr>
<td>SC</td>
<td>SWC1</td>
<td>111</td>
<td>DSLL * DSRL * DSRA * DSLL32 * DSRL32 * DSRA32</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>opcode field</th>
<th>rt field</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDI</td>
<td>JAL</td>
<td>000</td>
<td>BLTZ BGEZ BLTZL BGEZL * * *</td>
</tr>
<tr>
<td>TGEI</td>
<td>TGEU</td>
<td>011</td>
<td>BLTZAL BGEZAL BLTZALL BGEZALL * * *</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>111</td>
<td>* * * * * * *</td>
</tr>
</tbody>
</table>

---

**CPU Instruction Set**  
**MIPS IV Instruction Set. Rev 3.2**  
_A-179_
Table A-40  CPU Instruction Encoding - MIPS IV Architecture

<table>
<thead>
<tr>
<th>opcode</th>
<th>bits 28..26</th>
<th>Instructions encoded by opcode field.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000</td>
<td>SPECIAL δ, REGIMM δ, J, JAL, BEQ, BNE, BLEZ, BGTZ</td>
</tr>
<tr>
<td>1</td>
<td>001</td>
<td>ADDI, ADDIU, SLTI, SLTIU, ANDI, ORI, XORI, LUI</td>
</tr>
<tr>
<td>2</td>
<td>010</td>
<td>COP0 δ, π, COP1 δ, π, COP2 δ, π, COP1X δ, π, BEQL, BNEL, BLEZL, BGTZL</td>
</tr>
<tr>
<td>3</td>
<td>011</td>
<td>DADDI, DADDIU, LDL, LDR, * , *, *</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>LB, LH, LWL, LW, LBU, LHU, LWR, LWU</td>
</tr>
<tr>
<td>5</td>
<td>101</td>
<td>SB, SH, SWL, SW, SDL, SDR, SWR, *</td>
</tr>
<tr>
<td>6</td>
<td>110</td>
<td>LL, LWC1 π, LWC2 π, PREF, LLD, LDC1 π, LDC2 π, LD</td>
</tr>
<tr>
<td>7</td>
<td>111</td>
<td>SC, SWC1 π, SWC2 π, * , SCD, SDC1 π, SDC2 π, SD</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>function</th>
<th>bits 2..0</th>
<th>Instructions encoded by function field when opcode field = SPECIAL.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000</td>
<td>SLL, MOVCI δ, µ, SRL, SRA, SLTV, * , SRLV, SRAV</td>
</tr>
<tr>
<td>1</td>
<td>001</td>
<td>JR, JALR, MOVZ, MOVN, SYSCALL, BREAK, * , SYNC</td>
</tr>
<tr>
<td>2</td>
<td>010</td>
<td>MFHI, MTHI, MFLO, MILO, DSLLV, * , DSRLV, DSRAV</td>
</tr>
<tr>
<td>3</td>
<td>011</td>
<td>MULT, MULTU, DIV, DIVU, DMULT, DMULTU, DDIV, DDIVU</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>ADD, ADDU, SUB, SUBU, AND, OR, XOR, NOR</td>
</tr>
<tr>
<td>5</td>
<td>101</td>
<td>* , * , SLT, SLTU, DADD, DADDDU, DSUB, DSUBU</td>
</tr>
<tr>
<td>6</td>
<td>110</td>
<td>TGE, TGEU, TLTI, TLTIU, TEQ, * , TNE, *</td>
</tr>
<tr>
<td>7</td>
<td>111</td>
<td>DSLL, * , DSRL, DSRA, DSSL32, * , DSRL32, DSRA32</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>rt</th>
<th>bits 18..16</th>
<th>Instructions encoded by the rt field when opcode field = REGIMM.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>BLTZ, BGEZ, BLTZL, BGEZL, * , * , * , *</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>TGEI, TGEIU, TLTI, TLTIU, TEQI, * , TNEI, *</td>
</tr>
</tbody>
</table>
Table A-41  Architecture Level in Which CPU Instructions are Defined or Extended.

| CPU Instruction Set | MIPS IV Instruction Set. Rev 3.2 | A-181 |
The architecture level in which each MIPS IV encoding was defined is indicated by a subscript 1, 2, 3, or 4 (for architecture level I, II, III, or IV). If an instruction or instruction class was later extended, the extending level is indicated after the defining level.

<table>
<thead>
<tr>
<th>opcode</th>
<th>function</th>
<th>rt</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 000</td>
<td>SPECIAL</td>
<td>REGIMM</td>
<td>J1</td>
<td>JAL1</td>
<td>BEQ1</td>
<td>BNE1</td>
<td>BLEZ1</td>
<td>BGTLZ1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 001</td>
<td>ADDI1</td>
<td>ADDIU1</td>
<td>SLTI1</td>
<td>SLTIU1</td>
<td>ANDI1</td>
<td>OR1</td>
<td>XOR1</td>
<td>LUI1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 010</td>
<td>COP1</td>
<td>COP1</td>
<td>COP1</td>
<td>COP1</td>
<td>BEQ1</td>
<td>BNE1</td>
<td>BLEZ1</td>
<td>BGTLZ1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 011</td>
<td>DADD1</td>
<td>DADDIU1</td>
<td>LDR3</td>
<td>LDR3</td>
<td>*1</td>
<td>*1</td>
<td>*1</td>
<td>*1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 100</td>
<td>LB1</td>
<td>LH1</td>
<td>LW1</td>
<td>LW1</td>
<td>LBU1</td>
<td>LHI1</td>
<td>LWR1</td>
<td>LWU1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5 010</td>
<td>SB1</td>
<td>SH1</td>
<td>SW1</td>
<td>SW1</td>
<td>SDL3</td>
<td>SDR3</td>
<td>SWR1</td>
<td>P2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6 110</td>
<td>LL1</td>
<td>LWC1</td>
<td>LWC1</td>
<td>LWC1</td>
<td>PRE1</td>
<td>LDC1</td>
<td>LDC2</td>
<td>LD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7 111</td>
<td>SC1</td>
<td>SWC1</td>
<td>SWC1</td>
<td>SWC1</td>
<td>*3</td>
<td>SCD3</td>
<td>SDC1</td>
<td>SDC2</td>
<td>SD</td>
<td></td>
</tr>
</tbody>
</table>

Instructions encoded by the rt field when opcode field = REGIMM.
Table A-42  CPU Instruction Encoding Changes - MIPS II Revision.

<table>
<thead>
<tr>
<th>opcode</th>
<th>bits 28..26</th>
<th>Instructions encoded by opcode field.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0 1 2 3 4 5 6 7</td>
<td>BEQL  BNEL  BLEZL  BGTZL</td>
</tr>
<tr>
<td>0 000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 001</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 010</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 011</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5 101</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6 110</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7 111</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>function</th>
<th>bits 2..0</th>
<th>Instructions encoded by function field when opcode field = SPECIAL.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0 1 2 3 4 5 6 7</td>
<td>SYNC  TGE  TGEU  TLT  TLTI  TLTIU  TEQ  TNE</td>
</tr>
<tr>
<td>0 000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 001</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 010</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 011</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5 101</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6 110</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7 111</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>rt</th>
<th>bits 18..16</th>
<th>Instructions encoded by the rt field when opcode field = REGIMM.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0 1 2 3 4 5 6 7</td>
<td>BLTZL  BGEZL  TGEI  TGEIU  TLTI  TLTIU  TEQI  TNEI  BLTZALL  BGEZALL</td>
</tr>
<tr>
<td>0 00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 11</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

An instruction encoding is shown if the instruction is added in this revision.
Table A-43  CPU Instruction Encoding Changes - MIPS III Revision.
An instruction encoding is shown if the instruction is added or modified in this revision.

### Opcode Field

<table>
<thead>
<tr>
<th>Bits (31..26)</th>
<th>Instructions encoded by opcode field.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 000</td>
<td></td>
</tr>
<tr>
<td>1 001</td>
<td></td>
</tr>
<tr>
<td>2 010</td>
<td>* (was COP3)</td>
</tr>
<tr>
<td>3 011</td>
<td>DADDI, DADDIU, LDL, LDR</td>
</tr>
<tr>
<td>4 100</td>
<td>LWU</td>
</tr>
<tr>
<td>5 101</td>
<td>SDL, SDR</td>
</tr>
<tr>
<td>6 110</td>
<td>* (was LWC3)</td>
</tr>
<tr>
<td>7 111</td>
<td>* (was SWC3)</td>
</tr>
</tbody>
</table>

### Function Field

<table>
<thead>
<tr>
<th>Bits (2..0)</th>
<th>Instructions encoded by function field when opcode field = SPECIAL.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 000</td>
<td></td>
</tr>
<tr>
<td>1 001</td>
<td></td>
</tr>
<tr>
<td>2 010</td>
<td>DSLLV, DSRLV, DSRAV</td>
</tr>
<tr>
<td>3 011</td>
<td>DMULT, DMULTU, DDIV, DDIVU</td>
</tr>
<tr>
<td>4 100</td>
<td>DADD, DADDU, DSUB, DSUBU</td>
</tr>
<tr>
<td>5 101</td>
<td></td>
</tr>
<tr>
<td>6 110</td>
<td></td>
</tr>
<tr>
<td>7 111</td>
<td>DSLL, DSRL, DSRA, DSLL32, DSRL32, DSRA32</td>
</tr>
</tbody>
</table>
Instructions encoded by the `rt` field when opcode field = REGIMM.
Table A-44  CPU Instruction Encoding Changes - MIPS IV Revision.

An instruction encoding is shown if the instruction is added or modified in this revision.

<table>
<thead>
<tr>
<th>opcode bits 28..26</th>
<th>Instructions encoded by opcode field.</th>
<th>function bits 2..0</th>
<th>Instructions encoded by function field when opcode field = SPECIAL.</th>
<th>rt bits 18..16</th>
<th>Instructions encoded by the rt field when opcode field = REGIMM.</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcod</td>
<td>31 26</td>
<td>0 1 2 3 4 5 6 7</td>
<td>functi</td>
<td>31 26</td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
</tbody>
</table>
Key to notes in CPU instruction encoding tables:

* This opcode is reserved for future use. An attempt to execute it causes a Reserved Instruction exception.

= This opcode is reserved for future use. An attempt to execute it produces an undefined result. The result may be a Reserved Instruction exception but this is not guaranteed.

δ (also italic opcode name) This opcode indicates an instruction class. The instruction word must be further decoded by examining additional tables that show values for another instruction field.

π This opcode is a coprocessor operation, not a CPU operation. If the processor state does not allow access to the specified coprocessor, the instruction causes a Coprocessor Unusable exception. It is included in the table because it uses a primary opcode in the instruction encoding map.

κ This opcode is removed in a later revision of the architecture. If a MIPS III or MIPS IV processor is operated in MIPS II-only mode this opcode will cause a Reserved Instruction exception.

μ This opcode indicates a class of coprocessor 1 instructions. If the processor state does not allow access to coprocessor 1, the opcode causes a Coprocessor Unusable exception. It is included in the table because the encoding uses a location in what is otherwise a CPU instruction encoding map. Further encoding information for this instruction class is in the FPU Instruction Encoding tables.

ρ This opcode is reserved for Coprocessor 0 (System Control Coprocessor) instructions that require base+offset addressing. If the instruction is used for COP0 in an implementation, an attempt to execute it without Coprocessor 0 access privilege will cause a Coprocessor Unusable exception. If the instruction is not used in an implementation, it will cause a Reserved Instruction exception.
B 1 Introduction

This appendix describes the instruction set architecture (ISA) for the floating-point unit (FPU) in the MIPS IV architecture. In the MIPS architecture, the FPU is coprocessor 1, an optional processor implementing IEEE Standard 754† floating-point operations. The FPU also provides a few additional operations not defined by the IEEE standard.

The original MIPS I FPU ISA has been extended in a backward-compatible fashion three times. The ISA extensions are inclusive as the diagram illustrates; each new architecture level (or version) includes the former levels. The description of an architectural feature includes the architecture level in which the feature is (first) defined or extended. The feature is also available in all later (higher) levels of the architecture.

In addition to an ISA, the architecture definition includes processing resources, such as the coprocessor general register set. The 32-bit registers in MIPS I were changed to 64-bit registers in MIPS III in a way that is not backwards compatible. For changes such as this, processors implementing higher levels of the architecture have a way to provide the processing resource model for earlier levels. For the FPU there is a mode to select the 32-bit or 64-bit register model. The practical result is that a processor implementing MIPS IV is also able to run MIPS I, MIPS II, or MIPS III binary programs without change.

If coprocessor 1 is not enabled, an attempt to execute a floating-point instruction will cause a Coprocessor Unusable exception. Enabling coprocessor 1 is a privileged operation provided by the System Control Coprocessor. Every system environment will either enable the FPU automatically or provide a means for an application to request that it be enabled.

Before the instruction set is described, there is an overview of the FPU data types, registers, and computational model. The FPU instruction set is summarized by functional group then each operation is described separately in alphabetical order. The description concludes with the FPU instruction formats and opcode encoding tables. See the CPU instruction set section titled “Description of an Instruction” for a description of the organization of the individual instruction descriptions and the notation used in them.

The architecture of the floating-point coprocessor consists of:

- Data types
- Operations
- A computational model
- Processing resources (registers)
- An instruction set

The IEEE standard defines the floating-point number data types, the basic arithmetic, comparison, and conversion operations, and a computational model.

The IEEE standard defines neither specific processing resources nor an instruction set. The MIPS architecture defines fixed-point (integer) data types, FPU register sets, control and exception mechanisms, and an instruction set. The architecture include non-IEEE FPU control operations, and arithmetic operations (multiply-add, reciprocal, and reciprocal square root) that may not supply results that match the IEEE precision rules.

### B 2 FPU Data Types

The FPU provides both floating-point and fixed-point data types. The single and double precision floating-point data types are those specified by the IEEE standard. The fixed-point types are the signed integers provided by the CPU architecture.
B 2.1 Floating-point formats

There are two floating-point data types provided by the FPU.
- 32-bit Single precision floating-point (type S)
- 64-bit Double precision floating-point (type D)

The floating-point formats represent numeric values as well as other special entities:
1. Numbers of the form: 
   \((-1)^s \times 2^E \cdot b_1 b_2 \ldots b_{p-1}\)
   where (see Table B-1):
   - \(s = 0\) or \(1\)
   - \(E\) is any integer between \(E_{min}\) and \(E_{max}\), inclusive
   - \(b_1 = 0\) or \(1\) (the high bit, \(b_0\), is to the left of the binary point)
   - \(p\) is the precision
2. Two infinities, \(+\infty\) and \(-\infty\)
3. Signaling non-numbers (SNaNs)
4. Quiet non-numbers (QNaNs)

Table B-1 Parameters of Floating-Point Formats

<table>
<thead>
<tr>
<th>parameter</th>
<th>Single</th>
<th>Double</th>
</tr>
</thead>
<tbody>
<tr>
<td>bits of mantissa precision, (p)</td>
<td>24</td>
<td>53</td>
</tr>
<tr>
<td>maximum exponent, (E_{max})</td>
<td>+127</td>
<td>+1023</td>
</tr>
<tr>
<td>minimum exponent, (E_{min})</td>
<td>-126</td>
<td>-1022</td>
</tr>
<tr>
<td>exponent bias</td>
<td>+127</td>
<td>+1023</td>
</tr>
<tr>
<td>bits in exponent field, (e)</td>
<td>8</td>
<td>11</td>
</tr>
<tr>
<td>representation of (b_0) integer bit</td>
<td>hidden</td>
<td>hidden</td>
</tr>
<tr>
<td>bits in fraction field, (f)</td>
<td>23</td>
<td>52</td>
</tr>
<tr>
<td>total format width in bits</td>
<td>32</td>
<td>64</td>
</tr>
</tbody>
</table>

The single and double floating-point formats are composed of three fields whose size is listed in Table B-1. The layouts are pictured in the figures below.
- A 1-bit sign, \(s\).
- A biased exponent, \(e = E + bias\)
- A binary fraction, \(f = .b_1 b_2 \ldots b_{p-1}\) (the \(b_0\) bit is not recorded)

Figure B-1 Single-Precision Floating-Point Format (S)
Values are encoded in the formats using the unbiased exponent, fraction, and sign values shown in Table B-2. The high-order bit of the fraction field, identified as $b_1$, is also important for NaNs.

### Table B-2 Value of Single or Double Floating-Point Format Encoding

<table>
<thead>
<tr>
<th>unbiased $E$</th>
<th>$f$</th>
<th>$s$</th>
<th>$b_1$</th>
<th>value $v$</th>
<th>type of value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E_{max} + 1$</td>
<td>≠ 0</td>
<td>1</td>
<td>SNaN</td>
<td>Signaling NaN</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td></td>
<td>QNaN</td>
<td>Quiet NaN</td>
<td></td>
</tr>
<tr>
<td>$E_{max} + 1$</td>
<td>0</td>
<td>1</td>
<td>$-\infty$</td>
<td>minus infinity</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td></td>
<td>$+\infty$</td>
<td>plus infinity</td>
<td></td>
</tr>
<tr>
<td>$E_{max}$ to $E_{min}$</td>
<td>1</td>
<td></td>
<td>$- (2^E)(1.f)$</td>
<td>negative normalized number</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td></td>
<td>$+ (2^E)(1.f)$</td>
<td>positive normalized number</td>
<td></td>
</tr>
<tr>
<td>$E_{min} - 1$</td>
<td>≠ 0</td>
<td>1</td>
<td>$- (2^{E_{min}})(0.f)$</td>
<td>negative denormalized number</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td></td>
<td>$+ (2^{E_{min}})(0.f)$</td>
<td>positive denormalized number</td>
<td></td>
</tr>
<tr>
<td>$E_{min} - 1$</td>
<td>0</td>
<td>1</td>
<td>- 0</td>
<td>negative zero</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>+ 0</td>
<td>positive zero</td>
<td></td>
</tr>
</tbody>
</table>

#### B 2.1.1 Normalized and Denormalized Numbers

For single and double formats, each representable nonzero numerical value has just one encoding; numbers are kept in normalized form. The high-order bit of the $p$-bit mantissa, which lies to the left of the binary point, is “hidden”, and not recorded in the fraction field. The encoding rules permit the value of this bit to be determined by looking at the value of the exponent. When the unbiased exponent is in the range $E_{min}$ to $E_{max}$, inclusive, the number is normalized and the hidden bit must be 1. If the numeric value cannot be normalized because the exponent would be less than $E_{min}$, then the representation is denormalized and the encoded number has an exponent of $E_{min}-1$ and the hidden bit has the value 0. Plus and minus zero are special cases that are not regarded as denormalized values.

#### B 2.1.2 Reserved Operand Values — Infinity and NaN

A floating-point operation can signal IEEE exception conditions, such as those caused by uninitialized variables, violations of mathematical rules, or results that cannot be represented. If a program does not choose to trap IEEE exception
conditions, a computation that encounters these conditions proceeds without trapping but generates a result indicating that an exceptional condition arose during the computation. To permit this, each floating-point format defines representations, shown in Table B-2, for +infinity (+∞), -infinity (−∞), quiet NaN (QNaN), and signaling NaN (SNaN).

Infinity represents a number with magnitude too large to be represented in the format; in essence it exists to represent a magnitude overflow during a computation. A correctly signed ∞ is generated as the default result in division by zero and some cases of overflow; details are in the IEEE exception condition descriptions and Table B-4 "Default Result for IEEE Exceptions Not Trapped Precisely”.

Once created as a default result, ∞ can become an operand in a subsequent operation. The infinities are interpreted such that −∞ < (every finite number) < +∞. Arithmetic with ∞ is the limiting case of real arithmetic with operands of arbitrarily large magnitude, when such limits exist. In these cases, arithmetic on ∞ is regarded as exact and exception conditions do not arise. The out-of-range indication represented by the ∞ is propagated through subsequent computations. For some cases there is no meaningful limiting case in real arithmetic for operands of ∞ and these cases raise the Invalid Operation exception condition. See the description of the Invalid Operation exception for a list of these cases.

SNaN operands cause the Invalid Operation exception for arithmetic operations. SNaNs are useful values to put uninitialized variables. SNaN is never produced as a result value.

NOTE: The IEEE 754 Standard states that “Whether copying a signaling NaN without a change of format signals the invalid operation exception is the implementor’s option”. The MIPS architecture has chosen to make the formatted operand move instructions (MOV.fmt MOVT.fmt MOVF.fmt MOVN.fmt MOVZ.fmt) non-arithmetic and they do not signal IEEE exceptions.

QNaNs are intended to afford retrospective diagnostic information inherited from invalid or unavailable data and results. Propagation of the diagnostic information requires that information contained in the QNaNs be preserved through arithmetic operations and floating-point format conversions.

QNaN operands do not cause arithmetic operations to signal an exception. When a floating-point result is to be delivered, a QNaN operand causes an arithmetic operation to supply a QNaN result. The result QNaN is one of the operand QNaN values when possible. QNaNs do have effects similar to SNaNs on operations that do not deliver a floating-point result, specifically comparisons. See the detailed description of the floating-point compare instruction (C.cond.fmt) for information.

When certain invalid operations not involving QNaN operands are performed but do not cause a trap (because the trap is not enabled), a new QNaN value is created. Table B-3 shows the QNaN value generated when no input operand QNaN value can be copied. The values listed for the fixed-point formats are the values supplied to satisfy the IEEE standard when a QNaN or infinite floating-point value is converted to fixed point. There is no other feature of the architecture that detects or makes use of these “integer QNaN” values.
### B 2.2 Fixed-point formats

There are two floating-point data types provided by the FPU.
- 32-bit Word fixed-point (type W)
- 64-bit Longword fixed-point (type L) (defined in MIPS III)

The fixed-point values are held in the two’s complement format used for signed integers in the CPU. Unsigned fixed-point data types are not provided in the architecture; application software may synthesize computations for unsigned integers from the existing instructions and data types.

<table>
<thead>
<tr>
<th>Format</th>
<th>New QNaN value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single floating point</td>
<td>7ebE eeee</td>
</tr>
<tr>
<td>Double floating point</td>
<td>7ef7 eeee eeee eeee</td>
</tr>
<tr>
<td>Word fixed point</td>
<td>7eee eeee</td>
</tr>
<tr>
<td>Longword fixed point</td>
<td>7eee eeee eeee eeee</td>
</tr>
</tbody>
</table>

#### Figure B-3  Word Fixed-Point Format (W)

#### Figure B-4  Longword Fixed-Point Format (L)

### B 3 Floating-Point Registers

This section describes the organization and use of the two separate coprocessor 1 (CP1) register sets. The coprocessor general registers, also called Floating General Registers (FGRs) are used to transfer binary data between the FPU and the rest of the system. The general register set is also used to hold formatted FPU operand values. There are only two control registers and they are used to identify and control the FPU.
There are separate 32-bit and 64-bit wide register models. MIPS I defines the 32-bit wide register model. MIPS III defines the 64-bit model. To support programs for earlier architecture definitions, processors providing the 64-bit MIPS III register model also provide the 32-bit wide register model as a mode selection. Selecting 32 or 64-bit register model is an implementation-specific privileged operation.

B 3.1 Organization

The CP1 register organization for 32-bit and 64-bit register models is shown in Figure B-5. The coprocessor general registers are the same width as the CPU registers. The two defined control registers are 32-bits wide.

B 3.2 Binary Data Transfers

The data transfer instructions move words and doublewords between the CP1 general registers and the remainder of the system. The operation of the load and move-to instructions is shown in Figure B-6 and Figure B-7. The store and move-from instructions operate in reverse, reading data from the location that the corresponding load or move-to instruction wrote it.
Figure B-6   Effect of FPU Word Load or Move-to Operations

Doubleword transfers to/from 32-bit registers access an aligned pair of CP1 general registers with the least-significant word of the doubleword in the lowest-numbered register.

Figure B-7   Effect of FPU Doubleword Load or Move-to Operations

NOTE: No 64-bit transfers are defined for the MIPS I 32-bit register model. MIPS II defines the 64-bit loads/stores but not 64-bit moves.
B 3.3 Formatted Operand Layout

FPU instructions that operate on formatted operand values specify the floating-point register (FPR) that holds a value. An FPR is not necessarily the same as a CP1 general register because an FPR is 64 bits wide; if this is wider than the CP1 general registers, an aligned set of adjacent CP1 general registers is used as the FPR. The 32-bit register model provides 16 FPRs specified by the even CP1 general register numbers. The 64-bit register model provides 32 FPRs, one per CP1 general register. Operands that are only 32 bits wide (W and S formats), use only half the space in an FPR. The FPR organization and the way that operand data is stored in them is shown in the following figures. A summary of the data transfer instructions can be found in section B 6.1 on page B-19.

Figure B-8 Floating-point Operand Register (FPR) Organization

Figure B-9 Single Floating Point (S) or Word Fixed (W) Operand in an FPR
B 3.4 Implementation and Revision Register

Coprocessor control register 0 contains values that identify the implementation and revision of the FPU. Only the low-order two bytes of this register are defined as shown in Figure B-11.

NOTE: MIPS I supports the Double floating-point (D) type; the fixed-point longword (L) operand is available starting in MIPS III

B 3.5 FPU Control and Status Register — FCSR

Coprocessor control register 31 is the FPU Control and Status Register (FCSR). Access to the register is not privileged; it can be read or written by any program that can execute floating-point instructions. It controls some operations of the coprocessor and shows status information:

- Selects the default rounding mode for FPU arithmetic operations.
- Selectively enables traps of FPU exception conditions.
- Controls some denormalized number handling options.
- Reports IEEE exceptions that arose in the most recently executed instruction.
- Reports IEEE exceptions that arose, cumulatively, in completed instructions.
- Indicates the condition code result of FP compare instructions.
The contents of this register are unpredictable and undefined after a processor reset or a power-up event. Software should initialize this register.

**Figure B-12**  MIPS I - FPU Control and Status Register (FCSR)

<table>
<thead>
<tr>
<th>31</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>18</th>
<th>17</th>
<th>12</th>
<th>11</th>
<th>7</th>
<th>6</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>c</td>
<td>0</td>
<td>cause</td>
<td>enables</td>
<td>flags</td>
<td>RM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>5</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>V</td>
<td>Z</td>
<td>O</td>
<td>U</td>
<td>I</td>
<td>V</td>
<td>Z</td>
<td>O</td>
<td>U</td>
<td>I</td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>15</td>
<td>14</td>
<td>13</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
</tr>
</tbody>
</table>

**Figure B-13**  MIPS III - FPU Control and Status Register (FCSR)

<table>
<thead>
<tr>
<th>31</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>18</th>
<th>17</th>
<th>12</th>
<th>11</th>
<th>7</th>
<th>6</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>FS</td>
<td>c</td>
<td>0</td>
<td>cause</td>
<td>enables</td>
<td>flags</td>
<td>RM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>V</td>
<td>Z</td>
<td>O</td>
<td>U</td>
<td>I</td>
<td>V</td>
<td>Z</td>
<td>O</td>
<td>U</td>
<td>I</td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>16</td>
<td>15</td>
<td>14</td>
<td>13</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
</tr>
</tbody>
</table>

**Figure B-14**  MIPS IV - FPU Control and Status Register (FCSR)

<table>
<thead>
<tr>
<th>31</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>18</th>
<th>17</th>
<th>12</th>
<th>11</th>
<th>7</th>
<th>6</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>FCC</td>
<td>FS</td>
<td>FCC</td>
<td>0</td>
<td>cause</td>
<td>enables</td>
<td>flags</td>
<td>RM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>E</td>
<td>V</td>
<td>Z</td>
<td>O</td>
<td>U</td>
</tr>
<tr>
<td>17</td>
<td>16</td>
<td>15</td>
<td>14</td>
<td>13</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
</tr>
</tbody>
</table>

All fields in the FCSR are readable and writable.

**FCC**  Floating-Point Condition Codes. These bits record the result of FP compares and are tested for FP conditional branches; the FCC bit to use is specified in the compare or branch instruction. The 0\textsuperscript{th} FCC bit is the same as the c bit in MIPS I.

**FS**  Flush to Zero. When FS is set, denormalized results are flushed to zero instead of causing an unimplemented operation exception. When a denormalized operand value is encountered, zero may be used instead of the denorm; this is implementation specific.

**c**  Condition Bit. This bit records the result of FP compares and is tested by FP conditional branches. In MIPS IV this becomes the 0\textsuperscript{th} FCC bit.
cause Cause bits.
These bits indicate the exception conditions that arise during the execution of an FPU arithmetic instruction in precise exception mode. A bit is set to 1 if the corresponding exception condition arises during the execution of an instruction and 0 otherwise. By reading the registers, the exception conditions caused by the preceding FPU arithmetic instruction can be determined. The meaning of the individual bits is:

- E Unimplemented Operation
- V Invalid Operation
- Z Divide by Zero
- O Overflow
- U Underflow
- I Inexact Result

enables Enable bits (see cause field for bit names).
These bits control, for each of the five conditions individually, whether a trap is taken when the IEEE exception condition occurs. The trap occurs when both an enable bit and the corresponding cause bit are set during an FPU arithmetic operation or by moving a value to the FCSR. The meaning of the individual bits is the same as the cause bits. Note that the “E” cause bit has no corresponding enable bit; the non-IEEE Unimplemented Operation exception defined by MIPS is always enabled.

flags Flag bits. (see cause field for bit names)
This field shows the exception conditions that have occurred for completed instructions since it was last reset. For a completed FPU arithmetic operation that raises an exception condition the corresponding bits in the flag field are set and the others are unchanged. This field is never reset by hardware and must be explicitly reset by user software.

RM Rounding Mode. The rounding mode used for most floating-point operations (some FP instructions use a specific rounding mode). The rounding modes are:

- 0 RN -- Round to Nearest
  Round result to the nearest representable value. When two representable values are equally near, round to the value that has a least significant bit of zero (i.e. is even).
- 1 RZ -- Round toward Zero
  Round result to the value closest to and not greater in magnitude then the result.
- 2 RP -- Round toward Plus infinity
  Round result to the value closest to and not less than the result.
- 3 RM -- Round toward Minus infinity
  Round result to the value closest to and not greater than the result.
B 4 Values in FP Registers

Unlike the CPU, the FPU does not interpret the binary encoding of source operands or produce a binary encoding of results for every operation. The value held in a floating-point operand register (FPR) has a format, or type and it may only be used by instructions that operate on that format. The format of a value is either uninterpreted, unknown, or one of the valid numeric formats: single and double floating-point and word and long fixed-point. The way that the formatted value in an FPR is set and changed is summarized in the state diagram in Figure B-15 and is discussed below.

The value in an FPR is always set when a value is written to the register. When a data transfer instruction writes binary data into an FPR (a load), the FPR gets a binary value that is uninterpreted. A computational or FP register move instruction that produces a result of type fmt puts a value of type fmt into the result register.

When an FPR with an uninterpreted value is used as a source operand by an instruction that requires a value of format fmt, the binary contents are interpreted as an encoded value in format fmt and the value in the FPR changes to a value of format fmt. The binary contents cannot be reinterpreted in a different format.

If an FPR contains a value of format fmt, a computational instruction must not use the FPR as a source operand of a different format. If this occurs, the value in the register becomes unknown and the result of the instruction is also a value that is unknown. Using an FPR containing an unknown value as a source operand produces a result that has an unknown value.

The format of the value in the FPR is unchanged when it is read by a data transfer instruction (a store). A data transfer instruction produces a binary encoding of the value contained in the FPR. If the value in the FPR is unknown, the encoded binary value produced by the operation is not defined.
B 5 FPU Exceptions

The IEEE 754 standard specifies that:

---

A, B: Example formats
Load: Destination of LWC1, LDC1, MTC1, or DMTC1 instructions.
Store: Source operand of SWC1, SDC1, MFC1, or DMFC1 instructions.
Src fmt: Source operand of computational instruction expecting format “fmt”.
Rslt fmt: Result of computational instruction producing value of format “fmt”.

Figure B-15  The Effect of FPU Operations on the Format of Values Held in FPRs.
There are five types of exceptions that shall be signaled when detected. The signal entails setting a status flag, taking a trap, or possibly doing both. With each exception should be associated a trap under user control, ...

This function is implemented in the MIPS FPU architecture with the cause, enable, and flag fields of the control and status register. The flag bits implement IEEE exception status flags, and the cause and enable bits control exception trapping. Each field has a bit for each of the five IEEE exception conditions and the cause field has an additional exception bit, Unimplemented Operation, used to trap for software emulation assistance.

There may be two exception modes for the FPU, precise and imprecise, and the operation of the FPU when exception conditions arise depends on the exception mode that is currently selected. Every processor is able to operate the FPU in the precise exception mode. Some processors also have an imprecise exception mode in which floating-point performance is greater. Selecting the exception mode, when there is a choice, is a privileged implementation-specific operation.

B 5.1 Precise Exception Mode

In precise exception mode, an exception (trap) caused by a floating-point operation is precise. A precise trap occurs before the instruction that causes the trap, or any following instruction, completes and writes results. If desired, the software trap handler can resume execution of the interrupted instruction stream after handling the exception.

The cause bit field reports per-instruction exception conditions. The cause bits are written during each floating-point arithmetic operation to show the exception conditions that arose during the operation. The bits are set to 1 if the corresponding exception condition arises and 0 otherwise.

A floating-point trap is generated any time both a cause bit and the corresponding enable bit are set. This occurs either during the execution of a floating-point operation or by moving a value into the FCSR. There is no enable for Unimplemented Operation; this exception condition always generates a trap.

In a trap handler, the exception conditions that arose during the floating-point operation that trapped are reported in the cause field. Before returning from a floating-point interrupt or exception, or setting cause bits with a move to the FCSR, software must first clear the enabled cause bits by a move to the FCSR to prevent the trap from being retaken. User-mode programs can never observe enabled cause bits set. If this information is required in a user-mode handler, then it must be passed somewhere other than the status register.

For a floating-point operation that sets only non-enabled cause bits, no trap occurs and the default result defined by the IEEE standard is stored (see Table B-4). When a floating-point operation does not trap, the program can see the exception conditions that arose during the operation by reading the cause field.
The flag bit field is a cumulative report of IEEE exception conditions that arise during instructions that complete; instructions that trap do not update the flag bits. The flag bits are set to 1 if the corresponding IEEE exception is raised and unchanged otherwise. There is no flag bit for the MIPS Unimplemented Operation exception condition. The flag bits are never cleared as a side effect of floating-point operations, but may be set or cleared by moving a new value into the FCSR.

B 5.2 Imprecise Exception Mode

In imprecise exception mode, an exception (trap) caused by an IEEE floating-point operation is imprecise (Unimplemented Operation exceptions must still be signaled precisely). An imprecise trap occurs at some point after the exception condition arises. In particular, it does not necessarily occur before the instruction that causes the exception, or following instructions, have completed and written results. The software trap handler can generally neither determine which instruction caused the trap nor continue execution of the interrupted instruction stream; it can record the trap that occurred and abort the program.

The meaning of the cause bit field when reading the FCSR is not defined. When a cause bit is written in the FCSR by moving data to it, the corresponding flag bit is also set.

All floating-point operations, whether they cause a trap or not, complete in the sense that they write a result and record exception condition bits in the flag field. When an IEEE exception condition arises during an operation, the default result defined by the IEEE standard is stored (see Table B-4).

A floating-point trap is generated when an exception condition arises during a floating-point operation and the corresponding enable bit is set. A trap will also be generated when a value with corresponding cause and enable bits set is moved into the FCSR. There is no enable for Unimplemented Operation; this exception condition always generates a trap.

The flag bit field is a cumulative report of IEEE exception conditions that arise during instructions that complete. Because all instructions complete in this mode, unlike precise exception mode, the flag bits include exception conditions that cause traps. The flag bits are set to 1 if the corresponding IEEE exception is raised and unchanged otherwise. There is no flag bit for the MIPS Unimplemented Operation exception condition. The flag bits are never cleared as a side effect of floating-point operations, but may be set or cleared by moving a new value into the FCSR.

B 5.3 Exception Condition Definitions

The five exception conditions defined by the IEEE standard are described in this section. It also describes the MIPS-defined exception condition, Unimplemented Operation, that is used to signal a need for software emulation assistance for an instruction.
Normally an IEEE arithmetic operation can cause only one exception condition; the only case in which two exceptions can occur at the same time are inexact with overflow and inexact with underflow.

At the program’s direction, an IEEE exception condition can either cause a trap or not. The IEEE standard specifies the result to be delivered in case the exception is not enabled and no trap is taken. The MIPS architecture supplies these results whenever the exception condition does not result in a precise trap (i.e. no trap or an imprecise trap). The default action taken depends on the type of exception condition, and in the case of the Overflow, the current rounding mode. The default result is mentioned in each description and summarized in Table B-4.

Table B-4  Default Result for IEEE Exceptions Not Trapped Precisely

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
<th>Default Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>V</td>
<td>Invalid Operation</td>
<td>Supply a quiet NaN.</td>
</tr>
<tr>
<td>Z</td>
<td>Divide by zero</td>
<td>Supply a properly signed infinity.</td>
</tr>
<tr>
<td>U</td>
<td>Underflow</td>
<td>Supply a rounded result.</td>
</tr>
<tr>
<td>I</td>
<td>Inexact</td>
<td>Supply a rounded result. If caused by an overflow without the overflow trap enabled, supply the overflowed result.</td>
</tr>
<tr>
<td>O</td>
<td>Overflow</td>
<td>Depends on the rounding mode as shown below</td>
</tr>
<tr>
<td></td>
<td>0 (RN)</td>
<td>Supply an infinity with the sign of the intermediate result.</td>
</tr>
<tr>
<td></td>
<td>1 (RZ)</td>
<td>Supply the format’s largest finite number with the sign of the intermediate result.</td>
</tr>
<tr>
<td></td>
<td>2 (RP)</td>
<td>For positive overflow values, supply positive infinity. For negative overflow values, supply the format’s most negative finite number.</td>
</tr>
<tr>
<td></td>
<td>3 (RM)</td>
<td>for positive overflow values supply the format’s largest finite number. For negative overflow values, supply minus infinity.</td>
</tr>
</tbody>
</table>

B 5.3.1 Invalid Operation exception

The invalid operation exception is signaled if one or both of the operands are invalid for the operation to be performed. The result, when the exception condition occurs without a precise trap, is a quiet NaN. The invalid operations are:

- One or both operands is a signaling NaN (except for the non-arithmetic MOV.fmt MOV.fmt MOV.fmt MOVN.fmt MOVZ.fmt instructions)
- Addition or subtraction: magnitude subtraction of infinities, such as: $(+\infty) + (-\infty)$ or $(-\infty) - (-\infty)$
- Multiplication: $0 \times \infty$, with any signs
- Division: $0 / 0$ or $\infty / \infty$, with any signs
- Square root: An operand less than 0 (-0 is a valid operand value).
- Conversion of a floating-point number to a fixed-point format when an overflow, or operand value of infinity or NaN, precludes a faithful representation in that format.
• Some comparison operations in which one or both of the operands is a QNaN value. The definition of the compare operation (C.cond.fmt) has tables showing the comparisons that do and do not signal the exception.

B 5.3.2 Division By Zero exception

The division by zero exception is signaled on an implemented divide operation if the divisor is zero and the dividend is a finite nonzero number. The result, when no precise trap occurs, is a correctly signed infinity. The divisions (0/0) and (∞/0) do not cause the division by zero exception. The result of (0/0) is an Invalid Operation exception condition. The result of (∞/0) is a correctly signed infinity.

B 5.3.3 Overflow exception

The overflow exception is signaled when what would have been the magnitude of the rounded floating-point result, were the exponent range unbounded, is larger than the destination format’s largest finite number. The result, when no precise trap occurs, is determined by the rounding mode and the sign of the intermediate result as shown in Table B-4.

B 5.3.4 Underflow exception

Two related events contribute to underflow. One is the creation of a tiny non-zero result between ±2^{E_{min}} which, because it is tiny, may cause some other exception later such as overflow on division. The other is extraordinary loss of accuracy during the approximation of such tiny numbers by denormalized numbers. The IEEE standard permits a choice in how these events are detected, but requires that they must be detected the same way for all operations.

The IEEE standard specifies that “tininess” may be detected either “after rounding” (when a nonzero result computed as though the exponent range were unbounded would lie strictly between ±2^{E_{min}}), or “before rounding” (when a nonzero result computed as though both the exponent range and precision were unbounded would lie strictly between ±2^{E_{min}}). The MIPS architecture specifies that tininess is detected after rounding.

The IEEE standard specifies that loss of accuracy may be detected as either “denormalization loss” (when the delivered result differs from what would have been computed if the exponent range were unbounded), or “inexact result” (when the delivered result differs from what would have been computed if both the exponent range and precision were unbounded). The MIPS architecture specifies that loss of accuracy is detected as inexact result.

When an underflow trap is not enabled, underflow is signaled only when both tininess and loss of accuracy have been detected. The delivered result might be zero, denormalized, or ±2^{E_{min}}. When an underflow trap is enabled (via the FCSR enable field bit), underflow is signaled when tininess is detected regardless of loss of accuracy.
B 5.3.5 Inexact exception

If the rounded result of an operation is not exact or if it overflows without an overflow trap, then the inexact exception is signaled.

B 5.3.6 Unimplemented Operation exception

This MIPS defined (non-IEEE) exception is to provide software emulation support. The architecture is designed to permit a combination of hardware and software to fully implement the architecture. Operations that are not fully supported in hardware cause an Unimplemented Operation exception so that software may perform the operation. There is no enable bit for this condition; it always causes a trap. After the appropriate emulation or other operation is done in a software exception handler, the original instruction stream can be continued.

B 6 Functional Instruction Groups

The FPU instructions are divided into the following functional groups:

- Data Transfer
- Arithmetic
- Conversion
- Formatted Operand Value Move
- Conditional Branch
- Miscellaneous

B 6.1 Data Transfer Instructions

The FPU has two separate register sets: coprocessor general registers and coprocessor control registers. The FPU has a load/store architecture; all computations are done on data held in coprocessor general registers. The control registers are used to control FPU operation. Data is transferred between registers and the rest of the system with dedicated load, store, and move instructions. The transferred data is treated as unformatted binary data; no format conversions are performed and, therefore, no IEEE floating-point exceptions can occur.

The supported transfer operations are:

- FPU general reg ↔ memory (word/doubleword load/store)
- FPU general reg ↔ CPU general reg (word/doubleword move)
- FPU control reg ↔ CPU general reg (word move)

All coprocessor loads and stores operate on naturally-aligned data items. An attempt to load or store to an address that is not naturally aligned for the data item will cause an Address Error exception. Regardless of byte-numbering order
endianness), the address of a word or doubleword is the smallest byte address among the bytes in the object. For a big-endian machine this is the most-significant byte; for a little-endian machine this is the least-significant byte.

The FPU has loads and stores using the usual register + offset addressing. For the FPU only, there are load and store instructions using register + register addressing.

MIPS I specifies that loads are delayed by one instruction and that proper execution must be insured by observing an instruction scheduling restriction. The instruction immediately following a load into an FPU register \( F_n \) must not use \( F_n \) as a source register. The time between the load instruction and the time the data is available is the “load delay slot”. If no useful instruction can be put into the load delay slot, then a null operation (NOP) must be inserted.

In MIPS II, this instruction scheduling restriction is removed. Programs will execute correctly when the loaded data is used by the instruction following the load, but this may require extra real cycles. Most processors cannot actually load data quickly enough for immediate use and the processor will be forced to wait until the data is available. Scheduling load delay slots is desirable for performance reasons even when it is not necessary for correctness.

Table B-5  FPU Loads and Stores Using Register + Offset Address Mode

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
<th>Defined in</th>
</tr>
</thead>
<tbody>
<tr>
<td>LWC1</td>
<td>Load Word to Floating-Point</td>
<td>MIPS I</td>
</tr>
<tr>
<td>SWC1</td>
<td>Store Word to Floating-Point</td>
<td>I</td>
</tr>
<tr>
<td>LDC1</td>
<td>Load Doubleword to Floating-Point</td>
<td>III</td>
</tr>
<tr>
<td>SDC1</td>
<td>Store Doubleword to Floating-Point</td>
<td>III</td>
</tr>
</tbody>
</table>

Table B-6  FPU Loads and Using Register + Register Address Mode

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
<th>Defined in</th>
</tr>
</thead>
<tbody>
<tr>
<td>LWXC1</td>
<td>Load Word Indexed to Floating-Point</td>
<td>MIPS IV</td>
</tr>
<tr>
<td>SWXC1</td>
<td>Store Word Indexed to Floating-Point</td>
<td>IV</td>
</tr>
<tr>
<td>LDXC1</td>
<td>Load Doubleword Indexed to Floating-Point</td>
<td>IV</td>
</tr>
<tr>
<td>SDXC1</td>
<td>Store Doubleword Indexed to Floating-Point</td>
<td>IV</td>
</tr>
</tbody>
</table>

Table B-7  FPU Move To/From Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
<th>Defined in</th>
</tr>
</thead>
<tbody>
<tr>
<td>MTC1</td>
<td>Move Word To Floating-Point</td>
<td>MIPS I</td>
</tr>
<tr>
<td>MFC1</td>
<td>Move Word From Floating-Point</td>
<td>I</td>
</tr>
<tr>
<td>DMTC1</td>
<td>Doubleword Move To Floating-Point</td>
<td>III</td>
</tr>
<tr>
<td>DMFC1</td>
<td>Doubleword Move From Floating-Point</td>
<td>III</td>
</tr>
<tr>
<td>CTC1</td>
<td>Move Control Word To Floating-Point</td>
<td>I</td>
</tr>
<tr>
<td>CFC1</td>
<td>Move Control Word From Floating-Point</td>
<td>I</td>
</tr>
</tbody>
</table>
## B 6.2 Arithmetic Instructions

The arithmetic instructions operate on formatted data values. The result of most floating-point arithmetic operations meets the IEEE standard specification for accuracy; a result which is identical to an infinite-precision result rounded to the specified format, using the current rounding mode. The rounded result differs from the exact result by less than one unit in the least-significant place (ulp).

### Table B-8 FPU IEEE Arithmetic Operations

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
<th>Defined in</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD.fmt</td>
<td>Floating-Point Add</td>
<td>MIPS I</td>
</tr>
<tr>
<td>SUB.fmt</td>
<td>Floating-Point Subtract</td>
<td>I</td>
</tr>
<tr>
<td>MUL.fmt</td>
<td>Floating-Point Multiply</td>
<td>I</td>
</tr>
<tr>
<td>DIV.fmt</td>
<td>Floating-Point Divide</td>
<td>I</td>
</tr>
<tr>
<td>ABS.fmt</td>
<td>Floating-Point Absolute Value</td>
<td>I</td>
</tr>
<tr>
<td>NEG.fmt</td>
<td>Floating-Point Negate</td>
<td>I</td>
</tr>
<tr>
<td>SQRT.fmt</td>
<td>Floating-Point Square Root</td>
<td>II</td>
</tr>
<tr>
<td>C.cond.fmt</td>
<td>Floating-Point Compare</td>
<td>I</td>
</tr>
</tbody>
</table>

Two operations, Reciprocal Approximation (RECIP) and Reciprocal Square Root Approximation (RSQRT), may be less accurate than the IEEE specification. The result of RECIP differs from the exact reciprocal by no more than one ulp. The result of RSQRT differs by no more than two ulp. Within these error limits, the result of these instructions is implementation specific.

### Table B-9 FPU Approximate Arithmetic Operations

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
<th>Defined in</th>
</tr>
</thead>
<tbody>
<tr>
<td>RECIP.fmt</td>
<td>Floating-Point Reciprocal Approximation</td>
<td>MIPS IV</td>
</tr>
<tr>
<td>RSQRT.fmt</td>
<td>Floating-Point Reciprocal Square Root Approximation</td>
<td>IV</td>
</tr>
</tbody>
</table>

There are four compound-operation instructions that perform variations of multiply-accumulate: multiply two operands and accumulate to a third operand to produce a result. The accuracy of the result depends which of two alternative arithmetic models is used for the computation. The unrounded model is more accurate than a pair of IEEE operations and the rounded model meets the IEEE specification.

### Table B-10 FPU Multiply-Accumulate Arithmetic Operations

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
<th>Defined in</th>
</tr>
</thead>
<tbody>
<tr>
<td>MADD.fmt</td>
<td>Floating-Point Multiply Add</td>
<td>MIPS IV</td>
</tr>
<tr>
<td>MSUB.fmt</td>
<td>Floating-Point Multiply Subtract</td>
<td>IV</td>
</tr>
<tr>
<td>NMADD.fmt</td>
<td>Floating-Point Negative Multiply Add</td>
<td>IV</td>
</tr>
<tr>
<td>NMSUB.fmt</td>
<td>Floating-Point Negative Multiply Subtract</td>
<td>IV</td>
</tr>
</tbody>
</table>
The initial implementation of the MIPS IV architecture, the R8000 (and future revisions of it), uses the unrounded arithmetic model which does not match the IEEE accuracy specification. All other implementations will use the rounded model which does meet the specification.

- **Rounded** or non-fused: The product is rounded according to the current rounding mode prior to the accumulation. This model meets the IEEE accuracy specification; the result is numerically identical to the equivalent computation using multiply, add, subtract, and negate instructions.

- **Unrounded** or fused (R8000 implementation): The product is not rounded and all bits take part in the accumulation. This model does not match the IEEE accuracy requirements; the result is more accurate than the equivalent computation using IEEE multiply, add, subtract, and negate instructions.

### B 6.3 Conversion Instructions

There are instructions to perform conversions among the floating-point and fixed-point data types. Each instruction converts values from a number of operand formats to a particular result format. Some convert instructions use the rounding mode specified in the Floating Control and Status Register (FCSR), others specify the rounding mode directly.

#### Table B-11 FPU Conversion Operations Using the FCSR Rounding Mode

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
<th>Defined in</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVT.S.fmt</td>
<td>Floating-Point Convert to Single Floating-Point</td>
<td>MIPS I</td>
</tr>
<tr>
<td>CVT.D.fmt</td>
<td>Floating-Point Convert to Double Floating-Point</td>
<td>I</td>
</tr>
<tr>
<td>CVT.W.fmt</td>
<td>Floating-Point Convert to Word Fixed-Point</td>
<td>I</td>
</tr>
<tr>
<td>CVT.L.fmt</td>
<td>Floating-Point Convert to Long Fixed-Point</td>
<td>I</td>
</tr>
</tbody>
</table>

#### Table B-12 FPU Conversion Operations Using a Directed Rounding Mode

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
<th>Defined in</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROUND.W.fmt</td>
<td>Floating-Point Round to Word Fixed-Point</td>
<td>II</td>
</tr>
<tr>
<td>ROUND.L.fmt</td>
<td>Floating-Point Round to Long Fixed-Point</td>
<td>III</td>
</tr>
<tr>
<td>TRUNC.W.fmt</td>
<td>Floating-Point Truncate to Word Fixed-Point</td>
<td>II</td>
</tr>
<tr>
<td>TRUNC.L.fmt</td>
<td>Floating-Point Truncate to Long Fixed-Point</td>
<td>III</td>
</tr>
<tr>
<td>CEIL.W.fmt</td>
<td>Floating-Point Ceiling to Word Fixed-Point</td>
<td>II</td>
</tr>
<tr>
<td>CEIL.L.fmt</td>
<td>Floating-Point Ceiling to Long Fixed-Point</td>
<td>III</td>
</tr>
<tr>
<td>FLOOR.W.fmt</td>
<td>Floating-Point Floor to Word Fixed-Point</td>
<td>II</td>
</tr>
<tr>
<td>FLOOR.L.fmt</td>
<td>Floating-Point Floor to Long Fixed-Point</td>
<td>III</td>
</tr>
</tbody>
</table>
B 6.4 Formatted Operand Value Move Instructions

These instructions all move formatted operand values among FPU general registers. A particular operand type must be moved by the instruction that handles that type. There are three kinds of move instructions:

- Unconditional move
- Conditional move that tests an FPU condition code
- Conditional move that tests a CPU general register value against zero

The conditional move instructions operate in a way that may be unexpected. They always force the value in the destination register to become a value of the format specified in the instruction. If the destination register does not contain an operand of the specified format before the conditional move is executed, the contents become undefined. There is more information in Values in FP Registers on page B-13 and in the individual descriptions of the conditional move instructions themselves.

Table B-13  FPU Formatted Operand Move Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
<th>Defined in</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVfmt</td>
<td>Floating-Point Move</td>
<td>MIPS I</td>
</tr>
</tbody>
</table>

Table B-14  FPU Conditional Move on True/False Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
<th>Defined in</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVfmt</td>
<td>Floating-Point Move Conditional on FP True</td>
<td>MIPS IV</td>
</tr>
<tr>
<td>MOVEfmt</td>
<td>Floating-Point Move Conditional on FP False</td>
<td>IV</td>
</tr>
</tbody>
</table>

Table B-15  FPU Conditional Move on Zero/Nonzero Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
<th>Defined in</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVZfmt</td>
<td>Floating-Point Move Conditional on Zero</td>
<td>MIPS IV</td>
</tr>
<tr>
<td>MOVNfmt</td>
<td>Floating-Point Move Conditional on Nonzero</td>
<td>IV</td>
</tr>
</tbody>
</table>

B 6.5 Conditional Branch Instructions

The FPU has PC-relative conditional branch instructions that test condition codes set by FPU compare instructions (C.cond.fmt).

All branches have an architectural delay of one instruction. When a branch is taken, the instruction immediately following the branch instruction, in the branch delay slot, is executed before the branch to the target instruction takes place. Conditional branches come in two versions that treat the instruction in the delay slot differently when the branch is not taken and execution falls through. The “branch” instructions execute the instruction in the delay slot, but the “branch likely” instructions do not (they are said to nullify it).
MIPS I defines a single condition code which is implicit in the compare and branch instructions. MIPS IV defines seven additional condition codes and includes the condition code number in the compare and branch instructions. The MIPS IV extension keeps the original condition bit as condition code zero and the extended encoding is compatible with the MIPS I encoding.

Table B-16  FPU Conditional Branch Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
<th>Defined in</th>
</tr>
</thead>
<tbody>
<tr>
<td>BC1T</td>
<td>Branch on FP True</td>
<td>MIPS I</td>
</tr>
<tr>
<td>BC1F</td>
<td>Branch on FP False</td>
<td>I</td>
</tr>
<tr>
<td>BC1TL</td>
<td>Branch on FP True Likely</td>
<td>II</td>
</tr>
<tr>
<td>BC1FL</td>
<td>Branch on FP False Likely</td>
<td>II</td>
</tr>
</tbody>
</table>

B 6.6  Miscellaneous Instructions

B 6.6.1  CPU Conditional Move

There are instructions to move conditionally move one CPU general register to another based on an FPU condition code.

Table B-17  CPU Conditional Move on FPU True/False Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
<th>Defined in</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVZ</td>
<td>Move Conditional on FP True</td>
<td>MIPS IV</td>
</tr>
<tr>
<td>MOVN</td>
<td>Move Conditional on FP False</td>
<td>IV</td>
</tr>
</tbody>
</table>

B 7  Valid Operands for FP Instructions

The floating-point unit arithmetic, conversion, and operand move instructions operate on formatted values with different precision and range limits and produce formatted values for results. Each representable value in each format has a binary encoding that is read from or stored to memory. The \textit{fmt} or \textit{fmt3} field of the instruction encodes the operand format required for the instruction. A conversion instruction specifies the result type in the \textit{function} field; the result of other operations is the same format as the operands. The encoding of the \textit{fmt} and \textit{fmt3} fields is shown in Table B-18.
Each type of arithmetic or conversion instruction is valid for operands of selected formats. A summary of the computational and operand move instructions and the formats valid for each of them is listed in Table B-19. Implementations must support combinations that are valid either directly in hardware or through emulation in an exception handler.

The result of an instruction using operand formats marked “U” is not currently specified by this architecture and will cause an exception. They are available for future extensions to the architecture. The exact exception mechanism used is processor specific. Most implementations report this as an Unimplemented Operation for a Floating Point exception. Other implementations report these combinations as Reserved Instruction exceptions.

The result of an instruction using operand formats marked “i” are invalid and an attempt to execute such an instruction has an undefined result.

### Table B-18  FPU Operand Format Field (fmt, fmt3) Decoding

<table>
<thead>
<tr>
<th>fmt</th>
<th>fmt3</th>
<th>Instruction Mnemonic</th>
<th>Size name</th>
<th>bits</th>
<th>data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-15</td>
<td>-</td>
<td>Reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>0</td>
<td>S</td>
<td>single</td>
<td>32</td>
<td>floating-point</td>
</tr>
<tr>
<td>17</td>
<td>1</td>
<td>D</td>
<td>double</td>
<td>64</td>
<td>floating-point</td>
</tr>
<tr>
<td>18-19</td>
<td>2-3</td>
<td>Reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>4</td>
<td>W</td>
<td>word</td>
<td>32</td>
<td>fixed-point</td>
</tr>
<tr>
<td>21</td>
<td>5</td>
<td>L</td>
<td>long</td>
<td>64</td>
<td>fixed-point</td>
</tr>
<tr>
<td>22–31</td>
<td>6-7</td>
<td>Reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Table B-19  Valid Formats for FPU Operations

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Operation</th>
<th>operand fmt</th>
<th>COP1 function value</th>
<th>COP1 X op4 value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABS</td>
<td>Absolute value</td>
<td>•</td>
<td>U</td>
<td>U</td>
</tr>
<tr>
<td>ADD</td>
<td>Add</td>
<td>•</td>
<td>U</td>
<td>U</td>
</tr>
<tr>
<td>C.cond</td>
<td>Floating-point compare</td>
<td>•</td>
<td>U</td>
<td>U</td>
</tr>
<tr>
<td>CEIL.L,</td>
<td>Convert to longword fixed-point, round toward +∞</td>
<td>•</td>
<td>i</td>
<td>i</td>
</tr>
<tr>
<td>(CEIL.W)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CVT.D</td>
<td>Convert to double floating-point</td>
<td>i</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CVT.L</td>
<td>Convert to longword fixed-point</td>
<td>•</td>
<td>i</td>
<td>i</td>
</tr>
<tr>
<td>CVT.S</td>
<td>Convert to single floating-point</td>
<td>i</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>CVT.W</td>
<td>Convert to 32-bit fixed-point</td>
<td>•</td>
<td>i</td>
<td>i</td>
</tr>
<tr>
<td>DIV</td>
<td>Divide</td>
<td>•</td>
<td>U</td>
<td>U</td>
</tr>
<tr>
<td>FLOOR.L,</td>
<td>Convert to longword fixed-point, round toward −∞</td>
<td>•</td>
<td>i</td>
<td>i</td>
</tr>
<tr>
<td>(FLOOR.W)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MADD</td>
<td>Multiply-Add</td>
<td>•</td>
<td>U</td>
<td>U</td>
</tr>
<tr>
<td>MOV</td>
<td>Move Register</td>
<td>•</td>
<td>i</td>
<td>i</td>
</tr>
</tbody>
</table>
B 8 Description of an Instruction

For the FPU instruction detail documentation, all variable subfields in an instruction format (such as $fs$, $ft$, immediate, and so on) are shown in lower-case. The instruction name (such as ADD, SUB, and so on) is shown in upper-case.

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

In some instructions, the instruction subfields $op$ and $function$ can have constant 6-bit values. When reference is made to these instructions, upper-case mnemonics are used. For instance, in the floating-point ADD instruction we use $op = COP1$ and $function = ADD$. In other cases, a single field has both fixed and variable subfields, so the name contains both upper and lower case characters. Bit encodings for mnemonics are shown at the end of this section, and are also included with each individual instruction.

B 9 Operation Notation Conventions and Functions

The instruction description includes an Operation section that describes the operation of the instruction in a pseudocode. The pseudocode and terms used in the description are described in Operation Section Notation and Functions on page A-18.
B 10 Individual FPU Instruction Descriptions

The FP instructions are described in alphabetic order. See Description of an Instruction on page A-15 for a description of the information in each instruction description.
**ABS.fmt**

*Floating-Point Absolute Value*

<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>0</td>
<td>0</td>
<td>0</td>
<td>fs</td>
<td>fd</td>
<td>ABS</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Format:**

ABS.S     fd, fs  
ABS.D     fd, fs

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

**Description:**  
fd ← absolute(fs)  

The absolute value of the value in FPR fs is placed in FPR fd. 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; see Floating-Point Registers on page B-6. If they are not valid, the result is undefined.

The operand must be a value in format fmt; see section B 7 on page B-24. If it is not, the result is undefined and the value of the operand FPR becomes undefined.

**Operation:**

StoreFPR(fd, fmt, AbsoluteValue(ValueFPR(fs, fmt)))

**Exceptions:**

Coprocessor Unusable  
Reserved Instruction  
Floating-Point  
Unimplemented Operation  
Invalid Operation
Floating-Point Add ADD.fmt

Format:

\[
\begin{array}{cccccc}
31 & 26 & 25 & 21 & 20 & 16 & 15 & 11 & 10 & 6 & 5 & 0 \\
\hline
COP1 & fmt & ft & fs & fd & \text{ADD} & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{array}
\]

Purpose:
To add FP values.

Description:

\[
\text{fd} \leftarrow \text{fs} + \text{ft}
\]

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

Restrictions:

The fields \text{fs}, \text{ft}, and \text{fd} must specify FPRs valid for operands of type \text{fmt}; see Floating-Point Registers on page B-6. If they are not valid, the result is undefined.

The operands must be values in format \text{fmt}; see section B 7 on page B-24. If they are not, the result is undefined and the value of the operand FPRs becomes undefined.

Operation:

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

Exceptions:

Coprocessor Unusable
Reserved Instruction
Floating-Point
  Unimplemented Operation
  Invalid Operation
  Inexact
  Overflow
  Underflow
Branch on FP False

Format:

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

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 and 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 architecture adds 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.

Restrictions:

MIPS I, II, III: 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.

MIPS IV: None.
Branch on FP False BC1F

Operation:

MIPS I, II, and III define a single condition code; MIPS IV adds 7 more condition
codes. 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.

MIPS I:
1-: condition ← COC[1] = tf
1: target_offset ← (offset$_{15}$)\textsuperscript{GPRLEN(16+2)} || offset || 0$^2$
1+ 1: if condition then
    PC ← PC + target
    endif

MIPS II and MIPS III:
1-: condition ← COC[1] = tf
1: target_offset ← (offset$_{15}$)\textsuperscript{GPRLEN(16+2)} || offset || 0$^2$
1+ 1: if condition then
    PC ← PC + target
    else if nd then
        NullifyCurrentInstruction()
    endif

MIPS IV:
1: condition ← FCC[cc] = tf
    target_offset ← (offset$_{15}$)\textsuperscript{GPRLEN(16+2)} || offset || 0$^2$
1+ 1: if condition then
    PC ← PC + target
    else if nd then
        NullifyCurrentInstruction()
    endif

Exceptions:
Coprocessor Unusable
Reserved Instruction
Floating-Point
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 more distant addresses.
BC1FL

Branch on FP False Likely

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>COP1</td>
<td>BC</td>
<td>cc</td>
<td>nd</td>
<td>tf</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 0 0 0 1</td>
<td>0 1 0 0 0</td>
<td>0 1 0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Format:

BC1FL offset (cc = 0 implied) MIPS II
BC1FL cc, offset MIPS IV

Purpose:
To test an FP condition code and do a PC-relative conditional branch; execute 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 FP condition code bit cc is false (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.

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

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 and 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 architecture adds 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.

Restrictions:

MIPS II, III: 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.

MIPS IV: None.
Branch on FP False Likely **BC1FL**

**Operation:**

MIPS II and III define a single condition code; MIPS IV adds 7 more condition codes. 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`.

MIPS II and MIPS III:

1. condition ← COC[1] = tf
2. target_offset ← (offset15)GPRLEN-(16+2) || offset || 02
3. if condition then
   - PC ← PC + target
4. else if nd then
   - NullifyCurrentInstruction()
5. endif

MIPS IV:

1. condition ← FCC[cc] = tf
2. target_offset ← (offset15)GPRLEN-(16+2) || offset || 02
3. if condition then
   - PC ← PC + target
4. else if nd then
   - NullifyCurrentInstruction()
5. endif

**Exceptions:**

- Coprocessor Unusable
- Reserved Instruction
- Floating-Point 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 more distant addresses.
**BC1T**

**Branch on FP True**

<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>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>COP1</td>
<td>BC</td>
<td>cc</td>
<td>nd</td>
<td>tf</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 0 0 0 1</td>
<td>0 1 0 0 0</td>
<td>0 1 0 0 0 0</td>
<td>0 1 1 1</td>
<td>0 1 1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:**

<table>
<thead>
<tr>
<th>MIPS I</th>
<th>MIPS IV</th>
</tr>
</thead>
<tbody>
<tr>
<td>BC1T offset (cc = 0 implied)</td>
<td>BC1T cc, offset</td>
</tr>
</tbody>
</table>

**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), branch 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, \( \text{C.c} \text{.cond.fmt} \).

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 and 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 architecture adds 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.

**Restrictions:**

**MIPS I, II, III:** 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.

**MIPS IV:** None
Branch on FP True BC1T

Operation:

MIPS I, II, and III define a single condition code; MIPS IV adds 7 more condition codes. 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.

MIPS I:
I - 1: \( \text{condition} \leftarrow \text{COC}[1] = \text{tf} \)
I: \( \text{target} \leftarrow (\text{offset}_{15})^{\text{GPRLEN}-(16+2)} || \text{offset} || 0^2 \)
I + 1: if condition then
      \( \text{PC} \leftarrow \text{PC} + \text{target} \)
      endif

MIPS II and MIPS III:
I - 1: \( \text{condition} \leftarrow \text{COC}[1] = \text{tf} \)
I: \( \text{target} \leftarrow (\text{offset}_{15})^{\text{GPRLEN}-(16+2)} || \text{offset} || 0^2 \)
I + 1: if condition then
      \( \text{PC} \leftarrow \text{PC} + \text{target} \)
      else if nd then
      NullifyCurrentInstruction()
      endif

MIPS IV:
I: \( \text{condition} \leftarrow \text{FCC}[cc] = \text{tf} \)
   \( \text{target} \leftarrow (\text{offset}_{15})^{\text{GPRLEN}-(16+2)} || \text{offset} || 0^2 \)
I + 1: if condition then
      \( \text{PC} \leftarrow \text{PC} + \text{target} \)
      else if nd then
      NullifyCurrentInstruction()
      endif

Exceptions:
- Coprocessor Unusable
- Reserved Instruction
- Floating-Point
- Unimplemented Operation

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 more distant addresses.
BC1TL  Branch on FP True Likely

<table>
<thead>
<tr>
<th>COP1</th>
<th>BC</th>
<th>cc</th>
<th>nd</th>
<th>tf</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0 1</td>
<td>0 1 0 0 0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

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

**Purpose:** To test an FP condition code and do a PC-relative conditional branch; execute 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), 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.

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

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 and 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 architecture adds 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.

**Restrictions:**
- **MIPS II, III:** 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.
- **MIPS IV:** None.
Branch on FP True Likely BC1TL

Operation:

MIPS II, and III define a single condition code; MIPS IV adds 7 more condition codes. 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.

MIPS II and MIPS III:

1 - 1: condition ← COC[1] = tf
   1: target ← (offset15)GPRLEN-(16+2) || offset || 02
   1 + 1 : if condition then
      PC ← PC + target
   else if nd then
      NullifyCurrentInstruction()
   endif

MIPS IV:

1: condition ← FCC[cc] = tf
   target ← (offset15)GPRLEN-(16+2) || offset || 02
   1 + 1 : if condition then
      PC ← PC + target
   else if nd then
      NullifyCurrentInstruction()
   endif

Exceptions:

Coprocessor Unusable
Reserved Instruction
Floating-Point
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 more distant addresses.
C.cond.fmt Floating-Point Compare

<table>
<thead>
<tr>
<th>COP1</th>
<th>fmt</th>
<th>ft</th>
<th>fs</th>
<th>cc</th>
<th>0</th>
<th>FC</th>
<th>cond</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0 1</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>

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  

MIPS I

MIPS IV

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

Description:

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 \( cond_{2..1} \) is true for the operand values, then the result is true, otherwise it is false. If no exception is taken, the result is written into condition code \( cc \); true is 1 and false is 0.

If \( cond_3 \) is set and at least one of the values is a NaN, an Invalid Operation condition is raised; the result depends on the FP exception model currently active.

- Precise exception model: 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 \( cc \).
- Imprecise exception model (R8000 normal mode): The Boolean result is written into condition code \( cc \). No FCSR flag is set. If the Invalid Operation enable bit is set in the FCSR, an Invalid Operation exception is taken, imprecisely, at some future time.

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 sixteen comparison predicates. The Boolean result of the instruction is obtained by substituting the Boolean value of each ordering relation for the two FP values into equation. If the equal relation is true, for example, then all four example predicates above would yield a true result. If the unordered relation is true then only the final predicate, “unordered or equal” would yield a true result.
Logical negation of a compare result allows eight distinct comparisons to test for sixteen predicates as shown in Table B-20. 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 (note that the False predicate is never true and False/True do not follow the normal pattern). When the first predicate is true, the second predicate must be false, and vice versa. 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 with the Branch on FP True (BC1T) instruction and the truth of the second with Branch on FP False (BC1F).

Table B-20  FPU Comparisons Without Special Operand Exceptions

<table>
<thead>
<tr>
<th>Instr</th>
<th>Comparison Predicate</th>
<th>Comparison CC Result</th>
<th>Cond field</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConD Mne- monic</td>
<td>predicate and logically negated predicate (abbreviation)</td>
<td>relation values</td>
<td>if predicate is true</td>
</tr>
<tr>
<td>F</td>
<td>False</td>
<td>T F F F</td>
<td>T F T T</td>
</tr>
<tr>
<td>UN</td>
<td>Unordered</td>
<td>T T T F</td>
<td>T</td>
</tr>
<tr>
<td>EQ</td>
<td>Equal</td>
<td>T F T F</td>
<td>T</td>
</tr>
<tr>
<td>UEQ</td>
<td>Unordered or Equal</td>
<td>T T F T</td>
<td>T</td>
</tr>
<tr>
<td>OLT</td>
<td>Ordered or Less Than</td>
<td>T F T F</td>
<td>T</td>
</tr>
<tr>
<td>ULE</td>
<td>Unordered or Less than or Equal</td>
<td>T F F F</td>
<td>T</td>
</tr>
<tr>
<td>key: “?” = unordered, “&gt;” = greater than, “&lt;” = less than, “=” is equal, “T” = True, “F” = False</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

FPU Instruction Set  MIPS IV Instruction Set. Rev 3.2
C. cond.fmt Floating-Point Compare

There is another set of eight compare operations, distinguished by a cond3 value of 1, testing the same sixteen 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, then an Invalid Operation exception occurs.

Table B-21  FPU Comparisons With Special Operand Exceptions for QNaNs

<table>
<thead>
<tr>
<th>Instr</th>
<th>Comparison Predicate</th>
<th>Comparison CC Result</th>
<th>Inv Op excep if Q NaN</th>
<th>cond field</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>name of predicate and logically negated predicate (abbreviation)</td>
<td>relation values &gt; &lt; = ?</td>
<td>If predicate is true</td>
<td></td>
</tr>
<tr>
<td>SF</td>
<td>Signaling False [this predicate always False] Signaling True (ST)</td>
<td>F F F F T T T T</td>
<td>F</td>
<td>0</td>
</tr>
<tr>
<td>NGLE</td>
<td>Not Greater than or Less than or Equal Greater than or Less than or Equal (GLE)</td>
<td>F F F T T T F</td>
<td>T</td>
<td>1</td>
</tr>
<tr>
<td>SEQ</td>
<td>Signaling Equal Signaling Not Equal (SNE)</td>
<td>F F T F T T F</td>
<td>T</td>
<td>2</td>
</tr>
<tr>
<td>NGL</td>
<td>Not Greater than or Less than Greater than or Less than (GL)</td>
<td>F F T T T T T T F F</td>
<td>T</td>
<td>Yes</td>
</tr>
<tr>
<td>LT</td>
<td>Less than Not Less Than (NLT)</td>
<td>F T F F T T F</td>
<td>T</td>
<td>4</td>
</tr>
<tr>
<td>NGE</td>
<td>Not Greater than or Equal Greater than or Equal (GE)</td>
<td>F T F F T T F</td>
<td>T</td>
<td>5</td>
</tr>
<tr>
<td>LE</td>
<td>Less than or Equal Not Less than or Equal (NLE)</td>
<td>F T F F T T F</td>
<td>T</td>
<td>6</td>
</tr>
<tr>
<td>NGT</td>
<td>Not Greater than Greater than (GT)</td>
<td>F T F F T T F</td>
<td>F</td>
<td>7</td>
</tr>
</tbody>
</table>

key: “?” = unordered, “>” = greater than, “<” = less than, “=” is equal, “T” = True, “F” = False

The instruction encoding is an extension made in the MIPS IV architecture. In previous architecture levels the cc field for this instruction must be 0.

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 and 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.

B-40  MIPS IV Instruction Set. Rev 3.2  FPU Instruction Set
Floating-Point Compare \texttt{C.cond.fmt}

The MIPS IV architecture adds 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.

Restrictions:

The fields $fs$ and $ft$ must specify FPRs valid for operands of type $fmt$; see Floating-Point Registers on page B-6. If they are not valid, the result is undefined.

The operands must be values in format $fmt$; see section B 7 on page B-24. If they are not, the result is undefined and the value of the operand FPRs becomes undefined.

MIPS I, II, III: 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.

Operation:

\begin{verbatim}
if NaN(Value FPR(fs, fmt)) or NaN(ValueFPR(ft, fmt)) then
  less ← false
  equal ← false
  unordered ← true
  if t then
    SignalException(InvalidOperation)
  endif
else
  less ← ValueFPR(fs, fmt) < ValueFPR(ft, fmt)
  equal ← ValueFPR(fs, fmt) = ValueFPR(ft, fmt)
  unordered ← false
endif

condition ← (cond 2 and less) or (cond 1 and equal) or (cond 0 and unordered)

FCC[cc] ← condition
if cc = 0 then
  COC[1] ← condition
endif
\end{verbatim}

Exceptions:

Coprocessor Unusable
Reserved Instruction
Floating-Point
  Unimplemented Operation
  Invalid Operation
Programming Notes:

FP computational instructions, including compare, that receive an operand value of Signaling NaN, will raise the Invalid Operation condition. The 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.

```assembly
# comparisons using explicit tests for QNaN
   c.eq.d $f2,$f4 # check for equal
   nop
   bc1t L2    # it is equal
   c.un.d $f2,$f4 # it is not equal, but might be unordered
   bc1t 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
   bc1t L2    # it is equal
   nop
# it is not unordered here...
# not-equal-case code here
...
#equal-case code here
L2:
```
Floating-Point Ceiling Convert to Long Fixed-Point

 CEIL.L.fmt

<table>
<thead>
<tr>
<th>COP1</th>
<th>fmt</th>
<th>0</th>
<th>fs</th>
<th>fd</th>
<th>CEIL.L</th>
<th>0 1 0 1 0 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0 0 1 0 1 0</td>
</tr>
</tbody>
</table>

Format:  
CEIL.L.S fd, fs  
CEIL.L.D fd, fs

Purpose:  
To convert an FP value to 64-bit fixed-point, rounding up.

Description:  
fd ← convert_and_round(fs)

The value in FPR fs in format fmt, is converted to a value in 64-bit long fixed-point format rounding toward +∞ (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 $-2^{63}$ to $2^{63}-1$, the result cannot be represented correctly and an IEEE Invalid Operation condition exists. The result depends on the FP exception model currently active.

- Precise exception model: 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, $2^{63}-1$, is written to fd.
- Imprecise exception model (R8000 normal mode): The default result, $2^{63}-1$, is written to fd. No FCSR flag is set. If the Invalid Operation enable bit is set in the FCSR, an Invalid Operation exception is taken, imprecisely, at some future time.

Restrictions:

The fields fs and fd must specify valid FPRs; fs for type fmt and fd for long fixed-point; see Floating-Point Registers on page B-6. If they are not valid, the result is undefined.

The operand must be a value in format fmt; see section B 7 on page B-24. If it is not, the result is undefined and the value of the operand FPR becomes undefined.

Operation:

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

Exceptions:

- Coprocessor Unusable
- Reserved Instruction
- Floating-Point
- Invalid Operation
- Inexact
- Unimplemented Operation
- Overflow
**CEIL.W.fmt** 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>
</tr>
</thead>
<tbody>
<tr>
<td>010001</td>
<td>000000</td>
<td></td>
<td></td>
<td></td>
<td>001110</td>
</tr>
</tbody>
</table>

**Format:**
- CEIL.W.S fd, fs
- CEIL.W.D fd, fs

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

**Description:**
- \( \text{fd} \leftarrow \text{convert\_and\_round}(\text{fs}) \)
  - The value in FPR \( \text{fs} \) in format \( \text{fmt} \), is converted to a value in 32-bit word fixed-point format rounding toward \( +\infty \) (rounding mode 2). The result is placed in FPR \( \text{fd} \).

- When the source value is Infinity, NaN, or rounds to an integer outside the range \(-2^{31} \text{ to } 2^{31}-1\), the result cannot be represented correctly and an IEEE Invalid Operation condition exists. The result depends on the FP exception model currently active.
  - Precise exception model: 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 \( \text{fd} \) and an Invalid Operation exception is taken immediately. Otherwise, the default result, \( 2^{31}-1 \), is written to \( \text{fd} \).
  - Imprecise exception model (R8000 normal mode): The default result, \( 2^{31}-1 \), is written to \( \text{fd} \). No FCSR flag is set. If the Invalid Operation enable bit is set in the FCSR, an Invalid Operation exception is taken, imprecisely, at some future time.

**Restrictions:**
- The fields \( \text{fs and fd} \) must specify valid FPRs; \( \text{fs} \) for type \( \text{fmt} \) and \( \text{fd} \) for word fixed-point; see Floating-Point Registers on page B-6. If they are not valid, the result is undefined.

- The operand must be a value in format \( \text{fmt} \); see section B 7 on page B-24. If it is not, the result is undefined and the value of the operand FPR becomes undefined.

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

**Exceptions:**
- Coprocessor Unusable
- Reserved Instruction
- Floating-Point Invalid Operation
- Unimplemented Operation
- Inexact
- Overflow
**Move Control Word from Floating-Point**

**CFC1**

<table>
<thead>
<tr>
<th>Format:</th>
<th>CFC1</th>
<th>rt, fs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose:</td>
<td>To copy a word from an FPU control register to a GPR.</td>
<td></td>
</tr>
<tr>
<td>Description:</td>
<td>rt ← FP_Control[fs]</td>
<td></td>
</tr>
<tr>
<td>Copy the 32-bit word from FP (coprocessor 1) control register fs into GPR rt, sign-extending it if the GPR is 64 bits.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Restrictions:**
- There are only a couple control registers defined for the floating-point unit. The result is not defined if fs specifies a register that does not exist.
- For MIPS I, MIPS II, and MIPS III, the contents of GPR rt are undefined for the instruction immediately following CFC1.

**Operation:**
- **MIPS I - III**
  - 1: temp ← FCR[fs]
  - I+ 1: GPR[rt] ← sign_extend(temp)

**Operation:**
- **MIPS IV**
  - temp ← FCR[fs]
  - GPR[rt] ← sign_extend(temp)

**Exceptions:**
- Coprocessor Unusable
CTC1  Move Control Word to Floating-Point

<table>
<thead>
<tr>
<th>COP1</th>
<th>CT</th>
<th>rt</th>
<th>fs</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>010001</td>
<td>00110</td>
<td>00000000000000</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Format:   
CTC1  rt, fs  

MIPS I

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 FP (coprocessor 1) control register fs.

Writing to control register 31, the Floating-Point Control and Status Register or FCSR, causes the appropriate exception if any cause bit and its corresponding enable bit are both set. The register will be written before the exception occurs.

Restrictions:

There are only a couple control registers defined for the floating-point unit. The result is not defined if fs specifies a register that does not exist.

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

Operation: MIPS I - III

1: temp ← GPR[rt]31..0
I+1: FCR[fs] ← temp
     COC[1] ← FCR[31]23

Operation: MIPS IV

temp ← GPR[rt]31..0
FCR[fs] ← temp
COC[1] ← FCR[31]23

Exceptions:

- Coprocessor Unusable
- Reserved Instruction
- Floating-Point
  - Unimplemented Operation
  - Invalid Operation
  - Division-by-zero
  - Inexact
  - Overflow
  - Underflow
Floating-Point Convert to Double Floating-Point \texttt{CVT.D.fmt}

<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>10001</td>
<td>00000</td>
<td>100001</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Format:
- \texttt{CVT.D.S} \(fd, fs\)  
- \texttt{CVT.D.W} \(fd, fs\)  
- \texttt{CVT.D.L} \(fd, fs\)  

MIPS I

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 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; see Floating-Point Registers on page B-6. If they are not valid, the result is undefined.
- The operand must be a value in format \(fmt\); see section B 7 on page B-24. If it is not, the result is undefined and the value of the operand FPR becomes undefined.

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

Exceptions:
- Coprocessor Unusable
- Reserved Instruction
- Floating-Point
  - Invalid Operation
  - Unimplemented Operation
  - Inexact
  - Overflow
  - Underflow
CVT.L.fmt

Floating-Point Convert to Long Fixed-Point

<table>
<thead>
<tr>
<th>COP1</th>
<th>fmt</th>
<th>0</th>
<th>fs</th>
<th>fd</th>
<th>CVT.L</th>
</tr>
</thead>
<tbody>
<tr>
<td>010001</td>
<td>000000</td>
<td></td>
<td></td>
<td>100101</td>
<td></td>
</tr>
</tbody>
</table>

Format: CVT.L.S fd, fs

MIPS III

CVT.L.D fd, fs

Purpose: To convert an FP value to a 64-bit fixed-point.

Description: fd ← convert_and_round(fs)

Convert the value in format fmt in FPR fs to long fixed-point format, round according to the current rounding mode in FCSR, and place the result in FPR fd.

When the source value is Infinity, NaN, or rounds to an integer outside the range \(-2^{63}\) to \(2^{63}\)-1, the result cannot be represented correctly and an IEEE Invalid Operation condition exists. The result depends on the FP exception model currently active:

- **Precise exception model:** 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, \(2^{63}\)-1, is written to fd.

- **Imprecise exception model (R8000 normal mode):** The default result, \(2^{63}\)-1, is written to fd. No FCSR flag is set. If the Invalid Operation enable bit is set in the FCSR, an Invalid Operation exception is taken, imprecisely, at some future time.

Restrictions:

The fields fs and fd must specify valid FPRs; fs for type fmt and fd for long fixed-point; see Floating-Point Registers on page B-6. If they are not valid, the result is undefined.

The operand must be a value in format fmt; see section B 7 on page B-24. If it is not, the result is undefined and the value of the operand FPR becomes undefined.

Operation:

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

Exceptions:

Coprocessor Unusable
Reserved Instruction
Floating-Point
   Invalid Operation
   Unimplemented Operation
   Inexact
   Overflow
Floating-Point Convert to Single Floating-Point

**CVT.S.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>CVT.S</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

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

**Purpose:** To convert an FP or fixed-point value to single FP.

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

The value in FPR \(fs\) in format \(fmt\) is converted to a value in single floating-point format 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; see Floating-Point Registers on page B-6. If they are not valid, the result is undefined.
- The operand must be a value in format \(fmt\); see section B 7 on page B-24. If it is not, the result is undefined and the value of the operand FPR becomes undefined.

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

**Exceptions:**
- Coprocessor Unusable
- Reserved Instruction
- Floating-Point
  - Invalid Operation
  - Unimplemented Operation
  - Inexact
  - Overflow
  - Underflow
**CVT.W.fmt**

**Floating-Point Convert to Word Fixed-Point**

<table>
<thead>
<tr>
<th>COP1</th>
<th>fmt</th>
<th>0</th>
<th>fs</th>
<th>fd</th>
<th>CVT.W</th>
</tr>
</thead>
<tbody>
<tr>
<td>010001</td>
<td>000000</td>
<td>001000</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:**

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

**Purpose:**

To convert an FP value to 32-bit fixed-point.

**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 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 and an IEEE Invalid Operation condition exists. The result depends on the FP exception model currently active.

- **Precise exception model:** 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 \(fd\).

- **Imprecise exception model (R8000 normal mode):** The default result, \(2^{31}-1\), is written to \(fd\). No FCSR flag is set. If the Invalid Operation enable bit is set in the FCSR, an Invalid Operation exception is taken, imprecisely, at some future time.

**Restrictions:**

- The fields \(fs\) and \(fd\) must specify valid FPRs; \(fs\) for type \(fmt\) and \(fd\) for word fixed-point; see Floating-Point Registers on page B-6. If they are not valid, the result is undefined.
- The operand must be a value in format \(fmt\); see section B 7 on page B-24. If it is not, the result is undefined and the value of the operand FPR becomes undefined.

**Operation:**

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

**Exceptions:**

- Coprocessor Unusable
- Reserved Instruction
- Floating-Point
- Invalid Operation
- Unimplemented Operation
- Inexact
- Overflow
Floating-Point Divide \texttt{DIV.fmt}

| Format: | \texttt{DIV.S} fd, fs, ft \\
|         | \texttt{DIV.D} fd, fs, ft \\
| Purpose: | To divide FP values. \\
| Description: | \( fd \leftarrow fs / ft \) \\
| 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 \( fmt \); see Floating-Point Registers on page B-6. If they are not valid, the result is undefined. \\
| The operands must be values in format \( fmt \); see section B 7 on page B-24. If they are not, the result is undefined and the value of the operand FPRs becomes undefined. \\
| Operation: | \texttt{StoreFPR (fd, fmt, ValueFPR(fs, fmt) / ValueFPR(ft, fmt))} \\
| Exceptions: | Coprocessor Unusable \\
|             | Reserved Instruction \\
|             | Floating-Point \\
|             | Inexact Unimplemented Operation \\
|             | Division-by-zero Invalid Operation \\
|             | Overflow Underflow |
DMFC1  Doubleword Move From Floating-Point

<table>
<thead>
<tr>
<th>COP1</th>
<th>DMF</th>
<th>rt</th>
<th>fs</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0 1</td>
<td>0 0 0 0 1</td>
<td>0 0 0 0 0 0 0 0 0 0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Format: DMFC1 rt, fs

Purpose: To copy a doubleword from an FPR to a GPR.

Description: rt ← fs

The doubleword contents of FPR fs are placed into GPR rt.

If the coprocessor 1 general registers are 32-bits wide (a native 32-bit processor or 32-bit register emulation mode in a 64-bit processor), FPR fs is held in an even/odd register pair. The low word is taken from the even register fs and the high word is from fs+1.

Restrictions:

If fs does not specify an FPR that can contain a doubleword, the result is undefined; see Floating-Point Registers on page B-6.

For MIPS III, the contents of GPR rt are undefined for the instruction immediately following DMFC1.

Operation: MIPS I - III

1: if SizeFGR() = 64 then /* 64-bit wide FGRs */
    data ← FGR[fs]
  elseif fs0 = 0 then /* valid specifier, 32-bit wide FGRs */
    data ← FGR[fs+1] || FGR[fs]
  else /* undefined for odd 32-bit FGRs */
    UndefinedResult()
  endif
1+1: GPR[rt] ← data

Operation: MIPS IV

if SizeFGR() = 64 then /* 64-bit wide FGRs */
    data ← FGR[fs]
  elseif fs0 = 0 then /* valid specifier, 32-bit wide FGRs */
    data ← FGR[fs+1] || FGR[fs]
  else /* undefined for odd 32-bit FGRs */
    UndefinedResult()
  endif
  GPR[rt] ← data

Exceptions:

Reserved Instruction
Coprocessor Unusable
Doubleword Move To Floating-Point DMTC1

Format: \texttt{DMTC1 \ rt, fs}  \quad \text{MIPS III}

Purpose: To copy a doubleword from a GPR to an FPR.

Description: \( \text{fs} \leftarrow \text{rt} \)

The doubleword contents of GPR \( \text{rt} \) are placed into FPR \( \text{fs} \).

If coprocessor 1 general registers are 32-bits wide (a native 32-bit processor or 32-bit register emulation mode in a 64-bit processor), FPR \( \text{fs} \) is held in an even/odd register pair. The low word is placed in the even register \( \text{fs} \) and the high word is placed in \( \text{fs} + 1 \).

Restrictions:

- If \( \text{fs} \) does not specify an FPR that can contain a doubleword, the result is undefined; see Floating-Point Registers on page B-6.

- For MIPS III, the contents of FPR \( \text{fs} \) are undefined for the instruction immediately following DMTC1.

Operation: MIPS I - III

\begin{verbatim}
 1:  data ← GPR[rt]  
 1+1: if SizeFGR() = 64 then /* 64-bit wide FGRs */
    FGR[fs] ← data
  else if fs0 = 0 then /* valid specifier, 32-bit wide FGRs */
    FGR[fs+1] ← data63..32
    FGR[fs] ← data31..0
  else /* undefined result for odd 32-bit FGRs */
    UndefinedResult()
  endif
\end{verbatim}

Operation: MIPS IV

\begin{verbatim}
data ← GPR[rt]  
if SizeFGR() = 64 then /* 64-bit wide FGRs */
  FGR[fs] ← data
else if fs0 = 0 then /* valid specifier, 32-bit wide FGRs */
  FGR[fs+1] ← data63..32
  FGR[fs] ← data31..0
else /* undefined result for odd 32-bit FGRs */
  UndefinedResult()
endif
\end{verbatim}
DMTC1  Doubleword Move To Floating-Point

Exceptions:
   Reserved Instruction
   Coprocessor Unusable
Floating-Point Floor Convert to Long Fixed-Point  

**FLOOR.L.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>0</td>
<td>0</td>
<td>0</td>
<td>fs</td>
<td>fd</td>
<td>FLOOR.L</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

**Format:**
- FLOOR.L.S  fd, fs
- FLOOR.L.D  fd, fs

**MIPS III**

**Purpose:** To convert an FP value to 64-bit fixed-point, rounding down.

**Description:**
\[
f_d \leftarrow \text{convert\_and\_round}(f_s)
\]

The value in FPR \(f_s\) in format \(fmt\), is converted to a value in 64-bit long fixed-point format rounding toward \(-\infty\) (rounding mode 3). The result is placed in FPR \(f_d\).

When the source value is Infinity, NaN, or rounds to an integer outside the range \(-2^{63}\) to \(2^{63}-1\), the result cannot be represented correctly and an IEEE Invalid Operation condition exists. The result depends on the FP exception model currently active.

- Precise exception model: 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 \(f_d\) and an Invalid Operation exception is taken immediately. Otherwise, the default result, \(2^{63}-1\), is written to \(f_d\).

- Imprecise exception model (R8000 normal mode): The default result, \(2^{63}-1\), is written to \(f_d\). No FCSR flag is set. If the Invalid Operation enable bit is set in the FCSR, an Invalid Operation exception is taken, imprecisely, at some future time.

**Restrictions:**

- The fields \(f_s\) and \(f_d\) must specify valid FPRs; \(f_s\) for type \(fmt\) and \(f_d\) for long fixed-point; see Floating-Point Registers on page B-6. If they are not valid, the result is undefined.

- The operand must be a value in format \(fmt\); see section B 7 on page B-24. If it is not, the result is undefined and the value of the operand FPR becomes undefined.

**Operation:**

\[
\text{StoreFPR}(f_d, L, \text{ConvertFmt(ValueFPR}(f_s, \text{fmt}), \text{fmt}, L))
\]

**Exceptions:**

- Coprocessor Unusable
- Reserved Instruction
- Floating-Point
  - Invalid Operation
  - Inexact
- Unimplemented Operation
- Overflow
**FLOOR.W.fmt**  
Floating-Point Floor Convert to Word Fixed-Point

<table>
<thead>
<tr>
<th>COP1</th>
<th>fmt</th>
<th>fs</th>
<th>fd</th>
<th>FLOOR.W</th>
</tr>
</thead>
<tbody>
<tr>
<td>010001</td>
<td>000000</td>
<td></td>
<td></td>
<td>0011111</td>
</tr>
</tbody>
</table>

**Format:**  
FLOOR.W.S  fd, fs  
FLOOR.W.D  fd, fs  

**MIPS II**

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

**Description:**  
\[ fd \leftarrow \text{convert} \_\text{and} \_\text{round}(fs) \]

The value in FPR \( fs \) in format \( fmt \), is converted to a value in 32-bit word fixed-point format rounding 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 and an IEEE Invalid Operation condition exists. The result depends on the FP exception model currently active.

- **Precise exception model:** 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 \( fd \).
- **Imprecise exception model (R8000 normal mode):** The default result, \(2^{31}-1\), is written to \( fd \). No FCSR flag is set. If the Invalid Operation enable bit is set in the FCSR, an Invalid Operation exception is taken, imprecisely, at some future time.

**Restrictions:**

The fields \( fs \) and \( fd \) must specify valid FPRs; \( fs \) for type \( fmt \) and \( fd \) for word fixed-point; see Floating-Point Registers on page B-6. If they are not valid, the result is undefined.

The operand must be a value in format \( fmt \); see section B 7 on page B-24. If it is not, the result is undefined and the value of the operand FPR becomes undefined.

**Operation:**

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

**Exceptions:**

- Coprocessor Unusable
- Reserved Instruction
- Floating-Point
  - Invalid Operation
  - Inexact
- Unimplemented Operation
  - Overflow
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>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**MIPS II**

**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.

If coprocessor 1 general registers are 32-bits wide (a native 32-bit processor or 32-bit register emulation mode in a 64-bit processor), FPR ft is held in an even/odd register pair. The low word is placed in the even register ft and the high word is placed in ft+1.

**Restrictions:**
If ft does not specify an FPR that can contain a doubleword, the result is undefined; see Floating-Point Registers on page B-6.

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

**MIPS IV:** The low-order 3 bits of the offset field must be zero. If they are not, the result of the instruction is undefined.

**Operation:**

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

if v\text{Addr}2..0 ≠ 0 then SignalException(AddressError) endif

(p\text{Addr}, uncached) ← AddressTranslation (v\text{Addr}, DATA, LOAD)

data ← LoadMemory(uncached, DOUBLEWORD, p\text{Addr}, v\text{Addr}, DATA)

if SizeFGR() = 64 then /* 64-bit wide FGRs */

FGR[ft] ← data

elseif ft0 = 0 then /* valid specifier, 32-bit wide FGRs */

FGR[ft+1] ← data_{63..32}

FGR[ft] ← data_{31..0}

else /* undefined result for odd 32-bit FGRs */

UndefinedResult()

endif

**Exceptions:**

Coprocessor unusable
Reserved Instruction
TLB Refill, TLB Invalid
Address Error
LDXC1 Load Doubleword Indexed to Floating-Point

Format:

LDXC1 fd, index(base)

MIPS IV

Purpose:

To load a doubleword from memory to an FPR (GPR+GPR addressing).

Description:

fd ← memory[base+index]

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

If coprocessor 1 general registers are 32-bits wide (a native 32-bit processor or 32-bit register emulation mode in a 64-bit processor), FPR fd is held in an even/odd register pair. The low word is placed in the even register fd and the high word is placed in fd+1.

Restrictions:

If fd does not specify an FPR that can contain a doubleword, the result is undefined; see Floating-Point Registers on page B-6.

The Region bits of the effective address must be supplied by the contents of base. If EffectiveAddress_{63..62} ≠ base_{63..62}, the result is undefined.

An Address Error exception occurs if EffectiveAddress_{2..0} ≠ 0 (not doubleword-aligned).

MIPS IV: The low-order 3 bits of the offset field must be zero. If they are not, the result of the instruction is undefined.

Operation:

vAddr ← GPR[base] + GPR[index]
if vAddr_{2..0} ≠ 0^3 then SignalException(AddressError) endif
(pAddr, uncached) ← AddressTranslation(vAddr, DATA, LOAD)
memb ← LoadMemory(unchched, DOUBLEWORD, pAddr, vAddr, DATA)
if SizeFGR() = 64 then /* 64-bit wide FGRs */
    FGR[fd] ← data
elseif fd_0 = 0 then /* valid specifier, 32-bit wide FGRs */
    FGR[fd+1] ← data_{63..32}
    FGR[fd] ← data_{31..0}
else /* undefined result for odd 32-bit FGRs */
    UndefinedResult()
endif
Load Doubleword Indexed to Floating-Point LDXC1

Exceptions:
  T LB R efill, T LB I nvalid
  A ddress E rror
  R eserved I nstruction
  C oprocessor U nusable
LWC1

Load Word to Floating-Point

<table>
<thead>
<tr>
<th>Format:</th>
<th>LWC1 ft, offset(base)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose:</td>
<td>To load a word from memory to an FPR.</td>
</tr>
<tr>
<td>Description:</td>
<td>ft ← memory[base+offset]</td>
</tr>
</tbody>
</table>

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.

If coprocessor 1 general registers are 64-bits wide, bits 63..32 of register ft become undefined. See Floating-Point Registers on page B-6.

Restrictions:
- An Address Error exception occurs if EffectiveAddress1..0 ≠ 0 (not word-aligned).
- MIPS IV: The low-order 2 bits of the offset field must be zero. If they are not, the result of the instruction is undefined.

Operation: 32-bit Processors

```
i: /* "mem" is aligned 64-bits from memory. Pick out correct bytes. */
vAddr ← sign_extend(offset) + GPR[base]
if vAddr1..0 ≠ 02 then SignalException(AddressError) endif
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, LOAD)
mem ← LoadMemory(uncached, WORD, pAddr, vAddr, DATA)
i + 1: FGR[ft] ← mem
```

Operation: 64-bit Processors

```
/* "mem" is aligned 64-bits from memory. Pick out correct bytes. */
vAddr ← sign_extend(offset) + GPR[base]
if vAddr1..0 ≠ 02 then SignalException(AddressError) endif
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← pAddrPSIZE-1..3 || (pAddr2..0 xor (ReverseEndian || 02))
mem ← LoadMemory(uncached, WORD, pAddr, vAddr, DATA)
bytesel ← vAddr2..0 xor (BigEndianCPU || 02)
if SizeFGR() = 64 then /* 64-bit wide FGRs */
    FGR[ft] ← undefined32 || mem31+8*bytesel..8*bytesel
else /* 32-bit wide FGRs */
    FGR[ft] ← mem31+8*bytesel..8*bytesel
endif
```
Load Word to Floating-Point \texttt{LWC1}

Exceptions:
- Coprocessor unusable
- Reserved Instruction
- TLB Refill, TLB Invalid
- Address Error
**LWXC1**

**Load Word Indexed to Floating-Point**

<table>
<thead>
<tr>
<th>COP1X</th>
<th>base</th>
<th>index</th>
<th>0</th>
<th>fd</th>
<th>LWXC1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 1 1</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>0 0 0 0 0 0</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:**
LWXC1   fd, index(base) 

**MIPS IV**

**Purpose:**
To load a word from memory to an FPR (GPR+GPR addressing).

**Description:**

\[ fd \leftarrow \text{memory}[\text{base}+\text{index}] \]

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 \( fd \). The contents of GPR \( \text{index} \) and GPR \( \text{base} \) are added to form the effective address.

If coprocessor 1 general registers are 64-bits wide, bits 63..32 of register \( fd \) become undefined. See Floating-Point Registers on page B-6.

**Restrictions:**

The Region bits of the effective address must be supplied by the contents of \( \text{base} \). If \( \text{EffectiveAddress}_{63..62} \neq \text{base}_{63..62} \), the result is undefined.

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

MIPS IV: The low-order 2 bits of the offset field must be zero. If they are not, the result of the instruction is undefined.

**Operation:**

\[ \text{vAddr} \leftarrow \text{GPR}[\text{base}] + \text{GPR}[\text{index}] \]

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

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

\[ \text{pAddr} \leftarrow \text{pAddr}_{\text{SIZE}-1..3} \| (\text{pAddr}_{2..0} \text{xor} (\text{ReverseEndian} \| 0^2)) \]

/* "mem" is aligned 64-bits from memory. Pick out correct bytes. */

\[ \text{mem} \leftarrow \text{LoadMemory}(\text{uncached}, \text{WORD}, \text{pAddr}, \text{vAddr}, \text{DATA}) \]

\[ \text{bytesel} \leftarrow \text{vAddr}_{2..0} \text{xor} (\text{BigEndianCPU} \| 0^2) \]

if \( \text{SizeFGR}() = 64 \) then /* 64-bit wide FGRs */

\[ \text{FGR}[\text{fd}] \leftarrow \text{undefined}\| \text{mem}_{31+8^*\text{bytesel}} \]

else /* 32-bit wide FGRs */

\[ \text{FGR}[\text{fd}] \leftarrow \text{mem}_{31+8^*\text{bytesel}} \]

endif

**Exceptions:**

TLB Refill, TLB Invalid
Address Error
Reserved Instruction
Coprocessor Unusable
Floating-Point Multiply Add \textbf{MADD.fmt}

<table>
<thead>
<tr>
<th>COP1X</th>
<th>fr</th>
<th>ft</th>
<th>fs</th>
<th>fd</th>
<th>MADD</th>
<th>fmt</th>
</tr>
</thead>
<tbody>
<tr>
<td>010011</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

Format: \textbf{MADD.S} fd, fr, fs, ft \textbf{MADD.D} fd, fr, fs, ft

Purpose: To perform a combined multiply-then-add of FP values.

Description: \textbf{fd} \leftarrow (fs \times ft) + fr

The value in FPR \textit{fs} is multiplied by the value in FPR \textit{ft} to produce a product. The value in FPR \textit{fr} is added to the product. The result sum is calculated to infinite precision, rounded according to the current rounding mode in FCSR, and placed into FPR \textit{fd}. The operands and result are values in format \textit{fmt}.

The accuracy of the result depends which of two alternative arithmetic models is used by the implementation for the computation. The numeric models are explained in Arithmetic Instructions on page B-21.

Restrictions:

The fields \textit{fr, fs, ft,} and \textit{fd} must specify FPRs valid for operands of type \textit{fmt}; see Floating-Point Registers on page B-6. If they are not valid, the result is undefined.

The operands must be values in format \textit{fmt}; see section B 7 on page B-24. If they are not, the result is undefined and the value of the operand FPRs becomes undefined.

Operation:

\begin{align*}
\textit{vfr} & \leftarrow \text{ValueFPR}(\textit{fr, fmt}) \\
\textit{vfs} & \leftarrow \text{ValueFPR}(\textit{fs, fmt}) \\
\textit{vft} & \leftarrow \text{ValueFPR}(\textit{ft, fmt}) \\
\text{StoreFPR}(\textit{fd, fmt, vfr + vfs \times vft})
\end{align*}

Exceptions:

- Coprocessor Unusable
- Reserved Instruction
- Floating-Point
- Inexact
- Unimplemented Operation
- Invalid Operation
- Overflow
- Underflow
**MFC1**: Move Word From Floating-Point

### Format:

```
MFC1   rt, fs
```

### MIPS I

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

### Description:

```
rt ← fs
```

The low word from FPR `fs` is placed into the low word of GPR `rt`. If GPR `rt` is 64 bits wide, then the value is sign extended. See Floating-Point Registers on page B-6.

### Restrictions:

For MIPS I, MIPS II, and MIPS III the contents of GPR `rt` are undefined for the instruction immediately following MFC1.

### Operation: MIPS I - III

```
I:    word ← FGR[fs]_{31..0}
I+1:  GPR[rt] ← sign_extend(word)
```

### Operation: MIPS IV

```
word ← FGR[fs]_{31..0}
GPR[rt] ← sign_extend(word)
```

### Exceptions:
Coprocessor Unusable
Floating-Point Move \texttt{MOV.fmt}

<table>
<thead>
<tr>
<th>COP1</th>
<th>fmt</th>
<th>fs</th>
<th>fd</th>
<th>MOV</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0 1</td>
<td>0 0 0 0 0</td>
<td>fd</td>
<td>fs</td>
<td>0 0 0 1 1 0</td>
</tr>
</tbody>
</table>

Format: \texttt{MOV.S} \( \text{fd}, \text{fs} \)  
\texttt{MOV.D} \( \text{fd}, \text{fs} \)  

MIPS I

Purpose: To move an FP value between FPRs.

Description: \( \text{fd} \leftarrow \text{fs} \)

The value in FPR \( \text{fs} \) is placed into FPR \( \text{fd} \). The source and destination are values in format \( \text{fmt} \).

The move is non-arithmetic; it causes no IEEE 754 exceptions.

Restrictions:

The fields \( \text{fs} \) and \( \text{fd} \) must specify FPRs valid for operands of type \( \text{fmt} \); see Floating-Point Registers on page B-6. If they are not valid, the result is undefined.

The operand must be a value in format \( \text{fmt} \); see section B 7 on page B-24. If it is not, the result is undefined and the value of the operand FPR becomes undefined.

Operation:

\text{StoreFPR}(\text{fd}, \text{fmt}, \text{ValueFPR}(\text{fs}, \text{fmt}))

Exceptions:

- Coprocessor Unusable
- Reserved Instruction
- Floating-Point
  - Unimplemented Operation
MOVF Move Conditional on FP False

| 31 | 26 | 25 | 21 | 20 | 18 | 17 | 16 | 15 | 11 | 10 | 6 | 5 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|
| SPECIAL | rs | cc | 0 | tf | 0 | rd | 0 | 0 | MOVCI | 0 | 0 | 0 | 0 | 1 |

Format: MOVF rd, rs, cc

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:
None

Operation:

active ← FCC[cc] = tf
if active then
    GPR[rd] ← GPR[rs]
endif

Exceptions:
Reserved Instruction
Coprocessor Unusable
Floating-Point Move Conditional on FP False MOVF.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>0 1 0 0 0 1</td>
<td>6 5 3 1 1 5</td>
<td>5 6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Format: MOVF.S fd, fs, cc
        MOVF.D fd, fs, cc

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 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; see Floating-Point Registers on page B-6. If they are not valid, the result is undefined.

The operand must be a value in format fmt; see section B 7 on page B-24. If it is not, the result is undefined and the value of the operand FPR becomes undefined.

Operation:

if FCC[cc] = tf then
    StoreFPR(fd, fmt, ValueFPR(fs, fmt))
else
    StoreFPR(fd, fmt, ValueFPR(fd, fmt))
endif

Exceptions:

Coprocessor Unusable
Reserved Instruction
Floating-Point
Unimplemented operation
MOVN.fmt Floating-Point Move Conditional on Not Zero

<table>
<thead>
<tr>
<th>COP1</th>
<th>fmt</th>
<th>rt</th>
<th>fs</th>
<th>fd</th>
<th>MOVN.fmt</th>
</tr>
</thead>
<tbody>
<tr>
<td>0100001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0100011</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

Format:  
- MOVN.S     fd, fs, rt  
- MOVN.D     fd, fs, rt  

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 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; see Floating-Point Registers on page B-6.  If they are not valid, the result is undefined.  
The operand must be a value in format fmt; see section B 7 on page B-24.  If it is not, the  
result is undefined and the value of the operand FPR becomes undefined.

Operation:  
if GPR[rt] \neq 0 then  
    StoreFPR(fd, fmt, ValueFPR(fs, fmt))  
else  
    StoreFPR(fd, fmt, ValueFPR(fd, fmt))  
endif

Exceptions:  
- Coprocessor Unusable  
- Reserved Instruction  
- Floating-Point  
- Unimplemented operation

MIPS IV
Move Conditional on FP True

MOVT

<table>
<thead>
<tr>
<th>SPECIAL</th>
<th>rs</th>
<th>cc</th>
<th>tf</th>
<th>rd</th>
<th>0</th>
<th>MOVCI</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 0</td>
<td>6</td>
<td>5</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>

MIPS IV

Format: MOVT rd, rs, cc

Purpose: To test an FP condition code then conditionally move a GPR.

Description: if (cc = 1) then rd ← rs

If the floating-point condition code specified by cc is one then the contents of GPR rs are placed into GPR rd.

Restrictions:
None

Operation:
if FCC[cc] = tf then
   GPR[rd] ← GPR[rs]
endif

Exceptions:
Reserved Instruction
Coprocessor Unusable
MOVT.fmt  Floating-Point Move Conditional on FP True

<table>
<thead>
<tr>
<th>COP1</th>
<th>fmt</th>
<th>cc</th>
<th>0</th>
<th>tf</th>
<th>fs</th>
<th>fd</th>
<th>MOVCF</th>
</tr>
</thead>
<tbody>
<tr>
<td>010001</td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>010001</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; see Floating-Point Registers  on page B-6.  If they are not valid, the result is undefined.

The operand must be a value in format  fmt; see section  B 7  on page B-24.  If it is not, the result is undefined and the value of the operand FPR becomes undefined.

Operation:  
if FCC[cc] = tf then  
StoreFPR(fd, fmt, ValueFPR(fs, fmt))  
else  
StoreFPR(fd, fmt, ValueFPR(fd, fmt))  
endif

Exceptions:  
Coprocessor Unusable  
Reserved Instruction  
Floating-Point  
Unimplemented operation
Floating-Point Move Conditional on Zero \texttt{MOVZ.fmt}

<table>
<thead>
<tr>
<th>31</th>
<th>26 25</th>
<th>21 20</th>
<th>16 15</th>
<th>11 10</th>
<th>6 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>MOVZ</td>
<td>0 1 0 0 0 1 0</td>
</tr>
</tbody>
</table>

Format: \texttt{MOVZ.S} fd, fs, rt  
\texttt{MOVZ.D} fd, fs, rt  

MIPS IV

Purpose: To test a GPR then conditionally move an FP value.

Description: \[\text{if} \ (rt = 0) \ \text{then} \ fd \leftarrow fs\]

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 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 \(fmt\); see Floating-Point Registers on page B-6. If they are not valid, the result is undefined.

The operand must be a value in format \(fmt\); see section B 7 on page B-24. If it is not, the result is undefined and the value of the operand FPR becomes undefined.

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
Unimplemented operation
**MSUB.fmt**

### Floating-Point Multiply Subtract

<table>
<thead>
<tr>
<th>COP1X</th>
<th>fr</th>
<th>ft</th>
<th>fs</th>
<th>fd</th>
<th>MSUB</th>
<th>fmt</th>
</tr>
</thead>
<tbody>
<tr>
<td>010011</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

**Format:**
- **MSUB.S** \( fd, fr, fs, ft \)
- **MSUB.D** \( fd, fr, fs, ft \)

**Purpose:**
To perform a combined multiply-then-subtract of FP values.

**Description:**
\[
fd \leftarrow (fs \times ft) - fr
\]
The value in FPR \( fs \) is multiplied by the value in FPR \( ft \) to produce an intermediate product. The value in FPR \( fr \) is subtracted from the product. The subtraction 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 \( fmt \).

The accuracy of the result depends which of two alternative arithmetic models is used by the implementation for the computation. The numeric models are explained in Arithmetic Instructions on page B-21.

**Restrictions:**
1. The fields \( fr, fs, ft, \) and \( fd \) must specify FPRs valid for operands of type \( fmt \); see Floating-Point Registers on page B-6. If they are not valid, the result is undefined.
2. The operands must be values in format \( fmt \); see section B 7 on page B-24. If they are not, the result is undefined and the value of the operand FPRs becomes undefined.

**Operation:**
- \( vfr \leftarrow \text{ValueFPR}(fr, fmt) \)
- \( vfs \leftarrow \text{ValueFPR}(fs, fmt) \)
- \( vft \leftarrow \text{ValueFPR}(ft, fmt) \)
- StoreFPR(fd, fmt, (vfs * vft) - vfr)

**Exceptions:**
- Reserved Instruction
- Coprocessor Unusable
- Floating-Point
  - Inexact
  - Invalid Operation
  - Underflow
  - Unimplemented Operation
  - Overflow
  - Underflow
Move Word to Floating-Point

MTC1

<table>
<thead>
<tr>
<th>COP1</th>
<th>MT</th>
<th>rt</th>
<th>fs</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0 1</td>
<td>0 0 1 0 0</td>
<td></td>
<td>0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
</tr>
</tbody>
</table>

Format:  MTC1 rt, fs

MIPS I

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. If coprocessor 1 general registers are 64-bits wide, bits 63..32 of register fs become undefined. See Floating-Point Registers on page B-6.

Restrictions:

For MIPS I, MIPS II, and MIPS III the value of FPR fs is undefined for the instruction immediately following MTC1.

Operation:  MIPS I - III

I:  data ← GPR[rt]_{31,0}  /* 64-bit wide FGRs */
I+1: if SizeFGR() = 64 then
     FGR[fs] ← undefined^{32} || data
     else  /* 32-bit wide FGRs */
              FGR[fs] ← data
     endif

Operation:  MIPS IV

data ← GPR[rt]_{31,0}
if SizeFGR() = 64 then  /* 64-bit wide FGRs */
     FGR[fs] ← undefined^{32} || data
     else  /* 32-bit wide FGRs */
              FGR[fs] ← data
     endif

Exceptions:
Coprocessor Unusable
MUL.fmt Floating-Point Multiply

<table>
<thead>
<tr>
<th>MUL.S  fd, fs, ft</th>
<th>MUL.D  fd, fs, ft</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose: To multiply FP values.</td>
<td></td>
</tr>
<tr>
<td>Description: fd ← fs × ft</td>
<td></td>
</tr>
</tbody>
</table>

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; see Floating-Point Registers on page B-6. If they are not valid, the result is undefined.

The operands must be values in format fmt; see section B 7 on page B-24. If they are not, the result is undefined and the value of the operand FPRs becomes undefined.

Operation:

StoreFPR (fd, fmt, ValueFPR(fs, fmt) × ValueFPR(ft, fmt))

Exceptions:

- Coprocessor Unusable
- Reserved Instruction
- Floating-Point
  - Inexact
  - Unimplemented Operation
  - Invalid Operation
  - Overflow
  - Underflow
Floating-Point Negate

### Format:

- **NEG.S**  \( \text{fd, fs} \)
- **NEG.D**  \( \text{fd, fs} \)

### Purpose:

To negate an FP value.

### Description:

\( \text{fd} \leftarrow - (\text{fs}) \)

The value in FPR \( \text{fs} \) is negated and placed into FPR \( \text{fd} \). The value is negated by changing the sign bit value. The operand and result are values in format \( \text{fmt} \).

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

### Restrictions:

- The fields \( \text{fs} \) and \( \text{fd} \) must specify FPRs valid for operands of type \( \text{fmt} \); see Floating-Point Registers on page B-6. If they are not valid, the result is undefined.

- The operand must be a value in format \( \text{fmt} \); see section B 7 on page B-24. If it is not, the result is undefined and the value of the operand FPR becomes undefined.

### Operation:

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

### Exceptions:

- Coprocessor Unusable
- Reserved Instruction
- Floating-Point
  - Unimplemented Operation
  - Invalid Operation
NMADD.fmt  Floating-Point Negative Multiply Add

<table>
<thead>
<tr>
<th>COP1X</th>
<th>fr</th>
<th>ft</th>
<th>fs</th>
<th>fd</th>
<th>NMADD</th>
<th>fmt</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 1 1</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>3 1 0</td>
<td>3</td>
</tr>
</tbody>
</table>

Format:  NMADD.S  fd, fr, fs, ft
        NMADD.D  fd, fr, fs, ft

Purpose:  To negate a combined multiply-then-add of FP values.

Description:  fd ← - ((fs × ft) + fr)

The value in FPR fs is multiplied by the value in FPR ft to produce an intermediate product. The value in FPR fr is added to the product. The result sum is calculated to infinite precision, rounded according to the current rounding mode in FCSR, negated by changing the sign bit, and placed into FPR fd. The operands and result are values in format fmt.

The accuracy of the result depends which of two alternative arithmetic models is used by the implementation for the computation. The numeric models are explained in Arithmetic Instructions on page B-21.

Restrictions:

The fields fr, fs, ft, and fd must specify FPRs valid for operands of type fmt; see Floating-Point Registers on page B-6. If they are not valid, the result is undefined.

The operands must be values in format fmt; see section B 7 on page B-24. If they are not, the result is undefined and the value of the operand FPRs becomes undefined.

Operation:

vfr ← ValueFPR(fr, fmt)
vfs ← ValueFPR(fs, fmt)
vtft ← ValueFPR(ft, fmt)
StoreFPR(fd, fmt, -(vfr + vfs * vft))

Exceptions:

Coprocessor Unusable
Reserved Instruction
Floating-Point
Inexact  Unimplemented Operation
Invalid Operation  Overflow
Underflow
Floating-Point Negative Multiply Subtract \texttt{NMSUB.fmt}

<table>
<thead>
<tr>
<th>COP1X</th>
<th>fr</th>
<th>ft</th>
<th>fs</th>
<th>fd</th>
<th>NMSUB</th>
<th>fmt</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 1 1</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

Format: \begin{align*}
\text{NMSUB.S} & \quad \text{fd, fr, fs, ft} \\
\text{NMSUB.D} & \quad \text{fd, fr, fs, ft}
\end{align*}

Purpose: To negate a combined multiply-then-subtract of FP values.

Description: \begin{align*}
\text{fd} & \leftarrow - \left( (\text{fs} \times \text{ft}) - \text{fr} \right) \\
\text{The value in FPR fr is multiplied by the value in FPR ft to produce an intermediate product. The value in FPR fr is subtracted from the product. The result is calculated to infinite precision, rounded according to the current rounding mode in FCSR, negated by changing the sign bit, and placed into FPR fd. The operands and result are values in format fmt.}
\end{align*}

The accuracy of the result depends which of two alternative arithmetic models is used by the implementation for the computation. The numeric models are explained in \textit{Arithmetic Instructions} on page B-21.

Restrictions:

The fields fr, fs, ft, and fd must specify FPRs valid for operands of type fmt; see \textit{Floating-Point Registers} on page B-6. If they are not valid, the result is undefined.

The operands must be values in format fmt; see section B 7 on page B-24. If they are not, the result is undefined and the value of the operand FPRs becomes undefined.

Operation:

\begin{align*}
\text{vfr} & \leftarrow \text{ValueFPR(fr, fmt)} \\
\text{vfs} & \leftarrow \text{ValueFPR(fs, fmt)} \\
\text{vft} & \leftarrow \text{ValueFPR(ft, fmt)} \\
\text{StoreFPR(fd, fmt, -(vfs \times vft) - vfr)}
\end{align*}

Exceptions:

\begin{itemize}
  \item Reserved Instruction
  \item Coprocessor Unusable
  \item Floating-Point
    \begin{itemize}
      \item Inexact
      \item Invalid Operation
      \item Underflow
    \end{itemize}
  \item Unimplemented Operation
  \item Overflow
\end{itemize}
**PREFX**

Prefetch Indexed

<table>
<thead>
<tr>
<th>COP1X</th>
<th>base</th>
<th>index</th>
<th>hint</th>
<th>0</th>
<th>PREFX</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 1 1</td>
<td></td>
<td></td>
<td></td>
<td>0 0 0 0 0</td>
<td>0 0 1 1 1 1</td>
</tr>
</tbody>
</table>

**Format:**

PREFX hint, index(base)

**MIPS IV**

**Purpose:**

To prefetch locations from memory (GPR+GPR addressing).

**Description:**

prefetch_memory[base+index]

PREFX adds the contents of GPR `index` to the contents of GPR `base` to form an effective byte address. It advises that data at the effective address may be used in the near future. The `hint` field supplies information about the way that the data is expected to be used.

PREFX is an advisory instruction. It may change the performance of the program. For all `hint` values, it neither changes architecturally-visible state nor alters the meaning of the program. An implementation may do nothing when executing a PREFX instruction.

If MIPS IV instructions are supported and enabled and Coprocessor 1 is enabled (allowing access to CP1X), PREFX does not cause addressing-related exceptions. If it raises an exception condition, the exception condition is ignored. If an addressing-related exception condition is raised and ignored, no data will be prefetched. Even if no data is prefetched in such a case, some action that is not architecturally-visible, such as writeback of a dirty cache line, might take place.

PREFX will never generate a memory operation for a location with an uncached memory access type (see Memory Access Types on page A-12).

If PREFX 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.

PREFX enables the processor to take some action, typically prefetching the data into cache, to improve program performance. The action taken for a specific PREFX instruction is both system and context dependent. Any action, including doing nothing, is permitted that does not change architecturally-visible state or alter the meaning of a program. It is expected that implementations will either do nothing or take an action that will increase the performance of the program.

For a cached location, the expected, and useful, action is for the processor 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 *hint* field supplies information about the way the data is expected to be used. No *hint* value causes an action that modifies architecturally-visible state. A processor may use a *hint* value to improve the effectiveness of the prefetched action. The defined *hint* values and the recommended prefetch action are shown in the table below. The *hint* table may be extended in future implementations.

**Restrictions:**

The Region bits of the effective address must be supplied by the contents of base. If EffeciveAddress\textsubscript{63..62} \neq base\textsubscript{63..62}, the result of the instruction is undefined.

**Operation:**

\begin{align*}
  \text{vAddr} & \leftarrow \text{GPR}[\text{base}] + \text{GPR}[\text{index}] \\
  (\text{pAddr, uncached}) & \leftarrow \text{AddressTranslation}(\text{vAddr, DATA, LOAD}) \\
  \text{Prefetch}(\text{uncached, pAddr, vAddr, DATA, hint})
\end{align*}

**Table B-22  Values of Hint Field for Prefetch 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>Data is expected to be loaded (not modified). Fetch data as if for a load.</td>
</tr>
<tr>
<td>1</td>
<td>store</td>
<td>Data is expected to be stored or modified. Fetch data as if for a store.</td>
</tr>
<tr>
<td>2-3</td>
<td></td>
<td>Not yet defined.</td>
</tr>
<tr>
<td>4</td>
<td>load_streamed</td>
<td>Data is expected to be loaded (not modified) but not reused extensively; it will “stream” through cache. Fetch data as if for a load and place it in the cache so that it will not displace data prefetched as “retained”.</td>
</tr>
<tr>
<td>5</td>
<td>store_streamed</td>
<td>Data is expected to be stored or modified but not reused extensively; it will “stream” through cache. Fetch data as if for a store and place it in the cache so that it will not displace data prefetched as “retained”.</td>
</tr>
<tr>
<td>6</td>
<td>load_retained</td>
<td>Data is expected to be loaded (not modified) and reused extensively; it should be “retained” in the cache. Fetch data as if for a load and place it in the cache so that it will not be displaced by data prefetched as “streamed”.</td>
</tr>
<tr>
<td>7</td>
<td>store_retained</td>
<td>Data is expected to be stored or modified and reused extensively; it should be “retained” in the cache. Fetch data as if for a store and place it in the cache so that will not be displaced by data prefetched as “streamed”.</td>
</tr>
<tr>
<td>8-31</td>
<td></td>
<td>Not yet defined.</td>
</tr>
</tbody>
</table>
Exceptions:
- Reserved Instruction
- Coprocessor Unusable

Programming Notes:
Prefetch can not 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 will not cause an exception to prefetch using an address pointer value before the validity of a pointer is determined.

Implementation Notes:
It is recommended that a reserved hint field value either cause a default prefetch action that is expected to be useful for most cases of data use, such as the “load” hint, or cause the instruction to be treated as a NOP.
## Reciprocal Approximation

<table>
<thead>
<tr>
<th>COP1</th>
<th>fmt</th>
<th>fs</th>
<th>fd</th>
<th>RECIP</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0 1</td>
<td>0 0 0 0 0</td>
<td></td>
<td></td>
<td>0 1 0 1 0 1</td>
</tr>
</tbody>
</table>

**Format:**

- RECIP.S  $fd, fs$
- RECIP.D  $fd, fs$

**Purpose:**

To approximate the reciprocal of an FP value (quickly).

**Description:**

\[
fd \leftarrow 1.0 / fs
\]

The reciprocal of the value in FPR $fs$ is approximated and placed into FPR $fd$.

The operand and result are values in format $fmt$.

The numeric accuracy of this operation is implementation dependent; it does not meet the accuracy specified by the IEEE 754 Floating-Point standard. The computed result differs from the both the exact result and the IEEE-mandated representation of the exact result by no more than one unit in the least-significant place (ulp).

It is implementation dependent whether the result is affected by the current rounding mode in FCSR.

**Restrictions:**

The fields $fs$ and $fd$ must specify FPRs valid for operands of type $fmt$; see Floating-Point Registers on page B-6. If they are not valid, the result is undefined.

The operand must be a value in format $fmt$; see section B 7 on page B-24. If it is not, the result is undefined and the value of the operand FPR becomes undefined.

**Operation:**

\[
\text{StoreFPR}(fd, fmt, 1.0 / \text{valueFPR}(fs, fmt))
\]

**Exceptions:**

- Coprocessor Unusable
- Reserved Instruction
- Floating-Point
  - Inexact
  - Division-by-zero
  - Overflow
  - Unimplemented Operation
  - Invalid Operation
  - Underflow
ROUND.L.fmt Floating-Point Round to Long Fixed-Point

<table>
<thead>
<tr>
<th></th>
<th>COP1</th>
<th>fmt</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>ROUND.L</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>010001</td>
<td>000000</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>
<td></td>
<td>001000</td>
</tr>
</tbody>
</table>

Format:  
- ROUND.L.S fd, fs  
- ROUND.L.D fd, fs  

MIPS III

Purpose:  
To convert an FP value to 64-bit fixed-point, rounding to nearest.

Description:  
- fd \leftarrow \text{convert} \_\text{and} \_\text{round}(fs)

The value in FPR $fs$ in format $fmt$, is converted to a value in 64-bit long 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 $-2^{63}$ to $2^{63}-1$, the result cannot be represented correctly and an IEEE Invalid Operation condition exists. The result depends on the FP exception model currently active.

- Precise exception model: 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, $2^{63}-1$, is written to $fd$.

- Imprecise exception model (R8000 normal mode): The default result, $2^{63}-1$, is written to $fd$. No FCSR flag is set. If the Invalid Operation enable bit is set in the FCSR, an Invalid Operation exception is taken, imprecisely, at some future time.

Restrictions:

The fields $fs$ and $fd$ must specify valid FPRs; $fs$ for type $fmt$ and $fd$ for long fixed-point; see Floating-Point Registers on page B-6. If they are not valid, the result is undefined.

The operand must be a value in format $fmt$; see section B 7 on page B-24. If it is not, the result is undefined and the value of the operand FPR becomes undefined.

Operation:

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

Exceptions:

- Coprocessor Unusable
- Reserved Instruction
- Floating-Point
  - Inexact
  - Unimplemented Operation
  - Overflow
  - Invalid Operation
Floating-Point Round to Word Fixed-Point ROUND.W.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>0</td>
<td>0</td>
<td>0</td>
<td>fs</td>
<td>fd</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

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

Purpose:   To convert an FP value to 32-bit fixed-point, rounding to nearest.

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 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. The result depends on the FP exception model currently active.

- Precise exception model: 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.
- Imprecise exception model (R8000 normal mode): The default result, \(2^{31}-1\), is written to fd. No FCSR flag is set. If the Invalid Operation enable bit is set in the FCSR, an Invalid Operation exception is taken, imprecisely, at some future time.

Restrictions:

The fields fs and fd must specify valid FPRs; fs for type fmt and fd for word fixed-point; see Floating-Point Registers on page B-6. If they are not valid, the result is undefined.

The operand must be a value in format fmt; see section B 7 on page B-24. If it is not, the result is undefined and the value of the operand FPR becomes undefined.

Operation:

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

Exceptions:

- Coprocessor Unusable
- Reserved Instruction
- Floating-Point Inexact
- Invalid Operation
- Unimplemented Operation
- Overflow
RSQRT.fmt

Reciprocal Square Root Approximation

<table>
<thead>
<tr>
<th>COP1</th>
<th>fmt</th>
<th>fs</th>
<th>fd</th>
<th>RSQRT</th>
</tr>
</thead>
<tbody>
<tr>
<td>010001</td>
<td>000000</td>
<td></td>
<td>010110</td>
<td></td>
</tr>
</tbody>
</table>

Format:
- RSQRT.S   fd, fs
- RSQRT.D   fd, fs

MIPS IV

Purpose:
To approximate the reciprocal of the square root of an FP value (quickly).

Description:
\[ fd \leftarrow \frac{1}{\sqrt{fs}} \]
The reciprocal of the positive square root of the value in FPR \(fs\) is approximated and placed into FPR \(fd\). The operand and result are values in format \(fmt\).

The numeric accuracy of this operation is implementation dependent; it does not meet the accuracy specified by the IEEE 754 Floating-Point standard. The computed result differs from the both the exact result and the IEEE-mandated representation of the exact result by no more than two units in the least-significant place (ulp).

It is implementation dependent whether the result is affected by the current rounding mode in FCSR.

Restrictions:
The fields \(fs\) and \(fd\) must specify FPRs valid for operands of type \(fmt\); see Floating-Point Registers on page B-6. If they are not valid, the result is undefined.

The operand must be a value in format \(fmt\); see section B 7 on page B-24. If it is not, the result is undefined and the value of the operand FPR becomes undefined.

Operation:
\[ \text{StoreFPR}(fd, fmt, 1.0 / \text{SquareRoot(valueFPR}(fs, fmt))) \]

Exceptions:
- Coprocessor Unusable
- Reserved Instruction
- Floating-Point
  - Inexact
  - Division-by-zero
  - Overflow
- Unimplemented Operation
- Invalid Operation
- Underflow
Store Doubleword from Floating-Point SDC1

<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>SDC1</td>
<td>ft</td>
<td>offset(base)</td>
<td>MIPS II</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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.

If coprocessor 1 general registers are 32-bits wide (a native 32-bit processor or 32-bit register emulation mode in a 64-bit processor), FPR ft is held in an even/odd register pair. The low word is taken from the even register ft and the high word is from ft+1.

Restrictions:

If ft does not specify an FPR that can contain a doubleword, the result is undefined; see Floating-Point Registers on page B-6.

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

MIPS IV: The low-order 3 bits of the offset field must be zero. If they are not, the result of the instruction is undefined.

Operation:

vAddr ← sign_extend(offset) + GPR[base]
if vAddr2..0 ≠ 0 then SignalException(AddressError) endif
(pAddr, uncached) ← AddressTranslation(vAddr, DATA, STORE)
if SizeFGR() = 64 then /* 64-bit wide FGRs */
data ← FGR[ft]
else if ft₀ = 0 then /* valid specifier, 32-bit wide FGRs */
data ← FGR[ft+1] || FGR[ft]
else /* undefined for odd 32-bit FGRs */
UndefinedResult()
endif
StoreMemory(uncached, DOUBLEWORD, data, pAddr, vAddr, DATA)

Exceptions:

Coprocessor unusable
Reserved Instruction
TLB Refill, TLB Invalid
TLB Modified
Address Error
SDXC1 Store Doubleword Indexed from Floating-Point

Format: SDXC1   fs, index(base)

Purpose: To store a doubleword from an FPR to memory (GPR+GPR addressing).

Description: memory[base+index] ← fs

- The 64-bit doubleword in FPR fs is stored in memory at the location specified by the aligned effective address. The contents of GPR index and GPR base are added to form the effective address.

- If coprocessor 1 general registers are 32-bits wide (a native 32-bit processor or 32-bit register emulation mode in a 64-bit processor), FPR fs is held in an even/odd register pair. The low word is taken from the even register fs and the high word is from fs+1.

Restrictions:

- If fs does not specify an FPR that can contain a doubleword, the result is undefined; see Floating-Point Registers on page B-6.

- The Region bits of the effective address must be supplied by the contents of base. If EffectiveAddress63..62 ≠ base63..62, the result is undefined.

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

- MIPS IV: The low-order 3 bits of the offset field must be zero. If they are not, the result of the instruction is undefined.

Operation:

vAddr ← GPR[base] + GPR[index]
if vAddr2..0 ≠ 03 then SignalException(AddressError) endif
(pAddr, uncached) ← AddressTranslation(vAddr, DATA, STORE)
if SizeFGR() = 64 then /* 64-bit wide FGRs */
data ← FGR[fs]
elseif fs0 = 0 then /* valid specifier, 32-bit wide FGRs */
data ← FGR[fs+1] || FGR[fs]
else /* undefined for odd 32-bit FGRs */
UndefinedResult()
endif
StoreMemory(uncached, DOUBLEWORD, data, pAddr, vAddr, DATA)
Store Doubleword Indexed from Floating-Point SDXC1

Exceptions:
- TLB Refill, TLB Invalid
- TLB Modified
- Address Error
- Reserved Instruction
- Coprocessor Unusable
**SQRT.fmt**  
Floating-Point Square Root

<table>
<thead>
<tr>
<th>COP1</th>
<th>fmt</th>
<th>fs</th>
<th>fd</th>
<th>SQRT</th>
</tr>
</thead>
<tbody>
<tr>
<td>010001</td>
<td>000000</td>
<td>05</td>
<td>55</td>
<td>000100</td>
</tr>
</tbody>
</table>

**Format:**

- SQRT.S  fd, fs
- SQRT.D  fd, fs

**MIPS II**

**Purpose:** To compute the square root of an FP value.

**Description:**

\[ fd \leftarrow SQRT(fs) \]

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 will be \(-0\).

**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 \(fmt\); see Floating-Point Registers on page B-6. If they are not valid, the result is undefined.

The operand must be a value in format \(fmt\); see section B 7 on page B-24. If it is not, the result is undefined and the value of the operand FPR becomes undefined.

**Operation:**

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

**Exceptions:**

- Coprocessor Unusable
- Reserved Instruction
- Floating-Point
  - Unimplemented Operation
  - Invalid Operation
  - Inexact
### Floating-Point Subtract (SUB.fmt)

<table>
<thead>
<tr>
<th>COP1</th>
<th>fmt</th>
<th>ft</th>
<th>fs</th>
<th>fd</th>
<th>SUB</th>
</tr>
</thead>
<tbody>
<tr>
<td>010001</td>
<td>000001</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Format:
- **SUB.S**: fd, fs, ft
- **SUB.D**: fd, fs, ft

#### 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; see Floating-Point Registers on page B-6. If they are not valid, the result is undefined.
- The operands must be values in format fmt; see section B 7 on page B-24. If they are not, the result is undefined and the value of the operand FPRs becomes undefined.

#### Operation:
- StoreFPR (fd, fmt, ValueFPR(fs, fmt) – ValueFPR(ft, fmt))

#### Exceptions:
- Coprocessor Unusable
- Reserved Instruction
- Floating-Point
  - Inexact
  - Invalid Operation
  - Underflow
- Unimplemented Operation
- Overflow
SWC1 Store Word from 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>SWC1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Base**: ft, offset(base)

**MIPS I**

**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).

- MIPS IV: The low-order 2 bits of the offset field must be zero. If they are not, the result of the instruction is undefined.

**Operation: 32-bit Processors**

vAddr ← sign_extend(offset) + GPR[base]

if vAddr1..0 ≠ 02 then SignalException(AddressError) endif

(pAddr, uncached) ← AddressTranslation (vAddr, DATA, STORE)

data ← FGR[ft]

StoreMemory (uncached, WORD, data, pAddr, vAddr, DATA)

**Operation: 64-bit Processors**

vAddr ← sign_extend(offset) + GPR[base]

if vAddr1..0 ≠ 02 then SignalException(AddressError) endif

(pAddr, uncached) ← AddressTranslation (vAddr, DATA, STORE)

pAddr ← pAddrPSIZE-1..3 || (pAddr2..0 xor (ReverseEndian || 02))

bytesel ← vAddr2..0 xor (BigEndianCPU || 02)

/* the bytes of the word are moved into the correct byte lanes */

if SizeFGR() = 64 then

data ← 032-8*bytesel || FGR[ft]31..0 || 08*bytesel /* top or bottom wd of 64-bit data */

else

data ← 032-8*bytesel || FGR[ft] || 08*bytesel /* top or bottom wd of 64-bit data */

endif

StoreMemory (uncached, WORD, data, pAddr, vAddr, DATA)

**Exceptions**:

- Coprocessor unusable
- Reserved Instruction
- TLB Refill, TLB Invalid
- TLB Modified
- Address Error
Store Word Indexed from Floating-Point SWXC1

Format: \texttt{SWXC1 \ fs, index(base)}

MIPS IV

Purpose: To store a word from an FPR to memory (GPR+GPR addressing).

Description: memory\texttt{[base+index]} ← \texttt{fs}

The low 32-bit word from FPR \texttt{fs} is stored in memory at the location specified by the aligned effective address. The contents of GPR \texttt{index} and GPR \texttt{base} are added to form the effective address.

Restrictions:

The Region bits of the effective address must be supplied by the contents of \texttt{base}. If \texttt{EffectiveAddress}_{63..62} \neq \texttt{base}_{63..62}, the result is undefined.

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

MIPS IV: The low-order 2 bits of the \texttt{offset} field must be zero. If they are not, the result of the instruction is undefined.

Operation:

\begin{align*}
\text{vAddr} & \leftarrow \text{GPR[base]} + \text{GPR[index]}
\text{if vAddr}_{1..0} \neq 0^2 \text{ then SignalException(AddressError) endif}
\text{pAddr} & \leftarrow \text{AddressTranslation(vAddr, DATA, STORE)}
\text{pAddr} & \leftarrow \text{pAddr}_{PSIZE-1..3} \oplus (\text{pAddr}_{2..0} \oplus (\text{ReverseEndian} \oplus 0^2))
\text{bytesel} & \leftarrow \text{vAddr}_{2..0} \oplus (\text{BigEndianCPU} \oplus 0^2)
\text{/* the bytes of the word are moved into the correct byte lanes */}
\text{if SizeFGR() = 64 then}
\text{data} & \leftarrow 0^{32-8*\text{bytesel}} || \text{FGR[fs]}_{31..0} || 0^{8*\text{bytesel}} /* top or bottom wd of 64-bit data */
\text{else}
\text{data} & \leftarrow 0^{32-8*\text{bytesel}} || \text{FGR[fs]} || 0^{8*\text{bytesel}} /* 32-bit wide FGRs */
\text{endif}
\text{StoreMemory (uncached, WORD, data, pAddr, vAddr, DATA)}
\end{align*}

Exceptions:

- TLB Refill, TLB Invalid
- TLB Modified
- Address Error
- Reserved Instruction
- Coprocessor Unusable
TRUNC.L.fmt Floating-Point Truncate to Long Fixed-Point

Format: TRUNC.L.S fd, fs
        TRUNC.L.D fd, fs

Purpose: To convert an FP value to 64-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 64-bit long fixed-point format 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 $-2^{63}$ to $2^{63}-1$, the result cannot be represented correctly and an IEEE Invalid Operation condition exists. The result depends on the FP exception model currently active.

- Precise exception model: 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, $2^{63}-1$, is written to fd.

- Imprecise exception model (R8000 normal mode): The default result, $2^{63}-1$, is written to fd. No FCSR flag is set. If the Invalid Operation enable bit is set in the FCSR, an Invalid Operation exception is taken, imprecisely, at some future time.

Restrictions:

The fields fs and fd must specify valid FPRs; fs for type fmt and fd for long fixed-point; see Floating-Point Registers on page B-6. If they are not valid, the result is undefined.

The operand must be a value in format fmt; see section B 7 on page B-24. If it is not, the result is undefined and the value of the operand FPR becomes undefined.

Operation:

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

Exceptions:

- Coprocessor Unusable
- Reserved Instruction
- Floating-Point
  - Inexact
  - Invalid Operation
  - Unimplemented Operation
  - Overflow
Floating-Point Truncate to Word Fixed-Point \textbf{TRUNC.W.fmt}

\begin{center}
\begin{tabular}{|c|c|c|c|c|c|c|c|c|}
\hline
& 31 & 26 & 25 & 21 & 20 & 16 & 15 & 11 & 10 & 6 & 5 & 0 \\
\hline
COP1 & 0 & 1 & 0 & 0 & 0 & 1 \\
fmt & & & & & & & & & & & & \\
0 & 0 & 0 & 0 & 0 \\
fs & & & & & & & & & & & & \\
fd & & & & & & & & & & & & \\
TRUNC.W & 0 & 0 & 1 & 1 & 0 & 1 \\
\hline
\end{tabular}
\end{center}

Format: \textbf{TRUNC.W.S} \hspace{1em} \textit{fd, fs} \hspace{1em} \textbf{MIPS II} \\
\textbf{TRUNC.W.D} \hspace{1em} \textit{fd, fs}

Purpose: To convert an FP value to 32-bit fixed-point, rounding toward zero.

Description: \textit{fd} $\leftarrow$ convert\_and\_round\(\textit{fs}\)

The value in FPR \textit{fs} in format \textit{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 \textit{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. The result depends on the FP exception model currently active.

- Precise exception model: 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 \textit{fd} and an Invalid Operation exception is taken immediately. Otherwise, the default result, \(2^{31}-1\), is written to \textit{fd}.
- Imprecise exception model (R8000 normal mode): The default result, \(2^{31}-1\), is written to \textit{fd}. No FCSR flag is set. If the Invalid Operation enable bit is set in the FCSR, an Invalid Operation exception is taken, imprecisely, at some future time.

Restrictions:

The fields \textit{fs} and \textit{fd} must specify valid FPRs; \textit{fs} for type \textit{fmt} and \textit{fd} for word fixed-point; see Floating-Point Registers on page B-6. If they are not valid, the result is undefined.

The operand must be a value in format \textit{fmt}; see section B 7 on page B-24. If it is not, the result is undefined and the value of the operand FPR becomes undefined.

Operation:

\texttt{StoreFPR(fd, W, ConvertFmt(ValueFPR(fs, fmt), fmt, W))}

Exceptions:

- Coprocessor Unusable
- Reserved Instruction
- Floating-Point Inexact
- Overflow
- Invalid Operation
- Unimplemented Operation
### B.11 FPU Instruction Formats

An FPU instruction is a single 32-bit aligned word. The distinct FP instruction layouts are shown in Figure B-16. Variable information is in lower-case labels, such as “offset”. Upper-case labels and any numbers indicate constant data. A table follows all the layouts that explains the fields used in them. Note that the same field may have different names in different instruction layout pictures. The field name is mnemonic to the function of that field in the instruction layout. The opcode tables and the instruction decode discussion use the canonical field names: opcode, fmt, nd, tf, and function. The other fields are not used for instruction decode.

#### Figure B-16   FPU Instruction Formats

<table>
<thead>
<tr>
<th>Immediate: load/store using register + offset addressing.</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 26 25 21 20 16 15 0</td>
</tr>
<tr>
<td>opcode base ft offset</td>
</tr>
<tr>
<td>6 5 5 16</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Register: 2-register and 3-register formatted arithmetic operations.</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 26 25 21 20 16 15 11 10 6 5 0</td>
</tr>
<tr>
<td>COP1 fmt ft fs fd function</td>
</tr>
<tr>
<td>6 5 5 5 5 6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Register Immediate: data transfer -- CPU ←→ FPU register.</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 26 25 21 20 16 15 11 10 0</td>
</tr>
<tr>
<td>COP1 sub rt fs 0</td>
</tr>
<tr>
<td>6 5 5 5 11</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Condition code, Immediate: conditional branches on FPU cc using PC + offset.</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 26 25 21 20 18 17 16 15 0</td>
</tr>
<tr>
<td>COP1 BC cc nd tf offset</td>
</tr>
<tr>
<td>6 5 3 1 1 16</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Register to Condition Code: formatted FP compare.</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 26 25 21 20 16 15 8 7 6 5 0</td>
</tr>
<tr>
<td>COP1 fmt ft fs cc 0 function</td>
</tr>
<tr>
<td>6 5 5 5 3 2 4</td>
</tr>
</tbody>
</table>
Condition Code, Register FP: FPU register move-conditional on FP cc.

| 31 26 25 21 20 18 17 16 15 11 10 6 5 0 |
| COP1 | fmt | cc | 0 | tf | fs | fd | MOVCF |

Register-4: 4-register formatted arithmetic operations.

| 31 26 25 21 20 16 15 11 10 6 5 3 2 0 |
| COP1X | fr | ft | fs | fd | function | op4 | fmt3 |

Register Index: Load/store using register + register addressing.

| 31 26 25 21 20 16 15 11 10 6 5 0 |
| COP1X | base | index | 0 | fd | function |

Register Index hint: Prefetch using register + register addressing.

| 31 26 25 21 20 16 15 11 10 6 5 0 |
| COP1X | base | index | hint | 0 | PREFX |

Condition Code, Register Integer: CPU register move-conditional on FP cc.

<p>| 31 26 25 21 20 18 17 16 15 11 10 6 5 0 |
| SPECIAL | rs | cc | 0 | tf | rd | 0 | MOVCI |</p>
<table>
<thead>
<tr>
<th><strong>BC</strong></th>
<th>Branch Conditional instruction subcode (op=COP1)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>base</strong></td>
<td>CPU register: base address for address calculations</td>
</tr>
<tr>
<td><strong>COP1</strong></td>
<td>Coprocessor 1 primary opcode value in op field.</td>
</tr>
<tr>
<td><strong>COP1X</strong></td>
<td>Coprocessor 1 eXtended primary opcode value in op field.</td>
</tr>
<tr>
<td><strong>cc</strong></td>
<td>condition code specifier. For architecture levels prior to MIPS IV it must be zero.</td>
</tr>
<tr>
<td><strong>fd</strong></td>
<td>FPU register: destination (arithmetic, loads, move-to) or source (stores, move-from)</td>
</tr>
<tr>
<td><strong>fmt</strong></td>
<td>destination and/or operand type (“format”) specifier</td>
</tr>
<tr>
<td><strong>fr</strong></td>
<td>FPU register: source</td>
</tr>
<tr>
<td><strong>fs</strong></td>
<td>FPU register: source</td>
</tr>
<tr>
<td><strong>ft</strong></td>
<td>FPU register: source (for stores, arithmetic) or destination (for loads)</td>
</tr>
<tr>
<td><strong>function</strong></td>
<td>function field specifying a function within a particular op operation code.</td>
</tr>
<tr>
<td><strong>function:</strong></td>
<td>op4 is a 3-bit function field specifying which 4-register arithmetic operation for COP1X, fmt3 is a 3-bit field specifying the format of the operands and destination. The combinations are shown as several distinct instructions in the opcode tables.</td>
</tr>
<tr>
<td><strong>hint</strong></td>
<td>hint field made available to cache controller for prefetch operation</td>
</tr>
<tr>
<td><strong>index</strong></td>
<td>CPU register, holds index address component for address calculations</td>
</tr>
<tr>
<td><strong>MOVC</strong></td>
<td>Value in function field for conditional move. There is one value for the instruction with op=COP1, another for the instruction with op=SPECIAL.</td>
</tr>
<tr>
<td><strong>nd</strong></td>
<td>nullify delay. If set, branch is Likely and delay slot instruction is not executed. This must be zero for MIPS I.</td>
</tr>
<tr>
<td><strong>offset</strong></td>
<td>signed offset field used in address calculations</td>
</tr>
<tr>
<td><strong>op</strong></td>
<td>primary operation code (COP1, COP1X, LWC1, SWC1, LDC1, SDC1, SPECIAL)</td>
</tr>
<tr>
<td><strong>PREFIX</strong></td>
<td>Value in function field for prefetch instruction for op=COP1X</td>
</tr>
<tr>
<td><strong>rd</strong></td>
<td>CPU register: destination</td>
</tr>
<tr>
<td><strong>rs</strong></td>
<td>CPU register: source</td>
</tr>
<tr>
<td><strong>rt</strong></td>
<td>CPU register: source / destination</td>
</tr>
<tr>
<td><strong>SPECIAL</strong></td>
<td>SPECIAL primary opcode value in op field.</td>
</tr>
<tr>
<td><strong>sub</strong></td>
<td>Operation subcode field for COP1 register immediate mode instructions.</td>
</tr>
<tr>
<td><strong>tf</strong></td>
<td>true/false. The condition from FP compare is tested for equality with tf bit.</td>
</tr>
</tbody>
</table>
B.12 FPU (CP1) Instruction Opcode Bit Encoding

This section describes the encoding of the Floating-Point Unit (FPU) instructions for the four levels of the MIPS architecture, MIPS I through MIPS IV. Each architecture level includes the instructions in the previous level; MIPS IV includes all instructions in MIPS I, MIPS II, and MIPS III. This section presents eight different views of the instruction encoding.

- Separate encoding tables for each architecture level.
- A MIPS IV encoding table showing the architecture level at which each opcode was originally defined and subsequently modified (if modified).
- Separate encoding tables for each architecture revision showing the changes made during that revision.

B 12.1 Instruction Decode

Instruction field names are printed in bold in this section.

The primary opcode field is decoded first. The opcode values LWC1, SWC1, LDC1, and SDC1 fully specify FPU load and store instructions. The opcode values COP1, COP1X, and SPECIAL specify instruction classes. Instructions within a class are further specified by values in other fields.

B 12.1.1 COP1 Instruction Class

The opcode =COP1 instruction class encodes most of the FPU instructions. The class is further decoded by examining the fmt field. The fmt values fully specify the CPU ↔ FPU register move instructions and specify the S, D, W, L, and BC instruction classes.

The opcode =COP1 + fmt =BC instruction class encodes the conditional branch instructions. The class is further decoded, and the instructions fully specified, by examining the nd and tf fields.

The opcode =COP1 + fmt = (S, D, W, or L) instruction classes encode instructions that operate on formatted (typed) operands. Each of these instruction classes is further decoded by examining the function field. With one exception the function values fully specify instructions. The exception is the MOVCF instruction class.

The opcode =COP1 + fmt = (S or D) + function =MOVCF instruction class encodes the MOVT(fmt and MOVF(fmt conditional move instructions (to move FP values based on FP condition codes). The class is further decoded, and the instructions fully specified, by examining the tf field.

† An exception to this rule is that the reserved, but never implemented, Coprocessor 3 instructions were removed or changed to another use starting in MIPS III.
B 12.1.2 COP1X Instruction Class

The opcode =COP1X instruction class encodes the indexed load/store instructions, the indexed prefetch, and the multiply accumulate instructions. The class is further decoded, and the instructions fully specified, by examining the function field.

B 12.1.3 SPECIAL Instruction Class

The opcode =SPECIAL instruction class is further decoded by examining the function field. The only function value that applies to FPU instruction encoding is the MOVCI instruction class. The remainder of the function values encode CPU instructions.

The opcode =SPECIAL + function =MOVCI instruction class encodes the MOVT and MOVF conditional move instructions (to move CPU registers based on FP condition codes). The class is further decoded, and the instructions fully specified, by examining the tf field.

B 12.2 Instruction Subsets of MIPS III and MIPS IV Processors.

MIPS III processors, such as the R4000, R4200, R4300, R4400, and R4600, have a processor mode in which only the MIPS II instructions are valid. The MIPS II encoding table describes the MIPS II-only mode.

MIPS IV processors, such as the R8000 and R10000, have processor modes in which only the MIPS II or MIPS III instructions are valid. The MIPS II encoding table describes the MIPS II-only mode. The MIPS III encoding table describes the MIPS III-only mode.
### Table B-23  FPU (CP1) Instruction Encoding - MIPS I Architecture

Instructions encoded by the opcode field.

<table>
<thead>
<tr>
<th>opcode bits 28..26</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 011</td>
<td>COP1 δ</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5 101</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6 110</td>
<td>LWC1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7 111</td>
<td>SWC1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Instructions encoded by the fmt field when opcode=COP1.

<table>
<thead>
<tr>
<th>opcode = COP1</th>
<th>fmt bits 23.21</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 000</td>
<td>MFC1 *</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 01</td>
<td>BC δ</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>2 10</td>
<td>S δ</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>3 11</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
</tbody>
</table>

Instructions encoded by the tf field when opcode=COP1 and fmt=BC.

<table>
<thead>
<tr>
<th>opcode = COP1</th>
<th>fmt = BC</th>
<th>tf bit 16</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BC1F</td>
<td>BC1T</td>
</tr>
</tbody>
</table>
### Table B-23 (cont.)  FPU (CP1) Instruction Encoding - MIPS I Architecture

Instructions encoded by the function field when opcode=CP1 and fmt = S, D, or W

#### Encoding when fmt = S

<table>
<thead>
<tr>
<th>bits 5..3</th>
<th>function</th>
<th>bits 0..7</th>
<th>31 26 25 24 23 22 21 20</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>ADD SUB MUL DIV * * * * *</td>
<td>000 001 010 011 100 101 110 111</td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
<tr>
<td>001</td>
<td>* * * * * * * * * * * *</td>
<td></td>
<td></td>
</tr>
<tr>
<td>010</td>
<td>* * * * * * * * * * * *</td>
<td></td>
<td></td>
</tr>
<tr>
<td>011</td>
<td>* * * * * * * * * * * *</td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>* CVT.D * * * CVT.W * * *</td>
<td></td>
<td></td>
</tr>
<tr>
<td>101</td>
<td>* * * * * * * * * * * *</td>
<td></td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>C.F α C.UN α C.EQ α C.UEQ α C.OLT α C.ULT α C.OLE α C.ULE α</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111</td>
<td>CSF α C.NGLE α C.SEQ α C.NGL α C.LT α C.NGE α C.LE α C.NGT α</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Encoding when fmt = D

<table>
<thead>
<tr>
<th>bits 5..3</th>
<th>function</th>
<th>bits 0..7</th>
<th>31 26 25 24 23 22 21 20</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>ADD SUB MUL DIV * * * * *</td>
<td>000 001 010 011 100 101 110 111</td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
<tr>
<td>001</td>
<td>* * * * * * * * * * * *</td>
<td></td>
<td></td>
</tr>
<tr>
<td>010</td>
<td>* * * * * * * * * * * *</td>
<td></td>
<td></td>
</tr>
<tr>
<td>011</td>
<td>* * * * * * * * * * * *</td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>* CVT.S * * * CVT.W * * *</td>
<td></td>
<td></td>
</tr>
<tr>
<td>101</td>
<td>* * * * * * * * * * * *</td>
<td></td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>C.F α C.UN α C.EQ α C.UEQ α C.OLT α C.ULT α C.OLE α C.ULE α</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111</td>
<td>CSF α C.NGLE α C.SEQ α C.NGL α C.LT α C.NGE α C.LE α C.NGT α</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
encoding when \( \text{fmt} = W \)

<table>
<thead>
<tr>
<th>bits 5..3</th>
<th>000</th>
<th>001</th>
<th>010</th>
<th>011</th>
<th>100</th>
<th>101</th>
<th>110</th>
<th>111</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>1</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>2</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>3</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>4</td>
<td>CVTS</td>
<td>CVT.D</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>5</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>6</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>7</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
</tbody>
</table>

opcode \( \text{COP1} \)

<table>
<thead>
<tr>
<th>function</th>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>fmt ( = W )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table B-24  FPU (CP1) Instruction Encoding - MIPS II Architecture

Instructions encoded by the **opcode** field.

<table>
<thead>
<tr>
<th>opcode bits 28..26</th>
<th>31 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 000</td>
<td>000 001 010 011 100 101 110 111</td>
</tr>
<tr>
<td>2 010</td>
<td>COP1 δ</td>
</tr>
<tr>
<td>3 011</td>
<td>χ</td>
</tr>
<tr>
<td>4 100</td>
<td>LWC1</td>
</tr>
<tr>
<td>5 101</td>
<td>LDC1</td>
</tr>
<tr>
<td>6 110</td>
<td>SWC1</td>
</tr>
<tr>
<td>7 111</td>
<td>SDC1</td>
</tr>
</tbody>
</table>

Instructions encoded by the **fmt** field when opcode=COP1.

<table>
<thead>
<tr>
<th>fmt bits 23..21</th>
<th>31 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 000</td>
<td>MFC1 CFC1 MTC1 CTC1 MFC1 * CFC1 * MTC1 * CTC1 *</td>
</tr>
<tr>
<td>1 01</td>
<td>BC δ * * * * * * * * * * * * * * * * * * * * * * * *</td>
</tr>
<tr>
<td>2 10</td>
<td>S δ D δ * * * * W δ * * * * * * * * * * * * * * * * * *</td>
</tr>
<tr>
<td>3 11</td>
<td>* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *</td>
</tr>
</tbody>
</table>

Instructions encoded by the **nd** and **tf** fields when opcode=COP1 and fmt=BC.

<table>
<thead>
<tr>
<th>t f bit 16</th>
<th>31 26 25 24 23 22 21 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>BC1F BC1T</td>
</tr>
<tr>
<td>1 1</td>
<td>BC1FL BC1TL</td>
</tr>
</tbody>
</table>
### Table B-24 (cont.)  FPU (CP1) Instruction Encoding - MIPS II Architecture

Instructions encoded by the function field when opcode=COP1 and fmt = S, D, or W

<table>
<thead>
<tr>
<th>function</th>
<th>bits 2..0</th>
<th>opcode = COP1</th>
<th>fmt</th>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>fmt = S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fmt = D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### encoding when fmt = S

<table>
<thead>
<tr>
<th>function</th>
<th>bits 2..0</th>
<th>opcode = COP1</th>
<th>fmt</th>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>fmt = S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fmt = D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### encoding when fmt = D

<table>
<thead>
<tr>
<th>function</th>
<th>bits 2..0</th>
<th>opcode = COP1</th>
<th>fmt</th>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>fmt = S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fmt = D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
encoding when fmt = W

<table>
<thead>
<tr>
<th>functi on</th>
<th>bits 2..0</th>
</tr>
</thead>
<tbody>
<tr>
<td>bits 5..3</td>
<td>000 001 010 011 100 101 110 111</td>
</tr>
<tr>
<td>0 000</td>
<td>* * * * * * * *</td>
</tr>
<tr>
<td>1 001</td>
<td>* * * * * * * *</td>
</tr>
<tr>
<td>2 010</td>
<td>* * * * * * * *</td>
</tr>
<tr>
<td>3 011</td>
<td>* * * * * * * *</td>
</tr>
<tr>
<td>4 100</td>
<td>CVTS CVT.D * * * * * *</td>
</tr>
<tr>
<td>5 101</td>
<td>* * * * * * * *</td>
</tr>
<tr>
<td>6 110</td>
<td>* * * * * * * *</td>
</tr>
<tr>
<td>7 111</td>
<td>* * * * * * * *</td>
</tr>
</tbody>
</table>
Table B-25  FPU (CP1) Instruction Encoding - MIPS III Architecture
Instructions encoded by the opcode  field.

<table>
<thead>
<tr>
<th>opcode</th>
<th>bits 28..26</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 000</td>
<td>000 001 010 011 100 101 110 111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5 101</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6 110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7 111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Instructions encoded by the fmt  field when opcode=COP1.

<table>
<thead>
<tr>
<th>opcode</th>
<th>fmt</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 000</td>
<td>MFC1 DMFC1 CFC1 * MTC1 DMT1 CTC1 *</td>
</tr>
<tr>
<td>1 01</td>
<td>BC δ * * * * * *</td>
</tr>
<tr>
<td>2 10</td>
<td>S δ D δ * * W δ L δ * *</td>
</tr>
<tr>
<td>3 11</td>
<td>* * * * * * * *</td>
</tr>
</tbody>
</table>

Instructions encoded by the nd  and tf  fields when opcode=COP1 and fmt=BC.

<table>
<thead>
<tr>
<th>opcode</th>
<th>fmt</th>
<th>nd</th>
<th>tf</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 000</td>
<td>MFC1 DMFC1 CFC1 * MTC1 DMT1 CTC1 *</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 01</td>
<td>BC δ * * * * * *</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 10</td>
<td>S δ D δ * * W δ L δ * *</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 11</td>
<td>* * * * * * * *</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table B-25 (cont.)  FPU (CP1) Instruction Encoding - MIPS III Architecture

Instructions encoded by the function field when opcode=COP1 and fmt = S, D, W, or L

testing when fmt = S

<table>
<thead>
<tr>
<th>function bits 2..0</th>
<th>bits 5..3</th>
<th>000</th>
<th>001</th>
<th>010</th>
<th>011</th>
<th>0111</th>
<th>100</th>
<th>101</th>
<th>110</th>
<th>111</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>ADD</td>
<td>SUB</td>
<td>MUL</td>
<td>DIV</td>
<td>SQRT</td>
<td>ABS</td>
<td>MOV</td>
<td>NEG</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>ROUND.L</td>
<td>TRUNC.L</td>
<td>CEILL</td>
<td>FLOOR.L</td>
<td>ROUND.W</td>
<td>TRUNC.W</td>
<td>CEILL</td>
<td>FLOOR.W</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>*</td>
<td>CVT.D</td>
<td>*</td>
<td>*</td>
<td>CVT.W</td>
<td>CVT.L</td>
<td>*</td>
<td>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>C.F α</td>
<td>C.UN α</td>
<td>C.EQ α</td>
<td>C.UEQ α</td>
<td>C.OLT α</td>
<td>C.ULE α</td>
<td>C.LE α</td>
<td>C.LT α</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>C.SF α</td>
<td>C.NGLE α</td>
<td>C.SEQ α</td>
<td>C.NGL α</td>
<td>C.NGE α</td>
<td>C.LNE α</td>
<td>C.NLE α</td>
<td>C.NGT α</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

testing when fmt = D

<table>
<thead>
<tr>
<th>function bits 2..0</th>
<th>bits 5..3</th>
<th>000</th>
<th>001</th>
<th>010</th>
<th>011</th>
<th>0111</th>
<th>100</th>
<th>101</th>
<th>110</th>
<th>111</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>ADD</td>
<td>SUB</td>
<td>MUL</td>
<td>DIV</td>
<td>SQRT</td>
<td>ABS</td>
<td>MOV</td>
<td>NEG</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>ROUND.L</td>
<td>TRUNC.L</td>
<td>CEILL</td>
<td>FLOOR.L</td>
<td>ROUND.W</td>
<td>TRUNC.W</td>
<td>CEILL</td>
<td>FLOOR.W</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>CVT.S</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>CVT.W</td>
<td>CVT.L</td>
<td>*</td>
<td>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>C.F α</td>
<td>C.UN α</td>
<td>C.EQ α</td>
<td>C.UEQ α</td>
<td>C.OLT α</td>
<td>C.ULE α</td>
<td>C.LE α</td>
<td>C.LT α</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>C.SF α</td>
<td>C.NGLE α</td>
<td>C.SEQ α</td>
<td>C.NGL α</td>
<td>C.NGE α</td>
<td>C.LNE α</td>
<td>C.NLE α</td>
<td>C.NGT α</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
encoding when \( \text{fmt} = W \text{ or } L \)

<table>
<thead>
<tr>
<th>bits 5..3</th>
<th>000</th>
<th>001</th>
<th>010</th>
<th>011</th>
<th>100</th>
<th>101</th>
<th>110</th>
<th>111</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>1</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>2</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>3</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>4</td>
<td>CVTS</td>
<td>CVT.D</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>5</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>6</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>7</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>opcode = ( \text{COP1} )</th>
<th>( \text{fmt} = W, L )</th>
<th>function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0000</td>
<td>0000</td>
</tr>
<tr>
<td>0001</td>
<td>0001</td>
<td>0001</td>
</tr>
<tr>
<td>0100</td>
<td>0100</td>
<td>0100</td>
</tr>
<tr>
<td>0110</td>
<td>0110</td>
<td>0110</td>
</tr>
<tr>
<td>1000</td>
<td>1000</td>
<td>1000</td>
</tr>
<tr>
<td>1010</td>
<td>1010</td>
<td>1010</td>
</tr>
<tr>
<td>1100</td>
<td>1100</td>
<td>1100</td>
</tr>
<tr>
<td>1110</td>
<td>1110</td>
<td>1110</td>
</tr>
</tbody>
</table>
### Table B-26  FPU (CP1) Instruction Encoding - MIPS IV Architecture

Instructions encoded by the opcode field.

<table>
<thead>
<tr>
<th>OPCODE</th>
<th>INSTRUCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SPECIAL δ, β</td>
</tr>
<tr>
<td>1</td>
<td>COP1 δ</td>
</tr>
<tr>
<td>2</td>
<td>COP1X δ, λ</td>
</tr>
<tr>
<td>3</td>
<td>COP1 δ</td>
</tr>
<tr>
<td>4</td>
<td>COP1X δ, λ</td>
</tr>
<tr>
<td>5</td>
<td>COP1 δ</td>
</tr>
<tr>
<td>6</td>
<td>COP1X δ, λ</td>
</tr>
<tr>
<td>7</td>
<td>COP1 δ</td>
</tr>
</tbody>
</table>

Instructions encoded by the fmt field when opcode=COP1.

<table>
<thead>
<tr>
<th>FMT</th>
<th>INSTRUCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>MFC1</td>
</tr>
<tr>
<td>1</td>
<td>DMFC1</td>
</tr>
<tr>
<td>2</td>
<td>CFC1</td>
</tr>
<tr>
<td>3</td>
<td>MTC1</td>
</tr>
<tr>
<td>4</td>
<td>DMTC1</td>
</tr>
<tr>
<td>5</td>
<td>CTC1</td>
</tr>
<tr>
<td>6</td>
<td>LWC1</td>
</tr>
<tr>
<td>7</td>
<td>SWC1</td>
</tr>
<tr>
<td>8</td>
<td>LDC1</td>
</tr>
<tr>
<td>9</td>
<td>SDC1</td>
</tr>
</tbody>
</table>

Instructions encoded by the nd and tf fields when opcode=COP1 and fmt=BC.

<table>
<thead>
<tr>
<th>T</th>
<th>N</th>
<th>VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>BC1F</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>BC1T</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>BC1FL</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>BC1TL</td>
</tr>
</tbody>
</table>
Table B-26 (cont.)  FPU (CP1) Instruction Encoding - MIPS IV Architecture

Instructions encoded by the function field when opcode=COP1 and fmt = S, D, W, or L

<table>
<thead>
<tr>
<th>encoding when fmt = S</th>
<th>opcode = COP1</th>
<th>fmt = S</th>
<th>function</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>31 26 25 21 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bits 5..3</td>
<td>0 1 2 3 4 5 6 7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000</td>
<td>ADD SUB MUL DIV SQRT ABS MOV NEG</td>
<td></td>
<td></td>
</tr>
<tr>
<td>001</td>
<td>ROUND.L TRUNC.L CEIL.L FLOOR.L ROUND.W TRUNC.W CEIL.W FLOOR.W</td>
<td></td>
<td></td>
</tr>
<tr>
<td>010</td>
<td>MOVCF MOVZ MOVN RECIP RSQRT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>011</td>
<td>MOVCF MOVZ MOVN RECIP RSQRT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>101</td>
<td>MOVCF MOVZ MOVN RECIP RSQRT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>C.F α C.UN α C.EQ α C.UEQ α C.OLT α C.ULT α C.OLE α C.ULE α</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111</td>
<td>C.SF α C.NGLE α C_SEQ α C.NGL α C.LT α C.NGE α C.LE α C.NGT α</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>encoding when fmt = D</th>
<th>opcode = COP1</th>
<th>fmt = D</th>
<th>function</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>31 26 25 21 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bits 5..3</td>
<td>0 1 2 3 4 5 6 7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000</td>
<td>ADD SUB MUL DIV SQRT ABS MOV NEG</td>
<td></td>
<td></td>
</tr>
<tr>
<td>001</td>
<td>ROUND.L TRUNC.L CEIL.L FLOOR.L ROUND.W TRUNC.W CEIL.W FLOOR.W</td>
<td></td>
<td></td>
</tr>
<tr>
<td>010</td>
<td>MOVCF MOVZ MOVN RECIP RSQRT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>011</td>
<td>MOVCF MOVZ MOVN RECIP RSQRT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>101</td>
<td>MOVCF MOVZ MOVN RECIP RSQRT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>C.F α C.UN α C.EQ α C.UEQ α C.OLT α C.ULT α C.OLE α C.ULE α</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111</td>
<td>C.SF α C.NGLE α C_SEQ α C.NGL α C.LT α C.NGE α C.LE α C.NGT α</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table B-26 (cont.)  FPU (CP1) Instruction Encoding - MIPS IV Architecture

Instructions encoded by the function field when opcode=COP1X.

<table>
<thead>
<tr>
<th>bits 5..3</th>
<th>000</th>
<th>001</th>
<th>010</th>
<th>011</th>
<th>100</th>
<th>101</th>
<th>110</th>
<th>111</th>
</tr>
</thead>
<tbody>
<tr>
<td>function</td>
<td>LWXC1</td>
<td>LDXC1</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td></td>
<td>SWXC1</td>
<td>SDXC1</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>PFX</td>
</tr>
<tr>
<td></td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MADD.S</td>
<td>MADD.D</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td></td>
<td>SUB.S</td>
<td>MSUB.D</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td></td>
<td>NMADD.S</td>
<td>NMADD.D</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td></td>
<td>NMSUB.S</td>
<td>NMSUB.D</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td></td>
</tr>
</tbody>
</table>

Instructions encoded by the tf field when opcode=COP1, fmt = S or D, and function=MOVCF.

1 bit 16 0 1

These are the MOVF.fmt and MOVT.fmt instructions. They should not be confused with MOVF and MOVT.
Instruction class encoded by the function field when opcode=SPECIAL.

```
 31 26  5 0
<table>
<thead>
<tr>
<th>opcode</th>
<th>function</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL</td>
<td></td>
</tr>
</tbody>
</table>
```

<table>
<thead>
<tr>
<th>functi on bits 2..0</th>
<th>0 1 2 3 4 5 6 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>bits 5..3</td>
<td>000 001 010 011 100 101 110 111</td>
</tr>
<tr>
<td>0 000</td>
<td>MOVCI δ</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>7 111</td>
<td>x</td>
</tr>
</tbody>
</table>

Instructions encoded by the tf field when opcode = SPECIAL and function=MOVCI.

```
 31 26  16  5 0
<table>
<thead>
<tr>
<th>opcode</th>
<th>t</th>
<th>function</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL</td>
<td></td>
<td>MOVCI</td>
</tr>
</tbody>
</table>
```

<table>
<thead>
<tr>
<th>t bit 16</th>
<th>0 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVF</td>
<td>MOVT</td>
</tr>
</tbody>
</table>

These are the MOVF and MOVT instructions. They should not be confused with MOVF:fmt and MOVT:fmt.
Table B-27  Architecture Level In Which FPU Instructions are Defined or Extended.

The architecture level in which each MIPS IV encoding was defined is indicated by a subscript 1, 2, 3, or 4 (for architecture level I, II, III, or IV). If an instruction or instruction class was later extended, the extending level is indicated after the defining level.

Instructions encoded by the opcode field.

<table>
<thead>
<tr>
<th>opcode bits 31-26</th>
<th>Architecture level is shown by a subscript 1, 2, III, or 4.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>bits 31-29</td>
</tr>
<tr>
<td>000</td>
<td>SPECIAL_4</td>
</tr>
<tr>
<td>001</td>
<td>COP1_1,2,3,4</td>
</tr>
<tr>
<td>010</td>
<td>COP1X_4</td>
</tr>
<tr>
<td>011</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td></td>
</tr>
<tr>
<td>101</td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>LWC1_1</td>
</tr>
<tr>
<td>111</td>
<td>SWC1_1</td>
</tr>
</tbody>
</table>

Instructions encoded by the fmt field when opcode=COP1.

<table>
<thead>
<tr>
<th>opcode = COP1</th>
<th>fmt</th>
</tr>
</thead>
<tbody>
<tr>
<td>bits 31-26</td>
<td>Architecture level is shown by a subscript 1, 2, 3, or 4.</td>
</tr>
<tr>
<td>000</td>
<td>MFC1_1</td>
</tr>
<tr>
<td>001</td>
<td>DMFC1_3</td>
</tr>
<tr>
<td>010</td>
<td>CFC1_1</td>
</tr>
<tr>
<td>011</td>
<td>MTC1_1</td>
</tr>
<tr>
<td>100</td>
<td>DMTC1_3</td>
</tr>
<tr>
<td>101</td>
<td>CTC1_1</td>
</tr>
<tr>
<td>110</td>
<td>LWC1_1</td>
</tr>
<tr>
<td>111</td>
<td>SWC1_1</td>
</tr>
</tbody>
</table>

Instructions encoded by the nd and tf fields when opcode=COP1 and fmt=BC.

<table>
<thead>
<tr>
<th>opcode = COP1</th>
<th>fmt = BC</th>
<th>n</th>
<th>d</th>
<th>t</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>bits 31-17</td>
<td>Architecture level is shown by a subscript 1, 2, 3, or 4.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000</td>
<td>BCIF1_4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001</td>
<td>BCIT1_4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010</td>
<td>BCIFL2_4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011</td>
<td>BCITL2_4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table B-27 (cont.) Architecture Level (I-IV) In Which FPU Instructions are Defined or Extended

Instructions encoded by the function field when opcode=COPI and fmt = S, D, W, or L

<table>
<thead>
<tr>
<th>function</th>
<th>bits 2..0</th>
<th>Architecture level is shown by a subscript 1, 2, 3, or 4.</th>
</tr>
</thead>
<tbody>
<tr>
<td>fmt = S</td>
<td>0 000</td>
<td>000 001 010 011 100 101 110 111</td>
</tr>
<tr>
<td></td>
<td>1 001</td>
<td>ROUND.L 3 TRUNC.L 3 CEIL.L 3 FLOOR.L 3 ROUND.W 2 TRUNC.W 2 CEIL.W 2 FLOOR.W 2</td>
</tr>
<tr>
<td></td>
<td>2 010</td>
<td>* 1 MOVCF 4 MOVZ 4 MOVN 4 * 1 RECIP 4 RSQRT 4 * 1</td>
</tr>
<tr>
<td></td>
<td>3 011</td>
<td>* 1 * 1 * 1 * 1 * 1 * 1 * 1 * 1 * 1 * 1</td>
</tr>
<tr>
<td></td>
<td>4 100</td>
<td>* 1 CVT.D 1,3 * 1 * 1 CVT.W 1 CVT.L 3 * 1 * 1</td>
</tr>
<tr>
<td></td>
<td>5 101</td>
<td>* 1 * 1 * 1 * 1 * 1 * 1 * 1 * 1 * 1 * 1</td>
</tr>
<tr>
<td></td>
<td>6 110</td>
<td>C.F 1,4 C.UN 1,4 C.EQ 1,4 C.UEQ 1,4 C.OLT 1,4 C.ULT 1,4 C.OLE 1,4 C.ULE 1,4</td>
</tr>
<tr>
<td></td>
<td>7 111</td>
<td>C.SF 1,4 C.NGLE 1,4 C.SEQ 1,4 C.NGL 1,4 C.LT 1,4 C.NGE 1,4 C.LE 1,4 C.NGT 1,4</td>
</tr>
</tbody>
</table>

encoding when fmt = D

<table>
<thead>
<tr>
<th>function</th>
<th>bits 2..0</th>
<th>Architecture level is shown by a subscript 1, 2, 3, or 4.</th>
</tr>
</thead>
<tbody>
<tr>
<td>fmt = D</td>
<td>0 000</td>
<td>000 001 010 011 100 101 110 111</td>
</tr>
<tr>
<td></td>
<td>1 001</td>
<td>ROUND.L 3 TRUNC.L 3 CEIL.L 3 FLOOR.L 3 ROUND.W 2 TRUNC.W 2 CEIL.W 2 FLOOR.W 2</td>
</tr>
<tr>
<td></td>
<td>2 010</td>
<td>* 1 MOVCF 4 MOVZ 4 MOVN 4 * 1 RECIP 4 RSQRT 4 * 1</td>
</tr>
<tr>
<td></td>
<td>3 011</td>
<td>* 1 * 1 * 1 * 1 * 1 * 1 * 1 * 1 * 1 * 1</td>
</tr>
<tr>
<td></td>
<td>4 100</td>
<td>CVT.S 1,3 * 1 * 1 * 1 CVT.W 1 CVT.L 3 * 1 * 1</td>
</tr>
<tr>
<td></td>
<td>5 101</td>
<td>* 1 * 1 * 1 * 1 * 1 * 1 * 1 * 1 * 1 * 1</td>
</tr>
<tr>
<td></td>
<td>6 110</td>
<td>C.F 1,4 C.UN 1,4 C.EQ 1,4 C.UEQ 1,4 C.OLT 1,4 C.ULT 1,4 C.OLE 1,4 C.ULE 1,4</td>
</tr>
<tr>
<td></td>
<td>7 111</td>
<td>C.SF 1,4 C.NGLE 1,4 C.SEQ 1,4 C.NGL 1,4 C.LT 1,4 C.NGE 1,4 C.LE 1,4 C.NGT 1,4</td>
</tr>
</tbody>
</table>
encoding when \( \text{fmt} = W \) or \( L \)

<table>
<thead>
<tr>
<th>bits</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Architecture level is shown by a subscript 1, 2, 3, or 4.

<table>
<thead>
<tr>
<th>function</th>
<th>bits 2..0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>000 001 010 011 100 101 110 111</td>
</tr>
<tr>
<td></td>
<td>*1 *1 *1 *1 *1 *1 *1 *1</td>
</tr>
<tr>
<td></td>
<td>*1 *1 *1 *1 *1 *1 *1 *1</td>
</tr>
<tr>
<td></td>
<td>*1 *1 *1 *1 *1 *1 *1 *1</td>
</tr>
<tr>
<td></td>
<td>*1 *1 *1 *1 *1 *1 *1 *1</td>
</tr>
<tr>
<td></td>
<td>*1 *1 *1 *1 *1 *1 *1 *1</td>
</tr>
<tr>
<td></td>
<td>*1 *1 *1 *1 *1 *1 *1 *1</td>
</tr>
<tr>
<td></td>
<td>*1 *1 *1 *1 *1 *1 *1 *1</td>
</tr>
<tr>
<td></td>
<td>*1 *1 *1 *1 *1 *1 *1 *1</td>
</tr>
</tbody>
</table>

Instructions encoded by the \( \text{function} \) field when \( \text{opcode} = \text{COP1X} \).

Table B-27 (cont.)  Architecture Level (I-IV) In Which FPU Instructions are Defined or Extended Instructions encoded by the \( \text{function} \) field when \( \text{opcode} = \text{COP1X} \).

<table>
<thead>
<tr>
<th>function</th>
<th>bits 2..0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>000 001 010 011 100 101 110 111</td>
</tr>
<tr>
<td></td>
<td>LWXC1 4  LDXC1 4  *4 *4 *4 *4 *4 *4</td>
</tr>
<tr>
<td></td>
<td>SWXC1 4  SDXC1 4  *4 *4 *4 *4 *4 *4</td>
</tr>
<tr>
<td></td>
<td>*4 *4 *4 *4 *4 *4 *4 *4</td>
</tr>
<tr>
<td></td>
<td>*4 *4 *4 *4 *4 *4 *4 *4</td>
</tr>
<tr>
<td></td>
<td>*4 *4 *4 *4 *4 *4 *4 *4</td>
</tr>
<tr>
<td></td>
<td>*4 *4 *4 *4 *4 *4 *4 *4</td>
</tr>
<tr>
<td></td>
<td>*4 *4 *4 *4 *4 *4 *4 *4</td>
</tr>
<tr>
<td></td>
<td>*4 *4 *4 *4 *4 *4 *4 *4</td>
</tr>
</tbody>
</table>

Instructions encoded by the \( t\) field when \( \text{opcode} = \text{COP1} \), \( \text{fmt} = S \) or \( D \), and \( \text{function} = \text{MOVCF} \).

<table>
<thead>
<tr>
<th>function</th>
<th>( t )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0 1</td>
</tr>
<tr>
<td></td>
<td>MOVF (fmt) 4 MOVVT (fmt) 4</td>
</tr>
</tbody>
</table>

These are the \( \text{MOVF.fmt} \) and \( \text{MOVVT.fmt} \) instructions. They should not be confused with \( \text{MOVF} \) and \( \text{MOVT} \).
Instruction class encoded by the function field when opcode=\textit{SPECIAL}.

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode = SPECIAL</td>
<td>function</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Architecture level is shown by a subscript 1, 2, 3, or 4.

<table>
<thead>
<tr>
<th>bits 5..3</th>
<th>000</th>
<th>001</th>
<th>010</th>
<th>011</th>
<th>100</th>
<th>101</th>
<th>110</th>
<th>111</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>MOVCI 4</td>
</tr>
<tr>
<td>7 111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Instructions encoded by the tf field when opcode = \textit{SPECIAL} and function=\textit{MOVCI}.

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode = SPECIAL</td>
<td>tf</td>
<td>function = MOVCI</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>t bit 16</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVF 4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MOVT 4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

These are the MOVF and MOVT instructions. They should not be confused with MOVF.fmt and MOVT.fmt.
Table B-28 FPU Instruction Encoding Changes - MIPS II Architecture Revision.

An instruction encoding is shown if the instruction is added or extended in this architecture revision. An instruction class, like COP1, is shown if the instruction class is added in this architecture revision.

Instructions encoded by the opcode field.

| Opcode Bits | 31 26 25 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|
| Opcode      | 000 001 010 011 100 101 110 111 | 000 001 010 011 100 101 110 111 | 000 001 010 011 100 101 110 111 |
|             | 0 000           | 1 001           | 2 010           | 3 011           | 4 100           | 5 101           | 6 110           | 7 111           |

Instructions encoded by the fmt field when opcode=COP1.

| Opcode | Fmt Bits | 31 26 25 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|--------|----------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|
|        | fmt=BC   | 000 001 010 011 100 101 110 111 | 000 001 010 011 100 101 110 111 | 000 001 010 011 100 101 110 111 |
|        |          | 0 00 1 01 2 10 3 11 | 0 00 1 01 2 10 3 11 | 0 00 1 01 2 10 3 11 |

Instructions encoded by the nd and tf fields when opcode=COP1 and fmt=BC.

| Opcode | Fmt | Nd | Tf | 31 26 25 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|--------|-----|----|----|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|
|        | COP1| BC |   | 000 001 010 011 100 101 110 111 | 000 001 010 011 100 101 110 111 | 000 001 010 011 100 101 110 111 |
|        |     |    |   | 0 0 1 0 1 0 1 | 0 0 1 0 1 0 1 | 0 0 1 0 1 0 1 |
|        | COP1| BC |   | 0 001 | BC1FL | BC1TL | BC1FL | BC1TL |
Table B-28 (cont.)  FPU Instruction Encoding Changes - MIPS II Revision.

Instructions encoded by the function field when opcode=COP1 and fmt = S, D, or W

<table>
<thead>
<tr>
<th>bits 5..3</th>
<th>bits 2..0</th>
<th>opcode = COP1</th>
<th>fmt</th>
<th>function</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>000</td>
<td>SQRT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>001</td>
<td>001</td>
<td>ROUND.W</td>
<td>TRUNC.W</td>
<td>CEIL.W</td>
</tr>
<tr>
<td>010</td>
<td>010</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011</td>
<td>011</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>100</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>101</td>
<td>101</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>110</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>111</td>
<td>111</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

encoding when fmt = D

<table>
<thead>
<tr>
<th>bits 5..3</th>
<th>bits 2..0</th>
<th>opcode = COP1</th>
<th>fmt = D</th>
<th>function</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>000</td>
<td>SQRT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>001</td>
<td>001</td>
<td>ROUND.W</td>
<td>TRUNC.W</td>
<td>CEIL.W</td>
</tr>
<tr>
<td>010</td>
<td>010</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011</td>
<td>011</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>100</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>101</td>
<td>101</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>110</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>111</td>
<td>111</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
encoding when $fmt = W$

<table>
<thead>
<tr>
<th>bits 5..3</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>101</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table B-29  FPU Instruction Encoding Changes - MIPS III Revision.

An instruction encoding is shown if the instruction is added or extended in this architecture revision. An instruction class, like COP1, is shown if the instruction class is added in this architecture revision.

Instructions encoded by the opcode field.

<table>
<thead>
<tr>
<th>opcode</th>
<th>31 26 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>bits 28..26</td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
<tr>
<td>31..29 000</td>
<td>001</td>
</tr>
<tr>
<td>0</td>
<td>000</td>
</tr>
</tbody>
</table>

Instructions encoded by the fmt field when opcode=COP1.

<table>
<thead>
<tr>
<th>opcode</th>
<th>fmt</th>
<th>31 26 25 21 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>bits 23..21</td>
<td>0 1 2 3 4 5 6 7</td>
<td></td>
</tr>
<tr>
<td>25..24 000</td>
<td>001</td>
<td>010</td>
</tr>
<tr>
<td>0</td>
<td>00</td>
<td>DMFC1</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td></td>
</tr>
</tbody>
</table>

Instructions encoded by the nd and tf fields when opcode=COP1 and fmt=BC.

<table>
<thead>
<tr>
<th>opcode</th>
<th>fmt</th>
<th>n</th>
<th>t</th>
<th>f</th>
<th>31 26 25 21 17 16 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>bits 15..14</td>
<td>0 1 2 3 4 5 6 7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15..14 00</td>
<td>01</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>BC1FL</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>BC1TL</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

B-120  MIPS IV Instruction Set. Rev 3.2  FPU Instruction Set
### Table B-29 (cont.) FPU Instruction Encoding Changes - MIPS III Revision.

Instructions encoded by the function field when opcode=COP1 and fmt = S, D, or L.

<table>
<thead>
<tr>
<th>Bits 5..3</th>
<th>Encoding when fmt = S</th>
<th>Encoding when fmt = D</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 000</td>
<td>000</td>
<td>000</td>
</tr>
<tr>
<td>1 001</td>
<td>001 TRUNC.L</td>
<td>001 TRUNC.L</td>
</tr>
<tr>
<td>2 010</td>
<td>010 ROUND.L</td>
<td>010 ROUND.L</td>
</tr>
<tr>
<td>3 011</td>
<td>CEIL.L</td>
<td>CEIL.L</td>
</tr>
<tr>
<td>4 100</td>
<td>FLOOR.L</td>
<td>FLOOR.L</td>
</tr>
<tr>
<td>5 101</td>
<td>CVT.L</td>
<td>CVT.L</td>
</tr>
<tr>
<td>6 110</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7 111</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### B-122 MIPS IV Instruction Set. Rev 3.2 FPU Instruction Set

**Encoding when**

fmt = \(L\)

---

<table>
<thead>
<tr>
<th>functi (n)</th>
<th>bits 2..0</th>
</tr>
</thead>
<tbody>
<tr>
<td>bits 5..3</td>
<td>000</td>
</tr>
<tr>
<td>0 000</td>
<td>*</td>
</tr>
<tr>
<td>1 001</td>
<td>*</td>
</tr>
<tr>
<td>2 010</td>
<td>*</td>
</tr>
<tr>
<td>3 011</td>
<td>*</td>
</tr>
<tr>
<td>4 100 CVTS</td>
<td>CVT.D</td>
</tr>
<tr>
<td>5 101</td>
<td>*</td>
</tr>
<tr>
<td>6 110</td>
<td>*</td>
</tr>
<tr>
<td>7 111</td>
<td>*</td>
</tr>
</tbody>
</table>
Table B-30  FPU Instruction Encoding Changes - MIPS IV Revision.

An instruction encoding is shown if the instruction is added or extended in this architecture revision. An instruction class, like COP1X, is shown if the instruction class is added in this architecture revision.

Instructions encoded by the opcode field.

<table>
<thead>
<tr>
<th>bits</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>31..29</td>
<td>000</td>
<td>001</td>
<td>010</td>
<td>011</td>
<td>100</td>
<td>101</td>
<td>110</td>
<td>111</td>
</tr>
</tbody>
</table>

Instructions encoded by the fmt field when opcode=COP1.

Instructions encoded by the nd and tf fields when opcode=COP1 and fmt=BC.
Table B-30 (cont.)  FPU Instruction Encoding Changes - MIPS IV Revision.

Instructions encoded by the function field when opcode=COP1 and fmt = S, D, W, or L.

### Encoding when fmt = S

<table>
<thead>
<tr>
<th>funct</th>
<th>bits</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>fmt</td>
<td></td>
<td>000</td>
<td>001</td>
<td>010</td>
<td>011</td>
<td>100</td>
<td>101</td>
<td>110</td>
<td>111</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MOVCF δ</td>
<td>MOVZ</td>
<td>MOVN</td>
<td>RECIP</td>
<td>RSQRT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>MOVCF δ</td>
<td>MOVZ</td>
<td>MOVN</td>
<td>RECIP</td>
<td>RSQRT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>C.F</td>
<td>C.UN</td>
<td>C.EQ</td>
<td>C.UEQ</td>
<td>C.LT</td>
<td>C.ULT</td>
<td>C.OLE</td>
<td>C.ULE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>C.F</td>
<td>C.UN</td>
<td>C.EQ</td>
<td>C.UEQ</td>
<td>C.LT</td>
<td>C.ULT</td>
<td>C.OLE</td>
<td>C.ULE</td>
</tr>
</tbody>
</table>

### Encoding when fmt = D

<table>
<thead>
<tr>
<th>funct</th>
<th>bits</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>fmt</td>
<td></td>
<td>000</td>
<td>001</td>
<td>010</td>
<td>011</td>
<td>100</td>
<td>101</td>
<td>110</td>
<td>111</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MOVCF δ</td>
<td>MOVZ</td>
<td>MOVN</td>
<td>RECIP</td>
<td>RSQRT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>MOVCF δ</td>
<td>MOVZ</td>
<td>MOVN</td>
<td>RECIP</td>
<td>RSQRT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>C.F</td>
<td>C.UN</td>
<td>C.EQ</td>
<td>C.UEQ</td>
<td>C.LT</td>
<td>C.ULT</td>
<td>C.OLE</td>
<td>C.ULE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>C.F</td>
<td>C.UN</td>
<td>C.EQ</td>
<td>C.UEQ</td>
<td>C.LT</td>
<td>C.ULT</td>
<td>C.OLE</td>
<td>C.ULE</td>
</tr>
</tbody>
</table>

### Encoding when fmt = W or L

<table>
<thead>
<tr>
<th>funct</th>
<th>bits</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>fmt</td>
<td></td>
<td>000</td>
<td>001</td>
<td>010</td>
<td>011</td>
<td>100</td>
<td>101</td>
<td>110</td>
<td>111</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MOVCF δ</td>
<td>MOVZ</td>
<td>MOVN</td>
<td>RECIP</td>
<td>RSQRT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>MOVCF δ</td>
<td>MOVZ</td>
<td>MOVN</td>
<td>RECIP</td>
<td>RSQRT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>C.F</td>
<td>C.UN</td>
<td>C.EQ</td>
<td>C.UEQ</td>
<td>C.LT</td>
<td>C.ULT</td>
<td>C.OLE</td>
<td>C.ULE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>C.F</td>
<td>C.UN</td>
<td>C.EQ</td>
<td>C.UEQ</td>
<td>C.LT</td>
<td>C.ULT</td>
<td>C.OLE</td>
<td>C.ULE</td>
</tr>
</tbody>
</table>
Table B-30 (cont.)  FPU Instruction Encoding Changes - MIPS IV Revision.
Instructions encoded by the function field when opcode=COPIX.

<table>
<thead>
<tr>
<th>functi</th>
<th>bits 2..0</th>
<th>bits 5..3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000</td>
<td>LWXCI</td>
</tr>
<tr>
<td>0</td>
<td>001</td>
<td>LDXCI</td>
</tr>
<tr>
<td>1</td>
<td>010</td>
<td>SWXCI</td>
</tr>
<tr>
<td>1</td>
<td>011</td>
<td>SDXCI</td>
</tr>
<tr>
<td>2</td>
<td>100</td>
<td>MADD.S</td>
</tr>
<tr>
<td>2</td>
<td>101</td>
<td>MADD.D</td>
</tr>
<tr>
<td>3</td>
<td>110</td>
<td>MSUBS</td>
</tr>
<tr>
<td>3</td>
<td>111</td>
<td>MSUB.D</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>NMADD.S</td>
</tr>
<tr>
<td>4</td>
<td>101</td>
<td>NMADD.D</td>
</tr>
<tr>
<td>5</td>
<td>110</td>
<td>NMSUBS</td>
</tr>
<tr>
<td>5</td>
<td>111</td>
<td>NMSUB.D</td>
</tr>
</tbody>
</table>

Instructions encoded by the tf field when opcode=COPI, fmt = S or D, and function=MOVCF.

<table>
<thead>
<tr>
<th>opcode = COPI</th>
<th>fmt = S, D</th>
<th>tf</th>
<th>function = MOVCF</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>MOVF (fmt)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>MOVT (fmt)</td>
</tr>
</tbody>
</table>

These are the MOVF.fmt and MOVT.fmt instructions. They should not be confused with MOVF and MOVT.

Instruction class encoded by the function field when opcode=SPECIAL.

<table>
<thead>
<tr>
<th>functi</th>
<th>bits 2..0</th>
<th>bits 5..3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000</td>
<td>MOVCI</td>
</tr>
<tr>
<td>7</td>
<td>111</td>
<td></td>
</tr>
</tbody>
</table>

FPU Instruction Set  
MIPS IV Instruction Set. Rev 3.2  
B-125
Instructions encoded by the \( tf \) field when opcode = *SPECIAL* and function=MOVCI.

<table>
<thead>
<tr>
<th>bit 16</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>t</td>
<td>f</td>
<td></td>
</tr>
</tbody>
</table>

These are the MOVF and MOVT instructions. They should not be confused with MOVE.fmt and MOVT.fmt.
Key to all FPU (CP1) instruction encoding tables:

* This opcode is reserved for future use. An attempt to execute it causes either a Reserved Instruction exception or a Floating Point Unimplemented Operation Exception. The choice of exception is implementation specific.

α The table shows 16 compare instructions with values named C.condition where “condition” is a comparison condition such as “EQ”. These encoding values are all documented in the instruction description titled “C.cond.fmt”.

β The SPECIAL instruction class was defined in MIPS I for CPU instructions. An FPU instruction was first added to the instruction class in MIPS IV.

δ (also *italic* opcode name) This opcode indicates an instruction class. The instruction word must be further decoded by examining additional tables that show values for another instruction field.

λ The COP1X opcode in MIPS IV was the COP3 opcode in MIPS I and II and a reserved instruction in MIPS III.

χ These opcodes are not FPU operations. For further information on them, look in the CPU Instruction Encoding information section A 8.

(fmt) This opcode is a conditional move of formatted FP registers - either MOVF.D, MOVFS, MOVT.D, or MOVT.S. It should not be confused with the similarly-named MOVF or MOVT instruction that moves CPU registers.